ChatGPT

Software Development Life Cycle (SDLC) for RCS Golf Tournament PWA

1. Requirements Gathering and Analysis

In this phase, the goal is to clearly understand what the **RCS** golf tournament application must do and what constraints it operates under. The primary objective of requirements analysis is to ensure the software meets user needs [1]. The team should gather detailed requirements from stakeholders (tournament organizers, players, etc.) and translate them into technical specifications. Below is a summary of the key requirements based on the user's input and goals:

- **Functional Requirements:**
- **User Authentication:** The app must support a **lightweight login** using a username and a 4-digit PIN (instead of traditional email/password). Admins will create user accounts and assign PINs for players, who will use these credentials to log in [2]. Guests (unauthenticated users) should have read-only access to public data (e.g. viewing tournament leaderboard or schedule).
- **User Roles and Permissions:** There will be at least three roles – **Admin**, **Player**, and **Guest**. **Admins** can create tournaments, manage all data (players, teams, matches, scores), and configure settings. **Players** can log in to view their schedules, enter or confirm scores, and see results. **Guests** can view public information (like tournament standings) without editing. The system must enforce these permissions both in the UI and at the database level (using Supabase's security features).
- **Tournament Management:** The app must allow admins to create and configure golf tournaments. This includes defining the tournament details (name, date, location, format), registering players (or teams), and organizing them into matches or rounds.
- **Team and Player Management:** The system should track **players** (with their details) and optionally **teams**. Admins can assign players to teams if the tournament format requires it. Teams and players can be associated with specific tournaments (e.g. a player can participate in multiple tournaments).
- **Match Scheduling:** The app needs to manage **matches** within a tournament. For example, pairing players or teams for match-play, or grouping players for stroke-play rounds. Admins should be able to schedule matches (set tee times, assign players/teams to matches) and the schedule should be visible to players.
- **Score Entry and Tracking:** During matches, scores need to be recorded. Players (or admins) should input scores (e.g. strokes per hole, or match play results) for each match. The system should calculate results (e.g. total strokes, match winners) and update leaderboards or brackets.
- **Real-Time Updates:** As a live event application, RCS should reflect score updates in real-time. When a score is entered or a match result is recorded, all connected users should see the updated leaderboard or match status instantly (using Supabase's real-time subscriptions). This is crucial for an engaging live scoreboard experience.

- **Progressive Web App (PWA) Features:** The application should behave like a native app in terms of responsiveness, offline capability, and installability. A Progressive Web App is built with web technologies but offers features like offline availability and the option to **"install"** the app on mobile

devices [3]. For RCS, this means it should load quickly on smartphones, work in areas with spotty internet (by caching critical assets and possibly allowing offline score entry to sync later), and be installable to a home screen.

- **Non-Functional Requirements:**

- **Usability:** The login with username and PIN should be extremely simple (especially since 4-digit PINs are easy to enter on mobile). The UI must be clean and easy to navigate on the golf course (likely on a phone screen). Short, clear instructions and minimal required input are ideal for users who may be less tech-savvy.
- **Performance:** The app should handle real-time updates efficiently. Using Supabase's real-time service, updates should propagate to clients within seconds. The design should minimize unnecessary network calls (e.g. use fine-grained subscriptions or polling for only relevant data) to preserve battery and bandwidth during tournaments.
- **Security:** Even with a simplified auth model, security is important. PINs are weak credentials, so additional measures should limit abuse: e.g. rate-limit login attempts and encourage admins to use unique (not easily guessable) usernames. All network communication must be over HTTPS for privacy. Supabase's Row-Level Security (RLS) will ensure users can only access or modify data they're authorized to [4] [5]. RLS acts as a **"defense in depth"** mechanism in the database, protecting data even if the client is exposed [6].
- **Reliability:** The system should be stable during live events. We should have a plan for backups (especially of the database) and a way to recover if something goes wrong (for example, an admin override if the real-time sync fails and scores need re-entering).
- **Compatibility:** As a web app, it should work across modern browsers and devices. Ensuring compatibility with Chrome, Safari (often on iPhones), etc., is necessary. Using SvelteKit's PWA approach will cover most modern browsers.
- **Maintainability:** The codebase should be organized and documented so that future updates (like new tournament formats or features) can be integrated without breaking existing functionality. Following a clear SDLC ensures we set a solid foundation for maintainability.

By thoroughly documenting these requirements and analyzing any constraints or risks now, we set a solid foundation for the later phases of development [7]. For example, knowing upfront that we need real-time updates and offline support will influence our design choices (like using Supabase's real-time channels and implementing a service worker for the PWA). All requirements should be validated with the stakeholders (e.g. confirming that a 4-digit PIN login is acceptable security-wise for this use case, and that the tournament workflows cover all necessary scenarios) before moving to design.

2. System Design

System design involves crafting the architecture of the application, designing the database schema, defining how components interact, and specifying how user roles will be enforced. In this phase, we translate requirements into a concrete plan: how will the system be built to satisfy those needs? The design should cover both the **high-level architecture** (how the frontend, backend, and database work together) and the **detailed design** of components (database tables, SvelteKit routes, etc.). Key considerations include structuring the database for tournament data, setting up the SvelteKit app architecture, and implementing role-based access.

