Major Project: Final Report

Symprove: A React Native Application for Systematic Food Reintroduction with Curated Recipes, Meal Planning and Symptom Tracking for a Digital Approach to IBS Management

---

N00220081

Christina Kaenmuang

BSc (Hons) Creative Computing

Module: Y4 Major Project

Project Coordinator: John Dempsey

Supervisor: Catherine Noonan

Second Reader: Mohammed Cherbatji

Table of Contents

1 Introduction & Project Context

Symprove is a cross-platform mobile application designed to support individuals managing irritable bowel syndrome (IBS) through the complete low-FODMAP dietary protocol. The application provides a curated recipe library, an ingredient classification index, structured symptom and meal logging, and a guided food reintroduction protocol within a single system.

IBS is a functional gastrointestinal disorder characterised by abdominal pain, cramping, bloating, flatulence, diarrhoea, and constipation, arising from disrupted gut-brain interaction (Cleveland Clinic, 2023). It affects approximately 10–15% of the global population. (Guts UK!, 2026). Dietary triggers, particularly foods containing FODMAPs (fermentable oligosaccharides, disaccharides, monosaccharides, and polyols), are a primary driver of symptom onset. Effective dietary management typically involves the low-FODMAP protocol, a three-phase clinical process consisting of a strict elimination phase (2–6 weeks), a structured reintroduction phase (several months), and a personalisation phase to establish the individual's long-term dietary baseline (Cleveland Clinic, 2023).

Access to the clinical support required to complete this protocol is limited. Specialist dietitian appointments carry long waiting times, and dietitian-led FODMAP programmes range from €90 to over €600 per visit (Bosman, et al., 2023), with multiple sessions required across all three phases. Indirect costs, including missed workdays, compound this burden (Bosman, et al., 2023). Restrictive diets impose further challenges: time-consuming meal planning, social difficulties when dining out, and the cognitive load of tracking multiple food groups and symptoms over an extended period (Bosman, et al., 2023). Without professional guidance, patients frequently abandon the protocol before completion or conduct food testing incorrectly, producing inconclusive results (Cleveland Clinic, 2023).

Symprove addresses these barriers through a digital system modelled on the clinical FODMAP protocol. The recipe database contains over 200 low-FODMAP recipes sourced from “A Little Bit Yummy”, a registered dietitian-authored resource. The ingredient index classifies over 400 ingredients by FODMAP status as well as allergens, reducing the need for manual lookups. Symptom and meal logging provide a structured digital alternative to paper journals. The guided reintroduction protocol replicates clinical testing procedures from Monash University, including appropriate challenge food sequencing, three-day testing periods, and three-day washout intervals. Together, these features provide a cost-accessible pathway through the full dietary management process without requiring specialist support.

1.1 Project Aim & Objectives

This project aims to develop a mobile application that supports individuals with IBS through the complete low-FODMAP dietary protocol, providing guided food reintroduction, recipe discovery, meal logging, and symptom tracking within a single cohesive system.

The project objectives are:

Implement a user authentication system and profile management that allows users to configure their current FODMAP phase, dietary restrictions, and allergens to personalise recipe recommendations and app behaviour.

Curate a minimum of 50–100 low-FODMAP recipes with a phase-based filtering system, ensuring users have access to safe meal options during elimination and controlled ingredient testing during reintroduction.

Develop a symptom tracking system that enables users to log meals and symptoms with timestamps and severity ratings, providing a digital alternative to paper journals and facilitating pattern recognition between consumed foods and IBS symptoms.

Develop a symptom logging system that captures structured symptom data, including pain scores, bloating scores, stool type, stool frequency, stress level, and sleep quality, with timestamps, to support pattern identification between diet and symptoms.

Create a meal planning system that allows users to select recipes from the database, organise them into daily or weekly meal plans, and maintain a meal history linked to symptom logs for correlation analysis.

Integrate USDA FoodData Central API to provide nutritional information for recipes, supporting informed dietary decisions and ensuring users meet nutritional requirements during restrictive elimination phases.

Build a secure backend using Supabase and PostgreSQL, with Row Level Security enforced across all user health data tables, in compliance with GDPR principles.

Conduct unit testing for classification and correlation logic, integration testing for database operations, and usability testing with 2–3 participants to validate core user flows.

1.2 Success Criteria & Scope

Functional Completeness: The application must deliver all core features that support the low-FODMAP protocol. Users must be able to register accounts, configure their dietary restrictions and allergens, browse and save low-FODMAP recipes, log meals and symptoms, and follow the guided reintroduction protocol.

Clinical Accuracy: The recipe database must contain a minimum of 50 low-FODMAP recipes sourced from a verified dietitian-authored resource. The reintroduction protocol must follow established clinical guidelines, including appropriate challenge food selection, three-day testing periods, and three-day washout periods between challenges.

User Experience: The application must provide intuitive navigation between features with a clear visual hierarchy. Users must be able to complete core tasks: finding recipes, logging symptoms, and following reintroduction steps without support.

Technical Performance: The application must function reliably on both iOS and Android platforms, with database queries completing in under two seconds and smooth transitions between screens. User authentication must be secure with no data loss during normal operation.

Accessibility: The interface must target WCAG 2.1 Level AA standards, including appropriate colour contrast ratios and screen reader compatibility, to ensure usability across a range of accessibility needs.

Data Security: User health data must be stored securely with appropriate encryption at rest and in transit. The backend must implement Row Level Security policies to prevent unauthorised access to user profiles, symptom logs, and meal history.

Code Quality: The codebase must follow React Native and Expo mobile development best practices, including appropriate component architecture, consistent error handling, and adequate inline documentation.

Testing Coverage: The application must undergo unit testing for correlation logic, integration testing for Supabase database operations, and usability testing with 2–3 participants to validate core user flows and interface design**.**

2 Research & Background

2.1 Literature Review

The literature review addresses two areas that directly informed the design of Symprove: the clinical evidence behind the low-FODMAP dietary protocol, and the barriers patients face when attempting to implement it without professional support. A review of comparable existing applications is presented in Section 3.1 as part of the requirements gathering process.

2.1.1 IBS and Dietary Management

Irritable bowel syndrome is a functional gastrointestinal disorder with a global prevalence of 10–15% (Guts UK!, 2026). Dietary management has become a main part of treatment, with at least two-thirds of patients associating symptom onset with specific food intake (Cozma-Petrut, Loghin, Miere, & Dumitrascu, 2017). The low-FODMAP diet is among the most studied dietary interventions for IBS. It involves restricting fermentable carbohydrates across five subcategories, known as FODMAPs (oligosaccharides, disaccharides, monosaccharides, and polyols), requiring classification of foods across vegetables, cereals, legumes, dairy, fruits, and sweeteners (Cozma-Petrut, Loghin, Miere, & Dumitrascu, 2017). This level of dietary complexity extends beyond simple food avoidance.

FODMAPs are poorly absorbed in the small intestine. Upon reaching the large intestine, they are fermented by gut bacteria and draw water into the bowel through osmotic pressure. In individuals with IBS, this produces excess gas, bloating, abdominal pain, and altered bowel habits (Lambiase, et al., 2024). The low-FODMAP diet targets this mechanism directly and has become one of the most evidence-supported dietary approaches to IBS symptom management (Lambiase, et al., 2024).

The protocol continues in three sequential phases: a strict elimination phase lasting two to six weeks, a systematic reintroduction phase spanning several months, and a personalisation phase in which the individual's long-term dietary baseline is established. Individual variation is central to the process; patients must find their personal tolerance levels through systematic food testing, meaning outcomes cannot be standardised and must be adapted to each person's symptom response (Weznaver, et al., 2024).

(Moayyedi, et al., 2017). In IBS-D, the osmotic and fermentative effects of FODMAPs are primary drivers of urgency and loose stool, making symptom response during the elimination phase typically pronounced. In IBS-C, restriction may reduce bloating and pain, but does not directly resolve constipation. In IBS-M, the reintroduction phase is particularly important for finding which FODMAP subgroups contribute to each end of the symptom spectrum (Moayyedi, et al., 2017). Capturing IBS subtype is therefore relevant to interpreting an individual's response across all three phases.

Cozma-Petrut et al. (2017) emphasise that "dietary counselling through a specialised dietitian is a key towards the success of this fairly complex dietary approach." Without expert support, guidance tends to focus narrowly on food avoidance, which "leaves these subjects with unnecessarily self-restrictive

diets, which could result in nutritional deficiencies" (Cozma-Petrut, Loghin, Miere, & Dumitrascu, 2017). This highlights the need for tools that go beyond prohibition and support nutritionally balanced food choices throughout each phase of the protocol.

2.1.2 Patient Barriers to Implementation

The clinical framework for IBS dietary management is well-established, but patient adherence is consistently reported as a significant challenge. Weznaver et al. (2024) conducted a qualitative study examining patient experiences during a structured dietary intervention and identified several compounding barriers. Time and energy demands were prominent; participants described the shift to structured daily cooking as burdensome, with one noting it was "a relatively big step going from cooking a few times a week, to cooking more, and more structurally. Pretty much every day!" (Weznaver, et al., 2024). Notably, the gastrointestinal symptoms themselves contributed to this burden, with participants reporting that their condition produced "draining and fatigue, which made it even more difficult to have the energy for the necessary dietary changes" (Weznaver, et al., 2024). The condition creates a barrier to its own management.

Social contexts added further difficulty. Patients felt "socially provocative" when declining certain foods and encountered resistance from unsupportive social environments (Weznaver, et al., 2024). Even in a well-resourced clinical intervention that provided recipes, menus, and home-delivered groceries, participants found the four-week programme reached "the maximum of what they could cope with" (Weznaver, et al., 2024).

A key finding across both studies is that long-term dietary adherence is shaped less by clinical optimality and more by alignment with the patient's daily circumstances. Weznaver et al. (2024) found that sustained changes were "not primarily related to the most effective changes for symptom alleviation but instead comprised changes that were not too challenging to maintain in the patients' current life situation." This is consistent with the position of Cozma-Petrut et al. (2017), who argue that "individualised dietary options should be encouraged to achieve long-term dietary changes."

2.1.3 Digital Tools for Dietary Management

Both studies converge on a structural difference: periodic clinical consultations cannot support the continuous, daily decision-making that IBS dietary management demands. Weznaver et al. (2024) found that participants had not been able to initiate or sustain dietary changes independently, despite expressing a desire to do so. Accessing specialist dietitian support remains a financial and logistical barrier for many patients, with programmes costing between €90 and over €600 per course (Bosman, et al., 2023) moreover, it requires multiple sessions across all three protocol phases.

The features of Symprove were designed to address this by implementing clinical guidance within an accessible mobile application. The features of the application, including the recipe library, ingredient classification index, symptom logging, and guided reintroduction protocol, were each shaped in direct response to the barriers identified in the literature.

2.2 Technical Research

2.2.1 FODMAP Classification Methodology

The ingredient classification system is built on an open-source FODMAP dataset published on GitHub under the name **'fodmap_list'** (oseparovic). It provides a structured list of ingredients with FODMAP

status, food category, maximum serving quantity, and a subcategory breakdown across oligosaccharides, fructose, polyols, and lactose (see Figure 1). The dataset cross-references multiple clinical and dietitian sources and was uploaded directly to the **master_ingredients** table in Supabase, forming the reference base for all ingredient lookups, meal logging and correlation analysis.

The dataset's original JSON structure was adapted into a normalised relational schema to suit the application's requirements. The subcategory values were flattened from a nested JSON object into individual integer columns with database-level CHECK constraints, the integer ID was replaced with a UUID primary key, and two additional arrays were introduced: allergens, to support allergen-based filtering, and aliases, to support fuzzy search across alternative ingredient names. The full schema design and the rationale for these decisions are discussed in Section 4.4.


```
{
    "id": "14",
    "name": "Almond butter",
    "fodmap": "low",
    "category": "Cooking ingredients, Herbs and Spices",
    "qty":"1 tbsp",
    "details": {"oligos":0,"fructose":0,"polyols":0,"lactose":0}
},
```


Figure 1 - Original JSON structure of a single entry from the osepariovic/fodmap_list dataset, showing fodmap status, recommended serving size, and fodmap subcategory values (Image from fodmap_list README.md (oseparovic))

2.2.2 Cross-Platform Mobile Development

React Native with Expo was selected to target both iOS and Android from a single codebase. Research focused on Expo Router's file-based routing system, which ties screen structure directly to the file system. This made navigation predictable and easier to maintain as the project scaled. Platform-specific UI differences, particularly in tab navigation, were handled using Expo's native tab component, which renders platform-appropriate controls automatically. Responsive layout management required research into React Native's Dimensions API and SafeAreaView to handle screen size variation across devices. All knowledge on the use of the framework was referenced using the Expo Documentation (Expo).

2.2.3 Backend Architecture and Data Security

The backend infrastructure required research across three areas: the selection of a database system, the implementation of row-level access control, and compliance with GDPR for applications handling personal health data.

2.2.3.1 PostgreSQL

Supabase uses PostgreSQL as its core database engine. Research into database system selection focused on comparing PostgreSQL with MySQL, the two most widely adopted open-source relational database management systems. PostgreSQL is a fully ACID-compliant object-relational database management system (ORDBMS) that provides strict data integrity, advanced native data types including JSONB and arrays, and a sophisticated query planner

suited to complex relational queries (IBM, n.d.). MySQL, by contrast, is optimised primarily for read-heavy workloads and simpler query patterns. For Symprove, which requires linked health records across user profiles, ingredient logs, symptom entries, meal history, and reintroduction progress, PostgreSQL's handling of relational complexity and high-frequency write operations was the more appropriate fit. Its native support for array types was directly applicable to the allergens and aliases columns in the master_ingredients table, which stores variable-length lists without requiring a separate join table (see Section 4.4 for further details.

2.2.3.2 Row Level Security

Row Level Security (RLS) is a PostgreSQL feature that restricts which rows a user can read, insert, update, or delete in a table, based on policies defined at the database level rather than at the application layer (PostgreSQL). When RLS is enabled on a table, every query issued against it has the policy condition appended automatically as an invisible filter. If a table has RLS enabled but no policies are defined, a default-deny rule applies, meaning no rows are accessible to any user. Policies are expressed as Boolean conditions evaluated against the authenticated session (PostgreSQL), for example, USING (user_id = auth.uid()), which ensures a user can only access rows where the user_id column matches their own authenticated identifier.

Implementing access control at the database level, rather than relying solely on application-layer logic, provides a more robust security boundary. If the application layer were to contain a logic error, the RLS policy would still prevent unauthorised data exposure (PostgreSQL). Supabase integrates RLS directly with its authentication layer, and the official Expo React Native quickstart documentation was used to configure the client-side Supabase instance and authentication flow (Supabase).

2.2.3.3 GDPR Compliance

Health data is classified as special category data under Article 9 of the General Data Protection Regulation (GDPR), which imposes stricter processing requirements than standard personal data (Cudny, 2026). Applications handling such data must obtain explicit and informed consent, apply data minimisation principles by collecting only what is functionally necessary, provide users with rights to access, correct, and delete their data, and implement technical safeguards, including encryption at rest and in transit (Cudny, 2026). In the event of a data breach, notification to the relevant supervisory authority is required within 72 hours (Cudny, 2026).

These requirements informed several decisions in Symprove's architecture. Supabase encrypts data at rest and in transit by default, satisfying the technical security requirements (Supabase). Data minimisation was applied by collecting only the health fields directly required by the application, including symptom severity scores, stool type, stress level, and meal entries, without capturing location data, device identifiers, or other incidental personal information. Row Level Security ensures each user's records are inaccessible to others, directly supporting the user rights requirement. The authentication flow built on Supabase Auth allows users to delete their accounts and associated data (Supabase). A privacy-by-

design approach was applied from the initial schema design stage rather than as a post-implementation addition.

2.2.4 Fuzzy Search

Ingredient lookup presented a practical challenge: users cannot be expected to input ingredient names with exact precision, particularly in a mobile context where typing is more error-prone. Research into client-side search solutions identified Fuse.js as suitable for this use case. Fuse.js is a lightweight, zero-dependency fuzzy-search library that uses the Bitap algorithm to return relevant results despite typographical errors or partial input (Fuse.js). It supports subjective field searching, allowing results to be ranked by relevance across multiple fields such as ingredient name and aliases. Research into its application in React Native confirmed its suitability for mobile environments, including its performance with datasets of 500 or more items without introducing lag (Agrawal, 2025).

2.2.5 AI Integration

The Google Gemini API was integrated to support three features (see Section 0 for implementation detail):

1) A fallback FODMAP lookup for ingredients not found in the database
2) Image-based ingredient identification from meal photographs
3) Ingredient suggestion from a typed meal name

Research into the Gemini API focused on its multimodal capabilities. Gemini models are designed to process both text and image inputs within a single prompt, supporting object detection, food recognition, and document parsing across common image formats, including PNG, JPEG, and WEBP (Gemini API). The API offers several model tiers, with Flash variants providing faster response times and higher throughput suited to mobile applications, and Pro variants offering more advanced reasoning for complex tasks (Gemini API). Symprove uses the Flash model, which is available under the free tier and provides 250 requests per day, sufficient for development and usability testing at the current scale.

Research also focused on prompt engineering to produce structured, consistent output that could be reliably parsed and mapped back to database records. Because Gemini responses are returned as natural language by default, prompts were constructed to constrain output to a specific JSON format. Google's prompt design documentation recommends providing clear, specific instructions and using structured output when a defined JSON schema is required, rather than relying on natural language descriptions of the expected format (Gemini API).

2.2.6 Web Scraping Pipeline

The recipe database required populating with a substantial volume of structured data that was not available as a downloadable dataset. This led to research into automated data pipelines as a method for extracting and structuring content from existing web sources. Research into data automation approaches, including the use of scripts to systematically retrieve and process data from external systems, identified web scraping as a suitable technique for this purpose (Jan Kadlec, 2022).

Web scraping is the automated process of extracting content from websites by sending HTTP requests, downloading the returned HTML, parsing it to locate specific elements, and saving the extracted data in a structured format such as JSON or SQL (Joe Osborne, 2024). It functions as a programmatic alternative to manually copying content and is commonly applied to data aggregation tasks where information exists in a consistent structure across multiple pages (Joe Osborne, 2024).

Initial research into web scraping was conducted using Python and the Beautiful Soup library, which provides tools for parsing HTML and extracting elements using tag and attribute selectors (Martin Breuss).
This established a working understanding of the scraping process:

1. Navigating to a URL
2. Retrieving the page source
3. Locating target elements within the HTML tree
4. Extracting their content.

However, the recipe source used for Symprove rendered its content dynamically using JavaScript, meaning a standard HTTP request returned an incomplete page without the recipe data. Beautiful Soup alone cannot execute JavaScript, as it only parses static HTML (Martin Breuss).

This limitation led to research into Puppeteer, a Node.js library that controls a headless browser instance capable of executing JavaScript, simulating user interactions, and exposing the fully rendered page for parsing (Furqan Ahmad, 2024). Because Symprove is built in JavaScript, migrating the scraping implementation from Python to Node.js using Puppeteer and Cheerio also ensured the pipeline remained within the same language ecosystem as the application (Joe Osborne, 2024). Cheerio provides jQuery-like HTML parsing using CSS selectors and was used alongside Puppeteer to extract structured recipe data from each loaded page (Furqan Ahmad, 2024).

