

RecipeGPT Project Report

Submitted for

- Avtans Kumar (AXK220317)
- Anusha Gupta (AXG230026)

Repo Link -

https://github.com/vanguard07/Project_Recipe

Video Link - https://youtu.be/U6_SnZBQbIA

Project Overview

RecipeGPT is an AI-powered application that helps users save, manage, and interact with recipes in a smarter way. These days, people often come across interesting recipes while browsing websites or social media. While many users bookmark or save these recipes, it can be hard to remember where they found them or to find the exact details later.

RecipeGPT makes this easier by allowing users to store recipes in one place and "chat" with them later. This means users can ask questions, get suggestions, or even modify recipes based on their preferences. The app uses advanced Natural Language Processing (NLP) and Retrieval-Augmented Generation (RAG) to understand and respond to user queries. It offers two main modes: a GPT based mode and a Langchain/RAG for answering cooking questions and giving suggestions based on the user's saved recipes. With a simple interface and intelligent backend, RecipeGPT helps users manage recipes more easily and adapt them to fit their dietary needs.

Problem Statement

Finding and using recipes online can be a frustrating experience. Users often see many recipes while browsing the internet or scrolling through social media. Although they may save or bookmark these recipes, it can be hard to find them later or remember the exact ingredients and steps. Even when users manage to store recipes, they face other challenges:

- Messy Recipe Formats: Recipes online are often unstructured, making it difficult to organize ingredients and instructions clearly.
- Difficult to Search: Bookmarked recipes are usually not searchable by ingredients, cooking time, or dietary needs.

- No Personalization: Most recipe platforms do not offer options to customize recipes based on user preferences or health requirements.
- No Easy Interaction: Users cannot ask follow-up questions or get suggestions based on their saved recipes.

RecipeGPT solves these problems by helping users store recipes in a structured way and interact with them through natural language. It makes it easy to search, modify, and understand recipes—turning casual inspiration into easy-to-follow cooking.

Tech Stack

Layer	Technology
Frontend	Next.js, React, Tailwind CSS
Backend	FastAPI, OpenAI APIs, LangChain
Vector Store	ChromaDB
Database	MongoDB
Container	Docker, Docker Compose

Theory

What is LangChain?

LangChain is a framework designed to simplify the development of applications that use large language models (LLMs). It provides modular building blocks that help developers combine models like GPT with other tools such as databases, APIs, and retrieval systems.

Key Features of LangChain:

- Memory: Maintains conversational history across messages.
- Retrievers: Connects to external vector stores (like Chroma) to fetch relevant documents.
- Chains: Composes sequences of LLM interactions into structured logic flows.
- Tool Integration: Easily integrates search engines, databases, APIs, and more.

LangChain is particularly useful for building Retrieval-Augmented Generation (RAG) systems, like in RecipeGPT, where the chatbot answers queries by fetching relevant recipe information from a vector store instead of relying only on the model's internal knowledge.

What is RAG (Retrieval-Augmented Generation)?

RAG is a method that enhances language models by combining them with an external knowledge base or vector store. Instead of relying solely on pre-trained model weights, it retrieves relevant documents in real time and provides them as context for generation.

In this project, LangChain implements RAG to enable smarter, grounded responses based on stored recipe data.

Backend Architecture

GPT Workflow in RecipeGPT:



1. You type a message

You might say something
“I want a quick vegetarian dinner”
or “Can you make this recipe gluten-free?”



2. The system figures out what you want

Are you looking for a new recipe?
Are you trying to tweak an existing one?
Or is it something else?



If you're searching for a recipe:

The system checks what you asked for
(e.g., vegetarian, under 30 mins).
Then it searches its recipe database
to find matches
If nothing matches exactly, it tries to
find the closest thing



4. If you're customizing a recipe:

It looks at your full chat history to
understand the recipe you're working
with

Here's ChatGPT-API-based chat (i.e. the `/search`, `/customize` and `/classify` endpoints) operates under the hood:

1. Client → API

- The front-end issues a POST to one of:
 1. `/classify` – to figure out if the user wants to search or customize

Each request carries JSON with at least:

```
{  
  "prompt": "...",  
  "chat_id": "..."    // optional for /search & /customize; required if  
 continuing a session  
}
```

2. classify_prompt Endpoint

1. **Parse request** → extract `prompt` and optional `chat_id`.
2. **Load `classify.txt`** from `prompts/` and inject the Recipe schema JSON.
3. **Build messages:**
 - System message with classification instructions + schema
 - Last N messages from Mongo (`chat_history_collection`) if `chat_id` provided
 - A user message: Classify this query: '{`prompt`}'
4. **Call OpenAI** via `oai_client.responses.create(...)` with `model="gpt-4.1-mini"` and a JSON-schema output format.
5. **Parse JSON** → { `type: "search"|"customize"|"other"`, `confidence`, `reasoning` }
6. **Branch:**
 - If "search" → forward the original request into `search_recipes(...)`
 - If "customize" → forward to `customize_recipe(...)`
 - Otherwise → return a "I can't help" message

