**Technical Documentation**

**KaneFlow**

**Finance Tracking Web Application**

**Author: Tasdid Kabir Adil**

Table of Contents

[Introduction 4](#_Toc203944678)

[Purpose 4](#_Toc203944679)

[Scope 4](#_Toc203944681)

[Architecture Overview 5](#_Toc203944682)

[Backend 5](#_Toc203944683)

[Frontend 5](#_Toc203944684)

[Database 6](#_Toc203944685)

[Infrastructure 6](#_Toc203944686)

[User Stories 6](#_Toc203944687)

[Development Roadmap 12](#_Toc203944688)

[Iteration 1 - User Identity & Core Layout 13](#_Toc203944689)

[Key Tasks Completed: 13](#_Toc203944690)

[Iteration 2 – Expense Management & Reporting 13](#_Toc203944691)

[Key Tasks Completed: 14](#_Toc203944692)

[Iteration 3 – Saving Goals, Visualization, and UX Polish 15](#_Toc203944693)

[Key Tasks Completed: 15](#_Toc203944694)

[Project Structure 17](#_Toc203944695)

[Installation and Setup 18](#_Toc203944696)

[Prerequisites 18](#_Toc203944697)

[Backend Setup 18](#_Toc203944698)

[Environment Variables 18](#_Toc203944699)

[Frontend Setup 19](#_Toc203944700)

[API & GraphQL Schema 19](#_Toc203944701)

[GraphQL Queries Functionalities 19](#_Toc203944702)

[User Queries 19](#_Toc203944703)

[Expense Queries 20](#_Toc203944704)

[Monthly Report Queries 21](#_Toc203944705)

[Saving Goals Queries 21](#_Toc203944706)

[GraphQL Mutation Functionalities 21](#_Toc203944707)

[User Mutations 21](#_Toc203944708)

[Expense Mutations 22](#_Toc203944709)

[Monthly Report Mutations 23](#_Toc203944710)

[Savings Goals & AI Suggestions 23](#_Toc203944711)

[Component & Data Flow 24](#_Toc203944712)

[State Management 24](#_Toc203944713)

[Data Fetching and Flow 24](#_Toc203944714)

[1. User Authentication Flow: 24](#_Toc203944715)

[2. Expense Entry Flow: 25](#_Toc203944716)

[3. Savings & Suggestions Flow: 25](#_Toc203944717)

[4. Reporting Flow: 25](#_Toc203944718)

[Data Flow (UML Sequence Diagram) 25](#_Toc203944719)

[Component Hierarchy 26](#_Toc203944720)

[Auth & Routing 27](#_Toc203944721)

[Testing Strategy 28](#_Toc203944722)

[Backend Testing 28](#_Toc203944723)

[Frontend Testing (Planned) 30](#_Toc203944724)

[Deployment Process 31](#_Toc203944725)

[Backend Deployment (Railway) 31](#_Toc203944726)

[Frontend Deployment (Netlify) 32](#_Toc203944727)

[References & Resources 33](#_Toc203944728)

# Introduction

## Purpose

## In today’s fast-paced and capitalistic economy, maintaining control over personal spending has become increasingly challenging. People often unintentionally overspend due to busy schedules, lack of organization, or simply forgetfulness. Existing expense tracking applications frequently come with limitations such as paid services, lack of updates, or insufficient personalization. Kaneflow addresses these issues by providing a fully-featured, personalized, and user-friendly financial tracking experience. With a streamlined and intuitive interface, personalized expense analysis, and intelligent AI-driven suggestions to optimize spending habits, Kaneflow delivers a seamless, accessible, and effective solution. This application is compatible across various platforms, including mobile devices, personal computers, and tablets, ensuring users have continuous and effortless access.

## Scope

Kaneflow is specifically tailored to meet the financial tracking and management needs of individual users. Core features include detailed daily and monthly expense analysis, customizable categorical tags for better expense organization, personalized monthly savings budgets, and AI-driven recommendations aimed at optimizing user spending habits. The application's intuitive design and cross-platform compatibility ensure usability and convenience for users managing their personal finances, making their budgeting journey easier and more efficient.

# Architecture Overview

## Backend

Kaneflow’s backend is built using Node.js and Express.js, providing a robust and scalable server-side environment. Apollo Server is utilized to efficiently manage GraphQL APIs, allowing for clear and structured data exchanges between the frontend and backend. JWT is implemented for secure user authentication, while bcrypt enhances password security. The backend also incorporates scheduled tasks for automated monthly report generation and expense analytics, ensuring seamless and reliable operation.