2.1 High-Level Architecture

At a high level, RCS will be a **client-server application** with a web-based client (the PWA) and a cloud-based backend (Supabase). **SvelteKit** will be used for the frontend, which can serve pages server-side and also behave as a single-page app once loaded. **Supabase** will serve as the backend-as-a-service, providing a PostgreSQL database, authentication service, and real-time data streams. The diagram below illustrates the overall architecture and data flow:

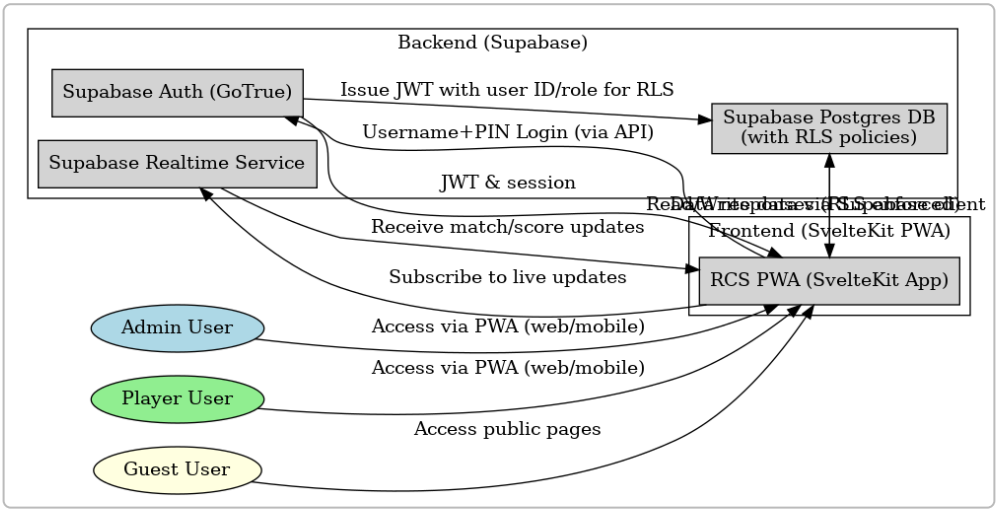

Figure: High-level architecture of the RCS PWA using SvelteKit and Supabase. Arrows indicate user interactions and data flow between the frontend app and backend services. The SvelteKit frontend (right side) runs on Vercel, serving the PWA to Admin, Player, and Guest users. When users log in with username & PIN, the frontend communicates with **Supabase Auth** (GoTrue) to verify credentials and obtain a JWT session token. This token is then used for subsequent requests to the **Supabase Postgres** database, with **Row-Level Security** ensuring each request only returns data the user is permitted to see. The frontend uses the Supabase JavaScript client to perform CRUD operations on tournaments, matches, etc., directly against the database via Supabase's generated APIs. Supabase's **Realtime service** pushes updates (such as new scores or match results) to subscribed clients over websockets, allowing the app to update leaderboards instantly for all viewers. In essence, the SvelteKit app is the presentation layer and orchestrator, while Supabase provides data storage, auth, and real-time infrastructure.

This design leverages the strength of a serverless architecture: the **frontend** is deployed on a platform (Vercel) optimized for SvelteKit, which can handle SSR and static asset delivery, and the **backend** is managed by Supabase (which abstracts a lot of the typical server code). Deploying SvelteKit to Vercel requires essentially **zero configuration**, and gives benefits like automated preview deployments and serverless function support [8]. Vercel can be connected to our Git repository for continuous deployment, generating preview URLs for each pull request [9] – this will streamline collaboration and testing of design changes in later phases.

2.2 Database Schema Design

Designing the database schema for RCS is critical, as it needs to capture all entities of a golf tournament. The main entities identified from requirements are: **Tournaments**, **Players**, **Teams**, **Matches**, and **Scores**.

We will use a relational model (PostgreSQL via Supabase) to define these tables and their relationships. Below is a proposed schema outline:

- **tournaments** – Stores information about each tournament.
  Fields: `id` (PK), `name`, `start_date`, `end_date`, `location`, `format`, `created_by` (FK to an admin's user id), etc.
  Description: Represents a golf event. It may include fields like format (stroke play vs match play), number of rounds, etc. The `created_by` links to the admin who created the tournament (for audit purposes).

- **players** – Stores data for each player (or user profile).
  Fields: `id` (PK) – possibly the same as the Supabase Auth user UUID for easy linkage, `name`, `email` (if any), `username` (for login), `pin_hash` (if using a custom auth table), etc. If using Supabase Auth, we might not store password/PIN here but instead utilize the auth system's credentials.
  Description: Represents an individual participant. If we use Supabase's built-in auth, we will have a separate `profiles` table (with one-to-one relationship to `auth.users` table provided by Supabase) to store additional info like name or handicap. Each player can join multiple tournaments, so a join table or relationship to tournament (see `tournament_players` below) is needed.

- **teams** – (Optional, if team play is supported) Groups of players playing together in a tournament.
  Fields: `id` (PK), `tournament_id` (FK to tournaments), `name`. Possibly captain or additional metadata.
  Description: If the tournament is a team-based event, teams are defined here. A team will have multiple players. We would also need a junction table like **team_members** (`team_id`, `player_id`) to assign players to teams. If the tournament is individual, this table can be unused. Each team belongs to a single tournament.