Research into ethical and functional scraping practices informed two additional implementation decisions:

1. Web servers use the User-Agent string in HTTP request headers as a primary mechanism for identifying automated traffic.
    a. Requests that carry no User-Agent, or that use default library identifiers such as python-requests/2.25.1, are commonly flagged as bots and rejected with 403 Forbidden responses, CAPTCHA challenges, or incomplete HTML (Muhammad Ikramullah Khan, 2025).
    b. To avoid being blocked while also being transparent about the nature of the tool, a custom User-Agent string was applied to all requests, identifying the scraper as a non-commercial academic research tool rather than mimicking a standard browser.
2. The second decision was the introduction of a 2.5-second delay between page loads.
    a. Sending requests in rapid succession without a delay can generate traffic volumes comparable to a denial-of-service attack from the server's perspective, which may result in IP-level bans and place an unreasonable load on the host server (Muhammad Ikramullah Khan, 2025).

2.2.7 Correlation Analysis

Correlation analysis is a statistical method used to measure the strength and direction of a relationship between two variables (Dovetail Editorial Team, 2023). A correlation coefficient ranging from -1 to +1

expresses this relationship: a value near +1 indicates that both variables increase together, a value near -1 indicates an inverse relationship, and a value near 0 indicates no meaningful association. A key limitation of correlation analysis is that it identifies co-occurrence, not causation; two variables may correlate consistently without one directly causing the other (Dovetail Editorial Team, 2023).

In the context of IBS dietary management, this method is applied to identify relationships between food intake and reported symptom severity. Meal logs and symptom records are compared across time to determine whether specific ingredients are consistently associated with elevated symptom scores. This approach is used in clinical FODMAP protocols to support trigger food identification (Cleveland Clinic, 2023).

Symprove implements this using the delta method, which calculates the difference between a variable measured under two conditions: a period of exposure and a baseline period (Lim, 2021). Expressed as $\Delta X = X_{t1} - X_{t0}$, where X_{t1} is the mean symptom score on days a given ingredient was consumed, and X_{t0} is the mean symptom score on days it was not, the resulting delta indicates whether symptoms were consistently elevated following consumption of that ingredient. This approach accounts for the within-person variability inherent in IBS symptom data and avoids the need for a population-level baseline.

3 Requirements Analysis

3.1 Requirements Gathering

3.1.1 Similar Applications:

A review of existing IBS management applications was conducted to identify the features currently available to users and to inform the requirements of Symprove.
Three applications were examined:

1. The Monash FODMAP App
2. Cara Care
3. Bowelle

Monash FODMAP App was selected as it represents the most clinically authoritative tool in this space, developed by the research team that created the low-FODMAP diet. Cara Care was selected as it had the most similarities to Symprove, and Bowelle was selected as an example of broader IBS symptom tracking applications.

Feature	Monash	Cara Care	Bowelle	Symprove
Symptom tracking	No	Yes	Yes	Yes
FODMAP ingredient database	Yes, traffic light rating with safe serving size indication	No	No	Yes, traffic light rating with FODMAP subgroup score and allergen, but no safe serving size indication
Recipe library	80+ recipes	Yes, tailored to the individual	No	200+ recipes, tailored to the individual
Guided FODMAP protocol	No, only contains an information booklet	Yes, tailored to the individual	No	Yes, tailored to the individual
Food-symptom correlation	No	Yes	Yes	Yes
AI-Assisted meal logging	No	No	No	No
Platform availability	iOS, Android	iOS, Android	iOS only	iOS, Android
Free access	No, €8.99 one time purchase	Yes, but needs a prescription to access	Yes	

Table 1 - Comparison of existing apps against Symprove

The comparison revealed a clear lack of features across different apps.

- The Monash FODMAP App provides strong dietary reference content but does not offer symptom tracking or a guided reintroduction protocol.

- Cara Care seems to be the better option out of the two other options as it provides symptom tracking with food-symptom correlation, tailored recipes with a guided diet protocol for all three stages, elimination, reintroduction and personalisation
- Bowelle is the simplest as its only purpose is a digital journal with visual analytics
- No existing application reviewed combines a curated low-FODMAP recipe library, an ingredient classification index, symptom tracking with correlation analysis, and a guided three-phase reintroduction protocol, as well as being AI-integrated. Making Symprove the first to include all these features, three different apps in one.

3.1.2 User Interview

Two informal interviews were conducted with individuals who manage dietary restrictions related to gastrointestinal conditions.

Participant Profile:

Participant 1: Adult with 50–60 years of experience managing a self-directed restrictive diet. She has previously had multiple GP visits and consultations with dietitians with no success. Before, she had a food journal to keep track of her meals and symptoms, which she gave up on. She currently uses a mental note to keep track of her diet; she is following a dairy-free, gluten-free, low-histamine and low-sulphur diet. She is unsure about what she can and cannot eat and chooses to follow this diet as a precaution. According to Claude Ai, she is avoiding approximately 100-150 ingredients, and roughly 60-80 of those ingredients would be everyday household staples (Sonnet 4.6, 2026).

Participant 2: Adult with 3 years of experience managing a self-directed restrictive diet. She has done food allergy and intolerance tests after experiencing years of unexplained illness; her results were dairy and gluten intolerance, and it is currently the diet she is following, she explained, which has improved her well-being. She keeps a mental note of her diet and has never used any apps.

The interview responses were summarised into one; these were the findings:

- **Recipe discovery is time-consuming and unreliable**.
    - **Participant 1** reported that finding a new suitable recipe requires significant time, and recipes frequently contain unusable ingredients, requiring substitution or abandonment.
    - **Participant 2** reported she did not find it too difficult to find a new recipe, as she finds substituting gluten and dairy products easy. However, she found it difficult to replicate the taste of her past favourite foods, specifically desserts and would spend a significant amount of time finding the perfect recipe.
    - **Both participants** relied solely on internet searches for recipe discovery, specifically YouTube and food blog websites. They both reported eating the same set of meals every day to save time finding new recipes.
- **Efficiency of digital symptom and meal tracking.**
    - **Participant 1** reported maintaining multiple dietary tracking journals over time, which were eventually abandoned due to the time and effort required. While the journals had assisted in identifying potential triggers, she expressed uncertainty about the accuracy of her conclusions, attributing this to the limitations of handwritten self-reporting. She currently relies on memory and avoids a broad range of ingredients as a precaution, though she acknowledged this is not a

sustainable or desirable long-term approach. She liked the idea of being able to log something digitally, where all her records could be easily accessed in one place, as well as speeding up the process.

- **Structured food testing is not being followed.**
    - **Both participants** had abandoned food reintroduction testing entirely, defaulting to a restricted fixed diet rather than systematically identifying tolerances. This reflects the findings in the literature review regarding the difficulty of sustaining the FODMAP protocol without structured support (Weznaver, et al., 2024).
- **Automatic filtering and visual patterns were identified as the most valuable potential features.**
    - **Participant 1** expressed a preference for ingredient-first recipe search, automatic filtering based on known triggers, and chart-based visual correlation between food and symptoms. Complexity and logging time were cited as the primary reasons they would stop using an app.
    - **Participant 2** stated that a feature which only shows recipes with ingredients she could have would be her main reason for using the app

These findings directly informed the core requirements of Symprove, particularly the recipe filtering system, the symptom correlation feature, symptom & meal logging and the well-being graphs.

3.2 Requirements Modelling

3.2.1 Persona

Click on the following link to view the personas:
https://miro.com/app/board/uXjVHbSgkhI=/?moveToWidget=3458764669730628887&cot=10

3.2.2 Functional and Non-Functional Requirements

FUNCTIONAL REQUIREMENT	Category
Allow users to register an account with an email and a password	**Authentication**
Allow users to discover their IBS type	**Profile**
Log in and log out securely.	**Authentication**
Reset password securely	**Authentication**
Configure users' fodmap phase, dietary restrictions and allergens	**Profile**
Display 200+ low-FODMAP recipes	**Recipe Discovery**
Recipe search and filter option using fuzzy search	**Recipe Discovery**
Save recipes	**Recipe Discovery**

Display recipe details, including name, ingredients, image, instructions, and nutritional information	**Recipe Discovery**
Ingredient index to discover FODMAP classifications	**Ingredients**
Ai-assisted meal logging	**Data Logs**
Symptom logging	**Data Logs**
Display correlation between meals and symptoms logged	**Data Logs**
Guide users through each FODMAP phase	**Dietary Protocol Guidance**
Track and display results from each FODMAP group tested	**Data Logs**

NON-FUNCTIONAL REQUIREMENT	Category
Database queries should complete under two seconds	**Performance**
The app should function on both iOS and Android	**Performance**
RLS policies should protect user health data	**Security**
User data is encrypted at rest and in transit	**Security**
Follow GDPR compliance for health apps **(Cudny, 2026)**	**Security**
Target WCAG 2.1 level AA accessibility standards	**User experience**
Implement user-friendly error messages for authentication and data failures.	**User experience**
Capture and monitor errors in real time during development	**Reliability**

3.2.3 Use Case Diagram

Click the following link to view the use case diagram:
https://miro.com/app/board/uXjVHbSgkhI=/?moveToWidget=3458764669743443364&cot=14

4 Design

4.1 System Architecture

Symprove's system follows a layered architecture, where each layer has a distinct responsibility. The presentation layer handles all user interface concerns, the business logic layer manages state and application rules, and the API layer handles all communication with external services. The backend is provided by Supabase, which supplies both a PostgreSQL database and an authentication service. Google Gemini AI is used for AI implementation, and Sentry is used for error monitoring. This separation keeps the codebase organised, easy to manage and ensures that each part of the system can be developed and tested independently.

4.1.1 Core Component Architecture

Figure 2 and Figure 3 shows the **component architecture** of Symprove (full diagram can be viewed in the submitted folder "3 Design", the file named "Component Diagram"). The three internal layers are:

1) Presentation layer (Screens)
2) Business logic layer (Custom Hooks and Services)
3) Data access layer (API Layer).

These connect externally to a Supabase backend and third-party services.

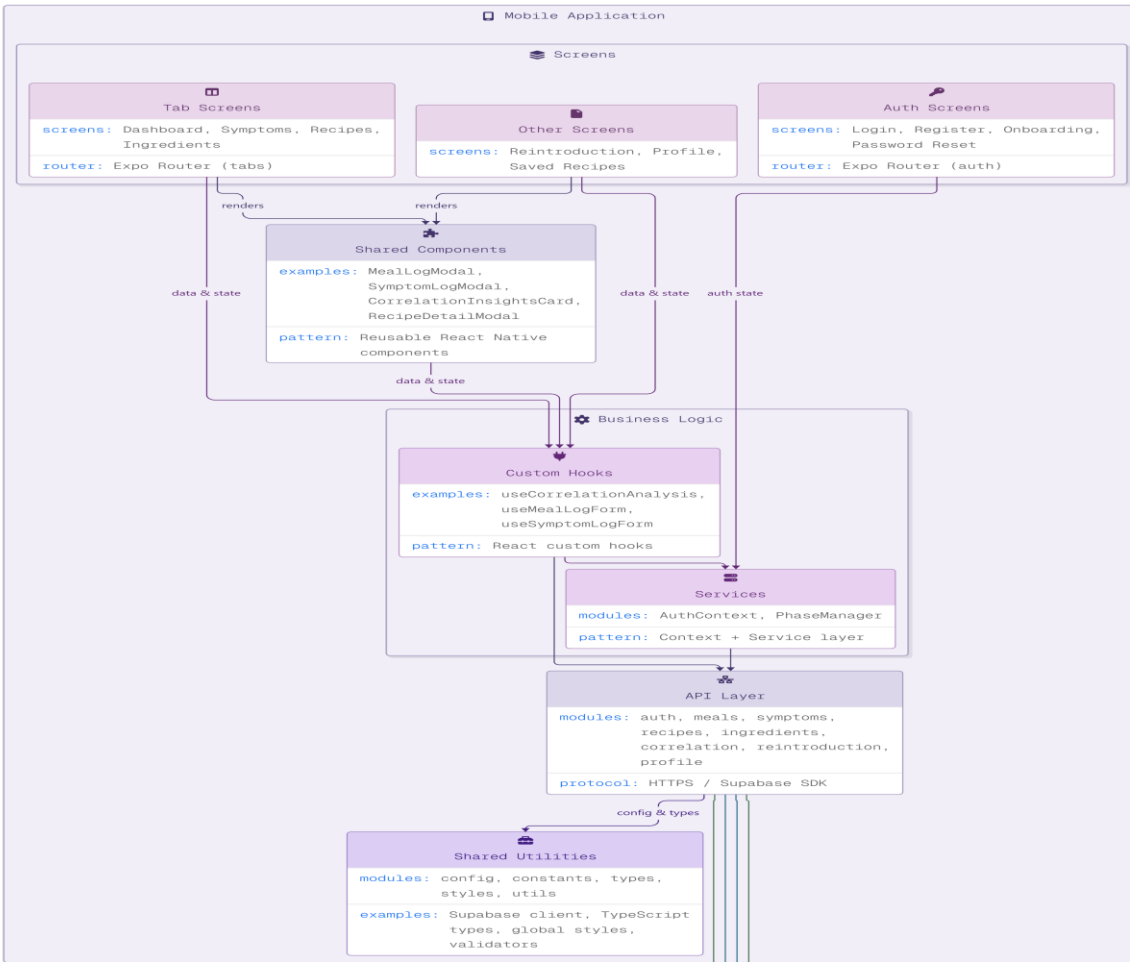


Figure 2 – Component Architecture: Client Layer

The presentation layer (Figure 2) uses Expo Router's file-based routing system. Screens are divided into three groups:

1) **Authentication** screens such as Login, Register and Onboarding
2) **Tab** screens including Dashboard, Symptoms, Recipes and Ingredients
3) **Other** screens, such as Reintroduction and Profile, remain outside the other two folders, kept in the general **App** folder

The business logic layer (Figure 2) is made up of custom React hooks and a services module. Custom hooks such as useCorrelationAnalysis and useMealLogForm handle data fetching and state management, keeping screen components focused on rendering. The examples for the services module are AuthContext, which provides the authenticated user session to the rest of the application, and PhaseManager, which is responsible for detecting and executing FODMAP diet phase transitions automatically. Shared utilities such as TypeScript type definitions, global styles, constants and configuration files are grouped separately and referenced across all layers of the application.

The API layer (Figure 2) is the only part of the application that communicates **with external systems** (Figure 3). Each module within the API layer corresponds to a specific domain, such as meals, symptoms or recipes, and communicates with Supabase using the Supabase TypeScript SDK. All API modules import the Supabase client from the central config file, which is where the client is initialised and shared across the application. Supabase provides two key services: a PostgreSQL database with Row Level Security (RLS) to ensure users can only access their own data, and an authentication service using JWT tokens. The API layer also connects to Google Gemini AI, which is used to match ingredient names that cannot be resolved directly against the master ingredient database. Sentry is integrated throughout the API layer to capture and report errors with relevant context.

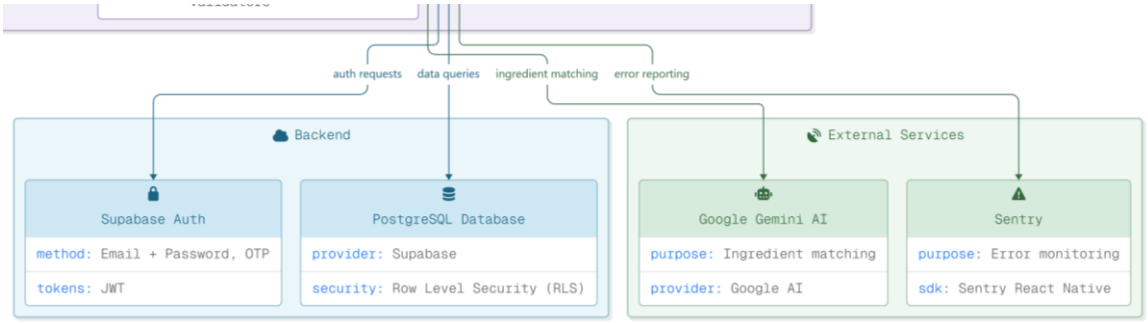


Figure 3 - Component Architecture: External System (Backend + External Services)

4.1.2 Architectural & Design Patterns

Figure 4 shows the folder structure of Symprove, The structure follows a layered architecture pattern, where code is organised into distinct folders, each with a specific responsibility, separating the presentation layer, business logic and data access into clearly defined locations (baeldung, 2021).

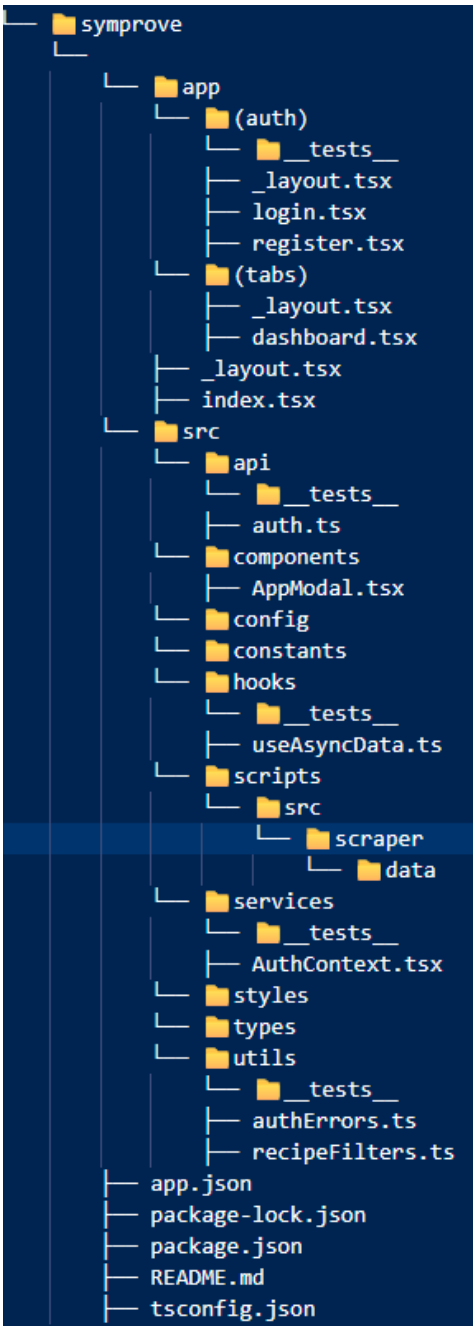


Figure 4 - Folder Structure of app with example files

- The app/ folder contains all screens and is managed by Expo Router. Unlike standard React Native navigation, Expo Router uses a file-based routing system where the folder structure automatically determines the navigation structure of the application. The folder structure shown in Figure 4, separates unauthenticated screens, tab screens and other screens into distinct locations. Further details on the navigation implementation can be found in Section. 5.
- The src/ folder contains all non-screen code and is divided into subfolders each with a single responsibility.
    - The api/ folder handles all communication with Supabase and would include CRUD functionalities and other complex logic.
    - The components/ folder contains reusable UI components shared across multiple screens.
    - The hooks/ folder contains all custom hooks responsible for data fetching by calling functions defined in the api/ folder and state management.
    - The services/ folder contains application-level logic and integrations that are shared across the entire application, regardless of what screen the user is on.
    - Shared resources such as type definitions, constants, styles, configuration and utility functions are each kept in their own dedicated folder.
    - The __tests__ folders are co-located within each folder, meaning tests sit alongside the code they are testing, rather than in a separate location. This makes it clear which tests belong to which file.
    - The scripts/ folder contains data scrapers and seeders used during development and is entirely separate from the main application code.

