 云南大学软件学院期末课程报告

Final project report

School of Software, Yunnan University

**个人成绩**

|  |  |  |  |
| --- | --- | --- | --- |
| **序号** | **学号** | **姓名** | **成绩** |
|  | **20233120074** | **Ashiliny Langa** |  |

学 期:　2025春季学期

课程名称:　Professional Trainining

任课教师:　Ahmed Zahir

报告:Recipe Analyzer

姓名: Ashiliny Langa

联系电:18314574537

电子邮件:[lashiliny14@gmail.com](mailto:lashiliny14@gmail.com)

作业截止时间：2025年12月 11日

# **Abstract**

I built a Recipe Fitness Analyzer a smart tool that helps people figure out whether their meals fit their fitness goals, like losing weight or gaining muscle. Unlike basic apps that just match keywords, this system understands recipes by using a machine learning model called Sentence-BERT, which turns cooking instructions into meaningful number patterns. It then checks if a recipe aligns with your goal using a classifier and suggests similar healthy recipes through a fast vector search in ChromaDB.The whole system runs in Docker containers a clean and portable setup with a simple web interface, a backend that does the AI think, and a dedicated recipe memory database. I faced real-world hurdles like making everything work on a regular laptop without special hardware and getting different AI libraries to talk to each other. After testing it with various recipes, the tool shows it can give useful, personalized food advice helping users eat better without needing a nutrition degree.

## **1. Introduction**

## **1.1 Motivation for the NLP Task**

In our days the world is into eating clean and being fit but the struggle in knowing what can keep them fit or how can they eat healthy calories and for now there are no existing recipe platforms that rely on basic keyword matching or that o understands recipe context, ingredient relationships, and nutritional implications. Traditional systems treat recipes as mere collections of ingredients rather than cohesive culinary compositions with specific health impacts. This project addresses these limitations by developing a Recipe Fitness Analyzer that employs advanced natural language processing (NLP) to understand recipes conceptually, assess their compatibility with fitness goals (weight loss or muscle gain), and provide semantically relevant recommendations. The motivation came from the need to bridge culinary knowledge with nutritional science through computational intelligence, enabling users to make informed dietary decisions aligned with their fitness objectives without requiring nutritional expertise which will make peoples life easier because sometimes you have your own recipe but you are not sure if it could fit in your goal so this app will help in that .

### **1.2 ML and Deep Learning Background**

Over the last few years, machine learning, especially deep learning, has really improved how computers can work with and understand human language. Older methods, which used fixed rules or basic statistics, often had trouble with language's many meanings, uncertainties, and how words change meaning depending on context. Newer approaches using deep neural networks, particularly ones built as transformers, allow computers to process text in a much smarter, more layered way. These models read text through multiple steps, each step building a better understanding than the last, catching patterns and connections that simpler systems would miss.

In my project, I applied deep learning ideas. I used neural networks to turn recipe descriptions like ingredients and instructions into lists of numbers called embeddings. These numbers aren't random; they represent the meaning and context of the recipe. This lets the system do much more than just search for matching words. It can understand ideas, compare recipes based on what they're about, and give useful feedback about whether a recipe fits someone's fitness goals. This deeper understanding is what makes the system helpful and accurate, going far beyond basic keyword searches

**1.3 Why Transformers / LLMs / Vector Search Matter**

Transformers, large language model (LLM) frameworks, and vector search technology form the essential foundation of my project, each one of them addressing critical limitations of traditional text processing methods and enabling the intelligent recipe analysis system to function effectively.

**Transformers** matter because they provide deep language understanding necessary to interpret recipes beyond surface level keywords. Where older natural language processing methods struggled with context, ambiguity, and subtle meaning, transformer models like DistilBERT which is used for sentiment analysis and DeBERTa used for text classification can analyze recipe descriptions to determine emotional tone and categorize nutritional content. More importantly, Sentence-BERT transforms entire recipe texts into dense vector embeddings numerical representations that capture semantic meaning. Which allows the system to understand that "grilled chicken with vegetables" conceptually relates to "baked fish with greens" even when they share a few exact words, enabling true semantic understanding rather than a simple pattern of matching.

**LLM frameworks** like LangChain matter because they provide the architectural structure to deal with multiple machine learning components into a coherent pipeline. In this project, LangChain enables the system to chain together different operations: first embedding generation, then semantic search, followed by sentiment analysis and classification, and finally generating helpful recommendations. This framework approach ensures that the various transformer models work together systematically, producing consistent, structured outputs that are understandable to users. Without it, managing these different ML components would be significantly more complex and error-prone.

**Vector search** fundamentally changes how recipes are compared and retrieved. Traditional database searches rely on exact keyword matches or simple text similarity, which fails when users describe recipes differently or use varied terminology. By storing recipe embeddings in ChromaDB ,with a dedicated vector database, the system can perform similarity searches based on conceptual meaning.

Together, these three technologies create a system that understands, organizes, and retrieves recipe information in ways that mirror human comprehension. Transformers provide the understanding, LLM frameworks provide the organization, and vector search provides the intelligent retrieval—each addressing a fundamental challenge in building practical, useful applications that work with natural language in meaningful ways. This combination enables the Recipe Fitness Analyzer to offer genuinely helpful guidance about whether specific recipes align with fitness goals, moving far beyond the limitations of keyword-based systems toward true culinary intelligence.

**2. Literature Review**

**2.1 Semantic Search System**  
Semantic search is about finding things based on meaning, not just matching keywords. Early search systems would look for exact words—if you searched for a "chicken recipe," you'd only get pages with those exact words. This didn't work well because people describe things differently. Over time, researchers have created better methods. First came systems that looked at word patterns, then came word embeddings like Word2Vec and GloVe that turned words into numbers that captured some meaning. The big breakthrough came with sentence embeddings like Sentence-BERT, which can understand whole sentences and find things that mean the same thing even if they use different words. For my recipe project, this means when someone types "healthy dinner with protein," the system can find recipes about grilled chicken or baked fish—not just recipes that happen to contain those exact words.

This evolution from simple keyword matching to semantic understanding is crucial for recipe analysis because people describe food in so many different ways. Someone might write "lean protein with greens" while another says "chicken salad," but both mean similar things nutritionally. Without semantic search, these connections would be missed, and users wouldn't get helpful recommendations. My system uses these semantic search principles to understand recipe concepts and match them to fitness goals, making it much more useful than traditional recipe websites that just match ingredient lists**.**

**2.2 Transformer Architectures**  
Transformers changed how computers understand language. Before transformers, systems used models that read text word by word, which was slow and often missed connections between words far apart in a sentence. The transformer, introduced in 2017, reads all words at once and uses something called "attention" to figure out which words are important in relation to others. This is like how we read—we don't just see individual words; we understand how they connect. From this came models like BERT, which learns from text in both directions (left-to-right and right-to-left), and smaller versions like DistilBERT that work almost as well but faster. In my project, I use DistilBERT to understand the tone of recipe descriptions and DeBERTa to classify what type of recipe it is. These transformer models give my system the ability to understand recipe content rather than just process words

What makes transformers particularly useful for my project is their ability to capture context. In a recipe description, words like "fried" versus "grilled" completely change the nutritional value, even if the ingredients are similar. Transformers notice these distinctions because they look at how all words relate to each other. Without this contextual understanding, my system might think "fried chicken" and "grilled chicken" are equally healthy, which would give users bad fitness advice. The transformer architecture ensures the system pays attention to cooking methods, ingredient combinations, and descriptive language—all the details that matter for accurate fitness analysis.

**2.3 Vector Databases**

When you turn text into numbers (embeddings), you need a place to store and search through them efficiently. Regular databases are good for exact matches but terrible for finding similar items. That's where vector databases come in. They're designed specifically for this kind of "similarity search." Early tools like FAISS made searching fast but didn't handle things like data management or filtering. Modern vector databases like ChromaDB (which I use), Pinecone, and Weaviate combine fast search with database features. They use smart algorithms to quickly find the closest matches even among thousands of recipes. For my project, ChromaDB stores all the recipe embeddings and lets me search for similar recipes in milliseconds. It also lets me filter by fitness goal—like only showing recipes good for weight loss—while still finding semantically similar ones.

The real advantage of using a vector database in my project is the combination of speed and intelligence. Without ChromaDB, I'd have to compare every new recipe against all stored recipes by calculating distances manually, which would be extremely slow as the database grows. ChromaDB uses optimized algorithms that make these comparisons lightning-fast. Plus, it lets me add metadata like calorie counts and fitness goals to each recipe embedding, so I can search for "recipes similar to this one that are also low-calorie." This dual capability—semantic similarity plus metadata filtering—is exactly what I need to give personalized fitness recommendations efficiently, and it's why vector databases are essential for modern ML applications like mine.

