ChatGPT

Comprehensive LiqLearns Platform Report

Introduction and Context

LiqLearns is an ambitious educational platform built with a modern web stack. The front-end uses **React 18** with **TypeScript**, **Redux Toolkit** for state management, **React Router v6** for routing, **TailwindCSS** for styling, and assorted UI libraries like **Lucide** icons. The back-end leverages **Supabase** for its PostgreSQL database, authentication, row-level security (RLS) and real-time services. Originally the project included **Stripe** for paid subscriptions, but this integration is currently being removed. Throughout development the team also utilised **Supabase Edge Functions** for serverless custom logic, and the site is deployed on **Vercel**.

The project repository is organised as follows:

Folder/File	Description
`src/components/`	Reusable components like buttons, cards, modals, and icons.
`src/pages/`	Page components grouped by feature (dashboards, marketplace, events, etc.).
`src/contexts/`	React contexts (e.g. `AuthContext.tsx`) for authentication and global state.
`src/services/`	Service clients such as `supabaseClient.ts` and API helpers.
`supabase/`	SQL migrations, triggers, policies and edge functions.
`.env`	Environment variables (not committed).
`public/`	Static assets and `index.html`.

During a series of interactive debugging sessions, multiple roles were tested: **Student**, **Teacher**, **Admin**, and **CEO**. These sessions revealed functional gaps, UI inconsistencies, and architectural shortcomings. The following pages synthesise those findings and provide a holistic examination of each feature and the underlying code. To make this report self-contained, the analysis is grouped by major feature area, with code insights and suggested fixes.

---

1. Student Experience

1.1 Dashboard Overview

Upon logging in as a student, users land on the **Student Dashboard** (`src/pages/role-based-dashboard-hub/student/StudentDashboard.tsx`). The dashboard is intended to be the learner's command centre. It displays:

- **Total Lessons**, **Total Badges**, **Total XP**, **Total Certificates** and **Current Streak** in summary cards. Each card is meant to open a detailed modal when clicked. During testing these modals were blank because the data fetch returned null and there was no fallback. This indicates either a missing API endpoint or a failure to pass the current user ID to the query. A proper loading state and error message should be added instead of rendering an empty modal.

- **Today's Quest**, a checklist of daily activities (e.g., "Complete daily quiz", "Watch a video lesson"). Completing tasks should reward XP or streak points. In the current implementation the tasks list appears but there is no persistent state to track completion; marking an item as done does not update the backend. A `student_quests` table and corresponding API should record completion.

- **Study Rooms** quick entry – a button in the sidebar leads to the study rooms area (see Section 2 below).

1.2 Learning Paths and Quest Page

The **Quest** page under the student role allows learners to select a learning path (Amharic Language, Ethiopian Culture, Mathematics, Science, English, Technology & Coding). Selecting a path opens a side panel listing live classes and course materials. The panel includes:

1. **Live Classes**: Scheduled sessions with start dates and times. Each entry offers a "Set Reminder" button. Currently this button does not interact with any calendar API; it simply logs to console. To make this useful, integrate with a calendar service or send a notification via Supabase functions.

2. **Course Materials**: This section should display PDFs, videos, audio files, and quizzes. However, it currently shows an error: "Course not found in database". The root cause appears to be that the `courses` table is empty or the front-end is passing an incorrect slug. Seeding the database with real courses and adjusting the query to use dynamic IDs will resolve this error.

3. **Learning Tools**: Tabs for audio, video, notes, assignments and movies. None of these tabs load content. The UI highlights the selected tab, but there is no data fetch. This indicates the API calls (likely through `supabase.from('materials')`) were never implemented.

1.3 Marketplace Integration

Initially the marketplace existed as a separate page. Through user feedback it was decided to embed the **Marketplace** into the student dashboard to encourage discovery. The marketplace allows students to purchase or download items such as books, videos, audio guides, games, flashcards and worksheets. In testing, several issues were noted:

- The category counts (e.g. “Books 45”) do not match the actual number of items shown (only 4 items appear). This is because the counts are hard-coded in the component rather than being computed from the database. To fix this, query `marketplace_items` with `supabase.from('marketplace_items').select('*', { count: 'exact' })` grouped by category and display the returned counts.

