FitFuel: A Full-Stack Personalized Fitness and Nutrition Management Platform Using React, Supabase, and Role-Based Access Control

Mr. Umesh Nanavare[1], Mr. Shantanu Patil[2], Mr. Om Patil[3],Mr. Sunny Khokle[4],Mr. Sarthak Patil[5]
Department of Computer Science and Engineering
MIT ADT University, Loni Kalbhor – 412201, India
{ps.sarthak07,patilshantanu260, ompatil0251,sunny.khokle,}@gmail.com

***Abstract*—Digital health and wellness platforms have emerged as a critical category of consumer-facing software, yet the majority of existing solutions suffer from fragmented architectures, poor personalization, or inadequate scalability. This work presents FitFuel, a comprehensive full-stack fitness and nutrition management web application engineered to deliver structured, personalized fitness experiences through a cohesive and scalable system design. FitFuel integrates workout plan management, nutrition tracking with macro-level dietary insights, a real-time capable backend powered by Supabase (PostgreSQL), and a role-based access control (RBAC) architecture that distinguishes between standard users and administrators. The frontend is implemented using React 18 with TypeScript, styled via Tailwind CSS and the Radix UI component system, and bundled using Vite for optimized production delivery. State management is handled through React Context API and TanStack React Query, ensuring efficient server-state synchronization. The platform employs Supabase Auth for secure authentication supporting both email/password and OAuth flows, with relational schema design enforcing referential integrity between user profiles and workout or nutrition records. Extensive evaluation covers system performance, UI responsiveness, database query optimization, and usability metrics gathered from test users. This paper details the system architecture, database design, feature engineering, security considerations, and deployment strategy, and discusses the implications of the platform for scalable digital health solutions. The contributions span a reproducible full-stack architecture pattern, a formal description of the RBAC model, and an empirical evaluation across functional, performance, and user experience dimensions.**

***Index Terms*—Fitness management, nutrition tracking, React, Supabase, TypeScript, role-based access control, full-stack web application, personalized health, Tailwind CSS, PostgreSQL.**

I. Introduction

The global digital health market continues to expand at an accelerating pace, driven by rising consumer awareness of preventive healthcare, increasing smartphone penetration, and the broader availability of cloud-based backend-as-a-service platforms. Within this landscape, fitness and nutrition applications represent one of the most actively developed and commercially competitive segments. Despite this proliferation, a notable gap remains between the richness of specialized tools—workout trackers on one side, calorie counters on another—and the absence of cohesive, integrated platforms that unite these dimensions in a single, scalable, and developer-maintainable system.

Existing popular fitness applications such as MyFitness- Pal, Nike Training Club, and Fitbit suffer from a com- bination of issues: vendor lock-in to proprietary hardware ecosystems, closed-source architectures that prevent institu- tional customization, limited administrative controls for con- tent managers, and, most importantly, a lack of meaningful personalization beyond simple user profile inputs. Academic prototypes addressing fitness tracking have likewise tended toward hardware-centric sensor solutions or isolated machine learning classifiers, rarely addressing the complete software engineering lifecycle from data modeling to deployment.

FitFuel was conceived and engineered to address these shortcomings. The design philosophy centers on four pillars. First, centralization: all user-facing fitness and nutrition data flows through a single cohesive platform, eliminating the cognitive overhead of managing multiple apps. Second, personalization: the system is architected to support profile-driven recommendations and customized workout and diet assignments. Third, scalability: the backend leverages Supabase, a fully managed PostgreSQL platform with real-time capabilities and row-level security (RLS) policies, allowing the system to scale from individual users to institutional deployments. Fourth, administrative governance: a dedicated role-based admin panel empowers content managers to create, update, and assign workout and nutrition plans without requiring direct database access.

From a technical standpoint, the frontend stack is built on React 18 with TypeScript, utilizing the full Radix UI primitive ecosystem for accessible, unstyled component composition, styled via Tailwind CSS utility classes. The application is structured as a single-page application (SPA) with React Router v6 for declarative, type-safe routing. Data fetching and server-state synchronization are handled by TanStack React Query v5, a battle-tested asynchronous state management library that provides caching, background refetching, and optimistic updates. The backend is provided entirely by Supabase, including authentication, relational database (PostgreSQL), and optionally real-time subscriptions via WebSocket channels.

This paper makes the following contributions:

- A reproducible, well-documented full-stack fitness and nutrition web platform architecture using modern

JavaScript/TypeScript tooling.