3. search_recipes Endpoint

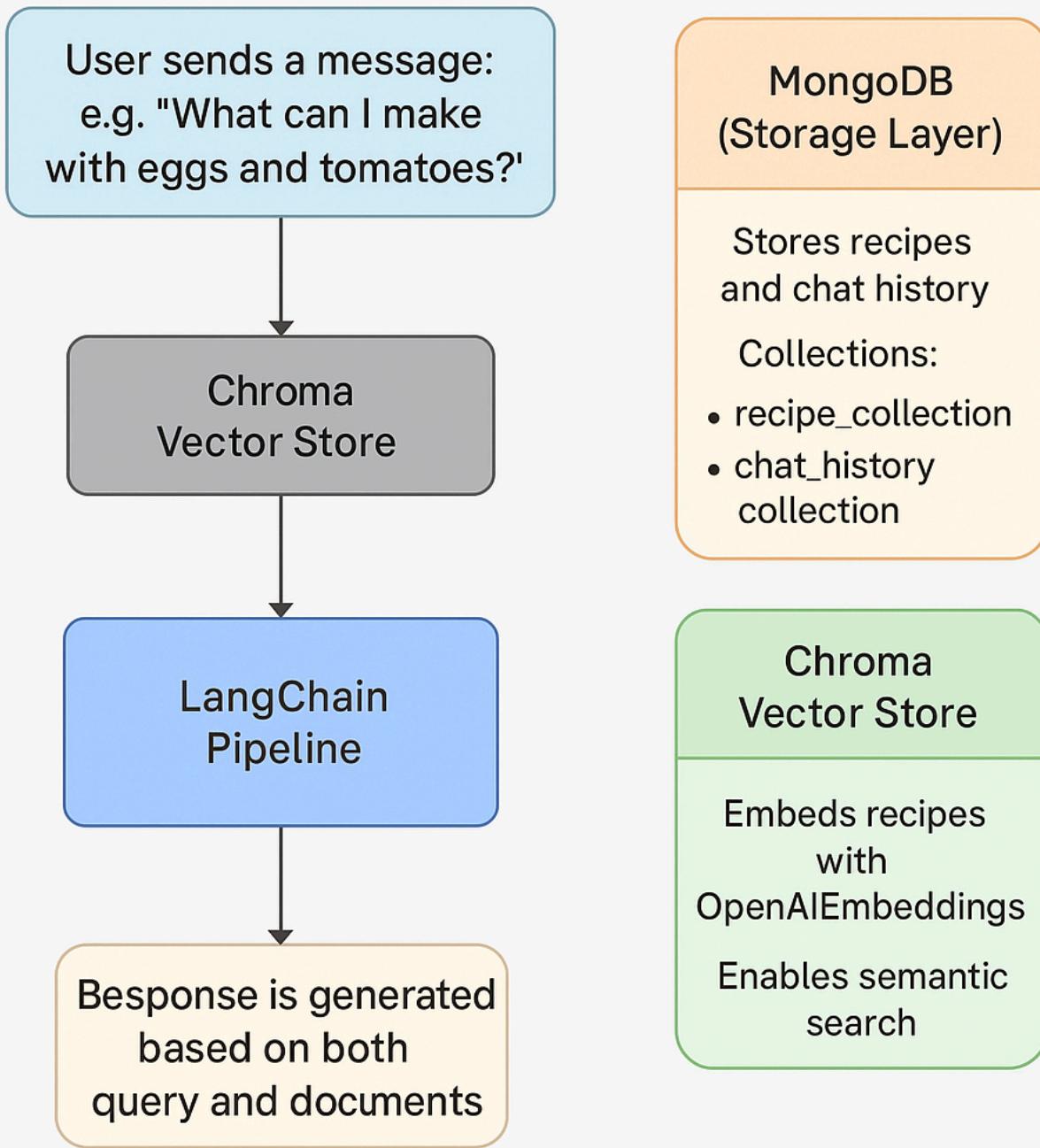
1. **Parse request** → extract `prompt`, `chat_id`.
 2. **Load `search.txt`** and inject Recipe schema JSON so ChatGPT knows your collection structure.
 3. **Assemble messages:**
 - System message (the search-template with schema)
 - Last 10 messages from Mongo (if `chat_id` given)
 - User message: Prepare the filter ...: `{prompt}`
 4. **Call OpenAI** (`model="o3-mini"`) to generate a MongoDB query object.
 5. **Parse** the response as JSON → e.g. `{ "meal_type": "vegetarian", "total_time_in_mins": { "$lte": 30 } }`
 6. **Execute** this filter against `recipe_collection`.
 - If zero hits, build an `$or`-style fallback query and retry.
 7. **Format results** into plain text snippets.
 8. **Prettify** via another OpenAI call (`gpt-4.1-nano`) to turn raw recipe docs into a chat-friendly UI block.
 9. **Upsert chat history** in Mongo:
 - Append user's prompt + assistant's prettified reply
 - Mark conversation `type: "gpt"`
 10. **Return** `{ result: <formatted string>, chat_id: <id> }`
-