- Some categories like “Games” or “Flashcards” display “No items available” despite the counts suggesting otherwise. This is again due to mismatched seeds or query filters.

- A new **Tagging System** was proposed: instead of a single `category` field, each item has a `tags` array. Tags include `interactive`, `template`, `tutorial`, `lecture`, `reference`, `spaced-repetition`, `gamified`, `memory`, `audio`, `video`, `books`, `flashcards`, `games`. SQL migrations were prepared to add a `tags` column and update existing rows. A custom `MarketplaceIcon` component renders appropriate icons based on tags. This tag-driven design allows multi-dimensional filtering and should replace the rigid categories.

- Items can be added to a cart. When adding free items the cart shows “0 points” and a “Proceed to Checkout” button. The checkout integration with Stripe is being removed; until payment is re-implemented, clicking checkout should either show a modal explaining that payment is disabled or simply allow downloading free resources.

1.4 Study Rooms

Study rooms are community spaces where students can join real-time sessions. The **Study Rooms** page (`src/pages/study-rooms/index.tsx`) lists available public rooms (e.g. “Language Practice Room”). Issues observed include:

- Clicking **Join Room** triggers a dark overlay with the error “Failed to join room”. This is likely due to missing RLS policies on the `study_rooms` or `study_room_members` tables. To fix, ensure that RLS policies allow authenticated students to insert a membership record into the `room_members` table where `student_id = auth.uid()` and to select any room with `is_public = true`.

- **Create Room** button does nothing. It should open a modal or page with a form to enter room name, description and maximum capacity. After submission, the front-end should call `supabase.from('study_rooms').insert(...)`. A row-level policy should permit students to insert new rows and become the host.

1.5 Events

Students can navigate to an **Events** page where a calendar is displayed and a **Create Event** button opens a modal. The modal asks for title, description, start and end time, type and a “make event public” checkbox. The date/time pickers do not display; clicking the fields does nothing. This is due to missing imports for the date picker library (likely `@headlessui/react` or `react-date-picker`) and CSS for the picker.

Importing the component and adding the CSS will enable proper date selection. In addition, the event should be created via `supabase.from('events').insert(...)` and optionally broadcast via real-time for other attendees.

1.6 Community and Help

The **Community** section shows group chats and a wall for posts. Clicking on a group does nothing – chat rooms are not implemented. A `chat_rooms` and `messages` tables should be added, along with a WebSocket (via Supabase Realtime) to send messages. The "Create Group" button should open a modal to define group name and participants.

The **Help** page lists FAQs. A **Submit a Ticket** button opens a form requiring title, category, priority and description. The "Create Ticket" button stays disabled until all fields are filled. Submitted tickets should be inserted into a `support_tickets` table, and responses should appear in a conversation thread. A **My Tickets** tab lists previous tickets; clicking a ticket opens a conversation view. The send button should call an edge function that inserts the message into `ticket_messages` table.

1.7 Settings and Profile

Under Settings, students can modify personal information, password, language, privacy settings, GDPR preferences and learning goals. Each tab corresponds to a separate component. For example, the **Learning Goals** tab presents a checklist of skills (e.g. vocabulary, grammar, speaking). Updates should be written to `student_profiles` via Supabase. In testing, no changes were saved. Implement `onSubmit` handlers and call `supabase.from('student_profiles').update()` accordingly.

---

2. Teacher Experience

2.1 Teacher Dashboard and Classes

Logging in as a teacher takes you to **TeacherDashboard** (`src/pages/role-based-dashboard-hub/teacher/TeacherDashboard.tsx`). This page shows metrics for **Total Students**, **Active Classes**, **Assignments Graded**, and **Upcoming Sessions**. Buttons like **Manage Class** and **Create Class** appear. In practice, they do nothing because the event handlers are undefined. Fixes include:

1. **Manage Class**: On click, navigate to a dedicated class management page (`/teacher/classes/[classId]`) showing rosters, grades, and materials. The route file (`src/Routes.tsx`) must define this path.

