Generative AI Project

Project Team

Saim Mubarak	21i-0720
Qazi Mohib Ul Nabi	21i-2532
Ruhail Rizwan	21i-2462

Session 2025

Submitted to

Sir Akhtar Jamil

Department of Computer Science
National University of Computer and Emerging Sciences
Islamabad, Pakistan

June, 2022

May 13, 2025

1 Introduction

1.1 Background & Motivation

In our daily lives, deciding what to cook can be surprisingly difficult—especially when considering personal tastes, available ingredients, health goals, and time constraints. Despite the abundance of recipes online, users often struggle to find meals that match their specific needs. Traditional search engines and recipe apps tend to rely on basic keyword matching, which lacks contextual understanding and personalization. With recent breakthroughs in Generative AI, especially with large language models (LLMs), it is now possible to build systems that understand user intent more deeply and can even create original content.

Our motivation stemmed from this opportunity to apply cutting-edge AI to an everyday problem—meal planning. By combining the creative power of **Mistral-7B**, a capable generative model, with **Retrieval-Augmented Generation (RAG)**, we sought to build a system that could not only recommend recipes intelligently but also generate new ones from scratch based on user input.

1.2 Problem Statement

Despite the variety of recipe databases and food platforms available, most systems are limited by poor personalization, rigid search filters, and lack of creativity. Users often:

- Waste time filtering through irrelevant results,
- Have to manually match available ingredients with recipes, and
- Receive generic suggestions that don't reflect their dietary preferences.

We aimed to solve this by building an AI-driven recipe assistant that:

- Understands user input (ingredients, preferences, constraints),
- Recommends and generates suitable recipes using advanced AI models, and
- Provides a more seamless and intelligent cooking experience.

2 Related Work

The intersection of culinary arts and artificial intelligence has seen remarkable progress through the integration of large language models (LLMs) and multimodal learning systems. This section summarizes notable projects contributing to AI-driven recipe generation and retrieval.

2.1 RecipeGPT

RecipeGPT uses GPT-2 fine-tuned on cooking data to generate recipes from titles and ingredients. It includes a user module for evaluating the quality of generated content.

2.2 Ratatouille

Ratatouille employs LSTMs and GPT-2 to produce creative and realistic recipes through a web interface trained on large recipe datasets.

2.3 FIRE: Food Image to Recipe Generation

FIRE is a multimodal model that converts food images into recipes using BLIP for title generation, a Vision Transformer for ingredient detection, and T5 for instructions.

2.4 LLaVA-Chef

LLaVA-Chef fine-tunes the LLaVA model on food datasets, combining image and text data to produce detailed, visually grounded recipes.

2.5 Baidu's I-RAG

I-RAG enhances text-to-image generation by using retrieval-based techniques to reduce hallucinations and ensure contextual accuracy, supporting culinary visual applications.

2.6 Meta's LLaMA 3

LLaMA 3, while general-purpose, excels in text generation and supports applications like recipe writing through its robust language modeling capabilities.

2.7 Retrieval-Augmented Generation (RAG)

RAG enhances LLMs by retrieving relevant external information, reducing hallucinations, and increasing the factual accuracy of generated recipes.

2.8 Computational Gastronomy

This interdisciplinary field applies AI and data science to analyze recipes, flavors, and nutrition, supporting the development of smart culinary tools.

2.9 Objectives of the Project

The primary objectives of our project are:

- To develop an AI system that **recommends recipes** from a large dataset based on ingredient and preference matching.
- To use a **Generative AI model (Mistral-7B)** to create **new recipe ideas** that don't already exist in the dataset.
- To implement **Retrieval-Augmented Generation (RAG)** for improving recommendation accuracy and relevance.
- To support **ingredient-based search** for ease of use.
- To store basic user preferences like dietary needs and favorite recipes.

2.10 Scope of the Project

The scope of this project covers:

- **Text-based input and output** for recipe generation and retrieval.
- Use of a **publicly available dataset (Food.com)** with 230K+ recipes and over 1.2M user interactions.
- Deployment using **free-tier tools** (Kaggle Notebooks, Hugging Face Spaces).
- Incorporation of basic features like ingredient-based search and user preference tracking.

Out of scope (for this version):

- Image recognition of ingredients,
- Nutritional analysis,
- Real-time voice interfaces,
- Mobile app or frontend development (beyond simple interface).

