
NutriChoice

Nutrition Tracking & Recipe Management Platform

Capstone II — Final Technical Report

Team Members

Joaquin Frangi

Isabella Correa

Pedro Remoir

Luis Duque

Catalina Cisneros

April 2026

Abstract

NutriChoice is a full-stack web application that unifies personalized recipe discovery, weekly meal planning, and macronutrient tracking into a single platform. The system is built on a Python/Django REST API backend and a React 19 TypeScript single-page application frontend, backed by a Supabase-hosted PostgreSQL database and Supabase Storage for user-uploaded media.

Users discover recipes through a personalized feed that ranks them by social proximity (followed creators), ingredient familiarity (overlap with previously tried recipes), and community popularity, with creation date as a stable tiebreaker. A structured weekly meal planner maps chosen recipes to named meal slots across a seven-day grid, while a macro dashboard aggregates daily and weekly nutritional totals against a selected date. Social features—including follows, blocks, and recipe interactions—extend discovery, and a cookbook system enables users to curate recipe collections.

This report documents the technical design and implementation of the system: its architecture, relational data model, REST API, frontend component and state management strategy, feed ranking and macro aggregation algorithms, and deployment infrastructure.

Contents

Abstract	1
1 Introduction	7
1.1 Project Overview	7
1.2 Motivation	8
1.3 Document Structure	8
2 System Architecture	9
2.1 Technology Stack	9
2.2 High-Level Architecture	9
2.3 Communication Patterns	10
2.3.1 REST over HTTP	10
2.3.2 Token Authentication	11
2.3.3 Password Reset Flow	11
2.3.4 Brokered Media Upload	11
2.4 CORS and Rate Limiting	12
3 Data Design	13
3.1 Database Overview	13
3.2 Domain Model	13
3.3 Entity–Relationship Overview	13
3.4 Key Design Decisions	13
4 Backend Implementation	17
4.1 API Design Principles	17
4.2 API Endpoint Reference	17
4.3 Service Layer	18
4.3.1 RecipeFeedService (recipes/services/feed.py)	18
4.3.2 RecipeLookupService (recipes/services/lookup.py)	18
4.3.3 StorageService (recipes/services/storage.py)	19
4.3.4 MacrosService (meal_planning/services/macros.py)	19
4.3.5 DietTagsService (recipes/services/diet_tags.py)	20
4.4 Middleware Stack	20

5	Frontend Implementation	22
5.1	Application Structure	22
5.1.1	Routing	22
5.2	State Management Strategy	22
5.2.1	Context Provider Hierarchy	22
5.2.2	TanStack Query Configuration	22
5.3	API Integration Layer	23
5.4	Design System	24
5.4.1	Color Palette	24
5.4.2	Typography	25
5.4.3	UI Component Library	25
5.5	Performance Optimizations	26
6	Core Feature Implementations	27
6.1	Personalized Recipe Feed	27
6.1.1	Exclusion Phase	27
6.1.2	Scoring Phase	27
6.2	Seeded Recipe Lookup	28
6.3	Meal Planning and Macro Aggregation	28
6.3.1	Meal Plan Data Model	28
6.3.2	Macro Aggregation	29
6.4	Social Graph Design	29
6.5	Cookbook System	29
7	Infrastructure and Deployment	32
7.1	Development Environment	32
7.1.1	Containerization with Docker Compose	32
7.2	Production Environment	32
7.2.1	Hosting Topology on Render	33
7.2.2	Backend Service Runtime	33
7.2.3	Frontend Static Site	34
7.2.4	Database and Object Storage	35
7.2.5	Transactional Email	35
7.2.6	Security Hardening	35
7.3	Continuous Integration	37
7.4	Environment Configuration	37
8	Conclusion	39
8.1	Summary of Delivered Work	39

8.2	Technical Reflections	40
8.3	Future Work	40

List of Figures

2.1	NutriChoice three-tier system architecture. The frontend uploads media directly to Supabase Storage using a short-lived signed URL obtained from the backend.	9
3.1	Simplified entity–relationship diagram. Junction tables (<code>RecipeIngredient</code> , <code>CookbookRecipe</code> , <code>RecipeLike</code> , <code>TriedRecipe</code>) and social-edge tables (<code>UserFollow</code> , <code>UserBlock</code>) are collapsed to labeled associations for visual clarity.	15
6.1	Personalized recipe feed generation algorithm (<code>RecipeFeedService.build_feed</code>).	31

List of Tables

2.1	NutriChoice Technology Stack	10
3.1	Django Application Decomposition	14
4.1	Authentication Endpoints (<code>/api/auth/</code>)	17
4.2	Recipe and Interaction Endpoints (<code>/api/</code>)	18
4.3	Cookbook and Meal-Planning Endpoints (<code>/api/</code>)	19
4.4	Social, User, and Storage Endpoints (<code>/api/</code>)	20
5.1	Application Routes (<code>src/router/index.tsx</code>)	23
5.2	Context Provider Summary	24
5.3	API Layer Function Groups (<code>src/api.ts</code>)	25
7.1	Docker Compose Services	32
7.2	Render Services Declared in <code>render.yaml</code>	33
7.3	Environment Variable Reference (Development vs. Production) . . .	38

Chapter 1

Introduction

1.1 Project Overview

NutriChoice is an integrated nutrition and recipe management platform developed as the main deliverable for the Capstone II course. The project addresses a well-identified gap in the consumer health-technology landscape: existing tools focus either narrowly on calorie logging or broadly on recipe sharing, but very few meaningfully combine personalized discovery, structured meal planning, and nutritional awareness in a single product.

The delivered system provides users with the following.

- A **personalized recipe feed** ordered by three priority signals: social graph (followed creators), ingredient familiarity (based on recipes previously tried), and community popularity (aggregate likes), with creation date as a final tiebreaker.
- A **recipe lookup** interface with stable seeded-random ordering, full dietary filtering, and full-text search across name and description fields.
- A **weekly meal planner** that assigns recipes to named meal slots (breakfast, first snack, lunch, second snack, dinner) across a Sunday-anchored seven-day grid.
- A **macronutrient dashboard** that calculates calories, protein, carbohydrates, and fat intake against user-configured targets.
- A **social graph** supporting user follows, content blocking, and recipe interactions including likes and a personal “tried” recipe log.
- A **cookbook system** for organizing saved recipes into user-defined collections.
- Secure, account-based access with full profile management, cross-device dietary preference synchronization, and a fully brokered cloud media upload pipeline.

1.2 Motivation

The motivation behind NutriChoice is twofold. First, nutritional health is increasingly recognized as a behavioral challenge rather than a knowledge gap: users know what constitutes a balanced diet but the tools currently available create friction in the meal planning experience, and make the user have to give up most of the dishes and recipes they have been eating their entire life. Second, purely algorithmic recommendation systems in the food domain often ignore the user’s stated dietary preferences and the social isolation that often accompanies calorie-restricted diets.

NutriChoice addresses both issues by coupling recipe discovery deeply with the user’s personal context (dietary preferences, allergens, followed creators, prior cooking history) and by connecting planning directly to nutrient computation, so users receive immediate, quantitative feedback on the macro impact of every meal decision.

1.3 Document Structure

This report is structured as follows:

Chapter 2 describes the overall system architecture and technology choices.

Chapter 3 presents the relational data model and schema design.

Chapter 4 details the backend REST API design and service layer.

Chapter 5 covers the frontend component architecture and state management strategy.