4.1.3 Data Flow & Interactions

Click on the following link to view the full data flow sequence diagram: Flow Sequence Diagram
The full diagram covers the full user journey from registration through to receiving correlation-based feedback on food intolerances. All communication between the application and Supabase takes place over HTTPS using the Supabase TypeScript SDK, with
Row Level Security ensures every query only returns data belonging to the authenticated user.

A close-up of the diagram in

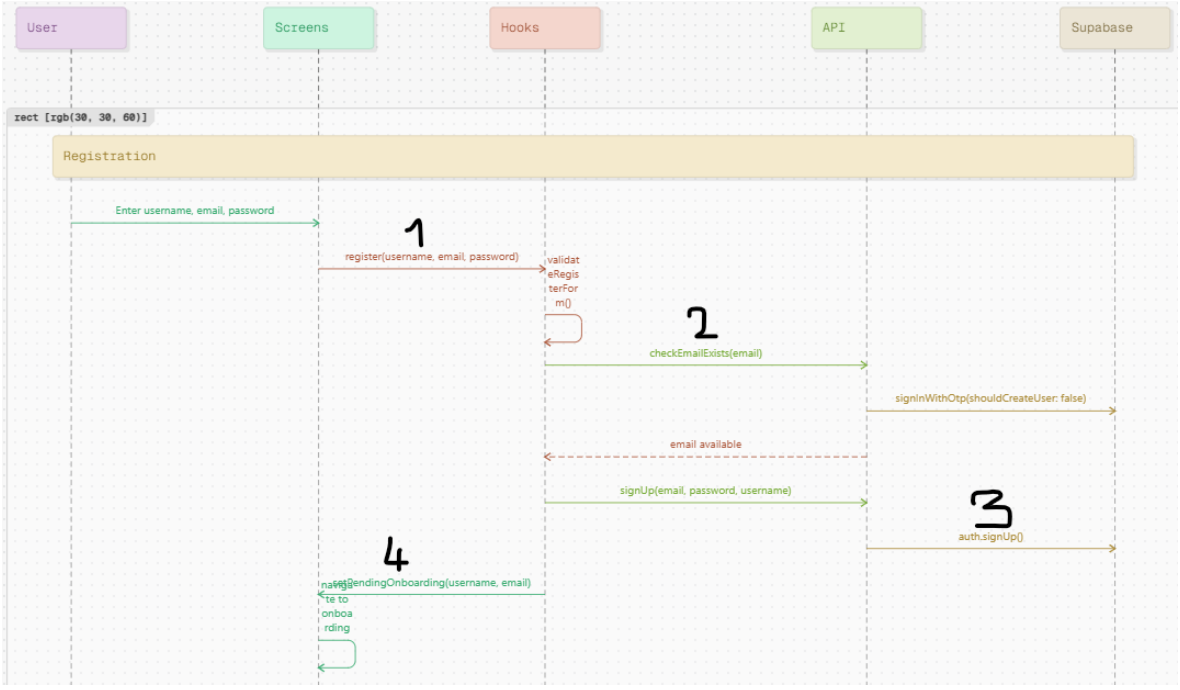


Figure 5 shows the registration of a new user:

1) The RegisterScreen passes the user's credentials to the useAuthActions hook, which first validates the form locally before.
2) calling checkEmailExists in the API layer. The API layer sends a request to Supabase Auth to verify whether the email is already in use.
3) If the email is available, signUp is called in the API layer, which creates the account in Supabase Auth and returns a session.
4) The user's username and email are then stored temporarily in AuthContext, and the user is navigated to the onboarding screen.

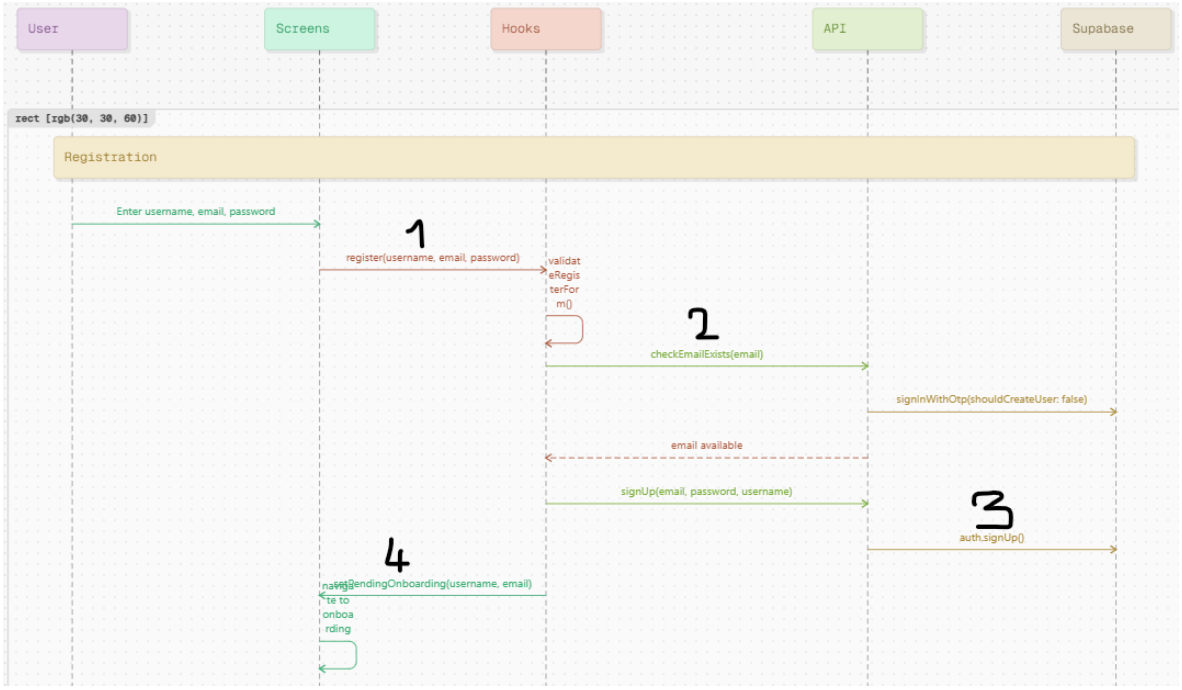


Figure 5 - Registration Sequence Diagram

During onboarding (Figure 6),

1) useOnboarding calls the API layer to fetch the list of available allergens from Supabase before the user reaches that step, once the user has selected their allergens and symptoms.
2) The hook determines the user's IBS type locally using determineIbsType without any external call.
3) When the user confirms they want to start the elimination phase
4) useOnboarding calls the API layer to save the user's IBS type, current phase and phase start date to the profiles table in Supabase, followed by a second call to save their selected allergens to the user_allergens table.

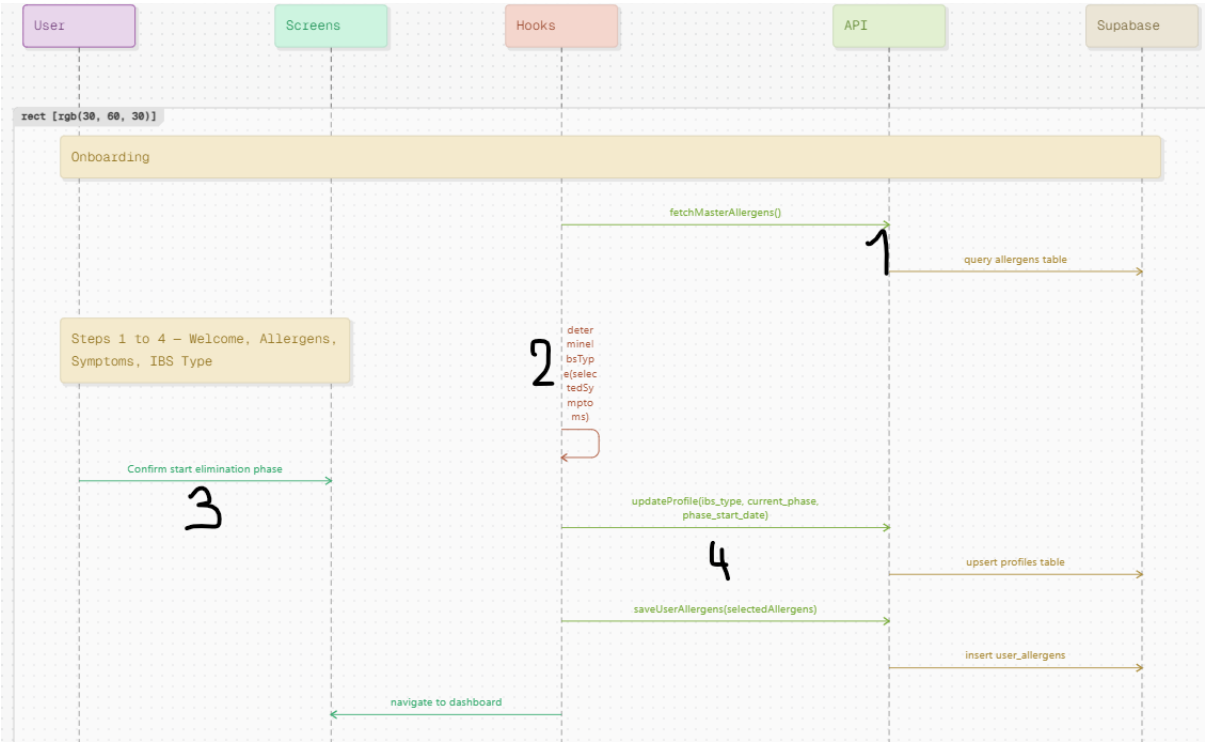


Figure 6 - Onboarding Sequence Diagram

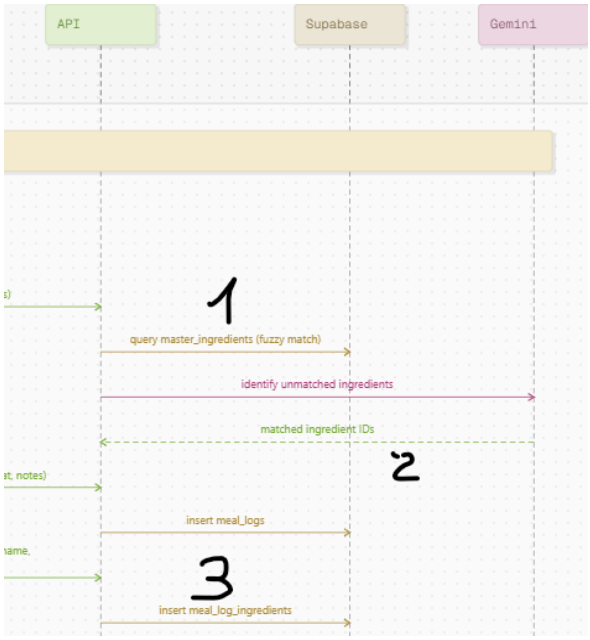


Figure 7 - Meal Logging Sequence Diagram

During daily use, the user logs their meals (Figure 7) and symptoms (Figure 8).

1) When logging a custom meal, the ingredient names entered by the user are first matched against the master ingredients table in Supabase using fuzzy matching.
2) Any ingredients that cannot be resolved through fuzzy matching are sent to Google Gemini AI, which identifies the closest match and returns a master ingredient ID.
3) The resolved ingredients are then saved to the meal_log_ingredients table alongside the meal log.

1) When the user logs their symptom, a range of data, including wellbeing score, pain score, bloating score, stress level and sleep quality, is saved in the UseSymptomLogForm state hook.
2) Once the user taps the log symptom button, the API layer connects to the Supabase database and pushes the value to the corresponding table
3) A symptom log can be directly linked to a meal log using a symptom_log_id, creating a connection between what the user ate and how they felt.
4) This accumulated data is what powers the correlation analysis.

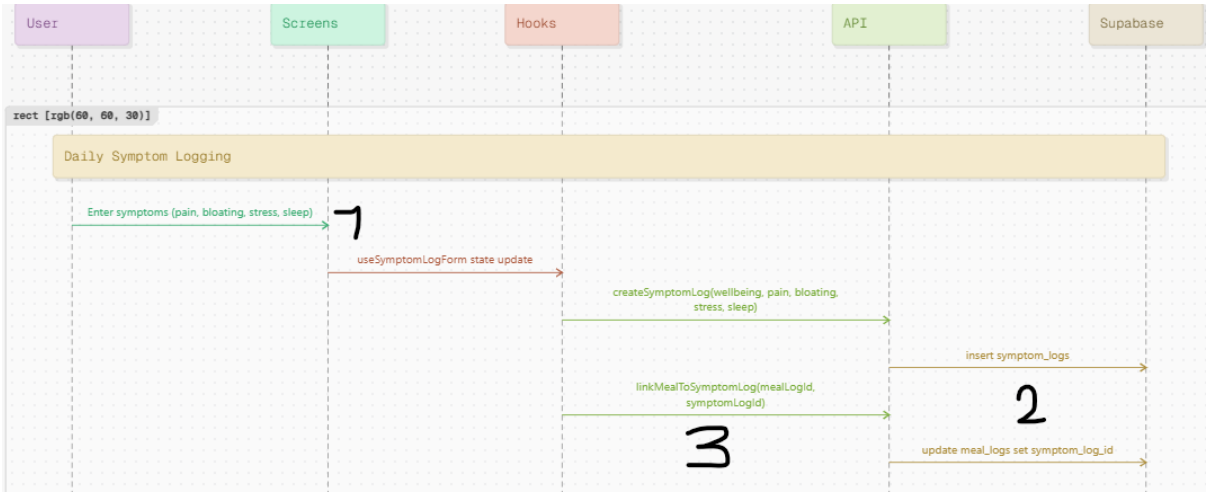


Figure 8 - Symptom Logging Sequence Diagram

1) When the user opens the Symptoms screen, useCorrelationAnalysis fetches all meal logs and symptom logs from the last 90 days.
2) For each meal, the analysis checks whether symptoms were logged within three-time windows after eating: 0 to 6 hours for immediate reactions, 6 to 24 hours for same-day reactions, and 24 to 48 hours for delayed reactions.
3) It then computes a delta score for each ingredient by comparing the average symptom severity on days the ingredient was eaten against days it was not. Based on this delta, each ingredient is classified as a likely trigger, a possible trigger or a weak signal
4) which is then displayed to the user through the CorrelationInsightsCard component.

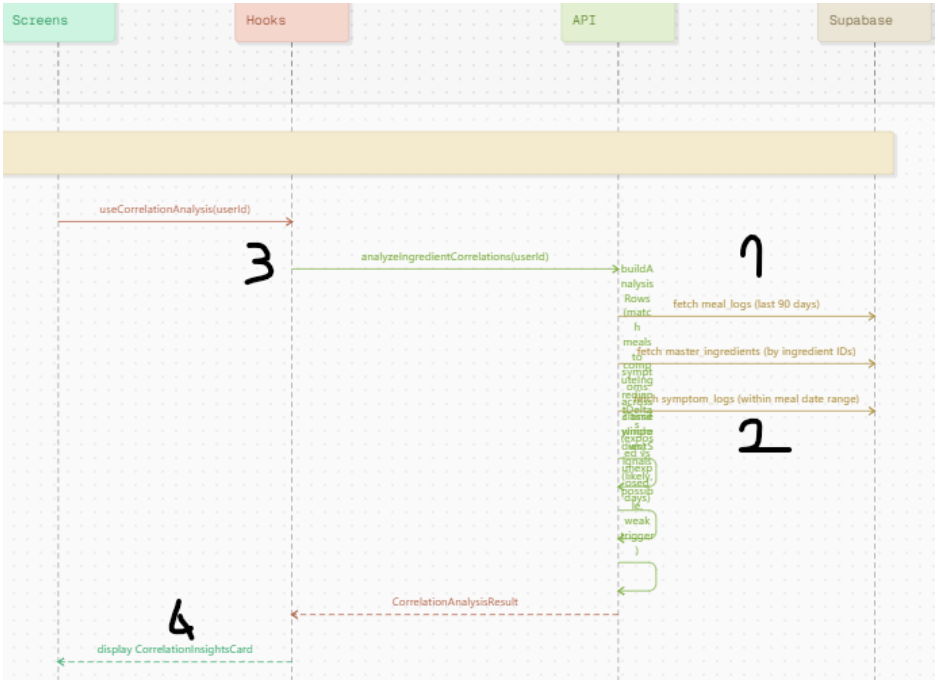


Figure 9 - Correlation Analysis Sequence Diagram

4.2 Interface Design

The following section outlines the interface design of Symprove, covering the user flow, wireframes, design decisions, and style guide.

4.2.1 User Flow

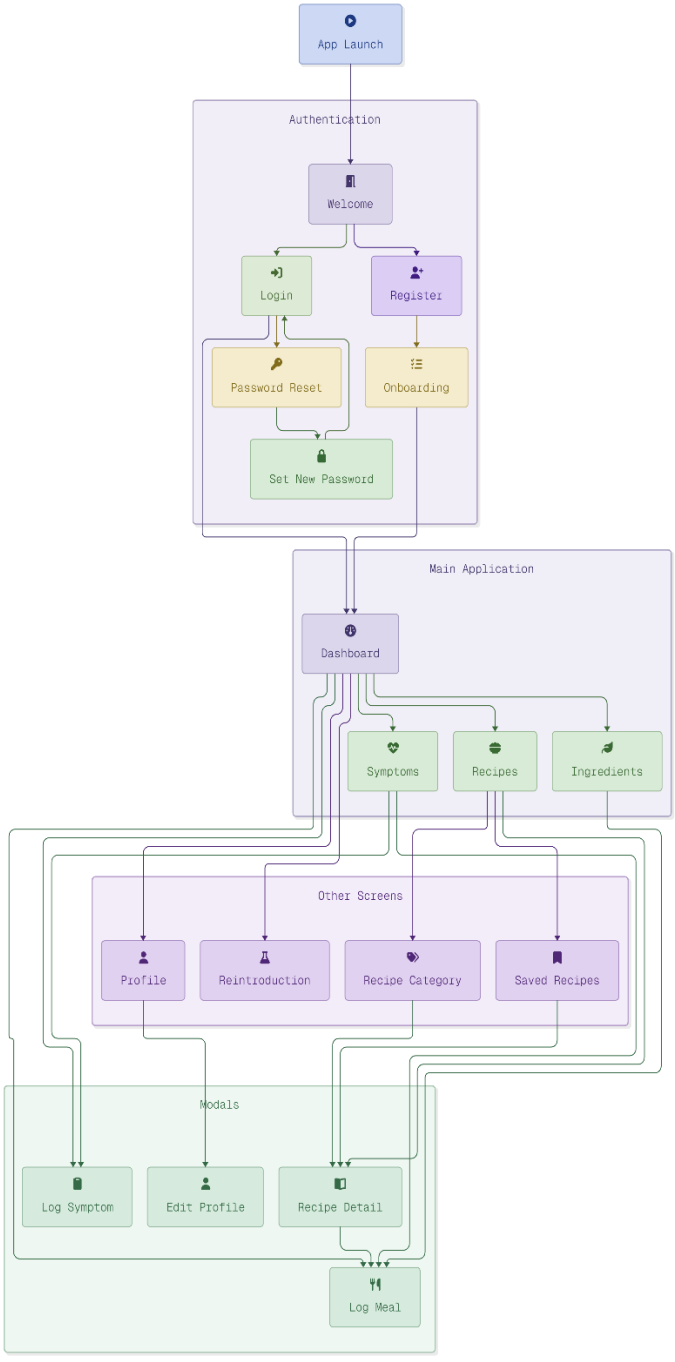


Figure 10 - User flow diagram of Symprove

Figure 10 shows the user flow of Symprove, illustrating how a user navigates through the application from launch to completing core tasks. The flow is divided into four areas:

1) Authentication (/app/(auth))
2) The main tabbed application (/app/(tabs))
3) Other Screens. (/app)
4) Modals. (/src/components)

When the app launches, the user is presented with a welcome screen where they can choose to log in or register. New users are directed through the onboarding process before reaching the dashboard.
Returning users are taken directly to the dashboard after logging in.
A password reset flow is also available from the login screen.