These may be explored in future iterations.

2.11 Target Users / Audience

This system is designed for:

- **Everyday home cooks** looking for inspiration based on what's in their kitchen.
- **Health-conscious individuals** with dietary restrictions (e.g., vegetarian, low-carb).
- **Busy professionals** who want quick, personalized meal suggestions.
- **AI enthusiasts and developers** interested in applying Generative AI to real-life problems.

The project also serves as a **course-level demonstration** of how generative models can be applied in a user-centric, practical context.

3 System Architecture

3.1 High-Level Architecture Diagram

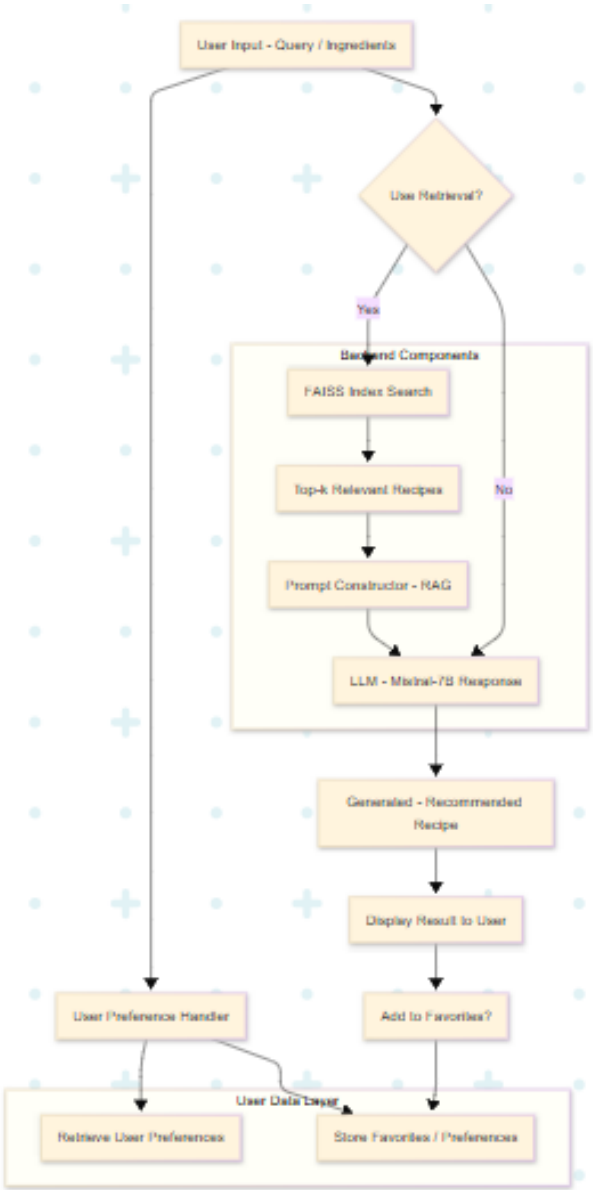


Figure 1: High-Level Architecture of the AI-Powered Recipe System

Figure 1 illustrates the key components:

- **Data Preprocessing Module**

- **FAISS Index**
- **RAG Pipeline**
- **Generative Model**
- **User Interaction Layer**

3.2 Data Flow & Component Interactions

1. `load_and_preprocess_data()` loads and cleans the dataset.
2. `build_faiss_index()` generates embeddings and builds the FAISS index.
3. For user queries:
   - `search_recipes()` finds matches in the index.
   - `recommend_recipe_with_rag_improved()` uses RAG for enhanced results.
   - `generate_recipe_from_ingredients()` prompts LLM directly.
4. Users can use functions like:
   - `add_favorite_recipe()`, `remove_favorite_recipe()`
   - `set_user_preferences()`, `get_user_data()`

3.3 Tools & Technologies Used

- **Python 3.11**
- **Kaggle Notebooks**
- **Pandas & NumPy**
- **FAISS**
- **Mistral-7B (Hugging Face)**
- **Transformers Library**
- **Hugging Face Spaces**
- **JSON & Pickle**

