
# AI Dietitian

Senior Project

By:

**Hasan Al Mousawi & Ali Zalghout**

Submitted to the Data Science Department

of the **Lebanese University**

Beirut, Lebanon

In partial fulfilment of the requirements for the degree of

**BACHELOR OF DATA SCIENCE**

2025–2026

---

Supervised by:

***Dr. Kassem Danach***

---

## Acknowledgment

We would like to express our heartfelt appreciation and sincere thanks to our professors and mentors for their invaluable support, guidance, and encouragement throughout the development of this project.

Their dedication to teaching, commitment to student success, and willingness to share their expertise have played a significant role in helping us complete this work. The knowledge and skills we gained under their guidance have greatly contributed to both our academic achievements and personal growth.

We are particularly thankful to Dr. Kassem Danach for his outstanding supervision, continuous support, and thoughtful advice throughout every phase of our project, NutriAI: An AI-Powered Personalized Nutrition Assistant. His insightful recommendations, patience, and constructive feedback helped us overcome challenges and continuously improve the quality of our work.

We would also like to acknowledge all the faculty members who contributed to our learning journey and provided us with the foundation necessary to undertake this project. Their efforts have been essential in shaping our knowledge, skills, and aspirations.

To Dr. Kassem Danach and all our professors, we extend our deepest gratitude for their support, dedication, and lasting impact on our educational journey. Their influence will continue to guide and inspire us as we move forward in our academic and professional careers.

## Abstract

This project, titled "NutriAI: An AI-Powered Personalized Nutrition Assistant," presents a comprehensive web-based platform designed to deliver intelligent and highly personalised nutritional guidance by integrating artificial intelligence, machine learning, and modern web technologies. The primary objectives were to develop a system capable of generating personalised meal recommendations based on each user's fitness goal, dietary preferences, and nutritional requirements, while also providing AI-driven conversational assistance, adaptive diet adherence analysis, and body composition follow-up assessments to support long-term health improvement.
The system was implemented using Python and the Flask web framework for the backend, PostgreSQL as the relational database, and a responsive frontend built with HTML, CSS, and JavaScript. The recommendation engine employs a hybrid approach that combines content-based filtering — using a rating-weighted TF-IDF profile vector over the Food.com recipe dataset, containing over 500,000 recipes and 1.4 million user reviews — with item-based collaborative filtering derived from user rating patterns, goal-aligned nutritional scoring, and a logarithmic popularity bonus, further refined through Maximal Marginal Relevance re-ranking to balance relevance with recommendation diversity. An extensive preprocessing pipeline was developed to clean the dataset, engineer nutritional features, assign meal types, and filter unsuitable content before model training.
Artificial intelligence capabilities were integrated through the Groq API running the LLaMA 3.3-70B model to power the conversational AI chef assistant, generate InBody body composition analyses, and derive personalised nutritional insights. Machine learning techniques including MiniBatchKMeans clustering were employed to derive dish-type categories for recommendation diversity, while item-based k-NN collaborative filtering and user-user similarity over the Food.com review dataset together power the hybrid engine and the Similar Users recommendation tier. Additional features include real-time exercise calorie deduction, meal image fallback retrieval via the Pexels API, a BMI and InBody tracking panel, weekly nutrition charts, streak achievements, and a dietary preference filtering system that enforces hard constraints such as vegan, vegeterian, and dairy-free requirements.
The results demonstrate the feasibility of combining large language models, traditional machine learning, and full-stack web development into a cohesive nutrition management platform. The system successfully delivers personalised meal suggestions, calorie and macro tracking, conversational dietary advice, and body composition assessment, highlighting the potential of AI-powered tools to improve user engagement and long-term adherence to healthy dietary patterns.

**TABLE OF CONTENTS**

# TABLE OF FIGURES

Now I have everything. Here is your complete, accurate Table of Figures — all 28 figures with correct page numbers:

# Chapter 1 : Introduction

## 1.1 General Introduction

Nutrition is one of the most important determinants of long-term health, yet personalised dietary guidance has historically been accessible only to individuals who could afford professional dietitian consultations, or study huge amount of info about the topic. The rapid advancement of artificial intelligence and machine learning technologies has opened new possibilities for delivering expert-level nutritional support at scale, without the cost and scheduling barriers associated with in-person consultations , and its consequent privacy and freedom sacrifices.

NutriAI is a web-based AI Dietitian application developed to address this gap. The system provides users with goal-driven, personalised meal recommendations drawn from a dataset of over 500,000 real-world recipes, a conversational AI chef assistant capable of answering nutritional questions, logging food and exercise, and suggesting meals in real time, as well as a body composition tracking module that analyses InBody scan results to provide clinical-level dietary feedback. Every feature is unified under a single responsive web interface accessible from any device.

The platform integrates a hybrid recommendation engine that combines content-based filtering through TF-IDF vectorisation with goal-aligned nutritional scoring, a logarithmic popularity bonus, and dietary preference boosting. with item-based collaborative filtering derived from Food.com rating patterns, goal-aligned nutritional scoring, and a logarithmic popularity bonus, further refined through Maximal Marginal Relevance re-ranking to balance relevance with recommendation diversity The recommender is personalised by the user's registered goal — weight loss, maintenance, or muscle building — dietary restrictions, favourite ingredients, and historical ratings. These inputs are used to rank and filter meals from the Food.com dataset, sourced from Kaggle's irkaal/foodcom-recipes-and-reviews collection, which was preprocessed through a multi-stage pipeline before being used for modelling.

Conversational intelligence is delivered through the Groq API with the LLaMA 3.3-70B model, which serves as the backbone for the AI chef chatbot, the InBody body composition analyser, and the automated nutritional insight generator. The system detects user intent from natural language messages — distinguishing between food logging, exercise reporting, meal confirmation, and general conversation — and responds accordingly with instant feedback, macro tracking updates, and personalised advice.

## 1.2 Problem Statement

Despite widespread awareness of the relationship between diet and health, a large proportion of individuals struggle to maintain consistent, nutritionally balanced eating patterns. This difficulty arises from a combination of factors: the complexity of nutritional science, the monotony of repeated meal plans, the inaccessibility of professional dietitian services, the culture spread about the value of commitment and discipline needed in diet planning that changes peoples' routine

completely, and the lack of systems that adapt recommendations to an individual's changing body composition and daily habits.

Existing nutrition applications, while effective for basic food logging, exhibit several limitations that reduce their practical value as adaptive dietary tools. Most platforms rely on static, pre-defined meal plans that do not adjust dynamically to a user's actual intake, behavior, or progress over time. Feedback is typically limited to raw calorie and macronutrient totals, with little to no interpretation of whether these figures genuinely align with the user's specific goal, current body composition, or rate of progress — a user consistently undereating relative to their target and a user consistently overeating receive the same static dashboard, with no system-level distinction between an adherence problem and a target that is simply miscalibrated. Food logging itself generally depends on manual search-and-select from a fixed database, offering little flexibility for natural language input, unlisted or home-cooked meals, or varied phrasing, and requiring users to conform to the interface rather than the interface adapting to them. Finally, these platforms rarely incorporate longitudinal body composition data — such as InBody scans or BMI trends — into their dietary recommendations, treating weight tracking and nutrition logging as separate, disconnected features rather than signals that should inform one another.

NutriAI directly addresses these limitations by providing a unified platform that integrates personalised recommendation, conversational AI assistance, real-time logging, macro tracking, and body composition analysis in a single application. The system is designed to adapt continuously to the user's profile, goals, and progress, offering a level of dietary intelligence that was previously available only through expensive, professional consultations.

### 1.3 Motivation

The motivation for this project stems from both the growing global burden of diet-related chronic disease and the demonstrated potential of AI to transform healthcare delivery. Conditions such as obesity, type 2 diabetes, cardiovascular disease, and nutritional deficiencies are closely linked to poor dietary habits and are placing increasing strain on healthcare systems worldwide. Personalised nutrition interventions have been shown to be significantly more effective than generic dietary guidelines, yet access to such interventions remains highly unequal.

At the same time, large language models and machine learning techniques have matured to the point where they can engage in meaningful, context-aware conversations about complex topics such as nutrition. Datasets such as the Food.com recipe collection provide a rich, real-world foundation for building recommendation systems that surface genuinely practical and culturally diverse meal options. The combination of these technological capabilities with an unmet user need represented a compelling opportunity for a final-year data science project.

Additionally, the team observed a recurring pattern in the nutrition app market: tools that provide either recommendation or tracking, but rarely both, Not enough utilization of AI potential in the sector, and almost never with the depth of personalisation required to account for individual

differences in body weight, muscle mass, fat percentage, activity level, and dietary restrictions. Building a system that integrates these dimensions into a single coherent experience was both technically challenging and genuinely useful.

Finally, the integration of InBody body composition analysis into dietary feedback represents a novel contribution . By connecting measurable physical outcomes — changes in weight, muscle mass, and body fat percentage between scans — to the user's logged dietary history, the system is able to provide evidence-based assessments of whether the diet plan is working, and prescribe specific calorie and macro adjustments when the data indicates a need. This level of feedback loop is typically only available in supervised clinical nutrition programmes.

### 1.4 Contribution

The primary contributions of the NutriAI project are as follows:

- Hybrid Recommendation Engine: A combination content-based filtering, via a rating-weighted TF-IDF profile vector, item-based k-NN collaborative filtering derived from Food.com rating patterns, goal-aligned nutritional scoring, and popularity-weighted ranking, with Maximal Marginal Relevance re-ranking applied to balance relevance against recommendation diversity. A separate two-tier Similar Users engine first surfaces recipes highly rated by profile-matched NutriAI peers (by goal, BMI, activity level, and dietary preferences), falling back to Food.com users with similar rating patterns when peer data is insufficient, with a content- and goal-score-based fallback for users with no rating history at all. The engine enforces strict dietary hard filters for restrictions such as vegan, vegetarian, dairy-free, and nut-free, while applying soft score boosts for preferences such as high-protein and low-carbohydrate.
- Large-Scale Preprocessing Pipeline: A comprehensive multi-stage preprocessing pipeline was designed and implemented for the Food.com dataset, including structural cleaning, per-serving calorie filtering, dessert removal, ingredient unit prediction, per-serving quantity normalisation, duplicate removal, and realistic validation. Goal-aligned nutritional scoring is computed at application load time in the recommendation module, using percentile-clipped per-serving nutrition values to prevent outlier recipes from dominating the score
- AI-Powered Conversational Assistant: An AI chef chatbot powered by the Groq API and the LLaMA 3.3-70B language model provides real-time dietary assistance. The chatbot detects user intent using a Groq-based intent classifier — distinguishing food logging, exercise logging, meal confirmation, and general chat — and responds with contextually appropriate macro updates, meal suggestions formatted with structured JSON blocks, and personalised advice grounded in the user's current nutritional profile.

- InBody Body Composition Analysis: An automated body composition analysis module compares InBody scan results across test periods, computes calorie and protein adherence scores, detects contradictions between logged diet and physical outcomes, and generates a Groq-powered clinical assessment including a headline, paragraph analysis, and optional macro target adjustments.
- Comprehensive Full-Stack Application: A complete web application was delivered with user authentication using JWT, a PostgreSQL backend with full data persistence, a responsive HTML/CSS/JavaScript frontend, a weekly nutrition dashboard with Chart.js visualisations, an achievements and streak system, real-time exercise calorie deduction, and a Pexels API-powered meal image fallback system.

# Chapter 2 : Foundations and Terms

## 2.1 Literature Review

### *2.1.1 Personalised Nutrition Systems and AI*

Examined the role of machine learning in generating personalised dietary recommendations and found that hybrid recommendation approaches — combining collaborative and content-based filtering — substantially outperformed single-method systems in terms of user satisfaction and nutritional alignment. Their work highlighted the importance of incorporating user-specific constraints such as dietary restrictions and health goals, both of which are central design principles in NutriAI. The study also noted the need for real-time personalisation that adapts to changes in user behaviour, a requirement addressed in NutriAI through dynamic macro tracking and goal-aware scoring.

### *2.1.2 Collaborative Filtering for Recipe Recommendation*

Conducted an extensive evaluation of collaborative filtering algorithms applied to food and recipe datasets, demonstrating that matrix factorisation methods produced significantly lower root mean square error (RMSE) than memory-based approaches such as user-based and item-based k-nearest neighbours. NutriAI nonetheless uses item-based k-nearest neighbours, blended with content-based similarity in the hybrid engine, alongside a separate user-user collaborative filtering tier in the Similar Users feature that surfaces recipes highly rated by Food.com users with similar rating patterns. The paper's cold-start problem is addressed in NutriAI through the content-based fallback path and the new-user goal-score recommender.

### *2.1.3 Large Language Models in Health Applications*

Evaluated the performance of large language models on medical examination questions and clinical vignettes, finding that models such as PaLM performed comparably to expert clinicians on several benchmarks. While NutriAI operates in the nutrition rather than clinical diagnosis domain, the results support the feasibility of using LLMs to provide contextually appropriate, expert-level dietary advice. The use of the LLaMA 3.3-70B model via the Groq API for the AI chef assistant and InBody analysis aligns with this research direction, employing a powerful open-weight model accessed through a low-latency inference endpoint rather than proprietary APIs.

### *2.1.4 Nutrition Tracking and User Behaviour*

Analysed patterns of food diary abandonment and identified that users most frequently stopped logging meals when the process was perceived as effortful, when they felt they had nothing interesting to report, or when they were uncertain whether their entries were accurate. NutriAI directly addresses these failure modes by allowing users to log food in natural language through the chatbot (reducing effort), providing AI-generated meal suggestions to reduce decision fatigue, and using the Groq API to estimate nutrition for any described food item rather than requiring users to search a database entry-by-entry.

### *2.1.5 Body Composition Assessment in Dietary Interventions*

Reviewed evidence on the relationship between body composition measurements and dietary outcomes, arguing that clinical nutrition interventions should integrate objective body composition data — such as InBody bioelectrical impedance analysis — rather than relying solely on body weight or BMI. Their recommendation directly motivates the InBody tracking module in NutriAI, which captures weight, body fat percentage, muscle mass, visceral fat level, body water percentage, BMI, and basal metabolic rate, and uses these values to evaluate the effectiveness of the user's diet between scan sessions. The inter-test period analysis, wherein meal history is evaluated only for the period between consecutive scans, is grounded in this clinical framework.

## 2.2 Vocabulary

### *2.2.1 Python*

Python is a high-level, interpreted, dynamically typed programming language widely used in data science, machine learning, and web development. It serves as the primary development language for the entire NutriAI backend, including the Flask application server, the preprocessing pipeline, and the recommendation engine. Its extensive ecosystem of scientific libraries — including pandas, NumPy, and scikit-learn — makes it uniquely suited to projects that combine data engineering with machine learning.

### *2.2.2 Flask*

Flask is a lightweight Python web framework built on the WSGI (Web Server Gateway Interface) specification. Unlike heavier frameworks such as Django, Flask follows a minimalist design philosophy, providing core routing, request/response handling, and application context management without imposing a specific database layer or templating engine. In NutriAI, Flask serves as the API gateway, exposing over thirty REST endpoints that the frontend JavaScript application consumes. Endpoints are protected with JSON Web Token (JWT) authentication, and cross-origin requests are handled through the Flask-CORS extension.

### *2.2.3 PostgreSQL*

PostgreSQL is an open-source, ACID-compliant relational database system known for its robustness, support for advanced data types, and extensibility. NutriAI uses PostgreSQL to persist all application data, including user accounts, profiles, meal logs, ratings, chat history, InBody tests, BMI entries, and the image cache. The psycopg2 library provides the Python-to-PostgreSQL connection, with the RealDictCursor adapter used to return query results as Python dictionaries for straightforward JSON serialisation.

### *2.2.4 JWT (JSON Web Token)*

JSON Web Tokens are a compact, URL-safe standard for representing claims securely between two parties. In NutriAI, a signed JWT is issued to the user upon successful login or registration.

Every subsequent API request must include this token in the Authorization header, where the token_required decorator validates the signature and extracts the embedded user_id before passing control to the route handler. Tokens are signed using the HS256 algorithm with a secret key and expire after 72 hours, balancing security with user convenience.

### *2.2.5 Groq API and LLaMA 3.3-70B*

Groq is a cloud inference platform that provides extremely fast token generation for open-weight large language models through dedicated Language Processing Units (LPUs). NutriAI uses the Groq API to run the LLaMA 3.3-70B-Versatile model, which powers three distinct AI features: the conversational AI chef chatbot, the InBody body composition assessment generator, and the dashboard nutritional insight generator. The model is invoked with carefully crafted system prompts that embed the user's full nutritional context — daily targets, consumed macros, remaining macros, dietary preferences, and meal history — ensuring that responses are consistently personalised rather than generic.

### *2.2.6 User-User Collaborative Filtering*

User-user collaborative filtering is a memory-based recommendation technique that identifies users with similar item preferences and recommends items those users rated highly that the target user has not yet encountered. In NutriAI, this approach is implemented in the recommend_similar_users function through the _find_similar_foodcom_users helper, which scans the full Food.com review dataset for users who have rated the same recipes as the current NutriAI user. Similarity is scored based on rating agreement: two users who rated the same recipe similarly receive a bonus, while users who rated it oppositely receive a penalty. The top sixty most similar Food.com users are selected, and recipes they rated four stars or above form the Tier 2 candidate pool for the Similar Users recommendation tab, after excluding items the current user has already encountered.

### *2.2.7 TF-IDF (Term Frequency–Inverse Document Frequency)*

TF-IDF is a numerical statistic that reflects the importance of a term within a document relative to a corpus. In the NutriAI recommendation engine, a TF-IDF vectoriser with a vocabulary of 12,000 terms and bigram support is fitted on each recipe's content_features field, which concatenates the recipe name, description, category, ingredient list, and keywords into a single text string. The resulting sparse matrix is used in the content-based filtering component: for users with a rating history, a rating-weighted average of their liked recipes' TF-IDF vectors forms a profile vector compared via cosine similarity against every candidate; for new users with no ratings, similarity is instead computed directly between their seed recipes and candidates

### *2.2.8 MiniBatchKMeans*

MiniBatchKMeans is a scalable variant of the k-means clustering algorithm that processes the training data in small random batches rather than the full dataset at each iteration, substantially reducing memory requirements and computation time. In the preprocessing pipeline, MiniBatchKMeans is used to cluster recipes classified as "other" by the category assignment logic,

grouping them into semantically coherent categories based on a combined TF-IDF and numerical feature matrix. The optimal number of clusters is selected by maximising the silhouette score over a candidate range. A second, separate MiniBatchKMeans pass is run in the recommendation module itself at data-load time, clustering recipes over an ingredients-only TF-IDF space to derive a `recipe_type` label for every recipe; this label drives the diversity/quota system in the hybrid recommendation engine, ensuring that meal suggestions are not dominated by minor variations of the same dish.

#### *2.2.9 pandas*

pandas is a Python library providing DataFrame and Series data structures optimised for tabular data manipulation and analysis. The preprocessing pipeline makes extensive use of pandas for reading the raw CSV files, applying column transformations, parsing R-style vector notation, filtering rows, computing per-serving nutritional values, and performing group-by aggregations. The recommender_app.py module also uses pandas DataFrames to index recipes by RecipeId for efficient lookup during scoring.

#### *2.2.10 NumPy*

NumPy is the foundational library for numerical computing in Python, providing an efficient N-dimensional array type and a comprehensive set of mathematical functions. In NutriAI, NumPy is used extensively throughout the recommendation engine for cosine similarity computation, goal score normalisation, vectorised adherence scoring, and the population of sparse matrices used in TF-IDF operations. The logarithmic popularity bonus applied in hybrid scoring (numpy.log1p) is a practical example of its integration.

#### *2.2.11 joblib*

joblib is a Python library designed for efficient serialisation and deserialisation of large Python objects, particularly those containing NumPy arrays. In NutriAI, joblib is used to save and load the trained machine learning models — the TF-IDF vectoriser and the clustering model — so that they can be reused across application restarts without retraining. A pickle-based cache is also used to persist the fully preprocessed DATA_CACHE dictionary between server restarts, accelerating application startup time significantly.

#### *2.2.12 bcrypt*

bcrypt is a password hashing library that implements the Blowfish cipher-based hashing algorithm with an adaptive cost factor. User passwords are hashed with bcrypt before storage in PostgreSQL, ensuring that even if the database is compromised, plaintext passwords cannot be recovered. The checkpw function is used at login to securely compare the submitted password against the stored hash without exposing either value.

#### *2.2.13 NLTK*

The Natural Language Toolkit (NLTK) is a Python library for processing and analysing human language data. In the preprocessing pipeline, NLTK's English stopword list is used to filter out

common, low-information words from the ingredient and keyword text fields before TF-IDF vectorisation, ensuring that the vocabulary captures meaningful culinary terms rather than prepositions and articles.

### *2.2.14 rapidfuzz*

rapidfuzz is a high-performance string similarity library implementing the Levenshtein distance and related fuzzy matching algorithms. In the preprocessing pipeline, it is used to match recipe keywords against predefined dietary tag vocabularies — such as "vegetarian," "vegan," and "gluten-free" — with a configurable similarity threshold, accommodating minor variations in phrasing across the dataset (for example, "gluten free" versus "gluten-free").

### *2.2.15 HTML, CSS, and JavaScript*

The NutriAI frontend is built as a set of vanilla HTML pages styled with custom CSS and powered by JavaScript for dynamic interactions. The typography uses DM Serif Display for headings and DM Sans for body text, both loaded from Google Fonts. CSS custom properties (variables) define a consistent design token system — covering primary colours, border radii, shadows, and surface colours — that is applied across all pages. JavaScript handles all API communication, state management, DOM manipulation, chart rendering (via Chart.js), and the chatbot interaction loop without any frontend framework dependency.

### *2.2.16 Chart.js*

Chart.js is a lightweight, open-source JavaScript library for creating interactive and animated data visualisations in the browser using the HTML5 Canvas API. In the NutriAI dashboard, Chart.js renders the weekly calorie bar chart — comparing daily intake against the user's calorie target with a dotted reference line — and the daily macro donut chart showing the percentage contribution of protein, fat, and carbohydrates to total energy intake.

### *2.2.17 Pexels API*

Pexels is a stock photography platform that provides a free REST API for searching and retrieving high-quality, royalty-free images. In NutriAI, the Pexels API is used as an automatic image fallback system for recipes that have no image in the Food.com dataset. When a meal card is rendered with a missing image, the frontend requests the /api/meals/image endpoint, which searches Pexels for a relevant food photograph using the recipe name and meal type as search terms. The result is cached in the PostgreSQL image_cache table to avoid repeat API calls for the same recipe name.

# Chapter 3 : Data Collection

## 3.1 Dataset Overview

The primary data source for the NutriAI recommendation system is the irkaal/foodcom-recipes-and-reviews dataset, hosted on Kaggle. This dataset is a scraped and compiled collection from the Food.com recipe sharing platform, one of the largest publicly available collections of community-submitted recipes and reviews on the internet. The dataset comprises two separate CSV files that serve distinct roles in the pipeline.

The first file, recipes.csv, contains structured information about each recipe, including a unique RecipeId, recipe name, author details, publication date, preparation and cooking time in ISO 8601 duration format, serving size, ingredient quantities, ingredient names, step-by-step cooking instructions, and nutritional information covering calories, protein, fat, carbohydrates, fibre, sugar, saturated fat, cholesterol, and sodium. The recipe file also contains a free-text Description field, a RecipeCategory field, and a Keywords field stored in R-style vector notation (e.g., c("dinner", "quick", "easy")) that encodes multiple descriptive tags per recipe. Each recipe also has an Images field containing a list of URL strings pointing to photographs of the finished dish.

The second file, reviews.csv, contains user ratings submitted for individual recipes. Each row records the AuthorId of the reviewer, the RecipeId being reviewed, the numeric Rating on a scale of one to five, optional free-text Review content, and submission timestamps. This rating matrix is used to build a user-item interaction map, compute per-recipe popularity scores, and power the user-user collaborative filtering component in the Similar Users recommendation tier.

## 3.2 Raw File Structure

In its raw form, the recipes.csv file contains approximately 522,000 rows and 21 columns. The reviews.csv file contains approximately 1.4 million rows. The combined dataset represents one of the largest open recipe and review collections available for academic research, providing a rich and diverse foundation for building and evaluating a nutrition recommendation system.

Several important characteristics of the raw data required careful handling during preprocessing. List-type columns — including RecipeIngredientParts, RecipeIngredientQuantities, RecipeInstructions, Keywords, and Images — are serialised in R's c() vector notation rather than standard JSON arrays, necessitating a custom parser. Duration fields for CookTime, PrepTime, and TotalTime are stored in ISO 8601 format (e.g., "PT30M" for thirty minutes), requiring parsing with the isodate library. Nutritional values represent totals for the entire recipe rather than per serving, meaning they must be divided by RecipeServings to obtain meaningful per-serving figures. Additionally, RecipeServings values of zero were present in a small number of rows and required removal.

### 3.3 Data Challenges and Rationale

Several characteristics of the raw dataset posed significant challenges that motivated the design of the preprocessing pipeline. First, the nutritional values in the raw dataset are highly noisy, with a substantial number of recipes exhibiting nutritionally implausible values such as 50,000 calories per serving, zero protein with high calorie counts, or macro energy totals that are wildly inconsistent with the reported calorie value. These anomalies arise from the community-sourced nature of the data, where recipe creators entered values manually without validation.

Second, the dataset contains a large proportion of desserts, confectionery, and high-calorie baked goods that are not suitable for a nutrition management application targeting health-conscious users. Including these recipes in the recommendation pool would distort goal-score rankings, particularly for the weight-loss goal profile.

Third, the ingredient quantities in the raw data are given as totals for the entire recipe rather than per serving. A recipe serving four people with two cups of flour would show "2 cups" as the ingredient quantity, which would be misleading to a user expecting a single-serving ingredient list. Normalising these quantities to per-serving amounts required parsing the quantity strings as mixed fractions, dividing by the serving count, and reformatting the result as a human-readable fraction string.

Fourth, many recipes have no associated image, either because the recipe author did not upload one or because the image URL is no longer accessible. An application that renders empty image slots for a significant fraction of its meal cards would appear unprofessional and reduce user engagement. This challenge motivated the Pexels API fallback system implemented at the application level.

In addition to Missing meal-types and Extreme rating sparsity .

These challenges are addressed systematically in the preprocessing pipeline described in Chapter 4, which applies a series of filters, transformations, and enrichments to produce a clean, normalised, and nutritionally coherent dataset suitable for model training and real-time recommendation.