4. `customize_recipe` Endpoint

1. **Parse** `prompt`, require `chat_id`.
2. **Fetch** the full conversation for that `chat_id` from Mongo.
3. **Load `customize.txt`**, assemble:
 - System message (customize template)
 - Entire conversation history
 - Current user message (`prompt`)
4. **Call OpenAI** (`gpt-4.1-nano`) to generate recipe edits/substitutions.
5. **Append** the new user & assistant messages to Mongo history.
6. **Return** `{ result: <customization text>, chat_id }`

RAG Workflow in RecipeGPT:

RAG Workflow in RecipeGPT



1. User sends a message: e.g., "What can I make with eggs and tomatoes?"
2. Retriever searches the Chroma vector store for similar recipes.
3. Relevant recipes are passed as context to the OpenAI GPT model.
4. Response is generated based on both query and documents.

This ensures accurate and recipe-specific answers — reducing hallucination and increasing user trust.

1. MongoDB (Storage Layer)

- Stores recipes and chat history
 - Collections:
 - recipe_collection
 - chat_history_collection

2. Chroma Vector Store

- Embeds recipes with OpenAIEmbeddings
- Enables semantic search

3. LangChain Pipeline

LangChain acts as the core orchestrator for intelligent, retrieval-based conversation in this project. Here's how it works:

- LLM Model: Uses ChatOpenAI (GPT-4.1 via OpenAI API) to generate responses.
- Memory Management: Uses MongoDBChatMessageHistory to persist and maintain multi-turn chat memory per user.
- Vector Retrieval: Integrates with ChromaDB as a retriever to find relevant recipe documents based on the current user input.
- Prompt Construction: Combines user input, retrieved documents, and past memory to build a rich prompt.
- Chain Execution: Uses LangChain's RetrievalQA or ConversationalRetrievalChain to deliver context-aware responses.

4. FastAPI Endpoints

The core chatbot logic is implemented in the `recipe.py` and `chat.py` module using FastAPI.

LangChain Components Initialization:

- ChatOpenAI: Connects to OpenAI GPT-4.1 for text generation.
- OpenAIEmbeddings: Used to create embeddings for recipe documents.
- Chroma: Acts as a vector database storing those embeddings for retrieval.
- MongoDBChatMessageHistory: Maintains a persistent chat history in MongoDB, enabling multi-turn conversations.
- Retriever Setup:
 - A retriever is built from the Chroma vector store using `vectorstore.as_retriever()`.

- LangChain's `create_history_aware_retriever()` is used to combine this retriever with past memory.
- Conversational Chain Creation:
 - LangChain's `create_retrieval_chain()` wires together the retriever and the LLM to construct a context-aware QA pipeline.
- FastAPI Route:
 - A POST endpoint `/langchain` receives user messages.
 - The message is passed to the LangChain chain.
 - The chain retrieves relevant recipe documents, injects past conversation memory, and invokes OpenAI to generate a reply.
 - The final response is returned to the frontend.

This modular pipeline enables accurate, context-aware, and grounded responses based on stored recipe data.

- `/chat`: CRUD operations on the `chat_history_collection`
- `/recipes`: Upload recipes and Processes user input and returns GPT response

Frontend

- Built using React
- Simple chat interface
- Displays interactive conversation history

Docker Setup

`docker-compose.yml` spins up:

- mongodb container
 - backend container (FastAPI + LangChain)
 - frontend container (React)
- MongoDB is accessible as `mongodb:27017` internally.

Workflow

1. Recipe Extraction
 - User submits a recipe URL.
 - Backend fetches the page, prompts OpenAI to extract structured recipe data (title, ingredients, instructions, nutrition, etc.) using a strict schema.
 - Data is validated and stored in MongoDB and vectorized in ChromaDB.
2. Recipe Search

- User enters a natural language query.
- Backend classifies the query (search/customize/other) and generates a MongoDB filter using OpenAI.
- Results are formatted and returned to the frontend.

3. Chat & Customization

- User chats with the AI assistant using one of two modes:
 - OpenAI GPT Mode: Direct interaction with GPT models for general recipe questions and customizations.
 - LangChain RAG Mode: Retrieval-augmented generation that uses your recipe database as context for more personalized responses.
- Chat history is used for context-aware responses and recipe customization.
- New recipes generated by AI can be stored for future use (Only available in the LangChain RAG Chat).