2. **Create Class**: Show a modal with fields for name, description, schedule and capacity. Insert new row into `classes` table and assign teacher as instructor.

3. **Upcoming Sessions**: This section should fetch from an `sessions` table where `teacher_id = auth.uid()` and display date/time. Without this query, it remains static.

2.2 Student Management

The **Students** page lists all students in a teacher's classes with progress bars and statuses (active/inactive). Buttons like **View Profile** and **Add Student** do nothing. Implementation should:

- Connect the **View Profile** button to navigate to `/teacher/student/[studentId]` which displays the student's details and performance.
- Use an **Add Student** button to open a modal where the teacher can invite a student via email. The backend should insert a record in `enrollments` linking the student to the class.

2.3 Content Library and Store

Teachers can manage course content and sell supplementary materials. In the current build:

- **Content Library** displays uploaded documents but the **Edit**, **Share**, and **Upload Content** buttons are inert. Proper handlers must call Supabase Storage to upload files and update `teacher_content` table with metadata.

- **Store** allows teachers to upload products (notes, templates). The **Add Product** and **Edit/Delete** buttons have no effect. Implementation should connect these buttons to the `marketplace_items` table using Supabase calls. Teacher products should be flagged with `teacher_id` and visible only to students enrolled in their classes.

2.4 Reports, Schedule and Help

The **Reports** page summarises performance metrics (average class performance, attendance rate). Exporting to PDF or CSV should call a serverless function (e.g. using `pdf-lib`) but currently does nothing. Implement an `exportReports` handler that collects data and triggers a file download.

Teachers also have a **Schedule** calendar with the ability to add sessions. The Add Session button doesn't work because the date-picker and the `onAddSession` callback were never wired. As with events, import a date picker component and implement insertion into `sessions` table.

2.5 Teacher Settings and Help

Settings are similar to students but include additional toggles for notifications and privacy. The Help tab lists FAQs but the questions do not expand, likely due to missing `onClick` toggles or collapsed content CSS. This can be fixed by ensuring that each question component maintains its own expanded state.

---

3. Admin Experience

3.1 Overview and User Management

The **Admin** role (log in with `admin@liqlearns.com`) presents metrics for total users, students, teachers, etc. It is primarily designed to manage content and users:

- **Users** tab lists all user accounts. However, the rows are static; there is no ability to view or edit user details. To implement this, add actions to each row (e.g., “View”, “Ban”, “Reset Password”). Use Supabase queries to update `auth.users` and `user_profiles` accordingly.

- **Content Management** link is present in the top nav. Clicking it currently leads to a 404 page because the route is missing. A new page (`src/pages/content-management-hub/index.tsx`) should be created with tools for uploading, approving and categorising content across the platform. Then update `Routes.tsx` to map the path.

3.2 Financial and Analytics

Admin may also access financial data (subscriptions revenue, sales from marketplace) and analytics (user growth, engagement metrics). These pages either load static placeholders or are entirely missing. Implementation would require queries against `subscriptions`, `payments`, and user metrics tables and proper charts (e.g. via Recharts). A `FinancialDashboard.tsx` should display monthly revenue, churn rate, and top selling items.

3.3 Settings and Security

Admins should control platform settings: roles, permissions, content approvals, compliance. A **Security & Compliance Management** page was proposed. It should display open security tasks (like verifying Stripe webhook signatures, adding CSP headers), monitor real-time alerts, and provide toggles to enable/disable features like sign-up or event creation. This page can leverage `supabase` to fetch and update flags stored in an `admin_settings` table.

---

4. CEO Experience

The **CEO** role (login `ceo@liqlearns.com`) sees high-level metrics: total revenue, user growth, teacher count, and marketing statistics. There is a section for a **Landing Page Demo Video** with an “Edit URL” button, and **Landing Page Statistics** showing views, sign-ups, and conversion rates. Additional sections (Business Analytics, Financial Overview) are placeholders. To make this meaningful:

- Integrate charts that query aggregated data from Supabase (`payments` table for revenue, `users` table for growth). Use `supabase.rpc()` functions to precompute metrics for improved performance.