## Frontend

The frontend leverages React to provide a dynamic and responsive user interface, with Apollo Client seamlessly integrating GraphQL queries and mutations for data fetching and state management. Vite is used as the build tool to enhance frontend development efficiency and performance. CSS Modules are adopted for scoped and maintainable styling, creating an organized and cohesive user interface that offers a seamless user experience across all platforms.

## Database

MongoDB is the chosen database solution, offering flexible and scalable data management for Kaneflow’s evolving schema needs. Its NoSQL nature allows for efficient storage and querying of data such as expenses, user profiles, and monthly financial reports. MongoDB’s robust querying capabilities provide fast and reliable access to user-specific financial data, ensuring quick and personalized responses to user actions.

## Infrastructure

Kaneflow’s infrastructure incorporates GitHub for version control and CI/CD integration. GitHub Actions facilitate automated testing and streamlined deployment workflows, improving development speed and reliability. Netlify is utilized for frontend deployment, providing a robust and scalable hosting solution that delivers continuous deployment capabilities with minimal downtime. Railway hosts the backend services, offering a straightforward and reliable cloud platform for seamless server deployment, scalability, and ease of management.

# User Stories

|  |  |  |
| --- | --- | --- |
| **User Story** | **Action** | **Acceptance Criteria** |
| As a user, I want to register with my personal information | * Navigate to Register page from Home page. * Input valid credentials in registration form. * Submit registration and verify email account. | * Registration form must validate email format and password length before submission * A success message is shown after submission, and a verification email is sent * Clicking the verification link marks the account as verified and allows login |
| As a user, I want to login to the app | * Navigate to the login page * Enter valid email and password * Submit the login form | * User is authenticated and redirected to dashboard * Invalid credentials show error message * JWT token is stored in localStorage |
| As a user, I want to be able to reset my password | * Enter registered email * Click on "Forgot Password" in login page * Click the reset link from email and enter a new password | * Reset email is sent with a working link * User can set a new password via reset form * User sees confirmation message after reset |
| As a user, I want to be able to see my personal info | * Navigate to profile page * View name, email, currency, and other saved preferences | * Profile page displays correct user data * Data matches backend user record |
| As a user, I want to be able to update my personal info | * Edit profile fields (name, currency, etc.) * Submit changes via save button | * Updated data is reflected in frontend and database * User sees a success message on update |
| As a user, I want to be able to delete my account | * Navigate to profile page * Click "Delete Account" and confirm deletion | * Account and related data are removed from backend * User is logged out and redirected to home page |
| As a user, I want to stay logged in and want to logout | * Remain signed in across page refresh * Click logout button from navbar or profile page | * JWT persists in localStorage until logout or expiry * Logout clears token and redirects to login or home * No access to protected routes after logout |
| As a user, I want to navigate between pages | * Click links or buttons in navbar * Use browser back/forward to navigate | * App uses client-side routing for page transitions * Active route is visually highlighted * Invalid routes show a 404 page or redirect |
| As a user, I want to add, edit, and delete my daily expense by name and category | * Fill in expense name, amount, category * Edit or delete items in expense list | * Added expenses appear instantly and persist after refresh * Editing updates the backend and UI * Deleting removes the record from UI and DB |
| As a user, I want to view the monthly total expense up to today | * Open dashboard or report view * View summary card or section showing current month's spending | * Total reflects sum of all expenses from 1st to current day * Updates automatically when new expense is added |
| As a user, I want to view expenses sorted by category, amount and date | * Use dropdown or filter to sort by category, amount or date * View sorted list or visual graph/chart | * Sorting applies instantly to the displayed data * Category, amount and date-based groupings are accurate * Default view is most recent date first |
| As a user, I want a monthly report of total expense, category-wise breakdown, savings, highest spending area, and comparisons to previous months. | * View the reports section on dashboard or Reports page * Select a specific month to review * Read breakdown chart, savings indicator, and comparison summaries | * Report shows total spent, savings, and category-wise breakdown * Displays top spending category * Includes comparison with previous month’s total and each category |
| As a user, I want to set a savings goal each month | * Navigate to the Savings page * Enter monthly savings target and optional thresholds per category * Submit the form to save goal | * Saving goal is stored and retrieved based on selected month * UI reflects active saving goal with progress bar * Existing goal can be updated or deleted |
| As a user, I want to view my progress toward the saving goal | * View dashboard or savings progress section * Compare total categorized spending against thresholds | * Progress bar visually indicates spent vs. threshold per category * Remaining savings amount is shown dynamically * Over-budget categories are visually highlighted |
| As a user, I want AI suggestions to optimize my spending habits | * Provide monthly income and savings target * Click "Get Suggestions" button * Review suggested budget distribution by category | * AI-generated suggestions follow internal logic rules (e.g., rent is fixed, entertainment is flexible) * Response includes a recommendation note and percentage adjustments * Suggestions are editable before saving as a goal |

# Development Roadmap

The product was developed using 3 consecutive iterations.

