System Architecture

1. Introduction

The system is designed as a **Retrieval-Augmented Generation (RAG)** pipeline for restaurant-related queries. It combines web-scraped real-world restaurant data with powerful natural language model to respond to user questions such as:

1. *Which restaurants serve Chinese food in Kanpur?*
2. *Show me Jain or vegan dishes from ABC restaurant*
3. *What's the price range of Paneer dishes?*

To answer such natural language queries accurately, we architected a pipeline integrating:

1. Semantic document retrieval (FAISS + BGE embeddings)
2. Local Language Model (LaMini-Flan-T5)
3. Prompt Engineering and Token-Safe Truncation
4. Manual context overrides for structured filters
5. Streamlit UI with persistent chat memory

Pre-processing Pipeline: Knowledge Collection, Chunking & Embedding

This phase is about preparing the knowledge base (restaurant data) for efficient retrieval using embeddings.

A. User Documents

- **Source**: JSON data scraped from Swiggy website (menus, cuisines, price range, restaurant names, locations, etc.)
- **Format**: Structured JSON format with fields like `restaurant_name`, `cuisines`, `menu_items`, `location`, etc.

B. Embedding Model: `BAAI/bge-base-en-v1.5`

- This SentenceTransformer-based model was used to **generate dense vector representations** (embeddings) of each chunk of the knowledge base.
- **Why this model?** Optimized for semantic search, it offers fast inference and works well on domain-specific texts like food items and restaurant descriptions.

C. Embeddings → FAISS Vector Store

- **FAISS** (Facebook AI Similarity Search) is used to **store the document embeddings** efficiently in a vector index.
- Enables **fast approximate nearest neighbor search** during query-time retrieval.

3. User Query Processing (Input)

- **User enters a question** in the Streamlit chatbot (e.g., *"Which restaurants serve Chinese food?"*)
- The query is sent through the pipeline for retrieval and generation.

4. Query Embedding & Retrieval

A. Embedding Model (again): `BAAI/bge-base-en-v1.5`

- The **user's query is also embedded** into a vector using the same model as above.
- Ensures that both the knowledge base and query lie in the same semantic vector space.

B. Vector Database: FAISS

- The query embedding is compared to stored document embeddings using **Maximum Marginal Relevance (MMR)** to retrieve **top-k relevant chunks**.
- This helps avoid redundancy and improves the **diversity and quality** of the retrieved context.

For deterministic queries (e.g., listing all restaurants or retrieving menus), the system bypasses retrieval and directly pulls from the structured JSON.

This ensures perfect accuracy for queries with clearly defined expected structure.

5. RAG Augmentation Step

Retrieved relevant contexts are **concatenated with the original query** to form a final prompt in the format:

```
Context:
- ABC restaurant serves Paneer Do Pyaza...
- XYZ restaurant serves Chinese and North Indian...
Question:
Which restaurants serve Chinese food?
```

This **query+context** becomes the **input to the Large Language Model**.

6. LLM for Response Generation

LLM Model Used: `MBZUAI/LaMini-Flan-T5-783M`

- A fine-tuned T5 model trained for instruction following.
- This model **generates fluent, natural language responses** based on the augmented prompt.
- **Why this model?**
    - Lightweight (fits local inference)
    - Good accuracy on generation tasks
    - Supports instruction-following behavior, which works well for RAG-style prompting.