- Provide a content editor for the landing page (e.g. update hero text, CTA copy) and a video uploader. This can be built with a WYSIWYG editor and a `landing_pages` table to store content.

- Implement segmentation filters (e.g. by time period, by marketing channel). Use Supabase or external analytics tools to record UTM parameters and conversion events.

---

5. Support Experience

While not deeply tested, the **Support** role (login `support@liqlearns.com`) is meant to manage help tickets and answer user questions. The `Help` page under other roles interacts with this by creating tickets. In the admin dashboard, support agents should see a queue of open tickets, respond through a chat interface, and assign statuses (open, in progress, resolved). A `support_messages` table should record each communication. Additional features like macro responses, tagging, and analytics (average resolution time) would make the support workflow robust.

---

6. Authentication and Sign-Up Flow

6.1 Login Form

The login page lives in `src/pages/login/index.tsx`. It toggles between a **LoginForm** and a multi-step **Sign-Up Wizard**. The login form uses `supabase.auth.signInWithPassword` to authenticate. On success it stores the session in context and navigates to `/role-based-dashboard-hub`. If there's an error (invalid email/password), an error message is displayed. One improvement is to implement rate limiting on login attempts (e.g., lock out after 5 failed attempts) to prevent brute force attacks.

6.2 Sign-Up Wizard

The sign-up flow originally spanned nine steps: role selection, personal info, application form (for teachers), general questions, phone/email verification, subscription selection, policy agreements, review, and approval confirmation. Over time the payment step (via Stripe) and subscription selection were removed. The wizard still retains extraneous components such as `SubscriptionStep.tsx` and `PaymentStep.tsx`; these are no longer referenced but should be deleted to avoid confusion.

The recommended simplified flow is:

1. **Role Selection** – choose Student, Teacher, Support or Admin. The `role` value is stored in local state.
2. **Basic Info** – enter full name, username, email, password and sponsor username (optional). Use debounced validation to check if username is unique and if sponsor exists. Supabase edge functions `check-username` and `check-sponsor` were created for this purpose. They query `user_profiles` to ensure uniqueness and enforce rules (e.g., only students and teachers can sponsor; admin cannot be a sponsor). Debouncing prevents API thrashing as users type.
3. **Contact Verification** – send OTP to email or phone. Instead of blocking progression until verification, allow users to continue and verify later via a link in the welcome email.
4. **Policy Agreement** – present terms of service, privacy policy and honour code with checkboxes. Users must acknowledge to proceed.

5. **Completion** – call `supabase.auth.signUp` with the email and password. After sign up, insert a row into `user_profiles` with fields `id` (matching `auth.users.id`), `username`, `full_name`, `sponsor_username`, `role` and metadata. Next, insert a row into `student_profiles` (if role is student) with `subscription_plan = 'free_trial'` and `trial_end_date` set to 14 days from now. Use `supabase.from('student_profiles').insert(...)`. For teachers, insert into `teacher_profiles`. Finally, redirect to the relevant dashboard.

6.3 Stripe Removal

The legacy code imported `createStripeCustomer` from `services/stripeService` and called it inside the `signUp` function. The current plan is to remove all references to Stripe. To do so:

1. **Comment out or delete** the import of `createStripeCustomer` in `AuthContext.tsx`.
2. **Comment out** the call to `createStripeCustomer(data.user.id, email, fullName)` and the associated console logs.
3. **Add** insertion logic for `user_profiles` and `student_profiles` as described above.
4. Clean up environment variables – remove `VITE_STRIPE_PUBLIC_KEY` and `VITE_STRIPE_PRICE_ID` from `.env`. Remove unused components like `SubscriptionStep` and `PaymentStep`.

6.4 Security Considerations

Several security improvements were discussed:

- **Rate Limiting** – login and sign-up endpoints should be rate limited. Supabase does not provide built-in rate limiting, so a middleware using Edge Functions (e.g. checking number of attempts per IP) can be deployed. Alternatively, use a third-party like Cloudflare Turnstile.