Once authenticated, the dashboard acts as the central hub of the application, providing access to all four tabs as well as the profile and reintroduction screens.
The meal and symptom logging modals are accessible directly from the dashboard as well as from the symptoms and ingredients tabs, reflecting the design decision to make logging available from any point in the application.
The recipes tab allows users to browse by category, view recipe details and access their saved recipes.
Saved recipes are also accessible from the user's profile as well as the edit profile modal for update and delete methods.

4.2.2 Wireframes/Mockups

The wireframes for Symprove were created in Figma prior to development and served as a behavioural reference throughout the build process, focusing on how the application would navigate and respond to user actions rather than defining the final visual design. The full interactive prototype can be viewed at the following link, which walks through the user flow with an app-like experience: Figma Prototype.

Several elements from the original wireframes carried through into the final application. The registration, login and onboarding flow remained largely consistent with the initial design.

- The profile screen, accessible from the dashboard, retained its original layout
- The core structure of the dashboard itself, which centres around:
    - a visual analytics graph
    - an activity list
    - meal and symptom logging action buttons.

  Several features were added during development that were not present in the original wireframes.
- The FODMAP phase protocol card was introduced above the visual graph on the dashboard to give users immediate visibility of their current diet phase and progress.
- An ingredient index screen featuring a searchable list of 400+ ingredients from the database, accessible from the tab bar.
- The recipes section expanded significantly, with the addition of:
    - Recipe detail modal
    - Category-based recipe screen
    - Saved recipes screen
    - Quick filter chips beneath the search bar for faster browsing.
- A FODMAP guidance screen was also added to provide users with educational content about the diet alongside the details for their progress.

4.2.3 Design Decisions

The overall visual direction for Symprove was inspired by minimalist UI designs found on Dribbble. The design board can be viewed at the following link: Design Board. The goal was to create an interface that felt like a wellness app and was easy to follow.

The decisions based on the goal are as follows:

4.2.3.1 Tab-Based Navigation

The application uses a tab-based navigation structure implemented using Expo Router's native tab component.

Native tabs automatically adapt to the conventions of each platform, ensuring the navigation feels familiar to users on both iOS and Android without requiring separate design decisions for each.

The downside of native tabs is that they do not function on desktop, which caused some difficulty when testing the design without a physical mobile device.

- **Modal Based Logging**
    - The logging experience was designed around modals rather than dedicated screens, keeping the user in context and reducing the number of navigation steps required to complete a log entry.

    - Since modals are reused across multiple screens for different purposes, a shared AppModal component was created as a base wrapper, keeping modal behaviour consistent and reducing the number of files needed.
    - The LogActionDock component renders two side-by-side buttons for meal and symptom logging and is placed at the bottom of multiple screens as a persistent fixture.
    - This decision was driven by the fact that daily logging is the core behaviour the app depends on, so making it accessible from any point in the application makes it more likely for the user to log daily.
- **Dashboard Layout**
    - The dashboard was designed to display the most relevant information to the user at a glance.
    - The FODMAP phase card sits at the top as it represents the user's current diet context, which affects every other feature in the app.
    - Below it, the visual analytics graph shows symptom and meal trends over time, followed by a recent activity list.
    - The dashboard was the most difficult screen to design as it needed to present several different types of data without feeling cluttered.
- **Recipes Screen**
    - The recipes screen includes a search bar with quick filter chips beneath it, allowing users to filter recipes by category or feature without opening the filter modal, looking through all the values available, and finally scrolling down to tap save.
    - A dedicated recipe category screen and a saved recipes screen were added to give users an option to browse through a list of those recipes.
    - Recipe details are presented in a modal rather than a separate screen, keeping the user within the recipe browsing context. Having it as a separate screen would cause the previous page to refresh, making the user lose the position of the list.
- **Ingredient Index**
    - The ingredient index screen was inspired by the iOS contacts list, presenting ingredients in an alphabetically sorted list with section headers for each letter of the alphabet.
    - This design pattern was chosen because it is immediately familiar to mobile users and provides a fast and predictable way to browse a large list of ingredients without needing a search query.
- **Tab Reduction**
    - The original design included six tabs, with the profile screen and reintroduction screen each occupying their own tab in the navigation bar.
    - This was reduced to four tabs during development, as having six tabs made the navigation bar feel cluttered and gave too much visual weight to screens that users would not visit frequently.
    - The profile and reintroduction screens were moved to be accessible from the dashboard instead, keeping the tab bar focused on other features of the app that do not fit into the dashboard.
- **Stylesheets**
    - Rather than creating a single separate file to hold all components and screen-specific styles, each screen and component file contains its own dedicated StyleSheet defined at the bottom of the file.

- This decision was made to keep styles co-located with the component they belong to, making it easier to find and update styles without navigating between files.
- Shared styles such as colours, spacing and typography are still centralised in globalStyles.ts, meaning only styles that are unique to a specific screen or component are defined locally.

4.2.4 Style Guidelines

The visual style of Symprove is defined in a centralised globalStyles.ts file, which exports a shared set of styles used consistently across every screen and component in the application. This ensures that no hardcoded style values appear in individual screen files, keeping the visual design consistent and easy to update.

- **Colours**
  - The primary colour is a teal (#0E7490) used for calls to action, active states and links. It has three variants: a light variant (#D5EAF1), a dark variant (#0C4A6E) and a very soft tint (#ECFEFF), each used for different levels of emphasis throughout the interface.
  - The secondary colour is a neutral slate (#334155) with a light variant (#B0BAC0) and a dark variant (#1E293B), used across borders, dividers and muted surfaces throughout the application.
  - A warm coral accent (#F4A261) is used sparingly for highlights and badges, supported by a light variant (#FBD8C9) and a warm tint (#FFF7ED) for subtle background fills.
  - Two additional accent colours are defined:
    - a cool accent (#0891B2) for informational elements
    - a warm amber (#D97706) used sparingly for symptom icons and warnings.
  - The application background is a warm off-white (#FAFAF7) rather than pure white, making it easier on the eyes. Card backgrounds use pure white (#FFFFFF) to contrast against it. A muted shade (#F5F6F8) is used for icon button backgrounds and subtle row highlights.
  - Semantic colours are defined for success (#15803D), warning (#B45309) and error (#B91C1C), each with a corresponding light tint, border colour and readable text colour variant to support consistent status indicators throughout the app.
  - Each FODMAP diet phase has its own tinted background and border colour. The elimination phase uses a soft teal tint (#ECFEFF) with a cyan border (#A5F3FC). The reintroduction phase uses a warm orange tint (#FFF7ED) with a peach border (#FED7AA). The personalisation phase uses a soft green tint (#F0FDF4) with a green border (#BBF7D0). These colours give the user a subtle visual indicator of their current phase throughout the application.
- **Typography**
  - Two typefaces are used throughout the application.
    - Outfit is used for all headings in regular, semi-bold and bold weights.
    - Plus Jakarta Sans is used for all body text in regular, semi-bold and bold weights.
  - The font size scale is as follows:
    - h1 at 30px
    - h2 at 24px

        - h3 at 19px
        - body text at 16px
        - small text at 14px
        - captions at 12px
        - button labels at 16px.
    - Line heights are defined as:
        - tight at 1.2 for headings
        - normal at 1.5 for body text
        - relaxed at 1.8 for modal text
    - Named text styles such as h1, h2, body, bodySmall, label, caption, button and modal are exported from globalStyles.ts and used directly in components rather than defining font sizes inline.
- **Spacing**
    - A fixed spacing scale is used throughout the application to ensure consistent padding and margins.
    - The scale is defined as:
        - xxs at 2px
        - xs at 4px
        - sm at 8px
        - md at 16px
        - lg at 24px
        - xl at 32px
        - xxl at 48px.
    - The standard screen padding is set to 24px, and card padding is set to 16px.
- **Layout and Border Radius**
    - Border radius values follow a named scale:
        - xs at 4px
        - sm at 8px
        - md at 12px
        - lg at 16px
        - xl at 20px
        - full at 9999px, which is used for fully rounded pill-shaped buttons and icon containers.
    - Icon sizes follow a similar-named scale from xs at 12px up to xl at 32px.
- **Shadows**
    - Three shadow levels are defined for use on cards and elevated elements.
    - The small shadow uses:
        - 2px vertical offset with 5% opacity
        - medium shadow uses a 4px offset with 8% opacity
        - large shadow uses an 8px offset with 12% opacity
    - Each shadow also defines an elevation value for Android compatibility.
- **Reusable Components**
    - A set of reusable component styles is defined in globalStyles.ts and shared across the application.
    - Primary buttons use the teal colour with fully rounded corners and a small shadow.
    - Secondary buttons use a white background with a light border and no shadow.

    - Input fields use a white background with a medium border radius and consistent padding.
    - Badge styles are defined for success, warning and error states.
    - A divider style, ghost button style and icon button style are also defined for use across multiple screens.
- **Containers**
    - A set of reusable container styles is defined in globalStyles.ts to standardise the layout structure across all screens and components.
    - The screen container applies a consistent horizontal padding and fixes the width to the device screen width, retrieved at runtime using React Native's Dimensions API via Dimensions.get("window"). This ensures the layout always fills the available screen width regardless of the device being used.
    - The screenSafe container is a full flex container used alongside React Native's SafeAreaView, ensuring screen content is never obscured by device notches, status bars or home indicators.
    - The centred container is a full flex container that centres its children both horizontally and vertically, used for loading states and empty state screens.
    - The card container defines the standard card style used throughout the application, combining a white background, a light border, a large border radius, consistent padding and a small shadow.
    - The row and rowBetween containers are horizontal flex containers used to align items side by side. rowBetween adds space-between justification for layouts where items need to sit at opposite ends of a row.
    - The column container is a vertical flex container used where content needs to be stacked explicitly.
    - The screenModal container is used for full-screen modals, applying the app background colour, consistent horizontal padding and top spacing.
    - The screenContent container is used for the scrollable content area within screens, applying vertical padding and a consistent gap between child elements.

4.3 Process Design

4.3.1 Frameworks & Libraries

4.3.1.1 React Native and Expo:

provides a cross-platform foundation that targets both iOS and Android from a single codebase.

Expo was selected over a bare React Native setup for its managed workflow, which simplifies device builds and provides pre-built packages for native device APIs. This includes camera access via expo-image-picker, which is used in the MealLogModal.tsx feature to allow AI to identify ingredients without manual input.

TypeScript is used throughout the codebase to enforce type safety and reduce runtime errors.

4.3.1.2 Expo Router:

Handles navigation using a file-based routing system.

This approach ties screen structure directly to the file system, making navigation predictable and easier to maintain as the project grows.

Tab-based navigation uses Expo's native tab component, which automatically renders platform-appropriate UI on iOS and Android.

4.3.1.3 expo-linking:

Used in useAuthActions and AuthContext for deep linking.

It generates a redirect URL for password reset, sends it to the user's email, and handles the incoming link when users tap the reset link to return to the app.

4.3.1.4 Supabase:

Relational structure: the app's data is highly relational. PostgreSQL enforces referential integrity across these linked records, which a document-based store such as MongoDB does not.

Single managed service: Supabase provides a PostgreSQL database, authentication, file storage, and a built-in email provider within a single platform, reducing backend configuration overhead.

Row Level Security: Security policies are enforced at the database level, ensuring users can only access their own data, a key requirement for an application handling sensitive personal health data in compliance with GDPR.

4.3.1.5 Sentry Error Tracking:

A third-party error monitoring service that records errors and crashes in real time to an online dashboard. It was chosen over alternatives because:

1) Expo's built-in crash reporting provides limited detail.
2) Supabase only captures server-side database and authentication errors.

Sentry fills this gap by monitoring the entire application. As shown in Figure 11, Sentry is initialised in the root layout with session replay enabled at a 100% sample rate, with maskAllText, maskAllImages, and maskAllVectors set to true to ensure no user data is captured in replay recordings. sendDefaultPii is set to false to prevent personally identifiable information from being transmitted.

```
Sentry.init({
  dsn: process.env.EXPO_PUBLIC_SENTRY_DSN,
  sendDefaultPii: false,
  // Enable Logs
  enableLogs: true,
  // Configure Session Replay
  replaysSessionSampleRate: 1.0,
  replaysOnErrorSampleRate: 1,
  integrations: [
    Sentry.mobileReplayIntegration({
      maskAllText: true,
      maskAllImages: true,
      maskAllVectors: true,
    }),
    Sentry.feedbackIntegration(),
  ],
  spotlight: true,
});
```

Figure 11 - Sentry initialisation configuration in the root layout file, showing session replay and privacy masking settings

As shown in Figure 12, the root layout wraps the entire application in a Sentry.ErrorBoundary, which renders a custom ErrorFallbackScreen component (Figure 14) on any unhandled exception, preventing the application from crashing silently and giving the user a recoverable state.

```
return (
  <Sentry.ErrorBoundary fallback={({ resetError }) => <ErrorFallbackScreen resetError={resetError} />}>
  <AuthProvider>
    <Stack screenOptions={{ headerShown: false }}>
      <Stack.Screen name="index" />
      <Stack.Screen name="(auth)" />
      <Stack.Screen name="(tabs)" />
    </Stack>
  </AuthProvider>
  </Sentry.ErrorBoundary>
);
```

Figure 12 - Sentry ErrorBoundary wrapping the application root with a custom fallback screen.

The following error-handling approach is applied consistently across the codebase:

1) All errors are tagged with feature and function identifiers, as shown in Figure 13, for example, an error in deriveOutcomeFromSymptomLogs is tagged feature: "reintroduction", making it immediately locatable in the dashboard without manual investigation

```
if (error) {
  Sentry.captureException(error, {
    tags: {
      feature: "reintroduction",
      function: "deriveOutcomeFromSymptomLogs",
    },
  });
  return "inconclusive";
```

Figure 13 - Example of a tagged Sentry error capture in deriveOutcomeFromSymptomLogs, showing feature and function identifiers.

2) Authentication functions throw errors so the calling screen can display a relevant message to the user
3) Data-fetching functions return null or an empty array on failure, preventing crashes from propagating through the UI, while Sentry captures the underlying error silently in the background

```tsx
export default function ErrorFallbackScreen({ resetError }: Props) {
  return (
    <View style={styles.container}>
      <Text style={styles.title}>Something went wrong</Text>
      <Text style={styles.body}>
        An unexpected error occurred. Please try again.
      </Text>
      <TouchableOpacity style={styles.button} onPress={resetError}>
        <Text style={styles.buttonText}>Try again</Text>
      </TouchableOpacity>
    </View>
  );
}
```

Figure 14 - ErrorFallbackScreen component rendered by the boundary on an unhandled exception

4.3.1.6 Fuse.js:

Fuzzy search across both ingredient and recipe lookup.

Fuzzy search allows the app to return relevant results even when a user's input does not exactly match a name in the database, for example, matching "tomatos" to "tomato" or "chick pea" to "chickpea".

The search is configured with a threshold of 0.35, meaning only reasonably close matches are returned, and fields are weighted to prioritise ingredient name over aliases, ensuring the most relevant results appear first.

The full ingredient list is fetched from Supabase once and cached in memory using module-level variables in **ingredientMatcher.ts**, alongside a pre-built Fuse index, exact name map, and alias map. Subsequent searches reuse these cached structures, avoiding repeated database calls on every keystroke.

4.3.1.7 Google Generative AI (Gemini):

Offers a free tier api key with a total of 250 requests per day.

Implementations:

4) **Ingredient name fallback:** When a user searches for an ingredient that does not closely match anything in the database, the query is passed to Gemini to identify the most likely ingredient and return a FODMAP classification
5) **Camera scanning:** When a meal is captured, the captured image is sent to Gemini, which identifies the ingredients present, maps them against the database, and suggests the name of the meal.
6) **Ingredient suggestions:** If the user does not want to use the camera feature, the meal name inserted, e.g. "Chicken Caesar salad", gets passed through Gemini, and suggests ingredients present without the need for manual lookup and insertion

4.3.1.8 Puppeteer + Cheerio for Data Scraping:

Puppeteer was chosen because the recipe source rendered content dynamically using JavaScript, meaning a simple HTTP request would return an incomplete page without the recipe data. Puppeteer launches a real browser instance that can execute JavaScript, scroll the page, and simulate clicks, allowing all recipe URLs to be collected automatically across paginated content.

Cheerio was used alongside Puppeteer to parse the raw HTML returned from each recipe page. Once Puppeteer had loaded a page, Cheerio extracted the relevant fields: recipe name, ingredients, instructions, and images, using CSS selectors, in a similar way to how jQuery works in the browser.

4.3.1.9 react-native-chart-kit:

Used to render the symptom trend line chart on the dashboard and symptoms screen.

The chart displays two datasets simultaneously, average symptom severity and average well-being score, plotted as bezier line charts over time.

The chart width is calculated dynamically at runtime using React Native's Dimensions API to ensure it fills the available screen width correctly across different device sizes.

The chart supports two time frame views, weekly and monthly, toggled via chip buttons beneath the chart.

4.3.1.10 Jest:

Used as the testing framework for unit and integration tests across the application.

Tests are co-located within __tests__ folders alongside the code they are testing and can be run using the test, test:watch and test:coverage scripts defined in package.json.

Further details on the testing approach and results can be found in Section 6

4.3.2 Algorithms

4.3.2.1 Scraping Pipeline

The recipe database was populated using a three-stage data pipeline built with Puppeteer and Cheerio, maintained in a separate scripts/ directory. The pipeline was a one-time development tool and does not run as part of the live application.

Stage 1 — URL Collection

7) Puppeteer launches a headless browser and navigates to the recipe index page on A Little Bit Yummy.
8) Because the site renders content dynamically with JavaScript, a basic HTTP request would return an incomplete page.
9) Puppeteer loads the full page and uses Cheerio to extract all recipe URLs matching the /recipe/en-us/ path pattern.
10) It then clicks the pagination next button and repeats this process until no further pages exist.
11) All collected URLs are deduplicated and saved to recipe_urls.json.
12) A 2.5-second delay between page loads was applied to avoid overloading the server.