## Iteration 1 - User Identity & Core Layout

The first development iteration focused on laying the foundational architecture for both backend and frontend, establishing secure user identity management, and building the navigation framework for the application.

### Key Tasks Completed:

* **User Authentication:**

Implemented user registration, login, and password reset functionalities using secure JWT-based authentication. Validation and error handling were added to ensure a smooth user experience.

* **Layout Configuration:**

Designed the initial app layout including navigation menus, route protection for authenticated pages, and UI consistency across views.

* **Personal Information:**

Developed the user profile page allowing users to view, update, or delete their account and personal details. Profile updates include name and preferred currency.

## Iteration 2 – Expense Management & Reporting

This iteration focused on implementing the core financial tracking features, including detailed expense management and monthly financial reporting. It emphasized usability, real-time updates, and data clarity to help users better understand and manage their finances.

### Key Tasks Completed:

* **Expense Tracking (CRUD):**  
  Implemented the ability for users to add, edit, and delete daily expenses by category and description. This included input validation, persistent backend storage, and real-time UI updates upon change.
* **Monthly Spending Summary:**  
  Displayed the user’s total monthly expenses up to the current day. This feature dynamically updates with new entries and reflects the ongoing financial snapshot.
* **Sorting & Filtering:**  
  Enabled users to sort expenses by category and date, helping them analyze trends and identify high-spending areas more clearly.
* **Detailed Monthly Report:**  
  Generated comprehensive reports for each month, showing:
  + Total spending
  + Category-wise breakdown
  + Savings performance
  + Highest spending area
  + Comparisons to previous months

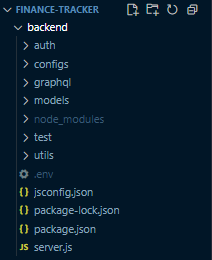
## Iteration 3 – Saving Goals, Visualization, and UX Polish

The third and final development iteration focused on advanced financial visualization, AI-assisted budgeting, and overall polish to the application’s user interface and responsiveness. This iteration delivered a complete and insightful budgeting experience for users.

### Key Tasks Completed:

* **Savings Goals Setup & AI Suggestions:**  
  Integrated a backend system for saving goals, including monthly targets and category-based thresholds. Implemented an AI-based suggestion engine that proposes optimized spending caps based on the user’s income and habits.
* **Spending vs. Threshold Display:**  
  Built a dynamic dashboard module that visually compares actual spending against user-defined thresholds, highlighting overspending areas and remaining savings.
* **Visual Analytics & Charts:**
  + **Categorical Spending Breakdown:** Bar and pie charts showing category-wise expenses for current month.
  + **Last 5 Months Trend Chart:** Combined bar and line chart to display spending patterns across the last five months.
  + **Savings Page Donut Chart:** Displayed remaining vs. used budget per category, with central text showing leftover savings.
* **Dashboard Enhancements:**  
  Added responsive cards showing:
  + Daily and monthly spending totals
  + Top expense categories
  + Highest single spending entry
* **UI/UX Styling & Responsiveness:**  
  Polished all styles using CSS modules and media queries. Improved mobile-first responsiveness across all views.
* **Landing Page Updates:**  
  Refreshed landing page with clear CTAs and minimal design. Improved accessibility and responsiveness.
* **Demo Account Functionality:**  
  Added support for demo accounts to let users explore the app without registration.
* **Final Bug Fixes & Polish:**  
  Wrapped up remaining backend logic, validations, loading spinners, and edge case handling to prepare for public release.

