Conversational AI for Restaurant Insights

Dhiraj Pimparkar

1. Introduction

1.1 Overview of the Retrieval-Augmented Generation (RAG) Pipeline for AI-Driven Applications

This report details the retrieval, augmentation, and generation pipeline for a system designed to provide structured and unstructured restaurant-related information, aligning with the requirements of the Menudata Challenge.

This system aims to efficiently extract, structure, and process restaurant and menu-related data from both internal structured databases and external sources (such as Google searches and knowledge graphs). Given the vast amount of real-time information available in restaurant reviews, blogs, and culinary trends, the ability to retrieve, process, and integrate insights dynamically is critical for an effective AI assistant.

To ensure optimal retrieval and augmentation, I employ multiple retrieval strategies, including:

- FAISS-based dense retrieval for similarity search on vectorized menu data.
- Structured SQL-like queries for direct lookups in relational databases.
- Knowledge Graph-based retrieval using Neo4j for multi-hop reasoning and relationship-based searches.
- Google Search API and web scraping for supplementing real-time external knowledge.

These retrieval approaches enhance response accuracy by providing contextually rich and multi-modal data sources before passing the information to the augmentation pipeline. This pipeline extracts named entities, detects user intent, re-ranks results, and structures the extracted information before sending it to the response generation model. The final step ensures that responses are not only relevant and informative but also presented in a structured, user-friendly format.

1.2 Why Asynchronous Processing is Necessary

One of the biggest challenges in implementing such a multi-agent retrieval and augmentation system is latency. Since multiple retrieval agents are involved—each querying different sources with different response times—it is crucial to design the system to handle these operations asynchronously.

1.2.1 Bottlenecks in Synchronous Processing

Currently, the retrieval agents are executed in a sequential manner, which means:

1. The FAISS search runs first, retrieving relevant embeddings.

2. The structured database query follows, extracting restaurant-specific details.
3. The Knowledge Graph agent runs, fetching multi-hop relationships.
4. The Google Search agent queries the web, extracts content, and processes it.

Since each step waits for the previous one to finish, the overall response time is constrained by the slowest agent in the pipeline. This is especially problematic when external APIs (such as Google Search) introduce unpredictable delays.

1.2.2 My Proposal: Use LangGraph to Solve These Issues

Rather than handling the complexity of managing parallel tasks manually, LangGraph provides an event-driven architecture that can:

1. Run retrieval agents asynchronously in parallel without blocking execution.
2. Dynamically route outputs from retrieval agents to augmentation and generation steps as soon as they are available.
3. Implement fallback strategies for agents that take too long or fail to return results.
4. Facilitate multi-step reasoning (i.e., hierarchical intent processing) without writing complex state management code.

2. Retrieval

2.1 Overview of the Retrieval System

The retrieval pipeline is the backbone of this entire agentic AI system, ensuring highly relevant, structured, and real-time information is fetched before being passed on to augmentation and generation. This section details the multi-agent retrieval process, covering:

- Each retrieval agent and its role
- Implementation details with actual function names and files
- Technology choices (embedding models, rerankers, knowledge graph, FAISS, structured DB, Google Search)
- Current limitations and future improvements with asynchronous execution

The system uses four major retrieval agents:

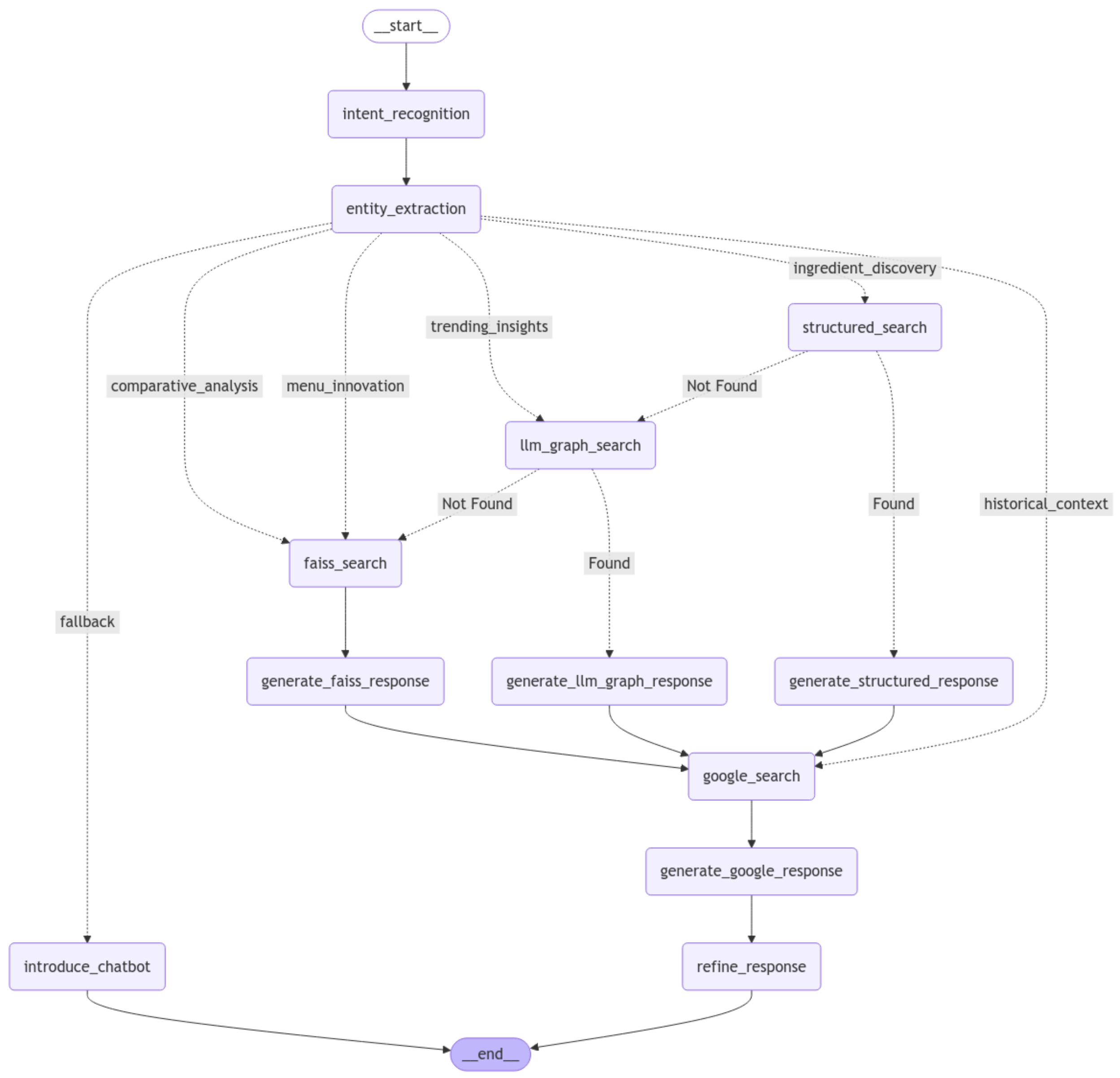
1. **FAISS Search Agent** → Fast similarity-based retrieval from embeddings.
2. **Structured DB Search Agent** → Querying structured restaurant and menu data.
3. **LLM Graph Search Agent** → Querying a knowledge graph (Neo4j) for relationship-based insights.
4. **Google Search Agent** → Fetching real-time data from external sources.

Each of these agents currently executes in series (in the future - parallel, using LangGraph) to reduce latency and maximize context relevance.

Before the retrieval starts, I start with understanding the user query first.

Key pre-retrieval steps in the implemetation.