- A formal description and implementation of a role-based access control (RBAC) system enforced at both the application layer and the database layer via Supabase Row-Level Security policies.
- A relational database schema design that supports extensible user profiling, workout plan management, nutrition tracking, and administrative content control with referential integrity.
- An empirical evaluation across functional correctness, UI performance (Core Web Vitals), and user experience dimensions with structured test user feedback.
- A discussion of deployment architecture, environment configuration, and strategies for production hardening of the platform.

The remainder of the paper is organized as follows. Section II surveys related work in digital fitness platforms and relevant web engineering literature. Section III describes the dataset and user profiling model. Section IV details the system architecture and component design. Section V formalizes the database schema and RBAC model. Section VI describes the frontend engineering methodology. Section VII covers the backend and API design. Section VIII presents experimental evaluation results. Section IX presents a case study of typical user journeys. Section X discusses deployment and operationalization. Section XI identifies limitations, and Section XII outlines future directions before the conclusion.

II. Related Work

A. Digital Fitness Application Architectures

The software architecture of fitness applications has evolved significantly over the past decade. Early mobile fitness apps operated largely in offline mode with local SQLite databases, synchronizing to cloud storage on demand. The introduction of cloud-native Backend-as-a-Service (BaaS) platforms—most notably Firebase and, more recently, Supabase—fundamentally altered this landscape by enabling real-time data synchronization, managed authentication, and serverless function execution without requiring teams to maintain traditional server infrastructure.

Several academic works have examined the architecture of mHealth (mobile health) platforms. Dehling et al. [1] provided a taxonomy of mHealth apps classifying them by data source, processing logic, and presentation, and identified that the absence of standardized architectures leads to fragmented ecosystems. Martinez et al. [2] specifically examined React-based progressive web apps (PWAs) for health monitoring, demonstrating that PWA performance metrics are comparable to native applications when implemented with service workers and optimized asset delivery.

B. Nutrition Tracking Systems

Nutrition tracking software has a long history, dating to early desktop applications such as NutriBase and DietPower. Modern systems like Cronometer and MyFitnessPal rely on large food composition databases (USDA FoodData Central, Open Food Facts) and user-contributed entries. Computational nutrition research has increasingly focused on macro-level tracking (proteins, carbohydrates, fats, calories) and micro-nutrient profiling. FitFuel's nutrition module is specifically designed around macro-level tracking aligned with common fitness goals (fat loss, muscle gain, maintenance), a design choice supported by evidence from sports nutrition literature [3] showing that macro distribution is a stronger determinant of body composition outcomes than precise caloric counting alone.

C. Role-Based Access Control in Web Applications

Role-Based Access Control (RBAC) is a mature access control paradigm formalized by Sandhu et al. [4] and subsequently standardized in NIST SP 800-53. In modern web applications, RBAC is typically implemented at multiple levels: the application layer (UI rendering based on role), the API layer (middleware guards), and the database layer (row-level security policies). Supabase's Row-Level Security (RLS) feature provides database-level RBAC through PostgreSQL policy definitions, ensuring that even direct API consumers cannot access unauthorized rows. This multi-layer approach, termed defense in depth, is considered a best practice in enterprise security architecture [5].

D. React Ecosystem for Complex Web Applications

The React ecosystem has matured considerably since its 2013 introduction. Hooks-based functional components, introduced in React 16.8, replaced the class component paradigm and enabled finer-grained composition of stateful logic. TypeScript adoption in large React codebases has been shown to reduce runtime type errors by up to 15% in empirical studies [6]. Component library ecosystems such as Radix UI provide accessible, unstyled primitives that allow teams to apply custom design systems without sacrificing WCAG 2.1 accessibility compliance. TanStack React Query provides a particularly powerful pattern for server-state management, decoupling remote data from local UI state and providing declarative cache invalidation semantics.

E. Gaps in Existing Literature

Despite the wealth of individual contributions, several gaps persist in the literature that FitFuel aims to address. First, there is a lack of open-source, fully documented full-stack fitness platforms that document both the software architecture and the empirical performance of the system. Second, the integration of BaaS platforms like Supabase with modern React tooling (React Query, React Router v6) has received limited academic treatment. Third, the design and enforcement of RBAC at the database layer using PostgreSQL RLS policies for fitness applications is underexplored. FitFuel is designed to bridge these gaps.

III. User Profiling and Data Model

A. User Profile Structure

FitFuel operates on a user-centric data model where every resource—workout assignments, nutrition plans, progress

records—is anchored to an authenticated user identity. Each user profile contains: a unique identifier linked to the Supabase Auth `auth.users` table, basic demographic fields (name, age, gender, height, weight), fitness goal classification (fat loss, muscle gain, endurance, general wellness), activity level, and dietary preferences. The fitness goal and activity level together drive the default macro distribution suggestions presented to users and the workout intensity filters applied during plan recommendations.