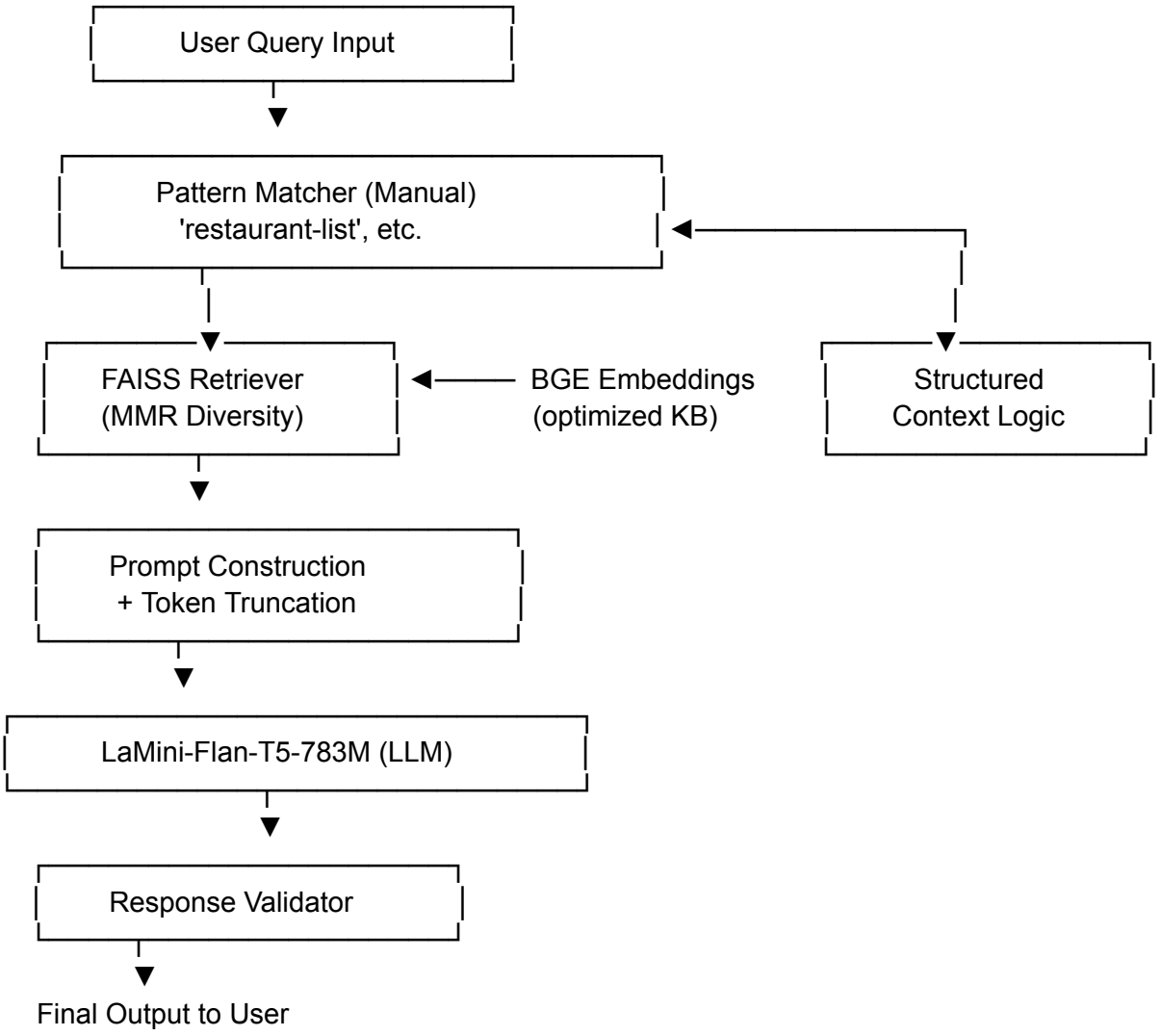
7. Final Output: Response to User

- The generated response is shown to the user in the Streamlit interface.
- UI features include:
    - Markdown card-style messages for user and bot
    - Send icon inside the chat input
    -

8. Session & Memory

- The chatbot maintains **chat history** in `st.session_state.chat_history`
- Enables smooth **conversational flow** and persistent context visibility
- Messages are displayed in **reverse order**, mimicking typical chat UIs

Architecture Flow



Implementation Details & Design Decisions

This section provides a **detailed breakdown of how the system was implemented**, the libraries and tools used, and **justifications for each major design decision** to optimize both performance and developer experience.