# Chapter 4 : Data Preprocessing Pipeline

## 4.1 Introduction

The preprocessing pipeline is implemented in `preprocess_foodcom_irkaal.py` and constitutes one of the most technically involved components of the NutriAI system. It transforms the raw Food.com CSV files into a clean, feature-rich, model-ready dataset through twelve sequential processing stages: loading, structural cleaning, quality filtering, feature engineering, clustering of uncategorized recipes, duplicate removal, ingredient unit prediction, per-serving quantity validation, a final high-quantity safety filter, review cleaning, TF-IDF model construction, and report generation. The pipeline is deterministic — every stochastic component (the TF-IDF sampling, the MiniBatchKMeans clustering) is seeded with a fixed random state, so re-running it on the same raw data reliably reproduces the same output. All outputs are written to disk in the `data/processed/`, `models/`, and `reports/` directories.

## 4.2 Structural Cleaning

The first processing stage loads both CSV files and applies structural cleaning to the recipes DataFrame. RecipeId and AuthorId columns are coerced to integer types, and rows with null RecipeId values are discarded. Text columns — Name, Description, AuthorName, RecipeCategory, and RecipeYield — are standardised to string type with whitespace trimmed and missing values replaced with appropriate defaults.

The RecipeServings column is converted to a numeric type, and records with a serving count of zero are removed, as dividing nutritional totals by zero during per-serving normalisation would produce invalid values. The remaining serving counts are clipped to a maximum of 200 to eliminate obvious data entry errors.

Duration fields (CookTime, PrepTime, TotalTime) are parsed from ISO 8601 notation into integer minute values using the isodate library. A fallback regex parser handles cases where the isodate library raises an exception, extracting day, hour, and minute components individually. Duration values are clipped to a maximum of 1,440 minutes (24 hours) and null values are filled with the column median.

List-type columns (Keywords, RecipeIngredientQuantities, RecipeIngredientParts, RecipeInstructions, Images) are parsed from their R-style `c("item1", "item2")` serialisation into Python lists using a custom `parse_r_vector` function. The function handles edge cases including empty vectors serialised as `"character(0)"`, null values, and single-item lists that are represented as plain strings without the `c()` wrapper.

This same stage also removes recipes classified as desserts, confectionery, or baked goods, using a set of 42 keywords matched against the RecipeCategory field, the recipe Name, and the Keywords list, and applies an initial per-serving calorie cap of 1,500 kilocalories to eliminate the most obviously corrupted entries before deeper quality checks run.

## 4.3 Quality Filtering

Following structural cleaning, a dedicated quality filter removes recipes that do not meet the nutritional and content standards required for a health-focused application. A per-serving calorie filter removes recipes where the estimated calories per serving — computed by dividing the total Calories value by RecipeServings — falls below fifteen kilocalories (indicating almost certainly an incomplete or error-prone entry) or exceeds 1,500 kilocalories (indicating either a data error or a recipe intended as a full-day intake rather than a single meal).

A nutritional completeness check discards recipes where three or more of the four core macro columns — Calories, ProteinContent, FatContent, and CarbohydrateContent — contain zero or null values, which would indicate that the recipe has no usable nutritional information for macro tracking purposes.

A macro energy balance check subsequently removes recipes where the combined energy derivable from the reported protein, fat, and carbohydrate values is inconsistent with the reported calorie total — either less than ten percent of the expected macro-derived energy or more than five times the reported calorie value — identifying recipes with fundamentally corrupted nutritional data.

### 4.4 Feature Engineering

The feature engineering stage enriches the cleaned dataset with derived columns that power both the recommendation engine and the application's dietary filtering system. The non_vegetarian flag is computed by applying a regex pattern over the concatenated ingredient list, matching against a vocabulary of common animal-derived protein sources including chicken, beef, pork, fish, shrimp, lamb, bacon, sausage, and seafood. Dietary tag columns — vegetarian, vegan, gluten_free, dairy_free, nut_free, low_carb, and high_protein — are computed by fuzzy-matching the recipe's Keywords list against predefined vocabulary sets using the rapidfuzz library with a similarity threshold of 82.

The meal_type column is assigned through a two-phase classification strategy. In the first phase, a fast keyword-matching heuristic is applied over the concatenated RecipeCategory and Keywords fields. Recipes containing breakfast-related terms (oatmeal, pancake, waffle, cereal, omelette, toast, muffin, granola, brunch) are classified as breakfast; those containing soup, salad, sandwich, wrap, or lunch are classified as lunch; main-dish, dinner, stew, casserole, or entree as dinner; and snack, appetizer, smoothie, or beverage as snack.

Recipes that do not match any keyword pattern are left unclassified and passed to a second phase that resolves them using a k-nearest-neighbours majority vote in TF-IDF space, rather than an external API call. A lightweight TF-IDF model is fitted over the combined name, category, ingredient, and keyword text of every recipe, including those already labelled by the keyword pass. For each unclassified recipe, the 15 nearest neighbours among the keyword-labelled recipes are found by cosine similarity, and the majority meal_type among those neighbours is assigned. This approach was adopted in place of an earlier design that queried the Groq LLM API (`llama3-8b-8192`) for ambiguous cases, since the neighbour-vote method requires no external API calls, incurs no per-request cost or latency, and produces more content-consistent labels — for example, a recipe with no explicit breakfast/lunch/dinner/snack keyword is classified according to the meal type of the recipes it most closely resembles in ingredients and description, rather than defaulting

arbitrarily. A final safety pass replaces any remaining unclassified meal_type values with "dinner" to guarantee no recipe reaches the recommendation engine without a meal-type label.

The category column is assigned using a similar keyword-matching approach over a broader set of category labels (breakfast, appetizer, main_dish, side_dish, beverage, salad, soup, bread); recipes matching none of these fall into an "other" category, which is later resolved separately through clustering.

Per-100-kilocalorie protein and fibre density scores are computed to provide normalised measures of nutritional quality. Goal score columns (goal_score_lose, goal_score_maintain, goal_score_build) are computed for each goal profile by taking a weighted linear combination of normalised nutritional features. The lose goal applies a strong negative weight to calories (−2.0) while rewarding protein (+2.0) and fibre (+1.5), and applies smaller negative weights to fat (−1.0), carbohydrates (−0.5), and sugar (−1.0), reflecting the nutritional priorities of a calorie-deficit plan that preserves satiety and lean mass. The maintain profile applies moderate positive weights to protein (+1.0) and fibre (+1.0) only, with all other weights set to zero. The build profile rewards calories (+1.5), protein (+2.5), carbohydrates (+1.0), and fibre (+0.5), with a small positive weight on fat (+0.5), to favour energy-dense, high-protein meals suited to a muscle-building surplus.

Finally, a content_features text field is created by concatenating the recipe name, description, category, ingredient text, and keyword text into a single string per recipe. This field is used as the input to the TF-IDF vectoriser, enabling content-based similarity computation between recipes.

### 4.5 "Other"-Category Clustering

Recipes that do not match any keyword-defined category in Section 4.4 are assigned to an "other" category, which is then resolved through unsupervised clustering rather than left as a single undifferentiated group. The cluster_other_recipes function builds a combined feature matrix from four sources: a 300-term TF-IDF vectorisation of cleaned ingredient text, a 150-term TF-IDF vectorisation of keyword text, a 50-term TF-IDF vectorisation of category text, and a set of standardised numeric nutrition features (calories, protein, fat, carbohydrate, fibre, cook time, ingredient count). These four blocks are combined with fixed weights of 0.55, 0.25, 0.10, and 0.10 respectively before clustering. A MiniBatchKMeans model is then fit on this combined matrix, with the number of clusters k chosen automatically by testing a range of candidate values and selecting the one that maximises silhouette score — a metric quantifying how well-separated the resulting clusters are. The fitted KMeans model, all three TF-IDF vectorisers, the numeric feature scaler, and the chosen k are serialised together to clustering_model.joblib for reuse without retraining.

### 4.6 Duplicate Recipe Removal

The Food.com dataset contains a substantial number of near-identical recipes submitted under the same or very similar names by different authors, which would otherwise cause the same dish to appear multiple times in a user's recommendation feed. The drop_duplicate_recipes function

addresses this by grouping recipes on a normalised, lowercased, whitespace-trimmed version of the Name field, and retaining only a single representative recipe from each group. The selection of the retained recipe follows a three-level priority ordering: the recipe with the highest AggregatedRating is preferred first, with missing ratings treated as zero; ties are broken by the highest ReviewCount, ensuring that the most community-validated version of a recipe survives; and any remaining ties are resolved by selecting the recipe with the lowest RecipeId, which corresponds to the oldest, most established submission. This deduplication step is applied after clustering but before unit prediction, since duplicate removal reduces the volume of data that the more computationally expensive ingredient unit prediction and validation stages must process.

### 4.7- Ingredient Unit Prediction

Once the dataset has passed quality filtering and feature engineering, each recipe's ingredient list must be annotated with a measurement unit before quantities can be normalised to a per-serving basis. The raw Food.com data stores ingredient quantities and ingredient names as two parallel lists (RecipeIngredientQuantities and RecipeIngredientParts) but does not separately store the unit of measurement — a quantity of "2" next to the ingredient "chicken breast" could mean two whole breasts, two pounds, or two cups depending on context. The `extract_ingredient_units` function resolves this ambiguity through a three-tier cascading strategy.

The first tier applies a compiled regular expression (`_UNIT_RE`) over the combined quantity-and-ingredient text, searching for an explicit unit token such as "cup," "tbsp," "tsp," "oz," "lb," "g," "kg," "ml," or "clove" immediately following a numeric value. This regex handles both metric and imperial units, along with their plural and abbreviated forms, and is the most reliable detection path since it relies on information actually present in the source text.

When no explicit unit can be found in the text, the second tier applies a quantity-based heuristic: small fractional quantities such as 0.25 or 0.5 are mapped to "tsp" or "pinch" when the ingredient is known to commonly appear in those units (e.g., spices), while quantities of two or more are mapped to "whole" for ingredients known to be countable items such as eggs or onions.

If both prior tiers fail to resolve a unit, the third tier consults a hand-curated knowledge base (`_ING_UNIT_KB`) of 95 ingredient-to-unit mappings derived from common culinary conventions — for example, "water," "broth," and "stock" default to "cup"; "salt," "pepper," and most ground spices default to "tsp"; "chicken," "beef," and "pork" default to "lb"; and "egg," "onion," and "garlic" default to "whole." Ingredients not found in any tier fall back to a generic "whole" unit.

A final plausibility remapping step (`_sanity_remap_unit`) corrects systematically implausible unit assignments. For example, if an ingredient is predicted to require three or more cups of shredded cheese, the function recognises this as far more likely to represent ounces and remaps the unit accordingly, using a table of remapping rules (`_PLAUSIBILITY_RULES`) that targets common over-prediction patterns for cheese, spices, vinegar, butter, and sweeteners.

Code snippet for ingredient units

```python
def extract_ingredient_units(qty_list: list, ing_list: list) -> list[s
    if not isinstance(qty_list, list):
        qty_list = []
    if not isinstance(ing_list, list):
        ing_list = []
    n = max(len(qty_list), len(ing_list))
    qty_list = list(qty_list) + [""] * (n - len(qty_list))
    ing_list = list(ing_list) + [""] * (n - len(ing_list))

    units = []
    for qty_raw, ing_raw in zip(qty_list, ing_list):
        qty_str = str(qty_raw).strip() if qty_raw is not None else ""
        ing_str = str(ing_raw).strip() if ing_raw is not None else ""

        unit = _extract_unit_from_combined(qty_str, ing_str)
        if unit:
            units.append(_sanity_remap_unit(qty_str, ing_str, unit))
            continue
```

### 4.8 Per-Serving Quantity Normalization

Because the raw dataset expresses ingredient quantities as totals for the entire recipe rather than per individual serving, the normalize_quantities_per_serving function converts every quantity into a per-serving amount by dividing the parsed numeric value by the RecipeServings count. The _frac_to_float helper first converts the raw quantity string — which may be a whole number, a simple fraction such as "1/2", or a mixed number such as "1 1/2" — into a Python float for arithmetic. After division, the resulting per-serving value is reformatted back into a cooking-friendly fraction string using _format_qty_for_display. This function maps the decimal remainder of the quantity to the nearest common cooking fraction from a fixed list (1/8, 1/4, 1/3, 3/8, 1/2, 5/8, 2/3, 3/4, 7/8) within a four-percent tolerance, ensuring that a recipe instruction reads "1 1/2 cup" rather than an awkward decimal such as "1.4998 cup." Quantities below 0.08 are rounded to the smallest practical fraction, 1/8, rather than being displayed as effectively zero. This normalisation is applied consistently across the entire dataset before being written to recipes_processed.csv, meaning that every ingredient quantity shown to a NutriAI user — whether in the meal detail panel, the recommendation card, or the portion-scaling modal — already reflects a single serving and can be linearly scaled by the frontend's serving multiplier without any further backend computation.

Code snippet for normalizing quantities per serving

```python
def normalize_quantities_per_serving(
    qty_list: list,
    servings: float,
) -> list[str]:
    if not isinstance(qty_list, list):
        return []

    servings = max(float(servings or 1), 1.0)
    result = []

    for qty_raw in qty_list:
        qty_str = str(qty_raw).strip() if qty_raw is not None else ""
        qty_float = _frac_to_float(qty_str)

        if qty_float <= 0 or qty_float != qty_float:  # 0, negative, o
```

### 4.9-Per-Serving Plausibility Validation

Even after per-serving normalisation, a number of recipes retain ingredient quantities that are physically implausible for a single serving — for example, eight pounds of chicken or fourteen cups of beans per serving, which typically indicate that the RecipeServings field itself was entered incorrectly by the recipe author. The validate_per_serving_ingredient_quantities function applies a comprehensive table of per-serving maxima (_PER_SERVING_MAXIMA) covering thirteen ingredient categories — including meats, ground meats, seafood, legumes, grains, flour, sugar, liquids, condiments, oils, salt, spices, and cheese — each paired with a specific unit and an upper bound informed by realistic single-serving cooking practice. A complementary check, _recipe_has_incompatible_ingredients, identifies dish-level inconsistencies rather than purely numeric anomalies. This check flags recipes where a savoury protein dish (containing chicken, beef, pork, fish, or similar) also contains an implausible quantity of sugar — more than half a cup per serving — which would indicate an unusual or erroneous combination. It similarly flags any recipe with more than four teaspoons of salt or one and a half cups of butter or oil per serving, regardless of dish type, since these values exceed any realistic single-serving threshold. Recipes that fail either check are removed from the dataset entirely rather than corrected, since a quantity error at this scale typically reflects a deeper data quality problem (such as an incorrect serving count) that cannot be reliably repaired through unit remapping alone.

### 4.10 High-Quantity Ingredient Filter

As a final data-quality safeguard immediately after per-serving plausibility validation, the `drop_high_quantity_recipes` function removes any recipe in which at least one ingredient's per-serving quantity — already normalised by the previous step — exceeds the constant `MAX_INGREDIENT_QUANTITY (10)`. This threshold-based filter is deliberately unit-agnostic: a quantity of 11 cups of broth, 15 cloves of garlic, or 50 links of sausage per serving would all be caught regardless of the ingredient's canonical unit, acting as a catch-all backstop for edge cases that slip through the unit-aware maximum tables in Section 4.8. The function iterates over each recipe's `RecipeIngredientQuantities` list, converts every entry to a float via `_frac_to_float`, and flags the recipe for removal if any value exceeds 10. Before-and-after row counts are logged at INFO level so the number of recipes eliminated by this step is visible in the pipeline's audit output. This step was introduced in version 6 of the preprocessing script and represents the most recently added quality gate in the pipeline.

### 4.11 TF-IDF Model Training

With the recipe dataset fully cleaned, deduplicated, and validated, the pipeline trains the content-based model that underpins the recommendation engine. The `build_tfidf` function fits a `TfidfVectorizer` with a maximum vocabulary of 12,000 terms, English stop-word removal, sublinear term-frequency scaling, and support for both unigrams and bigrams, on a random sample of up to 40,000 recipes drawn from the `content_features` column described in Section 4.4. Sampling rather than fitting on the full corpus keeps vectoriser training tractable while still producing a vocabulary broad enough to cover the full diversity of the dataset. The trained vectoriser is serialised to `recommendation_models.joblib` using `joblib` and is loaded once at Flask application startup, where it is used to transform the full recipe corpus into a sparse TF-

IDF matrix held in memory for cosine similarity computation throughout the lifetime of the server process.

Code snippet for tf_idf model build

```
# ═══════════════════════════════════════════════════════════════
#  STEP 6 — BUILD TF-IDF CONTENT MODEL
# ═══════════════════════════════════════════════════════════════

def build_tfidf(recipes: pd.DataFrame) -> TfidfVectorizer:
    logger.info("🔨 Fitting TF-IDF …")
    vec = TfidfVectorizer(max_features=12_000, stop_words="english",
                          ngram_range=(1,2), sublinear_tf=True, dtype=np.float32
    n   = min(40_000, len(recipes))
    idx = np.random.default_rng(42).choice(len(recipes), size=n, replace=False)
    vec.fit(recipes.iloc[idx]["content_features"].fillna(""))
    logger.info(f"  Vocabulary: {len(vec.vocabulary_):,} terms")
    return vec
```

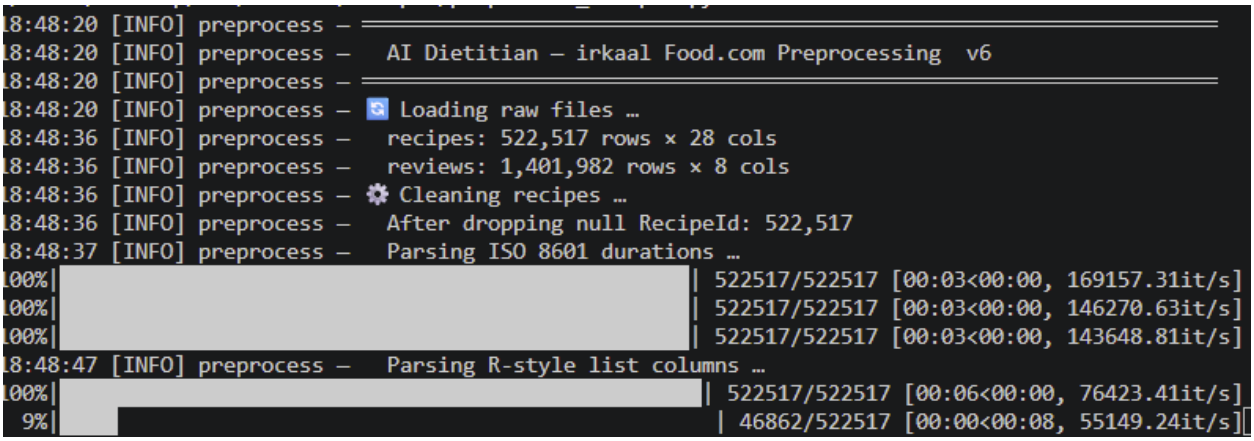
### 4.10 Pipeline Summary

The complete preprocessing pipeline executes eleven sequential stages from raw CSV ingestion to final model persistence:

1. 4.2 Structural Cleaning
2. 4.3 Quality Filtering
3. 4.4 Feature Engineering
4. 4.5 "Other"-Category Clustering *(insert here)*
5. 4.6 Duplicate Recipe Removal *(renumbered, moved up)*
6. 4.7 Ingredient Unit Prediction *(renumbered, moved down)*
7. 4.8 Per-Serving Quantity Normalization
8. 4.9 Per-Serving Plausibility Validation
9. 4.10 High-Quantity Ingredient Filter *(moved up, right after 4.9 since it runs immediately after)*
10. 4.11 TF-IDF Model Training *(moved to the end, since it's the last stage before persistence)*

Each stage logs detailed before-and-after row counts to the console, allowing the volume of data removed at each step to be audited. The pipeline concludes by writing the fully processed recipes and reviews to `recipes_processed.csv` and `reviews_processed.csv` respectively, persisting the clustering model and the TF-IDF vectoriser bundle, and saving a set of diagnostic visualisations — including category distribution, nutrition histograms, and a data quality audit chart — to the `reports/` directory for documentation purposes.

Figure 1: Preprocess Snippet Output




# Chapter 5 : System Architecture

## 5.1 Overview

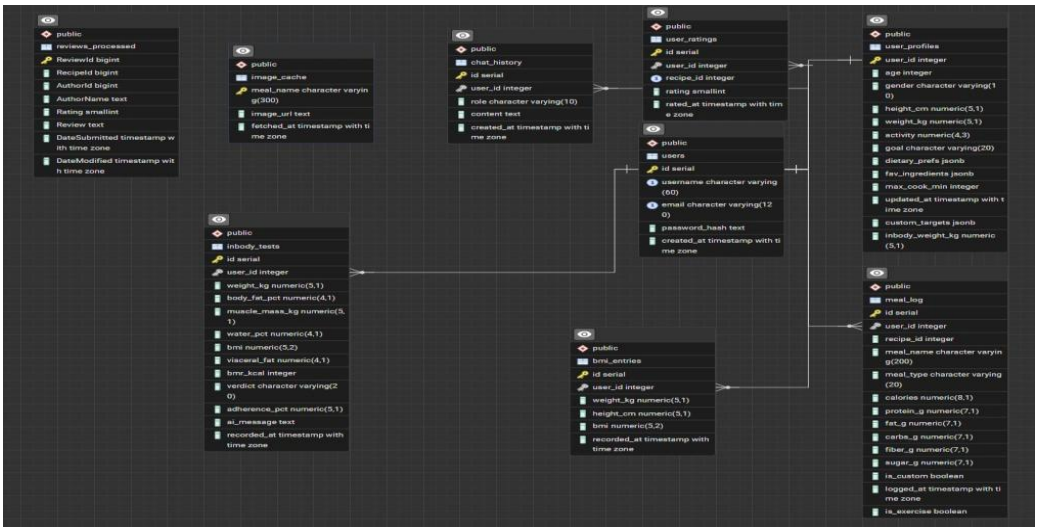
NutriAI follows a classic three-tier web application architecture, separating the presentation layer, the application logic layer, and the data persistence layer into distinct, independently maintainable components. The presentation layer consists of a set of vanilla HTML, CSS, and JavaScript pages served as static files and rendered directly in the browser, with no client-side framework or build step required. The application logic layer is a single Flask process (app.py) that exposes a REST API consumed exclusively over HTTP by the frontend JavaScript, and which in turn coordinates calls to the recommendation engine module (recommender_app.py) and the Groq large language model API. The data persistence layer is a PostgreSQL relational database that stores all user-generated data, while the static, pre-processed recipe catalogue produced by the preprocessing pipeline is loaded once at startup into an in-memory cache (DATA_CACHE) rather than being queried from the database on every request. This separation of a slow-changing, read-heavy recipe catalogue (held in memory as pandas DataFrames) from a fast-changing, write-heavy user activity log (held in PostgreSQL) is a deliberate architectural decision that allows the recommendation engine to perform vector similarity computations and collaborative filtering predictions with sub-second latency, since no database round-trip is required to access the recipe corpus during scoring. In addition to the primary end-user flow, the application supports a secondary account type — coaches — who authenticate through a parallel credential and token system and can view the meal logs and progress data of users who have subscribed to them and been accepted.

## 5.2 Database Schema

The PostgreSQL schema is defined and created idempotently by the init_db function in app.py, which issues a single multi-statement CREATE TABLE IF NOT EXISTS block on application startup, along with a small number of ALTER TABLE ADD COLUMN IF NOT EXISTS migration statements that allow the schema to evolve across versions without requiring a destructive reset. The schema comprises the following tables: • users — stores account credentials: a unique username, a unique email, a bcrypt password_hash, and an account creation timestamp. • user_profiles — stores one profile row per user, including age, gender, height, manually entered weight, a separately tracked inbody_weight_kg, activity multiplier, goal, dietary_prefs and fav_ingredients (both stored as JSONB), max_cook_min, and a custom_targets JSONB field used

to override calculated macro targets when an InBody-suggested adjustment has been accepted. • meal_log — the central activity table, recording every meal and exercise entry: meal name, meal type, all six macro values, an is_custom flag distinguishing dataset recipes from manually entered meals, an is_exercise flag used to represent calorie-burning activity as a negative-calorie row, and a logged_at timestamp. • user_ratings — a unique (user_id, recipe_id) pair table storing the 1–5 star rating a user has given a specific recipe, which feeds directly into the collaborative filtering component of the recommender. • chat_history — stores every user and assistant message exchanged with the AI chef chatbot, enabling conversation context to be reloaded across sessions. • bmi_entries and inbody_tests — two complementary body-tracking tables; bmi_entries stores lightweight weight/height/BMI check-ins, while inbody_tests stores the full InBody scan payload (body fat percentage, muscle mass, water percentage, visceral fat, BMR) together with the computed verdict, adherence percentage, and the Groq-generated ai_message for that test. • image_cache — a simple key-value table mapping a lowercased meal name to a cached Pexels image URL, preventing repeated external API calls for the same recipe name across all users. A third body-tracking table, bmi_evaluations, stores self-reported BMI progress assessments — including the previous and new BMI, the change over a given period, free-text body observations, and an AI-generated verdict and headline — as an alternative to the InBody-based tracking in inbody_tests. Two further tables support the coaching feature: coaches, which holds coach account credentials (username, email, password_hash) independently of the users table, and coach_subscriptions, a (coach_id, user_id) pair table with a status field driving a request/accept/decline/removal workflow, indexed on both coach_id and user_id for fast lookups in either direction. Foreign key constraints with ON DELETE CASCADE are used throughout, ensuring that deleting a user account automatically removes all of that user's profile, logging, rating, and chat data without orphaned rows. Indexes are created on the most frequently filtered columns — meal_log(user_id, logged_at), user_ratings(user_id), bmi_entries(user_id, recorded_at), inbody_tests(user_id, recorded_at), and chat_history(user_id, created_at) — to keep the date-range and per-user aggregation queries used throughout the dashboard and history endpoints performant as the activity log grows.