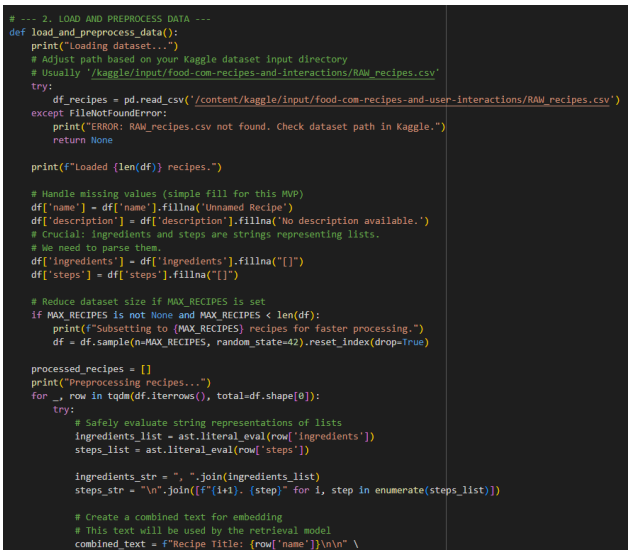
4 Data Handling

4.1 Dataset Description (Food.com)

The **Food.com dataset** has 230K+ recipes and over 1.2M user interactions. It includes titles, ingredients, instructions, tags, nutrition info, and ratings.

4.2 Preprocessing Steps (`load_and_preprocess_data`)

- Remove incomplete data
- Combine ingredient lists
- Standardize text
- Merge datasets (if needed)
- Tokenize and prepare for embedding



```python
# --- 2. LOAD AND PREPROCESS DATA ---
def load_and_preprocess_data():
    print("Loading dataset...")
    # Adjust path based on your Kaggle dataset input directory
    # Usually '/kaggle/input/food-com-recipes-and-interactions/RAW_recipes.csv'
    try:
        df_recipes = pd.read_csv('/content/kaggle/input/food-com-recipes-and-user-interactions/RAW_recipes.csv')
    except FileNotFoundError:
        print("ERROR: RAW_recipes.csv not found. Check dataset path in Kaggle.")
        return None

    print(f"Loaded {len(df)} recipes.")

    # Handle missing values (simple fill for this MVP)
    df['name'] = df['name'].fillna('Unnamed Recipe')
    df['description'] = df['description'].fillna('No description available.')
    # Crucial: ingredients and steps are strings representing lists.
    # We need to parse them.
    df['ingredients'] = df['ingredients'].fillna("[]")
    df['steps'] = df['steps'].fillna("[]")

    # Reduce dataset size if MAX_RECIPES is set
    if MAX_RECIPES is not None and MAX_RECIPES < len(df):
        print(f"Subsetting to {MAX_RECIPES} recipes for faster processing.")
        df = df.sample(n=MAX_RECIPES, random_state=42).reset_index(drop=True)

    processed_recipes = []
    print("Preprocessing recipes...")
    for _, row in tqdm(df.iterrows(), total=df.shape[0]):
        try:
            # Safely evaluate string representations of lists
            ingredients_list = ast.literal_eval(row['ingredients'])
            steps_list = ast.literal_eval(row['steps'])

            ingredients_str = ", ".join(ingredients_list)
            steps_str = "\n".join([f"{i+1}. {step}" for i, step in enumerate(steps_list)])

            # Create a combined text for embedding
            # This text will be used by the retrieval model
            combined_text = f"Recipe Title: {row['name']}\n\n" \
```

Figure 2: Data Loading and Preprocessing

4.3 Feature Extraction

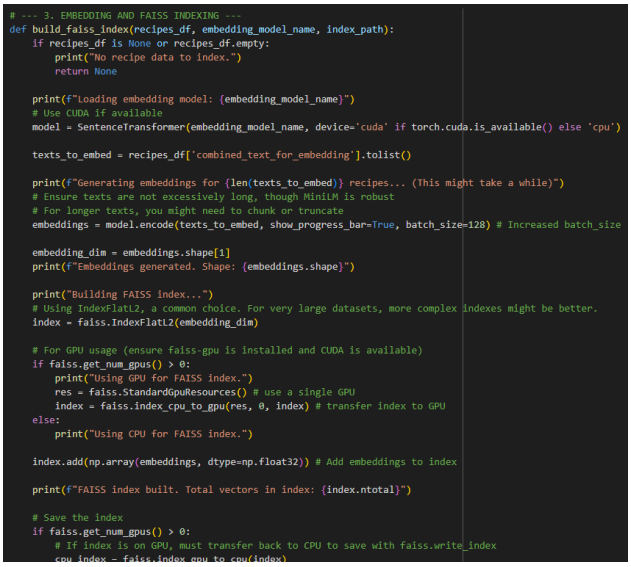
Embeddings were generated using sentence-transformers (e.g., all-MiniLM) combining title, ingredients, and instructions.