1. Web Scraping Strategy

Strategy used

- Scraped restaurant data from multiple real sources (e.g., Swiggy) using `Selenium` + `BeautifulSoup`.
- Extracted:
  - Restaurant name, location, city, rating
  - Menu items (dish name, price, description, rating, tags)
  - Cuisine types and delivery time
- Stored each restaurant's data in individual CSVs.

Design Rationale

- Modular CSV format made cleaning and aggregation easier.
- Structure ensured data could be directly transformed into a JSON corpus for indexing.

2. Frontend Interface with Streamlit

Library Use:

- Streamlit — It is an ideal choice for rapid UI development
- Python-based and ideal for LLM prototyping

Key Features Implemented:

- **Chat interface** with two-column card layout (user & bot responses)
- **Custom Send Button**: U+2B06 up-arrow emoji in a circle for a clean, intuitive input submission
- **Markdown rendering**: Maintains formatting and structure in responses
- **Session memory**: Retains chat history using `st.session_state`, enabling conversational continuity
- **Input auto-clear**: After each submission, input is cleared for better UX

Design Decisions:

- **Why Streamlit?**

  Because its Lightweight, open-source, and perfect for internal tools and LLM interfaces

- **Why custom markdown rendering?** To support rich formatting like bullet points, tables, and emphasis for chatbot replies

3. Embedding and Vector Search Pipeline

Tools Used:

- **Model**: `BAAI/bge-base-en-v1.5` (via `sentence-transformers`)
- **Vector DB**: FAISS (local, fast ANN search)

Implementation Details:

- Knowledge base or restaurants documents are stored in structured format as JSON file. This JSON file is preprocessed to flatten nested structures (e.g., menus, cuisines, pricing)
- Text chunks were generated per restaurant using custom chunking (fields like description, menu items, etc.)
- Each chunk was converted into dense vector representations using the embedding model

- FAISS index was created and persisted (`index.faiss`, `index.pkl`) for fast loading and querying

Design Decisions:

- **Why BAAI/bge-base-en-v1.5?**
    - Optimized for **semantic similarity** search
    - More accurate than older models like `all-MiniLM`
    - Excellent tradeoff between accuracy and speed
- **Why FAISS?**
    - Scalable for thousands of restaurants
    - Fast search with support for MMR and cosine similarity
    - Easy integration with LangChain

4. Retrieval Mechanism

Retrieval Method: Maximum Marginal Relevance (MMR)

- **MMR balances relevance and diversity**, ensuring that redundant contexts are avoided
- Retrieved top-k results (typically `k=4`) are concatenated into the RAG prompt

Design Decision:

- **Why MMR?**
    - Standard similarity search tends to return similar chunks
    - MMR ensures a **varied yet contextually rich input**, improving generation quality

5. Prompt Engineering

Key Features Implemented:

- Created a LangChain `PromptTemplate` with contextual formatting.
- Ensured safety using a custom `truncate_prompt()` function:
  `tokens = tokenizer.encode(prompt, max_length=1024, truncation=True)`

Design Rationale

- Ensures LLM won't fail on overlong prompts.
- Prompt instructs model to answer briefly or in list form, improving consistency.

Model is also told to say "not available" if answer cannot be found → improves trustworthiness.

6. LLM Generation Pipeline

Tools Used:

- Hugging Face Transformers
- Model: `MBZUAI/LaMini-Flan-T5-783M` (783M parameters)

Prompt Format:

```
Context:
<retrieved context>
```

```
Question:
<user query>
```

Design Decisions:

- **Why LaMini-Flan-T5?**
    - Light enough for local inference (no API keys needed)
    - Model is CPU-compatible and runs within <4GB RAM, ideal for low-resource setups.
    - Trained to follow instructions and generate structured outputs
    - Produces **high-quality completions** on knowledge-grounded inputs
- **Why Hugging Face pipeline?**
    - Easy integration with LangChain's `LLMChain`
    - Future upgrade path to other models (like Mixtral, Phi, or Llama2)

7. Manual Structured Filtering

Strategy used

- Added a custom fallback path via `manual_context.py`:
    - Predefined JSON search logic for keywords like:
        - `restaurant-list`, `menu-list`, `serves-dish-item`

Design Rationale

- Avoids hallucinations for objective queries.
- Guarantees full and accurate list-style answers.
- Great for scalable filtering in the future.