# Project Structure



# Installation and Setup

## Prerequisites

* Node.js v16+
* MongoDB
* npm/yarn

## Backend Setup

*cd backend*

*npm install*

*cp. env.example. env*

*# Populate. env variables*

*npm run dev*

## Environment Variables

|  |  |  |
| --- | --- | --- |
| **Variables** | **Description** | **Example** |
| MONGO\_URI | MongoDB connection URI | mongodb://localhost/db |
| PORT | Backend server port | 4000 |
| JWT\_SECRET | Secret key for JWT token | <your\_secret\_key> |
| EMAIL\_USER | Your gmail account email | yourname@gmail.com |
| EMAIL\_PASSWORD | You gmail app-password | < your\_gmail\_app\_password> |
| GEMINI\_API\_KEY | Valid Gemini api key | <gemini\_api\_key> |

## Frontend Setup

*cd frontend*

*npm install*

*npm run dev*

# API & GraphQL Schema

## GraphQL Queries Functionalities

The Kaneflow backend exposes a wide set of GraphQL queries that power all user, expense, savings, and report-related views on the frontend. Below is an overview of the main query operations:

### User Queries

* **users**  
  Returns all registered users in the system (used for admin/debug).
* **user(id)**  
  Fetches personal information (name, email, income, country, etc.) of a specific user by ID.

### Expense Queries

* **expenses(userId)**  
  Retrieves all expenses associated with a user, sorted by date. Used for listing and analytics.
* **expense(id)**  
  Fetches a specific expense by its unique ID.
* **dailyExpense(userId, targetDay)**  
  Summarizes the selected day’s total expense, compares it with the previous day, counts number of entries, and highlights the top spending category.
* **totalMonthlyExpense(userId, targetMonth)**  
  Calculates the total amount spent by a user in a given month.
* **categoryExpense(userId, targetMonth)**  
  Groups and totals monthly spending per category (e.g. FOOD, RENT, GAMES, etc.).
* **monthlyTotal(userId)**  
  Returns total monthly spending for the last 5 months, formatted for charting.
* **typicalSpent(userId)**  
  Calculates the average monthly spending over the past 3 months — used by the AI suggestion system to create realistic saving thresholds.

### Monthly Report Queries

* **monthlyReports(userId)**  
  Fetches all stored monthly reports for a user. Each report includes category breakdown, savings, top category, and comparison data.
* **monthlyReport(userId, targetMonth)**  
  Returns a single report for a specified month if available.

### Saving Goals Queries

* **savingGoal(userId, month)**  
  Retrieves the user’s monthly saving goal and associated thresholds if one has been defined.

## GraphQL Mutation Functionalities

The Kaneflow backend exposes a variety of GraphQL mutations to support user account management, expense CRUD operations, savings goal configuration, and intelligent financial suggestions. These mutations power all user interactions with write/update actions across the platform.

### User Mutations

* **register**  
  Registers a new user with name, email, password, income, and currency. Sends a verification email with a secure token.
* **login**  
  Authenticates a user using email and password. Issues a JWT token if the user is verified. Otherwise, triggers a re-verification email.
* **resendVerificationEmail**  
  Resends a verification link to unverified users if their original token expired or was never received.
* **resetPassword**  
  Sends a password reset email with a secure one-time token.
* **confirmResetPassword**  
  Confirms the password reset by validating the token and updating the user's password.
* **updateProfile**  
  Allows users to update personal info such as name, address, income, country, and currency.
* **deleteUser**  
  Permanently deletes the user account and all related data (expenses, reports, saving goals).

### Expense Mutations

* **addExpense**  
  Creates a new expense entry with name, amount, category, and date.
* **updateExpense**  
  Updates an existing expense’s fields by ID.
* **deleteExpense**  
  Deletes an expense entry by ID.

### Monthly Report Mutations

* **addMonthlyReport**  
  Generates a detailed monthly report based on current month expenses and compares with the average of previous months. Tracks top category, savings, and categorized breakdown.
* **updateAcknowledge**  
  Updates the acknowledged flag on a report to mark it as reviewed by the user.

### Savings Goals & AI Suggestions

* **suggestSavingGoals**  
  Uses past reports and current monthly income/saving target to generate AI-based spending thresholds per category. Returns adjusted suggestions following internal financial rules.
* **addSavingGoal**  
  Saves a new saving goal for the selected month, including total amount and per-category thresholds.
* **updateSavingGoal**  
  Edits an existing saving goal with updated savings amount or threshold values.
* **deleteSavingGoal**  
  Permanently removes a saving goal by ID.

# Component & Data Flow

Kaneflow follows a modular, data-driven architecture on the frontend, powered by Apollo Client for GraphQL data fetching and React for component-based rendering. The backend exposes a consistent GraphQL schema that supports a reactive and scalable UI experience.