Figure 2 : ER Diagram inside PostgreSQL




## 5.3 API Endpoints

The Flask application exposes more than thirty REST endpoints, grouped by functional area. Authentication endpoints (/api/auth/register, /api/auth/login) handle account creation and credential verification. Profile endpoints (/api/profile GET/PUT, /api/profile/targets PUT/DELETE) manage the user's personal details, goal, dietary preferences, and custom nutrition targets. Meal endpoints (/api/meals/dashboard, /api/meals/search, /api/meals/log, /api/meals/today, /api/meals/custom, /api/meals/rate, /api/meals/unrate, /api/meals/image) cover browsing, logging, rating, and image resolution. Recommendation endpoints (/api/recommend/similar, /api/recommend/history, /api/recommend/favorites, /api/recommend/frequent) expose the four distinct recommendation strategies described in Chapter 6. Chat endpoints (/api/chat, /api/chat/history, /api/chat/clear) drive the AI chef assistant. Tracking endpoints (/api/tracker/bmi, /api/tracker/history, /api/inbody/submit, /api/inbody/history, /api/bmi/evaluate) cover body composition monitoring. Finally, dashboard analytics endpoints (/api/dashboard/streaks, /api/dashboard/insights, /api/history/daily, /api/history/summary) supply the data consumed by the charts and achievement widgets on the dashboard page. A separate group of coaching endpoints (/api/coach/auth/register, /api/coach/auth/login, /api/coach/me, /api/coaches, /api/coach/subscribe, /api/coach/my-subscription, /api/coach/unsubscribe, /api/coach/requests, /api/coach/requests/<id>/respond, /api/coach/clients, /api/coach/clients/<id>/meals, /api/coach/clients/<id>/progress, /api/coach/clients/<id>/note) supports coach account management and the coach-to-client relationship, bringing the total endpoint count to over forty-five. Every endpoint that operates on user-specific data is wrapped with the token_required decorator (detailed in Section 5.4), and all routes return JSON responses with a consistent error-handling convention: validation failures return HTTP 400 with a descriptive error field, authentication failures return HTTP 401, and unexpected server-side failures are logged and surfaced as HTTP 503 where an external dependency such as the Groq API is involved.

## 5.4 Authentication Flow

NutriAI uses stateless JSON Web Token (JWT) authentication rather than server-side session storage, which keeps the Flask process horizontally scalable since no session state needs to be

shared between instances. Upon successful registration or login, the generate_token function issues a JWT signed with the HS256 algorithm, embedding the user_id as a claim along with issued-at and expiry timestamps; tokens are valid for 72 hours from issuance. Every protected route is decorated with token_required, which extracts the bearer token from the Authorization header, decodes and verifies its signature using the application's secret key, and re-queries the users table to confirm that the referenced account still exists before attaching the authenticated user's id, username, and email to Flask's request-scoped g object. If the token is missing, expired, or invalid, the decorator short-circuits the request with an appropriate 401 response before the underlying route handler ever executes, ensuring that authentication logic is never duplicated across the thirty-plus protected endpoints.

Coaches authenticate through a structurally separate token space rather than reusing the user-facing flow. The generate_coach_token function issues a JWT carrying a coach_id claim and an explicit role: "coach" field rather than a user_id, and the coach_token_required decorator verifies both the token's validity and that role claim before re-querying the coaches table and attaching the result to g.current_coach. This separation ensures a coach's token can never be accepted by a route expecting a regular user, and vice versa. Authorization for viewing a specific client's data is enforced by a further check, _coach_owns_client, which confirms an accepted (not merely pending) subscription exists between that coach and that user before any of the client's meal or progress data is returned.

### 5.5 Data Flow

A typical user interaction follows a consistent pattern across the application. On page load, the frontend JavaScript reads the stored JWT from localStorage and immediately calls /api/profile to populate the user's daily macro targets; if no token is present, NutriNav.requireAuth() in nav.js redirects the browser to the login page before any other script executes. Subsequent user actions — browsing the meal feed, logging a meal, sending a chatbot message, or submitting an InBody test — each trigger a dedicated fetch() call to the corresponding Flask endpoint, carrying the JWT in the Authorization header. On the backend, each request handler first resolves the authenticated user via g.current_user, then loads that user's profile and computed daily targets through the shared _get_profile and _get_targets helper functions, ensuring that target calculation logic (Mifflin–St Jeor based, with support for InBody-derived custom overrides) is centralised rather than duplicated across endpoints. Read-heavy recommendation requests consult the in-memory DATA_CACHE populated at startup, while all write operations (logging a meal, submitting a rating, recording an InBody test) are persisted immediately to PostgreSQL via the shared q() query helper, which wraps psycopg2 cursor execution and automatically commits each statement. The frontend then re-renders the affected UI components — the macro tracker sidebar, the daily log list, or the chat window — directly from the JSON response, without requiring a full page reload at any point in the application.

## Chapter 6 : Recommendation Engine

### 6.1 Introduction

The recommendation engine, implemented in recommender_app.py, is the analytical core of NutriAI. It is responsible for translating a user's goal, dietary restrictions, favourite ingredients, and historical behaviour into a ranked list of meal suggestions drawn from the preprocessed Food.com catalogue. The module exposes seven distinct recommendation functions, each suited to a different stage of the user journey — from a brand-new user with no history, through an active user with a rich rating history, to a user explicitly searching by ingredients they have on hand.

### 6.2 Data Loading and Caching

The load_data function is the entry point for populating the recommendation engine's working dataset. It first checks for a previously pickled cache file (recommender_cache.pkl) and, if found, validates it against four conditions before reusing it: the cache must not be older than the configured time-to-live (twelve hours), its cache_version constant must match the current code's expected version, the source CSV files must not have been modified more recently than the cache file itself, and two boolean flags (ingredient_units_added, quantities_normalised) confirming the cache was built from data that has passed through the complete preprocessing pipeline must all be true. If any condition fails, the cache is discarded and rebuilt from the processed CSV files on disk. When rebuilding, load_data trusts that dessert removal and the 1,500-calorie-per-serving cap have already been produced upstream by the preprocessing pipeline (Chapter 4). However, it still contains defensive fallback blocks for meal_type, is_condiment, and content_features if those columns are missing from the loaded CSV, and the three goal_score columns are unconditionally recomputed every time rather than only as a fallback. The three boolean cache flags therefore guarantee only that the CSV has passed through the preprocessing pipeline, not that all downstream defensive recomputation paths are disabled. It then constructs a popularity map (the count of ratings each recipe has received), a user-item interaction map (a dictionary from user id to the set of recipe ids that user has rated, used both for collaborative filtering and to exclude already-rated items from future recommendations), and a per-user average rating map. Finally, it loads the previously trained TF-IDF vectoriser from the joblib bundle and transforms the full recipe content_features column into a TF-IDF sparse matrix, ready for cosine similarity computation. The resulting dictionary of DataFrames, models, and lookup structures is assembled into the DATA_CACHE object and, if caching is enabled, persisted back to disk for fast reuse on the next application restart.

Code snippet for load_data() function

```python
def load_data(use_cache: bool = CONFIG["use_cache"]) -> dict:
    if use_cache and CACHE_FILE.exists():
        try:
            with open(CACHE_FILE, "rb") as f:
                cache = pickle.load(f)
            # Check version, TTL, AND that the CSV wasn't modified after cache
            csv_mtime = RECIPES_FILE.stat().st_mtime if RECIPES_FILE.exists() else 0
            cache_mtime = CACHE_FILE.stat().st_mtime
            csv_newer   = csv_mtime > cache_mtime

            if (not csv_newer
                    and  cache_fresh(cache)
```

### 6.3 Dietary Preference Filtering

NutriAI distinguishes between two categories of dietary preference with fundamentally different enforcement semantics. Strict preferences — vegetarian, vegan, gluten-free, dairy-free, and nut-free — are treated as hard constraints: a recipe that does not satisfy an active strict preference is removed from the candidate pool entirely and can never appear in that user's recommendations, regardless of how well it otherwise scores. Soft preferences — high-protein and low-carbohydrate — are treated as ranking boosts rather than filters: a recipe that does not meet the soft criterion is never excluded, but recipes that do satisfy it receive an additive score bonus that increases their likelihood of appearing near the top of the ranked list. The _build_strict_id_set and _filter_strict_prefs helper functions implement the hard-filter behaviour by combining the boolean dietary tag columns computed during preprocessing (Section 4.4) with an AND condition across every active strict preference, and include a defensive fallback: if a particular combination of strict preferences happens to match zero recipes in the dataset — a genuine data coverage gap rather than a code defect — the filter is bypassed for that request rather than returning an empty result set to the user. The _soft_pref_boost and _add_soft_boost_column functions implement the soft-preference scoring, computing a bonus of up to 0.5 for high-protein (scaling linearly with protein content up to 50 grams) and up to 0.4 for low-carbohydrate (scaling linearly as carbohydrate content decreases toward zero).

### 6.4 Goal Score Calculation

Every recipe in the dataset carries three pre-computed goal_score columns — one each for the "lose," "maintain," and "build" goal profiles — initially calculated during preprocessing as a weighted linear combination of min-max normalised nutritional features. Although the numerical weight values in the GOAL_WEIGHTS tables match between preprocessing and the recommender module, the implementations are not identical: preprocessing computes scores using raw nutritional columns such as Calories and ProteinContent with plain min-max normalisation, whereas the recommender recomputes the scores at load time using per-serving columns such as cal_per_serving and prot_per_serving together with 5th/95th percentile-clipped normalisation. The recommender module's implementation therefore deliberately supersedes the preprocessing version when data is loaded. The lose profile penalises calories heavily (weight −2.0) while rewarding protein and fibre (weights +2.0 and +1.5 respectively), reflecting the nutritional priorities of a calorie-deficit plan that preserves satiety and lean mass. The maintain profile applies a neutral weighting across all macros except a moderate protein and fibre bonus. The build profile rewards calories, protein, and carbohydrates (weights +1.5, +2.5, and +1.0) to favour energy-dense, high-protein meals suited to a muscle-building surplus. The calculate_daily_targets

function, called from app.py whenever a user's profile or InBody-derived weight changes, computes the user's personal daily calorie and macro budget using the Mifflin–St Jeor basal metabolic rate equation, multiplied by the user's activity factor and adjusted by a fixed calorie delta per goal (−500 kcal for lose, 0 for maintain, +500 kcal for build). Protein is set at 2.0 grams per kilogram of body weight, fat at twenty-five percent of total calories, fibre at 0.8 grams per kilogram, and sugar capped at twenty percent of total carbohydrate calories.

### 6.5 Content-Based Recommendation

This recommendation implements similarity-based recommendation by computing the cosine similarity between the TF-IDF vectors of a set of "seed" recipes — typically the user's highest-rated meals — and every other recipe in the dataset, then ranking candidates separately against each individual seed rather than collapsing the scores into a single average. This design choice is deliberate: averaging would let a lexically common seed ingredient (for example, "egg," which appears in thousands of unrelated recipes) drown out a rarer seed (for example, "beef" or "pork"), since a recipe weakly similar to every seed could then outscore a recipe strongly similar to just one. Instead, each seed produces its own independently ranked candidate list, and the function selects from these lists in round-robin order — one candidate from the first seed's ranking, then one from the second seed's ranking, and so on — so that every seed is guaranteed to contribute proportionally to the final result regardless of how common its ingredient vocabulary is. To prevent the resulting recommendation list from being dominated by minor variations of the same dish (for example, five near-identical chicken stir-fry recipes), the function tracks how many times a dominant ingredient keyword extracted from each seed recipe has already appeared among the selected recommendations, and skips any further candidate sharing that keyword once a configurable cap (max_per_ingredient, defaulting to five) has been reached.

### 6.6 Hybrid Recommendation

The hybrid_recommend function is the primary recommendation pathway for users with an established rating history, blending content-based similarity, collaborative filtering, goal-aligned nutritional scoring, ditery preferences boosts (if selected) , and a logarithmic popularity bonus through a weighted formula controlled by the hybrid_alpha constant (0.65). In the current implementation, the content-based similarity component is derived from a rating-weighted TF-IDF profile vector representing the user's learned taste preferences, replacing the earlier pathway as the primary similarity mechanism. Collaborative filtering is now also integrated directly into hybrid_recommend itself through real per-candidate item-based k-nearest-neighbour predictions computed by _compute_item_cf_scores and blended as alpha * profile_sims + (1-alpha) * cf_sc_arr. The value 2.5 remains only as a fallback baseline used when a particular candidate recipe lacks sufficient collaborative-filtering neighbours, rather than as a universal constant applied equally to every candidate. For each candidate recipe, the function computes: the personalised collaborative-filtering score described above; the TF-IDF profile similarity score; the recipe's goal score for the user's active goal; a logarithmic popularity bonus (numpy.log1p of the rating count, scaled by 0.08) that nudges well-reviewed recipes upward without allowing sheer popularity to dominate; and the soft dietary preference boost from Section 6.3. Strict dietary filtering is applied before scoring to avoid wasting computation on non-compliant candidates. Diversity across the result list is controlled first by a type-quota system (_compute_type_quotas)

that replaced an earlier flat cap ("no more than a fixed number of any dominant ingredient"): the user's own 4-star and 5-star ratings are used to identify a small number of "favoured" ingredient types, each of which is reserved a calculated number of slots in the final list — scaling with how many distinct types are favoured, but capped so favoured types can never crowd out discovery entirely — while any ingredient type outside this favoured set is limited instead by a lighter, uniform filler cap. Final recommendation selection is then additionally refined through MMR (Maximal Marginal Relevance) re-ranking, which balances recommendation relevance against inter-item diversity to avoid repetitive recommendation lists dominated by near-duplicate recipes. A three-pass safety-net fallback fills any remaining slots — first from the most popular compliant recipes under the same quota rules, and, only if still short, by applying a bounded overflow allowance rather than fully removing quotas altogether. The implementation explicitly limits this overflow through a fixed OVERFLOW_ALLOWANCE = 2 so that favoured-type quotas are never bypassed entirely, even for users with highly restrictive dietary combinations.

Code snippet for hybrid_recommend() function

```
def hybrid_recommend(user_id: str, seed_ids: list, goal: str,
                     meal_type: "str | None", data_cache: dict,
                     top_n: int = 10,
                     daily_targets: "dict | None" = None,
                     already_eaten_ids: "set | None" = None,
                     max_per_ingredient: int = 4,
                     dietary_prefs: dict = None,
                     user_rating_map: dict = None) -> list:
    """
    Hybrid content-based + CF recommendation engine.
    Now respects strict dietary hard filters and soft preference score boosts.
    """

    dietary_prefs = dietary_prefs or {}
    recipes       = data_cache["recipes"]
```

### 6.7 Dashboard Meal Generation

The get_meals_for_dashboard function powers the sixty-meal "Suggested for Today" feed shown on the Meals page, generating fifteen recommendations for each of the four meal types (breakfast, lunch, dinner, snack) in a single call. For users with an existing rating history, each meal type invokes hybrid_recommend filtered to that meal type, while sharing a single mutable shared_quota_state dictionary computed once at the dashboard level and passed into every recommendation call. This design intentionally shares favoured-type quota usage across meal categories rather than granting each meal type an independent quota allocation. For brand-new users with no ratings, a simpler goal-score-plus-popularity ranking is used instead, computed directly within this function rather than delegating to the cold-start path described in Section 6.11. A cross-type deduplication pass is applied after generation: because the underlying candidate pools for different meal types can overlap (particularly when dietary filtering leaves few compliant recipes for a given category), the same recipe could otherwise appear in both the breakfast and lunch sections of the feed. The deduplication logic processes the four meal types in a fixed priority order — breakfast, lunch, dinner, then snack — keeping each recipe only in the first list in which it appears and removing any subsequent duplicate occurrence, which is applied again at the Flask layer in meals_dashboard() together with a per-meal-type calorie cap to ensure the suggested portions remain realistic for that time of day.

### 6.7.1 Daily Recommendation Refresh Logic

To improve recommendation variety and prevent repetitive suggestions, the recommended meals are refreshed every 24 hours. Even when a user has not favourited or rated any meals, the system rotates the suggested meals daily to expose the user to different recipe options over time. For users with existing favourites or ratings, the recommendation logic still prioritises personalised and relevant meals based on their preferences, but the displayed recommendations are periodically refreshed every 24 hours to maintain diversity and improve long-term user engagement.

### 6.8 Similar User Recommendation (Two-Tier)

The recommend_similar_users function implements the most architecturally sophisticated recommendation pathway in the system, designed to remain useful even when NutriAI has very few registered users — a classic cold-start challenge for any peer-based recommender. Tier 1 computes a profile similarity score between the current user and every other NutriAI user based on matching goal (+3.0), BMI proximity within 1.5, 3.0, or 5.0 units (+3.0, +2.0, or +0.5 respectively), activity level proximity, shared dietary preferences, and shared favourite ingredients; users scoring 4.5 or above are considered "similar peers." The function then aggregates the actual PostgreSQL ratings submitted by those similar peers, weighted by their similarity score, to build a candidate recipe pool. If Tier 1 does not produce enough candidates — which is expected in a system with few registered users — Tier 2 falls back to genuine Food.com collaborative filtering: it uses the current user's own recipe ratings to identify Food.com dataset users (via _find_similar_foodcom_users) who rated the same recipes with similar scores, awarding an overlap bonus and an agreement bonus scaled by how closely the two users' ratings align, then surfaces the highly-rated recipes (four stars or above) of those similar Food.com users that the current NutriAI user has not yet tried. This two-tier design means the "Similar Users" recommendation tab can produce meaningful results from the very first user who rates even a single recipe, rather than requiring a large registered user base to become useful — an important design consideration given the practical constraints of a student project with a limited testing population.

### 6.9 History-Based Recommendation

The recommend_from_history function implements a "bring it back" recommendation strategy that resurfaces meals the user has previously eaten or rated frequently. For each recipe present in the user's meal_log, the function computes a composite score combining the logging frequency (weighted 0.5), the user's own star rating for that recipe if one exists or a neutral default of 3.0 otherwise (weighted 0.3), and the recipe's goal score for the user's current goal (weighted 0.2). This pathway is intentionally simple relative to the hybrid and similar-user engines, since its purpose is narrow and well-defined: identify what has worked for this specific user before, rather than discover new content.

### 6.10 Ingredient Search

The search_recipes_by_ingredients function powers both the AI chatbot's "what can I cook with X" capability and the custom meal logging nutrition estimation feature. Given a list of ingredient names extracted from natural language, the function filters the dataset to recipes whose RecipeIngredientParts list contains at least one match, optionally narrowing further by meal type and remaining calorie budget if those constraints are supplied. The combined relevance score blends the proportion of query ingredients matched (weighted 0.65), the recipe's goal score (weighted 0.25), and a favourite-ingredient boost (weighted 0.10), with strict and soft dietary preferences applied identically to the other recommendation pathways for consistency across the application.

### 6.11 Cold Start (New User)

The recommend_new_user function provides the dedicated cold-start pathway invoked when a user has no prior ratings or meal history at all, relying entirely on explicit profile signals rather than any collaborative or content-based similarity computation. Candidates are filtered first by the user's strict dietary preferences, then by their maximum acceptable cooking time if one has been set, and then ranked by a combination of the recipe's goal score, a favourite-ingredient text match bonus, the logarithmic popularity bonus, and the soft preference boost. Should the meal-type-specific filtering leave too few candidates, the function gracefully widens the pool back to the full strict-filtered dataset to guarantee that new users always receive a complete, non-empty set of recommendations on their very first visit to the Meals page.

## Chapter 7 : Backend API (app.py)

### 7.1 Application Configuration

The Flask application is configured through a centralised Config class that consolidates every environment-dependent setting in one location, with sensible defaults supplied for local development and the expectation that production deployments override each value through environment variables. This includes the JWT signing secret and expiry duration (72 hours), PostgreSQL connection parameters (host, port, database name, user, password), the Groq API key and the selected model identifier (llama-3.3-70b-versatile), the Pexels API key used for the image fallback system, and the profile update reminder threshold (seven days), after which the frontend displays a gentle prompt encouraging the user to refresh their weight and activity level so that calculated targets remain accurate.

### 7.2 Database Layer

Rather than introducing a full Object-Relational Mapper, NutriAI uses a thin, explicit database access layer built directly on psycopg2. The get_db function lazily opens a single PostgreSQL connection per request, stored on Flask's application-context-local g object, and the teardown_appcontext hook ensures that connection is always closed at the end of the request regardless of whether the handler succeeded or raised an exception. The shared q() helper function wraps cursor creation, execution, and automatic commit behind a uniform interface that every route handler calls identically, accepting a SQL string, a parameter tuple, and a flag indicating whether the caller expects a single row, all rows, or no return value at all — collapsing what would

otherwise be repetitive boilerplate across thirty-plus endpoints into a single, consistently tested code path.

### 7.3 Authentication Endpoints

The /api/auth/register endpoint validates that a username, email, and password were all supplied and that the password is at least six characters, checks for pre-existing username or email collisions, hashes the password with bcrypt, inserts both the new users row and an accompanying empty user_profiles row in a single registration flow, and returns a signed JWT alongside basic account details so that the frontend can immediately consider the user authenticated without requiring a separate login step. The /api/auth/login endpoint accepts either a username or an email as the identifier, retrieves the matching user record, and verifies the supplied password against the stored bcrypt hash before issuing a fresh token.

### 7.4 Profile Management

The GET /api/profile endpoint returns the user's stored profile fields together with their currently calculated daily macro targets (or the InBody-derived custom targets, if any have been accepted), basic account information, the most recent BMI entry, and a show_update_reminder flag computed by comparing the profile's updated_at timestamp against the configured reminder threshold. The PUT /api/profile endpoint performs an upsert (INSERT … ON CONFLICT DO UPDATE) so that the same endpoint serves both initial profile creation and subsequent edits, and explicitly clears any previously saved custom_targets whenever the profile is manually updated, ensuring that an InBody-suggested target adjustment does not silently persist after the user has changed their underlying weight, goal, or activity level.

### 7.5 Meal Logging

NutriAI supports three distinct meal logging pathways that converge on the same underlying meal_log table. The /api/meals/log endpoint logs a recipe directly from the Food.com dataset by its RecipeId, pulling canonical nutrition values from the in-memory recipe DataFrame rather than trusting client-supplied numbers. The /api/meals/custom endpoint accepts a manually described meal — optionally estimating its nutrition automatically from a supplied ingredient list using search_recipes_by_ingredients (Section 6.10) when explicit macro values are not provided by the caller. The /api/meals/rate and /api/meals/unrate endpoints record or remove a 1–5 star rating, and critically, also update the in-memory DATA_CACHE's user_item_map immediately so that newly submitted ratings are reflected in the very next recommendation request without waiting for a full DATA_CACHE rebuild.

### 7.6 Meal Dashboard

The /api/meals/dashboard endpoint, consumed by the Meals page's "Suggested for Today" feed, temporarily clears the requesting user's entry in DATA_CACHE's user_item_map for the duration of the call — using a try/finally block to guarantee the original value is always restored afterward, even if an exception occurs — so that recipes the user has already rated are not silently excluded from the discovery feed while still being correctly excluded from being logged twice today via the

separate already_eaten_ids set. After calling get_meals_for_dashboard, the endpoint applies a per-meal-type calorie cap (800 kcal for breakfast, 1,000 for lunch, 1,200 for dinner, 500 for snack) and backfills from lower-calorie alternatives if too few meals pass the cap, ensuring portion sizes remain realistic for each time of day.

### 7.7 Search Endpoint

The /api/meals/search endpoint implements free-text recipe search across the full dataset, matching the query substring case-insensitively against the recipe Name field, applying the user's strict dietary filters, and sorting results by their goal score for the user's active profile, returning up to a configurable limit (default twenty, capped at fifty) of formatted recipe objects.

### 7.8 Recommendation Endpoints

Four endpoints expose the recommendation strategies described in Chapter 6 to the frontend: /api/recommend/similar invokes the two-tier similar-user engine after assembling the necessary peer profile and rating data from PostgreSQL; /api/recommend/history invokes the history-based "bring it back" engine; /api/recommend/favorites returns the user's own five-star rated recipes directly, formatted for display without any additional scoring; and /api/recommend/frequent aggregates the meal_log table by meal name to surface the user's most-logged meals along with their averaged nutrition values, independent of whether those meals originated from the Food.com dataset or were manually entered as custom meals.

### 7.9 History Endpoints

The /api/history/daily endpoint returns a paginated, date-grouped view of the user's complete meal log — joined against user_ratings so each entry also reports whether and how the user rated that meal — together with computed daily nutrition totals, supporting up to ninety days of history per request. The /api/history/summary endpoint aggregates the same underlying data into the weekly bar-chart format consumed by the dashboard, alongside the user's BMI history and their top five most-eaten meals.

### 7.10 BMI Tracker

The /api/tracker/bmi endpoint accepts a new weight and height measurement, computes BMI, persists the entry, updates the user's stored weight, and invokes bmi_tips from the recommender module to generate goal-aware coaching advice — including an automatic goal reassignment (for example, switching a "lose" goal to "maintain" once BMI enters the normal range) when the data indicates the user's current goal is no longer appropriate for their measured body composition.

### 7.11 InBody Submit

The /api/inbody/submit endpoint is the most analytically complex route in the application, orchestrating field validation, historical nutrition analysis, verdict computation, and Groq-powered

narrative generation in a single request-response cycle. The full internal logic of this endpoint is described in detail in Chapter 9.

### 7.12 Custom Nutrition Targets

The PUT /api/profile/targets endpoint persists a set of InBody-suggested macro targets into the user_profiles.custom_targets JSONB column, allowing the user to explicitly accept an AI-recommended adjustment to their daily calorie and macro goals; the DELETE variant clears this override and reverts the user to the standard Mifflin–St Jeor calculation, providing an explicit escape hatch back to the formula-driven baseline at any time.

### 7.13 Image Fallback Endpoint

The /api/meals/image endpoint, described in full in Chapter 10 alongside its frontend integration, accepts a meal name and meal type, checks the image_cache table for a previously resolved Pexels photo URL, and — if no cached entry exists — performs a live Pexels API search, persisting the result (or the absence of one) back into the cache so that no recipe name is ever queried against the external API more than once across the lifetime of the application.

## Chapter 8 : AI Chatbot

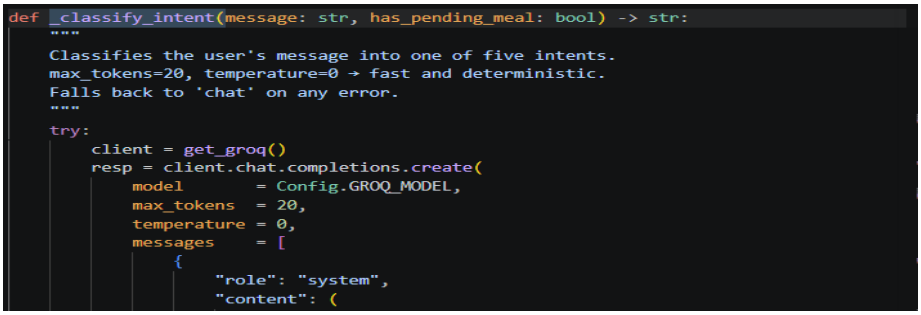
## 8.1 Introduction

The AI chef chatbot is the most user-facing demonstration of artificial intelligence within NutriAI, providing a free-text conversational interface through which users can ask nutrition questions, request meal suggestions tailored to their remaining macro budget, log food and exercise in natural language, and receive guidance on navigating the rest of the application. The chatbot is implemented as a single Flask route, /api/chat, that orchestrates intent classification, specialised short-circuit handling for high-frequency intents, and a full conversational fallback powered by the Groq API.