LangChain vs Direct ChatGPT Comparison

Aspect	LangChain RAG Chat (/langchain)	Direct ChatGPT API (e.g. /search, /customize, /classify)
Core Idea	Retrieval-Augmented Generation: ground answers in your own recipe corpus via a vector DB.	Plain GPT calls: send user prompt (and optional chat history) along with the recipe schema straight to OpenAI to generate a filter object. The returned object is used to fetch recipes from the Mongo collection.
Setup & Dependencies	<ul style="list-style-type: none"> • langchain, langchain_chroma, chromadb • Persistent vector store (ChromaDB) • Retrieval chains, prompts, embedding functions 	<ul style="list-style-type: none"> • openai client only • No vector DB, no embedding libraries
Data Flow	<ol style="list-style-type: none"> 1. Load/convert chat history → HumanMessage/AIMessage list 2. <pre>rag_chain.invoke({input, chat_history}):</pre> <ul style="list-style-type: none"> • Contextualize question • Retrieve top-k docs via 	<ol style="list-style-type: none"> 1. Read chat_id & history from Mongo 2. Build a system+user message array (optionally with a JSON schema for search/classify) 3. <pre>oai_client.responses.create(model, input=messages, ...) → raw JSON or text</pre>

Aspect	LangChain RAG Chat (/langchain)	Direct ChatGPT API (e.g. /search, /customize, /classify)
	<p>Chroma</p> <ul style="list-style-type: none"> • Stuff docs into QA prompt <p>3. Return concise, context-grounded answer</p> <p>4. Fallback: if RAG yields “I don’t know,” re-invoke GPT to extract/store a new recipe</p>	<p>4. Post-process (run Mongo query, prettify, update history)</p>
Personalization	Leverages your actual recipes via vector search → highly relevant, up-to-date suggestions	Relies on GPT’s pre-trained knowledge → generic advice, no grounding in user data
Latency	Extra retrieval & embedding steps → higher response time (hundreds of ms to seconds)	Single API call → faster overall
Cost	You pay for embeddings + multiple ChatOpenAI calls (contextualization + QA), plus vector DB storage	Only pay for direct ChatGPT calls
Complexity	More moving parts: vector store, retriever chains, multi-step prompts, fallback logic	Simpler: build messages → call OpenAI → parse response
Error Handling	<ul style="list-style-type: none"> • Must handle Chroma failures, empty retrievals, fallback extraction storage • More edge cases to test 	<ul style="list-style-type: none"> • Mostly JSON-parsing & HTTP errors • Fewer integration points

Conclusion

- Use LangChain RAG when you need accurate, personalized responses grounded in the user’s stored recipes—ideal for customization, contextual Q&A, chaining multi-turn conversations about a user’s own data.

- Use Direct ChatGPT API when you need a fast, simple interaction or general cooking advice that doesn't require grounding in the user's recipe database.

Testing on External Users

To assess the usability and effectiveness of the system, we conducted a controlled user study with five participants. Each user was granted access to the platform without any prior guidance, aside from a brief instruction to "*Think of this as a GPT tailored specifically for recipes.*" This setup allowed us to observe genuine, uninfluenced interactions and gather insights into the system's intuitiveness and reliability.

Users engaged with both available interaction modes: the **OpenAI GPT API integration** and the **Langchain-based conversational interface**. In addition to using the chat features, participants also tested the functionality for saving recipes sourced from external websites.

Participants submitted a diverse range of natural-language prompts, such as:

- *"Can you give me a quick 10-minute recipe with high protein and low fat? Make it gluten-free as well."*
- *"Can you give me a recipe with broccoli?"*
- *"Show me pasta recipes under 30 minutes."*
- *"Can you help me find a high-protein Indian dinner recipe?"*
- *"Can you give me the recipe for ciorbă rădăuțeană?"*

In most cases, the system performed effectively. Users were able to search for recipes, request customizations, and receive personalized suggestions aligned with their dietary needs or ingredient constraints. For instance, during an interaction in Langchain mode, one user asked to modify a traditional *ciorbă rădăuțeană* recipe to exclude eggs. The initial response suggested general egg alternatives, but upon further clarification, the system was able to offer substitutions specifically suited for that dish, demonstrating context awareness and adaptability.

However, users also reported some limitations. Although both interaction modes generally met expectations, there were occasional instances where neither mode was able to return a recipe, particularly for niche or highly constrained queries. These edge cases highlighted the need for further improvements in fallback mechanisms, dataset coverage, and prompt handling to ensure a more consistent user experience.

Overall, the feedback confirmed that the platform was intuitive and valuable, while also identifying opportunities to enhance its robustness in handling less common or overly specific requests.

Langchain chat mode -

The screenshot shows a chat interface with two messages from a user ('U') and three responses from an AI ('AI'). The AI messages provide information about egg substitutes, details on a specific recipe, and a simplified version of the recipe steps.

User Message: I want to replace eggs
10:32 PM

AI Response: You can replace eggs in recipes with several alternatives depending on the purpose they serve. Common substitutes include 1/4 cup unsweetened applesauce, 1/4 cup mashed banana, or 1 tablespoon ground flaxseed mixed with 2.5 tablespoons of water for each egg. Choose the substitute based on whether you're baking or cooking and the desired texture.
10:32 PM

User Message: What about substituting eggs in this recipe?
10:32 PM

AI Response: In the recipe for ciorbă rădăuțeană, the eggs are used to enrich the soup and create a creamy texture. You can substitute the 2 egg yolks with 1/2 cup of silken tofu blended until smooth or use 1/4 cup of heavy cream for a similar creaminess without using eggs. Another option is to use 1/4 cup of sour cream, though it may alter the flavor slightly.
10:32 PM

User Message: Can you simplify the steps for me?
10:32 PM

AI Response: Sure! Here's a simplified version of the ciorbă rădăuțeană recipe:

1. Boil beef bones in water for 1 hour to make a stock, then strain and return the liquid to the pot.
2. Add beef chunks and cook until tender, then add chopped vegetables (onion, carrots, parsnip, celery root, potatoes) and cook until everything is tender.
3. Mix sour cream, egg yolks (or substitute), garlic, and vinegar in a bowl, temper with hot broth, then stir back into the pot. Season with salt and pepper, and garnish with fresh dill before serving.