Chapter 6 documents the core algorithmic implementations powering key features.

Chapter 7 describes the deployment infrastructure and CI/CD pipeline.

Chapter 8 concludes with a reflection on the delivered system and directions for future work.

Chapter 2

System Architecture

2.1 Technology Stack

NutriChoice uses a three-tier architecture: a browser-resident presentation layer, a stateless REST application tier, and a managed data tier. Table 2.1 summarizes the principal technologies at each tier together with their functional roles.

2.2 High-Level Architecture

Figure 2.1 illustrates the three-tier structure of NutriChoice. The React SPA runs entirely in the browser and communicates with the Django backend over HTTP/REST. The backend mediates all access to the data tier and external services. Media uploads are brokered via short-lived signed URLs, keeping the Supabase service-role credential strictly server-side.

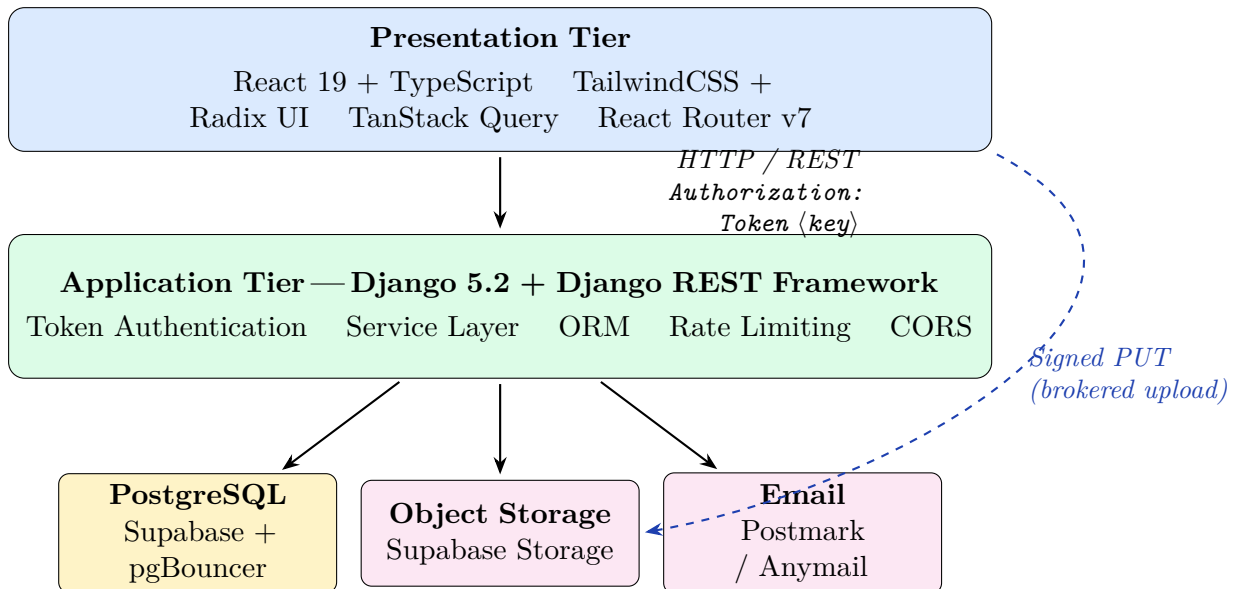


Figure 2.1: NutriChoice three-tier system architecture. The frontend uploads media directly to Supabase Storage using a short-lived signed URL obtained from the backend.

Table 2.1: NutriChoice Technology Stack

Tier	Technology	Role
Frontend	React 19 + TypeScript 5.8	Component framework and compile-time type safety
	Vite 7	Build tooling, hot-module replacement, manual chunk splitting
	TailwindCSS 3 + Radix UI	Design system primitives and accessible UI components
	TanStack Query v5	Server-state caching, infinite pagination, and background refetch
	React Router v7	Client-side routing with lazy loading and Suspense boundaries
Backend	Django 5.2	Web framework, ORM, signal infrastructure
	Django REST Framework	Serialization, viewsets, pagination, and token authentication
	django-anymail + Postmark	Transactional email delivery (password reset)
Data	PostgreSQL (Supabase)	Primary relational database
	pgBouncer (transaction mode)	Connection pooling for concurrency safety
	Supabase Storage	Object store for recipe images and profile pictures
Infra	Docker + Docker Compose	Container definition and local orchestration
	GitHub Actions + CodeQL	Continuous integration and automated security analysis

2.3 Communication Patterns

2.3.1 REST over HTTP

All client-server communication follows REST conventions using standard HTTP verbs (GET, POST, PUT, PATCH, DELETE). Response bodies are JSON-encoded. List endpoints use DRF’s page-number pagination with a default page size of 20 and a configurable maximum of 50. Explicit path declarations precede router-generated paths in `urls.py` to prevent URL pattern ambiguity (e.g., `/api/recipes/create/` is never captured by the `RecipeViewSet` detail pattern).

2.3.2 Token Authentication

NutriChoice uses DRF’s built-in `TokenAuthentication` scheme. A successful login or registration response returns a token; the client persists it in `localStorage`. Every subsequent authenticated request includes the header:

Authorization: Token <key>

Token rotation is supported: a refresh request atomically deletes the current token and issues a replacement, invalidating any subsequent request using the previous token. On a successful password reset, all existing tokens for the user are deleted, closing any active sessions.

2.3.3 Password Reset Flow

The password reset mechanism uses Django’s `default_token_generator`, which produces an HMAC-SHA256 token keyed on the user’s password hash and `last_login` timestamp. The token is embedded in a frontend URL alongside the user’s `public_id` (UUID v4) and delivered via Postmark. Because the token is bound to the password hash, it automatically invalidates once the password is changed. On successful reset confirmation, all DRF tokens for the user are deleted, enforcing a global session logout.

2.3.4 Brokered Media Upload

Images follow a four-step brokered upload pipeline designed to keep the Supabase service-role key strictly server-side:

1. **Client-side conversion:** The browser converts the raw image to WebP format via a `canvas` API call prior to transmission, standardizing the format before upload.
2. **Signed URL acquisition:** A POST to `/api/storage/signed-url/` returns a 120-second Supabase signed upload URL.
3. **Direct upload:** The client PUTs the WebP payload directly to Supabase Storage using the signed URL.
4. **URL persistence:** A POST to `/api/storage/save-url/` writes the resulting public CDN URL to the relevant model field (`Recipe.image_1`, `image_2`, `image_3`, or `UserProfile.profile_picture`).

2.4 CORS and Rate Limiting

The backend enforces strict CORS policy via `django-cors-headers`, whitelisting only the known frontend development origins (`http://localhost:3000` and `http://localhost:5173`); production origins are injected via environment variable. All API endpoints require authentication via a global `IsAuthenticated` permission class, with explicit `AllowAny` overrides only on the registration, login, and password-reset views. DRF throttling is configured globally at 30 requests/minute for anonymous access and 300 requests/minute for authenticated users. The password-reset request endpoint carries an additional `ScopedRateThrottle` of 5 requests/hour to mitigate abuse.

Chapter 3

Data Design

3.1 Database Overview

NutriChoice uses a PostgreSQL database hosted on Supabase’s managed cloud. Connection pooling is handled by pgBouncer running in transaction mode. This configuration requires two adjustments in Django’s database settings: `CONN_MAX_AGE=0` (no persistent connections, since pgBouncer multiplexes them) and `DISABLE_SERVER_SIDE_CURSORS=True` (server-side cursors are incompatible with transaction-mode pooling). Binary objects, recipe images and profile pictures are stored in Supabase Storage and referenced in the relational database only by URL, keeping the schema free of binary blobs and enabling direct CDN delivery.