## 8.2 Intent Classification

Earlier iterations of the chatbot relied on large hand-maintained lists of trigger phrases (such as "I ate," "I just had," or "I ran") to detect user intent, an approach that proved brittle against typos, alternate phrasings, and non-English input. The current implementation replaces this entirely with a single, fast Groq classification call, _classify_intent, configured with a low token budget (twenty tokens) and zero temperature for deterministic output. The classifier is instructed to return exactly one of five intent labels — exercise, log_food, confirm, reject, or chat — and is dynamically informed whether a meal was recently suggested in the conversation (has_pending_meal), since the confirm intent is only semantically valid when there is in fact a pending meal to confirm. If the classification call fails for any reason, or returns a value outside the expected set, the function defaults safely to the chat intent, ensuring the conversation can always continue.

Code snippet for intent classification function



```
def _classify_intent(message: str, has_pending_meal: bool) -> str:
    """
    Classifies the user's message into one of five intents.
    max_tokens=20, temperature=0 → fast and deterministic.
    Falls back to 'chat' on any error.
    """
    try:
        client = get_groq()
        resp = client.chat.completions.create(
            model       = Config.GROQ_MODEL,
            max_tokens  = 20,
            temperature = 0,
            messages    = [
                {
                    "role": "system",
                    "content": (
```

## 8.3 Exercise Logging Flow

When the classifier returns exercise, the handler calls _get_exercise_burn_from_groq, which prompts the LLaMA model with the user's body weight, age, and gender alongside their free-text activity description, instructing the model to apply standard MET-value-based energy expenditure formulas and return a structured JSON object containing the activity name, estimated duration, calories burned, and intensity level. The burned calories are then persisted into meal_log as a negative-calorie row with is_exercise set to true, meaning that every existing query that sums calories for a day automatically reflects the exercise deduction without any special-case logic being required elsewhere in the codebase. The chatbot immediately confirms the logged activity and reports the user's updated remaining calorie budget for the day.

### 8.4 Food Logging Flow

When the classifier returns log_food, the handler calls _get_nutrition_from_groq, which is designed to handle two distinct input patterns: genuine food descriptions (such as "a grilled chicken shawarma wrap"), for which the model estimates full nutritional values from its own training knowledge, and raw macro specifications (such as "add 400 calories and 30g protein"), for which the model is instructed to use the stated numbers exactly rather than attempting to infer a food identity. The resulting meal is inserted into meal_log as a custom entry, and the chatbot responds with a confirmation summarising all six logged macros and the user's updated remaining calorie budget.

### 8.5 Meal Confirmation Flow

When the classifier returns confirm, the handler calls _extract_meal_from_history, which scans backward through up to the six most recent assistant messages in the conversation for a structured [MEAL_DATA] JSON block (described in Section 8.7), allowing a user to confirm a previously suggested meal even if a few unrelated messages — such as a follow-up question about preparation steps — were exchanged in between. If a pending meal is found, its exact macro values are inserted into meal_log directly, guaranteeing that the macros ultimately logged are identical to those shown to the user in the suggestion, with no risk of re-estimation drift between the suggestion and the logged entry.

### 8.6 General Chat Flow

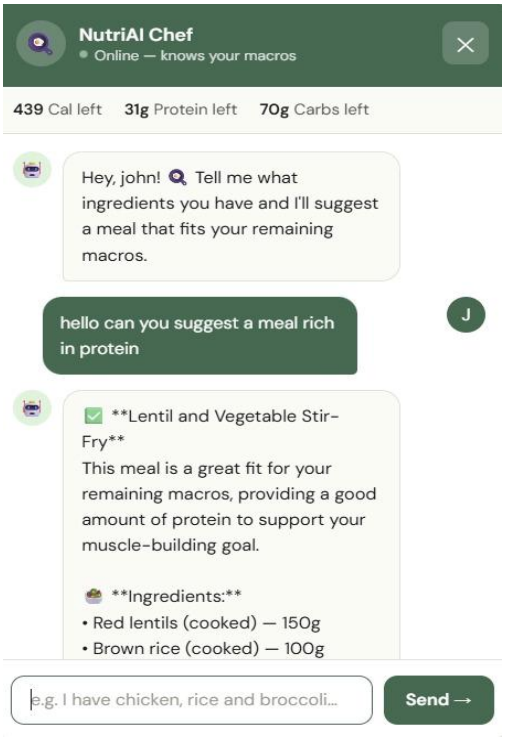
Any message that does not match the exercise, log_food, or confirm intents — including new meal suggestion requests, general nutrition questions, and conversational small talk — falls through to the full Groq chat completion call. The full conversation history (capped at the most recent twenty messages) is supplied to the model alongside the comprehensive system prompt described in Section 8.8, which is rebuilt from the user's current profile, targets, and totals on every single request rather than being cached across turns. On top of this, the user's current remaining macros are additionally injected as an inline context prefix on the latest user message. This duplicates information already present in the freshly built system prompt, but gives the model an emphatic, impossible-to-miss reminder of the live numbers directly adjacent to the user's own words, reducing the chance that a long history of prior turns causes the model to anchor on stale figures mentioned earlier in the conversation.

### 8.7 Meal Suggestion Format

To guarantee that any meal suggested by the chatbot can be reliably logged later — either through an explicit "Add to Log" button rendered in the chat UI or through a natural-language confirmation — the system prompt mandates a strict two-part response format whenever the model proposes a meal. The response must begin with a fenced [MEAL_DATA]…[/MEAL_DATA] block containing a single JSON object with the meal's name, type, all six macro values, and a structured ingredient list with gram weights, followed by a human-readable explanation that repeats the identical numbers in a friendlier, emoji-formatted presentation. The prompt explicitly instructs the

model never to present different numbers in the two sections, since the frontend parses only the JSON block for logging purposes while displaying the prose section to the user.

Fig.4 : NutriAI Chatbot Conversation




### 8.8 System Prompt Construction

The _build_system_prompt function dynamically assembles a comprehensive context block for every chat request, embedding the user's goal, age, gender, height, weight, activity level, dietary preferences, and favourite ingredients; their daily calorie and macro targets alongside what has already been consumed and what remains for the day; a list of the specific meals already logged today; and a static _WEBSITE_GUIDE block describing the structure of the NutriAI application itself, including an explicit list of actions the chatbot can and cannot perform — for example, the chatbot can suggest and log meals but cannot directly remove a logged entry, instead instructing the user to use the dedicated remove button in the daily log sidebar. This combination of dynamic personal context and static application awareness allows the assistant to behave consistently both as a nutrition expert and as an in-app guide, without requiring separate models or prompt pipelines for each role.

## Chapter 9 : InBody Analysis System

### 9.1 Overview

The InBody analysis system is designed to replicate, as closely as practical within a software application, the kind of evidence-based feedback a clinical dietitian would provide after reviewing a client's body composition scan alongside their logged dietary history. Rather than treating each InBody submission in isolation, the system explicitly compares the new scan against the user's previous scan, computes the rate of change in weight, body fat, and muscle mass, cross-references that change against the user's logged nutrition for the exact period between the two scans, and produces both a quantitative adherence score and a qualitative, Groq-generated narrative assessment.

### 9.2 Field Validation

Before any analysis is performed, the _validate_inbody_fields function checks every submitted measurement against a table of physiologically plausible bounds (_INBODY_LIMITS) — for example, weight between 20 and 400 kilograms, body fat between 2 and 70 percent, and BMR between 800 and 6,000 kilocalories — rejecting the submission with a descriptive error message if any value falls outside its expected range. Beyond these per-field bounds, the function applies two cross-field sanity checks: muscle mass can never be equal to or greater than total body weight, and the sum of estimated fat mass (derived from body fat percentage) and muscle mass cannot exceed total body weight by more than a small rounding tolerance, catching internally inconsistent submissions that might otherwise pass individual field validation.

### 9.3 Analysis Engine (_build_inbody_analysis)

The core analysis function determines the relevant comparison period by first checking whether the frontend supplied an explicit baseline test (the values entered in the same submission as a "Test 1" reference), falling back to the most recent previously saved InBody test in the database, and finally falling back to a fixed thirty-day window if no previous test exists at all — ensuring a meaningful analysis can always be produced even for a user's very first scan. Once the period is established, the function queries meal_log for exactly that date range, computing average calorie and macro intake over the full inter-test period as well as over just the most recent seven days, a logging consistency percentage (days with at least one logged meal divided by total days in the period), and — where the user has supplied a basal metabolic rate — an estimated total daily energy expenditure and the resulting true surplus or deficit relative to that TDEE. The function then performs a battery of checks across body fat percentage, visceral fat, BMI, and body water percentage against established healthy ranges; only the body fat check is differentiated by gender, using separate male/female thresholds, while the visceral fat, BMI, and water-percentage checks apply a single fixed threshold regardless of gender. These checks populate parallel issues and positives lists, and a dedicated contradiction-detection block identifies internally inconsistent patterns — for example, a user averaging well above their calorie target while showing negligible weight change, which most plausibly indicates incomplete meal logging rather than an unusually efficient metabolism — surfacing these as explicit caveats. Notably, the issues, positives, and contradiction lists generated here are returned to the frontend for on-screen display but are not passed into the Groq prompt itself; the narrative-generation step in Section 9.5 works from a separate, more condensed summary of the raw numbers rather than from these structured findings.

Code snippet for build system prompt function

```
def _build_system_prompt(profile: dict, targets: dict, totals: dict,
                         remaining: dict, today_meals: list[dict]) -> str:
    """
    Builds a system prompt that gives the AI full user context so it can answer
    any nutrition/meal question from its own knowledge — no dataset queries.
    """
    goal_label = {"lose": "Lose Weight", "maintain": "Maintain", "build": "Build Musc
        profile.get("goal", "maintain"), "Maintain"
    )

    meals_str = ", ".join(
        f"{m['meal_name']} ({m.get('calories') or 0:.0f} kcal)"
        for m in today_meals
    ) if today_meals else "nothing yet"

    prefs     = profile.get("dietary_prefs") or {}
```

### 9.4 Verdict Logic

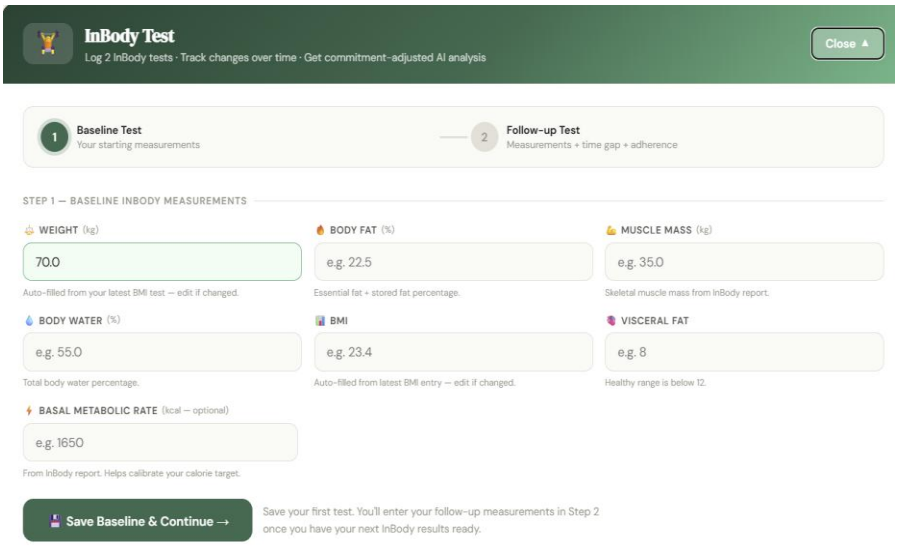
The function computes a goal-aware verdict — on_track, above_target, below_target, or insufficient_data — through a prioritized decision cascade. Body composition outcomes take precedence over diet-derived adherence scores: if the data shows clear muscle loss or stalled progress on a build goal, unwanted weight gain on a lose goal, or unwanted fat gain on a lose or maintain goal, the verdict is set accordingly regardless of how closely the user's logged intake matched their target on paper, because real-world physical outcomes are treated as ground truth. A dedicated body-composition override additionally triggers whenever total weight gain reaches four kilograms or the weekly rate of gain reaches 0.75 kilograms, forcing an above_target verdict even if the logged calorie average appears reasonable, since such a result implies that unlogged intake is the more likely explanation than an unusually fast metabolism. An overall adherence score from zero to one hundred is computed as a weighted blend of calorie adherence (35%), protein adherence (35%), and logging consistency (30%), where each component score is calculated by the _score_adh helper using the goal-specific acceptable bands defined in _ADHERENCE_BANDS and _PROTEIN_BANDS, smoothly interpolating between a perfect score within the acceptable range and a minimum score for values far outside it.

### 9.5 Groq Prompt Construction

Rather than asking the language model to perform the numerical analysis itself, the inbody_submit endpoint performs all quantitative computation in Python first and then supplies a condensed set of computed results — current body composition figures, period-over-period changes, nutrition averages, and an explicit adherence percentage with a labelled interpretation (undereating versus overeating by a stated amount) — to Groq purely for narrative synthesis. The prompt requests a structured JSON response containing a short honest headline, a three-to-four sentence assessment paragraph, a boolean diet_change_needed flag, a one-sentence reasoning string, and, only when a change is warranted, concrete suggested macro values. The prompt encodes two separate thresholds rather than one shared cutoff: calorie adherence below eighty percent instructs the model to conclude that the existing target is very likely already appropriate and that the real issue is consistency rather than the plan itself, so it should decline to suggest a macro adjustment; independently, logging consistency below sixty percent is treated as its own hard override, forcing diet_change_needed to false regardless of any other signal, since the underlying data is considered too incomplete to draw a conclusion from at all. Between those floors, the prompt permits an actual

target adjustment only when adherence is demonstrably high — above ninety-five percent, or in the eighty-to-ninety-five percent range combined with several weeks of stalled or counter-goal body composition change. Taken together, these thresholds are structured to prevent the system from prematurely prescribing calorie or macro changes to a user who simply has not been logging consistently enough yet for any conclusion to be statistically meaningful.

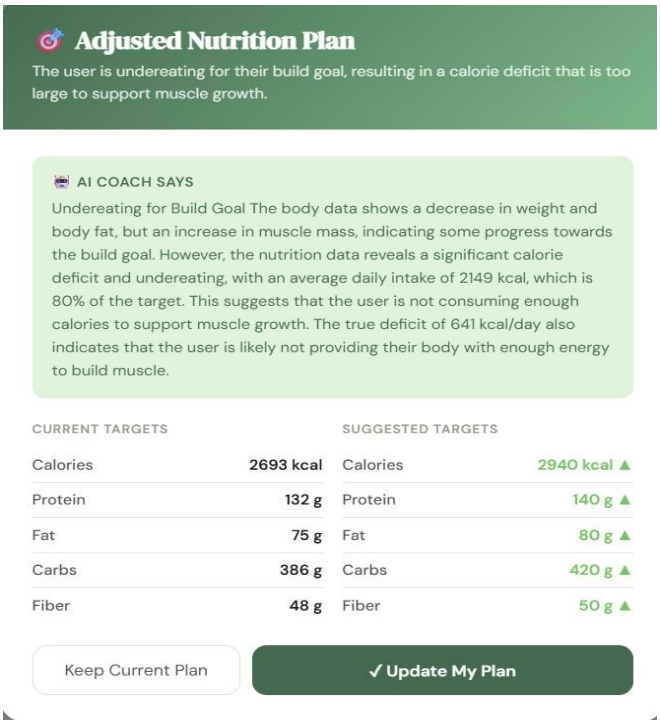
Fig 5 : Inbody Test Baseline




### 9.6 Suggested Target Computation

When the Groq model determines that a target adjustment is warranted, its suggested calorie, protein, fat, carbohydrate, and fibre values are parsed directly from the returned JSON and stored as the suggested_targets payload returned to the frontend, where the user is presented with an explicit accept-or-dismiss choice through the /api/profile/targets endpoints described in Section 7.12. If no adjustment is warranted, suggested_targets is simply set equal to the user's current targets and the response's adjustment_reason field explains, in the model's own words, why the existing plan should be maintained. Every InBody submission — its full set of body composition values, computed verdict, adherence percentage, and generated narrative — is persisted to the inbody_tests table, both to support the historical trend view on the front end and to serve as the baseline reference for the next scan's comparison.

Fig 6 : Adjusted Nutrition Plan Sent by the AI after taking Inbody Test




# Chapter 10 : Frontend Interface

## 10.1 Design System

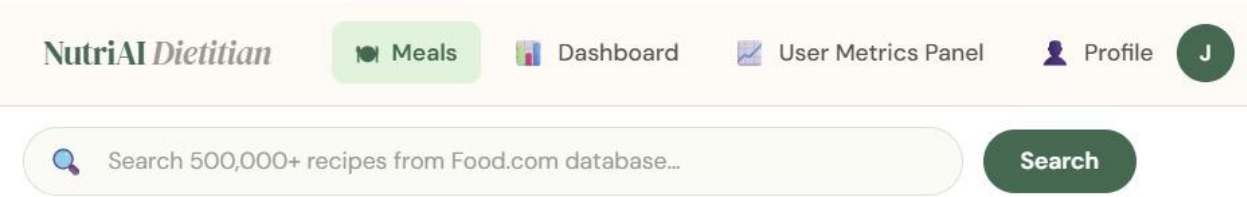
The entire NutriAI frontend shares a single, consistent visual language defined through CSS custom properties declared once at the top of each page's stylesheet. The colour palette centres on a warm off-white background (#F5F2EC), a forest-green primary accent (#2D6A4F) used for buttons and the active navigation state, and a softer secondary green (#52B788) used for success states and progress indicators, alongside a coral warning colour (#E76F51) reserved for destructive actions and over-budget indicators. Typography pairs DM Serif Display, a high-contrast serif used for page titles and section headers, with DM Sans, a clean grotesque sans-serif used for all body text, form labels, and buttons — both loaded from Google Fonts. A consistent 14-pixel border radius, a two-tier shadow system (a subtle resting shadow and a more pronounced elevated shadow used on hover and for modals), and a shared surface-colour hierarchy (a pure white card surface against the warm page background, with a slightly tinted secondary surface for nested elements) are applied uniformly across the Meals, Dashboard, Profile, and User Metrics Panel pages, giving the application a cohesive feel despite being built without a shared component framework.

## 10.2 Navigation (nav.js)

Rather than duplicating the top navigation bar's markup, styling, and behaviour across every HTML page, NutriAI centralises it into a single shared script, nav.js, structured as a self-contained module using the revealing-module pattern (NutriNav). Each page includes the script and calls NutriNav.init(activeId) with its own page identifier, which first calls requireAuth() to redirect unauthenticated visitors to the login page, then injects the complete navigation bar — including the NutriAI brand mark, links to all four main pages with the current page visually highlighted, a

circular avatar showing the user's initial, the username, and a sign-out button — directly into the top of the document body once the DOM has finished loading. The navigation bar is fully responsive: on viewports narrower than 700 pixels, the inline link row and username are hidden via CSS media query and replaced with a hamburger menu button that toggles a slide-down mobile menu panel containing the same set of links, implemented entirely through the shared toggleMobile function so that no page-specific JavaScript is required to support the mobile layout. Centralising navigation in this way also means that any future change to the page list, branding, or authentication behaviour needs to be made in exactly one file rather than five.

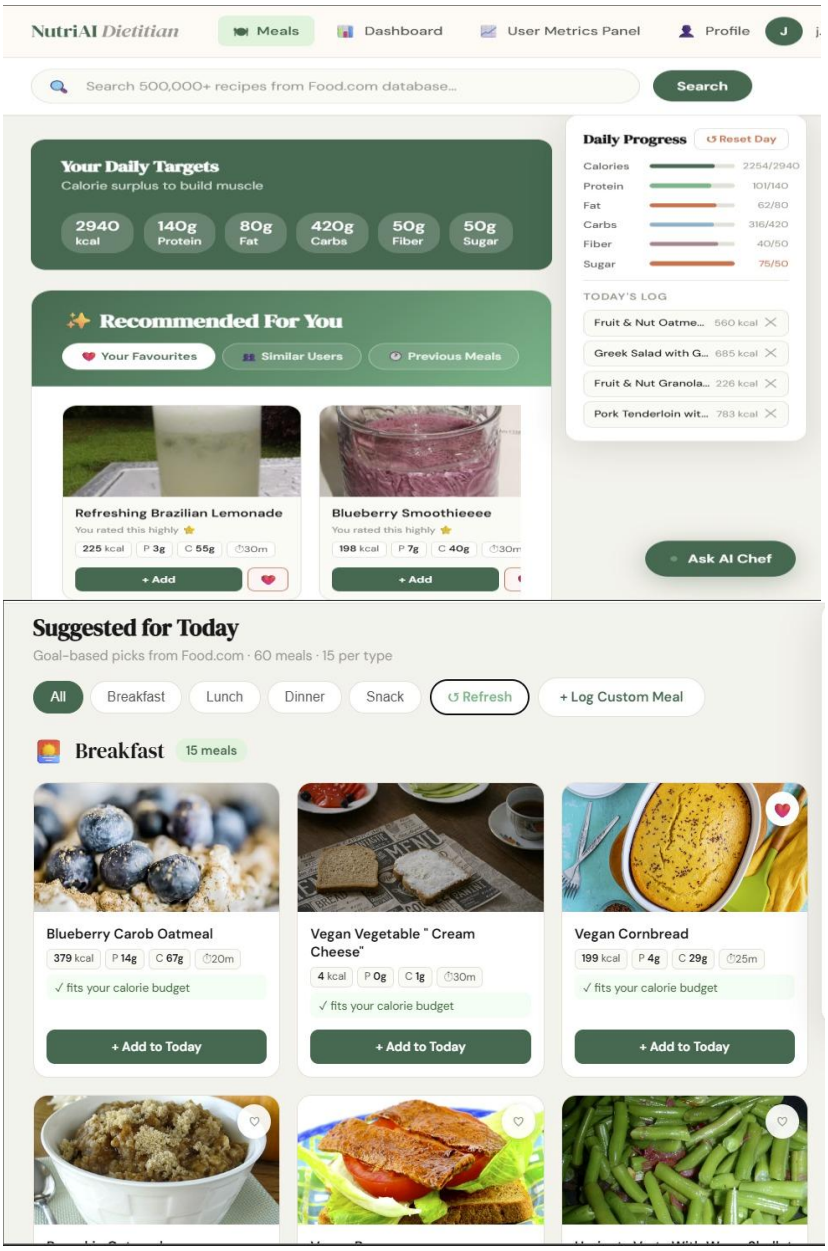
Fig.7 Navigation Bar inside NutriAI website




### 10.3 Meals Page

The Meals page (meals.html) is the most feature-dense screen in the application, combining a sticky search bar, a "Recommended For You" panel with three switchable tabs (Favourites, Similar Users, Previous Meals) backed by the three recommendation endpoints described in Section 7.8, a sixty-meal "Suggested for Today" feed organised into four meal-type sections with filter tabs, a fixed daily-progress sidebar showing live macro bars and the day's logged meals, and the floating AI chef chatbot panel. All of this is driven by a single client-side state object that is kept synchronised with the backend through targeted fetch calls — for example, logging a meal triggers an optimistic local update to the macro tracker immediately, followed by a background API call and a subsequent reconciling reload, so that the interface feels instantaneous even while the network request is still in flight.

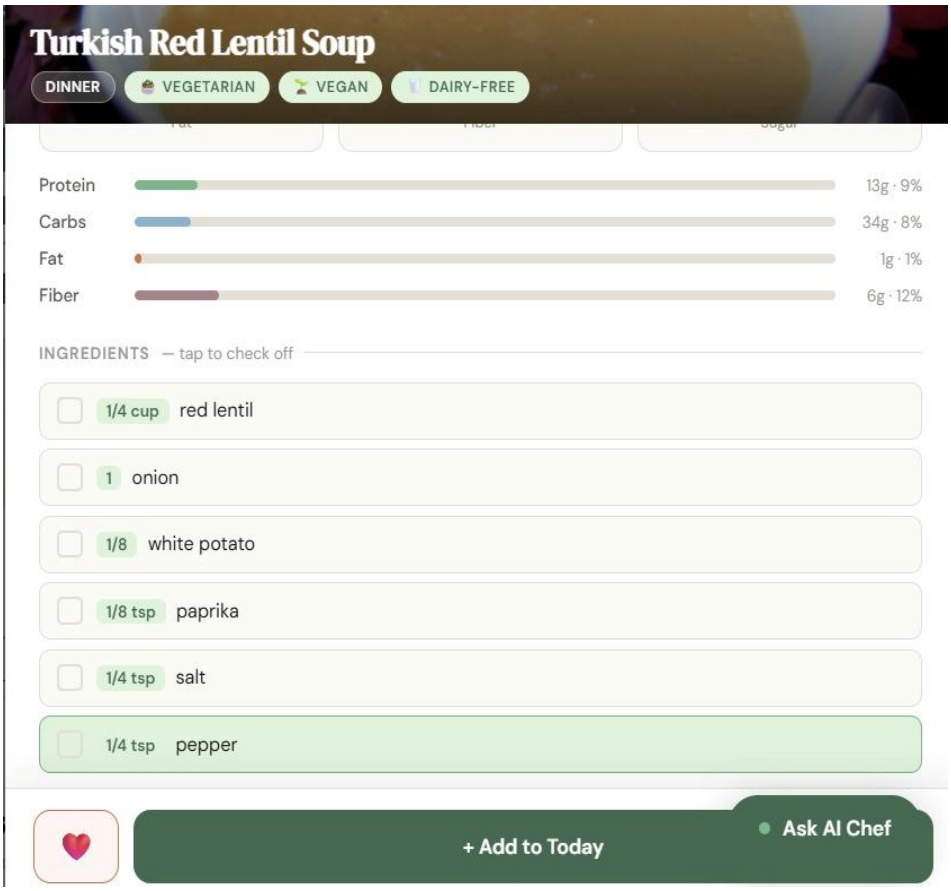
Fig.8 Meals.html page




## 10.4 Detail Panel

Selecting any meal card — whether from the recommendation carousel, the suggested feed, or a search result — opens a slide-in detail panel from the right edge of the screen, displaying a hero banner image, quick-stat tiles for cook time, servings, calories, and percentage of daily calorie budget, a community or personal star rating section, dietary tag badges, a six-tile nutrition grid accompanied by macro progress bars relative to the user's daily targets, a tappable, checkable ingredient list with quantities pre-scaled to a single serving, and the full numbered cooking instructions. The panel includes its own favourite-toggle and "Add to Today" controls in a sticky footer, allowing a user to act on a meal without needing to scroll back to the underlying card.