Stage 2 — Recipe Detail Scraping

- Each URL from Stage 1 is visited in sequence.
- Cheerio parses the loaded HTML to extract structured recipe data, including title, image, preparation and cook times, serving size, ingredient sections with amounts, step-by-step method, nutritional information, dietary feature badges, and FODMAP tips.
- Ingredient sections are parsed by locating each h4 heading within the ingredients container and collecting the amount and name fields from the sibling elements that follow.

- A custom user agent string identifying the bot as a non-commercial academic research tool was applied to all requests.

Stage 3 — Supabase Upsert

- The scraped data from Stage 2 is read from recipes.json and mapped to the recipes table schema.
- Each recipe is upserted using source_url as the unique conflict key, meaning the pipeline can be rerun safely without creating duplicate entries.
- Equipment items are deduplicated before insertion to fix a scraping artefact where certain items were captured twice.

4.3.2.2 Correlation Analysis Algorithm

The correlation analysis is the core algorithm of Symprove and runs entirely on the client side without any server-side processing.

It begins by fetching the user's meal logs from the last 90 days, up to a limit of 500 entries. The ingredient IDs from those meal logs are collected and used to fetch the corresponding master ingredient records from Supabase. Symptom logs are then fetched for the same date range.

For each meal log and each ingredient associated with it, the algorithm checks whether any symptom logs were recorded within three defined time windows after the meal was eaten: Window A covers 0 to 6 hours for immediate reactions, Window B covers 6 to 24 hours for same-day reactions, and Window C covers 24 to 48 hours for delayed reactions. Window B is used as the primary analysis window.

An outcome score is computed for each matching symptom log as the average of the pain score and bloating score. Rows are marked as inconclusive if the user's stress level was 7 or above and sleep quality was 4 or below on that day, as these are recognised confounding factors that could skew the results independently of food intake.

To avoid double-counting multiple logs on the same day, scores are aggregated per ingredient per day before computing the delta. The delta is the difference between the mean outcome score on days the ingredient was eaten and the mean outcome score on days it was not eaten.

A minimum of three exposed days and three unexposed days is required before an ingredient is considered to have sufficient data for a meaningful comparison. Ingredients meeting this threshold are then classified into three signal tiers. A delta of 2.5 or above with no inconclusive days and at least three exposed days is classified as a likely trigger. A delta of 1.5 or above with up to one inconclusive day is classified as a possible trigger. Anything below these thresholds is classified as a weak signal or given no label.

4.3.3 Event Handling

Debouncing is applied to the ingredient search in useIngredients to avoid triggering a search on every keystroke. A 300 millisecond delay is applied after the user stops typing before the query is passed to the Fuse.js index. This prevents unnecessary processing during fast input and avoids calling the ingredient matcher on intermediate characters.

A similar debounce pattern is applied in MealLogModal when the user types a meal name. After a short delay, the meal name is passed to Gemini to suggest likely ingredients, ensuring the API is only called once the user has paused rather than on every character change.

The useCorrelationAnalysis hook uses an AbortController to cancel any in-flight Supabase requests if the user navigates away from the symptoms screen before the analysis completes. The abort signal is passed through to every query in the analysis, preventing stale results from being applied to the state after the component has unmounted.

4.3.4 Error Handling

Error handling in Symprove follows a consistent strategy across the API layer based on the context in which the error occurs.

Authentication functions such as signUp and signInWithPassword throw errors on failure, allowing the calling hook to catch the error and display a relevant message directly to the user on the screen.

Data fetching functions such as fetchMealLogs and fetchSymptomLogs return an empty array or null on failure rather than throwing. This prevents errors from propagating through the UI and causing crashes, while Sentry captures the underlying exception silently in the background.

All Gemini AI requests are wrapped in a 30-second timeout using Promise. race. If the request does not resolve within this window, it is rejected with a timeout error, ensuring the application does not hang indefinitely waiting for a response.

User input passed to Gemini prompts is sanitised before being included, removing newline and tab characters and capping the length at 200 characters to prevent prompt injection through user-entered text.

All errors captured by Sentry are tagged with a feature identifier and a function name, for example, feature: "correlation_analysis", function: "buildAnalysisRows_meals", making it straightforward to locate the source of an error in the Sentry dashboard without manual investigation.

4.3.5 Use of Concurrency

Promise.all is used across the application wherever multiple independent asynchronous operations are needed before the app can continue. Rather than running these sequentially, they are fired simultaneously, and the application waits for all of them to resolve before proceeding. For example, when a user session is loaded in AuthContext, the user profile and allergens are fetched at the same time rather than one after the other, reducing the overall loading time on app initialisation.

Promise.race is used across the application wherever a timeout needs to be enforced on an asynchronous operation. Without it, if a Gemini request was made and the API was slow to respond or failed silently, the application would wait indefinitely for a response that may never arrive, leaving the user with a frozen interface and no way to recover without closing the app. Promise.race solves this by racing the Gemini request against a 30-second timeout, meaning if the API does not respond in time, the timeout rejects the promise first, and the application handles the failure gracefully rather than hanging.

The batch ingredient matching function in ingredientMatcher.ts applies a two-stage approach to reduce unnecessary API calls. All ingredients are first matched locally in parallel using the cached Fuse index. Only the ingredients that could not be resolved locally are collected and sent to Gemini in a single batched request, rather than making one API call per unresolved ingredient. This significantly reduces the number of Gemini requests when logging a meal with multiple ingredients.

The AbortController used in useCorrelationAnalysis allows the multiple sequential Supabase queries within the analysis to be cancelled as a group if the user navigates away mid-fetch, preventing any in-flight requests from writing stale data to state after the component has unmounted.

4.4 Database Design

Symprove uses a PostgreSQL database hosted on Supabase. The full ERD can be viewed at the following link: Full Entity-Relational Diagram. The database is structured across 12 tables grouped into four logical domains:

1) **Reference data:**
   - Comprises three tables that are shared across all users and are not user-owned.
   - Figure 15 shows the master_ingredients table, containing the list of known food ingredients and their FODMAP classifications.
   - Each row stores the ingredient name, category, overall FODMAP level (low or high), four numeric columns for individual FODMAP subgroups (oligosaccharides, fructose, polyols, and lactose), allergens and aliases, which are alternative naming conventions for an ingredient.
   - A value of null assigned to the FODMAP subgroup rows indicates no data, 0 means not present, 1 is moderate, and 2 is high, a scale that allows the correlation analysis to make subgroup-level assessments rather than just a binary low/high classification.
   - Figure 16 shows the recipes table holding all the scraped recipes.
   - Rather than normalising every recipe field into separate columns, complex nested structures such as ingredient sections, method steps, tips, and nutrition data are stored as JSONB.
   - This is discussed further under design rationale.
   - Figure 17 shows the challenge_foods table, listing the recommended test foods for each of the six FODMAP subgroups used during reintroduction, along with portion guidance and allergen flags.

Figure 15 - master_ingredients data table

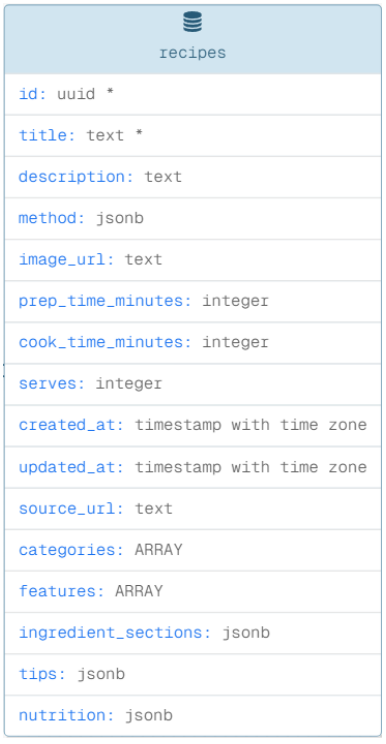


Figure 16 - recipes data table

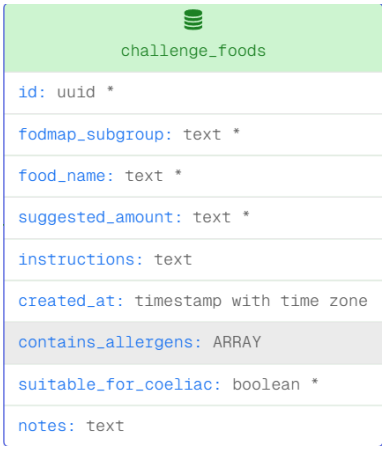


Figure 17 - challenge_foods data table

2) **User data** :

- Covers tables that store personal preferences and profile settings.
- Figure 18 shows the profiles table extending Supabase's built-in auth.
    - The auth table is used for storing user email and password; its primary key is a foreign key into the authentication system, meaning no separate user creation step is needed.
    - The profile table stores the user's username, IBS type, current FODMAP phase, and the date that phase began.
- Figure 19 shows the user_allergens table, which records the allergen categories a user has selected during onboarding or updated through the profile screen.
    - These are used to filter out unsuitable challenge foods during reintroduction, e.g. replacing bread with mango for users with coeliac or wheat allergies.
- Figure 20 shows the user_favourite_recipes table, a junction table linking users to saved recipes.
- Figure 21 shows the user_trigger_foods table, which stores the output of the correlation analysis; each row links a user to a specific master_ingredient and classifies it as a trigger or a safe food based on observed symptom patterns.

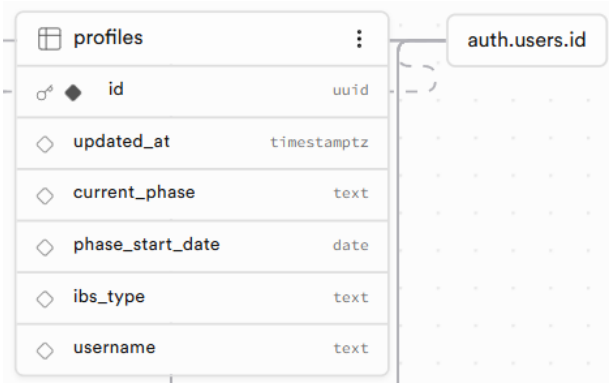


Figure 18 - Profiles data table extending auth.user.id

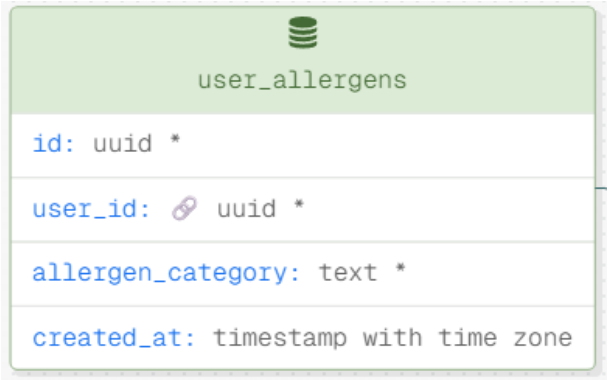


Figure 19 - user_allergens data table.

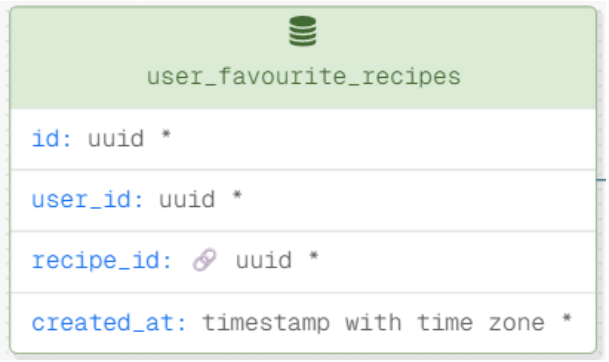


Figure 20 - Favourite recipes data table

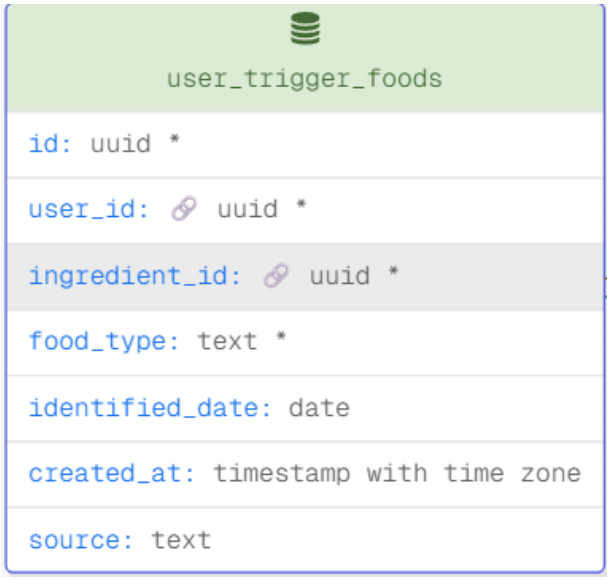


Figure 21 - User trigger food data table

3) **Daily logs:**
   - The core data tables that the app writes to most frequently.
   - Figure 22 shows the symptom_logs table, which records a timestamped health entry per day.
     - The only required field is well-being_score (1–10), which gives every log a usable signal for correlation analysis, even if the user does not complete the optional fields.
     - All other fields: pain score, pain location, bloating score, stool frequency, stool consistency, stress level, sleep quality, and several boolean flags, are nullable.
     - This allows for quicker logging: users may not experience every symptom every day, and forcing all fields would reduce logging compliance.
   - Figure 23 shows the meal_logs table, which records a single meal entry.
     - It supports two modes of entry:
       1. recipe-linked entry (where recipe_id is set and meal_name is null).
       2. Custom entry (where meal_name is set and recipe_id is null).
     - Meals can optionally be linked to a symptom_log via symptom_log_id, which is used when a user logs a meal in the context of reviewing a symptom entry. If no link is made, the meal exists as a standalone record.
   - Figure 24 shows the meal_log_ingredients table, a junction table that stores the individual ingredients for custom meals.
     - Each row preserves the user's original text as raw_name, and optionally stores a master_ingredient_id if the ingredient was matched to the reference database by the fuzzy or AI matching pipeline. Keeping the raw name independent of the match means no user input is lost if a match fails or is later corrected.

Figure 22 - Logged symptoms data table

Figure 23 - Logged meals data table

Figure 24 - Junction table, linking ingredients to logged meals

4) **Reintroduction tracking:**
   - Manages the six-week FODMAP reintroduction process.
   - Figure 25 shows the reintroduction_protocols table, which stores one record per user representing their overall plan, with a status of active, paused, or completed.
   - Figure 26 shows the reintroduction_challenges table, which stores one row per FODMAP subgroup challenge within that plan.
     - It tracks how many challenge days and washout days have been completed, transitions through a fixed status sequence (pending -> active -> washout -> completed), and stores the outcome (tolerated, partial, not tolerated, or inconclusive).
     - The outcome_overridden flag distinguishes between system-derived outcomes calculated from symptom logs and manual overrides entered by the user.

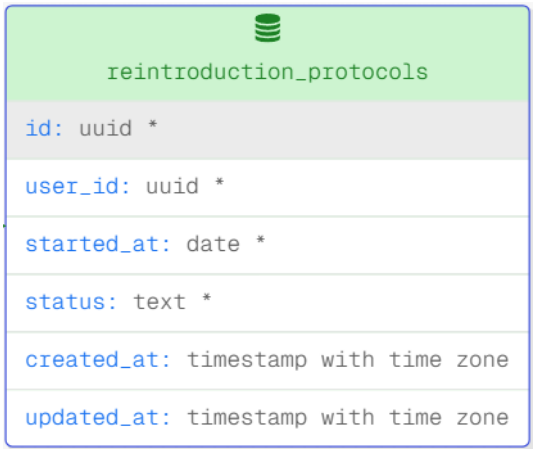


Figure 25 - Reintroduction protocol data table

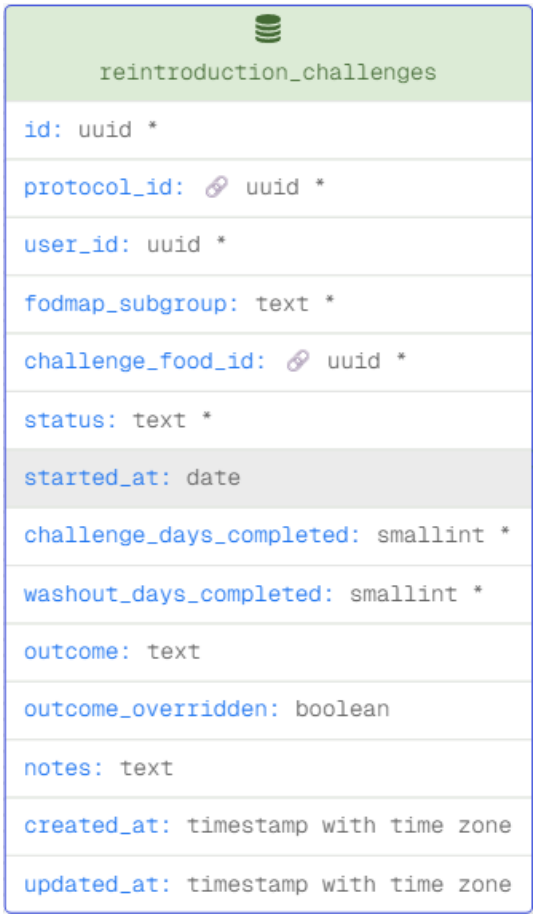


Figure 26 - Reintroduction challenged data table

4.4.1 Use of SQL

PostgreSQL was chosen over a document-based NoSQL database because the data in Symprove is inherently relational, meaning it consists of multiple distinct entities that are meaningfully connected (IBM).

Meal logs reference master ingredients, reintroduction challenges reference both protocols and challenge foods, and the correlation analysis joins meal logs, meal log ingredients, and symptom logs across time windows. Enforcing these relationships with foreign keys at the database level prevents orphaned records and ensures data integrity.

Supabase's Row Level Security (RLS) policies are applied to all user-owned tables so that the database itself rejects any attempt to read or write another user's data, even if an API call were constructed incorrectly.

4.4.2 Use of JSONB

JSONB is a PostgreSQL data type that stores JSON data in a decomposed binary format, as opposed to plain JSON, which stores the raw text string (geeksforgeeks, 2025). The binary representation allows PostgreSQL to index and query the contents of a JSONB field efficiently, whereas plain JSON must be parsed on every read (geeksforgeeks, 2025).

In Symprove, JSONB is used for the method, ingredient_sections, tips, and nutrition columns in the recipes table. Each of these fields holds structured but variable data; for example, ingredient_sections groups ingredients under named headings, and nutrition stores a varying set of nutrient key-value pairs.

Normalising this into additional tables would require multiple joins for every recipe read and add schema complexity with no practical benefit, since the application always reads the full recipe in one go, JSONB allows the full recipe structure to be stored and retrieved as a single document while still benefiting from PostgreSQL's transactional guarantees and indexing capabilities.