3.2 Domain Model

The backend is decomposed into seven domain Django applications plus a thin `api` aggregator app that owns URL routing and view definitions. Table 3.1 summarizes the seven domain apps.

3.3 Entity–Relationship Overview

Figure 3.1 presents a simplified entity–relationship diagram covering the core entities and their primary associations. Junction tables responsible for many-to-many relationships (e.g., `RecipeIngredient`, `CookbookRecipe`, `RecipeLike`, `TriedRecipe`) are present in the schema but omitted from the diagram for visual clarity.

3.4 Key Design Decisions

Public UUIDs for external identifiers. The `User` model exposes a `public_id` (UUID v4, auto-generated and immutable) on all external-facing API paths.

Table 3.1: Django Application Decomposition

App	Primary Models	Responsibility
users	User	Custom <code>AbstractUser</code> extending Django’s auth model; adds a UUID <code>public_id</code> for all external API paths, preventing sequential integer enumeration.
profiles	UserProfile	Nutritional goals (calories, protein, carbs, fat), dietary type flags, allergy list, bio, profile picture URL, and onboarding completion flag (boolean); one-to-one relationship with <code>User</code> .
recipes	Recipe, RecipeIngredient, RecipeInstruction, Cookbook, CookbookRecipe	Recipe definitions (name, description, cuisine type, dietary tags, images), ingredient join table, ordered step instructions, and user-created cookbook collections.
ingredients	Ingredient	Read-only nutrition reference database (calories, protein, carbs, fat, fiber, sugar, sodium per 100 g). <code>save()</code> and <code>delete()</code> are overridden to raise <code>ValidationError</code> , treating the catalog as an immutable seed dataset.
nutrition	RecipeNutrition	Pre-computed macro totals per recipe (one-to-one with <code>Recipe</code>); avoids per-request aggregation joins.
meal_planning	MealPlanEntry	User meal-slot assignments (date + slot + recipe); a composite unique constraint on (<code>user_id</code> , <code>date</code> , <code>meal_slot</code>) enforces upsert semantics.
social	UserFollow, UserBlock, RecipeLike, TriedRecipe, RecipeReview	Directed social-graph edges (follow, block), recipe interaction records (like, tried), and user-authored recipe ratings.

Internal joins still use integer primary keys for performance. This prevents sequential integer enumeration of user accounts via public-facing endpoints.

Pre-computed nutritional totals. Rather than aggregating macro totals on every request by joining `RecipeIngredient` to `Ingredient`, each recipe is assigned a `RecipeNutrition` record that stores pre-computed values. Meal-plan macro aggregation then requires only a single `SUM` over `RecipeNutrition`, regardless of how many ingredients a recipe contains.

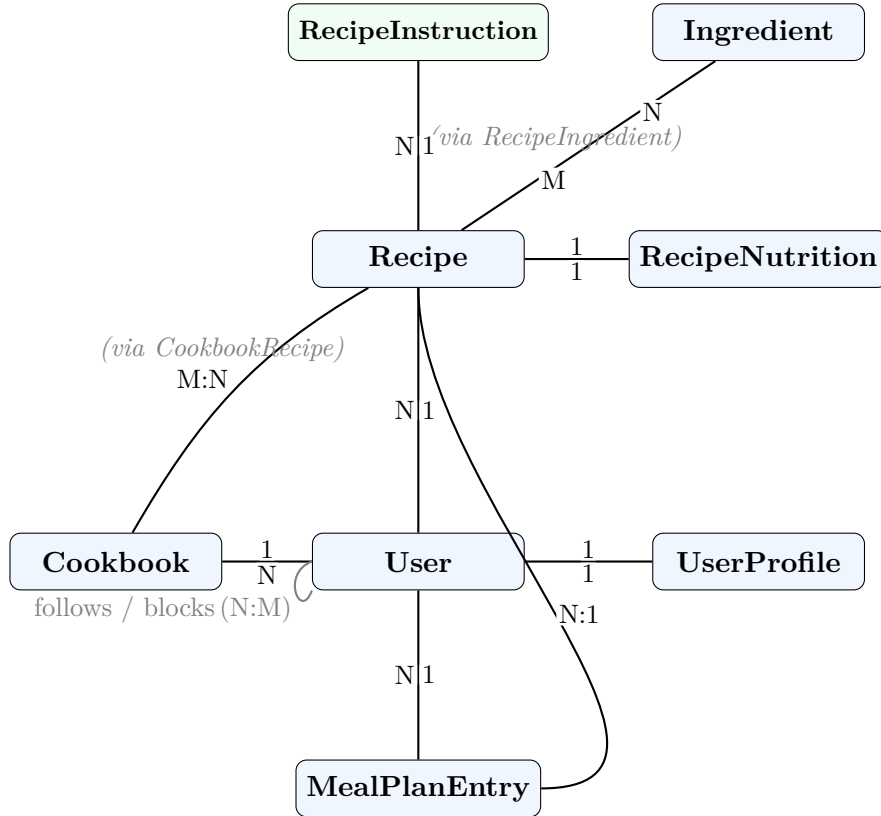


Figure 3.1: Simplified entity-relationship diagram. Junction tables (*RecipeIngredient*, *CookbookRecipe*, *RecipeLike*, *TriedRecipe*) and social-edge tables (*UserFollow*, *UserBlock*) are collapsed to labeled associations for visual clarity.

Composite unique constraints. Every junction and interaction table carries a database-level unique constraint on its foreign-key pair (e.g., `UNIQUE(user_id, recipe_id)` on *RecipeLike*). This ensures idempotent writes under concurrent requests.

Self-referential check constraints. Both *UserFollow* and *UserBlock* carry `CheckConstraints` at the database level preventing self-follows and self-blocks (`follower_id ≠ followee_id`).

Cascading deletes on *TriedRecipe*. The `recipe` foreign key on *TriedRecipe* is declared nullable (`null=True`) but uses `on_delete=CASCADE`, so when a recipe is deleted its associated tried-recipe entries are removed alongside it. The nullable column is retained to leave room for a future migration to soft-deletion semantics, where the cooking-history record could outlive its source recipe and gracefully degrade the ingredient-familiarity feed signal.

Immutable ingredient catalog. `Ingredient.save()` and `Ingredient.delete()` are overridden to raise `ValidationError`, treating the catalog as a seeded read-only reference dataset.

Chapter 4

Backend Implementation

4.1 API Design Principles

The REST API follows resource-oriented URL design with explicit action paths for non-CRUD operations (e.g., `/api/cookbooks/{id}/add-recipe/`). All routes are registered in `backend/api/urls.py` under a single `/api/` prefix. Authenticated endpoints return HTTP 403 for unauthenticated callers and HTTP 404 for resources not owned by the requesting user, preventing indirect object reference leaks.

4.2 API Endpoint Reference

Tables 4.1–4.4 list all API endpoints grouped by domain.

