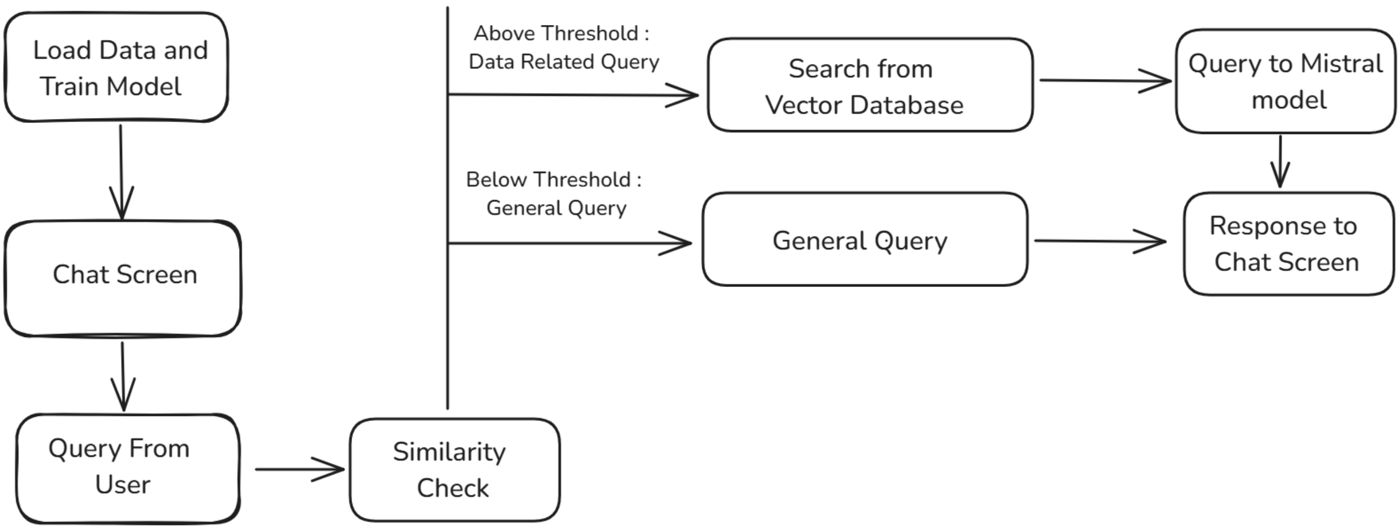
TECHNICAL DOCUMENTATION

System Architecture explanation



1. Load Data and Train Model
   - The system begins by loading scraped restaurant/menu data and possibly embedding it using a model.
   - A vector index (Pinecone) is prepared for retrieval tasks.
2. Chat Screen
   - The frontend interface where users input their queries.
3. Query from User
   - User submits a question—this could be about menus, pricing, ratings, or general queries.
4. Similarity Check
   - The query is embedded and compared to vectors in the database using cosine similarity check.
   - A similarity threshold is used to determine the nature of the query:
     - Above threshold → Considered data-related, requires retrieval.
     - Below threshold → Treated as a general query.

For Data-Related Queries (Above Threshold)

5. Search from Vector Database
    - Queries are matched against vector representations of restaurant/menu data to fetch relevant context.
6. Query to Mistral Model
    - The context + user query is sent to the Mistral language model (or equivalent) to generate a smart response.
7. Response to Chat Screen
    - The final answer is displayed to the user.

Implementation details and design decisions

Implementation Details

1. Framework: The chatbot is built using Flask, which serves both as the API layer and optionally the chat frontend.
2. Scraping Tool:
    a. Selenium is used to scrape Zomato for restaurant and menu information.
    b. The data is stored as CSV files, one for restaurant metadata and another for menu items.
3. Data Preprocessing:
    a. The scraped data is cleaned, structured, and then converted into vector embeddings using *multi-qa-MiniLM-L6-cos-v1*.
    b. Each menu item and restaurant is embedded individually to ensure fine-grained retrieval.
4. Vector Database:
    a. Pinecone is used to store and search through high-dimensional vector representations of the menu and restaurant data.
5. Query Handling:
    a. Each user query is embedded and compared against stored vectors.

    b. A similarity threshold is used to determine if the query is related to the restaurant data or is a general query.

6. Language Model:
    a. Mistral-7B is used to generate context-aware responses when a relevant context is retrieved.
    b. If the query doesn't require data context (falls below threshold), the model responds directly.
7. API Tokens:
    a. Hugging Face API and Pinecone API keys are loaded into rag_engine.py.
8. Health Check:
    a. A simple health check endpoint (/health-check) is available for verifying setup and deployment.

Design Decisions

1. Threshold-based Query Routing
   → This avoids unnecessary Pinecone hits for queries that don't relate to restaurant data, optimizing cost and latency.
2. Vector Embedding Granularity
   → Menu items and restaurant info are embedded separately so the bot can return detailed responses about dishes, pricing, and dietary tags.
3. Use of Open Models
   → Using Hugging Face and Mistral ensures the system is cost-efficient and open-source compatible along with specific amount of accuracy.
4. Lightweight UI with Flask
   → Keeps the frontend minimal while leaving room to scale into a richer UI with just HTML serving as frontend along with flask as backend.

Challenges faced and solutions implemented

1. Data Collection

Problem:
A lot of restaurants don't have their own websites.

What We Did:
We had to rely on the Zomato site to extract data like restaurant names, locations, contact details, menu items, and other useful info.

2. Model Selection

Problem:
There were many free models available on Hugging Face, so choosing the right one was tricky.

What We Did:
We tested 4–5 different models to compare their outputs and picked the one that worked best for our use case.

3. Setting the Right Similarity Threshold

Problem:
If the similarity score was too low, data that belonged together didn't get grouped. If it was too high, unrelated data got matched.

What We Did:
We fine-tuned the similarity threshold to make sure the right data was matched properly — neither too strict nor too loose.

Future Improvement opportunities

1. Focus more on the dining experience by adding extra context like:
   - Real-time location data (how far the restaurant is)
   - Traffic conditions (so users can estimate travel time)
   - Weather info (because people might not want to go out when it's raining)
2. Handle Multi-Cuisine Restaurants Better
   Some places serve multiple cuisines, which can confuse the tagging.
   - Use NLP techniques like Named Entity Recognition (NER) to classify food types better
   - Build a small cuisine ontology or food category tree for consistency
3. Include Price & Offers in Analysis
   People often choose based on price or discounts.
   - Extract and highlight average pricing per person
   - Use OCR or regex parsing to detect offers from images or menus

4. Voice Search Support
   A lot of people use voice for convenience.
   - Add speech-to-text functionality for queries
   - Use intent classification to understand what the user is
5. Make it Mobile-Friendly
   Since most users are on phones:
   - Use responsive UI design
   - Add swipe-based filters for easier interaction