10:32 PM

Comparable equivalent chat using the OpenAI GPT Integration.

I want to replace eggs.
11:14 PM

Certainly! To replace the eggs in ciorbă rădăuțeană (which are primarily used for thickening and adding richness via the sour cream and egg yolks), you can use a few alternatives depending on the texture and flavor you want:

Egg Substitutes:

1. **Cornstarch or Flour Slurry:**
 - Mix 1 tablespoon of cornstarch or all-purpose flour with 2-3 tablespoons of cold water until smooth.
 - Stir into the hot soup toward the end of cooking. This will thicken the soup similarly to the egg yolks.
2. **Cashew Cream (Dairy-Free):**
 - Blend 1/4 cup of soaked cashews with a little water until smooth.
 - Add this to the soup for creaminess.
3. **Silken Tofu:**
 - Blend 1/4 cup of silken tofu until smooth.
 - Mix into the soup for creaminess without eggs.
4. **Additional Sour Cream or Greek Yogurt:**
 - If avoiding eggs but wanting similar richness, increase the amount of sour cream or substitute with Greek yogurt added at the end.

How to proceed:

- If you want to replicate the slight richness and tang, I recommend the cornstarch slurry combined with the sour cream, or just adding extra sour cream/yogurt without eggs.
- Add the thickening agent near the end of cooking, stirring well and making sure it's heated through but not boiled vigorously.

Would you like a updated step-by-step for the recipe without eggs?
11:14 PM

Could you simplify the steps for me?
11:15 PM

Of course! Here's a simplified version of the Ciorbă Rădăuțeană recipe without eggs, using cornstarch as a thickening alternative:

The users reported that while both the modes mostly met their expectations, there were cases where there were cases where both the modes were not able to find the recipes.

Project Evolution

Throughout the development lifecycle, one of the core challenges we continually addressed was refining the prompts used for interacting with OpenAI's GPT APIs and effectively handling the variety of responses returned. A common issue arose when users queried specific aspects such as *preparation time*. The model would often return a broader set of results—for instance, all recipes where `prep_time` was present in the database—rather than filtering by the user's stated preference. This behavior was misaligned with user expectations.

Upon investigation, we identified two main causes: (1) our prompts were too permissive, leading the model to generate overly complex aggregation pipeline queries, and (2) there was insufficient precision in how the prompt conveyed our intent. To resolve this, we iteratively refined the prompt structure to make the expected output more deterministic.

Prompt Samples (Early Versions):

Prompt Sample 1:

Input

System

Here's a concise yet clear prompt that you can use to generate a MongoDB aggregation pipeline query with OpenAI's API: Given the following MongoDB schema context for recipes: {properties: {title: {title: 'Title', 'type': 'string'}, 'ingredients': {items: {type: 'string'}, 'title: 'Ingredients', 'type: 'array'}, 'instructions: {items: {type: 'string'}, 'title: 'Instructions', 'type: 'array'}, 'cuisine: {anyOf: [{type: 'string'}, {type: 'null'}], 'default: None, 'title: 'Cuisine}, 'meal_type: {anyOf: [{type: 'string'}, {type: 'null'}], 'default: None, 'title: 'Meal Type}, 'prep_time: {anyOf: [{type: 'string'}, {type: 'null'}], 'default: None, 'title: 'Prep Time}, 'cook_time: {anyOf: [{type: 'string'}, {type: 'null'}], 'default: None, 'title: 'Cook Time}, 'tags: {anyOf: [{items: {type: 'string'}, 'type: 'array'}, {type: 'null'}], 'default: [], 'title: 'Tags}, 'source_url: {anyOf: [{type: 'string'}, {type: 'null'}], 'default: None, 'title: 'Source Url}}, 'required: [title, ingredients, instructions], 'title: 'RecipeCreate', 'type: 'object' Generate a MongoDB aggregation pipeline to search recipes in the collection based on the user's natural language input query.

- Your output must strictly be a valid JSON array of pipeline stages without the `db.collection.aggregate` prefix.
- The pipeline should effectively interpret the user's intent, searching relevant fields like `title`, `ingredients`, `cuisine`, `meal_type`, and `tags`.
- Use stages like `$match`, `$search`, `$project`, or `$limit` as needed to accurately fulfill the user's query intent. User input query example: "Find quick vegetarian recipes under 30 minutes" Your output should strictly follow this format example: `[{"$match": {"$text": {"$search": "quick vegetarian recipes"}}, {"$limit": 10}]` Do not include explanatory text, just the JSON array for the pipeline stages.