4.4.3 Design rationale summary

The schema was designed to support three key requirements:

1) Data completeness: every meaningful piece of health data a user might log has a dedicated column with an appropriate type and nullable constraint, rather than being packed into a free-text notes field.
2) Flexibility in meal logging: the dual-mode design of meal_logs (recipe-linked or custom) and the raw_name plus optional master_ingredient_id pattern in meal_log_ingredients means the app can handle both structured and unstructured input without losing data.
3) Correlation readiness: because symptom logs and meal log ingredients both reference master_ingredients by ID, the correlation analysis can join across them directly without requiring additional normalisation at query time.

5 Implementation

5.1 Development Process & Methodology

The application was built incrementally, following a layered approach to feature development. Authentication was implemented first, establishing the foundation for all user-specific functionality. From there, each feature was built end-to-end, beginning with the database table, then the API functions, then the custom hook, and finally the screen and components, before moving on to the next feature. This ensured that each part of the application was functional and integrated before new functionality was added on top of it. Sprint tasks were tracked in Notion, with priorities and deadlines set at the start of each sprint to guide the order of development; this is discussed further in Section 7.

The structure of the codebase evolved significantly throughout development. The src folder initially contained only config, services, and styles directories. As the application grew in complexity, additional directories were introduced to maintain separation of concerns: api, hooks, types, utils, constants, components, and scripts were each added as the need arose. The scripts folder was introduced following the decision to integrate AI-assisted ingredient matching, with the original intention of using it to match the existing recipe dataset against the master ingredient table. Due to complications encountered during this process, discussed further in Section 5.2, the folder was repurposed to house the web scraping pipeline, the database seeding scripts, and a user seeding script used for testing. The scripts folder maintains its own package.json and dependency tree, keeping it independent from the main application since it serves development purposes only and is not part of the production build.

The database schema also underwent considerable revision throughout development. As features were added, changed, or replaced, the schema was updated to reflect the current state of the application rather than the original design. These changes and their implications are discussed in Section 5.2.

Testing was carried out in two stages. During active development, each feature was tested manually through the application interface on a physical device after implementation. Unit tests using Jest were written after the majority of the application had been built, as the frequency of schema and feature changes during development made writing tests in parallel impractical.

Bug handling followed a similar pattern. Errors that surfaced during development were addressed immediately, with TypeScript's static type checking catching type-related issues at compile time and Sentry flagging runtime errors during device testing. Bugs that were not immediately visible were identified later in the development process through the Claude code review skill, which was used to review completed sections of the codebase and surface issues that had not been caught during initial development.

The overall development process was more adaptive than planned. Several features that were initially scoped were redesigned or replaced mid-development, and accommodating those changes required revisiting parts of the codebase that had already been built. While this created additional work, it also produced a more considered final implementation, and the decisions made as a result of those changes are reflected throughout the design and implementation discussed in this report.

5.2 Challenges & Technical Solutions

5.2.1 FODMAP Data Architecture and Recipe System Redesign

One of the most significant and far-reaching challenges of the project was establishing a reliable data architecture for FODMAP ingredient classification and recipe management. This challenge went through several iterations before reaching its final form.

The original plan for the recipe system was to source recipe data from the Spoonacular API, which provided FODMAP classifications, allergen information, and standard recipe details within a single response. Nutritional information at the ingredient level was to be sourced separately from the USDA FoodData Central API. The USDA integration was scoped early in development, but was subsequently abandoned due to the complexity of accurately mapping nutritional data across 400 or more ingredients and 200 or more recipes. The Spoonacular API was later also ruled out as a viable option. Although a free tier was available, it imposed a severely limited daily request allowance that was insufficient for populating a recipe database of the intended size. Caching the API responses was considered as a workaround; however, Spoonacular's terms of service restrict cached data to a maximum of 24 hours, after which it must be deleted, making persistent local storage of the data impractical.

Prior to the decision to scrape recipe data, several open-source datasets from Kaggle were evaluated as potential sources. The first dataset considered was the Food.com Recipes and Interactions dataset, which contained over 230,000 recipes crawled from the Food.com platform, including recipe names, ingredients, steps, tags, and nutritional information. However, the dataset did not include recipe images, which were considered essential for the user interface, and it was ruled out on this basis. A second dataset, the Food Ingredients and Recipes Dataset sourced from the Epicurious website, was then evaluated. This dataset contained approximately 13,500 recipes with ingredients, instructions, and associated image names. The CSV file was downloaded and cleaned using a Python script to match the Supabase table structure. However, the images were stored separately in a compressed folder, requiring each image file to be downloaded and managed locally. Storing and serving that volume of image files was considered impractical. Additionally, neither dataset contained FODMAP classifications, meaning the original problem of determining recipe safety for IBS users remained unresolved. Following a review meeting with the project supervisor, the approach of using a pre-existing general recipe dataset was abandoned in favour of scraping data directly from a dedicated recipe source.

A web scraping pipeline was then built using Python and Beautiful Soup, scraping recipes from RecipeTin Eats. The scraping was successful in retrieving recipe data, but the problem of FODMAP classification remained unresolved. The original intent was not to restrict the recipe database to low-FODMAP recipes only, but instead to match each recipe's ingredients against the master_ingredients table to identify which FODMAP subgroups were present. This would allow users to consume a variety of recipes while understanding which subgroups each dish contained, particularly useful during the reintroduction phase. A fuzzy matching algorithm was built to link recipe ingredients to entries in the master_ingredients table. This worked to a degree for common ingredients but failed consistently on compound ingredients such as "curry paste" and branded products that had no direct equivalent in the master ingredients table. With a significant proportion of ingredients unmatched, it was not possible to determine with confidence whether a recipe was safe for a user on a specific FODMAP subgroup test. Following a second meeting with the project supervisor and co-reader, the approach of using AI-assisted categorisation to handle unmatched ingredients was introduced.

The initial recipe schema used a recipe_ingredients junction table to link recipes to entries in the master_ingredients table, with the intention of calculating each recipe's FODMAP status by matching its ingredients against the classification table. An AI-assisted matching pipeline using LangChain was explored to improve matching accuracy, which required altering the recipe_ingredients schema from a normalised relational structure with foreign key constraints to a JSONB-based structure storing ingredients as an embedded array. However, during implementation, it became clear that matching free-text recipe ingredients against a structured ingredient table could not be done with sufficient accuracy. Ingredients listed in recipes rarely correspond exactly to entries in a classification table, and treating unmatched ingredients as safe would produce misleading results for users managing a medical dietary protocol.

This led to a complete redesign of the recipe system. The decision was made to restrict the recipe database exclusively to pre-verified low-FODMAP recipes, removing the need to calculate the FODMAP status algorithmically. The recipe_ingredients junction table was dropped entirely, and ingredient sections were stored directly on the recipes table as a JSONB column. A new scraping pipeline was built using Node.js with Puppeteer and Cheerio, targeting A Little Bit Yummy, a dedicated low-FODMAP recipe website. This replaced both the Python-based pipeline and the Kaggle dataset approach. As the site included detailed nutritional information for each recipe, nutritional data was reintroduced to the final schema via a nutrition JSONB column on the recipes table, resolving the earlier problem without requiring a third-party API. A total of 251 low-FODMAP recipes were scraped and seeded into the database.

Additionally, the master_ingredients table was repurposed to support the ingredient index feature and user-logged meal ingredient tracking, where ingredients are looked up and classified individually rather than as part of a recipe calculation. A new tab was added to the application, providing users with a searchable index of ingredients and their FODMAP classifications.

5.2.2 Allergen Matching

The initial allergen implementation included a dedicated allergen table with two junction tables: one linking users to their specific allergens, and another linking those allergens to the corresponding ingredient entries in the master_ingredients table. This allowed allergen data to be stored relationally at the ingredient level, with user allergens tied directly to specific database entries rather than broad categories.

Populating this structure required matching allergen data against the master_ingredients table. A Kaggle allergen dataset was used as the data source, and a multi-step Python script was written to perform the matching. The pipeline worked through the following stages: exact name matching, singular and plural variant matching, first-word matching for compound ingredients, category-based matching for obvious groupings, and a final manual review of remaining unmatched entries. The script executed successfully and produced matches across the ingredient table.

However, the allergen dataset itself proved to be the core problem. It contained rare and highly specific allergens with no practical value, where allergen data serves two purposes: filtering recipes based on a user's known allergens, and flagging considerations when selecting challenge foods during the reintroduction protocol. The relational junction table structure also added unnecessary complexity for what was ultimately a filtering concern.

The approach was redesigned. The junction tables were removed and replaced with a single user_allergens table storing a user_id and an allergen_category text value per row, and an allergens array column was added directly to the master_ingredients table. The allergen set was reduced to common allergens relevant to IBS and FODMAP management, including dairy, wheat, gluten, eggs, nuts, and soy. A new, simpler script assigned allergens primarily by food category, where an ingredient categorised as dairy would receive the dairy allergen directly, with a small number of remaining ingredients matched by name. This reduced complexity significantly while producing a more clinically relevant result.

5.2.3 Symptom Logging Redesign

The original symptom logging implementation used a generic symptom_entries table with a flat symptom type and a uniform 1–10 severity scale applied across all symptom types. This design was clinically inaccurate for several reasons (Further details on IBS symptoms discussed in Section 2.1). Binary symptoms such as urgency and mucus in stool do not have meaningful intensity; they are either present or absent, and applying a numerical scale to them produces data that cannot be meaningfully interpreted. Similarly, applying a severity scale to symptoms such as incomplete evacuation or diarrhoea does not reflect clinical IBS assessment practice. A further problem was that the original design did not account for external confounding factors such as stress and sleep quality, both of which can independently trigger IBS flare-ups through the gut-brain axis. Without capturing these factors, it would be impossible to distinguish food-triggered symptoms from those caused by stress or poor sleep, undermining the reliability of any correlation analysis performed on the data.

The symptom_entries table was dropped and replaced with structured columns added directly to the symptom_logs table. Each column was defined to reflect the clinical nature of the symptom it captured:

Column	Type	Description
pain_score	1–10	Abdominal pain intensity is only logged when pain is present
pain_location	text	Upper, lower, left, right, or general
pain_after_eating	boolean	Whether pain was triggered by eating
pain_relieved_by_bm	boolean	Whether pain was relieved by a bowel movement
stool_frequency	integer	Number of bowel movements that day
stool_consistency	text	Hard, normal, or loose — simplified from the Bristol Stool Scale
incomplete_evacuation	boolean	Whether the evacuation felt incomplete
bloating_score	1–10	Bloating intensity, only logged when present
visible_swelling	boolean	Whether visible abdominal swelling was present
urgency	boolean	Whether urgency was experienced
mucus_in_stool	boolean	Whether mucus was present in the stool
stress_level	1–10	Collected on every log as a confounding factor
sleep_quality	1–10	Collected on every log as a confounding factor

The logging interface was rebuilt to reflect this structure. Symptoms are presented across six clinical sections: Overall Wellbeing, External Factors, Abdominal Pain, Bowel Habits, Bloating, and Other Symptoms. Pain and bloating fields are hidden by default and only revealed when the user indicates that those symptoms are present, preventing users from logging zero scores for symptoms they did

not experience. Only the overall well-being score is required; all other fields are optional. The interface also adapts based on the user's IBS type as recorded in their profile: IBS-C users see constipation-relevant fields highlighted, and IBS-D users see diarrhoea-relevant fields highlighted. This is a display-only adaptation; the same data is collected regardless of IBS type.

5.2.4 Reintroduction Protocol Schema

The initial approach to challenge food selection for the reintroduction protocol was to source test foods directly from the master_ingredients table. This approach had two problems. First, serving size information had been removed from the master_ingredients table earlier in development, as it added complexity without sufficient benefit across the ingredient index as a whole. However, quantity is essential for the reintroduction protocol. Clinical FODMAP reintroduction requires testing a specific amount of a food on each challenge day, and without a suggested serving size, the protocol could not give users meaningful guidance. Second, the standard test foods for certain FODMAP subgroups were not appropriate for all users. For example, bread is a commonly recommended food for testing fructan tolerance, and dairy milk for lactose tolerance. However, these options are unsuitable for users with wheat allergies, coeliac disease, or dairy allergies.

A separate challenge_foods table was created to solve both problems. Unlike the master_ingredients table, each row in challenge_foods includes a suggested_amount column specifying the recommended test quantity. Multiple food options are provided per FODMAP subgroup so that users who cannot consume a standard test food are offered an appropriate alternative. Each row includes a contains_allergens text array and a suitable_for_coeliac boolean, derived from the same allergen classification logic applied to the broader ingredients system. When a user begins a challenge, the application filters available options based on their allergen profile and coeliac status stored in their profile, ensuring that only appropriate foods are presented.

5.2.5 Styling Architecture

The initial styling approach used two separate files: a theme file containing the application's primary design tokens, colours and typography, and an index file containing all styles, including both global styles and component-specific styles. As the application grew, this approach became difficult to manage. All component styles were co-located in a single file regardless of which component they belonged to, creating naming conflicts between similarly named style properties, making it harder to locate styles relevant to a specific component, and leaving orphaned styles in the file when components were removed.

The approach was restructured to a single globalStyles.ts file containing only truly global styles, shared layout values, typography, and colour tokens. In contrast, component-specific styles were moved into their respective component files. This meant that when a component was modified or removed, its styles were updated or deleted alongside it rather than remaining as dead code in a centralised file. It also eliminated naming conflicts, reduced the number of files to manage, and made the codebase more consistent, as a developer working on a component could find all of its styles in the same file without needing to navigate elsewhere.

A separate issue was encountered with typography during this process. The original approach downloaded fonts as TrueType font files and imported them manually. For reasons that could not be determined, the fonts failed to import and apply correctly in certain files across the project. Rather than spending time diagnosing the issue, an alternative solution was found: the expo-google-fonts packages, which allow any Google Font to be installed as an npm package and loaded via Expo's useFonts hook in the root layout file. This made the fonts available globally throughout the

application from a single import point. The fonts used in the final application, Outfit and Plus Jakarta Sans, were installed and loaded this way.

5.2.6 Development Environment Issues

Several smaller issues were encountered during development that were resolved without significant rework.

During early development, the physical iOS device could not connect to the Expo development server on the local network. As a temporary workaround, the --tunnel flag was used with the Expo CLI to route the connection through an external server. The root cause was subsequently identified as the Windows firewall blocking Node.js from communicating over the local network. Changing the Node.js firewall settings to allow access on both private and public networks resolved the issue, and the tunnel workaround was no longer needed.

Screen width inconsistencies were encountered across different components, where elements would not render correctly or extend beyond the visible area. These were resolved using React Native's Dimensions API to retrieve the device screen width at runtime, and SafeAreaView to ensure content was correctly inset within the safe area boundaries of the device.

Sentry's initial setup did not function as expected when first integrated into the project. This was resolved by reviewing the official Sentry React Native documentation and correcting the initialisation configuration in the root layout file.

5.3 Development Environment & Tools

Category	Tool	Purpose
IDE	Visual Studio Code	Primary code editor
Runtime	Node.js, ts-node	Application build process, seeding scripts, Jest test runner
Framework	Expo CLI, Expo Go	Development server and physical device testing
Language	TypeScript	Static type checking across the full codebase
Version Control	GitHub	Source control using a feature branch workflow
Wireframing	Figma	Wireframes
Diagramming & Design	Miro, toDiagram	Visual design board, use case diagram, personas, architecture diagrams, Entity-relationship diagrams and Mermaid flowcharts.
Project Management	Notion	Sprint planning, task tracking, development notes and change log
Error Monitoring	Sentry	Runtime error tracking and crash reporting
Testing	Jest	Unit and component testing

Prototyping	Python	Initial web scraping prototypes before migrating to Node.js
Data Research	Kaggle	Evaluated as a source for recipes and allergens datasets before a custom dataset was developed

Emulator-based testing was also attempted using Android Studio and BrowserStack; however, the configuration overhead for both tools required substantial setup time that was not justified, given the stage of development. Physical device testing was used exclusively afterwards.

Node.js served as the underlying runtime for the Expo CLI and development server. It was also used to execute the data seeding scripts (seed-recipes.ts, seed-user.ts) that populated the Supabase database with recipe and ingredient data, and to seed a test user account for testing the application with a realistically populated interface. These scripts were run directly from the terminal using ts-node, which compiles and executes TypeScript without requiring a separate build step. Node.js also served as the runtime environment for the Jest test suite, executed via the npm test command.

TypeScript was used throughout the codebase in place of JavaScript. The reason for this decision was that JavaScript is dynamically typed, meaning variable types are only checked at runtime; an incorrectly typed value passed to a function will only produce an error when that code is executed. TypeScript adds a static type system on top of JavaScript, allowing type errors to be caught at compile time before the application runs (Imoh). In a project of this complexity, with multiple data models, defining explicit types for each entity ensured that data was handled consistently across the codebase. For example, passing a string where a UUID was expected, or accessing a property that did not exist on a response object, would be flagged immediately in the IDE rather than surfacing as a bug during testing.

Python was also used during early development to prototype the web scraping pipeline before the decision was made to migrate to a Node.js-based solution using Puppeteer and Cheerio. This transition is discussed further in Section 5.2.

Kaggle was also evaluated as a potential source for FODMAP ingredient data; however, no suitable dataset was found, and a custom dataset was developed instead.

Git was used for version control with GitHub as the remote repository, following a feature branch workflow where each sprint's work was developed on the development branch before being merged into the main branch.

Claude (Anthropic) was used as an AI-assisted development tool throughout the implementation process. It was used to generate complex code sections, including the web scraping scripts, to debug TypeScript type errors, and to draft the initial database schema. Claude was also used to help create the global stylesheet and implement the visual design based on the design board discussed in Section 4.2, translating design decisions into consistent styles across the application. Claude's built-in skill tools were also used during development, specifically the code review and accessibility review skills available through the Engineering and Design plugins, to assess code quality and identify potential issues. Beyond the codebase itself, Claude was used to assist with creating user testing and interview scripts, planning and structuring the project sprints, and creating and reviewing design diagrams. All AI-generated output was reviewed and validated before being incorporated into the project.

6 Testing and Evaluation

6.1 Test Plan and Executed Tests

Testing for Symprove was carried out across three approaches:

1) Automated unit testing: conducted using Jest and the React Native Testing Library, targeting the API layer, service layer, custom hooks, screen components, and utility functions. The acceptance criterion for unit tests was that all tests must pass with no failures before a feature was considered complete. Coverage was measured using Jest's built-in coverage reporter.
2) Manual interface testing was carried out throughout development on a physical iOS device via Expo Go. After each feature was implemented, the interface was tested manually to verify that the feature functioned as expected, that navigation behaved correctly, and that no visible errors were present.
3) Usability testing was conducted with external participants to evaluate the application's ease of use and identify usability issues not captured by automated tests. This is discussed further in Section 6.2.

The testing strategy was designed to cover both the correctness of individual functions and the usability of the application as experienced by real users.

Error monitoring via Sentry provided an additional passive layer of testing during development, capturing runtime errors and exceptions that occurred during manual and usability testing sessions. This is discussed further in Section 6.3.