B. Goal-Based Categorization

Users are classified into one of four primary fitness goal categories upon registration:

1) *Fat Loss*: Caloric deficit regime with higher protein allocation to preserve lean mass.
2) *Muscle Gain (Hypertrophy)*: Caloric surplus with elevated protein and carbohydrate targets to support anabolic training.
3) *Endurance Training*: Moderate caloric intake with high carbohydrate allocation to sustain aerobic capacity.
4) *General Wellness*: Balanced macro distribution aligned with standard dietary reference intakes.

This categorization informs both the nutrition panel defaults and the workout plan filtering within the user dashboard, reducing cognitive load and improving the relevance of surfaced content.

C. Admin Content Management Model

The platform distinguishes two primary roles: `USER` and `ADMIN`. Administrators have the ability to create and manage the library of workout plans and nutrition plans that users can browse or be assigned. This separation of content creation (admin-controlled) from content consumption (user-controlled) mirrors standard content management system (CMS) architectures, ensuring data integrity and preventing unauthorized modification of shared fitness content.

IV. System Architecture

A. High-Level Architecture

FitFuel follows a three-tier architecture: a React-based frontend SPA as the presentation tier, the Supabase platform as the data and authentication tier, and the browser client as the session tier. There is no custom application server; all backend logic is handled by Supabase's PostgREST auto-generated REST API and Supabase Edge Functions where server-side logic is needed. This serverless pattern minimizes operational overhead while maintaining strong data integrity through database-level constraints and policies.

B. Frontend Architecture

The frontend is structured around the following organizational principles:

[System Architecture Diagram]
Browser (React SPA) ↔ Supabase REST API / Realtime
↓
Supabase Auth → JWT Token → RLS Policy Enforcement
↓
PostgreSQL Database (profiles, workouts, nutrition, logs)

Fig. 1. High-level architecture of FitFuel illustrating the SPA-to-Supabase communication pattern and row-level security enforcement chain.

1) Component Hierarchy: Pages are composed from atomic UI components drawn from the Radix UI primitive library and styled with Tailwind CSS. Compound components (e.g., a `WorkoutCard` composed of `Badge`, `Progress`, and `Button` atoms) encapsulate domain logic and reduce duplication. All components are written as TypeScript functional components with typed props interfaces.

2) Routing: React Router v6 provides declarative routing through a nested route tree. Protected routes are wrapped in a `PrivateRoute` component that reads the authentication state from the Supabase client and redirects unauthenticated users to the login page. Admin-only routes enforce an additional role check before rendering.

3) State Management: Two categories of state are managed distinctly: server state (remote data from Supabase, managed by React Query) and client state (UI state such as modals, form step progress, managed by React `useState` and `useContext`). This separation prevents the anti-pattern of duplicating server state in a global store, which historically leads to cache invalidation bugs.

4) Form Handling: All forms are managed using React Hook Form v7 with Zod schema validation. Zod schemas serve as the single source of truth for both runtime validation and TypeScript type inference, eliminating duplication between validation logic and type declarations.

C. Backend Architecture

Supabase provides the following backend services consumed by FitFuel:

1) PostgreSQL Database: The primary data store is a managed PostgreSQL instance. All application data are stored in well-normalized relational tables. Supabase exposes a PostgREST-based auto-generated REST API, allowing the React frontend to perform CRUD operations directly against the database using the Supabase JavaScript client SDK.

2) Authentication: Supabase Auth handles user registration, login, session management, and OAuth provider integration. Upon successful authentication, Supabase issues a JWT (JSON Web Token) that the client includes in subsequent API requests. PostgreSQL RLS policies inspect this JWT to enforce per-row access control.

3) Real-Time Subscriptions: Supabase's real-time engine, built on Phoenix Channels and WebSocket, allows the frontend to subscribe to database change events. FitFuel uses this capability for live progress updates on the user dashboard.

4) Edge Functions: Supabase Edge Functions (Deno runtime) handle server-side logic that should not be exposed to the client, such as sending notification emails upon plan assignment or computing aggregated progress statistics.

V. Database Schema and RBAC Model

A. Relational Schema Design

The FitFuel database is organized around the following core tables. A summary is presented in Table I.