- **Two-Factor Authentication** – the current implementation uses email or phone OTP in a basic way. Integrate a robust 2FA library or Supabase's built-in multi-factor authentication to secure admin and teacher accounts.

- **Role-Based Access Control** – currently enforced on the client. RLS policies must ensure that only appropriate roles can read or write certain tables. For example, teachers should not read `admin_settings` and students should not insert into `subscriptions`.

- **Secret Management** – environment variables should not be hard coded. Provide an `.env.example` file and ensure `JWT_SECRET`, `STRIPE_WEBHOOK_SECRET` and other sensitive keys are loaded from environment.

---

7. Supabase Database and RLS Policies

7.1 Key Tables

The following tables underpin LiqLearns:

Table	Purpose	Key Columns
`auth.users` (Supabase built-in)	Authentication table storing email and hashed password.	`id`, `email`, `encrypted_password`
`user_profiles`	Main user profile table storing username, full name, role, sponsor, and contact info.	`id` (FK to `auth.users.id`), `username`, `full_name`, `role`, `sponsor_username`, `phone`, `subscription_plan`
`student_profiles`	Extends user profile for students; stores subscription info, XP, streak, aura points.	`id` (PK, FK), `subscription_plan`, `trial_end_date`, `xp`, `gold`, `streak`, `level`, `aura_points`
`teacher_profiles`	Similar to student profile but for teachers; stores rating, earnings, etc.	`id`, `rating`, `total_earnings`
`courses`	Stores courses.	`id`, `title`, `description`, `creator_id`, `lesson_type`
`marketplace_items`	Items available for purchase or free download.	`id`, `title`, `description`, `price`, `tags` (array), `download_url`, `points_cost`
`study_rooms`	Metadata about public/private study rooms.	`id`, `name`, `description`, `is_public`, `host_id`
`study_room_members`	Many-to-many linking users to rooms.	`room_id`, `user_id`
`events`	Student and teacher events.	`id`, `title`, `description`, `start_time`, `end_time`, `type`, `host_id`
`support_tickets`	Help centre tickets.	`id`, `creator_id`, `title`, `category`, `priority`, `status`
`ticket_messages`	Chat messages between support and users.	`id`, `ticket_id`, `sender_id`, `message`, `timestamp`

Table	Purpose	Key Columns
`quizzes`, `lessons`, `assignments`	Additional learning content.	…

7.2 RLS Policies

Supabase's Row-Level Security is key to ensuring that each role sees only what it should. The following are examples of policies and the issues discovered:

- **student_profiles**: The policy `students_manage_own_student_profiles` allows authenticated users to insert/update their own row if `id = auth.uid()`. Another policy `admins_view_all_student_profiles` allows admins to select all rows. This is generally correct.

- **study_rooms**: Policies were set for students to create rooms and update their own rooms. However, no `select` policy existed to allow reading public rooms. A correct policy would be:

```sql
CREATE POLICY students_view_public_rooms ON study_rooms
  FOR SELECT
  USING (is_public = true);
```

Without this, the query returns zero rows, causing the join failure.

- **study_room_members**: There was no policy enabling insertion into this join table, preventing users from joining rooms. The fix is:

```sql
CREATE POLICY students_join_rooms ON study_room_members
  FOR INSERT
  WITH CHECK ( user_id = auth.uid() );
```

- **user_profiles**: Initially only administrators could select other users because a misconfigured policy filtered everything. To allow everyone to view public profiles (e.g. for sponsor checks), a policy like `SELECT USING (true)` may be appropriate, or at least `role IN ('admin','student','teacher')`.

- **marketplace_items**: Items should be publicly readable (`SELECT USING (true)`), but only admins or the seller (teacher) should be able to `INSERT` or `UPDATE` (e.g. `WITH CHECK (auth.role() = 'admin' OR teacher_id = auth.uid())`).

Whenever a new feature is added (e.g. `events`, `support_tickets`), remember to create appropriate RLS policies; otherwise queries return 0 rows and the UI appears broken.