8. Error Handling & UX Enhancements

Strategy used

- Safeguarded against:
    - Empty input
    - Retrieval failures
    - Model token length limits

Show fallback messages instead of crashing:

```
Bot: Sorry, I couldn’t find a confident answer.
```

Design Rationale

- Ensures smooth UX during demo, testing, or low-resource operation.
- Designed for both graceful degradation and future extensibility.

Challenges Faced & Solutions Implemented

This section outlines the technical, architectural, and UX-related hurdles faced during the development of the Restaurant RAG Chatbot, along with the **specific solutions** or **workarounds** applied to address them.

1. Dynamic Data Collection from Swiggy

Challenge:

- **Swiggy's data is loaded dynamically via JavaScript**, making traditional scraping with `requests` or `BeautifulSoup` ineffective.
- The restaurant data had to be scraped across **multiple cities**, dynamically scrolling through pages and parsing embedded JS.

Solution:

- Used **Selenium WebDriver with headless Chrome** to emulate a real browser and scroll/load dynamically rendered content.
- Added `sleep()` + `wait_for_element()` logic to ensure DOM stability before extracting elements.
- Combined results from multiple city crawls into a **single, structured JSON knowledge base**.

2. Combining & Cleaning Restaurant Data

Challenge:

- Each city scrape created its own data file, resulting in **inconsistent structure**, and **duplicated fields or keys**.
- JSON was nested (e.g., menus inside menus), complicating vector embedding.

Solution:

- Flattened and normalized all restaurant records using a **custom data merge script**.
- Built a master knowledge base JSON with consistent fields: `name`, `cuisine`, `location`, `menu`, `price_level`, etc.

3. Embedding and Vector Store Creation

Challenge:

- Chunking menu text or nested JSON into semantically meaningful units was tricky.
- Some restaurants had sparse data, while others had large menus causing uneven chunking.

Solution:

- Created a **custom text chunking strategy**, grouping similar fields (e.g., menu + cuisine + location) into coherent text chunks.
- Used `BAAI/bge-base-en-v1.5` embeddings which handled varied length inputs well.
- Indexed them in **FAISS**, which provided efficient retrieval during runtime.

4. Handling Streamlit UI Issues

Challenges:

- Displaying user messages immediately upon submission (before response generation) was non-trivial due to Streamlit's **single-threaded nature** and rerender behavior.
- Errors like `st.experimental_rerun()` being deprecated or causing session resets.
- Avoiding duplicate `set_page_config()` errors when modularizing logic.

Solutions:

- Used `st.session_state` extensively to manage **chat history**, **input memory**, and **stateful logic**.
- Implemented a **form submit trigger** and Markdown rendering trick to display user queries immediately.
- Ensured `st.set_page_config()` was the **first line** in the script to avoid runtime errors.

5. LLM Quality and Resource Constraints

Challenge:

- Local generation with models like `LaMini-Flan-T5` requires balancing between **response quality** and **resource usage** (CPU-only inference).
- Hallucination and irrelevant outputs occasionally occurred when context retrieval was poor.

Solutions:

- Used **MMR (Max Marginal Relevance)** in FAISS to balance diversity and relevance of retrieved chunks.
- Improved prompt engineering by:
    - Clear separation of `Context` and `Question`
    - Using compact, clean JSON input as context
- Plans for further improvement with **streaming generation**, **query rewriting**, and **hierarchical summarization**.