TABLE I
Core Database Schema Summary

Table	Key Columns and Purpose
profiles	id (FK → auth.users), name, age, gender, height, weight, goal, activity level, role
workout_plans	id, title, description, category, difficulty, duration weeks, created by (FK → profiles)
workout_exercis	eisd, plan id (FK), exercise_name, sets, reps, rest seconds, order index
nutrition_plans	id, title, description, goal_type, calories_per_day, protein_g, carbs_g, fat_g, created by
meal_items	id, plan_id (FK), meal_name, food_name, quantity g, calories, protein g, carbs g, fat g
user_workout_lo	gisd, user id (FK), plan id (FK), exercise_id (FK), completed at, notes
user_nutrition_	liodg,suser id (FK), plan id (FK), meal_id (FK), logged at, quantity consumed
user_plan_assig	nimde,ntusser id (FK), plan type, plan id, assigned by (FK), assigned at, status

Referential Integrity and Constraints

VI. Frontend Engineering

A. Component Architecture and Design System

FitFuel's UI is built on a design system composed of Radix UI primitives and Tailwind CSS utilities. Radix UI provides unstyled, accessible React components (e.g., `Dialog`, `DropdownMenu`, `Tabs`, `Accordion`) that implement correct keyboard navigation, ARIA attributes, and focus management out of the box. Styling is applied via Tailwind CSS utility classes, enabling rapid UI iteration without maintaining a custom CSS file hierarchy.

The `class-variance-authority` (CVA) library is used to define component variant maps, enabling type-safe variant-driven styling. For example, the `Button` component defines `variant` props (`default`, `destructive`, `outline`, `ghost`, `link`) and `size` props (`default`, `sm`, `lg`, `icon`) as CVA variant configurations. `tailwind-merge` is used to resolve Tailwind class conflicts when consumer code passes additional classes.

B. Dashboard Architecture

The user dashboard is the central feature of the platform, providing an at-a-glance view of:

- Current workout plan progress (visualized via a `recharts` progress bar chart).
- Today's nutrition summary with macro breakdown (protein, carbohydrates, fat) represented as a pie chart.
- Upcoming workout sessions pulled from the assigned plan schedule.
- Recent activity log entries.
- Quick-action buttons to log a workout or log a meal.

Dashboard data is fetched using React Query with a stale-time of 60 seconds and a background refetch interval of 300 seconds, balancing data freshness against API call overhead.

C. Workout Management Module

The workout management module presents users with a browse-and-filter interface for the workout plan library. Plans are categorized by fitness goal (strength, cardio, HIIT, yoga, flexibility) and difficulty (beginner, intermediate, advanced). Each plan card displays the title, category badge, difficulty badge, estimated duration, and a brief description. Selecting a plan opens a detailed view displaying the complete exercise list with sets, reps, rest periods, and embedded instructional notes.

For administrators, the same module exposes CRUD controls: a `WorkoutPlanForm` component backed by React Hook Form and Zod validation allows creation or editing of plans and their constituent exercises. Exercise items support drag-and-drop reordering via keyboard-accessible controls to set `order_index`, ensuring the sequence of exercises is explicitly captured in the database.

D. Nutrition Tracking Module

The nutrition module provides two views: a plan browser (mirroring the workout plan browse experience) and a daily logging interface. In the logging interface, users select a nutrition plan, choose a meal, and log individual food items with portion adjustments. The front-end computes adjusted macro values in real-time as the user modifies quantity inputs:

$$\text{adjusted\_macro} = \frac{\text{quantity\_consumed}}{\text{reference\_quantity}} \times \text{reference\_macro}$$

Daily totals are accumulated and compared against the target macros from the assigned plan, with a color-coded progress indicator (green for on-target, amber for slightly over/under, red for significantly deviant).

E. Admin Panel

The admin panel is accessible exclusively to users whose `profiles.role = 'admin'`. It provides:

- A user management table listing all registered users, their goals, assigned plans, and account status.
- Workout plan CRUD interface.
- Nutrition plan CRUD interface.
- Plan assignment controls: ability to assign a workout or nutrition plan to a specific user or group of users sharing a goal type.
- Activity overview: aggregate statistics on user engagement (plans assigned, logs submitted in the last 30 days).

The admin panel uses the same component library as the user-facing panels, reducing the total codebase surface area. Role-based rendering logic is centralized in a custom

`useRole` hook that reads the current user's role from the profiles table via React Query.

F. Authentication Flow

The authentication flow supports two pathways: email/password registration and login, and OAuth via supported providers (Google, GitHub) through Supabase Auth. The `AuthContext` provider wraps the entire application and exposes the current session, user object, and helper functions (`signIn`, `signUp`, `signOut`). On session initialization, the `AuthContext` fetches the user's profile from the `profiles` table to populate role and preference data.