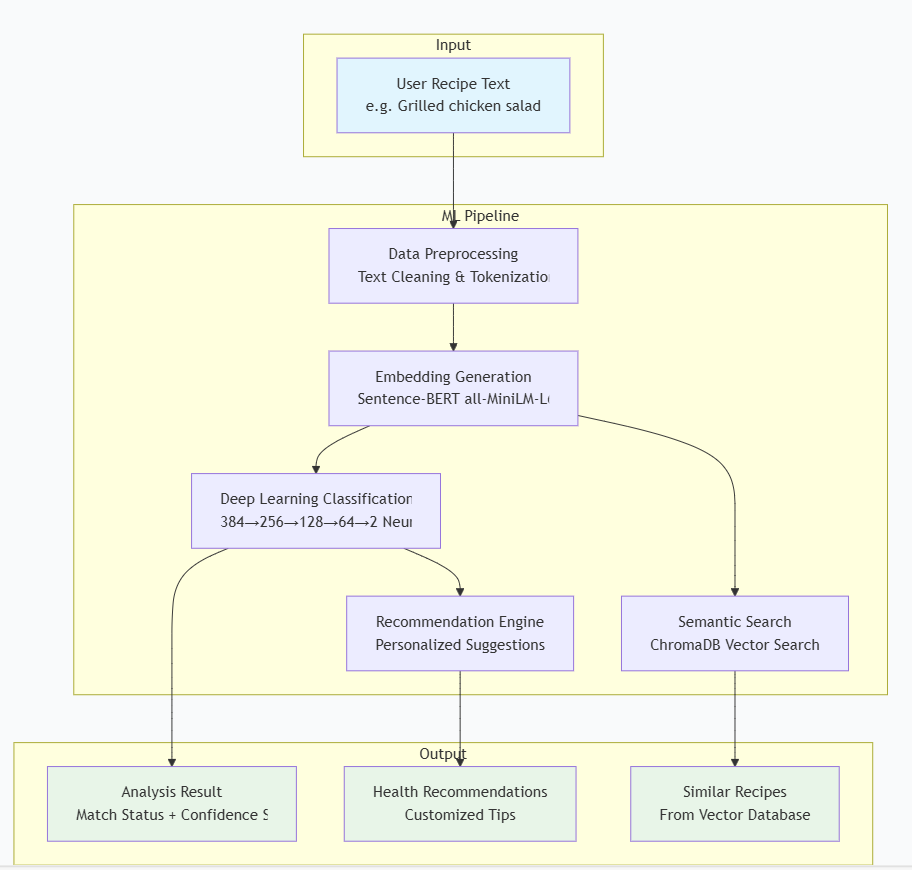
**2.4 LLM Frameworks**

As language models got more powerful, developers needed tools to work with them more easily. Hugging Face created Transformers, a library that gives you access to thousands of pre-trained models with a simple, consistent interface. This meant I could use state-of-the-art models like Sentence-BERT and DistilBERT without having to train them from scratch or write complex code to load them. Then came frameworks like LangChain, which helps build applications that chain multiple steps together. Instead of just calling one model, you can create pipelines where text goes through embedding, then search, then analysis, then recommendation generation—all organized neatly. In my project, I use both: Hugging Face for the models themselves and LangChain to organize how they work together to analyze recipes step by step.

Using these frameworks dramatically simplified my development process. Without Hugging Face, I'd have to download each model separately, figure out how to load it, handle different input formats, and manage compatibility issues—tasks that would take weeks instead of days. Without LangChain, my code would become a messy collection of disconnected function calls that are hard to maintain and extend. These frameworks provide structure and standardization, allowing me to focus on what makes my application unique rather than reinventing basic infrastructure. They also make my system more future-proof; if better models come out, I can swap them in easily without rewriting everything, which is important for keeping the application up-to-date with advancing AI technology.

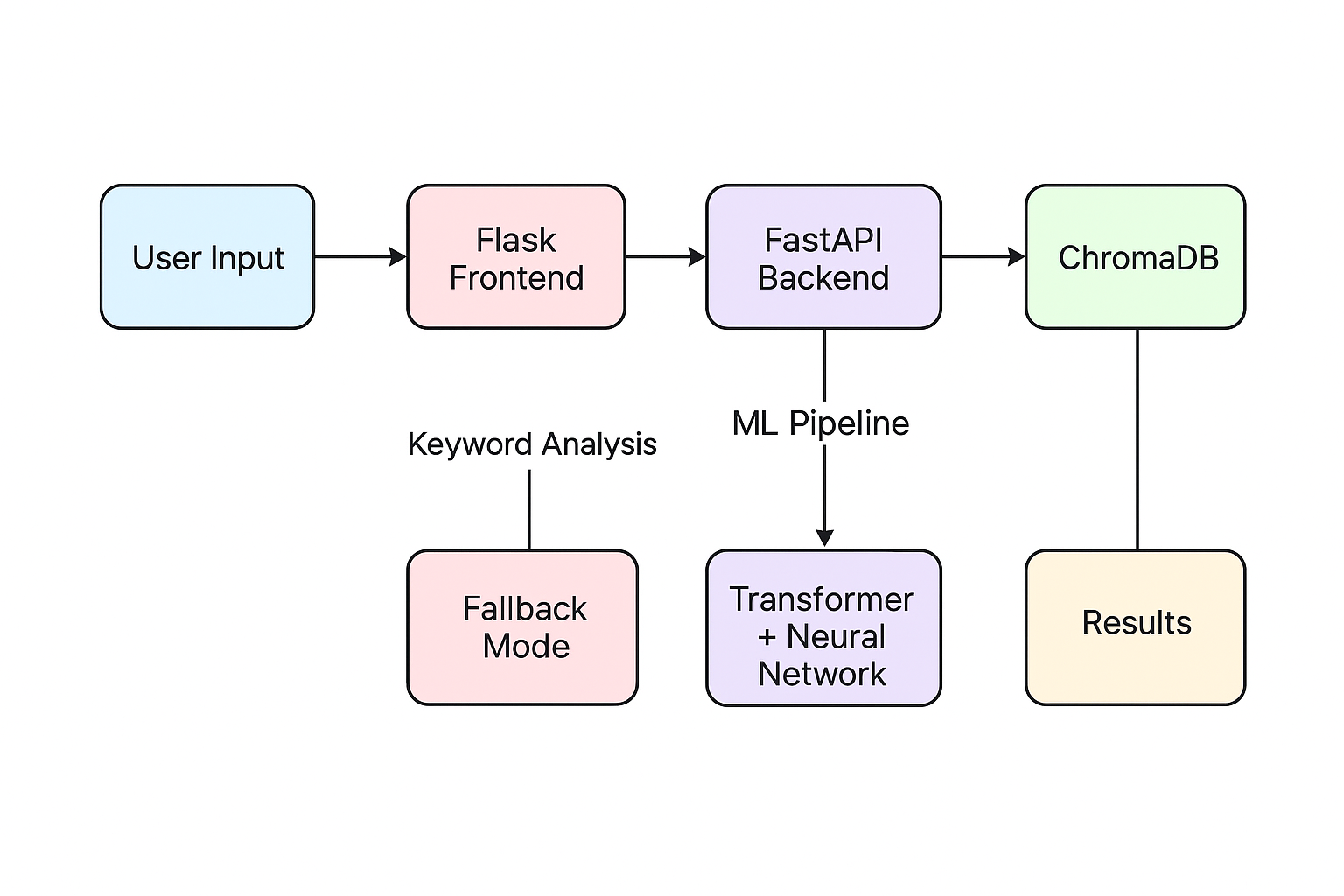
**3.System Design**

**3.1 ML pipeline:**



The system design centers on a complete ML pipeline that transforms recipe analysis through containerized services. The architecture runs in three interconnected Docker containers: a Gradio frontend for user interaction, a FastAPI backend hosting the core AI models, and a ChromaDB vector database for semantic storage. When a user submits a recipe and fitness goal, the text first goes through an NLP preprocessing module that cleans, tokenizes, and normalizes the input. This cleaned text is then converted into a 384-dimensional semantic embedding using the Sentence-BERT transformer model. These embeddings serve two purposes simultaneously: they feed into a deep neural network classifier that determines whether the recipe aligns with the specified fitness goal (outputting MATCH or MISMATCH), while also enabling semantic search in the vector database to find similar healthy recipes. The vector database stores recipes as embeddings alongside metadata, allowing for fast similarity comparisons that understand meaning rather than just keywords. This modular, containerized approach ensures the system is portable and scalable, with clear separation between the user interface, AI processing, and data storage layers.

**3.2 Data preprocessing workflow**



This diagram illustrates the system architecture of the Recipe Fitness Analyzer, depicting the sequential flow of data and the dual processing pathways available.

The workflow begins with User Input, where the recipe and fitness goals are submitted. This input is first received by the Flask Frontend, which serves as the web interface. A notable feature within this component is the Keyword Analysis module, which provides a Fallback Mode.