6.2 Evidence of Different Types of Testing

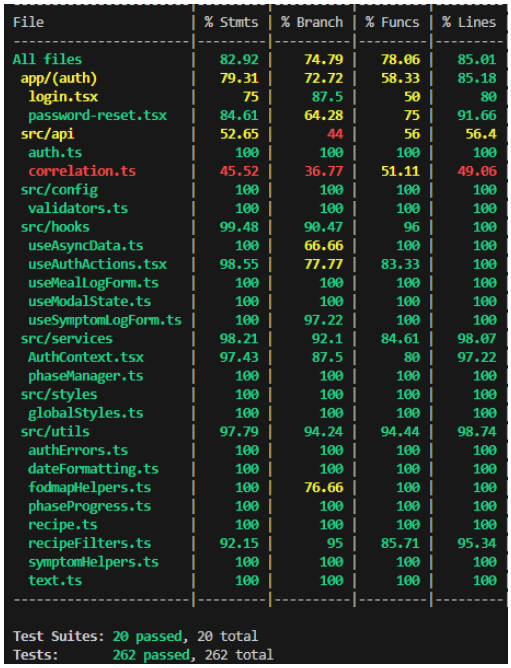
6.2.1 Unit Tests

Unit tests were written using Jest and the React Native Testing Library across 20 test suites, covering the API layer, service layer, custom hooks, screen components, and utility functions. All 262 tests pass with no failures. Tests were executed via the npm test command, with coverage reported using the --coverage flag.

The overall coverage results are as follows:

Metric	Coverage
Statements	82.92%
Branches	74.79%
Functions	78.06%
Lines	85.01%

Coverage is highest across the utility layer, service layer, and custom hooks, where the majority of files achieve full or near-full statement coverage. The most notable gap is src/api/correlation.ts, which has 45.52% statement coverage and 36.77% branch coverage. The correlation analysis function contains a large number of conditional branches handling edge cases in the data, including insufficient exposed or unexposed days, inconclusive day filtering, and multi-window time analysis, many of which require specific combinations of realistic meal and symptom log data to trigger. The core algorithm paths are covered, but full branch coverage of the complete analysis function was not achieved within the scope of this project. Coverage by file is broken down in Figure 27:



File	% Stmts	% Branch	% Funcs	% Lines
All files	82.92	74.79	78.06	85.01
app/(auth)	79.31	72.72	58.33	85.18
login.tsx	75	87.5	50	80
password-reset.tsx	84.61	64.28	75	91.66
src/api	52.65	44	56	56.4
auth.ts	100	100	100	100
correlation.ts	45.52	36.77	51.11	49.06
src/config	100	100	100	100
validators.ts	100	100	100	100
src/hooks	99.48	90.47	96	100
useAsyncData.ts	100	66.66	100	100
useAuthActions.tsx	98.55	77.77	83.33	100
useMealLogForm.ts	100	100	100	100
useModalState.ts	100	100	100	100
useSymptomLogForm.ts	100	97.22	100	100
src/services	98.21	92.1	84.61	98.07
AuthContext.tsx	97.43	87.5	80	97.22
phaseManager.ts	100	100	100	100
src/styles	100	100	100	100
globalStyles.ts	100	100	100	100
src/utils	97.79	94.24	94.44	98.74
authErrors.ts	100	100	100	100
dateFormatting.ts	100	100	100	100
fodmapHelpers.ts	100	76.66	100	100
phaseProgress.ts	100	100	100	100
recipe.ts	100	100	100	100
recipeFilters.ts	92.15	95	85.71	95.34
symptomHelpers.ts	100	100	100	100
text.ts	100	100	100	100

Test Suites: 20 passed, 20 total
Tests: 262 passed, 262 total

Figure 27 - Unit test coverage results

6.2.2 Usability and User Testing

Usability testing was conducted with three participants using a think-aloud protocol on a physical iOS device. Participants were not given guidance during the sessions and were asked to verbalise their thoughts as they completed each task. Following the structured tasks, participants were invited to explore the application freely and provide open feedback on what they liked, what they found confusing, and what they would want added or changed.

Participants were asked to complete the following tasks without assistance:

1) Create an account and complete the onboarding process
2) Update information on their profile
3) Log a symptom entry, then log a meal using the camera scan feature, log a second meal using only the meal name, and log a third meal by selecting from the recipes database
4) Navigate to the recipe page, find a recipe, and save it to favourites
5) Freely explore the remaining features of the application

The usability sessions identified a range of bugs and usability issues. The following were addressed following testing:

Issue	Action Taken
The keyboard could not be dismissed on the sign-in screen	Fixed
Screen quivering when typing in the auth fields	Fixed
No show/hide password toggle on sign-in	Added
Onboarding data not reflected in profile view	Fixed
Edit profile used a separate button instead of inline editing	Redesigned to inline field selection
The password change process lacked a confirmation step	Improved with confirmation flow
Email not pre-filled on the "forgot password" screen	Fixed

Modal close button obscured by overlapping elements	Fixed
The notes field in the symptom log modal is not visible while typing	Fixed
Reintroduction protocol unclear without guidance.	Added explanatory detail and instructions
No option to undo a completed reintroduction day	Undo functionality added
The Recent Activity section on the dashboard is not expandable	Expand button added
Search text clipped and dropped from the frame in recipes	Fixed
The recipe detail view did not show ingredients by default	Defaulted to ingredients
No option to copy all ingredients in recipe detail	Copy all button added
Calorie detail insufficient: user was unsure if the calories were per serving or for the entire dish	Added “per serving” text
Delay on reset button response	Fixed
Page refreshed when navigating back from recipe list	Fixed
Delay when toggling recipe favourite	Fixed via optimistic UI update
Show more / show less state not preserved in recipes	Fixed

The following issues identified during usability testing were not resolved and are outstanding in the final application:

Issue	Notes
No email keyboard suggestion (@gmail, .com) on auth inputs	Minor UX improvement, not critical
"Urgency" label unclear to participants	Terminology retained from clinical IBS practice
Recipe images not shown in meal log modal	Not implemented
Recipe images cannot be expanded in the detail view	Not implemented
The "Show more” button position should align with where the text ends	Not implemented
No confirmation feedback when favouriting a recipe	No haptic or toast notification implemented
Filter chip placement in recipe search is not ideal	Not repositioned
A–Z index does not support hold-and-scroll	Not implemented
Symptom chart legend misaligned with colour indicators	Not resolved
Symptom severity is not reflected in the background colour	Not implemented

6.3 Error Handling

6.3.1 Runtime Error Monitoring

As discussed in Section 4.3.1.5, Sentry was integrated as the primary runtime error monitoring tool throughout the application. During development and usability testing, Sentry captured runtime errors in real time as the application was used on a physical device, with each error tagged by feature and function to make it immediately locatable in the dashboard. This allowed errors encountered during development to be diagnosed and resolved without needing to reproduce them manually. The ErrorBoundary at the root level ensured that any unhandled exception presented the user with a recoverable fallback screen rather than a silent crash, maintaining application stability throughout development and testing.

6.3.2 Compile-Time Error Prevention

TypeScript's static type system served as the first line of defence against errors during development. Type mismatches, missing properties, and incorrect function arguments were flagged by the compiler and the IDE before the application was run, preventing a category of errors that would otherwise only surface at runtime.

In addition to TypeScript's static type checking, ESLint was configured with the eslint-config-expo ruleset to enforce code quality standards across the codebase, as seen in Figure 28 - ESLint configuration applying the Expo ruleset with React Native and TypeScript-specific rules. The rules applied include warnings for missing React hook dependencies (react-hooks/exhaustive-deps), unused variables (@typescript-eslint/no-unused-vars), explicit any types (@typescript-eslint/no-explicit-any), inline styles (react-native/no-inline-styles), and hardcoded colour literals (react-native/no-color-literals). Console statements are flagged as warnings, with console.warn and console.error permitted. These rules caught potential issues during development, such as missing hook dependencies that could cause stale state, before they surfaced as bugs at runtime.

```
const { defineConfig } = require("eslint/config");
const expoConfig = require("eslint-config-expo/flat");

module.exports = defineConfig([
  expoConfig,
  {
    ignores: ["dist/*"],
  },
  {
    rules: {
      "react-hooks/exhaustive-deps": "warn",
      "react-native/no-inline-styles": "warn",
      "react-native/split-platform-components": "warn",
      "react-native/no-color-literals": "warn",
      "react-native/no-single-element-style-arrays": "warn",
      "@typescript-eslint/no-unused-vars": "warn",
      "@typescript-eslint/no-explicit-any": "warn",
      "no-console": ["warn", { allow: ["warn", "error"] }],
    },
  },
]);
```

Figure 28 - ESLint configuration applying the Expo ruleset with React Native and TypeScript-specific rules.

6.3.3 Debugging Techniques

Debugging during development was carried out using a combination of the Expo development console, TypeScript compiler output, and Sentry error reports. For issues that could not be resolved through these alone, the Claude code review skill was used to audit completed sections of the codebase and identify issues not caught during manual testing. External resources, including the official documentation for Expo, Supabase, React Native, and Sentry, were consulted throughout development. Stack Overflow was used on occasion for specific error messages encountered during configuration and setup.

6.3.4 User-Facing Error Handling

Where errors are expected and recoverable, user-facing error messages are shown rather than allowing the application to fail silently. Authentication errors are mapped to plain-language messages using the authErrors.ts utility rather than exposing raw Supabase error strings to the user. As shown

in Figure 29, example error mapping in authErrors.ts, converting a raw network error into a user-facing message, each error condition is matched against the raw message string and returned as a readable alternative. For example, a network failure returns "Network error. Please check your connection and try again." rather than the underlying fetch exception.

```
// Network
if (message.includes("fetch") || message.includes("network")) {
  return "Network error. Please check your connection and try again.";
}
```

Figure 29 - Example error mapping in authErrors.ts, converting a raw network error into a user-facing message.

6.3.5 Unresolved Bugs

The unresolved usability issues documented in Section 6.2.2 represent the known outstanding bugs and missing features in the final application. No critical bugs defined as issues that prevent core functionality from operating were present in the final build. All unresolved issues are either minor UX improvements or non-critical feature additions.

6.4 Evaluation against Success Criteria and Objectives

The following evaluates the final application against each success criterion defined in Section 1.2.

Criterion	Outcome
Functional Completeness	Met
Clinical Accuracy	Met
User Experience	Partially met
Technical Performance	Partially met
Accessibility	Partially met
Data Security	Met
Code Quality	Met
Testing Coverage	Partially met

6.4.1 Functional Completeness: Met

All core features defined in this criterion were implemented in the final application. Users can register accounts, configure dietary restrictions and allergens during onboarding, browse and save low-FODMAP recipes, log meals and symptoms, and follow the guided reintroduction protocol across all six FODMAP subgroups. No Must Have requirements from the MoSCoW analysis in Section 3.2.2 were left unimplemented.

6.4.2 Clinical Accuracy: Met

The recipe database contains 200+ low-FODMAP recipes scraped from A Little Bit Yummy, a resource authored by an accredited practising dietitian, exceeding the minimum of 50 specified. The reintroduction protocol follows established clinical guidelines: six FODMAP subgroups are tested individually, each with a three-day challenge period and a three-day washout period between

challenges. Challenge foods are filtered by allergen profile and coeliac status to ensure clinical appropriateness for each user.

6.4.3 User Experience: Partially Met

Usability testing with three participants confirmed that core tasks: finding recipes, logging symptoms, and following reintroduction steps, could be completed without guidance, satisfying the core requirement. However, ten usability issues identified during testing remain unresolved in the final build, including unclear labelling, missing feedback on interactions, and interface elements that participants found difficult to locate. These are documented in Section 6.2.2. While none of these prevents core task completion, they represent gaps in the overall user experience quality.

6.4.4 Technical Performance: Partially Met

The application functions reliably on iOS and was tested extensively on a physical iOS device throughout development. Android compatibility was not verified, and emulator testing was attempted but abandoned due to setup complexity, as discussed in Section 5.3. A physical Android device was not available for testing. Database query performance was not formally measured, though no perceptible delays were observed during manual or usability testing under normal conditions. Screen transitions were smooth, and authentication operated without data loss during all testing sessions.

6.4.5 Accessibility: Partially Met

WCAG 2.1 Level AA compliance was targeted throughout development. An accessibility review was conducted using the Claude accessibility review skill, and the identified issues were addressed. However, formal screen reader compatibility testing was not conducted, and one usability issue related to colour contrast for symptom severity indicators remains unresolved. Full WCAG 2.1 Level AA compliance cannot be confirmed without formal audit tooling.

6.4.6 Data Security: Met

User health data is stored in Supabase with Row Level Security policies applied to all user-owned tables, ensuring that authenticated users can only access their own records. Authentication tokens are stored using expo-secure-store on native platforms, which uses hardware-backed, encrypted storage. All client-server communication occurs over HTTPS. Sentry is configured with sendDefaultPii: false and session replay masking enabled, ensuring no personally identifiable information is transmitted to the error monitoring service.

6.4.7 Code Quality: Met

The codebase follows React Native and Expo best practices throughout. A layered component architecture is maintained as documented in Section 4.1, with clear separation between routing, presentation, state management, data access, and configuration. Error handling is applied consistently across the codebase as documented in Section 6.3. ESLint is configured with the Expo ruleset and TypeScript-specific rules to enforce code quality standards. Inline comments are present throughout the codebase, particularly in complex logic such as the correlation analysis and ingredient matching pipeline.

6.4.8 Testing Coverage: Partially Met

Unit testing was conducted across 20 test suites with 262 passing tests and 82.92% statement coverage, covering the correlation logic, authentication functions, hooks, utilities, and screen

components. Usability testing was conducted with three participants, satisfying the specified range of two to three. However, formal integration testing for Supabase database operations was not implemented as a separate test suite; database interactions were validated through manual testing during development rather than automated integration tests.

7 Project Management

7.1 Methodology

Symprove was developed using an Agile methodology, structured across four sprints. Three sprints ran for four weeks each, with the final sprint running for three weeks. This approach allowed development priorities to be reassessed at the end of each sprint based on progress made and any challenges encountered, rather than committing to a fixed plan at the onset.

Sprint planning and progress tracking were managed in Notion. As shown in Figure 30, each sprint was set up as a dedicated section within a board view, with tasks assigned one of three statuses: Not Started, In Progress, or Done. Each sprint had a defined start date and deadline, a weekly checklist of tasks to be completed, and a dedicated challenges and solutions section for documenting any issues encountered during that sprint and how they were resolved. Figure 31 shows the timeline view, which provides a calendar-based overview of sprint deadlines across the full project duration. A separate notes tab, independent of the sprint boards, was used to document research findings throughout the project.

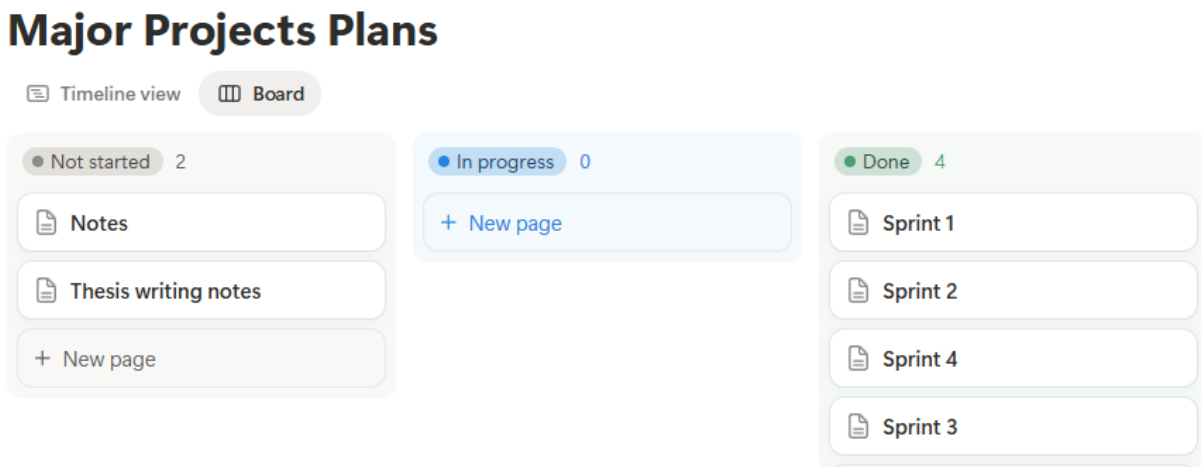


Figure 30 - Notion board view showing the four sprints organised by status: Not Started, In Progress, and Done.

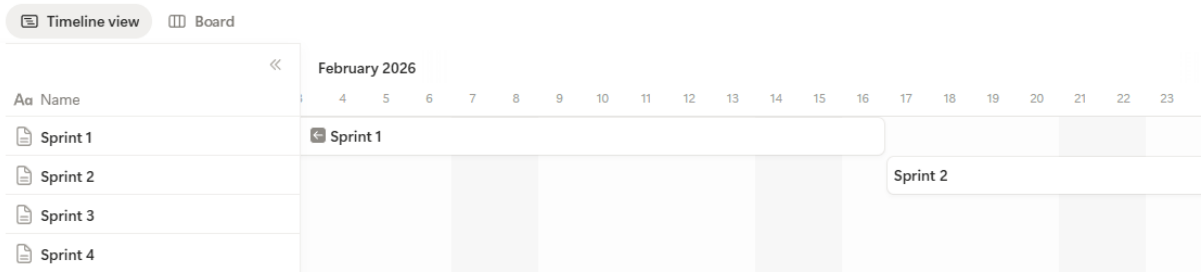


Figure 31 - Notion timeline view showing sprint scheduling across the project duration.

7.2 Sprint 1

Environment Setup, Authentication and Database Foundation

Duration: 20 January 2026 – 16 February 2026

7.2.1 Initial Goals

Sprint 1 focused on establishing the foundational infrastructure of the project. The primary objectives were to configure the development environment, set up the Supabase backend with authentication, design the initial database schema, implement authentication UI flows, build the user profile and onboarding system, and begin populating the recipe database.

7.2.2 Week-by-Week Breakdown

Week 1 covered environment setup: the React Native and Expo development environment was configured, the Supabase project was initialised, GitHub was set up with a branch strategy using main, development, and feature branches, a basic navigation shell was created, and initial recipe source research was begun.

Week 2 focused on authentication and database foundations: the initial database schema for profiles, ingredients, and recipes was designed, registration and login UI flows were implemented, SecureStore was integrated with Supabase authentication for secure token storage, and a Node.js CLI script was built to assist with consistent recipe data entry.

Week 3 covered profile and navigation: database relationships and foreign keys were implemented, the user profile creation and editing interface was developed, the dietary restrictions configuration system was built, the bottom tab navigation structure was established, and the UI component library and styling system were started. An allergens column was added to the master_ingredients table.

Week 4 completed the sprint: the onboarding flow was built, covering the welcome screen, account setup, and dietary profile configuration. The UI component library and styling system were finalised, and technical decisions and challenges were documented.

7.2.3 Final Outcomes

All planned deliverables for Sprint 1 were completed. A working development environment, configured Supabase backend, GitHub repository, authentication flows, initial database schema, user profile management, dietary restrictions configuration, and a complete onboarding flow were all delivered by the end of the sprint. The ERD produced at the end of Sprint 1 is shown in Figure 32.

Figure 32 - Entity-relationship diagram produced at the end of Sprint 1, showing the initial database schema.