- **matches** – Stores each match or round pairing in a tournament.
  Fields: `id` (PK), `tournament_id` (FK), fields to identify the competitors, and scheduling info. For competitors, we might have two nullable FKs for **team1_id** and **team2_id** (if teams are playing), and similarly maybe **player1_id** and **player2_id** for individual matches. Another approach is a separate mapping table for match participants, but to keep it simple we can include two competitor fields in matches. Other fields: `scheduled_time`, `course` (if relevant), `status` (scheduled, completed, etc.), `winner_id` or outcome.
  Description: Represents a head-to-head match or a grouping of players. In stroke play, a "match" might just be a grouping (e.g., a tee time group of 4 players) – in that case, competitor fields would hold player IDs for all in the group (which might require more than two participants; we might model stroke play rounds differently, such as having a **rounds** table and then link players to rounds). For match play, two competitors (players or teams) per match make sense. The design should be flexible to accommodate both formats if needed (this might mean our schema includes some optional fields or uses one approach consistently, like always using teams even for individual play by treating each player as a single-member team).

- **scores** – Stores scoring details.
  Fields: Potentially: `id` (PK), `match_id` (FK), `player_id` (FK, if individual scores), `team_id` (FK, if team score), and fields for the score itself. In golf, scoring could be recorded per hole or as a total.

For simplicity, we might store a total score or points in this table (for example, strokes in stroke play, or points won in match play). If hole-by-hole detail is needed, we might have a separate table **scores_by_hole** (with `match_id`, `player_id`, `hole_number`, `strokes`).
Description: This table captures the outcome of matches or rounds. For stroke play, each player's total score (and perhaps their rank or to-par) for a round might be stored. For match play, this could record the result like "Player A vs Player B: Player A won 2&1" – that might be encoded as fields like `points_player1`, `points_player2` or a text result. The design will reflect the chosen format.

- **tournament_players** – (Join table) Links players to tournaments (and possibly tracks extra info like their handicap or starting tee).
Fields: `tournament_id` (FK), `player_id` (FK), maybe `team_id` if players can be pre-assigned to teams here, etc. Composite PK on (tournament_id, player_id).
Description: This makes it easy to query all players in a given tournament, and is useful for enforcing that matches in a tournament can only involve players registered in that tournament.

In summary, the schema is normalized to avoid data duplication. **Foreign keys** ensure relational integrity (e.g. a match must belong to a valid tournament, score must reference a valid match and player). The use of join tables (like tournament_players, team_members) helps model many-to-many relationships (players in many tournaments, players in teams). During design, we will draw ER diagrams and review them with sample data to ensure all use-cases (like an admin adding a late entry player, or a team changing members) are supported.

We will also **enable Row-Level Security (RLS)** on all tables containing sensitive data, as required by Supabase for secure client access [4]. RLS policies will be written to enforce role permissions. For example: a **player** should be able to `SELECT` their own scores and matches, but not those of other players (unless those matches are public). An **admin** will have broader access – perhaps we assign admins to a Supabase role or use a claim in JWT (like `role=admin`) to allow full access. A **guest (anon)** might have `SELECT` rights on only public data (like completed tournaments marked public). We will define these rules using SQL policies. For instance, a policy for the scores table might look like:

```
-- Allow players to view their own scores
create policy "Players can view their scores"
  on scores for select
  using ( auth.uid() = player_id );
```

And an admin policy might bypass restrictions (or we include an `OR is_admin(auth.uid())` in each policy condition, where `is_admin()` is a custom Postgres function or based on a metadata table). These design decisions ensure that even if the frontend has a bug, the database will prevent unauthorized data access. (Supabase's documentation emphasizes enabling RLS for any tables accessed from the browser, as it's critical for security [10] [5].)

2.3 SvelteKit Component Architecture

With the data model in mind, we outline the frontend structure. SvelteKit uses **file-based routing**, so pages and API endpoints are organized by file location. We will create routes corresponding to the main sections of the app:

- **Login Page:** A route (e.g. `/login`) where users enter their username and PIN. This will be a simple form that calls Supabase Auth API. Since Supabase by default expects an email/password, we'll implement the username/PIN by using a known domain hack: if we turn off email verification, we can treat the username as the local-part of an email (e.g. `"john"` becomes `"john@supabase"` as a dummy email) [2]. The PIN will serve as the password. The login page will handle this transformation and call `supabase.auth.signInWithPassword({ email: username + '@supabase', password: pin })` (or use a custom RPC if we implement our own auth). On success, Supabase provides a session and JWT which we store (Supabase JS library handles storing it and will keep the user logged in across page refreshes via `supabase.auth.onAuthStateChange` events [11]).

- **Admin Dashboard:** A protected route (e.g. `/admin` or part of `/dashboard`) accessible only by admin users. It might contain sub-pages: e.g. `/admin/tournaments` (list of tournaments with options to create or edit), `/admin/tournaments/new` (form to create a new tournament), `/admin/tournaments/[id]` (manage a specific tournament: add players, create matches, enter scores, view reports). We will use SvelteKit's **load functions** or endpoints to fetch data from Supabase for these pages. For instance, the tournament management page will load the list of players (from `players` table or perhaps just from tournament_players join via a view), and the matches for that tournament.

- **Player Dashboard:** Upon login, a player might see `/dashboard` showing their upcoming matches, tournament standings, etc. This page is tailored to the logged-in player: it will query for the tournaments they are in (via `tournament_players`) and any match records involving them. The player can navigate to a match detail page to input or view scores (`/match/[id]`).

- **Public View Pages:** Some pages may be accessible without login (especially if we want guests to see the leaderboard). For example, `/tournaments/[id]/leaderboard` could be a public page showing the scores of that tournament. We will ensure that the data fetching for these pages uses an **anon** Supabase client (no JWT, or a stateless request) and that our RLS policies allow read access to the necessary tables for `anon` role, but only for non-sensitive fields. (For instance, we could mark a tournament as public, and our policies allow `select` on scores for anon if the tournament is marked public.)