1. Entity Extraction (entity_extraction.py)
   a. Extracts structured entities (location, menu_item, ingredient_name) using LLM prompting. (a smaller faster model can be used for this)
   b. These entities shape the query execution and response generation.
2. Intent Recognition (intent_recognition.py)
   a. Determines how the response should be structured based on user intent using LLM prompting. (a smaller faster model can be used for this)
   b. Example: If intent is comparative analysis, the system returns tabular output instead of text.



2.2 FAISS Search Agent: Embedding-Based Retrieval

- File: faiss_search.py
- Function: search_faiss(state: State)

FAISS (Facebook AI Similarity Search) is used for fast nearest-neighbor search on embeddings. The agent helps find semantically similar menu items, restaurant names, and descriptions.

Why FAISS?

FAISS was chosen over ChromaDB primarily due to indexing speed concerns during experimentation. While testing ChromaDB, the embedding insertion process took significantly longer, which raised concerns about potential slowdowns during real-time inference as well. Since retrieval speed is critical for handling restaurant menu lookups and ingredient-based queries, FAISS was selected as it provides faster indexing and retrieval with optimized Approximate Nearest Neighbor (ANN) search. Although ChromaDB offers built-in persistence and ease of use, the priority was on ensuring low-latency vector search, leading to the decision to use FAISS for efficient and scalable retrieval.

Implementation Details:

The embedding model used in the retrieval pipeline is sentence-transformers/all-mpnet-base-v2, selected for its optimal balance between accuracy and efficiency (~100M parameters). This model was shortlisted after reviewing Hugging Face's top embedding models and analyzing its common usage across blogs and implementations, ensuring reliability in semantic search tasks. To further refine retrieval, a reranking model (cross-encoder/ms-marco-MiniLM-L-6-v2) is applied, as FAISS initially retrieves the top 10 nearest neighbors, but these results need re-ranking to improve relevance. The search process begins by converting the user query into embeddings, followed by searching the FAISS index to retrieve the top 10 closest vectors. The corresponding metadata is then extracted, and the cross-encoder model re-ranks these results based on query relevance. Finally, the top 5 most relevant results are returned, ensuring that the system delivers the most contextually accurate matches.

Limitations & Future Enhancements:

Currently, the FAISS search operates synchronously, blocking execution before augmentation begins; implementing LangGraph can enable parallel execution of FAISS search alongside other retrieval agents, significantly reducing latency.

2.3 Structured Database Search Agent: SQL-Like Queries

- File: structured_db_search.py
- Function: query_database(state: State)

Why a Structured DB Search?

- Direct lookup queries are more efficient for exact restaurant details.
- Menu items, categories, and ingredients are stored in structured format.
- FAISS is ineffective when we need precise database queries.

How It Works?

The structured database search operates on a preprocessed dataset (cleaned_menu_data.csv), ensuring efficient retrieval of restaurant, menu, and ingredient-related data. When a query is received, the system first extracts structured entities using an LLM, identifying key elements such as restaurant names, menu items, ingredients, and categories (e.g., "restaurant_name": "20 Spot"). These entities are then used to apply SQL-like filtering, refining results based on menu items, ingredient compositions, and pricing details. Finally, the system returns structured results, including menu descriptions, price ranges, and customer ratings, ensuring accurate and contextually relevant information is retrieved for response generation.

Limitations & Future Enhancements

Currently, the database query executes synchronously, meaning it waits for FAISS and Knowledge Graph searches to complete before proceeding, which can introduce unnecessary delays. To optimize response time, implementing parallel execution with LangGraph would allow the database query to run simultaneously with other retrieval agents, ensuring faster and more efficient processing.

2.4 LLM Graph Search Agent: Knowledge Graph-Based Retrieval

- File: llm_graph_search.py
- Function: query_knowledge_graph(state: State)

How this works?