Table 4.1: Authentication Endpoints (`/api/auth/`)

Method	Path	Auth	Description
POST	<code>register/</code>	No	Create account; atomically provisions <code>UserProfile</code>
POST	<code>login/</code>	No	Validate credentials; return token + user snapshot
POST	<code>logout/</code>	Yes	Delete auth token
GET	<code>me/</code>	Yes	Return current user with profile snapshot
POST	<code>complete-onboarding/</code>	Yes	Persist dietary preferences; set <code>is_onboarded=True</code>
POST	<code>token-refresh/</code>	Yes	Rotate token: delete old, issue new
POST	<code>password-change/</code>	Yes	Change password; delete all tokens
POST	<code>password-reset-request/</code>	No	Send HMAC-signed reset link by email; 5 requests/hour limit
POST	<code>password-reset-confirm/</code>	No	Verify token; set new password; delete all tokens

Table 4.2: Recipe and Interaction Endpoints (/api/)

Method	Path	Auth	Description
GET	recipes/	Yes	Paginated list; search , ordering , ingredient , scope , creator filters
GET	recipes/{id}/	Yes	Full detail with ingredients and instructions
POST	recipes/create/	Yes	Create recipe; creator set to requesting user
GET	recipes/{id}/nutrition/	Yes	Pre-computed macro totals and per-serving values
GET	recipe-feed/	Yes	Scored personalized feed (paginated)
GET	recipe-lookup/	Yes	Seeded-random browse with dietary filters
GET	recipe-likes/	Yes	Current user’s liked recipes
POST	recipe-likes/	Yes	Like a recipe
DELETE	recipe-likes/{id}/	Yes	Unlike a recipe
GET	tried-recipes/	Yes	Current user’s tried recipes
POST	tried-recipes/	Yes	Mark recipe as tried
DELETE	tried-recipes/{id}/	Yes	Remove tried-recipe entry
GET	tried-recipes/most_tried/	Yes	Top 10 most-tried recipes globally

4.3 Service Layer

Non-CRUD business logic is encapsulated in a per-app **services/** package, keeping views thin and domain logic independently testable.

4.3.1 RecipeFeedService (recipes/services/feed.py)

Constructs the personalized recipe feed queryset through an exclusion-then-scoring pipeline. See Section 6.1.

4.3.2 RecipeLookupService (recipes/services/lookup.py)

Builds a deterministic seeded-random recipe queryset using a database-level `MD5(CAST(id AS TEXT) || seed)` expression for stable, pagination-safe ordering. See Section 6.2.

Table 4.3: Cookbook and Meal-Planning Endpoints (/api/)

Method	Path	Auth	Description
GET	cookbooks/	Yes	List user's cookbooks
POST	cookbooks/	Yes	Create cookbook
GET	cookbooks/{id}/	Yes	Retrieve with nested recipe cards
PUT/PATCH	cookbooks/{id}/	Yes	Update name or description
DELETE	cookbooks/{id}/	Yes	Delete cookbook
POST	cookbooks/{id}/add-recipe/	Yes	Add recipe to cookbook
DELETE	cookbooks/{id}/remove-recipe/	Yes	Remove recipe from cookbook
GET	meal-plan/week/	Yes	Seven-day grid (?week_start=YYYY-MM-DD)
GET	meal-plan/macros/	Yes	Daily macro totals vs. targets (?date=YYYY-MM-DD)
POST	meal-plan/entry/	Yes	Create/replace meal-slot entry (upsert)
DELETE	meal-plan/entry/{id}/	Yes	Remove entry

4.3.3 StorageService (recipes/services/storage.py)

Supabase client wrapper (cached via `lru_cache`). Exposes `generate_signed_upload_url()`, `upload_file()`, `delete_file()`, `get_public_url()`, and `delete_recipe_images()`. The `delete_recipe_images()` hook is registered as a Django `post_delete` signal receiver on `Recipe` to delete associated Supabase Storage objects on recipe deletion.

4.3.4 MacroService (meal_planning/services/macros.py)

`compute_daily_macros(user, date)` executes a single SQL `SUM` aggregation joining `MealPlanEntry` → `Recipe` → `RecipeNutrition` and dividing by serving count to return per-serving macro totals. `get_user_targets(user)` retrieves the profile's configured goals, falling back to defaults (2000 kcal, 120 g protein, 250 g carbohydrate, 65 g fat) for users without configured targets.

Table 4.4: Social, User, and Storage Endpoints (/api/)

Method	Path	Auth	Description
GET	follows/	Yes	List users the current user follows
POST	follows/	Yes	Follow a user
DELETE	follows/{id}/	Yes	Unfollow
GET	blocks/	Yes	List blocked users
POST	blocks/	Yes	Block a user
DELETE	blocks/{id}/	Yes	Unblock
GET	users/	Yes	List users (optional ?username= filter)
GET	users/{id}/	Yes	Get user by public UUID
GET	ingredients/	Yes	List or search ingredient catalog
GET	health/	No	Health check ({ <code>"status": "ok"</code> })
POST	storage/signed-url/	Yes	Request signed Supabase upload URL
POST	storage/save-url/	Yes	Persist uploaded media URL to model

4.3.5 DietTagsService (recipes/services/diet_tags.py)

Maintains a canonical `ALLOWED_DIET_KEYS` frozenset of eight diet types (vegetarian, vegan, gluten_free, dairy_free, nut_free, keto, paleo, low_carb) and a `normalize_dietary_tags()` function that maps variant aliases to canonical keys before storage (e.g., `"gluten-free"` → `gluten_free`).

4.4 Middleware Stack

The Django middleware stack is ordered to ensure correct behavior for cross-origin requests and response compression:

1. **CorsMiddleware** (outermost) — intercepts preflight `OPTIONS` requests before authentication middleware runs.
2. **GZipMiddleware** — compresses all responses exceeding 200 bytes.
3. Django security, **WhiteNoise** (static-file serving), session, common, conditional-get, CSRF, authentication, messages, and clickjacking middle-

wares.

Chapter 5

Frontend Implementation

5.1 Application Structure

The frontend is a React 19 TypeScript single-page application built with Vite. All route components are code-split via `React.lazy()` with `Suspense` boundaries and a custom animated `PageLoader` component. The router tree distinguishes public routes (authentication, password flows) from protected routes wrapped inside an authenticated `Layout` component.

5.1.1 Routing

Table 5.1 lists all application routes and their corresponding page components.

5.2 State Management Strategy

NutriChoice uses a two-layer state model: **TanStack Query** for server state (remote data, caching, pagination) and **React Context providers** for shared application state that must persist across navigation events.

5.2.1 Context Provider Hierarchy

Providers are split into two mounting levels. Root-level providers are mounted unconditionally (including on unauthenticated routes); layout-level providers are mounted inside `Layout` and only activate after successful authentication, preventing unnecessary API calls on public pages.

5.2.2 TanStack Query Configuration

The shared `QueryClient` is configured with:

- `staleTime: 300 000 ms`—prevents redundant background refetches during an active session.

Table 5.1: Application Routes (`src/router/index.tsx`)

Path	Component	Description
/	AuthPage	Login / registration toggle (public)
/reset-password	PasswordResetRequestPage	Email entry for reset link
/reset-password/confirm	PasswordResetConfirmPage	Token + new password form
/change-password	PasswordChangePage	Authenticated password update
/home	HomePage	Seeded-random browse with dietary filter override
/recipe-feed	RecipeFeedPage	Personalized scored feed (infinite scroll)
/favorites	FavoritesPage	Liked recipes grid
/my-recipes	FavoritesPage	Creator-scoped recipe grid
/cookbooks	CookbooksPage	User cookbook collection
/cookbooks/:id	CookbookViewPage	Cookbook detail with recipe cards
/meal-planning	MealPlanningPage	Seven-day meal grid + macro cards
/nutrition	NutritionPage	Daily macro overview and weekly trends
/account	AccountPage	Profile, security, and dietary settings

- `gcTime`: 600 000 ms—garbage-collects cached queries unused for more than 10 minutes.
- `retry`: 1—one automatic retry on transient network errors.