User

Write a mongo query which finds the recipe based on the prompt: Help me find an Indian recipe.. If the user query prompt doesn't feel related to related to recipes, return empty list.

Output

Assistant

```
{  
  "query": "[{\\"$match\": {\\\"cuisine\\\": \\"Indian\\\"}}, {\\\"$limit\\\": 10}]"
```

Prompt Sample 2:

Input

System

Given the following MongoDB schema context for recipes: {properties: {title: {title: 'Title', 'type: 'string'}, 'ingredients: {items: {type: 'string'}, 'title: 'Ingredients', 'type: 'array'}, 'instructions: {items: {type: 'string'}, 'title: 'Instructions', 'type: 'array'}, 'cuisine: {anyOf: [{type: 'string'}, {type: 'null'}], 'default: None, 'title: 'Cuisine}, 'meal_type: {anyOf: [{type: 'string'}, {type: 'null'}], 'default: None, 'title: 'Meal Type}, 'prep_time: {anyOf: [{type: 'string'}, {type: 'null'}], 'default: None, 'title: 'Prep Time}, 'cook_time: {anyOf: [{type: 'string'}, {type: 'null'}], 'default: None, 'title: 'Cook Time}, 'tags: {anyOf: [{items: {type: 'string'}, 'type: 'array'}, {type: 'null'}], 'default: [], 'title: 'Tags}, 'source_url: {anyOf: [{type: 'string'}, {type: 'null'}], 'default: None, 'title: 'Source Url}}, 'required: [title, ingredients, instructions], 'title: 'RecipeCreate', 'type: 'object'.

Generate a dictionary containing key value pairs of the components which can be used in MongoDB search query to search recipes in the collection based on the user's natural language input query.

Return ONLY the dictionary in valid JSON format without any formatting, explanations or comments.

User

What is meal prep time for this recipe?

Output

Assistant

```
{  
  "prep_time": {  
    "$exists": true  
  }  
}
```

We eventually adopted a revised prompt that explicitly asked GPT to return only the filter

conditions, not the full query. This shift allowed us to handle query construction programmatically, giving us greater control and consistency. The response was now a dictionary containing filtering parameters, which we could reliably convert into MongoDB queries on the backend.

Updated Prompt (Final Version):

Current Prompt:

```
'title': 'Meal Type', 'prep_time_in_mins': {'anyOf': [{type: 'integer'}, {type: 'null'}], 'default': None, 'title': 'Prep Time In Mins'}, 'cook_time_in_mins': {'anyOf': [{type: 'integer'}, {type: 'null'}], 'default': None, 'title': 'Cook Time In Mins'}, 'total_time_in_mins': {'anyOf': [{type: 'integer'}, {type: 'null'}], 'default': None, 'title': 'Total Time In Mins'}, 'tags': {'anyOf': [{items: {type: 'string'}, type: 'array'}, {type: 'null'}], 'default': [], 'title': 'Tags'}, 'source_url': {'anyOf': [{type: 'string'}, {type: 'null'}], 'default': None, 'title': 'Source Url'}, 'estimated_calories': {'anyOf': [{type: 'number'}, {type: 'null'}], 'default': None, 'title': 'Estimated Calories'}, 'protein_grams': {'anyOf': [{type: 'number'}, {type: 'null'}], 'default': None, 'title': 'Protein Grams'}, 'fat_grams': {'anyOf': [{type: 'number'}, {type: 'null'}], 'default': None, 'title': 'Fat Grams'}, 'nutrients_present': {'anyOf': [{items: {type: 'string'}, type: 'array'}, {type: 'null'}], 'default': [], 'title': 'Nutrients Present'}, 'image_url': {'anyOf': [{type: 'string'}, {type: 'null'}], 'default': None, 'title': 'Image Url'}, 'required': ['title', 'ingredients', 'instructions'], 'title': 'RecipeCreate', 'type': 'object'}
```

Instructions:

- Output ONLY a dictionary which contains key and value pairs suitable for MongoDB filtering.
- Use MongoDB operators like \$lte, \$gte, \$in, \$regex as appropriate.
- Ignore the tags attribute.
- Use reasonable fixed thresholds for vague terms:
 - "high protein": protein_grams >= 20
 - "low fat": fat_grams <= 10
 - "low calorie": calories <= 300
 - "quick": total_time_in_mins <= 15
 - "short prep time": prep_time_in_mins <= 10
 - "long cook time": cook_time_in_mins >= 60
- If the query contains ingredient or cuisine names, match them using:
 - {"ingredients": {"\$regex": "ingredient_name", "\$options": "i"}}
 - {"cuisine": {"\$regex": "cuisine_name", "\$options": "i"}}
- Do not include any explanation, markdown, comments, or extra formatting.
- Respond ONLY with the JSON object representing the filter.
- The output must be valid JSON.
- If you're not able to formulate a query, return empty object.