8. Real-Time and Performance Considerations

8.1 Real-Time Stats vs. Polling

The project attempted to display real-time stats (XP, streak, aura points) using Supabase Realtime. However, due to restrictive Content-Security-Policy settings, WebSocket connections were blocked, resulting in repeated “Realtime connection failed” logs. Debugging steps included:

1. **CSP Headers**: `vercel.json` or an `_headers` file must specify `Content-Security-Policy` that allows `wss://*.supabase.co` and `https://*.supabase.co`. Without this, browsers will block WebSocket connections. Similarly, `connect-src` should include `https://api.stripe.com` if Stripe is used.

2. **Realtime Enabled**: In the Supabase dashboard, ensure “Realtime” is turned on for the tables you want to subscribe to (e.g. `student_profiles`). Without enabling, subscription events will never fire.

3. **Environment Variables**: In production deployments on Vercel, ensure `VITE_SUPABASE_URL` and `VITE_SUPABASE_ANON_KEY` are correctly set. If these differ between preview and production, Realtime may silently fail.

Given these complexities and the limited number of stat updates, a simpler solution is to poll an Edge Function or a `student_stats` view every few seconds. An example Edge Function (`supabase/functions/student-stats`) could accept a user ID, fetch the row from `student_profiles`, return a JSON object of XP, gold, aura points and streak, and ensure a row exists by inserting default values if none. The React component can call this function using `supabase.functions.invoke('student-stats', { headers: { Authorization:` Bearer ${session.access_token}`} })` inside a `useEffect` with interval. This avoids WebSocket restrictions and still provides near-real-time updates.

8.2 Removing Mock Data

At times the StudentDashboard used fallback mock data such as `stats ?? mockStats` if the API call failed. This masked errors and allowed the UI to render seemingly correct values even when the backend failed. All mock/fallback references should be removed. Instead, display a loading state while fetching data and an error state if the query fails. This encourages developers to fix the actual API rather than hiding the problem.

8.3 Performance Improvements

The site audit identified several performance bottlenecks:

- **Bundle Size**: Large libraries like Recharts and Moment.js were included. Replace Moment.js with native `Date` or `date-fns` (already a dependency) and use lighter chart libraries or dynamic imports to reduce the initial bundle.
- **Image Optimisation**: Use `<img loading="lazy">` and size attributes so images don’t block initial render. Serve images through a CDN (Supabase Storage or Vercel) with compression.

- **Prefetching**: Use `link rel="prefetch"` in `index.html` to prefetch critical resources like fonts or other route bundles.
- **Caching**: Introduce a global `APP_VERSION` constant to bust caches when deploying new versions. Append `?v=${APP_VERSION}` to script and style URLs.

---

9. User Interface and Design Enhancements

9.1 Hero Section and Landing Page

Early feedback focused heavily on the **Hero Section** of the landing page. The user wanted a warmer, light orange gradient behind the heading and icons to better reflect the LiqLearns brand. After several revisions, the gradient was set to a soft orange (#fca311) fading into white, with the text left aligned and call-to-action buttons clearly visible on small screens. To ensure the hero did not overflow on mobile devices, the height class `h-screen` was replaced with `min-h-[calc(100vh-4rem)]` and `flex-col` layout to allow content to wrap.

The CTA buttons were updated to use gradient backgrounds on hover and improved contrast ratios for accessibility. A new **PricingCard** component was created to display subscription plans (although subscription selection was later removed) and uses icons to highlight plan features. The card supports a "Most Popular" ribbon and scales gracefully from mobile to desktop.

9.2 Accessibility and Colour Contrast

The site audit noted several **accessibility** violations: icon-only buttons lacked `aria-label`s, some text colours failed contrast ratios (e.g. badges with orange text on white), and focus outlines were removed globally by Tailwind preflight. Fixes implemented include:

- Adding `aria-label` attributes to all icon buttons (e.g. hearts, message icons).
- Adjusting colours using Tailwind's `text-orange-600` for better contrast or darkening grey text by 30 % to meet WCAG 2.1 AA guidelines.
- Re-enabling the focus ring by adding `focus-visible:outline` classes and removing the global `* { outline: none; }`.
- Providing closed captions for video lessons by including a `.vtt` file and setting `<track kind="captions" src="..." />` inside the video player.