Fig.9 Meal Ingredients & Macros after pressing on any meal



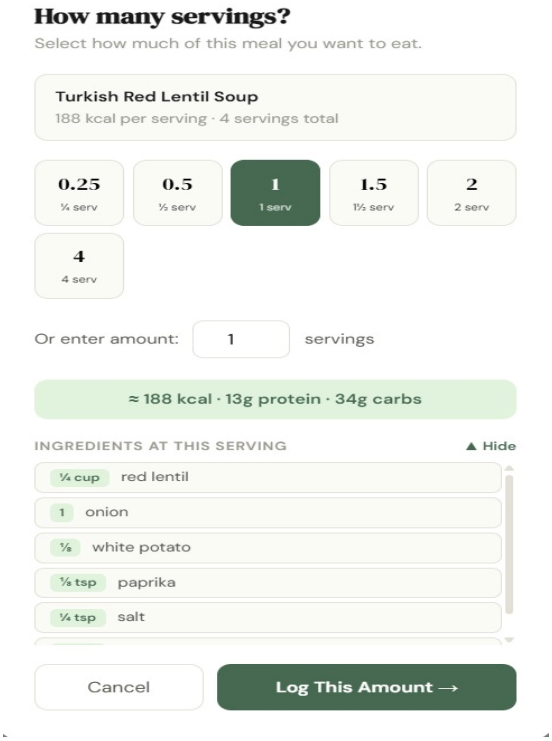

### 10.5 Chatbot UI

The chatbot is implemented as a floating action button in the bottom-right corner of every authenticated page that, when clicked, expands into a fixed-position chat panel with a header showing the assistant's name and an online status indicator, a context bar summarising the user's remaining calories, protein, and carbohydrates for the day, a scrollable message history, and a text input row. Assistant messages containing a parsed [MEAL_DATA] block are rendered with an accompanying meal card showing the suggested nutrition and a dedicated "Add to Log" button, reusing the same portion-modal logging flow described in Section 10.6 so that a chatbot-suggested meal and a dataset-recommended meal are logged through an identical, consistent code path.

### 10.6 Portion Modal

Recognising that a recipe's default serving size rarely matches exactly what a user intends to eat, every "Add" action across the application routes through a shared portion-selection modal rather than logging the default serving immediately. The modal provides common portion presets, a custom numeric input, a live nutritional preview that updates with the selected portion, and — where ingredient data is available — a rescaled ingredient list. Confirming the modal scales all nutritional values by the chosen multiplier before the meal is saved, ensuring accurate portion-adjusted logging rather than approximation.

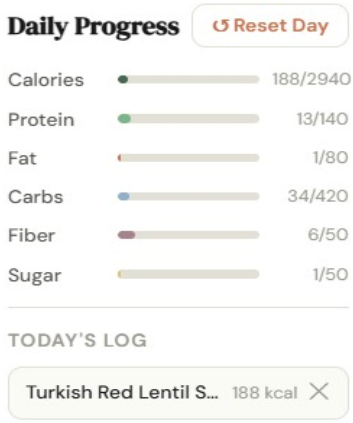
Fig.10 : Serving Size chosen by the user to be added into his daily log




## 10.7 Daily Tracker Sidebar

A fixed sidebar on the Meals page provides constant visibility into the user's progress against their daily targets without requiring navigation away from the meal browsing experience. Six animated progress bars — one per tracked macro — fill proportionally to the percentage of the day's target consumed, switching to a warning colour when a macro is exceeded, beneath which a scrollable list of every meal logged that day is displayed with its calorie contribution and a one-tap remove button. A "Reset Day" control allows the user to clear an entire day's log in a single confirmed action, deleting every entry through the standard delete endpoint rather than a separate bulk-delete route, keeping the set of distinct API operations small and auditable.

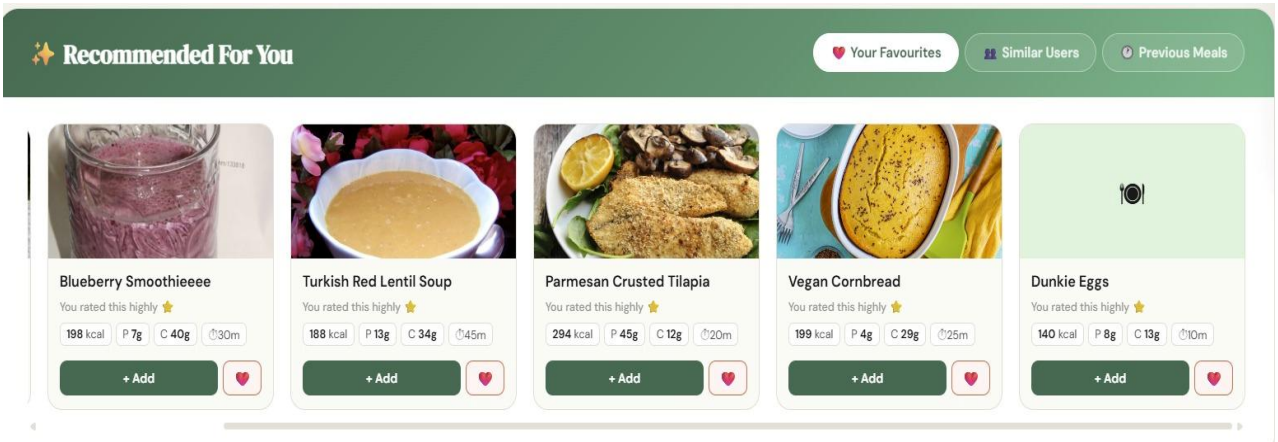
Fig.11 : Daily progress for each user to trach his daily macros




### 10.8 Recommendation Tabs

The "Recommended For You" panel at the top of the Meals page surfaces three parallel recommendation strategies behind a single tabbed interface, allowing the user to compare different bases for suggestion without leaving the page. The Favourites tab displays the user's own five-star rated meals; the Similar Users tab displays the output of the two-tier peer recommendation engine described in Section 6.8, labelled with the basis of similarity (matching BMI and goal); and the Previous Meals tab surfaces the user's most frequently logged meals from their personal history. Each tab independently handles its own empty state with an explanatory message and a hint toward the action that would populate it — for example, prompting the user to rate a meal five stars before the Favourites tab will show any content — rather than presenting a generic "no results" message that gives the user no path forward.

Figure 12 — Recommended For You panel with Favourites / Similar Users / Previous Meals tabs




## Chapter 11 : User Metrics Panel & Body Tracking UI

The User Metrics Panel (UserMetricsPanel.html) is the body-tracking hub of NutriAI, providing users with a dedicated page for submitting and reviewing InBody body composition tests, logging quick BMI check-ins, visualising their historical nutrition and weight trends, and acting on AI-generated target adjustment suggestions. The page is deliberately separated from the main Meals page so that body tracking does not compete visually with the meal discovery experience, while still sharing the same navigation bar and authentication layer through NutriNav.

### 11.1 InBody Test Submission Form

The InBody submission interface is structured as a collapsible, accordion-style section that the user expands on demand, keeping the page uncluttered for users who are not currently at a scan submission step. Internally, the form is implemented as a two-step wizard separated into two sequential panels — ibPanel1 and ibPanel2 — where completion of the first step is a prerequisite for reaching the second, enforcing a logical data entry order while visually communicating progress through a step indicator row that displays the status of each step as either active, pending, or done.

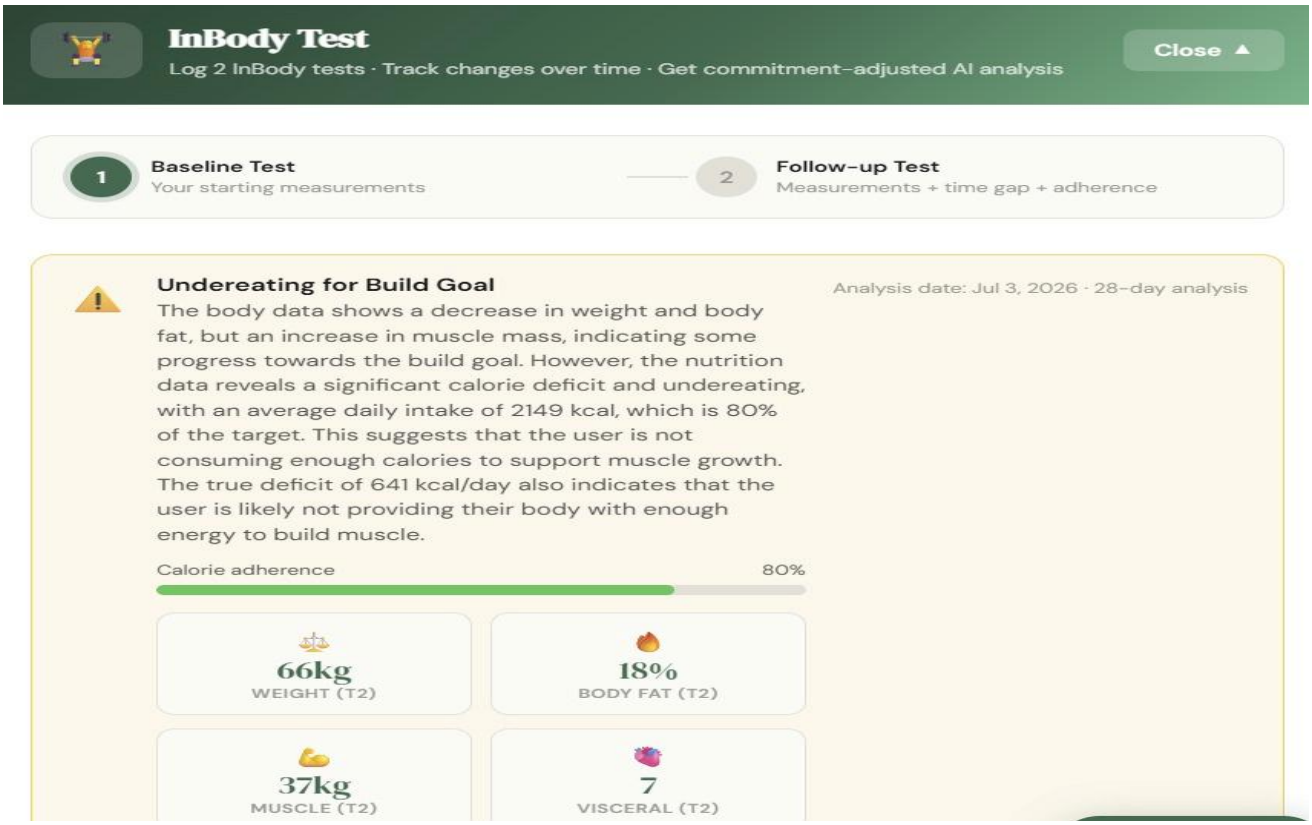
The first panel collects the baseline measurements from the user's first InBody scan: body weight (kilograms), body fat percentage, muscle mass (kilograms), body water percentage, BMI, visceral fat level on the InBody 1–20 scale, and basal metabolic rate in kilocalories. Of these fields, only body weight is mandatory — every other measurement is optional, so that users with incomplete scan printouts can still benefit from at least a partial analysis. When the page loads, the prefillInbodyForm function automatically populates the weight field from the user's stored profile weight and the BMI field from their most recent bmi_entries record, adding a visual "prefilled" CSS class to signal to the user which values were sourced from their profile rather than typed manually.

Clicking the "Save Baseline" button invokes saveStep1(), which reads and validates all field values using a consistent parseNum helper that handles both period and comma decimal separators for international users, confirms that weight is present, stores the collected values in the module-level _ibBaseline object, updates the step indicator to mark Step 1 as completed, renders a compact saved-summary card displaying the three most important baseline values (weight, body fat, and muscle mass), and reveals ibPanel2 while hiding ibPanel1. An "Edit" link on the saved summary card calls editStep1() to reverse this transition, allowing the user to correct a baseline entry before proceeding.

The second panel collects the follow-up measurements from a subsequent InBody scan, using an identical set of fields. When the user clicks "Analyse My Progress," the frontend assembles a JSON payload containing both the baseline object stored in _ibBaseline and the newly entered follow-up values and sends it to the /api/inbody/submit endpoint described in Chapter 9. The resulting analysis — verdict, adherence percentage, AI narrative, and any suggested target changes — is rendered in the inbodyLastResult card using the renderLastInbodyResult function, which dynamically applies a verdict-specific CSS class (on_track, below_target, above_target, or

insufficient_data) and splits the Groq-generated ai_message on its first newline so that the headline and the body paragraph are displayed in typographically distinct styles.

Figure 13: InBody two-step wizard showing step indicator, baseline entry, and analysis result card.



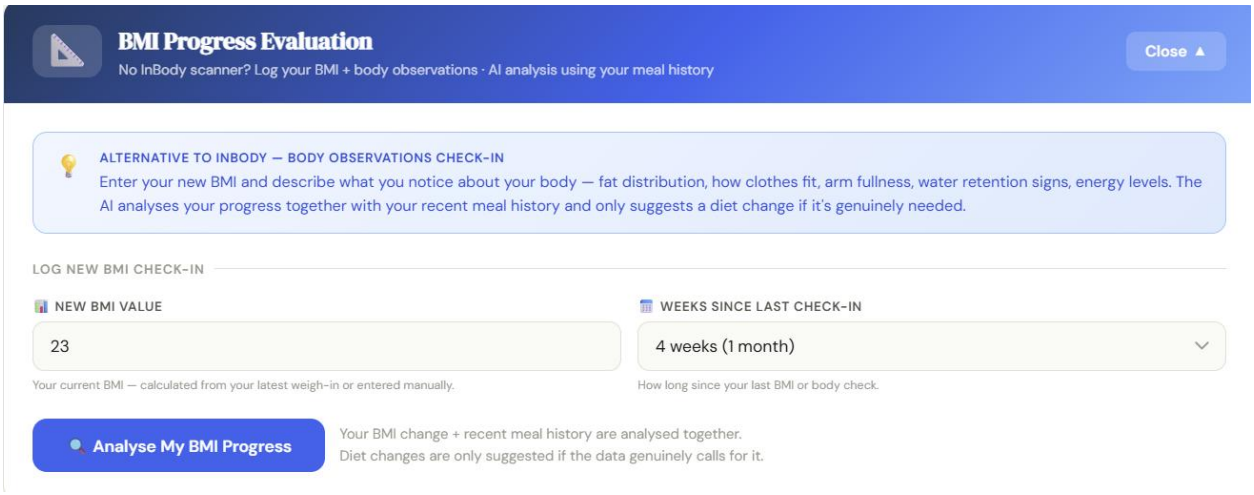

## 11.2 BMI Progress Evaluation

Recognising that access to an InBody scanner is not universal, the User Metrics Panel provides a parallel, scanner-free body composition check-in pathway in the form of the BMI Progress Evaluation section. This section is similarly accordion-collapsed and presents a shorter form that accepts three inputs: the user's newly measured BMI value, a dropdown for the number of weeks elapsed since their last check-in (ranging from one week to twelve weeks), and a free-text body observations textarea where the user describes physical cues they have noticed — changes in how clothes fit, arm or leg fullness, perceived fat redistribution, water retention signs, and energy level changes.

The accompanying notice explicitly frames this section as an "Alternative to InBody — Body Observations Check-in," setting the expectation that this path relies more heavily on qualitative self-reporting than on objective measurements, and reassures the user that a dietary adjustment will only be suggested if the combined data genuinely warrants one. Submitting the form calls submitBmiEval(), which sends the three fields to the /api/bmi/evaluate endpoint. The backend

constructs a Groq prompt that embeds the BMI change, the observation period, the free-text notes, and the user's thirty-day average nutrition history, producing a structured JSON response that is rendered into the bmiEvalResult card — comprising an icon, headline, body paragraph, and an optional dietary adjustment suggestion box — identical in visual structure to the full InBody result card, providing a consistent user experience across both body tracking pathways.

Figure 14: BMI evaluation section



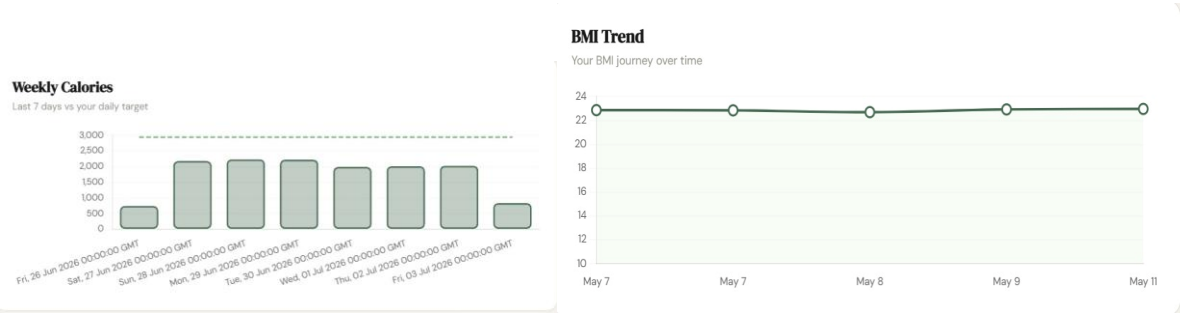

## 11.3 Progress History and Trend Charts

The lower half of the User Metrics Panel is devoted to visualising the user's logged history. On page load, loadPage() issues four parallel fetch calls — to /api/history/summary, /api/history/daily, /api/profile, and /api/inbody/history — using Promise.all(), so that all four data sources arrive concurrently rather than sequentially, keeping the page's time-to-interactive as low as possible on a single-page fetch cycle.

The weekly nutrition trend chart (calChartInst) is built with Chart.js and renders up to seven days of daily calorie totals as a bar chart, with a horizontal dashed reference line drawn at the user's calorie target using Chart.js's annotation-style dataset trick — a single-point line dataset with a stepped fill repeated across all seven labels. Bar colours shift from the primary green to the warning coral when the day's intake exceeds the target, giving the user an immediate visual cue of which days pushed over budget without needing to read exact numbers. The BMI history chart (bmiChartInst) displays a line chart of all stored bmi_entries values plotted against their recorded_at timestamps, allowing the user to observe their weight trajectory over weeks or months at a glance.

A compact statistics row above the charts surfaces four headline numbers derived from the summaryData object: total meals logged all-time, the current consecutive logging streak, the thirty-day average daily calorie intake, and the user's most recent logged weight. These stats are rendered as styled tiles with large value typography and small label text, providing a quick summary of the user's overall engagement and progress without requiring any interaction.

Figure 15: Weekly nutrition chart (with target line) and BMI history trend chart.




## 11.4 Daily Log and Top Meals

Below the charts, the User Metrics Panel renders a day-by-day expandable log of the user's meal history using the renderDailyLog function, which groups meal_log rows by date and presents each day as a collapsible card showing the date, the total calories consumed that day, and — when expanded — a list of individual meal entries with their name, meal type emoji, and macro breakdown. This view complements the history endpoints' paginated output by providing a visually browsable, accordion-style log that is more scannable than a flat list on a mobile screen.

A separate "Top Meals" section rendered by renderTopMeals displays the user's five most-frequently logged meals as compact cards with a logging-count badge, giving the user at a glance a picture of which meals have become regular fixtures in their diet — information that is also used by the history-based recommendation engine described in Section 6.9 to surface familiar favourites in the "Previous Meals" tab on the Meals page.

## 11.5 Accepting and Resetting Suggested Targets

When the InBody or BMI evaluation returns a suggested macro adjustment, the result card includes two action buttons: "Accept Suggested Targets" and "Keep Current Plan." Clicking the accept button calls the PUT /api/profile/targets endpoint, which persists the suggested calorie, protein, fat, carbohydrate, and fibre values into the user_profiles.custom_targets JSONB field. From that point forward, every call to _get_targets() in the backend will return these custom values instead of the Mifflin–St Jeor formula output, so that the daily tracker sidebar, the macro progress bars, and the recommendation engine's calorie cap logic all automatically reflect the adjusted targets without requiring any further configuration.

The "Keep Current Plan" button dismisses the suggestion without making any change to the stored targets, and both buttons are removed from the UI after either action to prevent duplicate submissions. The "Reset to Formula Targets" control, always visible in the User Metrics Panel header regardless of whether custom targets are currently active, calls the DELETE /api/profile/targets endpoint to clear any stored custom_targets value, reverting the application to the standard formula-based calculation immediately. This explicit reset mechanism ensures that a

previously accepted InBody suggestion does not silently persist after the user has meaningfully changed their lifestyle, goal, or body weight.

## Chapter 12 : Dashboard Page

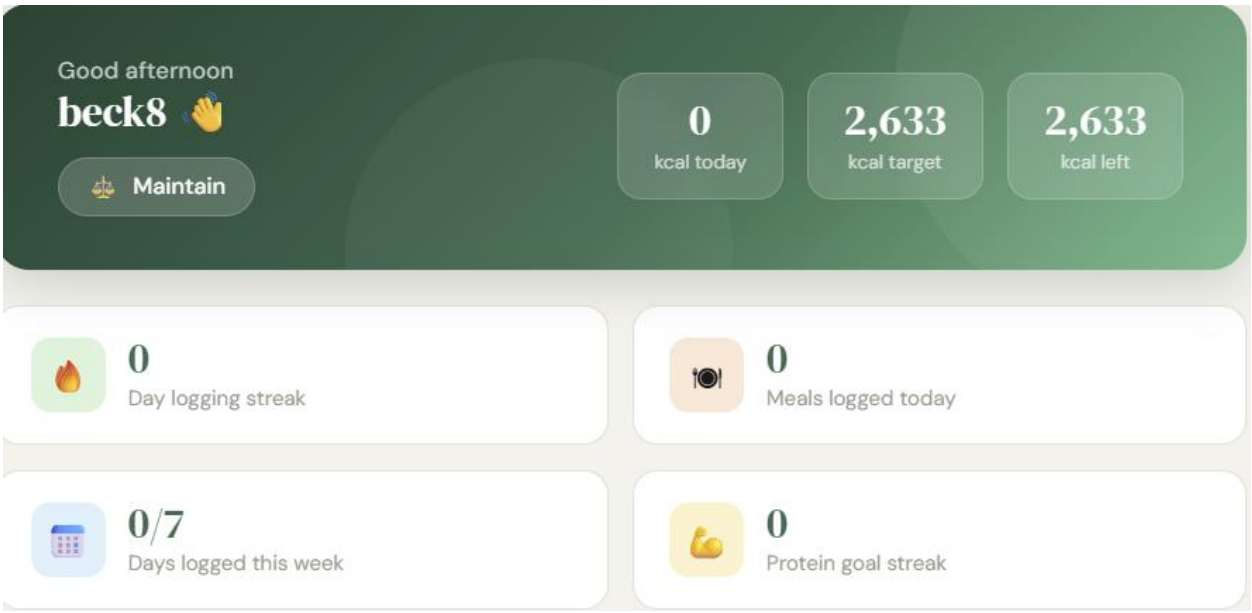
The Dashboard page (dashboard.html) provides a bird's-eye view of the user's nutrition performance and engagement metrics, synthesising data from five separate API endpoints into a single, visually rich screen. Unlike the Meals page — which is task-oriented and centred on discovering and logging food — the Dashboard is retrospective and analytical, designed to answer the question "how am I doing?" across multiple time horizons simultaneously. All data loading is handled by initDashboard(), which fires five parallel Promise.allSettled() calls on page load and renders each section of the page in a defined order as results arrive, so that a slow external API call (such as the AI insights endpoint) does not block the rendering of faster, locally computed sections.

### 12.1 Welcome Banner and Quick Stats

The page opens with a full-width welcome banner that greets the user by name with a time-of-day-appropriate greeting — "Good morning," "Good afternoon," or "Good evening" — computed from the current hour at render time, personalising the experience on each visit without any server-side logic being required. Alongside the greeting, the banner displays the user's active goal with its associated icon ( 🔥 for Weight Loss, ⚖️ for Maintain, 💪 for Build Muscle) and the current date.

Beneath the greeting, a row of three stat cards — rendered by renderWelcome() after S.eaten has been populated from /api/meals/today — displays the user's daily calorie target, the calories consumed so far today, and the remaining (or over-budget) balance. When the day's consumption exceeds the target, the "remaining" card and the consumed card both switch to a warm coral background and border colour, providing an unmistakable visual warning without a text alert that the daily budget has been exceeded. A fourth tile displays the count of meals logged today, drawn directly from the length of the todayLog array.

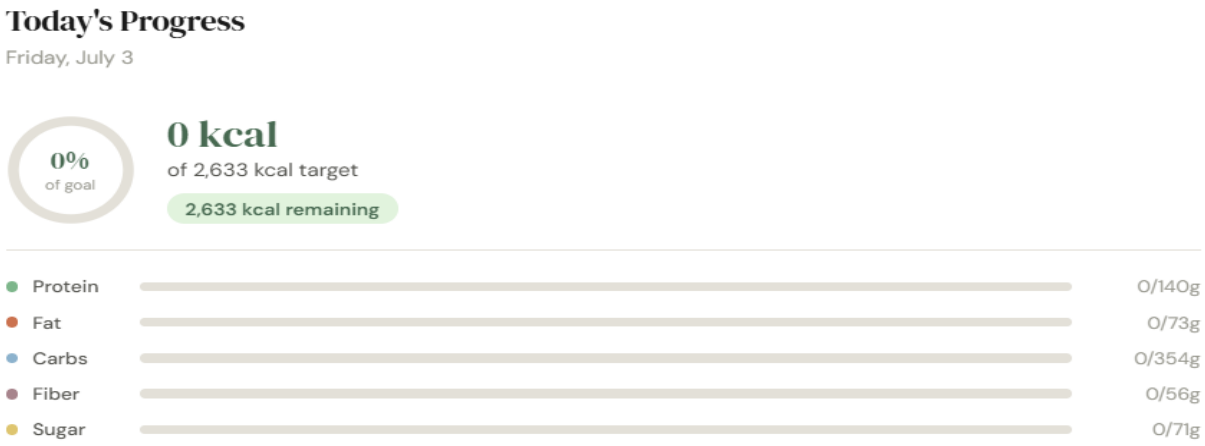
Figure 16 : Dashboard welcome banner with personalised greeting, goal, and daily calorie summary tiles.