Examples:

User query: "Show me high protein chicken recipes"

Output: {"protein_grams": {"\$gte": 20}, "ingredients": {"\$regex": "chicken"}}

User query: "Quick low fat Italian dishes"

Output: {"total_time_in_mins": {"\$lte": 15}, "fat_grams": {"\$lte": 10}, "cuisine": {"\$regex": "Italian", "\$options": "i"}}

Collapse ^

In parallel, we experimented with LangChain's conversational interface as a comparative benchmark. Its interaction quality aligned more closely with our envisioned user experience.

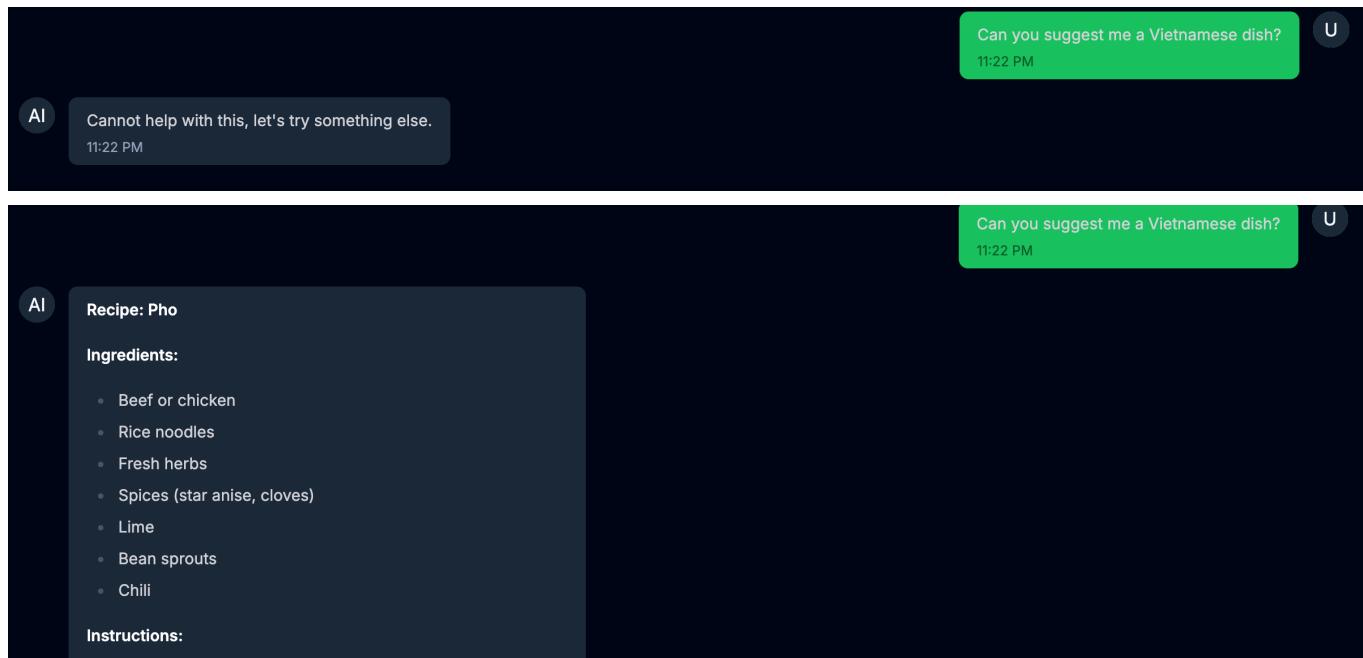
This motivated continued improvements in our GPT-based interaction mode to reach similar conversational fluency and reliability.

Initially, the system leveraged the [GPT-4o](#) model for generating database queries and recipe customizations. However, during development, OpenAI released [GPT-4.1](#), which offered improved response quality at reduced cost. We promptly integrated GPT-4.1 into our pipeline and observed noticeable gains, particularly in response clarity, consistency, and overall performance efficiency.

User Feedbacks

1. Fallback Recipe Generation

To enhance the user experience when a search query yielded no results, we introduced a fallback mechanism. If a matching recipe was not found in the database, the system would attempt to generate a new recipe using external sources, which was then stored locally for future queries. This ensured that users consistently received meaningful responses and also helped gradually enrich our recipe dataset.



2. Nutritional Filters and Metadata

A user requested: "*Can you give me a quick 10-minute recipe with high protein and low fat?*" Initially, our system returned no results, largely because nutritional information was not tracked in the recipe schema. In response, we expanded our data model to include fields such as `estimated_calories`, `protein_grams`, `fat_grams`, and `nutrients_present`. This allowed users to query based on dietary requirements and improved the system's usefulness for health-conscious users.