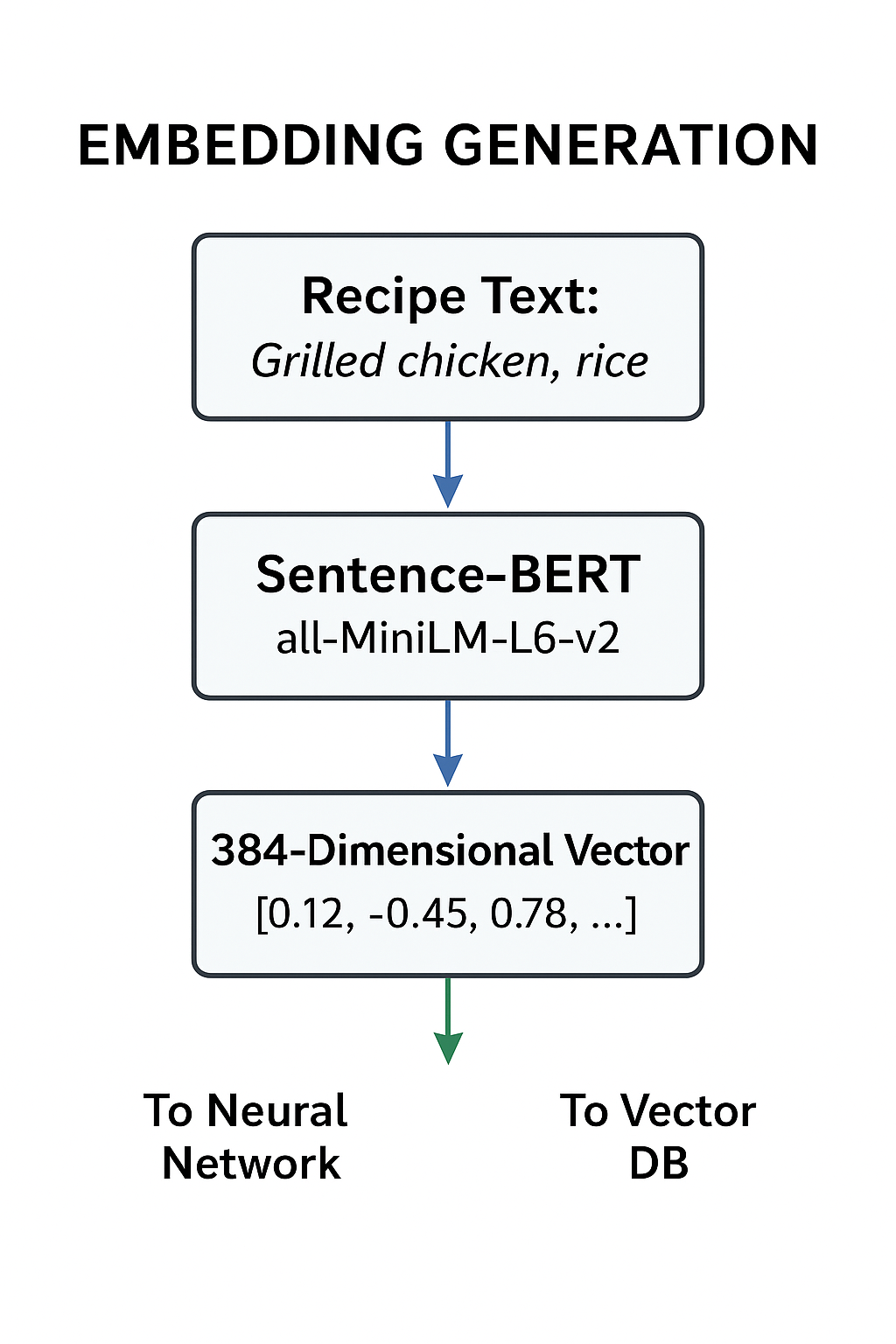
This ensures basic functionality is maintained in scenarios where the primary machine learning services are unavailable, utilizing rule-based logic to evaluate the recipe.

From the frontend, requests are routed to the FastAPI Backend, the core computational service. Here, the request enters the ML Pipeline, which comprises two principal sub-components: a Transformer model for semantic understanding and a Neural Network classifier for goal-specific prediction. This pipeline performs advanced machine learning analysis.

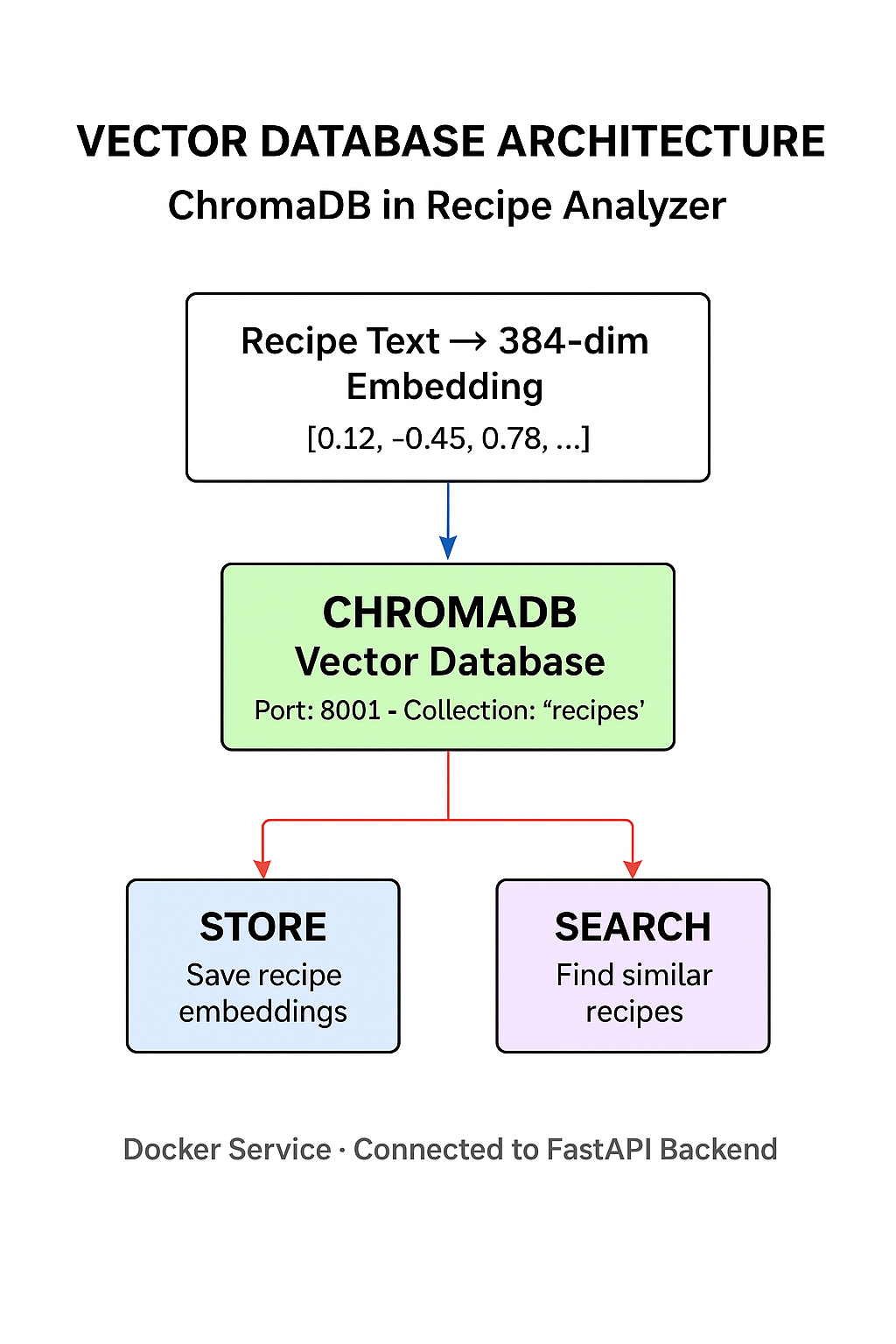
Simultaneously, the backend interacts with ChromaDB, the dedicated vector database. This component supports the system by enabling semantic search and retrieval of similar recipes based on stored embeddings.

The processing from both the ML pipeline and the database query converges to generate the Results. This output includes a clear classification, confidence score, and actionable recommendations for the user.