A custom `ProtectedRoute` wrapper component guards all authenticated routes. An additional `AdminRoute` wrapper guards admin-only routes, rendering a 403 Forbidden page for authenticated non-admin users rather than silently redirecting, improving transparency.

VII. Backend and API Design

A. Supabase Client Configuration

The Supabase JavaScript client (`@supabase/supabase-js` v2.91.0) is initialized with the project URL and anon key sourced from environment variables:

```
const supabase =
  createClient( import.meta.env.VITE_SUPAB
  ASE_URL,
  import.meta.env.VITE_SUPABASE_ANON_KEY
);
```

The anon key is a safe-to-expose public key: it grants access only to data permitted by RLS policies for unauthenticated requests. Authenticated requests automatically include the user's JWT, activating user-specific RLS policies.

B. Query Patterns and Optimization

FitFuel leverages Supabase's PostgREST query API for all database operations. Common query patterns include:

1) Selective Columns with Related Data: To minimize network payload, queries select only required columns and use PostgREST's embedding syntax to join related tables:

```
const { data } = await supabase
  .from('workout_plans')
  .select(`
    id, title, description, difficulty,
    workout_exercises (
      exercise_name, sets, reps,
      rest_seconds, order_index
    )
  `)
  .order('order_index',
    { referencedTable: 'workout_exercises
    ', ascending: true
  });
```

2) Filtered Queries: Dashboard data fetches use compound filters to retrieve only the user's relevant records within a date range:

```
const { data } = await supabase
  .from('user_workout_logs')
  .select('*')
  .eq('user_id', userId)
  .gte('completed_at', startDate)
  .lte('completed_at', endDate)
  .order('completed_at', { ascending: false });
```

3) Upsert for Idempotent Profile Updates: Profile updates use `upsert` to handle both initial creation and updates idempotently:

```
await supabase
  .from('profiles')
  .upsert({ id: userId, ...profileData });
```

C. Real-Time Subscriptions

The user dashboard subscribes to real-time changes on `user_workout_logs` and `user_nutrition_logs` to reflect updates (e.g., from a concurrent mobile session) without requiring manual refresh:

```
const channel = supabase
  .channel('user_logs')
  .on('postgres_changes',
    { event: 'INSERT', schema: 'public',
      table: 'user_workout_logs',
      filter: `user_id=eq.${userId}` },
    (payload) =>
      queryClient.invalidateQueries( ['workoutLogs
      '] )
  )
  .subscribe();
```

D. Error Handling

All Supabase client calls are wrapped in standardized error handlers. React Query's `onError` callbacks surface user-friendly toast notifications via the `sonner` library, ensuring that API failures are visible to users without exposing raw error details. Critical errors (e.g., authentication failures, RLS policy violations) are additionally logged to the browser console in development mode.

VIII. Experimental Evaluation

A. Evaluation Dimensions

FitFuel was evaluated across four dimensions: functional correctness, frontend performance (Core Web Vitals), database query performance, and user experience (UX) usability testing.

B. Functional Correctness Testing

Functional tests were written using Vitest and React Testing Library. The test suite covers: authentication flows (registration, login, logout, OAuth redirect), profile CRUD operations, workout plan browsing and filtering, nutrition log submission,

admin plan creation, and RBAC enforcement (attempting admin actions as a standard user). Table II summarizes test coverage by module.

TABLE II
FUNCTIONAL TEST COVERAGE BY MODULE

Module	Test Cases	Pass Rate (%)
Authentication	18	100
User Profile	12	100
Workout Browse & Filter	15	100
Workout Logging	10	100
Nutrition Browse	14	100
Nutrition Logging	11	100
Admin Plan CRUD	16	100
Admin User Management	9	100
RBAC Enforcement	14	100
Total	**119**	**100**

All 119 test cases pass, confirming functional correctness across the full feature set. Edge cases such as duplicate email registration, invalid macro inputs, and concurrent session handling were explicitly included in the test suite.

C. Frontend Performance Evaluation

Frontend performance was measured using Lighthouse (v11) on a production build deployed to Vercel, evaluated on a simulated mid-range Android device over a 4G network connection. Core Web Vitals results are summarized in Table III.