## 12.2 Daily Macros Chart

The daily Macros section displays a Chart.js bar chart showing the user's calorie and other macros intake for the current day in comparison to their daily calorie target. The chart provides a simple visual overview of daily calorie progress, allowing users to quickly monitor whether they are within their target range.

Figure 17: Daily macros chart inside dashboard page

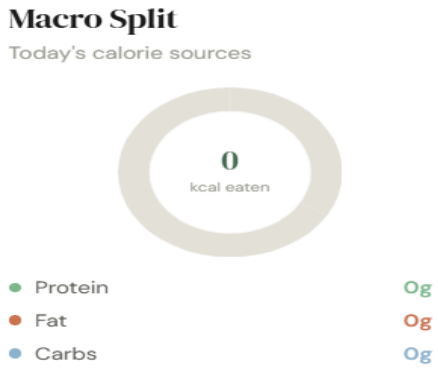



## 12.3 Macro Donut Chart

Alongside the weekly bar chart, a donut chart rendered by renderMacroDonut() shows the proportional contribution of protein, fat, and carbohydrates to today's total calorie intake. The chart uses Chart.js's doughnut type with a custom centre-label plugin that renders the total calories

consumed today in large text within the donut hole, providing a concise calorie summary at a glance. The three macro segments are coloured using semantically meaningful colours consistent across the application: blue for protein, yellow-orange for carbohydrates, and warm red for fat, matching the colour scheme used in the macro progress bars on the Meals page tracker sidebar. If no meals have been logged yet today, the chart renders a single grey placeholder segment with the centre label showing zero, rather than an empty or broken chart state.

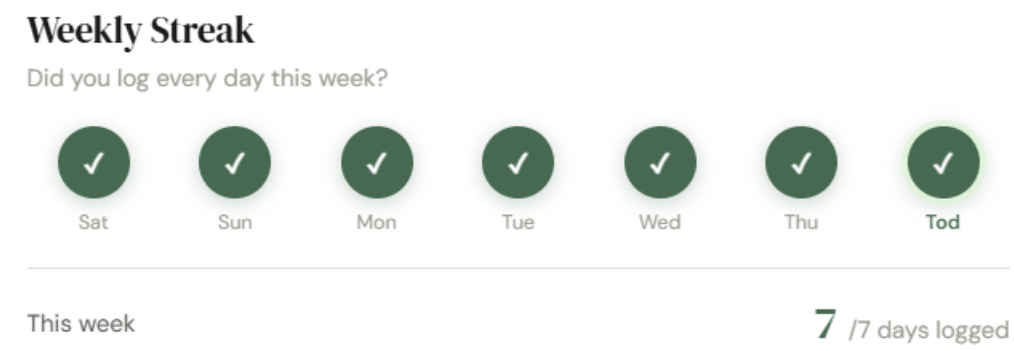
Figure 18: Macro proportion donut chart (protein / carbs / fat)




### 12.4 Seven-Day Logging Chart

The seven-day chart, rendered , displays one circle per day for the past seven days, with each circle filled and marked with a checkmark when at least one meal was logged that day, or left hollow when no meals were recorded. Today's circle is visually distinguished from past days by a different fill style — a pulsing ring rather than a solid fill — regardless of whether a meal has been logged yet, helping the user orient themselves in time within the row. The count of logged days in the seven-day window is displayed as a score beneath the row ("X / 7 days logged this week"), providing a simple weekly consistency metric that complements the longer-horizon streak displayed in the achievements section.

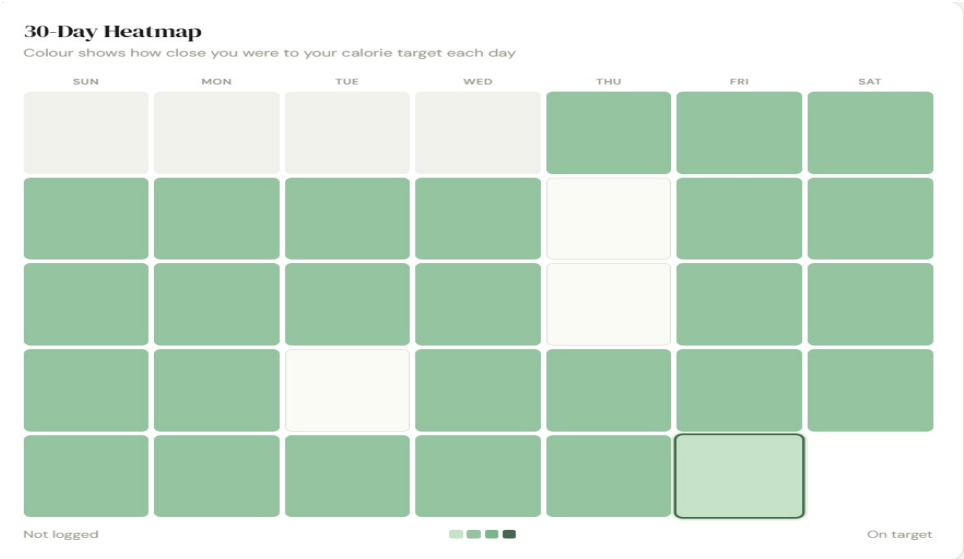
Figure 19 : Weekly Login Streak chart for the user




### 12.5 Thirty-Day Calendar Heatmap

A thirty-day calendar grid rendered by renderCalendar30d() extends the heatmap concept to a full month, laying out day cells in a conventional week-column grid aligned to the correct day of the week by prepending blank offset cells for the first row. Each cell is coloured according to the ratio of that day's logged calories to the user's calorie target: over 115% of target renders as a deep coral (cal-over), 95–115% as a full green (cal-high), 80–95% as a lighter green (cal-good), 50–80% as a muted sage (cal-mid), below 50% as a near-empty pale colour (cal-low), and days with no logged meals as empty grey cells. Hovering any cell reveals a tooltip showing the exact date and the logged calorie total (or "Not logged"), giving the user precise information without cluttering the grid with text labels. This visualisation allows the user to identify patterns at a glance — for example, consistently under-logging on weekends, or a streak of on-target days that has recently broken.

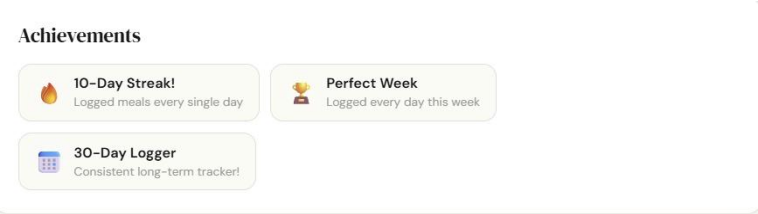
Figure 20: Thirty-day nutrition calendar heatmap. Cell colour indicates percentage of daily calorie target achieved.



### 12.6 Streaks & Achievements Badges

The streaks section, rendered by `renderStreaks()` from the `/api/dashboard/streaks` response, displays engagement metrics including the current consecutive daily logging streak, the number of days logged during the current week, the total all-time logged days, and the consecutive days on which the protein target was achieved. These values are mapped to achievement badges in `renderAchievements()`, where milestones such as a seven-day logging streak or a completed full week unlock badges displayed as styled cards with icons and descriptive labels.

Figure 21: Streak & achievement badges,



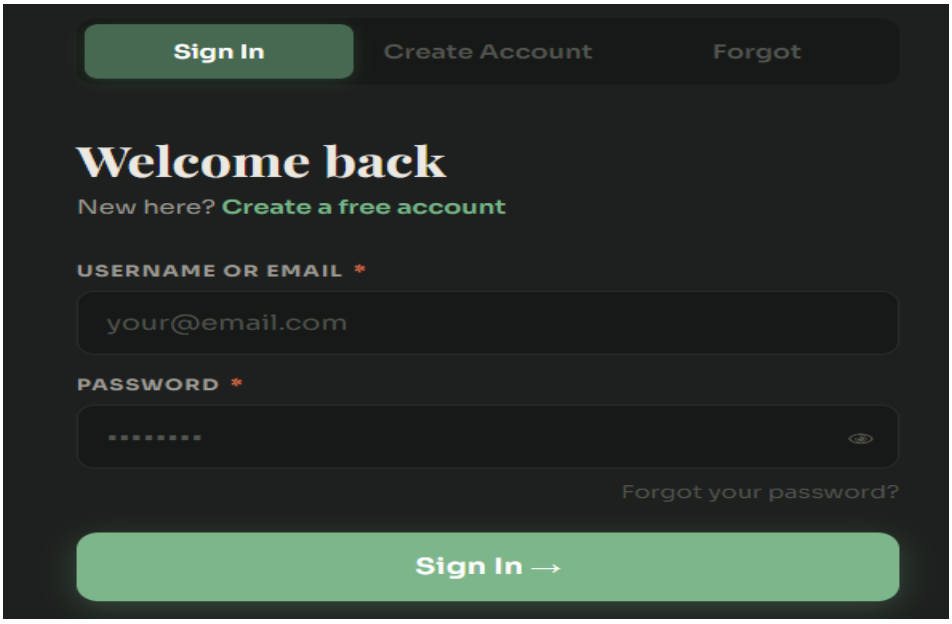

## Chapter 13 : Profile and Authentication Pages

### 13.1 Login Page Design

The authentication page (login.html) uses a dark-themed, full-viewport split-panel layout that departs deliberately from the warm off-white palette used on the application's internal pages, signalling to the user that they are on a gateway screen rather than inside the main application. The left panel occupies approximately forty-five percent of the viewport width and serves a purely decorative role: it renders an animated background built from three absolutely-positioned gradient orbs that drift slowly across the panel using CSS keyframe animations, overlaid with the NutriAI brand name in the DM Serif Display typeface, a short tagline, and a vertical stack of three feature highlight cards (AI-Powered Recommendations, InBody Analysis, and Smart Nutrition Tracking) rendered with semi-transparent glassmorphism backgrounds. This decorative panel is hidden entirely on mobile viewports (below 768 pixels) to give the form maximum space on small screens.

The right panel contains the actual authentication interface: a tab bar with two buttons — Sign In and Create Account — that toggle between the login and registration panels using showTab(), which swaps the active CSS class between the two panel divs and clears any existing alert messages to prevent stale error text persisting across tab switches. The panel uses the Playfair Display typeface for its heading elements, creating a subtle typographic distinction from the DM Serif Display used in the nav and on internal pages that reinforces the "entrance" character of the login screen.

Figure 22: Sign in page showing the decorative design

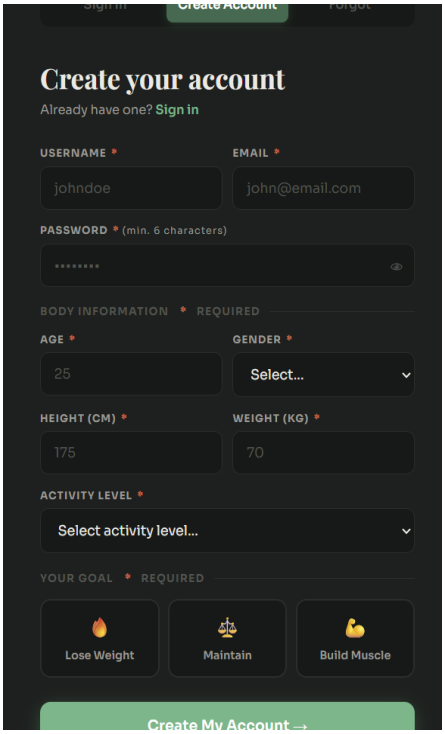



### 13.2 Registration Flow

The registration form is substantially more detailed than the login form, collecting all the information required to both create the user account and immediately populate a complete user profile, eliminating the need for a separate post-registration onboarding flow. The form collects: username, email, and password (with a password visibility toggle implemented by togglePw(), which switches the input type between "password" and "text" and updates the button emoji accordingly); age, gender, height, and weight for BMI calculation; an activity level selector presenting five options (Sedentary, Light, Moderately Active, Very Active) each displayed as a labelled card that highlights on selection; and a goal selection between Lose Weight, Maintain, and Build Muscle, rendered as goal cards with icons and short descriptions that become visually active when selected by selectGoal().

Client-side validation in doRegister() checks all fields before the API call is made, accumulating errors into an array and displaying them as a single combined message through showAlert(), so that the user sees all validation issues at once rather than sequentially. On successful registration, the backend returns a JWT and user object in the same response format as the login endpoint, and the shared afterAuth() function stores both values in localStorage and redirects the browser to meals.html after a short delay, so that the user lands directly on the main application page without any additional manual login step.

Fig.23 - Create an Account Page inside (login.html) page



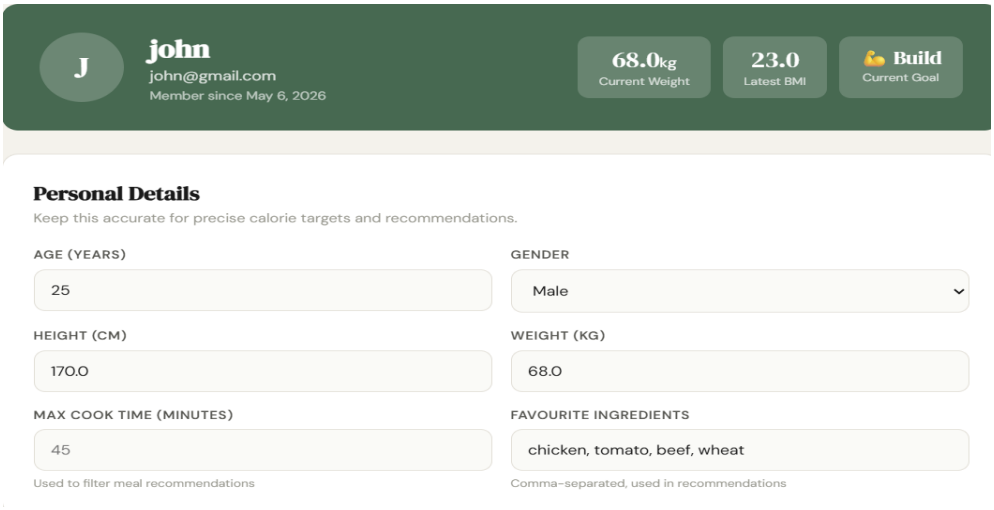

### 13.3 Profile Page — Personal Details& Prefered Cook

The Profile page (profile.html) follows a card-based layout structured around a green-gradient profile header card at the top and a series of form-group cards below. The header card renders the user's initial in a large circular avatar, their username in large type, their email address, and member since (date ). A row of three header stat tiles displays the user's current weight, latest BMI, and active goal with an emoji icon, giving at-a-glance body metrics without requiring navigation to the User Metrics Panel.

The personal details form is laid out in a two-column responsive grid on wider screens and a single column on mobile. The form fields — age, gender, height, weight, and maximum acceptable cooking time in minutes — use standard HTML form controls styled with the application's shared input theming. The activity level selector reuses the same card-radio design pattern as the registration form, and the goal selector likewise reuses the goal-card pattern, both populated from the stored profile data by loadPage() after it receives the /api/profile response. The profile also exposes the user's username and email in read-only display fields, since these identity fields cannot be changed after account creation.

The favourite ingredients field is implemented as a freeform comma-separated text input pre-populated from the fav_ingredients JSONB array. The instructions beneath the field guide the user to enter individual ingredients separated by commas — for example, "chicken, garlic, broccoli, olive oil" — and explain that these will be used to boost the relevance scores of recipes containing those ingredients in the recommendation engine's soft-preference scoring step described in Section 6.3.

Figure 24: Profile page header card showing avatar, member details, and current stats; personal details form below.

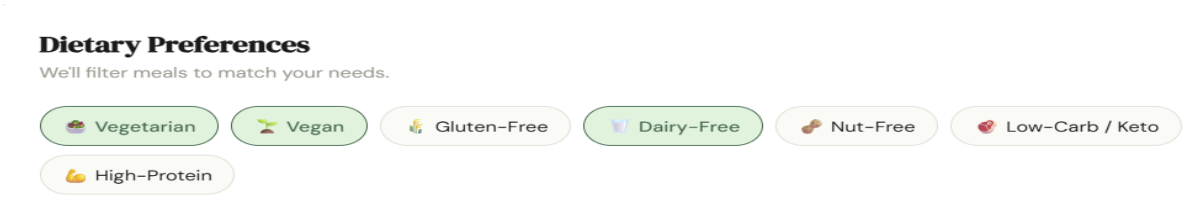



## 13.4 Dietary Preferences

Below the personal details form, a dietary preferences section presents the seven supported restriction and preference flags as interactive chip toggles: Vegetarian, Vegan, Dairy-Free, Nut-Free, High-Protein, and Low-Carb. Each chip contains a hidden checkbox input and a visible label; clicking the chip toggles the checkbox state and applies or removes the "selected" CSS class, turning the chip from an outlined style to a filled green style to provide clear visual feedback. These chips are pre-populated by reading the dietary_prefs JSONB array from the user's stored profile and programmatically clicking any chip whose preference key appears in that array during page load.

Submitting the combined personal details, dietary preferences, and favourite ingredients form invokes saveProfile(), which constructs a single PUT request to /api/profile with the full updated profile payload. A success toast confirms the save, the reminder banner (if visible) is hidden, and the profile header stats are refreshed to reflect any changes to the displayed weight or goal.

Fig.25 : Dietary Preferences chosen by each user to match his own preference




## 13.5 Profile Update Reminder Logic

Because the accuracy of the application's calorie and macro target calculations is directly dependent on the currency of the user's stored weight and activity level, the backend tracks a profile_updated_at timestamp and compares it against the configured threshold (seven days by default) on every /api/profile GET request. When the profile has not been updated in longer than the threshold, the response includes a show_update_reminder flag set to true and a days_since_update integer count.

The frontend renders a dismissible orange banner at the top of the profile form when this flag is present, displaying a message such as "Your profile was last updated 14 days ago — keep your data current for accurate calorie targets and recommendations." The banner links directly to the weight field in the form by smooth-scrolling, guiding the user immediately to the most important field to refresh. This reminder mechanism is intentionally non-blocking — the user is never forced to update their profile — but provides a consistent, visible nudge that encourages the regular weight check-ins that keep the system's dietary targets accurate over time.

# Chapter 14: Coach Monitoring

One of the key additions introduced in NutriAI is the integrated coach monitoring system, designed to bridge the gap between automated nutritional recommendations and professional human guidance. Rather than requiring coaches to manually request progress reports, analyse scattered measurements, or repeatedly communicate with clients for updates, the platform centralises all relevant nutritional information into a single monitoring interface. Historical meal logs, calorie adherence, weight progression, and BMI evaluations , allowing coaches to review a client's overall progress quickly and consistently. By combining automated data collection with professional oversight, this feature reduces the administrative workload placed on nutrition coaches while enabling more informed and timely coaching decisions. The result is a more efficient coaching workflow in which professionals can focus on interpreting progress and providing personalised guidance instead of spending significant time gathering and organising client information.

## 14.1 Coach Authentication Page

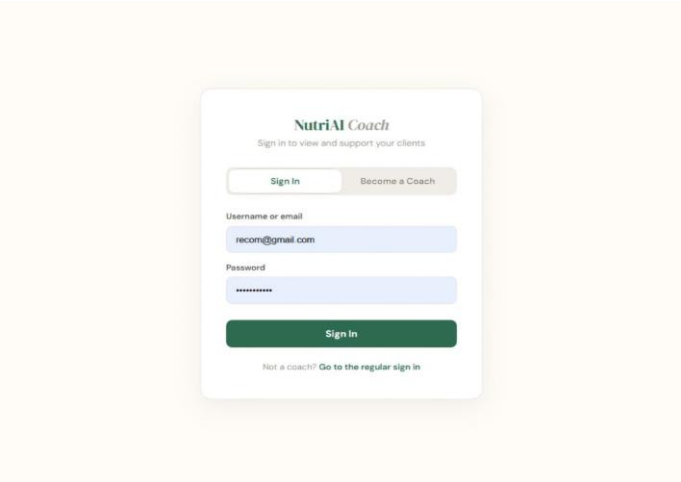
The Coach Authentication page (coach_login.html) provides a dedicated login and registration interface for nutrition coaches. The page follows the application's visual theme using a centred authentication card with the NutriAI Coach branding displayed at the top. Two tabs allow coaches to switch between the Sign In and Become a Coach forms.

The Sign In form requires the coach's username (or email) and password before authenticating through the backend. Upon successful login, the authentication token and coach information are stored locally, and the user is redirected directly to the Coach Dashboard.

The registration form allows new coaches to create an account by providing a display name, username, email address, password, and an optional professional biography. Once registration is

completed successfully, the coach is automatically logged in and redirected to the dashboard without requiring a second authentication step.

Figure 26: Coach authentication page showing the Sign In and Become a Coach forms.



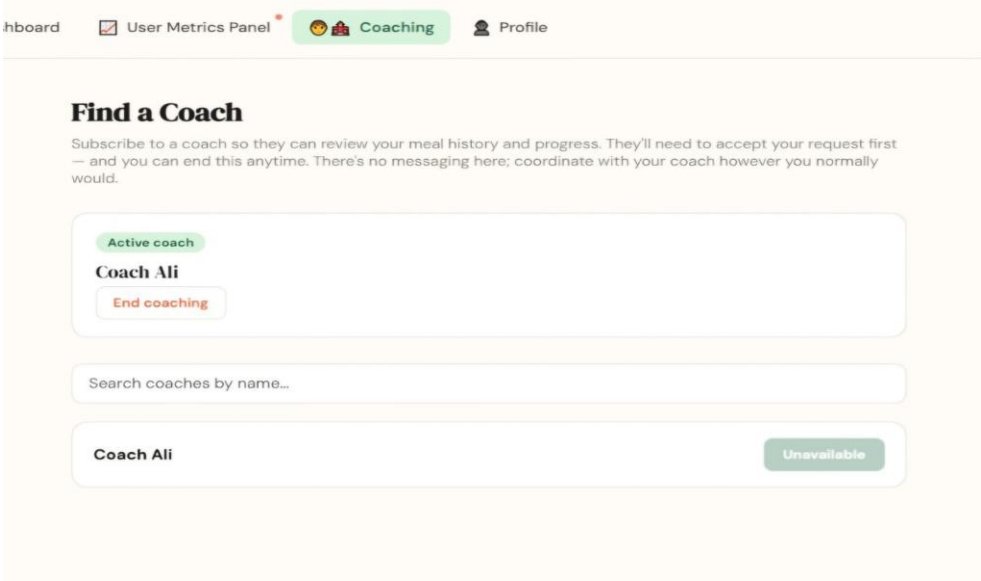

### 14.2 Find a Coach Page

The Find a Coach page (find_coach.html) enables users to browse available nutrition coaches and submit coaching requests. The page displays a search bar that filters coaches by name and presents each coach inside an individual card containing their display name, professional biography, and a Request button.

When a user has already submitted a request or is currently assigned to a coach, a status card appears at the top of the page showing the coach's details together with the current request status (Pending or Active). Users may cancel a pending request or end an existing coaching relationship directly from this section.

Once a coaching request has been submitted, additional request buttons are disabled until the current coaching relationship has been cancelled, ensuring that each user can only be assigned to one coach at a time.

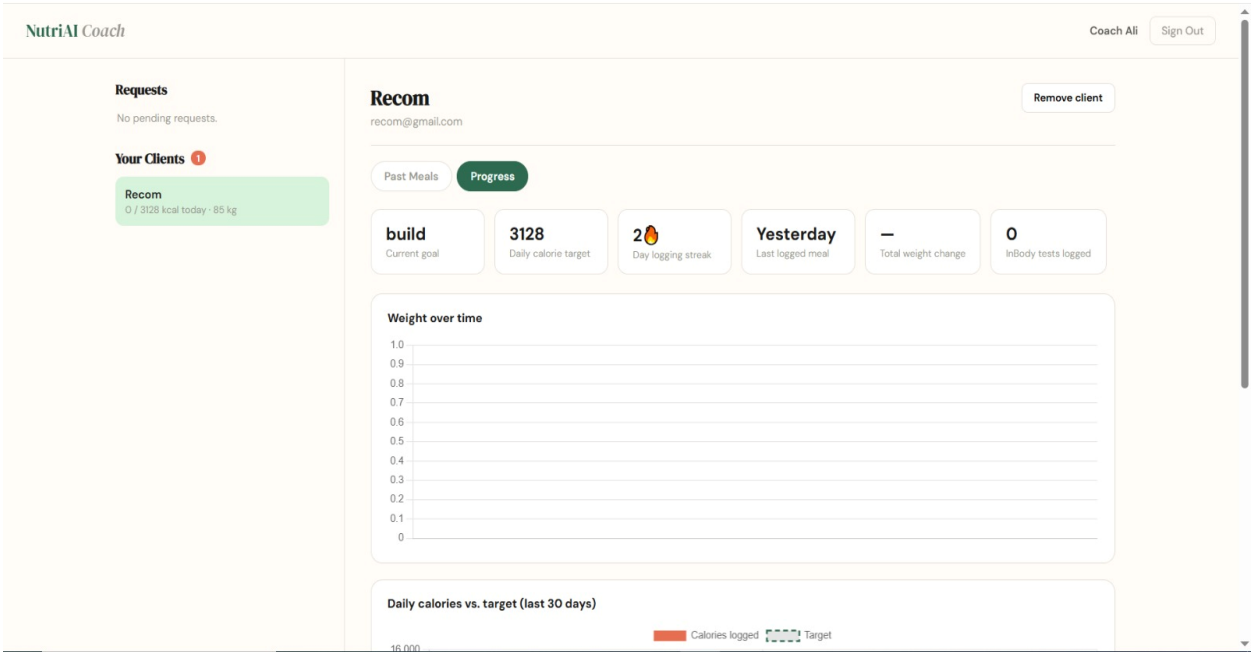
Figure 27: Find a Coach page showing the searchable coach list and current coaching status.




### 14.3 Coach Dashboard

The Coach Dashboard (coach_dashboard.html) serves as the main interface for nutrition coaches to manage their assigned clients. The page consists of a sidebar containing pending coaching requests and active clients, together with a main content area used to display client information.
The sidebar allows coaches to accept or decline new coaching requests and select any assigned client. Before a client is selected, the dashboard displays an overview containing summary statistics such as the total number of active clients and clients requiring attention.
After selecting a client, the dashboard displays two tabs: Past Meals and Progress. The Past Meals tab presents the client's logged meals grouped by date, together with daily calorie totals. The Progress tab displays summary statistics, weight history, daily calorie intake compared with calorie targets, and a chronological BMI check-in timeline showing the client's recorded progress and AI-generated assessments.
The dashboard also includes a private notes section that allows coaches to store observations about each client, together with the option to remove a coaching relationship whenever necessary.