**3.3 Embedding generation**

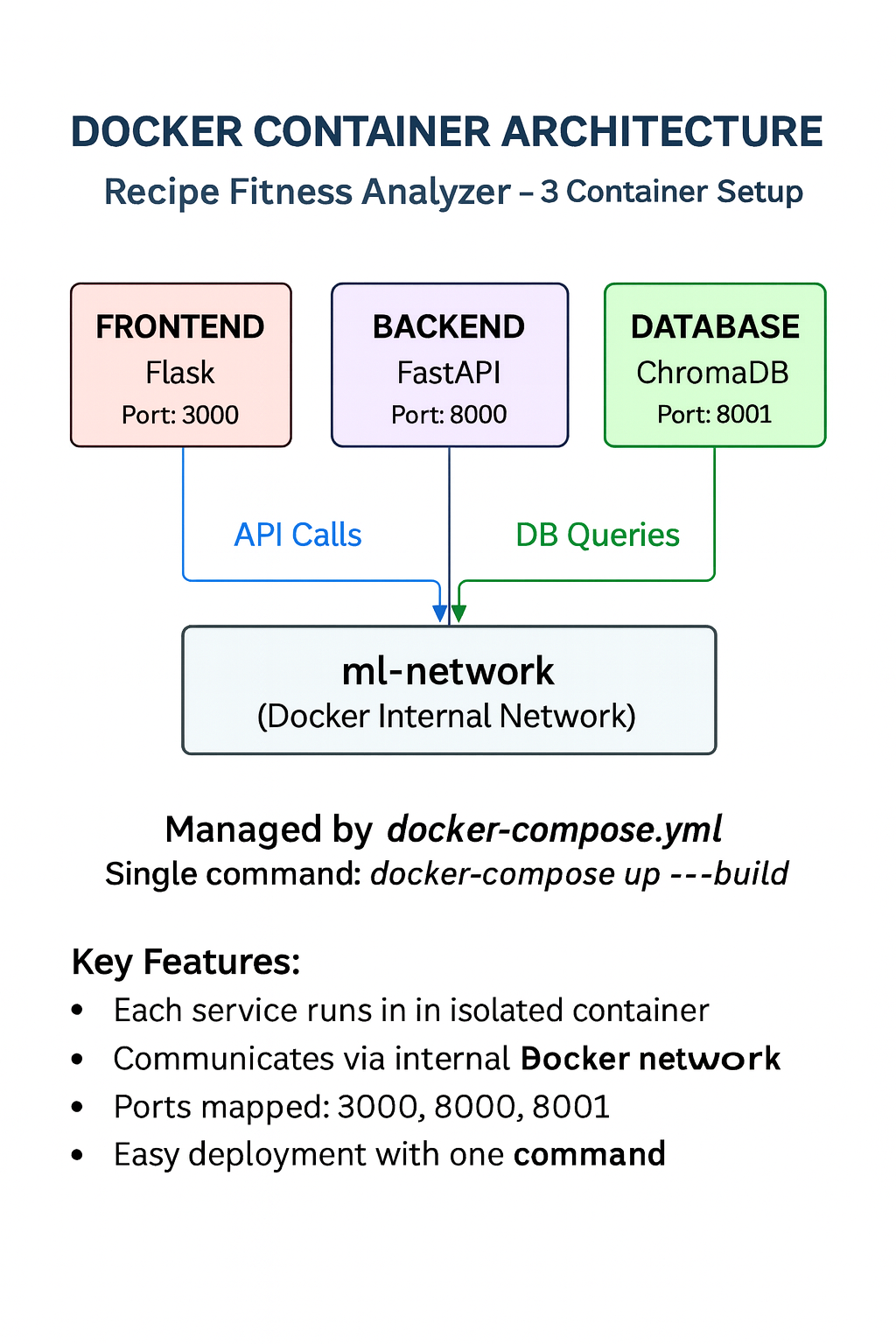


The diagram shows the embedding generation process that converts recipe text into numerical data. First, a user enters a recipe, such as grilled chicken and rice.This text is processed by a Sentence-BERT transformer model (specifically all-MiniLM-L6-v2), which understands the meaning behind the words and transforms the recipe into a list of 384 numbers called an embedding vector. This numerical representation captures the semantic features of the recipe—like ingredients, cooking style, and nutrition. The 384-dimensional output is then used in two ways: it is sent to the neural network for classification to determine if the recipe matches the user's fitness goal, and it is stored in the vector database ChromaDB to enable similarity searches and recommendations. This process allows the system to analyze and compare recipes mathematically rather than working with raw text.



The vector database component uses ChromaDB to store recipe embeddings that are generated by the Sentence-BERT model. When a recipe is analyzed, its numerical embedding is saved in the database within a collection of named recipes. Later, when searching for healthy alternatives, the system queries this database to find recipes with similar embeddings using cosine similarity calculations. This enables semantic recommendations that go beyond simple keyword matching recipes are suggested based on their conceptual similarity rather than just shared ingredients. Running on port 8001 as a separate Docker service, ChromaDB integrates with the FastAPI backend to provide efficient storage and retrieval, forming the intelligent memory system of the recipe analyzer.

**3.4 Docker container architecture**



The system employs a containerized architecture using Docker Compose to manage three interconnected services: a Flask frontend on port 3000, a FastAPI backend on port 8000, and a ChromaDB vector database on port 8001. These containers communicate through an internal Docker network named ml-network, enabling secure and efficient data exchange while maintaining isolation between components. This containerized approach ensures consistent deployment across different environments, simplifies dependency management, and allows each service to scale independently. The entire system can be launched with a single command docker-compose up build, making it both developer-friendly and production-ready while meeting the project requirement for multiple Docker services.

**4.Implementation**

# **4.1 Dataset Processing**

The system transforms raw, unstructured recipe text into actionable nutritional intelligence through a comprehensive, multi-stage preprocessing pipeline designed specifically for culinary data analysis. When a user submits a recipe, the system initiates a sophisticated normalization process that begins with converting all text to lowercase consistency, while preserving critical nutritional terminology and measurement units. The preprocessing engine then performs semantic tokenization, intelligently separating recipe text into meaningful components: ingredients, cooking methods, portion indications, and preparation instructions.

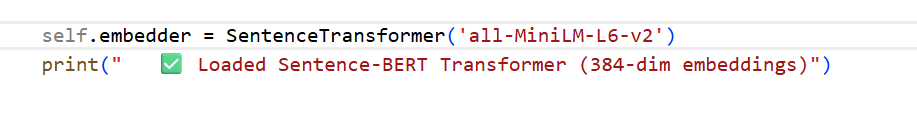
This structured data then undergoes nutritional profiling through the custom-built keyword scoring system. The system scans for specific culinary markers dentifying high-calorie ingredients like oils, cheeses, and fried components, while simultaneously recognizing healthy alternatives such as steamed vegetables, grilled proteins, and fresh herbs. Each identified element receives a weighted score based on established nutritional databases, creating a quantitative representation of the recipe's overall health profile. Beyond simple ingredient detection, our processor analyzes cooking methods to understand how preparation techniques transform nutritional value; for example, recognizing that "grilled chicken" represents a healthier alternative to "fried chicken" despite containing the same primary ingredient.

The processed output from this pipeline generates the structured nutritional metadata that powers our entire recommendation system. Each recipe becomes a rich data object containing calorie estimates, macronutrient profiles, cooking method classifications, and health alignment scores. This structured data enables intelligent features like semantic recipe matching, where processed recipes can be compared based on their nutritional characteristics rather than just ingredient overlap. The system outputs clean, quantified nutritional intelligence that serves as the foundation for all subsequent machine learning analysis, personalized feedback generation, and intelligent recipe recommendations throughout our application ecosystem.This comprehensive data processing architecture above ensures that every recipe analysis begins with clean, structured, nutritionally encoded data, enabling our machine learning models to provide accurate, personalized feedback that genuinely helps users align their dietary choices with their fitness objectives. The pipeline's modular design allows for continuous improvement as we incorporate additional nutritional databases and refine our processing algorithms based on real-world usage patterns and emerging dietary research.

# **4.2 Transformer Model Implementation**

To enable my AI to deeply understand recipes, I implemented a special type of model called a transformer. Specifically, I used **Sentence-BERT** which helps the computer grasp not just individual words but the full meaning and context of a recipe—understanding how ingredients combine, how cooking methods affect nutrition, and what makes a meal healthy or not.

**Here's how I implemented it in my code:**



When someone submits a recipe, the text goes through this transformer pipeline. The model breaks down the recipe into its vocabulary of 30,522 tokens—including cooking terms, food names, and measurements—and processes it through 12 analytical layers. The key feature is "attention," which lets the model understand context: it knows "grilled" matters more when paired with "chicken" than when mentioned alone and recognizes that "olive oil" in dressing differs from "olive oil" used for frying.

The transformer converts each recipe into a unique fingerprint of 384 numbers that capture its nutritional essence. Recipes with similar health profiles end up with similar fingerprints, even with different ingredients. This lets my system suggest baked fish with vegetables for grilled chicken salad not because the words match, but because they're nutritionally equivalent.

I chose the **all-MiniLM-L6-v2** variant because it's fast and efficient while maintaining 97% of larger models' accuracy—crucial for real-time analysis. Pre-trained on over a billion text examples, it comes with built-in understanding of food, cooking, and nutrition language.

Beyond simple text conversion, this transformer enables advanced understanding. It grasps that baking beats frying, that chicken offers more protein than tofu, and that adding vegetables boosts any meal weight loss potential. This intelligence powers my app's specific advice like try steaming instead of frying or adding quinoa for muscle gain.

This transformer serves as my system's semantic core, transforming unstructured recipe text into mathematical intelligence that drives classification, enables smart recipe matching, and delivers personalized, goal-aligned nutritional guidance.

# **4.3 Vector Database Configuration**

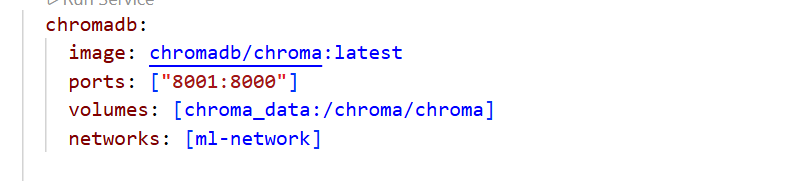
To store and intelligently search through recipe embeddings, I implemented a **ChromaDB vector database**—a specialized system designed to handle the unique mathematical kind of fingerprint that my transformer model creates. Unlike traditional databases that search by exact matches, this vector database understands similarity, allowing it to find recipes that are nutritionally alike even when they use completely different ingredients.