TABLE III
CORE WEB VITALS — PRODUCTION BUILD (LIGHTHOUSE V11)

Metric	Score	Rating
Largest Contentful Paint (LCP)	1.8s	Good
First Input Delay (FID)	12ms	Good
Cumulative Layout Shift (CLS)	0.03	Good
Time to Interactive (TTI)	2.4s	Good
Total Blocking Time (TBT)	85ms	Good
Speed Index	1.9s	Good
Overall Performance Score	**91/100**	**Good**

The LCP of 1.8 seconds is achieved through Vite's efficient code splitting (dynamic `import()` for route-level lazy loading), Tailwind CSS purging (removing unused utility classes from the production bundle, reducing CSS bundle from 3MB to 12KB), and Supabase's global CDN for asset delivery. The low CLS score (0.03) is a direct consequence of defining explicit width and height on image and skeleton loader elements, preventing layout reflow as data loads.

D. Database Query Performance

Query performance was measured for the five most frequently executed queries in the application, using PostgreSQL's `EXPLAIN ANALYZE` on a dataset of 5,000 users, 200 workout plans, 150 nutrition plans, and approximately 50,000 log entries. Results are presented in Table IV.

All user-facing queries execute in under 5ms on this dataset, well within the perceptible latency threshold of 100ms [7]. The admin user listing query at 11.4ms remains within acceptable bounds for an infrequently executed administrative operation. Composite indexes on `(user_id, completed_at)` and `(user_id, logged_at)` were critical to achieving this performance for log queries.

TABLE IV
DATABASE QUERY PERFORMANCE (POSTGRESQL EXPLAIN ANALYZE)

Query	Avg. Time (ms)	Index Used
Fetch user profile	0.8	Primary key
Fetch assigned plans	3.2	FK index on user_id
Fetch workout logs (30 days)	4.7	Composite (user_id, completed_at)
Fetch nutrition logs (7 days)	3.9	Composite (user_id, logged_at)
Admin: list all users	11.4	Seq. scan + role index

E. Usability Testing

Usability testing was conducted with 15 test participants (8 regular fitness enthusiasts, 4 fitness instructors acting as admins, 3 users with no prior fitness app experience) over structured task scenarios. Table V summarizes the System Usability Scale (SUS) scores by user group.

TABLE V
SYSTEM USABILITY SCALE (SUS) RESULTS

User Group	Avg. SUS Score	Grade
Regular Fitness Users	84.2	Excellent
Admin / Instructor Users	79.6	Good
First-time Fitness App Users	76.4	Good
Overall	**81.7**	**Excellent**

The overall SUS score of 81.7 falls in the "Excellent" range on the standard SUS grading scale (scores above 80.3 are rated Excellent). Admin users reported the admin panel as slightly less intuitive than the user-facing dashboard, primarily around the multi-step plan creation workflow—an observation that informs future UI improvements. First-time users found the dashboard easy to navigate but requested more contextual help text alongside nutrition macro fields.

F. Bundle Size Analysis

The production JavaScript bundle was analyzed using `vite-bundle-visualizer`. The initial JS bundle is 187KB (gzipped), with route-level code splitting deferring an additional 340KB of non-critical module code to lazy-loaded chunks. The Radix UI and Recharts libraries together account for approximately 62% of the initial bundle size; future optimization efforts may target selective imports from these libraries.

IX. CASE STUDY: TYPICAL USER JOURNEY

A. New User Registration and Onboarding

A new user registers using their email and password. Supabase Auth creates an entry in `auth.users`, which triggers the `handle_new_user` function to create a corresponding `profiles` row. The user is redirected to a multi-step onboarding form collecting demographic data, fitness goal, and activity level. This data is written to `profiles` via an `upsert` call. Upon completion, the dashboard renders

with recommended workout and nutrition plans filtered by the user's declared goal.

B. Weekly Workout Tracking

The user browses the workout plan library, selects an "Intermediate Strength Training" plan, and views the 5-day per-week exercise schedule. They begin Week 1, Day 1: Bench Press (4 sets × 8 reps), Squat (4 × 6), Overhead Press (3 × 8), and Barbell Row (3 × 8). After completing the session, they log the workout through the quick-log interface, which creates entries in `user_workout_logs`. The dashboard's progress ring immediately updates via the React Query cache invalidation triggered by the real-time subscription.

C. Nutrition Logging

The user is assigned a 2,800 kcal muscle gain nutrition plan by their trainer (an admin user). The plan defines four meals per day with macro targets: 210g protein, 350g carbohydrates, 80g fat. The user logs their breakfast (oats, eggs, milk) with portion adjustments, and the nutrition panel updates the daily progress bars. By end of day, the user has reached 195g protein, 330g carbs, and 78g fat—within 95% of target across all macros.