- **Common Components and Stores:** We will identify components that can be reused, such as a **ScoreCard component** (to display and update scores for a match), a **Leaderboard component** (to display ranking of players/teams in a tournament), and forms for entering data (player registration form, match setup form, etc.). Svelte's component-based structure will help organize the UI. Additionally, Svelte's **context or stores** can manage state like the current user's session (Supabase's auth state) so that all components can reactively know if a user is logged in or their role. We may use a Svelte store to hold the Supabase client and the current `session` (with user info), updating it via

`supabase.auth.onAuthStateChange` events [11], so that components can show/hide based on auth.

- **Routing and Navigation:** The app will have a navigation bar or menu that adapts based on role. For example, an Admin user after login might see links to "Manage Tournaments" and "Manage Players," whereas a Player might see "My Schedule" or "Enter Scores". We'll use SvelteKit's layout features to define a top-level layout that checks the user's session (perhaps in `+layout.svelte` or a hook) and directs to the appropriate start page (e.g. redirect a logged-out user to `/login`, redirect a logged-in player trying to access an admin page back to their dashboard, etc.).

- **Service Worker (for PWA):** In order to fulfill PWA requirements, we will add a service worker. SvelteKit allows adding a service worker file (e.g. `src/service-worker.js`) which Vite can build. This service worker will cache static assets and possibly use a caching strategy for API calls. We might use the Workbox library or manually implement a cache-first strategy for certain requests (like caching the tournament list for offline viewing). We'll also include a PWA manifest (JSON) referencing icons, app name, etc., so that the app is installable. The Rodney Lab guide notes that we need icons and a manifest, and the service worker to enable offline support [12]. During design, we will decide which parts of the app should be available offline (perhaps allow players to open their schedule that was previously loaded even if connection drops). This might involve storing some data in localStorage or IndexedDB for offline mode.

2.4 User Roles and Security Design

As identified, **Admin**, **Player**, and **Guest** are the roles. The design must ensure each role's capabilities are enforced both in the client application and in the backend:

- **Admin:** Can perform any action within their domain (usually scoped to the organization or tournament they manage). In the UI, admins will see management screens that are not visible to players (we can conditionally render admin menus if `session.user.role == 'admin'` for example, storing a custom claim or checking against an admin list). In the database, we might maintain an `admins` table or flag in the `profiles` table. Supabase allows adding custom JWT claims – for instance, we could have a `is_admin` boolean in the `profiles` table and configure Supabase Auth to include that in JWT. Then an RLS policy can allow full access if `auth.jwt() ->> 'is_admin' = 'true'`. Alternatively, we can simply write policies that check if `auth.uid()` is in some pre-defined list of admin user IDs (not as scalable). We will also use Supabase's built-in **Policy for staff** pattern for certain tables: e.g., to allow an admin to bypass normal user restrictions, an example policy might be: `using ( auth.role() = 'service_role' OR auth.uid() = tournament.created_by )` – Supabase's `service_role` is a powerful key but not used on client, instead we might tag admin users with a role.

- **Player:** In the UI, players get access only to their own data or general info. They won't see the admin menus. On pages like a match detail, if a player not involved in that match somehow tries to access it (by URL manipulation), the client should prevent or redirect them. But importantly, the backend will also stop it: RLS on the `matches` table might be `using ( match.id IN (SELECT match_id FROM tournament_players tp WHERE tp.player_id = auth.uid()) )` or something similar to ensure a player can only select matches that involve them (we would derive this with a join between matches and participants). For `scores`, a player might only insert or update their own

score entry. We might allow read access to all completed scores in their tournament if transparency is desired (or if tournaments are public, all players might see the full leaderboard).

- **Guest:** Unauthenticated users get very limited access. By default, Supabase treats them as role `anon`. We likely will not allow anon to write anything. Read access will be restricted to specific endpoints – for instance, maybe allow `SELECT` on tournaments table but only for basic fields of public tournaments (we can enforce `using (public = true)` in a policy). The design might include a separate **public API** or use Supabase's `invoke()` to call a Secure Function if needed to aggregate public data safely. However, given Supabase's flexibility, we can probably expose what we need with careful RLS. Guests, in the UI, will only see publicly available pages like a list of tournaments marked public or a live leaderboard page if we design one that doesn't require login.

- **Authentication Mechanism:** The design of the username+PIN auth needs special attention. Supabase's Auth (GoTrue) natively supports email/password, OAuth, magic links, and phone OTP, but not a short PIN login by default. To implement this, we have a couple of design choices:

- **Use Supabase email/password with dummy emails:** As mentioned, we can adopt a convention that every username corresponds to an email like `<username>@rcs.app` (or any fake domain). We will disable email confirmations so that these addresses don't need to be real [2]. The 4-digit PIN will be set as the password on account creation. This approach works within Supabase's existing system (so we can still use `supabase.auth.signIn`). **Security note:** 4-digit PIN as a password is weak (only 10,000 combos). To mitigate, we can enforce rate limiting of logins (Supabase might have some in-place, but we might implement a Cloudflare or Vercel function proxy to limit attempts) and ensure that the PIN is not easily guessable (truly random). Admins should communicate PINs securely to players. For additional safety, we could require players to change to a longer password after first login – but since the requirement is explicitly a 4-digit PIN for simplicity, we assume a controlled environment where this is acceptable.
- **Custom authentication table:** Alternatively, we design our own table (e.g. `auth_users` with username and a hashed PIN) and use Supabase **Edge Functions** or a custom Node server to verify login attempts. On successful verification, we could issue a JWT ourselves using Supabase's JWT secret so that the user can still use Supabase services. (Supabase JWT can include a `sub` claim for user id and any custom claims like role). This gives us complete control (like locking out after 3 failed attempts, etc.), but it's more complex to build and maintain. Given time and complexity, the first approach (piggybacking on email/password) is likely sufficient and much faster to implement.