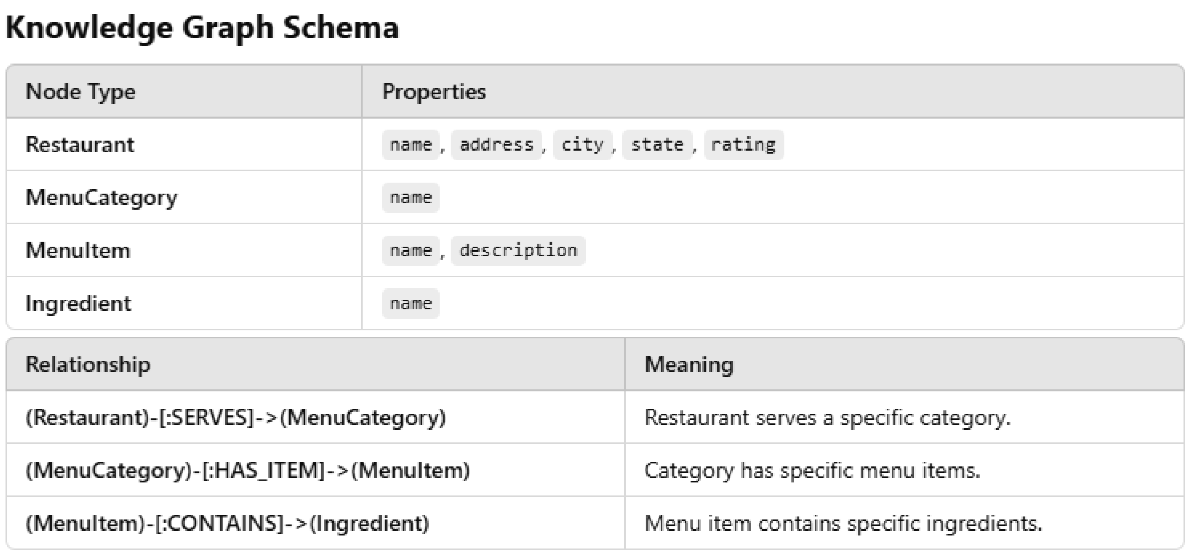
LLM receives the query prompt → It understands how to generate structured Cypher queries based on the schema.
Query optimization rules apply:

- Uses case-insensitive matching (toLower()).
- Ensures OPTIONAL MATCH is used to handle missing nodes.
- Ensures that aggregated results are returned (avoiding duplicates).

Returns only the Cypher query (no explanation, only executable code).
The generated Cypher query is executed using Neo4j's run() method.
The returned data is structured (JSON-like format).
If the query execution fails, it returns an empty list to prevent system crashes.
The final results are stored in the chatbot's state dictionary, so that they can be used in subsequent augmentation and response generation steps.

Why Use a Knowledge Graph (Neo4j)?

- Entities and relationships matter. - The problem is entirely based on entities and relationships. (eg. (Restaurant)-[:SERVES]->(MenuCategory))
- Multi-hop reasoning helps answer complex queries.
- Track menu evolution over time.