D. Admin Plan Assignment

A fitness trainer logs into the admin panel and navigates to the user management table. They filter users by goal = "Muscle Gain" and select a cohort of 12 users. Using the bulk assignment control, they assign the "12-Week Hypertrophy Program" workout plan to all 12 users. An entry is created in `user_plan_assignments` for each user, and the RLS policy ensures the trainer can only assign plans (not modify user credentials or access payment data). The selected users see the new plan appear on their dashboards upon next login.

X. Deployment and Operationalization

A. Build and Deployment

FitFuel is built using Vite v5.4, which generates a highly optimized production bundle with tree-shaking, minification, and code splitting. The production build is deployed on Vercel, which provides automatic HTTPS, global CDN distribution, and zero-configuration preview deployments for pull requests. Environment variables (`VITE_SUPABASE_URL` and `VITE_SUPABASE_ANON_KEY`) are configured as project secrets in the Vercel dashboard, never committed to the repository.

B. Environment Configuration

Local development uses a `.env.local` file (gitignored). Supabase provides a local development CLI (`supabase start`) that spins up a local PostgreSQL instance with the production schema applied, enabling full offline development and testing without impacting the production database.

C. Database Migrations

Schema changes are managed through Supabase Migration files stored in the `supabase/migrations/` directory (tracked in version control). This ensures that schema changes are reproducible, auditable, and applied consistently across development, staging, and production environments. Migrations include table creation, RLS policy definitions, trigger functions, and index creation.

D. Security Hardening

Beyond RLS policies, several additional security measures are implemented. The Supabase anon key is restricted via RLS so that unauthenticated requests can only access explicitly public data (e.g., the public workout plan catalog). Sensitive operations (e.g., reading other users' logs) are protected by RLS policies that evaluate the authenticated user's role and identity. Input validation via Zod schemas prevents malformed data from reaching the database layer. No sensitive data is stored in `localStorage`; Supabase Auth sessions use secure, HttpOnly cookies in production.

E. Monitoring and Maintenance

Application health is monitored through Vercel Analytics (deployment error rates, edge request volumes) and Supabase's built-in database metrics dashboard (query performance, connection pool utilization, storage growth). Supabase sends weekly digest emails summarizing database activity. Future production deployments would integrate error tracking via Sentry for client-side JavaScript errors and a dedicated APM tool for database query regression detection.

XI. Limitations

The current implementation has several limitations that represent opportunities for future improvement:

- *Absence of AI/ML-driven recommendations*: Workout and nutrition plan recommendations are currently based on user-declared goal and activity level rather than behavioral data analysis. Integrating a recommendation engine or collaborative filtering model would improve personalization fidelity.
- *No wearable device integration*: FitFuel currently lacks integration with wearable fitness devices (Fitbit, Apple Watch, Garmin) and thus cannot ingest automated activity data. REST API adapters for common wearable platforms are a planned extension.
- *Limited food database*: The current nutrition module requires admins to manually input food items. Integration with open food databases such as USDA FoodData Central or Open Food Facts would dramatically expand the nutritional content library.
- *No offline support*: The application currently requires an active internet connection. Implementing a service worker with background sync would enable offline log entry and deferred synchronization.
- *Single-language support*: The UI is currently English-only. Internationalization (i18n) using `react-i18next`

would expand the platform's reach to non-English-speaking fitness communities.

- *No mobile native app*: FitFuel is a web-first application. A React Native version sharing the Supabase backend and business logic (via a shared TypeScript library) would expand accessibility to mobile-only users.

XII. Future Work

Based on the limitations identified and user feedback gathered during usability testing, several directions are prioritized for future development:

- *AI-Powered Plan Personalization*: Integration of a lightweight recommendation model trained on anonymized user outcome data (weight progress, workout completion rates) to dynamically adjust plan difficulty and macro targets.
- *Wearable Device Integration*: OAuth-based integration with Fitbit and Google Fit APIs to automatically import step counts, heart rate data, and active calorie expenditure, enriching the dashboard's contextual data.
- *Food Database Integration*: Supabase Edge Function serving as a proxy to the USDA FoodData Central API, enabling real-time food search and automatic macro population during meal logging.
- *Progress Photography*: An optional feature allowing users to securely upload progress photographs stored in Supabase Storage, enabling visual tracking of body composition changes over time.
- *Social Features*: A trainer-client communication channel and the ability for users to share workout achievements within a private network, increasing engagement and accountability.
- *Probabilistic Goal Forecasting*: Using historical log data, project when a user is on track to meet their stated goal, providing forward-looking motivational feedback on the dashboard.
- *Progressive Web App*: Implementing a service worker and manifest to enable installation as a home-screen PWA with offline support and push notification capabilities.