We will proceed with approach #1 for design, as it stays within Supabase's managed auth. Admins will use the Supabase dashboard or an admin UI in RCS to create users: essentially calling `supabase.auth.signUp({ email: username+'@rcs.app', password: pin })` for each player. We will store the username in the user metadata or in the profiles table for display. Logging in then just requires that same combination. (We should also consider how password reset would work if a player forgets their PIN – since we have no real email, the admin might have to reset it manually via an interface that calls `auth.admin.updateUser` with a new password. We'll document this procedure.)

- **Real-Time Design:** Another design aspect is how real-time updates propagate. Supabase Realtime can broadcast changes in specific tables to all subscribed clients. In our architecture, when a score is inserted or updated in the `scores` table, we want players' leaderboards to update live. We will design the client to subscribe to relevant channels on mount. For example, on the tournament

leaderboard page, the SvelteKit component can subscribe to `supabase.from('scores').eq('tournament_id', currentTournamentId).on('INSERT', payload => { ... update state ... }).subscribe()` (using Supabase JS v1 syntax) or the new channel API in v2. Supabase's real-time can filter by RLS as well – each client's subscription respects RLS, meaning a player will only receive the changes they are allowed to see. (If our RLS for scores allows anyone in the tournament to see all scores once posted, then all players will get updates. If not, we might have separate channels or a logic to send only aggregated info.)

The design thus includes which pages/components will use real-time: clearly the live leaderboard, perhaps a match page if two players are entering scores from two devices (to see each other's input live), and any admin dashboard that wants to reflect new signups or changes without refresh. We will likely use one or more **Supabase Realtime channels** for these. Supabase v2 allows a more unified `channel` API for realtime. For instance, we can create a channel per tournament: `const channel = supabase.channel('TournamentUpdates:'+id).on('postgres_changes', {event: '*', schema: 'public', filter: 'tournament_id=eq.'+id}, handler).subscribe()` – this would listen to all inserts/updates/deletes on tables (scores, matches, etc.) with that tournament_id [13]. This is efficient and scoped. On the client side, we will update Svelte stores or component state upon receiving these events, causing the UI to re-render the new data.

In summary, the system design sets up a robust architecture: a normalized relational database with secure access, a modular frontend with clear separation of concerns (pages for each feature, reusable components, and centralized state for auth), and a real-time communication mechanism. We also ensure the **PWA considerations** (manifest, service worker) are part of design. Additionally, we plan for how to test and deploy this architecture, which is detailed in the next sections.

3. Implementation Plan

With the design in place, the implementation phase outlines the actual tasks and steps needed to build the RCS application. This will involve setting up the Supabase backend, configuring authentication for usernames and PINs, developing the SvelteKit front-end pages and components, and integrating everything (including real-time updates). We will approach development iteratively, possibly following an agile approach (implementing basic functionality, then enhancing). Below is a step-by-step breakdown of the implementation tasks:

1. **Set Up Supabase Project and Database:**
2. Create a new Supabase project (through the Supabase dashboard). Define the database schema using the Supabase Table Editor or SQL queries. Start by creating tables for **tournaments**, **players (profiles)**, **teams**, **matches**, **scores**, and any necessary join tables (like `tournament_players`). Ensure that each table has appropriate primary keys and foreign keys linking to others (Supabase UI allows setting foreign key relationships, which also helps with RLS policies later). For example, set `matches.tournament_id` as FK to `tournaments.id`, etc.
3. After creating tables, **enable RLS** on each of them (in Supabase, RLS is enabled by default if you create via UI, but double-check). Add Row Level Security policies as per the design for each role. Initially, you might add a liberal policy for development (e.g. allow the admin user full access, and maybe allow select to all for initial testing), then tighten these rules as the app logic for roles is ready. Supabase provides a handy quickstart policy in their docs, e.g., to make a table publicly

readable: `create policy ... for select to anon using (true);` [14] (we will adapt such policies to our needs).

4. Use Supabase **SQL Editor** or CLI to insert some seed data for development/testing – like a sample tournament, a couple of players, etc. This will help test the app as we build it.

5. **Configure Authentication for Username & PIN:**

6. In Supabase **Auth Settings**, disable email confirmations (since we will use dummy emails) and enable email/password provider. Also, consider turning off any password complexity requirement if it's too high for 4-digit PIN – by default Supabase might allow a simple 4-character password, but we'll verify. Supabase's GoTrue (auth service) will accept something like `username@yourdomain` as an email as long as it contains an `@`, even if the domain is not real [2].
7. Implement an **admin script or UI** for creating users: For example, create an Admin page in the app where an admin can input a username and PIN, and it calls `supabase.auth.admin.createUser()` (Supabase provides admin auth functions) or uses the client with service key to sign up the user without email confirmation. Alternatively, do this manually in Supabase dashboard for initial setup. When creating a user, store their username in the `players` (profiles) table alongside the generated `user_id` (the UUID from auth). We can use a database trigger to automatically insert a row into `players` when a new `auth.users` is created, if we use the typical Supabase `auth.users` + `public.profiles` pattern [15]. For example, a trigger can copy `auth.users.id` into `profiles.id` and set the username from the `raw_user_meta_data`. Supabase has examples of using triggers to populate profile tables with usernames [16].