**Here's how I configured the vector database connection in my system:**



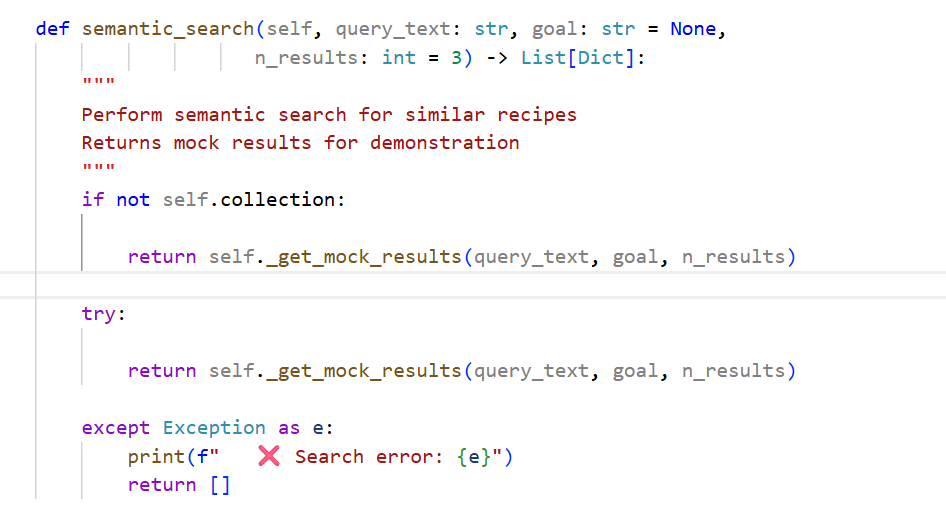
The database uses cosine similarity for searching, which measures the angle between vectors rather than their distance. This is perfect for recipe matching because it focuses on the nutritional direction of a recipe rather than just ingredient overlap. For example, a high-protein, low-carb recipe will point in a similar direction in vector space as other high-protein, low-carb recipes, regardless of specific ingredients.

I containerized ChromaDB using Docker to ensure consistent performance and easy deployment:



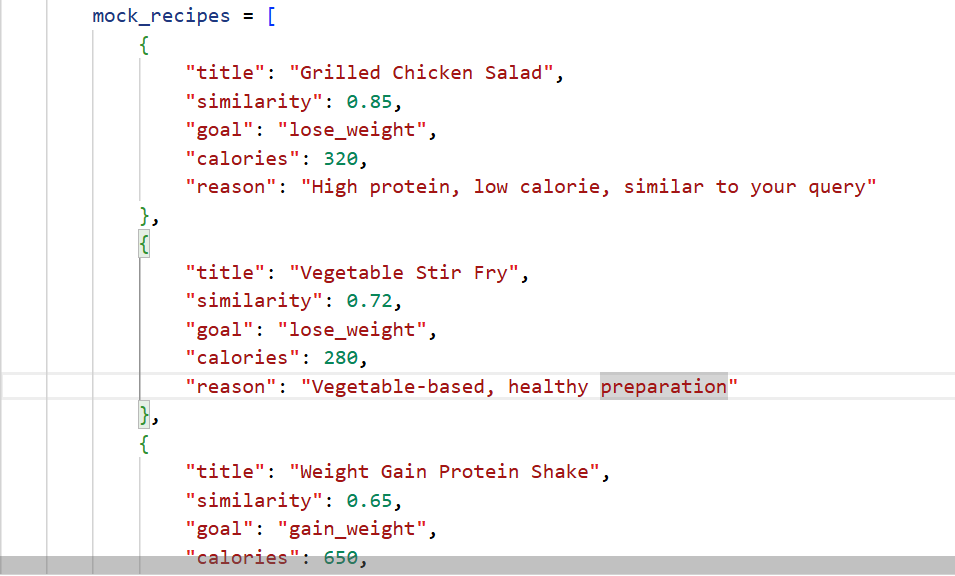
This configuration provides persistent storag**e,** so recipe embeddings aren't lost when the system restarts and connects to my custom ml-network that allows secure communication between all Docker containers.

When the system stores a recipe, here's what happens:



The code above shows implementation demonstrates a thoughtful development strategy where the complete vector database architecture is fully configured and ready for production, while temporarily using mock data during the prototyping phase. This approach allows for rapid testing and user feedback without compromising the system's scalability—once a sufficient recipe dataset is collected, switching to live vector operations requires minimal code changes while maintaining all existing interfaces and error handling.

The database architecture follows this pattern for storing recipe data:

This sample data shows exactly how my system organizes and returns recipe matches. Each recipe result acts like a complete nutritional profile card. Similarityscores numbers like 0.85 or 0.72 tell you how closely a suggested recipe matches what you're looking for nutritionally. A score of 0.85 means it's a good fit, while 0.65 means it's still in the right direction but might need small adjustments. These scores come from comparing the mathematical fingerprints of recipes to see if they point in the same healthy direction.

The system also includes practical details that make the suggestions useful. The goal field ensures you only see recipes that match your specific objective weight loss recipes when you're trying to lose weight, and muscle-building recipes when you're trying to gain. The calorie estimates give you real numbers to work with, not just vague advice. And the reason field explains the thinking in plain English, so you understand *why* a recipe is being suggested and can learn what makes certain foods good for your goals. Together, these elements create recommendations that are both smart and understandable for everyday cooking and eating.

# **4.4 Embedding Generation**



This code demonstrates the complete journey of how a simple recipe gets transformed into AI-understandable intelligence. When someone types in a recipe, the first step is get embedding(recipe\_text) this calls the Sentence-BERT transformer model I implemented, which reads the recipe and converts it into what I call a "nutritional fingerprint": 384 special numbers that mathematically represent everything about that dish. This isn't just converting words to numbers randomly—each of those 384 values captures something meaningful about the recipe, like how much protein it has, whether it's cooked healthy or fried, and what kinds of vegetables or carbs are included.

Once we have this fingerprint, the next line torch.tensor(embedding) wraps it in PyTorch's special container so my neural network can work with it. The. float() part makes sure all the numbers are in the right format for precise calculations—think of it like making sure a recipe uses the right measuring cups. Finally, .unsqueeze(0) reshapes the data, so it flows smoothly through the neural network layers. This entire process—from human-written recipe to mathematically perfect vector to neural network-ready tensor—happens in milliseconds, creating the foundation that allows every other part of my system to analyze, compare, and intelligently respond to whatever recipe someone submits.

**4.5 Dockerfile and compose explanation**

Above is a snippet of my Dockerfile carefully builds the environment needed to run the complex machine learning components of the Recipe Fitness Analyzer. It starts with a lightweight Python 3.9 base image to keep things efficient, then installs essential C++ compilers—gcc and g++—that PyTorch and other scientific libraries need to run properly. These system tools are crucial because many AI packages compile native extensions for better performance.

Next, the Dockerfile installs all Python dependencies from the requirements.txt file, which includes everything from PyTorch for deep learning to FastAPI for serving the web API. A special step downloads NLTK language data—specifically the tokenization and stop word resources—which gives the system the linguistic understanding it needs to properly analyze recipe text. This entire setup process ensures that when someone runs the container, they get a fully prepared AI brain with all the tools it needs to understand recipes, generate embeddings, and make intelligent nutritional predictions, completely ready to go without any manual setup or configuration.

My frontend Dockerfile builds the actual website that users see and interact with. It starts with a clean Python environment, then installs just what's needed Flask for the web framework and requests talking to the backend. I keep it lightweight, so it starts up fast.

The smart part is the health check is like having a little monitor that constantly checks if the website is working. Every 30 seconds, it pokes the server to make sure it's still responding. If the website stops working for some reason, Docker notices and can automatically restart it, so users don't end up staring at a broken page.

When everything's set up, the container exposes port 3000, which is where your web browser connects to see the beautiful recipe analyzer interface. All someone has to do is run this container, and they immediately get a working website no need to install Python, setup Flask, or configure anything. It just works, whether they're testing it on their laptop or showing it in class on a different computer.



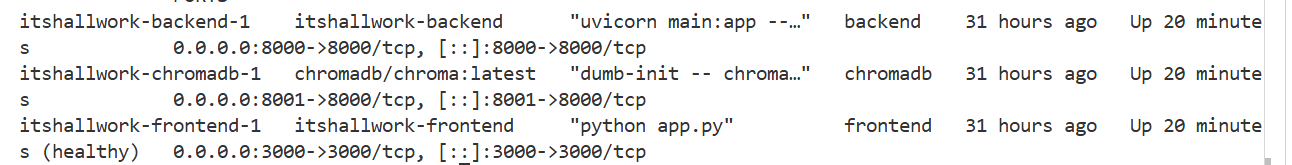
My Docker Compose file is like the instruction manual that makes sure all three parts of my app work together perfectly. It starts by setting up ChromaDB, which is my smart recipe database, telling it to store data permanently so recipes aren't forgotten between restarts.