9.3 Mobile Responsiveness and Overflow

Many pages used `h-screen` or fixed heights that caused content to overflow on small screens, hiding CTA buttons. Replacing these with `min-h-screen` or dynamic heights (e.g. `min-h-[calc(100vh-4rem)]`) and using `flex-col` layouts ensures that content flows vertically. For long forms (like sign-up), using a sticky progress indicator at the top helps users track their progress and prevents important buttons from being scrolled out of view.

---

10. Gamification, Questing and Leaderboards

10.1 Gamification Layer

To make learning engaging, a gamification layer was proposed. Key components include:

- **XP and Gold**: Completing lessons, quizzes and quests grants XP and gold. XP determines level; gold can be spent in the marketplace.
- **Daily Streaks**: Logging in and completing tasks daily increases your streak. A current streak counter is displayed on the dashboard. When the streak ends, an animation plays (e.g. fireworks) to celebrate the milestone.
- **Badges and Achievements**: Earn badges for reaching milestones (e.g. 100 XP, 30-day streak). A `badge_profiles` table and triggers maintain badge ownership. Badges appear in the student profile.
- **Leaderboards**: A leaderboard shows top students by XP or number of recruits (for multi-level marketing). The query uses a materialized view joining `user_profiles` with a `mlm_network` table to compute downline counts. The leaderboard can be filtered by period (daily/weekly/all-time) and by metric (XP/gold/streak).
- **Quest System**: Daily quests (e.g. complete 5 flashcards) are generated via an edge function `generate-daily-quests`. The function ensures variety by selecting different activity types and awarding random XP/gold. A `quests` table stores quest templates and a `daily_quests` table stores assignments per user per day. Users can claim rewards via an API call that updates their XP and marks the quest as complete.

Implementing these features requires additional tables (e.g. `user_activities`) and triggers to increment XP and streak upon completion of tasks. RLS policies must ensure only the rightful user can update their own stats.

10.2 Multi-Level Marketing (MLM) Features

The platform has a sponsorship model where users can specify a **sponsor** upon sign-up. A `mlm_network` table stores relationships: `sponsor_id` and `child_id`. This network allows referral bonuses and a leaderboard of top sponsors. Rules were defined via an edge function `check-sponsor` to enforce that admin/support roles cannot be sponsors. A `check-username` edge function ensures uniqueness of usernames. The sign-up form uses `apiClient` with a debounced `useEffect` to call these functions and show real-time validation messages.

---

11. Security, Privacy and Compliance

11.1 Environment Management and Secrets

One of the earliest tasks was creating a `.env.example` file. This file lists environment variables (e.g. `VITE_SUPABASE_URL`, `VITE_SUPABASE_ANON_KEY`, `JWT_SECRET`, `STRIPE_WEBHOOK_SECRET`) but contains placeholder values. Team members can copy this to `.env` and insert real keys. Hard-coded secrets in the codebase must be removed. In CI/CD, the environment is loaded from Vercel’s Secrets.

11.2 Content Security Policy (CSP)

The application lacked proper security headers. Adding a `vercel.json` or `_headers` file with a strict `Content-Security-Policy` mitigates XSS and clickjacking. A recommended CSP includes:

```
default-src 'self';
script-src 'self' https://cdn.jsdelivr.net;
style-src 'self' 'unsafe-inline';
img-src 'self' data: https://avatars.githubusercontent.com https://
*.supabase.co;
connect-src 'self' https://*.supabase.co wss://*.supabase.co;
frame-ancestors 'none';
base-uri 'none';
```

This allows Supabase API and WebSocket connections while blocking other domains. Additional domains like Stripe or analytics providers can be appended as needed.