8. Implement the **login form** on the frontend to collect username and PIN. On form submit, convert the username to our fake email (`username@rcs.app` or similar) and call `supabase.auth.signInWithPassword({ email, password: pin })`. Handle errors (e.g. wrong credentials) with an error message (and consider after 5 failed attempts, maybe show a "contact admin" message or simple captcha to prevent brute force). If login is successful, the Supabase client stores the session. We should then redirect the user to the appropriate dashboard (if admin, `/admin`, if player, `/dashboard`). SvelteKit's navigation can be done via `$app/redirect` or using the `goto()` function in a form action.

9. **Implement Core CRUD Interfaces (Tournaments, Players, Teams):**

10. Start with **Tournament management** pages. Create a SvelteKit page for listing tournaments (for admin). Use `supabase.from('tournaments').select('*')` to fetch tournaments (likely server-side in `+page.server.ts` or client-side in `onMount` – since admin page is protected and requires login, using server load with the user's JWT (obtained via cookies or session) might be convenient to pre-fetch data). Provide a button/link to "Create Tournament".
11. Create a page or modal for **New Tournament**. This form will capture tournament details and on submit, use `supabase.from('tournaments').insert({...})` to create a new record. Ensure the `created_by` field is set to the current admin's user ID. After insertion, navigate back to the list or directly to the tournament detail page.
12. **Tournament Detail (Admin view):** This page (`/admin/tournaments/[id]`) will have sub-sections or tabs: e.g. *Players*, *Teams*, *Matches*, *Scores/Leaderboard*. Implement functionality to **add players** to

the tournament: possibly by selecting from all players or entering new ones. If adding an existing user, we insert into `tournament_players`. If creating a new player on the fly, that might involve creating a new auth user – which may be too much for one interface. Instead, likely all players should exist as users beforehand (via the user creation step). So here, we might just provide an interface to choose from `players` table (which is essentially all registered users) and add them. Use Supabase RPC or multiple calls: first insert into `tournament_players`, and possibly also into `team_members` if assigning to a team at the same time.

13. For **teams**, allow admin to create teams (insert into `teams` table) for that tournament, and assign players to teams. This could be a simple form: input team name, select members (multi-select of players who are in this tournament). On save, insert the team, then insert entries in `team_members`.

14. These CRUD operations (create/update/delete tournaments, players, teams) should be straightforward with Supabase JS SDK calls. We will use SvelteKit form actions or onSubmit handlers. After each operation, update the local state or re-fetch the list to show changes. Setting up optimistic UI (immediate update) is nice but initially we can rely on re-fetch or the real-time to update the list (e.g. after adding a player, a real-time event on `tournament_players` could trigger the UI to refresh the player list).

15. **Implement Match Scheduling Interface:**

16. For each tournament, admin needs to create matches. Provide a form where admin selects two players (or teams) and perhaps a date/time for the match. If the tournament format is round-based, you might have a concept of rounds and automatically generate matches (could be a future enhancement using a bracket generation algorithm). Initially, implement manual scheduling: on the Matches tab of tournament detail, have "Add Match" button. The form will show a dropdown of available competitors (teams if team tournament, else players) for slot 1 and slot 2, and a datetime picker for the tee time. On submit, insert into `matches` table. If we treat all matches as 2-competitor, that covers match play. For stroke play (everyone plays the course), an alternative approach is needed (like auto-generating groups of 4). For now, assume match play or small group play where scheduling is needed.
17. Display the list of matches for the tournament, sorted by time or round. For each match, show the participants and either the scheduled time (if upcoming) or the result (if completed). Admin should be able to edit or delete matches (e.g. if a match is cancelled or changed). Use Supabase update and delete for those actions, and reflect changes in UI (with real-time or manual refresh).

18. **Real-time integration:** It would be useful that if one admin creates or edits a match and another admin (or a player viewing schedule) has the app open, they see the update. By subscribing to `matches` changes, this can happen. We will implement `supabase.channel('matches:tourn_'+id)...` subscription on the matches list component so any new match or update triggers a state refresh.

19. **Implement Scoring Input and Live Updates:**

20. The scoring system can be implemented in phases. First, ensure we can record final scores for matches (or total strokes for players). Provide a **Score Entry** form. Perhaps the match detail page (`/matches/[id]`) is where scores are entered. When a player views their ongoing match, they should

see an interface to input their scores (if we allow players to self-report) or an admin might do it. For a simple start, let's say only admins or a designated scorer enters scores. In match play, the score might be just the result (win/loss) – but if we want to record hole-by-hole, we might build a UI for that (18 input fields, etc.). This can be complex, so optionally, start with final result entry: e.g. "Team A won 3&2" or "Player X scored 72 strokes".