Query keys are parameterised (e.g., `['recipe-lookup', { search, seed, dietOverride }]`) so that each unique filter combination occupies an independent cache slot, enabling instant cache hits when a user reverts to a previously used filter state.

5.3 API Integration Layer

All HTTP communication is centralised in `src/api.ts`. Authenticated requests flow through a shared `authenticatedFetch()` helper, which reads the stored

Table 5.2: Context Provider Summary

Provider	State Managed	Key Behavior
ThemeProvider	Light/dark mode	Persisted to <code>localStorage</code> ; falls back to <code>prefers-color-scheme</code>
UserPreferencesProvider	Dietary filter object	Seeded from <code>localStorage</code> on mount; then backend hydration overrides for multi-device sync
RecipeActionsProvider	Liked + tried sets	Optimistic toggles with rollback on API failure
SocialActionsProvider	Following + blocked lists	Optimistic updates; edge-ID maps for $O(1)$ deletion
CookbooksProvider	Cookbook list + detail	Full list on mount; per-cookbook detail lazy-loaded
MealPlanningProvider	Weekly meal grid	Full week fetched on <code>loadWeek(startDate)</code> call

token from `localStorage` and injects the `Authorization: Token <key>` header along with `Content-Type: application/json`. Error payloads from DRF are normalized by `formatApiError()`, which extracts messages from the `password`, `username`, `email`, `detail`, and `non_field_errors` fields.

Table 5.3 groups the exported API functions by domain.

5.4 Design System

The visual design system is implemented as CSS custom properties (using HSL color values, defined in `src/index.css`) consumed via Tailwind utility classes. Dark mode is driven by toggling the `.dark` class on `<html>`, enabling full palette switching via CSS alone.

5.4.1 Color Palette

The primary accent is a vivid green (`hsl(142.1 76.2% 36.3%)` light / `hsl(142.1 70.6% 45.3%)` dark). Destructive actions use red (`hsl(0 84.2% 60.2%)`).

Table 5.3: API Layer Function Groups (src/api.ts)

Group	Key Functions
Authentication	register, login, logout, getCurrentUser, completeOnboarding, refreshToken, requestPasswordReset, changePassword, confirmPasswordReset
User / Profile	updateUser, updateUserProfile, getMyProfile
Recipes	getRecipes, getRecipe, createRecipe, apiRecipeToRecipe (shape converter)
Likes & Tried	getRecipeLikes, likeRecipe, unlikeRecipe, getTriedRecipes, markRecipeTried, unmarkRecipeTried
Cookbooks	getCookbooks, getCookbookDetail, createCookbook, updateCookbook, deleteCookbook, addRecipeToCookbook, removeRecipeFromCookbook
Feed & Lookup	getRecipeFeed, getRecipeLookup
Meal Planning	fetchWeekPlan, fetchDailyMacros, createMealPlanEntry, deleteMealPlanEntry
Social	getFollowing, createFollow, deleteFollow, getBlocks, createBlock, deleteBlock
Storage	requestSignedUrl, saveImageUrl
Ingredients	getFoods, searchIngredients

5.4.2 Typography

The application relies on Tailwind’s default system font stack (`system-ui, -apple-system, BlinkMacSystemFont, "Segoe UI", Roboto`) to eliminate web-font load latency. The type scale uses: `text-xs` (12px) for badges and metadata labels; `text-sm` (14px) for body text, inputs, and buttons; `text-lg` (18px) for dialog titles; `text-2xl/text-3xl` for page headings.

5.4.3 UI Component Library

UI primitives are `shadcn/ui` wrappers over Radix UI, providing accessible keyboard-navigable implementations of: `Dialog`, `Popover`, `Accordion`, `Tabs`, `Select`, `Checkbox`, `Slider`, `Switch`, and `Progress`. Component variants are declared with `class-variance-authority`; class merging uses `clsx` and `tailwind-merge` to prevent conflicts.

5.5 Performance Optimizations

Code splitting. Vite's `manualChunks` configuration partitions the bundle into four chunks grouped by update frequency: `vendor` (React/DOM), `router` (React Router), `query` (TanStack Query), and `radix` (all Radix UI primitives). Stable chunks maximize CDN and browser cache hit rates across deployments.

Debounced search. Recipe search inputs debounce by 300 ms before triggering a new query, reducing unnecessary backend round trips.

Optimistic updates. Like, tried, follow, and block interactions update local state immediately and issue the API call concurrently in the background. On failure, the optimistic state is reverted. This pattern targets sub-50 ms perceived response times on interactive elements.

Multi-device preference sync. `UserPreferencesProvider` seeds from `localStorage` on mount for instant display, then overwrites with the backend-stored value on first successful hydration. Writes update `localStorage` immediately and dispatch a backend PATCH concurrently.

Infinite scroll. The home and feed pages implement infinite scroll via the `IntersectionObserver` API watching a sentinel element with `rootMargin: "300px"` and `threshold: 0.1`, connected to TanStack Query's `useInfiniteQuery` via the `fetchNextPage` callback. Recipe IDs are deduplicated across page boundaries to prevent duplicate cards appearing during rapid page fetches.

Chapter 6

Core Feature Implementations

6.1 Personalized Recipe Feed

The personalized feed is implemented in `RecipeFeedService.build_feed()` (`recipes/services/feed.py`) and follows a two-phase pipeline: exclusion filtering followed by multi-axis score annotation.

6.1.1 Exclusion Phase

Before scoring, candidates are reduced by eliminating:

- Recipes created by users the requesting user has **blocked** (one-directional enforcement: a recipe is hidden from the blocker's feed; reverse-direction filtering is identified as future work).
- Recipes the user has already **marked as tried**.
- Recipes whose **dietary tags** conflict with the user's active diet preferences (AND logic: all selected diet types must be satisfied).
- Recipes that **contain allergen ingredients** present in the user's allergy list (OR logic; case-insensitive ingredient name matching).

6.1.2 Scoring Phase

Each surviving recipe is annotated with three signals computed in a single SQL pass:

1. **is_from_followed (0 or 1)**: A `Case/When` expression that returns 1 if the recipe's creator is in the requesting user's follow list, else 0.
2. **ingredient_overlap**: A `Count` of recipe ingredients that also appear among the ingredients of recipes the user has previously tried, computed via a correlated subquery over `RecipeIngredient`.

3. **like_count**: A Count of all `RecipeLike` rows referencing the recipe (raw global popularity).

The annotated queryset is then ordered *lexicographically* by these three signals as a priority tiebreaker chain, with `date_created` and `id` as final stable tiebreakers: `ORDER BY -is_from_followed, -ingredient_overlap, -like_count, -date_created, -id`. Note that the algorithm currently does not score recency; `date_created` appears only as a tiebreaker. Figure 6.1 shows the full algorithm.

6.2 Seeded Recipe Lookup

The recipe lookup endpoint (`RecipeLookupService.build_lookup()`) provides a browseable recipe catalog with **stable pagination**. Database queries without an explicit `ORDER BY` return rows in arbitrary order; a user paginating through results would see duplicates or skipped recipes as the query planner reorders rows between page requests.