7.2.4 Challenges

Several challenges were encountered during Sprint 1. A Windows firewall issue prevented the iOS device from connecting to the development server, which was resolved by updating Node.js firewall permissions. Sentry integration required additional configuration before functioning correctly. The USDA FoodData Central API was found to lack FODMAP classifications, requiring a rethink of the ingredient data strategy. Screen width inconsistencies were resolved using React Native's Dimensions API and SafeAreaView. The styling architecture was reconsidered and restructured from two stylesheets to a single global stylesheet with component-level styles. Allergen data population required a multi-step matching script due to naming discrepancies between the allergen dataset and the master_ingredients table. These challenges are discussed in detail in Section 5.2.

7.3 Sprint 2

Recipe System, Symptom Tracking and Data Visualisation

Duration: 17 February 2026 – 16 March 2026

7.3.1 Initial Goals

Sprint 2 aimed to build the core recipe browsing system with FODMAP classification, implement the symptom and meal logging system, and introduce data visualisation for symptom trends. The sprint also included a retrospective review of Sprint 1 and a supervisor meeting demo.

Week-by-Week Breakdown

Week 5 focused on the FODMAP classification algorithm: a web scraping script was written, a basic recipe browsing interface was implemented, ingredient matching logic for FODMAP classification was developed, and n8n was investigated as a potential AI-assisted scraping pipeline. A Sprint 1 end-to-end review and supervisor demo were also completed.

Week 6 built out the recipe browsing experience: recipe card components with images were built, filtering by FODMAP content, meal type, and dietary restrictions was implemented, recipe search functionality was created, and a recipe detail view displaying ingredients and instructions was developed.

Week 7 introduced symptom tracking: a symptom logging interface was designed with severity scale inputs, meal logging linked to symptom entries was implemented, data structures for symptom and meal storage were created, and basic data visualisation components were built.

Week 8 completed the sprint: React Native Chart Kit was integrated to implement line charts for symptom trends over time, a summary dashboard for symptom patterns was built, a Sprint 2 retrospective was conducted, and core features were tested with informal user feedback.

7.3.2 Final Outcomes

The recipe browsing system, symptom logging, meal logging, and basic data visualisation were all delivered by the end of Sprint 2. However, the ingredient matching approach required significant revision during the sprint. The initial plan to classify recipes by matching ingredients against the master_ingredients table could not be achieved with sufficient accuracy; unmatched ingredients, such as branded products and compound ingredients, could not be reliably classified. The approach was adjusted so that unmatched ingredients were treated as unknown rather than safe or unsafe, a 0–100 safety scale replaced the binary low/high classification, and users were advised to check packaging labels for allergens on unmatched ingredients. These decisions are discussed in detail in Section 5.2.

7.3.3 Challenges

The web scraping script did not initially account for timestamps and required image resizing logic to be added. The allergen data in the master_ingredients table were revised to use more common allergen categories. A fodmap_status column was added to the recipe table to track ingredient classification coverage. The most significant challenge was the fundamental limitation of ingredient matching accuracy, which led to a revised approach to how FODMAP status was communicated to users.

7.4 Sprint 3

Reintroduction Protocol, Correlation Analysis and Recipe System Redesign

Duration: 17 March 2026 – 16 April 2026

7.4.1 Initial Goals

Sprint 3 aimed to implement the FODMAP reintroduction protocol, build the correlation analysis system for identifying food-symptom patterns, and extend the recipe browsing interface with nutritional information. This sprint also carried forward unresolved architectural decisions from Sprint 2 regarding the recipe classification system.

7.4.2 Week-by-Week Breakdown

Week 9 focused on the reintroduction protocol foundation: the database structure was designed, the challenge foods table was built with isolated FODMAP subgroups, the phase tracking system was implemented, the reintroduction timeline and scheduling logic were developed, and the reintroduction interface and user flows were designed. Favourite recipe functionality was also added as a priority task.

Week 10 completed the reintroduction protocol implementation: the step-by-step challenge instructions interface was built, three-day challenge and three-day washout timing controls were implemented, reintroduction progress was linked to symptom logging, automated phase notifications were created, and a results tracking system was developed.

Week 11 addressed correlation analysis: the delta-based correlation algorithm was researched and implemented, pattern recognition for food-symptom relationships was developed, the trigger food identification system was built, and the correlation results dashboard was created and tested with sample data.

Week 12 completed the display of macronutrients and micronutrients in recipe details and API optimisation tasks.

7.4.3 Final Outcomes

Sprint 3 delivered the reintroduction protocol, correlation analysis system, and correlation results dashboard. However, the sprint also involved the most significant architectural changes of the entire project. The ingredient matching approach was abandoned in favour of restricting the recipe database to low-FODMAP recipes only. The recipe_ingredients junction table was dropped, ingredients were restructured as JSONB on the recipes table, the Python scraping pipeline was replaced with a Node.js pipeline using Puppeteer and Cheerio, and 251 low-FODMAP recipes were scraped from A Little Bit Yummy. A new ingredient index tab was added to the application. The symptom logging system was also completely redesigned to reflect clinical IBS practice, with the generic symptom_entries table replaced by structured columns on symptom_logs. These changes are discussed in detail in Section 5.2.

7.4.4 Challenges

Sprint 3 was the most change-intensive sprint of the project. LangChain was explored as an AI-assisted ingredient matching solution, requiring a schema change from a normalised junction table to a JSONB structure, before the approach was abandoned entirely in favour of low-FODMAP recipe data. The symptom logging system was identified as clinically inaccurate and was rebuilt from the ground up.

Three new tables were introduced to support the reintroduction protocol. The recipe scraping pipeline was rewritten in Node.js to replace the Python implementation. These challenges are discussed in detail in Section 5.2.

7.5 Sprint 3

Testing, Evaluation and Final Submission

Duration: 14 April 2026 – 1 May 2026

7.5.1 Initial Goals

Sprint 4 was the final sprint of the project, running for three weeks up to the submission deadline. The primary objectives were to complete unit and integration testing, conduct usability testing with external participants, implement refinements based on feedback, finalise all thesis documentation, and prepare the screencast and presentation deliverables.

7.5.2 Week-by-Week Breakdown

Week 13 focused on testing and bug fixes: unit tests were written across 20 test suites covering the correlation algorithm, authentication, hooks, utilities, and screen components, achieving 82.92% statement coverage with 262 passing tests. Usability issues identified during device testing were documented and fixed. Cross-platform testing was conducted on iOS; Android testing was attempted but could not be completed due to emulator setup constraints.

Week 14 covered user evaluation and refinement: usability testing was conducted with three participants using a think-aloud protocol on a physical iOS device, as documented in Section 6.2.2. Priority refinements identified during testing were implemented, resulting in over twenty fixes and improvements to the application.

Week 15 completed the final documentation and submission: all thesis chapters were written and compiled, code comments and README documentation were finalised, and all deliverables were prepared for submission.

7.5.3 Final Outcomes

Sprint 4 delivered a fully tested application with documented unit test results, completed usability testing, and a finalised thesis report. Formal integration testing against the live Supabase database and Android cross-platform testing were not completed within the sprint, as noted in Section 6.4. The accessibility review was conducted using the Claude accessibility review skill rather than formal WCAG audit tooling.

7.5.4 Challenges

The primary challenge of Sprint 4 was managing the scope of remaining work within a three-week window that covered testing, usability evaluation, bug fixing, and full thesis documentation simultaneously. Android testing was not achievable within the available time, as the emulator setup issues encountered in Sprint 1 were not revisited.

7.6 Tools

Three primary tools supported project management. Notion served as the central planning and tracking platform, used to manage sprint boards, task statuses, deadlines, and challenge documentation throughout all four sprints. GitHub provided version control using a feature branch

workflow, with each sprint's development carried out on a dedicated branch before being merged into the main branch on completion. Miro was used for visual planning. A full breakdown of all development tools used across the project is provided in Section 4.

8 Conclusion & Future Work

8.1 Summary and Reflection on Project Aim

This project aimed to develop a mobile application supporting individuals with IBS through the complete low-FODMAP dietary protocol, providing guided food reintroduction, recipe discovery, meal logging, and symptom tracking within a single cohesive system. The final application delivers on this aim. Users can create accounts, configure their FODMAP phase, dietary restrictions, and allergens, browse and save 251 verified low-FODMAP recipes, log meals and symptoms with structured clinical data, follow the guided six-subgroup reintroduction protocol, and view correlation analysis identifying potential food-symptom patterns. These features collectively constitute a complete digital support tool for IBS dietary management.

Reflecting on the individual objectives, the majority were met. The authentication and profile system, recipe database exceeding the 50–100 target, structured symptom logging capturing pain, bloating, stool type, stress, and sleep quality, the Supabase backend with Row Level Security, and the unit and usability testing requirements were all delivered. Two objectives were not met. The meal planning system was not implemented; it was deprioritised as the project scope evolved, and the depth of other features took precedence. The USDA FoodData Central API integration was abandoned following the discovery that the API does not provide FODMAP classifications, and the nutritional data goal was ultimately resolved by scraping recipe-level nutritional information directly from A Little Bit Yummy.

8.2 Limitations and Future Work

8.2.1 Limitations

The current application has several limitations. Android compatibility was not verified; the application was developed and tested exclusively on iOS, and Android behaviour cannot be guaranteed without dedicated testing on a physical device or a working emulator. Formal integration testing against the live Supabase database was not implemented, with database interactions validated through manual testing only. A formal WCAG 2.1 Level AA accessibility audit was not conducted, meaning full compliance cannot be confirmed. Ten usability issues identified during testing remain unresolved in the final build, as documented in Section 6.2.2.

8.2.2 Future Work

Several directions for future development are identified:

8.2.2.1 User interface improvements

The application's navigation does not always flow naturally between features, and several interface elements were identified as confusing by usability participants. A dedicated UX review and redesign pass would improve the overall coherence of the experience.

8.2.2.2 Meal planning system

A weekly meal planning feature was included in the original objectives but was not implemented. This would allow users to plan meals, with the planned meals automatically linked to symptom logs for more accurate correlation analysis.

8.2.2.3 Extended AI integration

The current AI implementation is limited to ingredient identification and meal name suggestions. Future development could introduce AI-powered features such as suggested ingredient substitutes for high-FODMAP items and personalised dietary insights based on logged symptom patterns.

8.2.2.4 Android support

The application should be tested and validated on Android devices to fulfil the cross-platform requirement stated in the success criteria.

8.2.2.5 Resolving outstanding usability issues

The ten unresolved issues from usability testing, including the A–Z ingredient index scroll behaviour, recipe image expansion, and favouriting feedback, represent a prioritised list of improvements for a future release.

8.2.2.6 Push notifications

Automated reminders for reintroduction challenge days and washout periods were planned but not implemented, and would improve protocol adherence for users actively completing the reintroduction phase.

8.3 Conclusion

Symprove was built to address a genuine issue: individuals managing IBS through the low-FODMAP protocol have limited access to a single tool that supports the full process from elimination through reintroduction to personalisation. The final application provides this. It is a functional, clinically informed, and technically considered mobile application that delivers the core features required for dietary IBS management.

The development process was more complex than initially anticipated, requiring multiple significant pivots in architecture and clinical design. These pivots, while disruptive, ultimately produced a more robust and accurate application than the original design would have yielded. The process reinforced the value of iterative development, domain research, and honest evaluation of what was and was not working at each stage.

The application is not without limitations, and several objectives remain as future work. However, the core aim: supporting individuals with IBS through the complete low-FODMAP protocol in a single cohesive system, was met.

9 References

Agrawal, H. (2025, Sep 2). *Implementing Fuzzy Search in React Native Apps Using Fuse.js.* Retrieved from Medium: https://medium.com/@harshitmadhav/implementing-fuzzy-search-in-react-native-apps-using-fuse-js-d33ed8710eba

baeldung. (2021, November 11). *Layered Architecture.* Retrieved from Baeldung: https://www.baeldung.com/cs/layered-architecture

Bosman, M. H., Weerts, Z. Z., Snijkers, J. T., Vork, L., Mujagic, Z., Masclee, A. A., . . . Keszthelyi, D. (2023). The Socioeconomic Impact of Irritable Bowel Syndrome: An Analysis of Direct and Indirect Health Care Costs. *Clinical Gastroenterology and Hepatology*, 2660-2669.

Cleveland Clinic. (2023, 11 16). *Irritable Bowel Syndrome (IBS)*. Retrieved from Cleveland Clinic: https://my.clevelandclinic.org/health/diseases/4342-irritable-bowel-syndrome-ibs

Cozma-Petrut, A., Loghin, F., Miere, D., & Dumitrascu, D. L. (2017). Diet in irritable bowel syndrome: What to recommend, not what to forbid to patients! *World Journal of Gastroenterology*, 3771-3783.

Cudny, A. (2026, March 20). *GDPR Consent Requirements for Health Data.* Retrieved from Momentum: https://www.themomentum.ai/blog/gdpr-consent-requirements-health-data#:~:text=GDPR%20classifies%20health%20data%20as,in%20consumer%20and%20clinical%20applications.

Dovetail Editorial Team. (2023, May 11). *What is correlation analysis?* Retrieved from Dovetail: https://dovetail.com/research/what-is-correlation-analysis/#:~:text=Correlation%20analysis%20is%20a%20staple,between%20two%20datasets%20and%20variables.

Dragon 1. (n.d.). *System Architecture.* Retrieved from Dragon 1: https://www.dragon1.com/resources/system-architecture#:~:text=The%20primary%20goal%20of%20system,Architecture%20Design%20Process%20and%20Deliverables

Expo. (n.d.). *Expo Docs.* Retrieved from Expo: https://docs.expo.dev/

Furqan Ahmad. (2024, Feb 10). *Building a Web Scraper with Puppeteer and Cheerio: A Step-by-Step Guide.* Retrieved from Medium: https://medium.com/@furqana405/building-a-web-scraper-with-puppeteer-and-cheerio-a-step-by-step-guide-e895c2ab7f52

Fuse.js. (n.d.). *Fuse.js Docs.* Retrieved from Fuse.js: https://www.fusejs.io/

geeksforgeeks. (2025, July 23). *What is JSONB in PostgreSQL?* Retrieved from geeksforgeeks: https://www.geeksforgeeks.org/postgresql/what-is-jsonb-in-postgresql/

Gemini API. (n.d.). *Gemini API docs.* Retrieved from Gemini API: https://ai.google.dev/gemini-api/docs

Gemini API. (n.d.). *Prompt design strategies.* Retrieved from Gemini API: https://ai.google.dev/gemini-api/docs/prompting-strategies

Guts UK! (2026). *Irritable Bowel Syndrome (IBS).* Retrieved from GutsUK!: https://gutscharity.org.uk/advice-and-information/conditions/irritable-bowel-syndrome/

IBM. (n.d.). *PostgreSQL vs MySQL: What's the Difference?* Retrieved from IBM: https://www.ibm.com/think/topics/postgresql-vs-mysql#:~:text=Performance%20and%20scalability%20%E2%80%94%20including%20geospatial,open%2Dsource%20database%20management%20technology.

IBM. (n.d.). *What is a relational database?* Retrieved from IBM: https://www.ibm.com/think/topics/relational-databases

Imoh, W. (n.d.). *TypeScript vs JavaScript: Which to Choose For Your Project.* Retrieved from Roadmaps: https://roadmap.sh/javascript/vs-typescript

James, E. (n.d.). *What is Correlation Analysis? A Definition and Explanation.* Retrieved from flexmr: https://blog.flexmr.net/correlation-analysis-definition-exploration#:~:text=Pearson%20Product%2DMoment%20Coefficient,1%20%E2%89%A4%20r%20%E2%89%A4%20+1.

Jan Kadlec. (2022, March 2021). *How To Automate Your Statistical Data Analysis.* Retrieved from Medium: https://medium.com/gooddata-developers/how-to-automate-your-statistical-data-analysis-852f1a463b95

Joe Osborne. (2024, April 7). *Intro to Web Scraping: Build Your First Scraper in 5 Minutes.* Retrieved from Medium: https://medium.com/@joerosborne/intro-to-web-scraping-build-your-first-scraper-in-5-minutes-1c36b5c4b110

Lambiase, C., Rossi, A., Morganti, R., Cancelli, L., Grosso, A., Tedeschi, R., . . . Bellini, M. (2024). Adapted Low-FODMAP Diet in IBS Patients with and without Fibromyalgia: Long-Term Adherence and Outcomes. *Nutrients*, 3419.

Lim, J. J. (2021, November 16). *How to analyse the statistical correlation between two delta values?* Retrieved from ResearchGate: https://www.researchgate.net/post/How_to_analyse_the_statistical_correlation_between_two_delta_values

Martin Breuss. (n.d.). *Beautiful Soup: Build a Web Scraper With Python.* Retrieved from Real Python: https://realpython.com/beautiful-soup-web-scraper-python/

Moayyedi, P., Mearin, F., Azpiroz, F., Andresen, V., Barbara, G., Corsetti, M., . . . Tack, J. (2017). Irritable bowel syndrome diagnosis and management: A simplified algorithm for clinical practice. *United European Gastroenterol J*, 773-788.

Muhammad Ikramullah Khan. (2025, Dec 24). *Why User Agent Matters in Web Scraping: A Beginner's Guide.* Retrieved from Medium: https://medium.com/@mikram2015/why-user-agent-matters-in-web-scraping-a-beginners-guide-d6c606b3daba

oseparovic. (n.d.). *fodmap_list.* Retrieved from https://github.com/oseparovic/fodmap_list

PostgreSQL. (n.d.). *5.9. Row Security Policies.* Retrieved from PostgreSQL: https://www.postgresql.org/docs/current/ddl-rowsecurity.html#:~:text=In%20addition%20to%20the%20SQL,INSERT%20%2C%20UPDATE%20%2C%20or%20DELETE%20.

Sonnet 4.6, C. (2026, 04 28). how many ingredients would i avoid if i followed a lactose free, gluten free, low histamine and low surfur diet. Anthropic.

Spectrum Health. (n.d.). Retrieved from Service Fees.: https://www.spectrumhealth.ie/fees

Statsig. (2025, Jan 05). *Delta variance: how it impacts experiment analysis.* Retrieved from Statsig: https://www.statsig.com/perspectives/deltavarianceimpactanalysis

Supabase. (n.d.). *Auth.* Retrieved from Supabase DOCS: https://supabase.com/docs/guides/auth

Supabase. (n.d.). *Use Supabase with Expo React Native.* Retrieved from Supabase DOCS: https://supabase.com/docs/guides/getting-started/quickstarts/expo-react-native

Weznaver, C., Nybacka, S., Simren, M., Tornblom, H., Jakobsson, S., & Storsrud, S. (2024). Patients' experiences of dietary changes during a structured dietary intervention for irritable bowel syndrome. *Journal of Human Nutrition and Dietetics*, 1336-1348.

Lambiase, C., Rossi, A., Morganti, R., Cancelli, L., Grosso, A., Tedeschi, R., Rettura, F., Mosca, M., de Bortoli, N., & Bellini, M. (2024). Adapted Low-FODMAP Diet in IBS Patients with and without Fibromyalgia: Long-Term Adherence and Outcomes. *Nutrients*, *16*(19), 3419. https://doi.org/10.3390/nu16193419