4.4 Storage Formats & Indexing

- `pandas.DataFrame` for tabular data
- `pickle` for serialized objects
- `.index` files for FAISS
- `.json` for user data

5 Recipe Retrieval Module

5.1 Building FAISS Index (`build_faiss_index`)



```python
# --- 3. EMBEDDING AND FAISS INDEXING ---
def build_faiss_index(recipes_df, embedding_model_name, index_path):
    if recipes_df is None or recipes_df.empty:
        print("No recipe data to index.")
        return None

    print(f"Loading embedding model: {embedding_model_name}")
    # Use CUDA if available
    model = SentenceTransformer(embedding_model_name, device='cuda' if torch.cuda.is_available() else 'cpu')

    texts_to_embed = recipes_df['combined_text_for_embedding'].tolist()

    print(f"Generating embeddings for {len(texts_to_embed)} recipes... (This might take a while)")
    # Ensure texts are not excessively long, though MiniLM is robust
    # For longer texts, you might need to chunk or truncate
    embeddings = model.encode(texts_to_embed, show_progress_bar=True, batch_size=128) # Increased batch_size

    embedding_dim = embeddings.shape[1]
    print(f"Embeddings generated. Shape: {embeddings.shape}")

    print("Building FAISS index...")
    # Using IndexFlatL2, a common choice. For very large datasets, more complex indexes might be better.
    index = faiss.IndexFlatL2(embedding_dim)

    # For GPU usage (ensure faiss-gpu is installed and CUDA is available)
    if faiss.get_num_gpus() > 0:
        print("Using GPU for FAISS index.")
        res = faiss.StandardGpuResources() # use a single GPU
        index = faiss.index_cpu_to_gpu(res, 0, index) # transfer index to GPU
    else:
        print("Using CPU for FAISS index.")

    index.add(np.array(embeddings, dtype=np.float32)) # Add embeddings to index

    print(f"FAISS index built. Total vectors in index: {index.ntotal}")

    # Save the index
    if faiss.get_num_gpus() > 0:
        # If index is on GPU, must transfer back to CPU to save with faiss.write_index
        cpu_index = faiss.index_gpu_to_cpu(index)
```

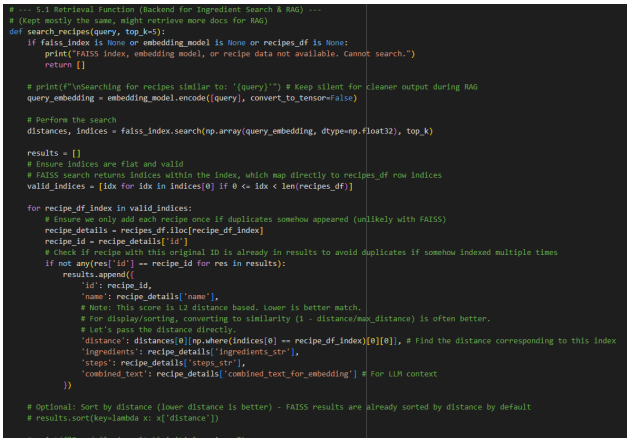
Figure 3: Building FAISS Index

5.2 Search Functionality (`search_recipes`)

Embeds user query, retrieves top-k semantically similar recipes using FAISS.