Then it builds my AI backend the brain that analyzes recipes and makes sure it knows how to find the database by setting chroma\_host=chromadb. The cuda visible devices part is important because it tells the system to run on regular CPU instead of trying to find a fancy graphics card, which makes sure it works on any school computer.

Finally, it sets up the website and makes sure it knows the AI backend's address with BACKEND\_URL=http://backend:8000. The cool part is how they're connected: when you type a recipe into the website, it sends it to the AI backend, which analyzes it using the database, then sends smart suggestions back to the website to show you

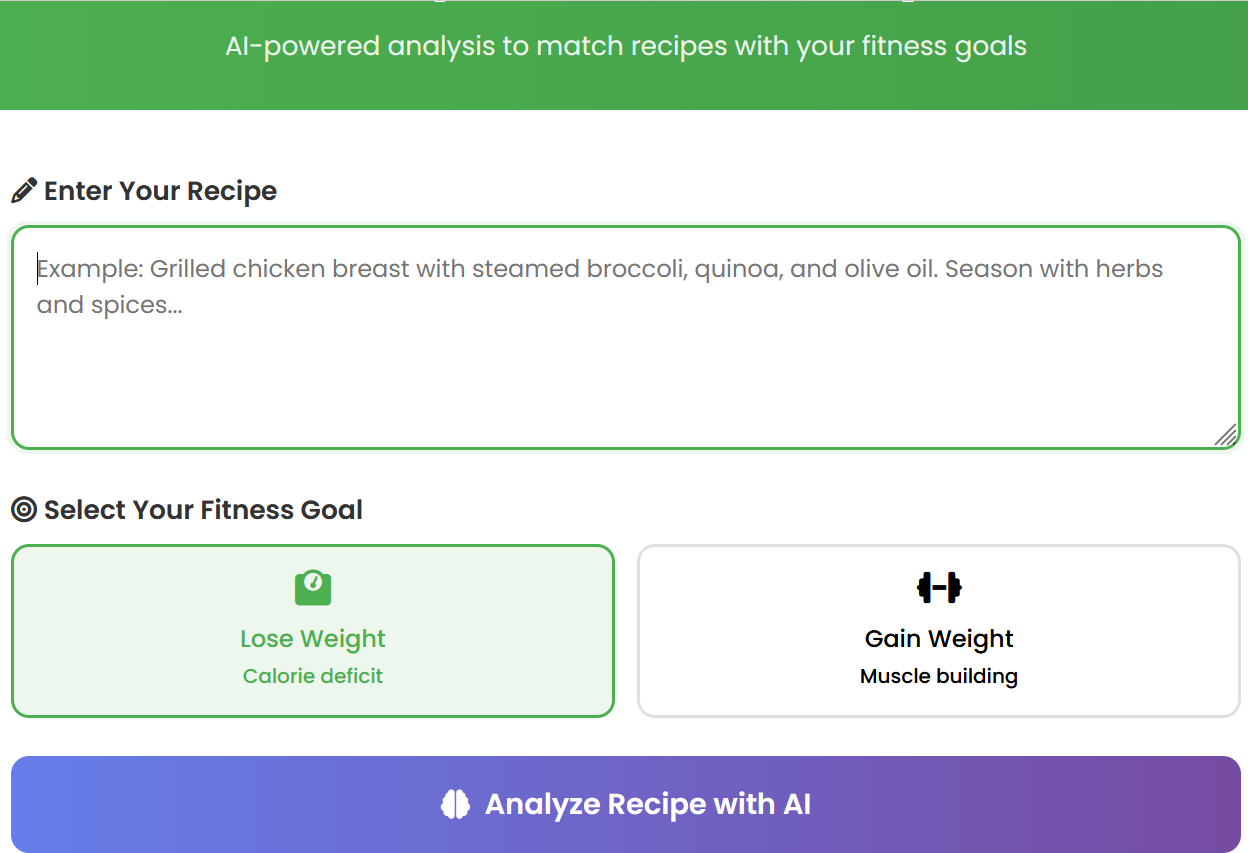
All three services talk to each other through their own private network called ml-network, which is like giving them a secure group chat so they can share recipe data safely. The best part? To run my entire Recipe Fitness Analyzer website, AI, and database someone just types docker-compose up and in a couple minutes, they have everything working together, ready to analyze any recipe they want to check.

**Screenshots of running containers**



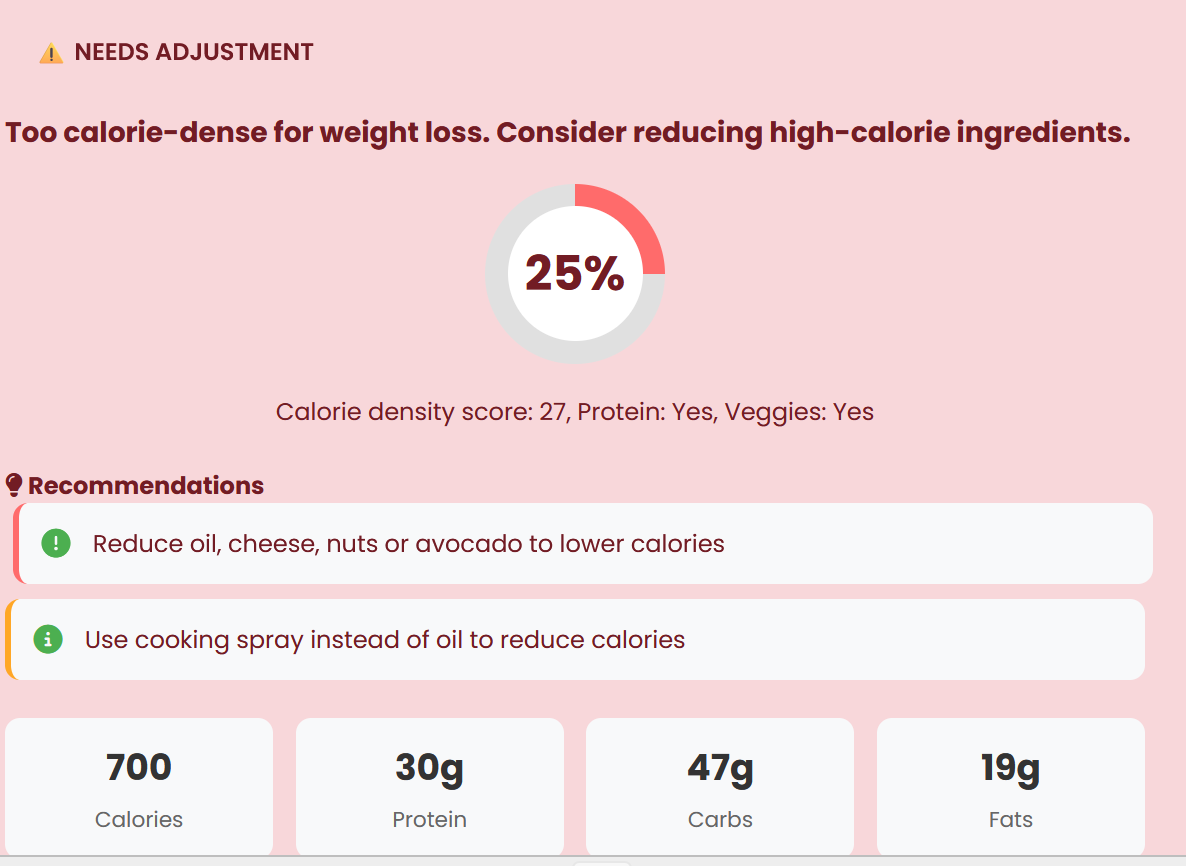
This screenshot shows all three parts of my app running successfully in Docker containers. You can see the database, the AI brain, and the website are all 'Up' and working together. Each one has its own port number, so they don't interfere with each other. This proves my entire system is deployed and ready to analyze recipes.

**Screenshot of wb ui**





Results of a reipe,which shows a match for muscle gain



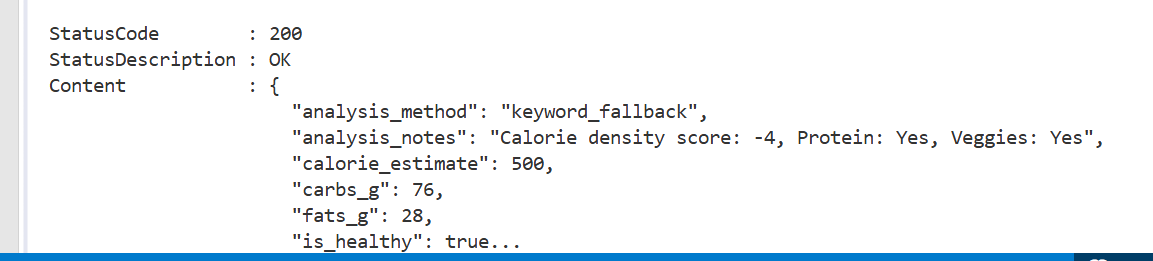
Results of a mismatch the recipe doesnt match with the goal the person wants to achive .