## State Management

* **Apollo Client Cache** is the primary mechanism for managing application state and syncing frontend data with the GraphQL server.
* Caching ensures that previously fetched data (e.g., user info, expenses) is stored and reused efficiently across views, reducing unnecessary network calls.

## Data Fetching and Flow

Components use Apollo’s ***useQuery*** and ***useMutation*** hooks to fetch or mutate data. Query responses are automatically cached and invalidated when mutations occur.

### User Authentication Flow:

* Upon login, a JWT token is issued and stored in ***localStorage*.**
* The token is included in each request's header for authentication.
* Protected routes on the frontend validate token presence and redirect unauthorized users to login.

### Expense Entry Flow:

* Users submit an expense via form → triggers ***addExpense*** mutation.
* On success, Apollo cache updates and re-renders the expense list.
* Edit/Delete follows the same reactive pattern via mutation hooks.

### Savings & Suggestions Flow:

* Saving goals are created using the ***addSavingGoal*** mutation.
* The ***suggestSavingGoals*** mutation is triggered with user income and desired savings → AI backend generates dynamic thresholds per category.
* Suggestions are reviewed and optionally saved into a goal.

### Reporting Flow:

* Daily, monthly, and category-specific summaries are built using query responses like ***dailyExpense, monthlyTotal***, and ***monthlyReport***.
* Monthly reports are posted at the start of the following month using the ***generateReport*** function running a cron job.

## Data Flow (UML Sequence Diagram)

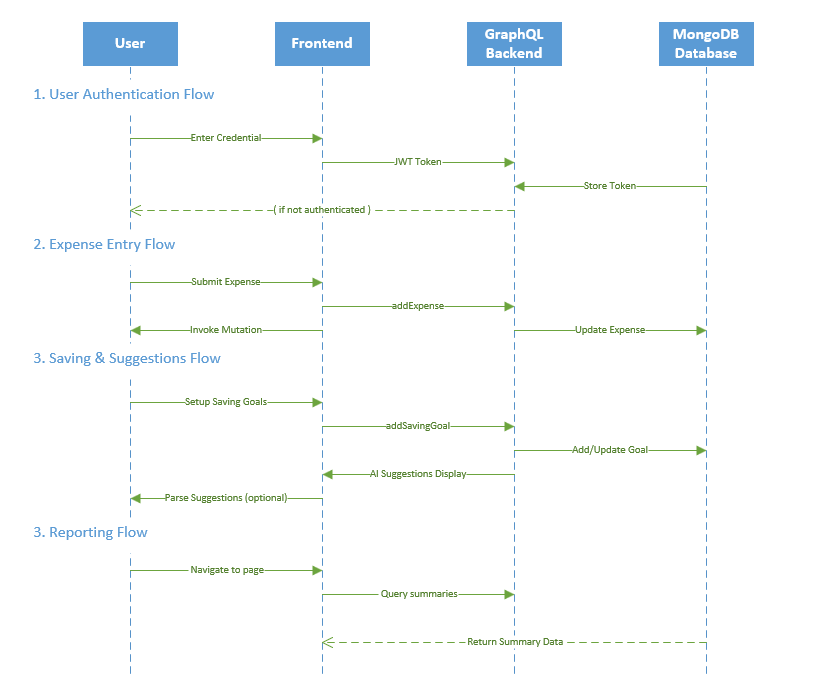


Figure: 1 – UML Sequence Diagram

## Component Hierarchy

* **App.jsx**

Root component with routing, global providers (Apollo, router), and layout wrapper.

* **Pages**
  + ***/Dashboard***: Displays summary cards, charts, and reports for the current month.
  + ***/Expenses***: Allows adding, editing, filtering daily expenses.
  + ***/SavingGoal***: Set and update monthly savings goals, thresholds, and suggestions.
  + ***/Savings***: Displays budget remainder analysis, categorical thresholds, and categoric spendings accordion.
  + ***/Profile***: Manage personal information and delete account.
  + ***/Login/Registration***: User authentication and validation page as well as navigation to resetting password.
  + ***/ResetPassword***: Dedicated page for password change.
* **Components**
  + ***ExpenseForm, ExpenseList, CategoryCharts, SavingProgress, CustomSelect, Footer, Icon, Navbar, PrivateRoute, Spinner***
  + Each is scoped and connected to the relevant GraphQL query or mutation.

## Auth & Routing

* React Router is used for navigation.
* Custom logic restricts access to authenticated routes.
* Token presence is checked on load; expired/invalid tokens redirect to login.