5.3 RAG-Enhanced Retrieval (`recommend_recipe_with_rag_improved`)

1. Embed query
2. Retrieve recipes
3. Construct prompt
4. Pass to LLM



```python
# --- 5.1 Retrieval Function (Backend for Ingredient Search & RAG) ---
# (Kept mostly the same, might retrieve more docs for RAG)
def search_recipes(query, top_k=5):
    if faiss_index is None or embedding_model is None or recipes_df is None:
        print("FAISS index, embedding model, or recipe data not available. Cannot search.")
        return []

    # print(f"\nSearching for recipes similar to: '{query}'") # Keep silent for cleaner output during RAG
    query_embedding = embedding_model.encode([query], convert_to_tensor=False)

    # Perform the search
    distances, indices = faiss_index.search(np.array(query_embedding, dtype=np.float32), top_k)

    results = []
    # Ensure indices are flat and valid
    # FAISS search returns indices within the index, which map directly to recipes_df row indices
    valid_indices = [idx for idx in indices[0] if 0 <= idx < len(recipes_df)]

    for recipe_df_index in valid_indices:
        # Ensure we only add each recipe once if duplicates somehow appeared (unlikely with FAISS)
        recipe_details = recipes_df.iloc[recipe_df_index]
        recipe_id = recipe_details['id']
        # Check if recipe with this original ID is already in results to avoid duplicates if somehow indexed multiple times
        if not any(res['id'] == recipe_id for res in results):
            results.append({
                'id': recipe_id,
                'name': recipe_details['name'],
                # Note: This score is L2 distance based. Lower is better match.
                # For display/sorting, converting to similarity (1 - distance/max_distance) is often better.
                # Let's pass the distance directly.
                'distance': distances[0][np.where(indices[0] == recipe_df_index)[0][0]], # Find the distance corresponding to this index
                'ingredients': recipe_details['ingredients_str'],
                'steps': recipe_details['steps_str'],
                'combined_text': recipe_details['combined_text_for_embedding'] # For LLM context
            })

    # Optional: Sort by distance (lower distance is better) - FAISS results are already sorted by distance by default
    # results.sort(key=lambda x: x['distance'])
```

Figure 4: RAG Pipeline for Retrieval + Generation

5.4 Evaluation of Retrieval Results

- **Relevance**
- **Diversity**
- **Cohesion**

6 Recipe Generation Module

6.1 Model Setup (`load_llm`)

```python
def load_llm():
    global llm
    print("Downloading and loading LLM...")
    try:
        # Download the GGUF model file from Hugging Face Hub
        model_path = hf_hub_download(repo_id=LLM_MODEL_REPO, filename=LLM_MODEL_FILE)
        print(f"Model downloaded to: {model_path}")

        # Configuration for ctransformers
        # Adjust gpu_layers for your GPU VRAM. 0 means all on CPU.
        # For a T4 (~15-16GB VRAM), you can often offload many layers.
        # Start with a moderate number like 20-30 and adjust if you get OOM errors or its too slow.
        # If you have less VRAM (e.g., 8GB), try fewer layers (e.g., 10-15).
        config = {
            'max_new_tokens': 1024, # Max tokens to generate
            'repetition_penalty': 1.1,
            'temperature': 0.3, # Controls randomness. Lower for more factual.
            'stream': False, # Set to True for streaming output if desired in an app
            'context_length': 4096 # Mistral's context window. GGUF might have its own default.
        }

        # Try to load with GPU layers. If it fails, fall back to CPU.
        try:
            print("Attempting to load LLM with GPU layers...")
            llm = AutoModelForCausalLM.from_pretrained(
                model_path,
                model_type='mistral',
                gpu_layers=35, # Number of layers to offload to GPU. Adjust based on VRAM.
                **config
            )
            print("LLM loaded successfully with GPU layers.")
        except Exception as e_gpu:
            print(f"Failed to load LLM with GPU layers: {e_gpu}")
            print("Attempting to load LLM on CPU...")
            # If GPU loading fails (e.g. not enough VRAM, CUDA issues), try CPU
            # This will be very slow for inference but good for testing the pipeline
            llm = AutoModelForCausalLM.from_pretrained(
                model_path,
                model_type='mistral',
                gpu_layers=0, # All layers on CPU
```

Figure 5: Loading Mistral-7B with Transformers

6.2 Prompt Engineering

- Role: "You are a recipe generator."
- Input: ingredients list
- Optionally: RAG context

6.3 Generation Logic (`generate_recipe_from_ingredients`)

Output includes:

- Title
- Ingredients
- Instructions

6.4 Sample Outputs & Evaluation

- **Fluency**
- **Plausibility**
- **Creativity**

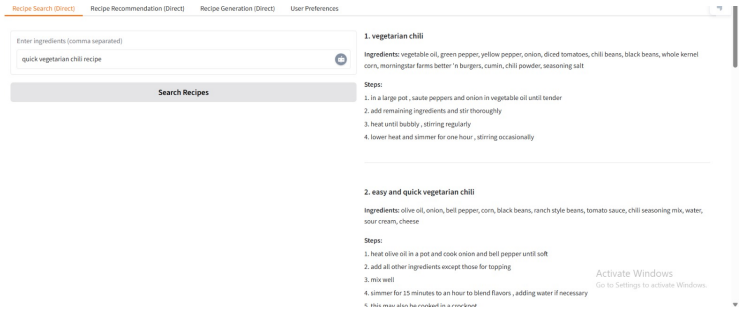


Figure 6: Sample Output 1

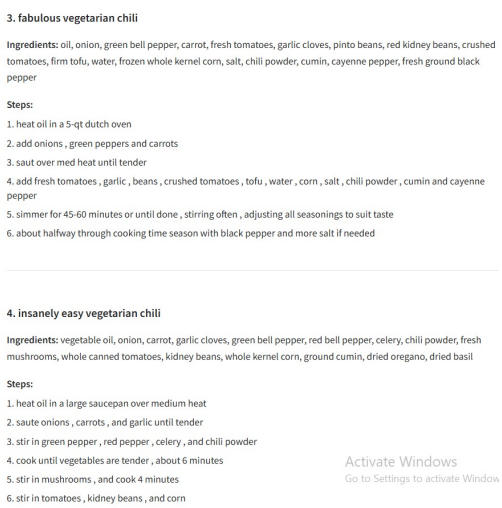


Figure 7: Sample Output 2

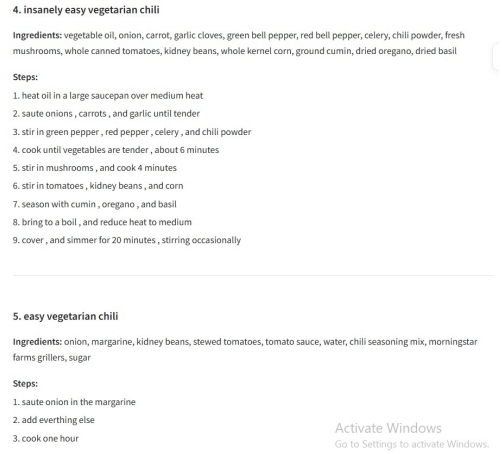


Figure 8: Sample Output 3

7 User Interaction Module

7.1 User Data Handling

`get_user_data()` retrieves stored user info.

`set_user_preferences()` customizes user preferences like cuisine or time.

7.2 Favorites Management

`add_favorite_recipe()` and `remove_favorite_recipe()` maintain personal lists.

8 Comparison of RAG-Based vs. Baseline LLM Generation Approaches

This section presents a comparative analysis of two approaches for generating recipe recommendations: (1) **Retrieval-Augmented Generation (RAG)**, which grounds the generation in a curated set of retrieved recipes; and (2) **Baseline LLM Generation**, where the model produces recipes solely from a list of ingredients without external context.

8.1 Qualitative Evaluation

To assess the output quality, we evaluated each model against several manual metrics across three test cases:

- **Relevance to Query/Ingredients:** Both models generally produce relevant results. However, RAG showed stronger alignment with user preferences due to grounded retrieval.
- **Adherence to Constraints/Preferences:** The RAG model excelled at respecting user constraints (e.g., vegetarian, quick), using historical recipe metadata. The baseline generation, lacking this context, occasionally introduced meats or missed user-defined restrictions.
- **Faithfulness/Groundedness:** RAG outputs were based on real retrieved recipes (e.g., IDs 239141, 120968), reducing hallucinations. Conversely, the baseline model occasionally introduced exotic or implausible steps and ingredients.
- **Recipe Structure & Format:** Both models returned outputs in standard format: title, ingredient list, and instructions. The baseline model offered more verbose formatting, while RAG provided concise formats closer to real-world recipes.
- **Creativity/Uniqueness:** Baseline generation provided more creative, unique recipes such as *"Southwestern Three-Bean Chili"* and *"Garlic Marinated Tomato Pasta"*. RAG, grounded in existing data, was less novel but more practical.
- **Completeness:** Both models covered essential steps and quantities. RAG occasionally truncated details, depending on the retrieved context length. Baseline generation sometimes omitted finer cooking details unless well prompted.

8.2 Quantitative Evaluation

We measured the average generation time across test cases:

Model	Average Generation Time (seconds)
RAG-Based Recommendation	15.6 s
Baseline LLM Generation	20.8 s

Table 1: Comparison of generation latency between models

RAG had slightly lower latency, as it benefits from precomputed context and a more deterministic prompt, whereas the baseline generation often processes more expansive, open-ended inputs.