NutriChoice solves this using a per-session UUID seed. The queryset ordering is computed as `MD5(CAST(id AS TEXT) || seed_value)` at the database level. Because the hash is deterministic, the same seed yields the same ordering for every page request. A new seed is issued on reload, avoiding server-side session state.

Dietary and allergen filters from the personalized feed’s exclusion phase are applied identically in the lookup, ensuring browse results always respect the user’s safety constraints.

6.3 Meal Planning and Macro Aggregation

6.3.1 Meal Plan Data Model

Each `MealPlanEntry` record associates a user, a calendar date, a named meal slot (one of: `breakfast`, `snack1`, `lunch`, `snack2`, `dinner`), and a recipe. The composite unique constraint `UNIQUE(user_id, date, meal_slot)` implements **upsert semantics**: assigning a recipe to an occupied slot replaces the previous entry at the database level, eliminating a client-side delete-then-create cycle.

The frontend maps meal slots to visual rows in a 5×7 grid (five meal types across seven days), with week navigation anchored to Sundays (hard-coded).

6.3.2 Macro Aggregation

`compute_daily_macros(user, date)` (`meal_planning/services/macros.py`) computes nutritional totals for a given day in a single database round trip:

1. Join `MealPlanEntry` $\rightarrow$ `Recipe` $\rightarrow$ `RecipeNutrition` for the target user and date.
2. Apply `SUM(calories / servings)`, `SUM(protein / servings)`, `SUM(fat / servings)`, and `SUM(carbohydrates / servings)` for each macro field.
3. Compare the resulting totals against the user's `UserProfile` targets (with default fallbacks).

On the frontend, `aggregateMacrosForDate()` and `aggregateMacrosForWeek()` are utility functions that compute aggregates client-side from the already-loaded weekly payload, avoiding additional API calls.

6.4 Social Graph Design

The social graph comprises two directed edge tables:

- **UserFollow (follower, followee)**: a directed subscription edge. Database `CheckConstraint` prevents self-follows.
- **UserBlock (blocker, blocked)**: a content suppression edge. Blocks are enforced one-directionally in the feed algorithm: a recipe by user B is excluded from user A's feed when user A has blocked user B. The reverse direction (B's block of A hiding A's recipes from B) is left to future work.

Both tables carry unique constraints on their FK pairs; the constraint raises an `IntegrityError` on concurrent duplicate inserts. The frontend's `SocialActionsProvider` maintains in-memory sets of following and blocked user IDs with edge-ID maps for $O(1)$ deletion without an extra round trip.

6.5 Cookbook System

Cookbooks are named, user-owned collections backed by the `Cookbook` and `CookbookRecipe` join table. A unique constraint on `(owner, name)` prevents duplicate cookbook names. The frontend's `CookbooksProvider` fetches the full cookbook list on mount and lazy-loads per-cookbook recipe detail only when

the user navigates to a specific cookbook, avoiding pre-loading recipes for all cookbooks.

Recipe add/remove operations are optimistic: the UI updates immediately while the API request is in flight, reverting on failure.

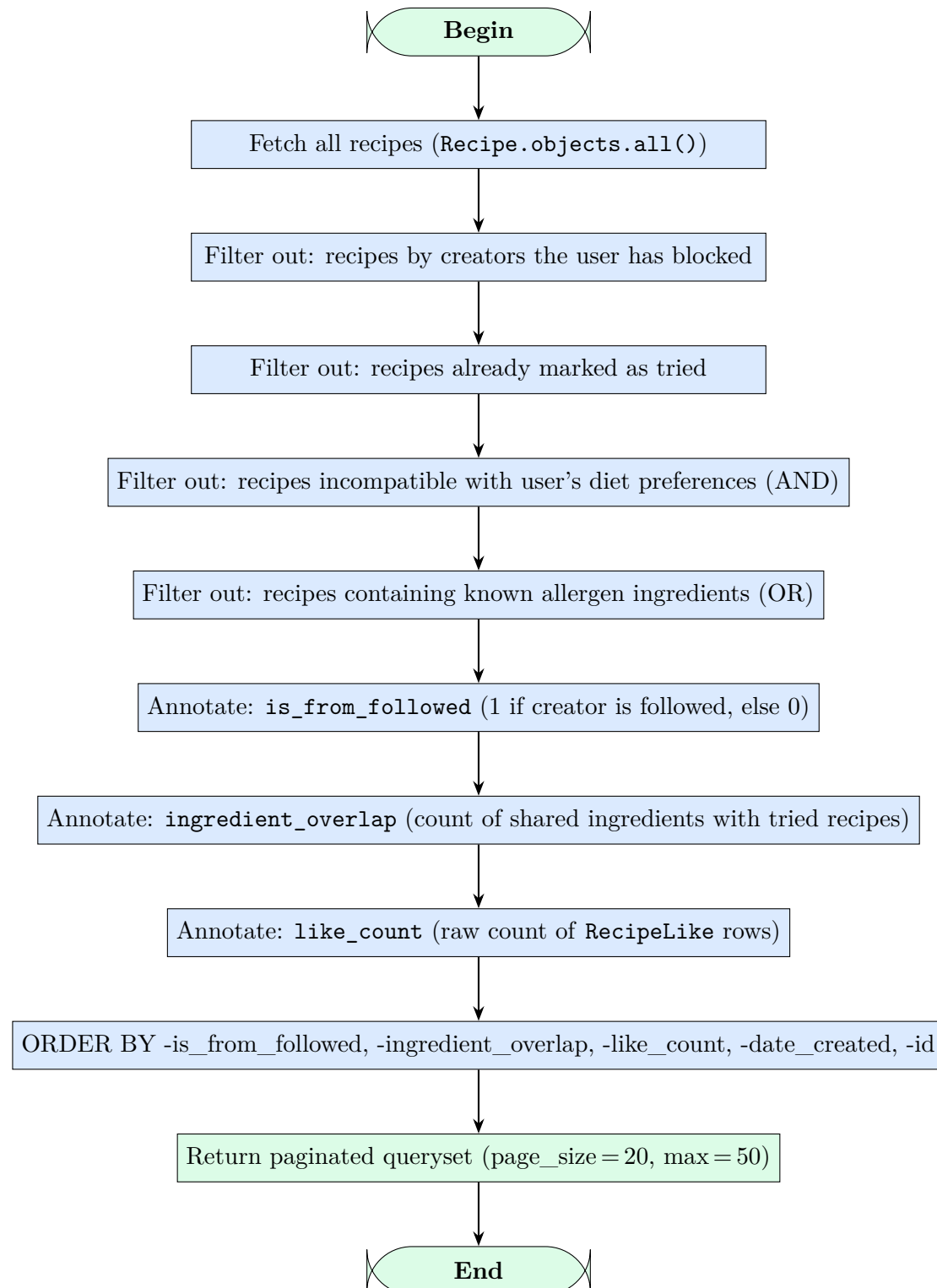


Figure 6.1: Personalized recipe feed generation algorithm (`RecipeFeedService.build_feed`).

Chapter 7

Infrastructure and Deployment

7.1 Development Environment

7.1.1 Containerization with Docker Compose

The development environment is containerized via Docker Compose, providing a reproducible one-command startup for the entire stack (`docker-compose up`). The compose stack is used exclusively for local development; production runs on a separate hosting platform documented in Section 7.2.