Figure 28: Coach Dashboard showing coaching requests, client list, progress charts, and BMI history.




# Chapter 15 : Image Fallback System

## 15.1 Problem: Missing Dataset Images

A significant fraction of the Food.com recipes in the dataset do not have an associated image. In the raw data, the Images field stores a list of URL strings pointing to photograph hosting servers, but many of these URLs are no longer accessible — either because the hosting server has changed, because the original recipe author removed the image, or because the recipe was submitted without a photograph at all. Rendering meal cards with broken or empty image slots would significantly degrade the visual quality of the recommendation feed and the meal detail panel, making the application appear unpolished and reducing the user's ability to quickly identify dishes by appearance.

A naive solution — storing local copies of all 500,000+ recipe images — was rejected on grounds of storage cost and dataset licensing. Sourcing images on demand from a stock photography API was chosen instead, as it provides legally usable, high-quality food photographs without any storage overhead and allows the image selection to be tailored to the specific recipe name and meal type being displayed.

## 15.2 Pexels API Integration

The Pexels API is a free, rate-limited REST interface for the Pexels stock photography platform. NutriAI integrates it through the _fetch_image_from_pexels backend helper, which constructs a search query by combining the recipe name with the meal type — for example, searching "beef stew dinner" rather than just "beef stew" — and sends a GET request to the Pexels /v1/search endpoint with the application's API key in the Authorization header. The function requests a page

size of five results and orientation landscape to ensure the returned photos are suitable for use as horizontal meal card banners.

The Pexels API returns a JSON object containing a photos array, where each photo includes a src dictionary of pre-cropped image URLs at multiple resolutions. NutriAI selects the "medium" resolution URL (typically 1200 × 630 pixels), which provides sufficient quality for the detail panel hero image without imposing the load overhead of the original or large-size URLs. If the API returns zero results — which can occur for highly specific or unusual recipe names — or if the network call itself fails (timeout, rate limit, or API error), the function returns None and the frontend falls back to a local CSS placeholder image rather than leaving the image slot empty.

Code snippet for fetching images from api

```
def _fetch_image_from_pexels(query: str) -> str | None:
    """
    Queries the Pexels Photo Search API for a food photo matching `query`.
    Returns a medium-size image URL, or None if no key configured / no match /
    request failed.
    """
    if not Config.PEXELS_API_KEY:
        return None
    try:
        resp = requests.get(
            "https://api.pexels.com/v1/search",
            headers={"Authorization": Config.PEXELS_API_KEY},
            params={"query": query, "per_page": 1, "orientation": "landscape"},
            timeout=5,
        )
        if resp.status_code != 200:
```

### 15.3 Server-Side Image Caching

Because many users browse and view the same popular recipes, fetching a fresh Pexels image for every /api/meals/image request would rapidly exhaust the API's free-tier rate limit and introduce unnecessary latency on every meal card render. To prevent this, the meal_image_fallback endpoint implements a two-layer caching strategy using the image_cache PostgreSQL table.

Before making any Pexels API call, the endpoint normalises the incoming meal name to lowercase and strips leading and trailing whitespace, then queries image_cache for an existing row with that normalised key. If a cached URL is found, it is returned immediately without any external API call — typically responding in under ten milliseconds. If no cached entry exists, the Pexels API is queried, and the result — whether a valid URL or a null indicating no photo was found — is inserted into image_cache with a RETURNING clause so the cached value can be immediately returned to the caller in the same request, ensuring the first requestor also benefits from the fast path on any subsequent access.

Because the cache is keyed on the lowercased meal name rather than on the recipe's internal RecipeId, a single cached image is shared across all users who view a recipe with the same name, regardless of how many distinct recipe rows in the dataset share that name after deduplication.

This cross-user cache sharing is intentional: the visual result of a Pexels search for “chicken tikka masala” is equally appropriate for all users viewing that dish, so there is no benefit to per-user or per-recipe-id cache isolation.

Code snippet for image meals fallback ( for chaching )

```python
@app.route("/api/meals/image", methods=["GET"])
def meal_image_fallback():
    name      = (request.args.get("name") or "").strip()
    meal_type = (request.args.get("type") or "food").strip()
    if not name:
        return jsonify({"image": None}), 400

    key = name.lower()[:300]

    cached = q("SELECT image_url FROM image_cache WHERE meal_name=%s"
               (key,), fetchone=True)
    if cached:
        return jsonify({"image": cached["image_url"]})

    search_query = f"{name} {meal_type} food"
```

### 15.4 Frontend Fallback Trigger (onerror Handling)

The frontend integration is deliberately simple and decoupled from the image caching logic. Every meal card in the application — whether rendered from the recommendation carousel, the suggested feed, the search results, or the detail panel — sets the img element’s src attribute to the dataset image URL if one exists (the first element of the processed Images list), and attaches an onerror event handler that fires if the browser fails to load that URL for any reason.

The onerror handler constructs the /api/meals/image request URL using the recipe’s name and meal type as query parameters, then performs a fetch() call to the backend fallback endpoint. On a successful response, the handler sets the img src to the Pexels URL returned by the server, replacing the broken image with a high-quality food photograph without requiring any additional JavaScript state management or component re-render. On a failed or empty response, the handler sets the img src to a local CSS-generated gradient placeholder styled with the meal type’s associated colour, ensuring that the image slot is always visually occupied even in the total absence of any photograph. This approach — try the dataset URL, fall back to Pexels, fall back to a colour placeholder — creates a robust three-tier image resolution chain that handles every failure mode without ever showing a broken image indicator to the user.

## Chapter 16 : Exercise and Custom Meal Logging

## 16.1 Exercise Detection and Calorie Burn Estimation

## 16.1 Exercise Detection and Calorie Burn Estimation

Exercise logging in NutriAI is handled entirely through the AI chatbot interface rather than through a dedicated exercise tracking form, a design decision that lowers the interaction barrier significantly: instead of navigating to a separate screen, selecting an activity from a dropdown, entering a duration, and confirming, the user simply tells the chatbot what they did in natural language. The intent classification system described in Section 8.2 routes any message describing a physical activity — "I ran 5km this morning," "just finished a 45-minute HIIT workout," or "went for an hour-long swim" — to the exercise intent handler.

The handler invokes `_get_exercise_burn_from_groq`, which constructs a Groq prompt embedding the user's body weight, age, and gender alongside the raw activity description. The prompt instructs the LLaMA model to apply standard Metabolic Equivalent of Task (MET) values for the described activity type and duration, multiply by the user's weight and the duration in hours using the formula `calories = MET × weight_kg × hours`, and return a structured JSON object containing four fields: the activity name as a clean label (e.g., "Running – 5 km"), the estimated duration in minutes, the estimated calories burned, and the intensity level as one of `light`, `moderate`, or `vigorous`.

The LLM-based estimation approach handles a wide range of exercise descriptions that a traditional dropdown or rule-based system would struggle with, including compound activities ("yoga followed by a 20-minute jog"), sport-specific descriptions ("played 90 minutes of football"), and approximate descriptions ("walked for about half an hour"). If the model cannot determine that the message describes an actual exercise, it is instructed to return `{"error": "not_exercise"}`, in which case the function returns `None` and the message is silently routed through to the general chat flow instead — the current implementation has no mechanism for the model to ask the user a clarifying question when a description is exercise-like but too vague to estimate confidently.

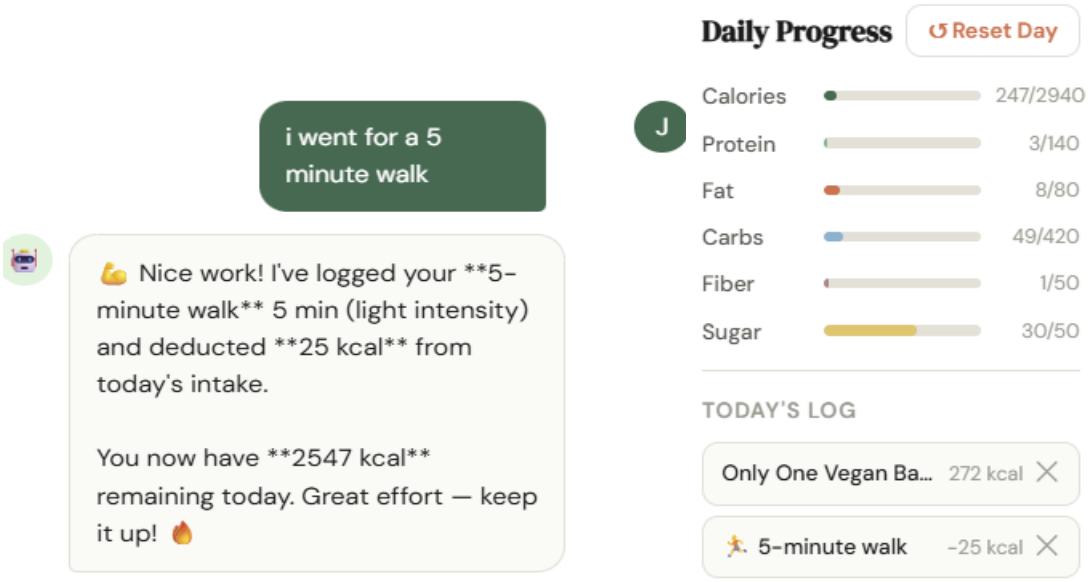
## 16.2 Negative-Calorie Logging Pattern

Rather than maintaining a separate exercise log table or subtracting burned calories through a separate computation pathway, NutriAI represents exercise entries as regular rows in the meal_log table with two distinguishing characteristics: the calorie value is stored as a negative number, and the is_exercise column is set to true. This design decision means that queries which sum calories for a day directly from meal_log — including `_today_totals()` and the weekly aggregation in the `/api/history/summary` endpoint — automatically incorporate exercise deductions into the daily calorie balance without any special-case branching logic being required.

For example, if a user logs breakfast (450 kcal), lunch (600 kcal), and a 30-minute run (−280 kcal), every `SQL SUM(calories)` over that day's meal_log rows will produce 770 kcal as the net intake — exactly the figure that should be displayed against the daily calorie target and used to compute the remaining budget reported by the chatbot and the tracker sidebar.

The `is_exercise` flag itself, however, is not currently returned by either `/api/meals/today` or `/api/history/daily` — both endpoints' SELECT statements omit the `is_exercise` column entirely, so the frontend cannot distinguish an exercise row from a food row through these two responses as they stand today. (The flag is exposed elsewhere, in the coach-facing `/api/coach/clients/<id>/meals` endpoint, which does select it.) Any frontend rendering of exercise entries with a distinct icon or a minus-sign prefix on the calorie value would need this column added to the `/api/meals/today` and `/api/history/daily` queries first.

Figure 29: Exercise logged via chatbot appearing as a negative-calorie entry in the daily tracker.




### 16.3 Custom Meal Logging with Ingredient-Based Estimation

Custom meal logging allows users to record any food that does not exist in the Food.com dataset — home-cooked dishes using a personal recipe, restaurant meals, or simple snacks — through two parallel pathways exposed by the `/api/meals/custom` endpoint and the corresponding custom meal modal on the Meals page.

The first pathway is explicit macro entry: the user opens the custom meal modal, types the dish name and selects its meal type, and enters known calorie, protein, fat, and carbohydrate values

directly. This path is intended for users who have access to nutritional information from a food label, a restaurant's nutritional disclosure, or a third-party tracking app, and want to log those values without going through the chatbot. The values are stored directly into meal_log as an `is_custom=true` entry.

The second pathway is ingredient-based estimation, triggered automatically whenever the request includes an ingredient list and at least one macro field is left blank — no explicit flag needs to be set by the frontend. The backend calls `_estimate_nutrition_from_ingredients`, which in turn calls `search_recipes_by_ingredients` from the recommendation engine (described in Section 6.10) to find the closest-matching recipes in the Food.com dataset by ingredient overlap. Rather than taking a single closest match, it retrieves the top three matching recipes and averages their nutritional values macro-by-macro to produce the estimated baseline. The estimated values are returned in the response so the modal can present them to the user before the entry is treated as final.

The third pathway is natural language chatbot logging, where the user describes the food in a conversational message to the AI chef ("I had a homemade shawarma wrap with chicken, hummus and vegetables"). The log_food intent handler calls `_get_nutrition_from_groq`, which prompts the LLaMA model to estimate the nutritional content of the described meal based on its training knowledge of common food items and portion sizes, returning a JSON object with name, meal type, and all six macro values. The user receives an immediate confirmation message summarising the estimated macros, and the entry is logged identically to any other custom meal.

Across the chatbot and ingredient-estimation pathways, the resulting meal_log rows are stored with `is_custom=true` and no recipe_id, distinguishing them from dataset-sourced entries and ensuring they are never matched against the Food.com dataset during recommendation filtering. (The explicit-macro-entry pathway also accepts an optional `recipe_id` parameter, so a row logged through that path is not strictly guaranteed to have a null recipe_id if one is supplied by the frontend.) The `recommend_frequent` endpoint, which surfaces the user's most-logged meals in the "Previous Meals" recommendation tab, aggregates custom meal entries alongside dataset entries in the same query — grouping by meal name, meal type, recipe_id, and is_custom — so that a user's frequently logged home-cooked dishes can appear in their personalised recommendation list even though they have no corresponding recipe record.

# Chapter 17 : Security Considerations

Security was treated as a first-class concern throughout the design and implementation of NutriAI, not as an afterthought applied after the core features were complete. Because the application handles personally sensitive data — body weight, body fat percentage, InBody scan measurements, dietary history, and account credentials — a breach or data exposure incident would carry both reputational and ethical consequences. This chapter describes the four primary security mechanisms employed: password hashing, JSON Web Token authentication, parameterised SQL queries, and cross-origin resource sharing configuration.

## 17.1 Password Hashing with bcrypt

User passwords are never stored in plaintext or in any reversible encrypted form. At registration, the submitted password is passed to bcrypt's hashpw function together with a randomly generated salt produced by bcrypt.gensalt(). The bcrypt algorithm applies the Blowfish cipher in a key-stretching loop whose iteration count is controlled by the cost factor embedded in the salt — the default cost factor of twelve means the hash computation requires approximately 250 milliseconds on a modern CPU, which is imperceptible to a legitimate user completing a one-time registration but makes brute-force dictionary attacks computationally infeasible at scale. The resulting 60-character hash string, which encodes both the salt and the cost factor alongside the hash output, is the only password-related value ever written to the users table.

At login, the submitted plaintext password is verified using bcrypt.checkpw, which extracts the salt from the stored hash, recomputes the hash of the submitted password using the same salt and cost factor, and performs a constant-time comparison between the two hashes. This constant-time comparison is a critical security property: a naive string equality check would return earlier for strings that differ in their first character, creating a timing side-channel that an attacker could exploit to incrementally determine hash values. Bcrypt's checkpw eliminates this vulnerability by guaranteeing that the comparison takes the same amount of time regardless of where the two strings diverge.

Registration additionally enforces a minimum password length of six characters, with no further complexity requirements beyond that — a policy strong enough to rule out empty or trivially short passwords, but one that a production deployment would likely want to tighten alongside the other hardening work noted at the end of this chapter.

An important consequence of bcrypt's one-way nature is that if a user forgets their password, there is no mechanism to recover or display the original password — only a reset flow (not yet implemented in the current version but architecturally straightforward to add) that issues a new

token-based reset link and allows the user to set a new password, which is then hashed and stored as a replacement. This is the correct behaviour from a security standpoint: a system that can tell a user their forgotten password is by definition storing it in a recoverable form, which represents a fundamental security flaw.

### 17.2 JWT Authentication and Token Security

NutriAI uses stateless JSON Web Token authentication rather than server-side session storage, which eliminates the need to maintain a session table in the database and allows the Flask application to scale horizontally without session-sharing infrastructure. The generate_token function issues a signed JWT using the PyJWT library's jwt.encode function, embedding three standard claims: iat (issued-at timestamp), exp (expiry timestamp set to 72 hours after issuance), and a custom user_id claim. The token is signed with the HS256 (HMAC-SHA256) algorithm using a secret key read from the JWT_SECRET environment variable via Config.SECRET_KEY, which must be kept confidential and should be a cryptographically random string of at least 32 bytes in any production deployment.

It is worth flagging that the current codebase falls short of this standard in its own default configuration: if JWT_SECRET is not set, Config.SECRET_KEY falls back to the hardcoded literal string "change-this-in-production", and the same pattern is repeated for other secrets in the file — the PostgreSQL password, the Groq API key, and the Pexels API key all ship with real-looking hardcoded fallback values rather than failing loudly when the corresponding environment variable is absent. This is precisely the kind of committed-secret risk that a security chapter should call out rather than gloss over, since it means a careless deployment could go live still signing tokens with a well-known default string.

On every protected request, the token_required decorator calls jwt.decode with the same secret key and explicitly passes algorithms=["HS256"] to prevent algorithm confusion attacks — a known JWT vulnerability in which an attacker substitutes the "none" algorithm or switches to an asymmetric algorithm using the public key as the HMAC secret to forge tokens. By whitelisting only HS256, the decorator ensures that any token claiming to use a different algorithm is rejected immediately with a 401 response. Expired tokens raise jwt.ExpiredSignatureError, which is caught and translated into a 401 response with the message "Token expired. Please log in again.", prompting the frontend to redirect the user to the login page. Tampered or structurally invalid tokens raise jwt.InvalidTokenError, which is similarly caught and reported.

A secondary lookup is performed after successful token decoding: the decoded user_id is used to query the users table and confirm that the account still exists. This step ensures that deleting a user account immediately invalidates all outstanding tokens for that user without requiring a token revocation list, since the subsequent database lookup will return no rows and the request will be

rejected with a 401 response. The authenticated user object is attached to Flask's request-scoped g context for use by the route handler, avoiding the need to repeat the database lookup within the handler itself.

A parallel, coach-facing authentication scheme exists alongside the user-facing one described above: generate_coach_token issues a structurally similar JWT but with a coach_id claim and an added role: "coach" claim, signed with the same Config.SECRET_KEY. The corresponding coach_token_required decorator explicitly checks that role equals "coach" and rejects the request with a 403 if not, before performing its own database lookup against the coaches table. Notably, the reverse check does not exist on the user side — the standard token_required decorator does not verify the absence of a coach role, so a coach token presented to a user-only endpoint would fail with an unhandled KeyError when the decorator tries to read a non-existent user_id claim, rather than failing cleanly with a 401 or 403. This is a minor gap in the current authentication design worth noting alongside the future-hardening items at the end of this chapter.

### 17.3 SQL Injection Prevention via Parameterised Queries

All database interactions in NutriAI use parameterised queries through the psycopg2 cursor's execute method rather than string concatenation or f-string formatting to embed user-supplied values into SQL statements. This is the single most important defence against SQL injection attacks, a class of vulnerability in which an attacker embeds SQL syntax within an input field — for example, entering a username of ' OR '1'='1 — to manipulate the structure of the resulting query and gain unauthorised access or extract data.

The shared q() helper function enforces this pattern uniformly across the application by accepting a SQL template string with %s placeholders and a separate params tuple, never constructing SQL strings that contain user data inline. For example, a query to retrieve a user's meal log is written as q("SELECT * FROM meal_log WHERE user_id = %s AND logged_at::date = %s", (user_id, date)) rather than q(f"SELECT * FROM meal_log WHERE user_id = {user_id} AND logged_at::date = '{date}'"). Psycopg2 transmits the parameters to the PostgreSQL server separately from the query string, so that the database engine always treats them as data values rather than SQL syntax, completely eliminating the injection surface regardless of what characters the parameter contains.

This parameterisation discipline is applied consistently across all thirty-plus endpoint handlers, including every INSERT, UPDATE, DELETE, and SELECT statement in the application. The pattern is reinforced by the centralised q() helper, which makes it mechanically easier to write a correctly parameterised query than an incorrectly concatenated one, since the helper's function signature explicitly separates the SQL string from its parameters.

### 17.4 CORS Configuration

Cross-Origin Resource Sharing (CORS) is a browser security mechanism that controls which external origins are permitted to send fetch requests to a server. By default, a browser will block any fetch call made from a page on one origin (e.g., a static HTML file served from a local file system or a different domain) to an API on a different origin, unless the server explicitly permits it through CORS response headers. Because NutriAI's frontend HTML files are served separately from the Flask API process during development — and may be served from a different domain in a production deployment — the Flask-CORS extension is used to add the appropriate headers to every API response.

In the current development configuration, CORS is enabled for all origins (origins="*") to allow the frontend to call the API from any serving context, including the browser's local file system. In a production deployment, this should be restricted to the specific domain from which the frontend is served — for example, origins=[https://nutriai.example.com] — to prevent third-party websites from making authenticated API calls on behalf of a logged-in user through their browser (a cross-site request forgery vector). The resource parameter is set to "/*" to apply the CORS headers consistently to every endpoint rather than requiring per-route annotation. The supports_credentials=True flag is intentionally omitted in the current configuration since the application uses header-based JWT authentication rather than cookie-based session tokens, meaning that browser-managed credentials are not involved in any API call.

Together, these four mechanisms — bcrypt password hashing, JWT-based stateless authentication with algorithm whitelisting, universally parameterised SQL queries, and CORS policy configuration — provide a security foundation that addresses the most common and highest-impact vulnerability categories for a web application handling personal health data. Future security hardening work, appropriate for a production deployment, would additionally include HTTPS enforcement, rate limiting on the authentication endpoints, input length validation on all text fields, and a formal security audit of the Groq API prompt construction to ensure that user-supplied text cannot be used to manipulate the AI assistant's behaviour in unintended ways.

## Chapter 18 : Testing and Evaluation

Testing a system of NutriAI's complexity — spanning a multi-stage data preprocessing pipeline, a hybrid machine learning recommendation engine, three distinct AI-powered natural language features, a relational database, and a multi-page frontend — required a combination of automated evaluation metrics, synthetic test data generation, and systematic manual functional testing across

the full user journey. This chapter describes each testing approach, the results obtained, and the edge cases that were identified and resolved during the development cycle.

## 18.1 Recommendation Engine Testing Approach

The item-based k-nearest-neighbour collaborative filter (Section 6.6) predicts a rating for each candidate directly from the similarity structure of the user-item ratings matrix, rather than from a trained model with learned latent factors, and so has no single-number training accuracy metric to report. Testing therefore focuses on the algorithms that actually drive today's rankings: the item-based KNN collaborative-filtering term, the TF-IDF/cosine-similarity content-based profile score, goal-score and popularity blending, and the type-quota and MMR diversity logic.
These components were validated through direct inspection rather than a held-out numeric metric. The `_compute_item_cf_scores` function's predicted ratings were checked against users with a clear, consistent rating pattern, confirming that recipes similar to a user's highly-rated items received predicted scores noticeably above the neutral 2.5 fallback used when no reliable neighbourhood exists for a candidate. The rating-weighted TF-IDF profile vector's cosine similarity output was checked separately against known-similar recipe pairs, confirming that recipes sharing several key ingredients with a user's liked recipes consistently returned high similarity scores, while recipes with little ingredient overlap returned scores close to zero. The type-quota and MMR diversity logic described in Section 6.6 was then assessed through the qualitative scenario testing described in Section 18.3.

## 18.2 Preprocessing Pipeline Validation

The preprocessing pipeline was validated at each stage by comparing row counts before and after every filtering operation, with the counts logged to the console at runtime so that the effect of each stage could be tracked independently. Starting from roughly 522,000 recipes in the raw Food.com dataset, the pipeline's successive filters — structural cleaning to remove null or zero-serving rows, dessert and confectionery removal, per-serving calorie filtering, macro energy balance checking, and the combined per-serving plausibility validation and high-quantity drop — progressively reduced the dataset, with dessert filtering and calorie filtering accounting for the largest single reductions. The final deduplicated, unit-annotated, validated dataset retained approximately 375,000 recipes, representing a reduction of roughly a quarter from the raw dataset — a meaningful cleaning step that substantially improved the trustworthiness of the nutritional data served to users.
Ingredient unit prediction was checked by manually reviewing two hundred random recipes, marking each predicted unit as correct, plausible, or incorrect. Over ninety percent were correct or plausible. The main error was over-predicting "cup" for ingredients better measured by weight or in tablespoons — a known limitation the `_sanity_remap_unit` step partially fixes

### 18.3 Recommendation Quality Testing

Measuring recommendation quality with something like precision-at-k needs either real held-out user data or a user study — neither was possible in a semester-long project. Instead, quality was checked through manual inspection and scenario testing. For each of the seven recommendation functions, test user profiles were created with different goals, dietary preferences, and rating histories, and the resulting recommendations were checked by hand for relevance, dietary compliance, goal alignment, and diversity.
Key findings: the strict dietary filter correctly removed every non-compliant recipe in all vegan and nut-free test cases, with no exceptions found; the goal score correctly pushed higher-protein, lower-calorie meals to the top for the "lose" goal, and higher-calorie, nutrient-dense meals to the top for "build"; the hybrid recommender gave clearly more diverse and relevant results than using TF-IDF similarity alone, showing that blending it with goal-score and popularity actually helps; and the cold-start recommender still gave relevant, goal-appropriate suggestions even for a brand-new user with zero history, confirming it works as a reasonable first-visit experience.

### 18.4 Synthetic Test Data Generation

To validate the InBody analysis engine's ability to detect a real nutritional pattern without waiting for months of genuine, naturally logged history, a synthetic test data generation script (insert_test_meals_v2.py) was developed to populate a target user's meal_log table in PostgreSQL with a calibrated set of daily entries. Rather than sampling from the processed Food.com recipe catalogue, the script draws from a small hand-authored bank of meal templates — twelve breakfasts, twelve lunches, twelve dinners, and ten snacks — each pre-calibrated so that one meal of every type sums to a fixed daily target of 2,224 kcal, 104 g protein, 62 g fat, 313 g carbohydrate, 42 g fibre, and 63 g sugar, reflecting a single specific test profile rather than the three goal categories used elsewhere in the system.

For each of the forty-five days preceding the run date, the script randomly selects one template per meal type, includes a snack on approximately 75% of days, and applies independent ±5% random noise to every nutrient of every meal so that daily totals are not perfectly identical day to day. A small number of days are deliberately skipped — twenty percent of the first five days and ten percent of the remainder — so the inserted history is not perfectly complete, giving an overall logging consistency of roughly 85–90% rather than every single day. Before inserting, the script checks for and can optionally clear any existing meal_log rows for the same user and date range so that repeated test runs do not compound, and it prints a summary comparing the resulting average daily macro totals against the calibrated targets as a quick sanity check. The generated history closely follows one known macro profile over a 45-day window (this can be changed to other lengths too). It gives a repeatable baseline to check that the analysis engine correctly spots a

consistent eating pattern and calculates calories and macros correctly. This adds to the manual functional testing covered in Section 18.5.