21. Use `supabase.from('scores').insert()` to record a new score entry. Alternatively, since we might have one row per match per player, we could insert two rows (one for each player's score) if using individual scores. Or we design it as a single match result row. We need to reconcile the schema choice here with implementation: let's assume for simplicity **scores table holds individual scores per round** (so in stroke play, each entry is a round score of one player; in match play, we might not use `scores` at all but store winner in `matches`). If match play, perhaps skip `scores` table usage except for maybe hole results. To keep things simple: for stroke play format, implement `scores` as each player's score in a round (with `match_id` possibly null or representing a round grouping). For match play, just update `matches.winner_id` or a field. We'll implement accordingly.
22. Once scores are submitted, update any related fields (for example, mark match status as completed). On the UI, after saving, perhaps navigate to a *Leaderboard* view. The **leaderboard** can be computed with a Supabase SQL query (like summing scores, ordering by lowest). We might create a **view** or use Supabase's stored procedures to compute standings. But initially, we can do it in the client: fetch all scores for that tournament and sort. That's fine for a small number of players; if scaling up, a view or SQL function would be better.
23. **Real-time updates (live scoring):** On any page that displays scores or results (leaderboard page, match page, etc.), set up a subscription to the `scores` table (and possibly `matches` table for match status). For example, on the Leaderboard Svelte component, subscribe to `scores` changes for that tournament: when a new score is inserted or an existing one updated, recalc the standings state. Supabase's real-time will push the changed row in the payload so we can integrate it easily [13]. Testing this, we should see that if one user enters a score on their device, another user's leaderboard updates within a second or two. This provides the "live update" experience as required.

24. Additionally, ensure that the **service worker** does not cache API calls in a way that would interfere with real-time (likely not, since realtime is websocket, but for safety ensure we don't cache the initial fetch of scores aggressively or if we do, we update it on events). The PWA service worker could perhaps cache only static assets and maybe data for offline mode, but when online we want fresh data.

25. **Implement Unit and Integration Testing (during development):**

26. As code is written, also write tests for critical units. For example, if there is a function that calculates tournament rankings from a list of scores, write a unit test for it using **Vitest** (which is SvelteKit's recommended testing framework [17]). SvelteKit doesn't impose a testing library, but we can use Vitest for unit tests and **Playwright** for end-to-end tests (SvelteKit projects often come with a Playwright setup for E2E). We will create a few Vitest specs for utility functions and maybe simple component logic (using Svelte Testing Library for components). Testing ensures our implementation meets the design and catches regressions [18].
27. Set up a few integration tests: for instance, using Playwright to simulate a user flow – open the login page, enter username/PIN, login (perhaps stub Supabase or use a test instance), then check that they can see their dashboard. Another integration test could call Supabase directly (with a service

key in a test environment) to verify RLS: for example, attempt to read another user's score as a non-admin and assert it fails. Supabase's JS client can be used in tests to simulate different users by supplying different access tokens. Writing such tests helps validate that our RLS policies truly restrict data as expected [19]. (This addresses the **RLS policy validation** point – we will, for instance, use the Supabase API with an test user's JWT to attempt to fetch someone else's data and confirm we get an empty result or error). Supabase's docs highlight that testing is critical especially when using RLS [19]. We can incorporate those in our test suite so we don't accidentally break a policy later.

28. Edge-case handling: implement and test scenarios like **invalid input** (e.g. if someone tries a PIN that's not 4 digits – the UI can enforce it but also backend should handle), **duplicate entries** (adding a player twice to a tournament – database unique constraint on tournament_players can prevent this, which we should test), **concurrent updates** (two admins editing the same match – last write wins, and UI should refresh via realtime). Also test the PWA offline mode: build the app, load it, then go offline and see if at least a cached page (like an offline fallback or previously loaded data) is available. Ensure no crashes happen when offline (perhaps add a check: if supabase calls fail due to no network, the app should inform the user or queue the action).

29. **UI Polish and PWA Features:**

30. Implement responsive design using CSS (maybe Tailwind or Skeleton if chosen) so that the app looks good on mobile phones. Test on different screen sizes.
31. Add a **loading state** and error handling in the UI for data fetching. For example, while waiting for Supabase to return query results, show a spinner; if an error occurs (like network down or permission denied), show a friendly message.
32. Set up the Web App Manifest (`manifest.webmanifest`) with the app name (e.g. "RCS Golf Manager"), icons (generate various sizes from a base logo), theme colors, etc. SvelteKit can auto-reference the manifest if placed in `static/` folder. Confirm that after deployment, the site passes Lighthouse PWA checks (i.e., it's installable).
33. Integrate the service worker for offline caching. We might use a Vite plugin like `vite-plugin-pwa` which can auto-generate a service worker based on our config, or write one manually. For manual: in `service-worker.js`, use the Cache API to pre-cache static files and maybe cache API responses. For instance, cache the tournament list so if the user opens the app offline after previously using it, they see the last known data. This requires careful consideration to not show stale data without warning, but as a PWA enhancement it can be optional.

Throughout implementation, we will commit code regularly and use version control (Git). Because we plan to deploy on Vercel, every push can trigger a deployment preview (as configured in the design) which allows testing the live behavior. We'll also make use of Supabase's local development tools (Supabase CLI) for a development environment: it lets us run a local Postgres with the same schema and test RLS policies and edge cases before pushing to prod.

By the end of implementation, we should have a functional application that meets the requirements: users can log in with PINs, admins can set up tournaments and matches, players can view and input scores, and everyone sees updates in real-time during a tournament.

4. Testing

Testing is a crucial part of the SDLC to ensure quality and catch issues early. We will employ multiple levels of testing for RCS: unit tests for individual functions/components, integration tests for how components and backend interact, security tests for RLS policies, and end-to-end tests for user workflows. SvelteKit is unopinionated about testing tools and supports using Vitest, Cypress, Playwright, etc. [18] – we will use **Vitest** for unit/integration tests and **Playwright** for end-to-end scenarios, aligning with SvelteKit's recommendations.