Table 7.1: Docker Compose Services

Service	Base Image	Port	Configuration
backend	python:3.11-slim	8000	Mounts <code>./backend:/app</code> ; loads <code>./backend/.env</code> ; restarts unless stopped
frontend	node:22-alpine	3000	Mounts <code>./frontend:/app</code> ; <code>node_modules</code> in a named volume to prevent host OS conflicts; the frontend Dockerfile invokes <code>vite -host 0.0.0.0</code> for container-network reachability; <code>CHOKIDAR_USEPOLLING=true</code> for file-watch compatibility inside Docker

Both services share a custom bridge network (`nutrichoice_network`). The frontend service declares a `depends_on` constraint on the backend to order container startup. Volume mounts enable hot-module replacement on both tiers without image rebuilds.

7.2 Production Environment

The production deployment runs on Render.com, a managed platform-as-a-service provider, with the database and object storage hosted on Supabase. The full deployment topology is declarative: a single `render.yaml` blueprint at the repository root

defines every service, build command, runtime, and environment variable needed to bring the system up. Pushes to the `main` branch trigger an automatic build and deploy on each affected service, removing the need for an external orchestration layer.

7.2.1 Hosting Topology on Render

The blueprint declares two web services that together form the public surface of the application:

Table 7.2: Render Services Declared in `render.yaml`

Service	Runtime	Plan / Region	Public Domain
<code>nutrichoice-backend</code>	Python 3.12.3	Free, Virginia	<code>nutrichoice-backend.onrender.com</code>
<code>nutrichoice-frontend</code>	Static	Free, Virginia	<code>nutrichoice-frontend.onrender.com</code>

Both services track the `main` branch and redeploy automatically on every push. The backend declares a health-check path of `/api/health/`. Render polls this endpoint to confirm site is live and to gate traffic during rolling deploys. Render also provisions TLS certificates for both domains and terminates HTTPS at its edge.

7.2.2 Backend Service Runtime

The backend service is built from the `backend/` subdirectory of the repository. The build command installs Python dependencies, collects static assets, and applies any pending database migrations in a single step:

```
1 pip install -r requirements.txt && \
2 python manage.py collectstatic --noinput && \
3 python manage.py migrate --noinput
```

Listing 7.1: Backend build command in `render.yaml`

The Python interpreter version is pinned twice: through the `PYTHON_VERSION` environment variable in the blueprint and through a `runtime.txt` file in the backend directory, both set to `3.12.3`. This redundancy guards against a single source of drift if either file is edited in isolation.

The runtime is served by Gunicorn, configured for the constraints of Render’s free plan:

```
1 gunicorn server.wsgi:application \  
2   --workers 2 --threads 4 --worker-class gthread \  
3   --timeout 60 --bind 0.0.0.0:$PORT \  
4   --access-logfile - --error-logfile -
```

Listing 7.2: Gunicorn start command

Two worker processes were chosen to fit within the memory budget of the free tier, while four threads per worker via the `gthread` worker class accommodate the I/O-bound nature of typical Django request handling (database queries, Supabase Storage calls, and outbound Postmark requests). The 60-second request timeout is conservative for a REST API but is necessary due to the cold-start latency that the free plan incurs after periods of inactivity.

7.2.3 Frontend Static Site

The frontend service is built from the `frontend/` subdirectory using `npm ci && npm run build`, which produces a fully static bundle in `./dist`. Render serves this directory directly from its global edge, with no Node process running in production.

Because the frontend is a single-page application driven by React Router, the blueprint installs a rewrite rule that resolves every unmatched path to the application shell:

```
1 routes:  
2   - type: rewrite  
3     source: /*  
4     destination: /index.html
```

Listing 7.3: SPA rewrite rule in `render.yaml`

Without this rule a hard refresh on any client-side route (for example, `/recipes/42`) would return a 404 from the static server. The rewrite preserves the URL bar while delegating route resolution to React Router on the client.

The build embeds `VITE_API_URL` (set in the blueprint to the production backend domain) at compile time. The frontend bundle therefore contains no runtime configuration lookup; switching the backend target requires a rebuild.

7.2.4 Database and Object Storage

The production database is a Supabase-managed PostgreSQL 17 instance reached through Supabase’s pgBouncer connection pooler in transaction mode on port 6543. Two non-negotiable Django settings are required for the application to operate correctly against a transaction-mode pooler:

- `CONN_MAX_AGE = 0`: connection reuse is delegated to the pooler. Persistent application-side connections would double-bind a pooled session and exhaust the pool under load.
- `DISABLE_SERVER_SIDE_CURSORS = True`: transaction-mode pgBouncer does not preserve server-side cursors across transaction boundaries, so Django must materialize result sets in the application process.

All connections enforce `sslmode=require`. The database hostname, username, password, and port are supplied as Render dashboard secrets and never appear in version control or in the `render.yaml` file.

User-uploaded recipe imagery is stored in Supabase Storage rather than on the application server’s local disk, which is ephemeral on Render. The backend accesses storage through the Supabase service-role key (held as a dashboard secret) and surfaces images to clients as time-limited signed URLs with a 120-second expiry. Short-lived URLs prevent indefinite hotlinking of private assets while keeping signing overhead negligible relative to image transfer time.

7.2.5 Transactional Email

Outbound email (currently used for the password-reset flow) is delivered through Postmark via the `django-anymail` package with the `[postmark]` extra. The Django email backend points to the `EmailBackend` class in `anymail.backends.postmark`, and the `POSTMARK_SERVER_TOKEN` is held as a dashboard secret. The `FRONTEND_URL` environment variable is embedded in reset links so users land on the production frontend regardless of where the email is opened. The `DEFAULT_FROM_EMAIL` address must correspond to a verified Postmark sender; unverified senders are rejected by the Postmark API at send time.

7.2.6 Security Hardening

Production-only security middleware activates conditionally inside the Django settings module (`backend/server/settings.py`) when `DEBUG` is false:

```
1 if not DEBUG:
2     SECURE_PROXY_SSL_HEADER = ("HTTP_X_FORWARDED_PROTO", "
    https")
3     SESSION_COOKIE_SECURE = True
4     CSRF_COOKIE_SECURE = True
5     SECURE_HSTS_SECONDS = 31536000
6     SECURE_HSTS_INCLUDE_SUBDOMAINS = True
7     SECURE_HSTS_PRELOAD = True
8     SECURE_CONTENT_TYPE_NOSNIFF = True
```

Listing 7.4: Production-only security settings in `settings.py`

The `SECURE_PROXY_SSL_HEADER` setting tells Django to trust the X-Forwarded-Proto header that Render’s edge proxy injects, so the application correctly identifies requests as HTTPS even though the WSGI server itself is reached over plain HTTP from inside the platform network. Session and CSRF cookies are restricted to secure transport. HSTS is enabled with a one-year max-age, subdomain coverage, and preload eligibility, instructing browsers to refuse plain HTTP for the domain and any subdomain. `SECURE_CONTENT_TYPE_NOSNIFF` blocks browser MIME-sniffing of responses.