6. Hallucinations for List-Type Questions

Challenge:

- LLMs often hallucinate or give partial completions when asked for enumerated or structured answers.
- Example**:**

  "Which restaurants serve Chinese food?"
  LLM responded:
  "The dish that is vegetarian is Veg Chinese Bowl." (instead of a list of restaurant names)

Solution

- We introduced a **manual filtering fallback** using keywords like:
  - `restaurant-list`, `serves-dish-item`, `menu-list`
- These trigger deterministic lookups from the knowledge base using structured Python logic.
- This ensured **precise, complete, and factual results** when needed.

7. Tried to Use Mistral-7B, Hit Memory Limits

Challenge:

- We initially tested `Mistral-7B-Instruct-v0.1` for generation using Hugging Face transformers.
- On a local CPU-based system, loading this 7B parameter model:
  - Caused paging file crashes
  - Threw errors like:

    ```
    OSError: The paging file is too small for this operation to complete.
    ```

Solution

- Switched to `MBZUAI/LaMini-Flan-T5-783M`:
    - Lightweight (<2GB)
    - Instruction-tuned
    - Fast response even on CPU
- This was a practical compromise balancing performance & accessibility.

8. Challenge: Token Overflow in Long Prompts

Challenge:

- Context chunks + question + prompt often exceeded model's max token length (512–1024).
- This caused:
    - Indexing errors
    - Truncation in the middle of dish names or instructions

Solution

- Introduced a `truncate_prompt()` function that safely encodes and slices token length before passing to the model.
- Ensures full prompt is always valid and decodable.

Future Improvements & Opportunities

1. Integrate Conversational Memory Chains

Current Limitation:

- Chat history is only stored and displayed but **not utilized** in response generation.
- Each query is treated in isolation.

Opportunity:

- Incorporate LangChain's `ConversationalRetrievalChain` to:
  - Use chat history as context
  - Enable follow-up questions like:

    "What about its desserts?" after asking about a restaurant

2. Switch to Quantized Local LLMs for Better Accuracy

Current Setup:

- Using `LaMini-Flan-T5` which is lightweight but has limitations in complex synthesis.

Opportunity:

- Integrate models like:
  - `Mistral-7B-Instruct` (via GGUF/GGML + llama.cpp)
  - `OpenHermes-2.5-Mistral` (optimized for chat)

- Load via `llama-cpp-python` or `ctransformers` for CPU inference
- Improves factuality and nuanced generation

3. Implement Query Rewriting + Reranking Pipelines

Motivation:

- Queries like *"Does XYZ have spicy food?"* may not directly map to vectorized text.

Opportunity:

- Use LLM to **rewrite user query** into a retrieval-optimized form:
  "Which dishes at XYZ restaurant mention spice level?"
- Then retrieve → rerank → generate

4. Advanced Response Synthesis Strategies

Motivation:

- Current system merges all retrieved content into one context block.

Opportunity:

- Use **async hierarchical summarization**:
  - Summarize each document chunk independently
  - Merge using tree-based or voting logic
- Reduces hallucinations and enhances readability

5. Add Schema-Aware Reasoning for Structured Comparisons

Use Case:

> *"Compare spice levels in Restaurant A vs B"*
> Currently, output may miss data or fail to format comparison properly.

Opportunity:

- Introduce **structured retrieval** using Pandas or DuckDB

Combine with prompt-injected reasoning:
`"Here is a table of spice levels. Compare them logically."`

6. Better UI/UX in Streamlit

Opportunity:

- Add:
  - Filters (e.g., price range, vegetarian)
  - Collapsible menus with dish previews
  - Downloadable menu cards (PDF)
  - Voice-based input and output

7. API Wrapper for Integration with Mobile/Web

Motivation:

- System is CLI/Streamlit bound.

Opportunity:

- Package core `get_rag_response()` into a **REST API** (FastAPI)
- Deploy to Hugging Face Spaces, Render, or localhost for mobile/web use