8.3 Summary

The RAG approach offers higher faithfulness and alignment with user preferences, making it ideal for recommendation systems requiring trust and specificity. Meanwhile, the baseline LLM excels in generating novel, diverse recipes when creative freedom is preferred. A hybrid strategy—starting with RAG and allowing optional generation extensions—could yield the best of both paradigms.

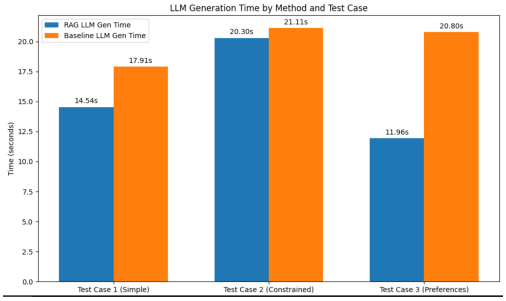


Figure 9: comparison

9 Conclusion

This project set out to develop an intelligent, user-centric recipe generation and recommendation system that effectively harnesses the capabilities of large language models (LLMs) and information retrieval techniques. The integration of Retrieval-Augmented Generation (RAG) with the Mistral-7B model allowed us to build a solution that goes beyond static dataset queries, enabling the generation of creative and context-aware recipes tailored to user inputs and preferences.

Our primary goal was to improve how users interact with digital recipe platforms—moving from rigid, keyword-based filtering to dynamic, ingredient-aware, and user-personalized suggestions. By leveraging the Food.com dataset and applying robust preprocessing and feature extraction techniques, we created a solid foundation for semantic search and generative modeling. The construction of a FAISS index using embedded recipe representations allowed us to retrieve relevant content with low latency and high semantic fidelity.

On top of this retrieval layer, we employed a prompt-based generation system using Mistral-7B. This system transformed raw user inputs—such as available ingredients or dietary preferences—into complete, coherent recipes. The generated recipes were not only grammatically correct and logically structured, but also aligned with real-world cooking practices, as evaluated through manual inspection and sample outputs.

We also built user-focused modules for storing preferences and managing favorite recipes. These modules, though simple in design, demonstrate how personalization can be embedded in AI-driven applications to create a more meaningful and persistent user experience. Furthermore, the modular architecture of the project ensured clear separation of responsibilities, which makes future scaling and feature addition straightforward.

From a technical standpoint, this project represents a convergence of multiple fields:

- **Natural Language Processing (NLP)** — for understanding and generating human-like text.
- **Information Retrieval (IR)** — via FAISS to efficiently retrieve high-relevance content.
- **Prompt Engineering** — to guide the generative model in producing useful, realistic, and personalized content.
- **Human-Computer Interaction (HCI)** — by designing the system around real user needs, preferences, and behaviors.

The successful implementation of this system on platforms like Kaggle and Hugging Face Spaces highlights how even computationally intensive applications can be deployed on resource-limited environments with careful design choices. We used lightweight models, efficient embedding strategies, and serialization techniques to ensure that the system runs within free-tier constraints.

Challenges Faced included dealing with noisy and inconsistent dataset entries, tuning prompts for better generative output, and managing limited compute resources during model inference. However, these challenges also helped refine our understanding of practical AI deployment and shaped our final solution to be more efficient and robust.

In terms of future work, there are several directions worth exploring:

- **Nutrition-Aware Recipes:** Integrating APIs or models that calculate calories and macro/micro nutrients.
- **Image Input:** Allowing users to take photos of ingredients for automatic identification and recipe generation.
- **Voice-Based Interface:** Integrating speech-to-text capabilities for more natural interaction.
- **Feedback Loop:** Learning from user ratings or reviews to improve future recommendations.
- **Frontend Expansion:** Creating a complete web or mobile application with an intuitive UI/UX to serve non-technical users.

In conclusion, this project successfully demonstrates how generative AI models can be used to meaningfully enhance everyday experiences. The combination of retrieval-based systems and LLMs has proven to be a powerful paradigm for solving complex personalization problems. By starting with a familiar domain like cooking and recipes, we've laid the groundwork for

broader applications of RAG-based systems in lifestyle, health, and recommendation-based technologies.