A separate block at the top of the same file injects Render’s externally visible hostname into both the `ALLOWED_HOSTS` list and the CSRF trusted origins:

```
1 RENDER_EXTERNAL_HOSTNAME = os.getenv("RENDER_EXTERNAL_HOSTNAME
    ")
2 if RENDER_EXTERNAL_HOSTNAME:
3     if RENDER_EXTERNAL_HOSTNAME not in ALLOWED_HOSTS:
4         ALLOWED_HOSTS.append(RENDER_EXTERNAL_HOSTNAME)
5     render_origin = f"https://{RENDER_EXTERNAL_HOSTNAME}"
6     # ... appended to CSRF_TRUSTED_ORIGINS
```

Listing 7.5: Dynamic hostname injection for Render deployments

Render auto-injects the `RENDER_EXTERNAL_HOSTNAME` variable into every web service. Reading it at startup keeps the service resilient to drift between the blueprint and the dashboard: if the explicit `ALLOWED_HOSTS` value is misconfigured, omitted, or temporarily out of sync after a domain change, the dynamic append still admits the canonical Render hostname and prevents `DisallowedHost` responses. The block is a no-op outside Render because the variable is unset.

7.3 Continuous Integration

Continuous integration is handled by a GitHub Actions workflow (`.github/workflows/codeql.yml`) running static security analysis via GitHub CodeQL.

- **Triggers:** Every push or pull request targeting `main` or `dev`; plus a weekly scheduled scan (Mondays at 00:00 UTC) to catch newly disclosed vulnerability patterns against unchanged code.
- **Language matrix:** `javascript-typescript` (React + TypeScript frontend) and `python` (Django backend) are analyzed as independent matrix jobs within the same workflow run.
- **Permissions:** `security-events: write` (to upload SARIF results to the GitHub Security dashboard), `actions: read` (required by the CodeQL action), and `contents: read`. The workflow has no write access to the repository's source.

CodeQL performs dataflow-based static analysis to detect known vulnerability classes (injection, path traversal, prototype pollution, insecure deserialization, etc.) without executing the application. Findings are surfaced directly within pull-request review checks and the repository's Security tab.

Continuous deployment is delegated to Render rather than to a custom workflow. Each push to `main` triggers Render to rebuild and redeploy whichever services are affected, using the build and start commands declared in `render.yaml` as the deploy contract.

7.4 Environment Configuration

Configuration is externalized in both environments, but the source of values differs by tier. In development each tier reads a local `.env` file (excluded from version control via `.gitignore`); in production the same variables are supplied by Render. Non-sensitive values (such as `ALLOWED_HOSTS` or the frontend domain) are committed to `render.yaml` as the declarative source of truth, while every credential is held as a Render dashboard secret marked `sync: false` in the blueprint, which prevents the value from being printed back into source control. Table 7.3 lists every variable consumed by the system and contrasts its development and production sources.

The concrete public domains for the backend and frontend services are listed

Table 7.3: Environment Variable Reference (Development vs. Production)

Context	Variable	Development	Production
Backend	PYTHON_VERSION	Host Python (any 3.12.x)	3.12.3 pinned in <code>render.yaml</code>
	DJANGO_SECRET_KEY	Local placeholder in <code>.env</code>	Render dashboard secret
	DEBUG	True	False
	ALLOWED_HOSTS	<code>localhost,127.0.0.1</code>	Backend service domain plus auto-injected <code>RENDER_EXTERNAL_HOSTNAME</code>
	CSRF_TRUSTED_ORIGINS	<code>http://localhost:3000</code>	HTTPS URL of backend service
	CORS_ALLOWED_ORIGINS	Localhost dev origins	HTTPS URL of frontend service
	SUPABASE_DB_NAME	Local Postgres DB name	Render dashboard secret
	SUPABASE_DB_USER	Local Postgres user	Render dashboard secret
	SUPABASE_DB_PASSWORD	Local Postgres password	Render dashboard secret
	SUPABASE_DB_HOST	<code>localhost</code> or compose service name	Supabase pgBouncer pooler hostname (secret)
	SUPABASE_DB_PORT	5432 (direct)	6543 (pgBouncer pooler)
	SUPABASE_URL	Supabase project URL (<code>.env</code>)	Render dashboard secret
	SUPABASE_SERVICE_ROLE_KEY	Service-role key (<code>.env</code>)	Render dashboard secret
	POSTMARK_SERVER_TOKEN	Optional (console backend acceptable)	Render dashboard secret
	DEFAULT_FROM_EMAIL	Local placeholder	Verified Postmark sender (secret)
Frontend	FRONTEND_URL	<code>http://localhost:5173</code>	HTTPS URL of frontend service
	VITE_API_URL	<code>http://localhost:8000</code>	HTTPS URL of backend service

in Table 7.2. The `RENDER_EXTERNAL_HOSTNAME` variable is not declared in the blueprint; Render injects it automatically into every web service at runtime, and the backend reads it during settings initialization (see Section 7.2, “Security Hardening”) to extend `ALLOWED_HOSTS` and the CSRF trusted origins.

Chapter 8

Conclusion

8.1 Summary of Delivered Work

NutriChoice delivers a functionally complete full-stack web application across seven Django backend applications and a React TypeScript frontend comprising thirty domain-specific components. The system provides end-to-end user journeys covering account creation and dietary onboarding, recipe creation and multimedia upload, personalized discovery, weekly meal planning, macronutrient tracking, social graph management, and cookbook curation.

Key technical achievements of the project include:

- A **personalized feed algorithm** combining social context, ingredient familiarity, and community popularity as priority tiebreakers in a single ranked queryset computed server-side via SQL annotation, avoiding application-layer sorting.
- A **seeded-random recipe browse** providing stable, pagination-safe ordering without a full application-layer sort or server-side cursor state.
- A **brokered media upload pipeline** enabling direct-to-CDN uploads without exposing cloud credentials to the browser.
- A **two-layer state management architecture** separating TanStack Query's server state caching from context-based shared application state, with optimistic updates delivering immediate feedback across all interactive elements.
- **Security-hardened API design:** UUID-based public identifiers to prevent enumeration, DRF token invalidation on password change, HMAC-bound password reset tokens with automatic invalidation, CORS restriction, global and per-endpoint rate limiting.

- **Automated continuous security analysis** via CodeQL on every pull request to the `main` and `dev` branches.

8.2 Technical Reflections

The most consequential architectural decision was pre-computing nutritional totals in a dedicated `RecipeNutrition` model rather than aggregating on demand, like it was originally planned. This trades one additional `INSERT` per recipe save for a simplification of read-time macro aggregation: the meal-planning dashboard executes a single-pass SQL `SUM` regardless of recipe ingredient count, and the result is always consistent with the stored recipe definition.

The choice of DRF token authentication over JWT was a well thought design implementation. JWTs are self-contained and cannot be revoked server-side without a denylist or 5-minute expiry windows. The password-reset flow requires that all active sessions be closed the moment a password changes. DRF tokens in a server-controlled table make this easy: a single `Token.objects.filter(user=user).delete()` call atomically invalidates every session.

The brokered signed-URL upload pattern adds a round trip compared to a direct server-side upload proxy, but it offloads binary transfers to Supabase's edge network. This reduces both bandwidth cost and latency for geographically dispersed users while freeing the application server for API requests.

8.3 Future Work

Possible extensions include:

- **AI-Powered Ingredient Substitution:** suggesting ingredient substitutions aligned with a user's dietary goals.
- **Grocery list generation** from a week's meal plan, aggregating and deduplicating ingredient quantities across all planned meals.
- **Partner Ship with Supermarket chains** for users to be able to export their NutriChoice grocery list to their supermarket of choice. Making checkout seamless and making grocery shopping for nutrition easier.
- **Push notifications** for social interactions (new followers, likes on published recipes).