11.3 GDPR, Cookie Consent and Account Deletion

The platform must comply with GDPR and other privacy laws. An early audit recommended:

- Displaying a **cookie consent banner** on first visit, explaining which cookies are used and allowing opt-in/out. Accepting stores the preference in local storage.
- Creating a **data export** tool where users can download all data associated with their account (from `user_profiles`, `student_profiles`, `user_activities`, etc.). A Supabase edge function can gather data and return a JSON file.
- Implementing an **account deletion** flow: from the settings page, users can request deletion. This triggers a confirmation email. On confirmation, an edge function deletes the user from `auth.users` and cascades to related tables (using `ON DELETE CASCADE`).

11.4 Security & Compliance Management Centre

For administrators, a dedicated page summarises security tasks: verifying Stripe webhook signatures, ensuring JWT secrets rotate periodically, auditing RLS policies, and monitoring rate limiting. Each task appears as a checklist item with status (open, in progress, resolved). Implementing this reduces the risk of leaving critical vulnerabilities unresolved.

---

12. Developer Experience and Testing

12.1 Code Organisation and Conventions

The codebase mixes components in `pages` and `components`. A proposed organisation is:

- `src/components/` for generic UI elements (buttons, form fields, cards, modal wrappers).
- `src/features/` for domain-specific logic (e.g. `auth`, `marketplace`, `gamification`).

- `src/pages/` for route-level components only.
- `src/contexts/` for global state providers (auth, theme, settings).

Naming conventions should be consistent (e.g. `StudentDashboard`, `TeacherDashboard`, `AdminDashboard` rather than mixing PascalCase and camelCase). Logs should use a unified logger instead of scattered `console.log` and `console.error`. Prettier and ESLint should be configured to enforce code style.

12.2 Testing and CI

Currently there is no automated test suite. Adding **unit tests** with Jest and **integration tests** with React Testing Library will catch UI regressions. For critical flows (sign-up, login, marketplace purchase), consider end-to-end tests using Cypress or Playwright. A **CI pipeline** (GitHub Actions) should run tests on every pull request, lint the code, and optionally run **Lighthouse CI** to track performance metrics (FCP, TTI). Add a `coverage` badge to the README to encourage comprehensive tests.

12.3 Supabase Migrations and Edge Functions

The `supabase` directory contains SQL migrations (e.g. `20260105154700_add_user_activities_table.sql`) and TypeScript edge functions. Use `supabase db push` to apply migrations locally and `supabase functions deploy` to deploy edge functions. Keep migrations atomic and idempotent – each file should modify one aspect (adding a table, adding RLS, seeding data). Use semantic versioning in filenames (YYYYMMDDHHMMSS). Document functions and triggers in a `SUPABASE_SCHEMA.md` so new team members can understand the database.

Conclusion

LiqLearns is a feature-rich platform with a broad vision: real-time study rooms, an integrated marketplace, gamified learning journeys, multi-level marketing referrals, administrative dashboards and support tools. Through the debugging sessions, we uncovered numerous issues – from simple UI bugs (blank modals, unresponsive buttons) to deeper architectural flaws (broken RLS policies, missing API endpoints) and security concerns (hard-coded secrets, missing CSP). The report summarises every feature, the associated files, and the recommended fixes.

Key recommendations moving forward:

1. **Simplify sign-up** – finalise the removal of Stripe and ensure proper insertion of profile rows with a free trial. Validate username and sponsor via edge functions.
2. **Seed the database** – populate `courses`, `marketplace_items`, and `quests` with real content. Ensure RLS policies allow read access.
3. **Implement missing handlers** – connect UI buttons to Supabase calls for study rooms, events, teacher content, and admin management. Add error handling and loading states throughout.
4. **Fix real-time** – adjust CSP, enable Realtime on needed tables, or use polling functions for stats. Remove mock fallbacks.
5. **Complete security tasks** – supply `.env.example`, add CSP and HSTS headers, rate-limit auth flows, implement GDPR and account deletion.

6. **Invest in developer tooling** – set up automated tests, CI, and consistent code formatting. Document the Supabase schema and edge functions.

Once these improvements are in place, LiqLearns will provide a robust, secure and engaging learning experience for all user roles.