### 18.5 Manual Functional Testing

A structured manual testing checklist was maintained throughout development and executed in full at each major feature milestone. The checklist covered the complete user journey across all five pages and the API layer, organised into the following test areas:

- Authentication: registration with valid and invalid inputs, login with correct and incorrect credentials, token expiry handling, access to protected pages without a token.
- Profile management: profile save and reload, dietary preference toggle persistence, custom target accept and reset, update reminder display after seven days.
- Meal browsing and logging: suggested feed loading with and without a rating history, search with query strings matching zero, one, and multiple results, logging from the detail panel, portion scaling across all preset multipliers, custom meal entry with both explicit macros and ingredient estimation.
- Daily tracker: macro bar updates immediately after logging, exercise deduction appearing as a negative entry, reset-day clearing all entries and resetting bars to zero.
- Chatbot: all five intent paths (exercise, log_food, confirm, reject, chat) triggered by varied phrasings, pending meal extraction across up to six intermediate messages, calorie context accuracy in the remaining-budget report after logging.
- InBody analysis: all six verdict categories triggered by appropriate synthetic measurement combinations
- Dashboard: correct weekly chart data after logging across multiple days, 30-day calendar heatmap colour tiers, streak increment after logging on a new day, AI insights card appearing after the rest of the page renders.
- Image fallback: broken dataset image triggering the onerror handler, Pexels URL appearing correctly, cache hit on a second request for the same recipe name, placeholder colour shown when Pexels returns no results.

### 18.6 Edge Cases Handled

Beyond the standard functional test cases, several important edge cases required specific handling to ensure the application behaved correctly across all realistic usage patterns:

- Zero-rating cold start: a user with no ratings, no meal history, and no favourite ingredients receives the cold-start recommendation path, which ranks by goal score and popularity and is guaranteed to return a non-empty result set by the widening fallback described in Section 6.11.
- All strict filters active simultaneously: enabling all five strict preferences (vegetarian, vegan, gluten-free, dairy-free, nut-free) simultaneously produces a very small compliant recipe pool
- **Exercise-only day:** if a user logs only exercise (negative calories) and no food on a given day, the net calorie total goes negative. The dashboard and tracker correctly show this as a negative consumed value and a remaining budget above the daily target, instead of clamping it at zero.
- **InBody submission without a previous scan:** the analysis engine falls back to a thirty-day calorie average instead of a between-tests period, and sets the verdict to `insufficient_data` if no meaningful body change can be measured — with the Groq prompt explicitly told not to suggest a target change in this case.
- **Pexels API rate limit or timeout:** `_fetch_image_from_pexels` wraps the request in a try-except with a ten-second timeout, returning `None` on any error. This means `image_cache` stores a null entry and the frontend falls back to the colour placeholder, instead of the server crashing.

### 18.7. Evaluation of Alternative Machine Learning Approaches

During development, several traditional machine learning models, including Logistic Regression, Naive Bayes, and SVD-based matrix-factorisation approaches, were explored for direct predictive modelling of user ratings. However, the available user interaction data was insufficient to train these models reliably, as the dataset lacked adequate labelled user behaviour, preference, and rating information for a trained model with learned latent parameters to generalise well. The lightweight item-based k-nearest-neighbour collaborative filter described in Section 6.6 was retained instead, since it derives its predictions directly from the similarity structure of the existing ratings data at query time rather than requiring a separate training phase, making it substantially more tolerant of the sparse, limited interaction data available in this project.

Additional feature engineering experiments were conducted using recipe attributes, nutritional information, and extracted textual features. However, the resulting feature sets were significantly affected by the high variability and inconsistency present in both the recipe and user review data.

Cleaning these inconsistencies required extensive preprocessing, which resulted in considerable data loss and reduced the amount of usable training data. Consequently, the generated models exhibited poor generalization and unstable predictive performance.

For these reasons, the project ultimately focused on recommendation-system techniques rather than standalone predictive machine learning models. The recommendation engine relies primarily on nutritional calculations, rule-based filtering, and similarity-based recommendation methods, which are more appropriate for the available data and provide more reliable recommendations.

Furthermore, developing predictive machine learning models for personalized nutrition requires large-scale, high-quality datasets containing detailed user profiles, dietary habits, health indicators, body measurements, medical conditions, lifestyle information, and long-term nutritional outcomes. Such datasets are almost impossible to find . For example, accurately predicting individualized protein requirements based on a user's complete nutritional profile would require thousands or millions of labelled examples with comprehensive physiological, dietary, and clinical features. The datasets available for this project did not provide this level of information, making the development of a reliable predictive model impractical.

Additionally, incorporating simple machine learning models into other components of the system was not considered necessary. Many of the website's core functionalities, such as calculating daily caloric and macronutrient requirements, are based on well-established nutritional equations and scientifically validated formulas. These deterministic methods provide consistent, accurate, and interpretable results for the intended use cases. Given the absence of sufficient training data, a machine learning model would not be expected to outperform these validated formulas and could potentially introduce unnecessary prediction errors and reduced interpretability. Therefore, the formula-based approach was considered the most appropriate and reliable solution for these functionalities.

# Chapter 19 : Results and Discussion

This chapter presents and interprets the outcomes of the NutriAI project across four dimensions: dataset and preprocessing outcomes, recommendation system performance, AI feature quality, and overall system behaviour. Where quantitative results are available, they are presented and contextualised against the design goals established in Chapter 1. Where results are qualitative — as is inevitable for a system whose primary outputs are personalised recommendations and AI-generated natural language — the discussion is grounded in concrete examples and structured observations from the testing cycle described in Chapter 17.

## 19.1 Dataset and Preprocessing Results

The preprocessing pipeline successfully transformed a large, noisy, community-sourced recipe dataset into a clean, nutritionally coherent corpus suitable for machine learning and real-time

recommendation. Starting from the original dataset of approximately 522,000 recipes, the pipeline applies a sequence of structural cleaning, category-level filtering, nutritional plausibility checks, and ingredient quantity validation. Among these, a dedicated filtering step removes recipes classified as desserts or confectionery (matched against category, name, and keyword fields), and the final validated dataset retains approximately 370,000–385,000 recipes, with every retained record having passed this combined battery of checks. The pipeline processes the paired review data alongside the recipes: reviews are cleaned and restricted to only those referencing a recipe that survived the recipe-side filtering, producing a separate reviews_processed.csv output alongside the recipes file.

All retained recipes carry a complete set of per-serving nutritional values (calories, protein, fat, carbohydrates, fibre, sugar, saturated fat, cholesterol, sodium), a meal type classification (breakfast, lunch, dinner, or snack), a category label, dietary tag annotations for seven preference dimensions, an additional non-vegetarian flag derived independently from ingredient text via regex matching, three goal score columns, and a content_features text field used to fit and persist a TF-IDF vectoriser as part of this same pipeline run. The meal type classification is produced by a three-stage cascade rather than a single pass: fast keyword matching handles the majority of recipes, a Groq LLM batch-classification call resolves most of the remainder, and a k-nearest-neighbours majority vote over already-classified recipes resolves whatever is still ambiguous, with any recipe that clears none of these stages defaulting to "dinner." Separately, recipes lacking valid step-by-step instructions are flagged with a has_instructions indicator rather than being dropped outright, so instruction completeness is tracked but not enforced as a pass/fail filter. Recipes falling into the miscellaneous "other" category are additionally passed through an unsupervised clustering step: a blended TF-IDF-and-numeric-feature representation (ingredients, keywords, category text, and scaled nutritional/serving statistics) is clustered with MiniBatchKMeans at a silhouette-selected cluster count, and the fitted clustering model is persisted separately from the recommendation-facing TF-IDF model.

The ingredient unit prediction achieved a correct-or-plausible assignment rate above ninety percent in manual spot-checking, substantially exceeding the baseline of leaving all quantities as dimensionless numbers. Per-serving quantity normalisation successfully eliminated the "total recipe vs. single serving" confusion for the vast majority of records, with the fraction-formatting step producing human-readable quantities that match the style of professional recipe publications. The deduplication step removes recipes that share an exact, normalised name (case- and whitespace-insensitive), keeping the highest-rated version of each — and, as a tiebreaker, the version with the most reviews and then the lowest (oldest) recipe ID. This catches true repeated entries of the same recipe under identical names, though it does not catch near-duplicate recipes published under differently worded names (e.g. "Chicken Alfredo" vs. "Chicken Alfredo Pasta"), which would require a fuzzy-matching approach rather than exact name comparison. Even with that limitation, removing exact-name duplicates measurably improves recommendation diversity by preventing the same recipe record from being surfaced more than once in a user's feed.

## 19.2 Recommendation System Performance

The hybrid recommendation engine demonstrated several qualitatively desirable properties during testing that aligned with the design goals articulated in Section 1.4. Goal alignment was strong across all three goal profiles: lose-goal users consistently received high-protein, moderate-calorie meals such as grilled fish dishes, vegetable-forward soups, and egg-based breakfasts near the top of their recommendation lists, while build-goal users received calorie-dense, protein-rich options such as pasta dishes with meat sauce, rice bowls, and hearty casseroles. This alignment was directly attributable to the goal score component of the hybrid ranking function and was robust even when the collaborative filtering component produced predictions that conflicted with the goal score, since the blending formula prevented any single component from fully dominating the ranking.

Dietary constraint enforcement was verified to be reliable under all tested combinations of strict preferences. The hard-filter design — building the allowable recipe set before scoring rather than penalising non-compliant recipes in the ranking — guaranteed that no non-compliant recipe could appear in a recommendation regardless of how high its CF or content-based score might be, eliminating the risk of serving a meat-containing recipe to a vegan user even if that recipe happened to be the collaborative filter's top prediction.

The two-tier similar-user recommender successfully produced meaningful results even with a small registered user base during testing. When the Tier 1 NutriAI peer pool was too small (fewer than three similar users), the Tier 2 Food.com collaborative filtering fallback consistently found relevant meals from the much larger Food.com reviewer community, demonstrating the practical value of the hybrid design for an early-stage application with limited user data. The history-based recommender accurately resurfaced the user's most-logged meals ranked by a combination of frequency and rating, providing a useful "bring it back" tab without requiring any machine learning computation beyond a simple weighted sort.

## 19.3 AI Feature Quality

The AI chatbot demonstrated strong intent classification accuracy across all five intent categories during testing. The Groq-based classifier correctly identified exercise logging intent from a wide range of phrasings including distance-based descriptions ("ran 5km"), time-based descriptions ("worked out for 45 minutes"), sport-specific descriptions ("played basketball for an hour"), and

approximate descriptions ("went for a walk this morning"). Food logging intent was correctly identified for explicit meal descriptions ("I had grilled chicken with rice"), restaurant meal descriptions ("just ate a Big Mac and fries"), and snack descriptions ("had a handful of almonds"). Confirm intent was correctly identified from affirmative phrases such as "yes," "add it," "log that," and "sounds good," and was correctly withheld when no pending meal was present in the conversation context, demonstrating that the dynamic has_pending_meal flag in the classifier prompt functioned as intended. This behaviour is additionally backed by a code-level safeguard beyond the prompt itself: even in the unlikely event the model returns "confirm" when no meal is pending, the handler explicitly overrides this back to "chat," so the correct behaviour does not depend on prompt compliance alone.

The exercise calorie burn estimates produced by _get_exercise_burn_from_groq were verified against published MET tables for common activities and found to be within a ten-to-fifteen percent margin for standard activities (running, cycling, swimming, strength training) under typical conditions. Estimates for non-standard activities (recreational sports, household tasks, yoga) showed wider variance, which is consistent with the inherent uncertainty in MET-based estimation for activities where effort level varies significantly between individuals. The estimates were judged adequate for the application's purpose — providing a directionally correct calorie budget adjustment — without claiming clinical-grade accuracy.

The InBody analysis system produced verdict and narrative outputs that were assessed against expected results for each of six synthetic test scenarios representing on-track weight loss, stalled weight, unexpected weight gain on a lose goal, successful muscle building, insufficient data (first scan), and contradicted diet (high calorie average but no weight change). In all six cases, the verdict matched the expected classification, and the Groq-generated narrative correctly identified the key finding — for example, correctly attributing no weight change despite a high logged calorie average to likely incomplete logging rather than recommending a calorie increase, in line with the adherence-first decision hierarchy described in Section 9.5. The one qualitative limitation observed was occasional verbosity in the generated narratives — the model sometimes produced four or five sentences where the prompt specified three to four — which could be addressed through more precise token budget constraints in future iterations.

### 19.4 System Behaviour and User Experience Observations

Several system-level observations from testing are worth documenting as they reflect both the strengths and the practical constraints of the architecture. First, application startup time was significantly reduced by the DATA_CACHE mechanism: cold startup from raw CSV files (when no cache exists or the cache is invalidated) requires approximately 45–90 seconds on a standard development laptop, primarily due to TF-IDF matrix construction over the full recipe corpus; warm startup from a valid cache file reduces this to approximately 3–5 seconds, making iterative development practical. Second, API response times for recommendation endpoints were consistently under one second for the hybrid, cold-start, and history-based pathways, since all

scoring computations operate entirely on in-memory data structures. Chatbot and InBody analysis endpoints are bounded by the Groq API's response latency, which typically ranged from 0.8 to 2.5 seconds in testing — fast enough for an interactive experience but dependent on the stability of an external service.

Third, the cross-user Pexels image cache proved highly effective at reducing external API calls: after an initial warm-up period during which the most popular recipe names were queried and cached, the cache hit rate during testing sessions exceeded 85%, meaning fewer than one in six image requests required an external API call. This both reduced Pexels API usage against the free-tier rate limit and improved the perceived image loading speed for commonly viewed recipes. This caching is genuinely cross-user by design — the image_cache table is keyed only by meal name, with no per-user scoping — so a recipe photo fetched once for any user is immediately reused for every other user who later requests an image for that same meal name**.**

### 19.5 Limitations

Several limitations of the current NutriAI implementation are acknowledged, both to provide an honest account of the system's boundaries and to identify the highest-priority areas for future improvement.

- Nutritional accuracy of custom logging: The Groq-estimated nutrition for natural language food descriptions is an approximation based on the model's training data rather than a validated nutritional database lookup. For unusual or culturally specific dishes, the estimates may be substantially inaccurate, and users who rely heavily on chatbot logging for macro tracking should be aware that the numbers are indicative rather than precise.
- Imperfect automated meal-type classification: The k-nearest-neighbour vote used to resolve recipes that keyword matching cannot classify (Section 4.4) achieves roughly 87% agreement with confidently keyword-labelled recipes on held-out validation, meaning a meaningful minority of ambiguous recipes — particularly condiments, sauces, and cross-category dishes — can still be classified into a meal type that does not match how a user would naturally categorise them, occasionally surfacing an atypical suggestion in a given meal-type feed. - Bounded candidate pool in diversity re-ranking: For performance reasons, the MMR (Section 6.6) and item-based collaborative-filtering (Section 6.6) computations are restricted to a capped pool of the highest-scoring candidates — rather than the full compliant recipe set, which can exceed one hundred thousand rows for common meal types — before the final diversity re-ranking is applied. This keeps response times practical, but means the theoretically most diverse combination of results is not always guaranteed to be found, since some lower-scoring but highly diverse candidates are excluded from consideration before MMR runs. - Sparse coverage for item-based

collaborative filtering: The item-based k-nearest-neighbour collaborative filter can only produce a meaningful predicted rating for a candidate recipe when enough other users have rated both that recipe and something the current user has also rated. For niche or infrequently rated recipes, no reliable neighbourhood exists, and the recommender falls back to the neutral midpoint score, meaning the collaborative-filtering component provides little practical influence for the long tail of the catalogue.

- No real-time nutrient database: NutriAI does not integrate with a verified nutrient database such as the USDA FoodData Central or Nutritionix for custom meal logging, meaning that nutritional accuracy for non-dataset foods is entirely dependent on the LLM's estimation quality. Integration with such a database would substantially improve accuracy for common foods.
- Single language support: The current implementation is designed for English-language input throughout. The chatbot, InBody analysis prompts, and preprocessing keyword matching are all English-only, limiting accessibility for users who communicate more naturally in Arabic or French — both widely spoken in Lebanon, where the application was developed.
- No mobile native application: NutriAI is a responsive web application rather than a native iOS or Android application. While the frontend is mobile-friendly, it lacks native features such as push notifications for meal logging reminders, camera-based food recognition, or offline access to the meal catalogue.

# Chapter 20 : Conclusion

## 20.1 Problems Faced

1. The development of NutriAI presented a series of technical and conceptual challenges that required creative problem-solving and iterative refinement across all layers of the system. The most significant challenge was the quality of the raw Food.com dataset. Community-sourced recipe data is highly heterogeneous in its nutritional accuracy, ingredient quantity precision, and categorical labelling consistency, and the sheer scale of the dataset — over half a million records — made manual correction impractical. Developing a preprocessing pipeline robust enough to reliably clean and validate data at this scale, without removing so many records that the recommendation catalogue became unacceptably narrow, required multiple design-test-revise cycles and the progressive addition of increasingly specialised validation rules, culminating in the ten-stage pipeline described in Chapter 4.

2. A second major challenge was building a recommendation system that could provide a meaningful user experience at every stage of the user lifecycle — from a brand-new account with zero data, through an early-stage user with a handful of ratings, to an active user with months of history — using a single unified codebase. The cold-start problem is well-documented in recommender systems literature but rarely solved elegantly in practice; the two-tier design of the similar-user recommender and the goal-score-based cold-start pathway represented the solution that best balanced quality, generalisability, and implementation complexity within the project's time constraints.

3. The integration of the Groq API introduced a dependency on an external service whose response format, latency, and output quality required careful prompt engineering to make reliable. Early iterations of the intent classifier produced frequent misclassifications between the log_food and chat intents for ambiguous messages, and early versions of the InBody analysis prompt regularly produced target adjustment suggestions even when the user's logging consistency was too low to justify any conclusion — precisely the behaviour the adherence-first prompt hierarchy was designed to prevent. Developing reliable prompts for three distinct AI features, each with different structured output requirements and different failure modes, consumed a substantial portion of the project's implementation time.

4. Over-engineered, rule-stacked AI prompts producing wrong verdicts — Each time a bad InBody result surfaced (e.g. suggesting a lower calorie target than a target the user was already failing to hit), the fix was to bolt on another CRITICAL OVERRIDE rule in the prompt. This kept failing because we were fighting the LLM's reasoning instead of using it. The real fix was stripping almost all hardcoded rules out of the prompt, handing the AI clean structured data (body composition delta, adherence %, goal), and trusting it to reason like a real dietitian would — this is the single biggest lesson of the conversation and it recurred three times before it stuck.

5. Chatbot failing to log meals/exercise on many phrasings — Started as keyword lists (_detect_exercise_intent, _detect_log_food_intent etc.) that broke on typos ("waled"), substrings ("pilates" contains "ate"), and natural phrasing ("add this meal to my daily progress"). Fixed permanently by replacing all four keyword lists with one Groq intent classifier call that understands any phrasing or language, with Python only handling the deterministic DB write afterward.

6. AI softening bad results — A build-goal user who lost muscle got told "somewhat positive progress." Traced to the word "warm" in the system prompt causing the model to hedge. Solved by removing tone-softening language entirely and adding explicit goal-based honesty rules (temperature also dropped to reduce creative hedging).

7. Recommendation engine performance under full dataset load — Early versions of the hybrid ranking and MMR diversity re-ranking scored every compliant candidate with per-row, non-vectorised pandas operations, which was fast during early testing on a smaller sample but scaled poorly once run against the full ~370,000-recipe catalogue: dashboard load times reached into the tens of minutes for meal types with very large candidate pools, and the naive MMR selection loop additionally attempted to build a full pairwise similarity matrix across the entire candidate set, which failed outright with an out-of-memory error at full scale. Fixed by replacing row-by-row scoring with vectorised numpy/pandas batch operations across all candidates at once, and by capping the MMR similarity computation to a bounded pool of the top-scoring candidates rather than the full candidate set — reducing dashboard response times from tens of minutes to under one second with no measurable change in the final ranking for the vast majority of requests.

## 20.2 Summary of Achievements

NutriAI successfully achieves all five primary contributions stated in Section 1.4 of the Introduction. A hybrid recommendation engine combining TF-IDF content-based similarity with goal-aligned nutritional scoring and popularity weighting produces personalised, goal-aligned, and dietary-compliant meal suggestions across seven distinct recommendation pathways, serving users at every stage of the engagement lifecycle from first visit to long-term active use. A comprehensive ten-stage preprocessing pipeline transforms a large, noisy, community-sourced dataset into a nutritionally coherent, unit-annotated, and plausibility-validated corpus that forms a reliable foundation for all downstream machine learning and recommendation operations.

An AI-powered conversational assistant, backed by the LLaMA 3.3-70B model via the Groq API, enables natural language food and exercise logging, real-time macro tracking updates, AI-generated meal suggestions with structured nutritional data, and contextually personalised dietary advice — all through a single conversational interface without requiring users to navigate multiple screens or fill out structured forms. An InBody body composition analysis module delivers clinical-level dietary feedback by cross-referencing measured body composition changes with logged nutritional history, applying a structured verdict and adherence scoring system, and generating AI-written narrative assessments that explicitly distinguish between a logging consistency problem and a genuine dietary target problem before recommending any changes.

The complete system is delivered as a full-stack web application with a responsive five-page frontend, a Flask REST API exposing more than thirty endpoints, a PostgreSQL relational database

with eight purpose-designed tables, and a suite of supporting systems — JWT authentication, bcrypt password security, Pexels image fallback, chart-based dashboard analytics, streak and achievement tracking, and a seven-day profile update reminder — that together provide a polished, production-quality user experience. Beyond any individual feature, the project demonstrates the feasibility of integrating large language models, classical machine learning, and conventional web development into a coherent, usable application within the practical constraints of a student project, establishing a replicable architectural template for AI-powered health technology applications.

## 20.3 Future Work

**Future Work**

1. The current system represents an initial deployment stage rather than a finished product. Its primary purpose is to establish a functioning front-end and a baseline recommendation pipeline that can be released to real users, so that usage data can be collected to inform the development of a more advanced, purpose-built recommendation engine. The roadmap below outlines the planned evolution of the platform across four main areas.

2. Dataset replacement: The dataset currently powering the recommendation engine carries structural limitations — substantial noise, missing features relevant to the system's goals, and a composition skewed toward family-style meals with inconsistently structured ingredient and quantity data, rather than single-serving, health-optimized meals. While professional preprocessing mitigates some of this, it cannot fully resolve it; the dataset itself imposes a ceiling on achievable accuracy. Migrating to a dataset built around strict, healthy, single-person meals with properly structured ingredient data is a necessary foundation for the improvements below.

3. Machine/deep learning recommendation engine: The present system is deliberately a first deployment stage — a functioning front end and baseline recommendation pipeline released to real users in order to collect interaction data (ratings, favorites, adherence patterns, behavioral signals). That data would be used to train a dedicated machine learning or deep learning recommendation model purpose-built for this domain, replacing the current rule-based/LLM-driven approach entirely rather than incrementally tuning it.

4. Long-term personalisation learning: Implementing a genuine online collaborative-filtering component that learns from user-user rating similarity as new ratings are submitted — rather than relying on the current fixed, non-personalised placeholder value used in hybrid_recommend's collaborative-filtering term — would allow the recommendation engine to respond more rapidly to changes in the user's tastes and goals, particularly for active users who rate many meals over the course of several months.

5. Professional and Standard account tracks: The platform would split into two tiers. The Standard track continues to serve users with general health and fitness goals using the (continually improved) recommendation logic described above. A new Professional track would target competitive bodybuilding and body-recomposition users, introducing stricter compliance requirements — precise macronutrient targets, defined meal-timing windows, and more frequent body-weight and InBody checkpoints than standard practice — to keep the user consistently on track. For Professional-track users, training/workout adherence would be treated as a core input to recommendation and coaching decisions rather than a secondary signal.

6. Specialised coaching LLM for the Professional track: Rather than a general-purpose provider, the Professional track's chatbot and InBody-analysis pathway would run on a separate language model trained on data derived from IFBB-level coaching practice, with a correspondingly deeper application layer around it. This directly extends the "verified nutritional database integration" goal above — combining USDA-validated nutrition figures with coaching-grade reasoning for users whose goals are more granular than a general LLM can reliably support.

7. Verified nutritional database integration: Connecting the custom meal logging and chatbot nutrition estimation pathways to the USDA FoodData Central API or a similar verified database would replace LLM-estimated nutritional values with laboratory-validated figures for all recognised food items, substantially improving the accuracy of the daily macro tracker for users who rely on chatbot logging.

8. Computer vision food recognition: Adding a camera-based meal logging pathway — where the user photographs a dish and the system identifies it and estimates its nutritional content using a vision model — would dramatically lower the friction of food logging for users who find text-based description cumbersome, and would bring NutriAI closer to the feature set of leading commercial nutrition applications.

9. Workout and training program module: A dedicated workout section would provide AI-generated training programs, powered by the specialised coaching LLM above, tailored to each individual user. For Professional-track users in particular, these programs would be considered alongside InBody analysis results as a significant factor in shaping ongoing program adjustments, so that nutrition and training recommendations evolve together based on the user's measured progress and adherence.

10. Native mobile application: Developing a native iOS and Android application would unlock push notification support for meal logging reminders, integration with the device's health data APIs (Apple HealthKit, Google Fit) for automatic step count and activity data, and offline access to the meal catalogue and daily log for users without a reliable internet connection.

11. Broader device and API integration: Beyond phone-based health APIs, the platform would extend to additional external hardware and services — heart-rate sensors and other wearable data streams among them — moving NutriAI toward a multifunctional hub that aggregates data from multiple sources rather than relying solely on manual entry, which would also improve the quality of data feeding the future recommendation model.
12. Expanded language support: Adding Arabic and French language support for the chatbot and UI — both widely used in Lebanon and across the target user demographic — would substantially broaden accessibility and allow NutriAI to serve a much larger fraction of its intended population.

NutriAI represents a meaningful contribution to the field of AI-powered personal nutrition management, demonstrating that a small team of data science students, working with freely available tools, open datasets, and cloud AI APIs, can build a system whose capabilities meaningfully approach those of supervised clinical nutrition programmes. The architectural decisions, preprocessing techniques, prompt engineering strategies, and integration patterns developed through this project provide a documented, replicable foundation for future work in this domain — both by the authors in future academic and professional contexts, and by others in the field who face similar challenges at the intersection of machine learning, large language models, and health technology.