AI-Based Recipe Generator & Recommender (LLM Cookbook)

Team Members:

- Raj Vithal Desai – RVD240000
- Tanvi Vijay Deore – TXD240009
- Khushi Bajaj Neeraj Bajaj – KXN240009

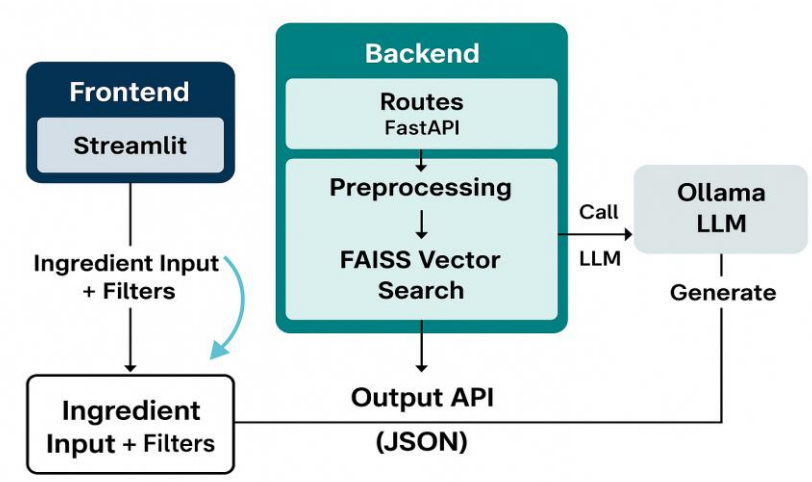
GitHub Repository:
https://github.com/RajDesai-18/llm-cookbook

YouTube Demo:
https://youtu.be/dU8jeaVeWb0

Project Description and Objectives:

This project addresses the common question: "What can I cook with what I already have at home?" By leveraging semantic search and generative NLP models, our application allows users to input available ingredients and optionally filter results based on allergens, dietary preferences, or cuisine types. If no suitable recipe is found, the system falls back to generating a custom recipe using a local LLM.

System Architecture:



The figure above illustrates the end-to-end architecture of the project. The frontend (built with Streamlit) accepts ingredient inputs and filters, which are processed through the backend (FastAPI). The backend performs preprocessing and semantic vector search using FAISS. If no close match is found, a local LLM (Ollama) is invoked to generate a new recipe. The final result is returned in a structured JSON format.

Technical Stack:

Frontend: Streamlit
Backend: FastAPI + Python

NLP Models:

- **Semantic Search**: MiniLM + FAISS
- **Recipe Generation**: Ollama (Nous Hermes)
  Data: Structured CSV of cleaned recipes including fields like title, ingredients, prep time, cook time, cuisine, allergens, instructions

Key Features:

- Parse free-text ingredient inputs
- Retrieve semantically similar recipes using cosine similarity
- Fallback recipe generation via LLM (if no close match found)
- Filters for:
  - Max preparation/cook time
  - Excluded allergens
  - Dietary preferences
  - Preferred cuisine
- Ingredient substitution for excluded allergens
- Structured JSON output for frontend integration

TEAM CONTRIBUTIONS

Raj Vithal Desai

- Built the Streamlit frontend for user interaction
- Integrated Ollama-based LLM (Nous Hermes) for fallback recipe generation
- Designed and implemented structured recipe generation logic
- Connected backend APIs to frontend UI

Lessons Learned:

→ **Frontend-first design enables faster iteration**: Building the UI early allowed for quick feedback loops and easier feature testing.
→ **Local LLMs are viable alternatives**: Using Ollama locally offered control and privacy advantages over cloud-based models.
→ **Prompt engineering is critical**: Structuring prompts clearly was key to generating high-quality, consistent recipe outputs.
→ **API integration uncovers edge cases**: Connecting frontend and backend revealed format mismatches and helped refine data consistency.