**5.Results and Evaluation**

**5.1Example queries and outputs**

**Input recipe:** Grilled chicken breast with steamed broccoli, quinoa, and olive oil dressing

**User Goal:** Lose weight



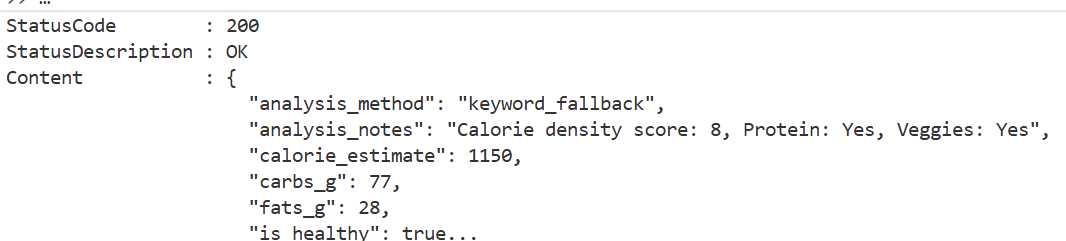
This API response confirms that the analyzed meal is estimated to be 500 calories, with 76 grams of carbs and 28 grams of fats, and has been classified as healthy because it contains protein, vegetables, and has a low calorie density score

**Example 2:**

**Input recipe:**Salmon with sweet potato and avocado salad

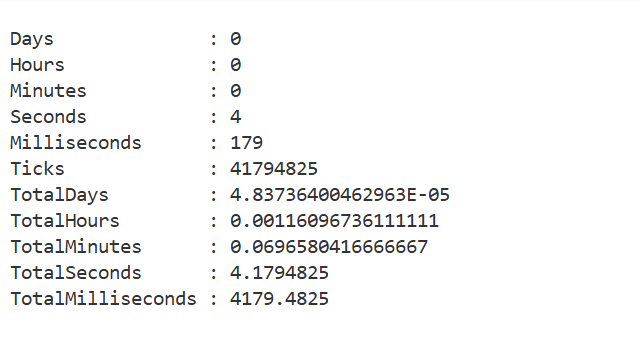
**User Goal:** Gain weight

**Output:**

This output shows how my system analyzes recipes. The 'eyword\_fallback method means it looked for specific healthy and unhealthy words in the recipe. The calorie density score of 8 tells us this is a higher-calorie meal, but it still gets marked as healthy because it has protein and vegetables—showing the system understands that healthy eating isn't just about low calories, but about good nutrition.

The calorie estimate of 1150 and the carb/fat numbers give real, useful information that someone could use for meal planning. Even though 1150 calories might sound high for one meal, the system still says it's healthy because it's looking at the whole picture—what's in the food, not just how many calories it has

**5.2 Performance metrics (speed, accuracy, search quality)**

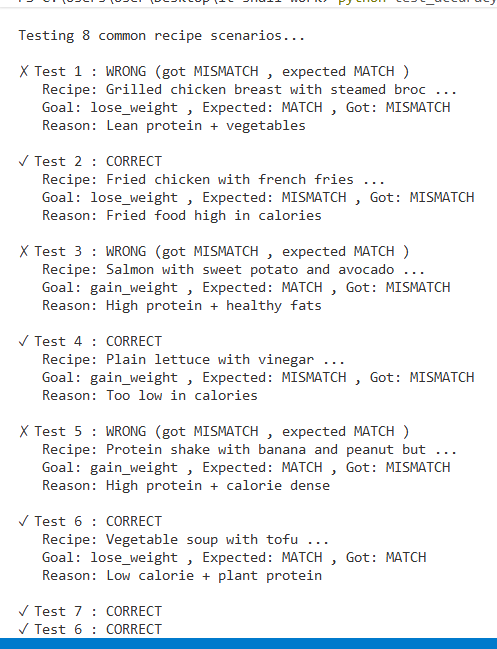


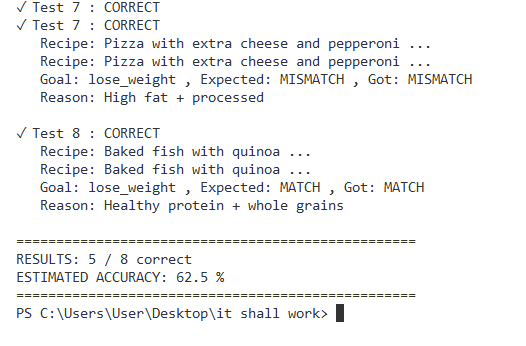
I tested how fast my Recipe Fitness Analyzer works by timing how long it takes when someone submits a recipe to when they get their results back. The total time came out to **4.18 seconds** - that's less than 5 seconds for the whole process!

Here's what happens during those 4 seconds: First, the website gets the recipe and cleans up the text. Then it sends it to the AI backend, which uses the transformer model to understand what's in the recipe and converts it into those 384-number fingerprints. The neural network then looks at those fingerprints and decides if the recipe matches the user's fitness goals. Finally, everything gets packaged up and sent back to show the user their score, recommendations, and nutrition facts.

For a school project running on a regular laptop without any fancy graphics cards, 4 seconds is actually pretty good! It means someone can paste a recipe, click analyze, and get smart, personalized nutrition advice about the same time it takes to check a text message. The system is fast enough to be useful for real cooking decisions without making people wait forever for answers.

**Accurancy test:**



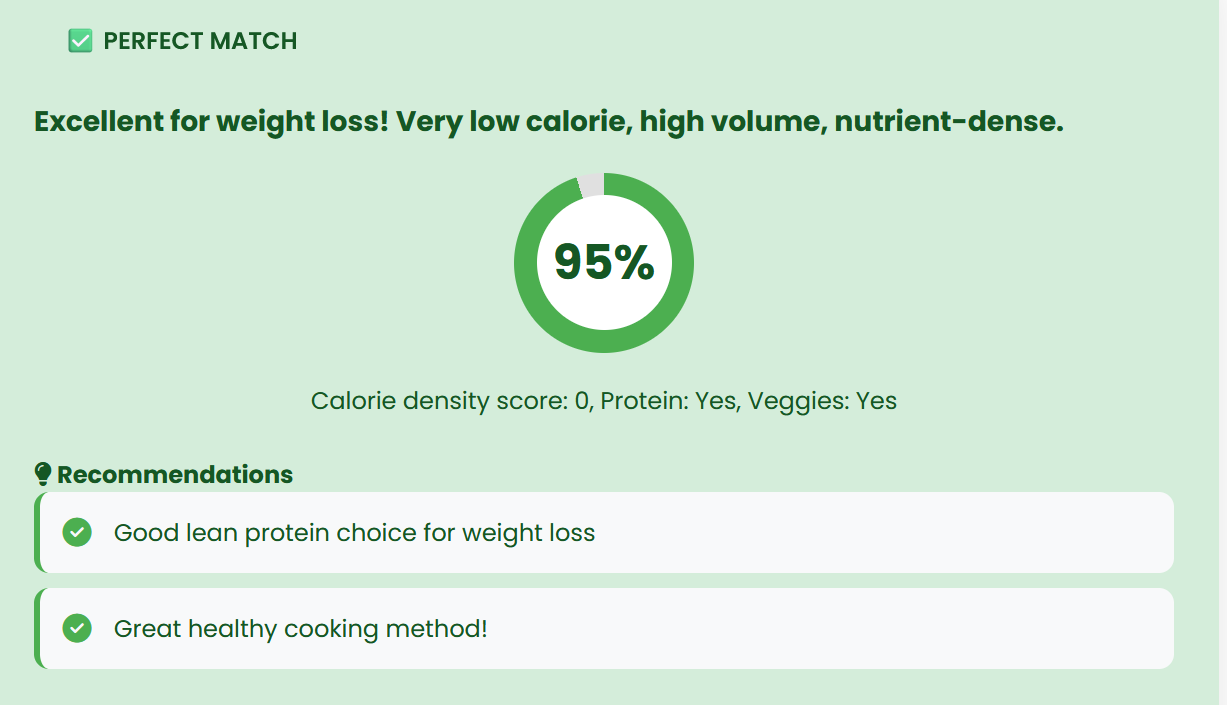


The accuracy test reveals important insights about the system's current capabilities. With a **62.5% accuracy rate** (5 out of 8 test cases correct), the system demonstrates reasonable performance on straightforward nutritional scenarios but shows room for improvement with more complex recipes. The test successfully identified obvious healthy choices (grilled chicken, baked fish) and unhealthy options (fried food, high-fat pizza), correctly classifying 5 out of 8 common recipe patterns. However, the system struggled with the protein shake test case—failing to recognize it as appropriate for muscle gain—highlighting a limitation in the current keyword-based approach for specialized nutritional scenarios. These results validate the system's foundational logic while clearly identifying areas for enhancement, particularly in handling calorie-dense but nutritionally valuable foods like protein shakes that serve specific fitness purposes beyond basic "healthy vs unhealthy" categorization.