XIII. Policy and Broader Implications

The design philosophy of FitFuel has implications beyond the immediate fitness application domain. The architectural pattern—a type-safe React SPA, BaaS-backed with database-layer RBAC, deployed serverlessly—represents a generalizable template for a wide class of consumer health applications including mental wellness trackers, chronic disease management tools, and rehabilitation monitoring systems.

From a data privacy perspective, FitFuel's health data falls within the scope of personal health information under relevant data protection frameworks (PDPA in India, GDPR in Europe). The use of Supabase RLS ensures that user health data is inaccessible except to the data owner and explicitly authorized administrators, a property critical to regulatory compliance. Future institutional deployments (hospitals, gyms, corporate wellness programs) would benefit from enhanced audit logging and consent management modules.

For gym and fitness center operators, FitFuel's admin panel model provides a blueprint for replacing manual spreadsheet-based client management with a structured digital system. The ability to assign specific plans to clients, track adherence, and update plans centrally represents a meaningful operational efficiency gain in the fitness service industry.

XIV. Conclusion

This paper presented FitFuel, a comprehensive full-stack fitness and nutrition management platform designed to deliver personalized, structured health experiences through a scalable and maintainable software architecture. The system integrates a React 18 / TypeScript frontend with a Supabase-backed PostgreSQL database, enforcing a formally defined RBAC model at both the application and database layers through PostgreSQL Row-Level Security policies.

Experimental evaluation demonstrated an overall Lighthouse performance score of 91/100 with an LCP of 1.8 seconds, sub-5ms database query times for all user-facing operations, 100% functional test pass rate across 119 test cases, and a System Usability Scale score of 81.7 (Excellent grade). A case study of typical user journeys confirmed the coherence of the end-to-end data flow from registration through plan assignment, workout tracking, and nutrition logging.

FitFuel's architecture demonstrates that modern BaaS platforms enable small development teams to build production-grade, secure, and performant health applications without maintaining custom server infrastructure. The platform is positioned as a foundation for future extensions including AI-powered recommendations, wearable device integration, and mobile application development. The codebase, migrations, and deployment configuration serve as a reproducible reference implementation for full-stack health application development using the React and Supabase ecosystem.

Acknowledgment

The authors thank [Your Institution Name] for providing the resources and mentorship that supported this work. The authors also acknowledge the open-source communities behind React, Supabase, Tailwind CSS, Radix UI, and TanStack React Query, whose contributions form the foundation of this platform.

References

[1] T. Dehling, F. Gao, S. Schneider, and A. Sunyaev, "Exploring the Far Side of Mobile Health: Information Security and Privacy of Mobile Health Apps on iOS and Android," *JMIR mHealth and uHealth*, vol. 3, no. 1, e8, 2015.

[2] C. Martinez and J. Rodriguez, "Performance Analysis of Progressive Web Applications for Health Monitoring Systems," *Journal of Medical Internet Research*, vol. 23, no. 6, e25027, 2021.

[3] L. M. Burke, J. A. Hawley, S. H. Wong, and A. E. Jeukendrup, "Carbohydrates for training and competition," *Journal of Sports Sciences*, vol. 29, Suppl 1, pp. S17–27, 2011.

[4] R. S. Sandhu, E. J. Coyne, H. L. Feinstein, and C. E. Youman, "Role-based access control models," *IEEE Computer*, vol. 29, no. 2, pp. 38–47, 1996.

[5] OWASP Foundation, "OWASP Application Security Verification Standard 4.0.3," OWASP, 2021. [Online]. Available: https://owasp.org/www-project-application-security-verification-standard/

[6] Z. Gao, C. Bird, and E. T. Barr, "To type or not to type: Quantifying detectable bugs in JavaScript," in *Proc. 39th IEEE/ACM ICSE*, Buenos Aires, Argentina, 2017, pp. 758–769.

[7] J. Nielsen, "Response times: The three important limits," in *Usability Engineering*, San Francisco, CA: Morgan Kaufmann, 1994, ch. 5.

[8] T. Linsley, "TanStack Query v5: Async State Management for TypeScript & React," 2023. [Online]. Available: https://tanstack.com/query/v5

[9] Supabase Inc., "Supabase Documentation: Row Level Security," 2024. [Online]. Available: https://supabase.com/docs/guides/database/postgres/row-level-security

[10] E. You, "Vite: Next Generation Frontend Tooling," 2024. [Online]. Available: https://vitejs.dev/