Tanvi Vijay Deore

- Scraped and cleaned the recipe dataset from multiple sources (e.g., Kaggle, Food.com)
- Performed comprehensive preprocessing. Evaluated multiple techniques (pretrained models, training custom models, rule-based methods) to choose optimal strategies for labelling prep time, allergens, and servings.
- Generated semantic embeddings and FAISS index for recipe search
- Handled outliers in prep time intelligently by creating subsets

Lessons Learned:

→ **Data quality is crucial**: Preprocessing had a massive impact on model performance, emphasizing the need for clean, labeled data.
→ **Outlier handling improves robustness**: Subsetting data for anomalous prep times enhanced model reliability and prevented skewed predictions.
→ **Strategy selection matters**: Trying different methods (pretrained, rule-based, LLM) for tasks like allergen detection and prep time labeling highlighted the importance of evaluating trade-offs between accuracy, complexity, and interpretability.
→ **Semantic search is powerful**: Gained hands-on experience with sentence embeddings and FAISS, showcasing their utility beyond typical classification tasks.

Khushi Bajaj Neeraj Bajaj

- Set up the FastAPI backend structure and modularized API routes
- Implemented ingredient parsing, recipe searching, and filtering logic
- Developed allergen exclusion and dietary preference filters
- Ensured API endpoints produced structured responses

Lessons Learned:

→ **Parsing requires nuance**: Ingredient parsing exposed the complexity of real-world text data, including edge cases and ambiguity.
→ **Filtering logic needs precision**: Building accurate allergen/dietary filters was key to maintaining trust and functionality.
→ **FastAPI is efficient and developer-friendly**: Enabled rapid backend iteration and testing.
→ **Team coordination is key**: Maintaining consistent data formats across components was crucial for seamless integration.

Joint Work:

- Collaborated to integrate the full system
- Shared responsibilities for bug-fixing, testing, and refining the experience

Challenges Faced & Solutions

1. Prep Time Outliers in Dataset

- Some recipes had extremely high prep times due to fermentation or soaking (>24 hours), while others were <15 minutes
- ***Solution***: Split dataset into low/high prep time subsets and focused embedding training on lower half to reduce model skew

2. Inconsistent Cuisine Detection

- LLM occasionally returned a list instead of a string (e.g., ["Indian"]) leading to .lower() errors
- ***Solution***: Applied type checking and standardization across all responses

3. Frontend Parsing Inconsistencies

- Some API responses weren't structured correctly or returned unexpected keys
- ***Solution***: Defined clean response schemas and added validation logic for robust integration

4. Raw Ingredient Input Cleaning

- Users entered ingredients in free form (e.g., "2 cups of diced tomatoes"), which hurt search accuracy
- ***Solution***: Created /parse-ingredients endpoint that extracts only the main ingredients

User Testing:

To evaluate the real-world usability of our system, we shared our MVP with 5 external users and gathered valuable feedback. A key usability challenge surfaced when users entered ingredients in free-form text (e.g., "2 cups of diced tomatoes"), which reduced the accuracy of semantic search. This was addressed by implementing a /parse-ingredients endpoint that extracts clean, standardized ingredient names for reliable matching.

Users also noted that generated recipes lacked personalized names and were labeled "Unnamed Recipe."

Key outcomes from user testing:

- Simplified the user interface for better navigation
- Made allergen filtering clearer and more intuitive
- Added a fallback mechanism to auto-generate recipes when no semantic match is found

Performance

- Recipe search via FAISS returns results in <1 second
- LLM fallback generation returns structured recipes in 4–5 seconds
- Backend supports all expected filters, including nested filters like vegan + gluten-free + <15 mins + exclude nuts

Next Steps

- Implement advanced ingredient substitution suggestions (e.g., for unavailable pantry items)
- Build a robust **React + Tailwind CSS frontend** for production use
- Add user feedback collection for result quality
- Containerize backend for easier deployment (Docker)
- Optional: Fine-tune open LLM on cooking-specific domain for better result quality

Self-Scoring Table:

Criteria	Max	Raj	Tanvi	Khushi
Significant Exploration	80	80	80	80
Innovation	30	30	25	25
Technical Complexity	10	10	10	10
Lessons & Improvements	10	10	10	10
Architecture & Repo Quality	10	10	10	10
User Testing	10	10	5	5
Revenue (not applicable)	10	0	0	0
Total		150	140	140

Conclusion

This project delivered a working NLP-powered MVP for intelligent recipe generation. By combining semantic search with fallback LLM generation, our system solves a real-world problem with both technical depth and usability in mind. The architecture is modular and scalable, and the core logic is built to handle nuanced user needs such as allergen filters and pantry-based personalization.