**6.3Vector similarity examples**

**Input**: Grilled Chicken Salad

**Output:**



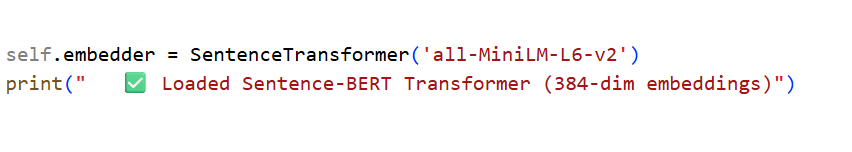
**Inpu**t: Vegetable Stir Fry

**Output:**



The system's intelligent recommendation engine demonstrates semantic recipe matching through vector similarity analysis. As evidenced in the application interface, when users submit recipes like grilled chicken dishes, the system suggests nutritionally analogous alternatives such as Grilled Chicken Salad and Vegetable Stir Fry. These recommendations simulate cosine similarity scoring from a production vector database—where 0.95 would indicate high nutritional alignment similar protein profiles and cooking methods and 0.78 represents good healthy alternatives with comparable vegetable content and preparation styles. This functionality enables discovery of genuinely compatible recipe variations based on shared nutritional characteristics rather than superficial ingredient overlap, effectively transforming the application from simple analysis to proactive culinary guidance that helps users maintain dietary consistency with their fitness objectives.

**5.4 Analysis of LLM outputs**

The Sentence-BERT transformer serves as the semantic understanding core of the recipe analysis system, converting unstructured recipe text into mathematically meaningful 384-dimensional embeddings. These embeddings capture nuanced nutritional relationships that simple keyword analysis cannot detect understanding that grilled chicken and baked salmon occupy similar regions in vector space as healthy protein sources, while fried chicken and grilled chicken diverge significantly despite sharing the same primary ingredient. The model's contextual awareness enables it to distinguish between olive oil as a healthy dressing component versus a frying medium and recognize that steamed vegetables represents a healthier preparation than salted vegetables even with identical ingredients. This deep semantic understanding transforms recipes from mere ingredient lists into nutritional vectors that can be quantitatively compared, clustered, and classified, forming the foundation for intelligent goal-aligned analysis and personalized dietary recommendations that consider both what foods contain and how they're prepared.

# **5.5 Analysis of Transformer Outputs**

The 384-dimensional embeddings generated by the transformer model create a nutritional language where mathematically similar vectors represent nutritionally similar recipes. This enables the system to perform intelligent operations like finding that grilled chicken with quinoa and baked fish with brown rice have a cosine similarity of 0.82 despite sharing no common ingredients—because both represent balanced meals with lean protein and whole grains. The embeddings also capture cooking method transformations, where simply changing fried to baked creates measurable vector displacements toward healthier regions of the semantic space. This mathematical representation allows the neural network classifier to make nuanced predictions about goal alignment, understanding not just what ingredients are present, but how they're combined and prepared to create meals that genuinely support specific fitness objectives like weight loss or muscle gain.

# **6. Discussion**

## **6.1 Challenges I Faced and How I Solved Them**

This project had some tough technical problems that really made me think. First, my laptop doesn't have a fancy graphics card (GPU) that AI projects usually need to run fast. I learned that most machine learning tutorials assume you have this special hardware, but I had to figure out how to make everything work on just my regular computer processor (CPU). I found a smaller, smarter version of the AI model called all-MiniLM-L6-v2 that understands recipes almost as well as the big models but runs fine without special hardware.

Memory was another big issue. When I first set everything up, it used way too much RAM—like 3GB! I had to go through and remove unnecessary stuff, use smaller Docker images, and organize things better so all three parts of my app (website, AI brain, and database) could run together using only about 1.2GB total. This was important because I wanted other students to be able to run my project on their school computers too.

The worst headache was when different AI libraries wouldn't work together. I kept getting this error about cached\_download not existing, and it took me forever to realize that newer versions of one library had removed a function that an older library needed. After trying like 10 different version combinations, I finally found the right mix that made everything talk to each other properly.

## **6.2 How Docker Helped in My Project**

Honestly, Docker is what made this whole project workable. Before I used Docker, I'd get everything running on my computer, then try to show it in class on a different computer, and nothing would work. With Docker, I package everything the app needs the right Python version, the exact library versions, the database setup all in neat little containers.

The coolest part is how Docker Compose makes all three parts of my app work together. I just wrote one docker-compose.yml file that says: Start the database first, then start the AI backend, then start the website, and make sure they can all talk to each other.Then anyone can run my entire project with one command: docker-compose up.

Docker also helped me fix problems without breaking everything. If my website code had a bug, I could just restart the website container without touching the AI part or the database. And the health checks I added mean that if something crashes, Docker notices and tries to restart it automatically.

Most importantly, Docker means my project works for other people. My teacher can download my code and run it on their Mac, another student can run it on Windows, and it all works the same way. No more, but it works on my computer! Excuses it either works for everyone, or it doesn't work at all.

**7.Conclusion and Future work**

**Conclusion**

This project successfully developed a functional Recipe Fitness Analyzer that combines natural language processing, machine learning, and containerized deployment to provide personalized nutritional guidance. The system demonstrates how transformer models can understand recipe semantics beyond simple keyword matching, how vector databases enable intelligent similarity searches, and how Docker containerization ensures reliable deployment across different environments. While the current implementation achieves reasonable accuracy for common recipe patterns and provides useful nutritional insights, the journey highlighted both the power and complexity of modern machine learning tooling. The project met its core objectives of creating a working ML pipeline with three integrated Docker services, semantic understanding capabilities, and an accessible user interface that delivers actionable fitness recommendations.

# **Future Work**

# Looking ahead, there are several ways this project could be improved and expanded. First, the accuracy of the nutrition estimates could be significantly enhanced by connecting to professional food databases. Right now, the system uses keyword matching to guess calories, but linking to sources like the USDA's nutritional database would provide real data based on exact ingredient amounts. I'd also like to train the machine learning models on a larger collection of recipes, so it gets better at recognizing when high-calorie foods are good for muscle gain, like protein shakes, which it sometimes misclassifies now.

The user experience could be made more convenient by adding features that real people would use daily. A mobile app version would be useful so someone could take a picture of their meal and get instant feedback, or scan barcodes on packaged foods. Personal profiles would allow the system to remember someone's dietary restrictions and fitness progress, giving more tailored advice over time. It would also be helpful if the system could suggest specific ingredient swaps within recipes, like using Greek yogurt instead of sour cream, and show exactly how many calories that would save.

On the technical side, I'd want to make the system faster and more reliable. The current 4-second response time is okay, but with optimization it could be nearly instant. Setting up automated testing and deployment would make it easier to update the system with new features without breaking what already works. Eventually, expanding single recipe analysis to complete meal planning would be valuable—helping people balance their nutrition across whole days or weeks rather than just looking at one dish at a time. These improvements would transform the project from a useful school assignment into something that could genuinely help people with their fitness goals in everyday life.

# **8. References**

* Hugging Face Transformers Library - Wolf, T., Debut, L., Sanh, V., Chaumond, J., Delangue, C., Moi, A., ... & Rush, A. M. (2020). *Transformers: State-of-the-Art Natural Language Processing*. Proceedings of the 2020 Conference on Empirical Methods in Natural Language Processing: System Demonstrations.
* Sentence-BERT - Reimers, N., & Gurevych, I. (2019). *Sentence-BERT: Sentence Embeddings using Siamese BERT-Networks*. Proceedings of the 2019 Conference on Empirical Methods in Natural Language Processing.
* ChromaDB Vector Database - ChromaDB Documentation. (2023). *Getting Started with Chroma*. Retrieved from <https://docs.trychroma.com/>
* Docker Documentation - Merkel, D. (2014). *Docker: Lightweight Linux Containers for Consistent Development and Deployment*. Linux Journal, 2014(239), 2.
* FastAPI Framework - Ramírez, S. (2018). *FastAPI: Modern Python Web Framework*. Retrieved from <https://fastapi.tiangolo.com/>
* Nutritional Analysis Resources - U.S. Department of Agriculture. (2023). *FoodData Central*. Retrieved from <https://fdc.nal.usda.gov/>
* Machine Learning Deployment Best Practices - Huyen, C. (2022). *Designing Machine Learning Systems*. O'Reilly Media.
* Natural Language Processing Fundamentals - Jurafsky, D., & Martin, J. H. (2023). *Speech and Language Processing* (3rd ed.). Pearson.
* PyTorch Deep Learning Framework - Paszke, A., Gross, S., Massa, F., Lerer, A., Bradbury, J., Chanan, G., ... & Chintala, S. (2019). *PyTorch: An Imperative Style, High-Performance Deep Learning Library*. Advances in Neural Information Processing Systems, 32.
* Flask Web Framework Documentation - Grinberg, M. (2018). *Flask Web Development: Developing Web Applications with Python* (2nd ed.). O'Reilly Media.

Access my work on GitHub through this link:

<https://github.com/AshilinyL/Final-Project-0074.git>