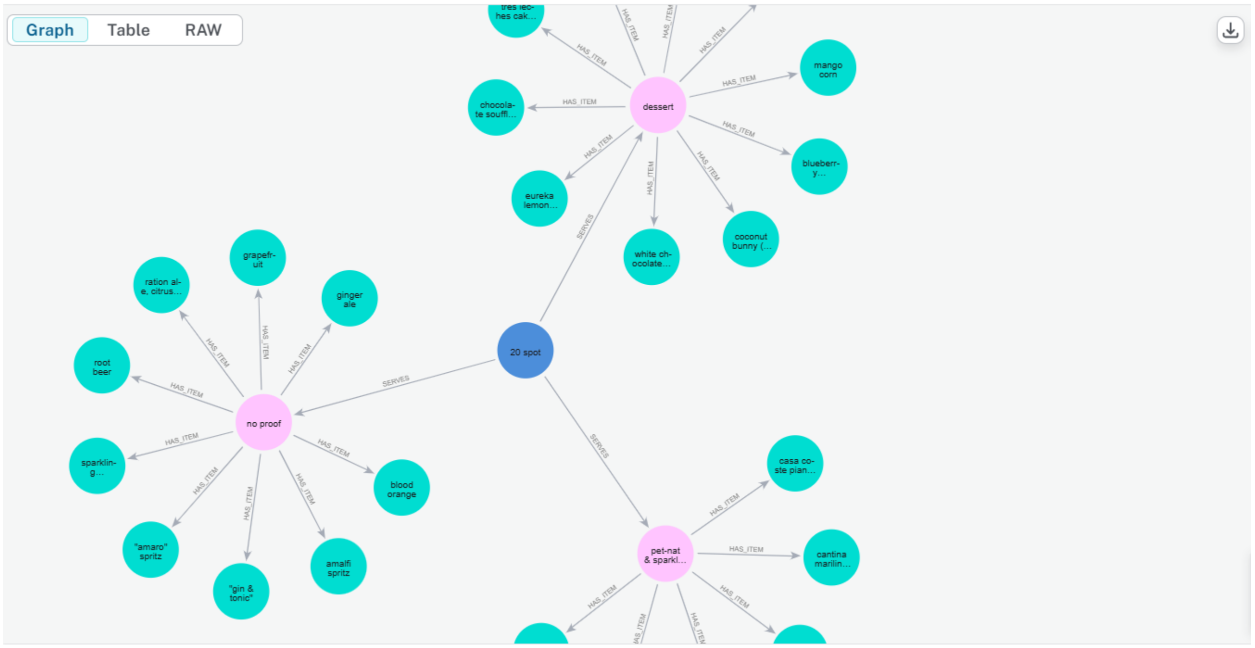
Knowledge Graph Schema

Node Type	Properties
Restaurant	`name`, `address`, `city`, `state`, `rating`
MenuCategory	`name`
MenuItem	`name`, `description`
Ingredient	`name`

Relationship	Meaning
(Restaurant)-[:SERVES]->(MenuCategory)	Restaurant serves a specific category.
(MenuCategory)-[:HAS_ITEM]->(MenuItem)	Category has specific menu items.
(MenuItem)-[:CONTAINS]->(Ingredient)	Menu item contains specific ingredients.

Example Query & Knowledge Graph Response:

MATCH (r:Restaurant {name: "20 Spot"}) OPTIONAL MATCH (r)-[serves:SERVES]->(mc:MenuCategory) OPTIONAL MATCH (mc)-[has_item:HAS_ITEM]->(mi:MenuItem) RETURN r, serves, mc, has_item, mi LIMIT 25;



Limitations & Future Enhancements

- Graph queries currently run after FAISS and DB Search.
- Future: Knowledge Graph should execute asynchronously and trigger other agents in parallel.

2.5 Google Search Agent: Fetching External Data

- File: google_search.py
- Function: google_search(state: State)
-

The Google Search Agent is designed to fetch external real-time data to supplement structured retrieval methods. It first queries the Google API to retrieve the top 3 results, then scrapes content using BeautifulSoup, and finally summarizes the extracted text using an LLM to generate a concise response. However, the current implementation faces several inefficiencies. Fetching search results can take 3-5 seconds, introducing latency, which can be mitigated by parallelizing execution with LangGraph. Additionally, some websites block scraping, limiting the availability of data, which can be addressed by using the Serper API for structured search results. Another major issue is that LLM-based summarization is slow and computationally expensive for long texts; to improve efficiency, models like Longformer or BART can be used for extractive summarization, reducing processing time while maintaining response quality.

2.6 Future Agent: Trend & Menu Innovation Analysis

Why Trends Matter?

Understanding food trends and menu innovation is critical for:

- Tracking the rise and decline of ingredients (e.g., plant-based meat, exotic spices).
- Identifying seasonal variations (e.g., pumpkin spice in fall, fresh berries in summer).
- Helping restaurants optimize menus by adding trending ingredients before they peak.

A dedicated Trend & Menu Innovation Agent would enable real-time tracking of food trends, analyzing menus dynamically, and forecasting future ingredient popularity.