We do note, however, that the nutritional values are estimated and may sometimes include

inaccuracies due to GPT-generated content. Users are advised to cross-check these details with trusted nutritional sources when necessary.

```
AI classification: {'type': 'search', 'confidence': 0.98, 'reasoning': 'The query requests a recipe with specific criteria (quick 10 min, high protein, low fat) which matches the SEARCH category.'}
Classification result: search
OpenAI response: {"total_time_in_mins": {"$lte": 10}, "protein_grams": {"$gte": 20}, "fat_grams": {"$lte": 10}}
```

3. Relaxed Filter Matching

Another useful suggestion we received was to avoid presenting a "no results found" message when strict filtering criteria were not met. Instead, users preferred to see recipes that matched at least some of the provided parameters. We addressed this by implementing a secondary query strategy: if the initial strict filter failed, the system would rerun the search with a relaxed `$or` condition to surface partially matching results. This provided users with alternative suggestions while maintaining transparency about the level of match.

```
AI classification: {'type': 'search', 'confidence': 0.98, 'reasoning': 'The query requests a recipe with specific criteria (quick 10 min, high protein, low fat) which matches the SEARCH category.'}
Classification result: search
OpenAI response: {"total_time_in_mins": {"$lte": 10}, "protein_grams": {"$gte": 20}, "fat_grams": {"$lte": 10}}
Response for MongoDB query: {'total_time_in_mins': {'$lte': 10}, 'protein_grams': {'$gte': 20}, 'fat_grams': {'$lte': 10}}
OR query: {'$or': [{{'total_time_in_mins': {'$lte': 10}}, {'protein_grams': {'$gte': 20}}, {'fat_grams': {'$lte': 10}}]}}
```

Potential Improvements

1. User Profiles & Dietary Preferences

- Accounts with saved preferences (allergies, calorie targets, cuisines)
- Auto-filter & personalize all searches/chats to their profile

2. Meal Planning & Grocery Lists

- Calendar/weekly meal-planner UI to assemble multiple recipes
- One-click Generate grocery list that consolidates ingredients and lets you check off items.

3. Ingredient Substitution & Nutrition API Integration

- Plug into external nutrition/food-db APIs (Nutritionix, USDA) for richer macro/micro data.
- Dynamic substitution engine: suggest vegan, keto, low-sodium swaps with nutritional trade-offs.

Lessons Learned

1. Prompt Engineering is Critical for Reliability

One of the most impactful lessons was the importance of precise and well-scoped prompt design. Initially, broad prompts led to inconsistent or overly complex GPT responses (e.g., aggregation pipelines or wide-ranging filters). By constraining the model to return only specific filter objects and handling query construction programmatically, we achieved far more consistent and deterministic outputs.

2. Fallback Logic Enhances User Experience

Users preferred to receive partially matching results rather than being shown a “no recipes found” message. Introducing a secondary search using relaxed filters (via `$or` conditions) helped maintain continuity and engagement, even when exact matches weren't found.

3. User Testing Surfaces Unanticipated Needs

Testing with actual users—without guided onboarding—provided authentic feedback. It exposed gaps such as missing nutritional data (e.g., protein or fat content) that were critical for personalized recipe queries. This emphasized the need for a more robust and nutrition-aware schema.

4. No One Model Fits All Use Cases

While both the OpenAI GPT API and Langchain-based chat modes were generally effective, neither was universally superior. Each had unique strengths depending on the interaction context. For example, Langchain excelled in sustained multi-turn conversations, whereas GPT API integration offered quicker responses for straightforward queries.

Contributions

Avtans Kumar

- **Prompt Engineering & Model Behavior Tuning:** I iteratively refined the prompts used for classification and filtering.
- **Search & Customization Workflows:** Designed and implemented three main flows —`/classify`, `/search`, and `/customize`—each backed by OpenAI's GPT API and designed for recipe-specific conversational behavior.
- **Langchain Integration:** Integrated a parallel chat mode using Langchain, allowing users to experience more conversational and context-rich interactions.
- **User Feedback Integration:** Addressed user feedbacks by extending the database with fields like `protein_grams`, `fat_grams`, and `estimated_calories`.

Self Scoring Table

Criteria	Points	Justification
Significant exploration	80	Developed end-to-end GPT and Langchain flows, fallback logic, and prompt engineering strategies
Innovation or Creativity	10	Introduced structured filter-object pattern and GPT-driven recipe regeneration fallback.

Criteria	Points	Justification
Highlighted complexity	10	Managed schema evolution, error analysis, UI development, and multi-modal integration complexities.
Lessons learned	10	Detailed analysis of GPT limitations, schema iteration, and prompt strategies and potential improvements.
Repo quality & visualization	10	Well-structured with proper README and .env guide
Testing on external users (5 ppl)	10	Received and incorporated feedback.
Earned money	0	Not monetized yet

Resources

- **OpenAI GPT APIs**
 - [OpenAI API Documentation](#)
 - [GPT-4.1 Release Notes](#)
 - [GPT-4o Introduction](#)
- **Langchain**
 - Langchain Documentation
 - Langchain OpenAI Integration Guide
- **MongoDB:** [MongoDB Query Language Reference](#)
- **Prompt Engineering**
 - OpenAI Cookbook: [Prompt Engineering Guide](#)
- **React Documentation:** <https://react.dev/>