# Testing Strategy

Kaneflow uses automated and manual testing to ensure correctness, reliability, and maintainability across both backend and frontend layers. The focus is on verifying business logic, data flow integrity, and UI functionality.

## Backend Testing

All backend GraphQL APIs are covered through automated tests using **Jest** and **Supertest**. The tests are written to validate queries, mutations, database interactions, and error handling logic.

**Test Scope:**

* **Authentication & User Management**
  + Register new users and send verification email
  + Login and JWT issuance
  + Profile update and account deletion
* **Expense Management**
  + Add, update, and delete expenses
  + Fetch daily/monthly/category-based summaries
  + Handle edge cases (e.g. invalid user ID, no data)
* **Monthly Reports**
  + Generate monthly reports from expenses
  + Track savings, top categories, and comparisons
  + Acknowledge report status
* **Saving Goals & AI Suggestions**
  + Add/update/delete saving goals
  + Generate AI-based saving threshold suggestions
  + Validate category thresholds and logic paths

**Test Frameworks:**

|  |  |
| --- | --- |
| **Framework** | **Description** |
| Jest | Test runner and assertion library |
| Supertest | For sending GraphQL POST requests and simulating API interactions |
| Mongoose | Used for database setup/cleanup per test session |
| Mocking | AI suggestion logic is mocked to isolate and verify expected flow |

**Sample Test:**

* mutation ***addExpense*** → should return expense with correct amount
* query ***dailyExpense*** → should return 0 values for missing dates
* mutation ***suggestSavingGoals*** → should return fallback mock when no data

## Frontend Testing (Planned)

Although frontend tests are not yet implemented, the intended testing strategy includes –

**Planned Test Types:**

|  |  |  |
| --- | --- | --- |
| **Type** | **Tools** | **Coverage Goal** |
| Unit Tests | Jest + React Testing Library | Validate component logic and rendering |
| Integration Tests | Cypress / Playwright | Validate user workflows (e.g. expense flow) |
| Mocked Apollo Tests | MSW or Apollo MockedProvider | Simulate GraphQL responses and loading states |

**Test Coverage Goals:**

* Expense Form: input validation, mutation trigger, success/error handling
* Dashboard Cards: show correct total/monthly stats from mock data
* Saving Goal Editor: render thresholds, update saving amount
* Profile Form: update user info and render saved data
* Navbar + Routing: ensure protected routes redirect unauthenticated users
* Responsive Layout: test critical breakpoints and UI scaling

**Future Automation:**

* GitHub Actions CI pipeline for test runs on PR
* Code coverage threshold enforcement

# Deployment Process

Kaneflow follows a streamlined deployment pipeline using **Railway** for backend services and **Netlify** for the frontend interface. Both platforms are connected to GitHub repositories, enabling continuous deployment through simple git push actions.

## Backend Deployment (Railway)

The backend GraphQL server (Node.js + Apollo Server) is deployed using [**Railway**](https://railway.app/), a cloud platform that simplifies infrastructure and deployment.

**Setup:**

* Connected the backend GitHub repository to Railway.
* Configured all necessary **environment variables** (e.g. MONGO\_URI, JWT\_SECRET, PORT) through Railway’s dashboard.
* Railway automatically installs dependencies, builds, and runs the server on push.

**CI/CD Pipeline:**

* Branch deploys are supported.
* On main or production push:
  + Railway triggers a deployment.
  + Logs and errors are visible in the dashboard.
* Railway provides a secure HTTPS endpoint for the live GraphQL API.

## Frontend Deployment (Netlify)

The React-based frontend is hosted on [**Netlify**](https://netlify.com/), which offers continuous deployment, global CDN, and performance monitoring out-of-the-box.

**Setup:**

* Linked the frontend GitHub repo to Netlify.
* Build command: npm run build
* Publish directory: dist/ (configured for Vite)
* Environment variable for VITE\_GRAPHQL\_ENDPOINT is defined in the Netlify dashboard.
* Netlify automatically builds and deploys on push to the main branch.

**Features:**

* Live Preview Deploys for PRs (optional).
* HTTPS, compression, and auto-refresh enabled.
* Rollbacks available via Netlify’s deployment history.

# References & Resources

* [GraphQL Documentation](https://graphql.org)
* [Apollo Client](https://www.apollographql.com/docs/react/)
* <https://www.proglobalbusinesssolutions.com/>
* <https://www.svgviewer.dev/>
* <https://www.svgrepo.com/>