How the Trend & Innovation Agent Works (Proposal)?

To track trends effectively, the system needs to ingest external data daily and structure it into a knowledge graph. The process involves:

Step 1: Scheduled Data Retrieval (Daily)

- Trigger automated data fetch every 24 hours.
- Fetch sources:
  - Restaurant menus (proprietary Menudata database).
  - Food blogs, industry reports, and news articles.
  - Google Trends data for ingredient searches.
  - Social media signals (e.g., mentions of ingredients on Twitter/Instagram).
- APIs Used:
  - Menudata's existing trend-tracking system (to be leveraged).
  - Web scraping for food blogs and review sites.

    - Google Trends API for ingredient search popularity.

Step 2: Structuring Data with an Agentic System

Once raw data is collected, an agentic system processes and organizes it into structured knowledge.

- Agent 1: Named Entity Extraction Agent
    - Extracts ingredients, dishes, restaurant names, cuisines from text data.
    - Example: Identifies "matcha" as a rising ingredient from a food blog.
- Agent 2: Sentiment & Popularity Analysis
    - Measures sentiment scores of menu trends (positive/negative reactions).
    - Counts mentions of ingredients across multiple sources.
- Agent 3: Temporal Structuring & Knowledge Graph Update
    - Adds trending ingredients to the Neo4j knowledge graph.
    - Creates new TRENDED_IN relationships for tracked ingredients.

Step 3: Storing in Knowledge Graph

Future Knowledge Graph Schema: To capture trends effectively, the following graph schema will be used.

Node Type	**Properties**
Trend	year, popularity_score, mentions
Ingredient	name, category, origin

Relationships	**Description**
(Ingredient) - [:TRENDED_IN {year, popularity_score, mentions}]->(Trend)	Stores ingredient trend data.
(MenuItem)-[:CONTAINS]->(Ingredient)	Links ingredients to menu items.

Step 4: Leveraging Menudata's Existing Technology

After my discussion with Sunny from Menudata, I learned that Menudata already has trend-tracking technology in place. Instead of building this system from scratch, we can:

- Leverage Menudata's proprietary database to retrieve existing trend data.
- Integrate their API into our knowledge graph instead of redundant scraping.
- Use their historical trend records to improve trend forecasting accuracy.

3. Generation: Structuring and Delivering Responses

Once all retrieval agents (FAISS, Structured DB, Knowledge Graph, and Google Search) return their results, the system first summarizes each source independently to extract the most relevant information.

This includes refining restaurant lists, menu items, ingredient trends, and historical insights into structured formats. Each dataset is processed separately based on its retrieval method and data structure—Google Search results undergo LLM-based summarization, while FAISS and Knowledge Graph results are filtered and ranked. After individual summarization, a final LLM call is made, where the system generates a coherent response by incorporating the user query, detected intent, and the structured summaries from all sources. The final response is formatted dynamically, ensuring that comparative queries use tables, historical queries are narrative, and entity-based responses are structured as bullet points or JSON outputs, making the information clear, concise, and query-relevant.

4. Conclusion

This report presents a comprehensive overview of the Retrieval-Augmented Generation (RAG) pipeline developed for AI-driven restaurant data retrieval and augmentation, addressing the Menudata Challenge. The system integrates multiple retrieval strategies—FAISS-based similarity search, structured database queries, knowledge graph-based reasoning, and real-time web search—to ensure robust, contextually relevant responses. A key enhancement proposed is the use of **LangGraph** for asynchronous execution, reducing latency and improving retrieval efficiency by parallelizing agent operations. Additionally, a **Trend & Menu Innovation Agent** has been outlined, leveraging Menudata's existing trend-tracking capabilities to forecast ingredient popularity and culinary shifts. Finally, a structured response generation mechanism ensures clarity and adaptability across different user queries. Future improvements include optimizing real-time search, refining multi-agent coordination, and enhancing response structuring for a seamless conversational AI experience.