Unit Testing (Frontend): We will write unit tests for pure logic functions and Svelte components where feasible. For example, if we have a function `calculateLeaderboard(scores)` that takes an array of score objects and returns a ranked list, we will create a `calculateLeaderboard.test.ts` and use Vitest's `expect` to verify it sorts and ranks correctly (including handling ties or missing scores). For Svelte components, we might use the Svelte Testing Library to render a component in isolation and assert that given certain props it displays the right content. For instance, test that the `<Leaderboard>` component properly highlights the winner or that `<LoginForm>` calls the Supabase login function with the transformed email. These tests help ensure our components behave as expected in isolation.

Integration Testing (Frontend-Backend): Integration tests will involve multiple parts of the system. One approach is to use the Supabase JavaScript client in a Node test environment to act as different users. We can use **Vitest** (which can run tests in Node or happy-dom) to call our running application's endpoints or directly the Supabase API. Example: Write a test that uses the Supabase anon key to call a function or endpoint that retrieves public tournament data – ensure it only returns expected fields for a guest user. Another integration test might register a new user via Supabase API, then simulate that user retrieving their own profile or inserting a score, verifying that works while inserting a score for another user is forbidden (testing RLS). By using Supabase's client with a service role (full rights) versus using it with a user JWT, we can validate RLS logic effectively [20] [21]. Supabase's guides mention approaches like pgTAP for direct DB tests [22], but here we can achieve a lot with just the JS client and our policies.

Security Tests (RLS Policies): Since we rely on RLS for data protection, we will test these thoroughly. For each table with RLS, we craft test cases: e.g., "Player cannot select another player's score." This would entail: using a test user's Supabase session, call `supabase.from('scores').select(...)` filtering to another user's match, and expecting an empty result or error. Another: "Only admins can insert new tournaments." – attempt to insert a tournament as a normal player user, expect a failure (403 Forbidden). These tests effectively simulate malicious or unexpected actions to ensure our policies are airtight. We might create dedicated test users: one admin, one regular player, one not in the tournament, etc., seed data appropriately, then run queries. We'll automate these tests so they can be run whenever we update policies (this is critical, as misconfigured RLS could either expose data or block legitimate access). As Supabase docs say, having a comprehensive approach to testing the database (including RLS) is important for robust security [19].

End-to-End Testing (UI flows): Using Playwright (which comes with SvelteKit by default for E2E), we will simulate actual user interactions in a headless browser. For example, an E2E test scenario: "Player login and view scoreboard":
- Launch the app in a test environment with a known state (we might run a local dev server connected to a test Supabase database with seeded data).
- Use Playwright to navigate to the login page, input a test username and PIN, click login. Verify that it

redirected to the player's dashboard page.
- On the dashboard, verify that it displays the correct player name and list of upcoming matches (which come from the test data).
- Navigate to a match page, enter a score, submit. Then verify that the leaderboard page updated accordingly (this might involve waiting for a realtime update or simply reloading the page in the test).
Another scenario: "Admin creates a tournament": simulate admin logging in, going to create tournament form, filling details, and saving. Then check that the new tournament appears in the tournament list. Perhaps also simulate adding a player to the tournament and ensure that the player now sees that tournament in their dashboard (which could be done by then logging in as that player or checking the DB state).

These E2E tests cover the most important user journeys and ensure that the system as a whole (frontend + backend) works as intended.

Edge Case and Performance Testing: We will also test edge cases like:
- **Invalid inputs:** Ensure the UI validation prevents bad data (e.g. letters in the PIN field, or missing tournament name) and that the backend has constraints (like not null fields, proper data types) so it rejects bad data if somehow received. Write tests for those rejections.
- **Concurrent actions:** What if two admins schedule matches at the same time? The system should handle it (Postgres will serialize transactions). We might not explicitly test race conditions, but using the app in a staging environment with multiple users can reveal if any real-time updates conflict.
- **Offline functionality:** Manually test (since automation is tricky) the PWA offline mode – load the app, turn off network, and see if the previously loaded data is accessible. Ensure no uncaught exceptions. Possibly write a integration test to simulate absence of network by disabling fetch requests, though this might be overkill.

Throughout testing, if any test fails, we will revisit the implementation to fix the issue, then re-run tests. This iterative test-driven approach for critical components will increase confidence in the system's reliability. By the time we are done, we should have a robust test suite that can be run as part of CI/CD to catch regressions on every code change.

5. Deployment

Deployment involves releasing the application to a production environment accessible to end users. In our case, the frontend (SvelteKit PWA) will be deployed on **Vercel**, and the backend is managed by **Supabase** (hosted Supabase cloud service). We'll also configure environment variables, HTTPS, and CI/CD for an automated and smooth deployment process. Here are the deployment considerations for each part of the stack:

- **Frontend Deployment to Vercel:**
  Vercel is a great fit for SvelteKit and requires minimal setup [8]. We will create a new project on Vercel and point it to our GitHub repository (assuming we use GitHub). Because SvelteKit's default adapter is `adapter-auto`, it will detect Vercel and generate the appropriate build (essentially serverless functions for SSR and static assets for the client). On each push to the main branch (or whichever branch we configure), Vercel will build the SvelteKit app and deploy. We should verify that the build output does not include any sensitive info (the public/secret env vars separation covers this as explained below).

- **Environment Variables:** In Vercel's project settings, we will set the necessary environment variables for the frontend. These include the Supabase project URL and the Supabase **anon public key**. SvelteKit allows environment variables to be exposed to the client if they are prefixed with, e.g., `PUBLIC_` (in newer SvelteKit, it's using `$env/static/public`). We will store the Supabase URL and anon key as public env vars since the client needs them to make requests. (The anon key is safe to expose, as it only grants the permissions we define in RLS [4].) Any other environment secrets (like a Sentry DSN for error reporting or other API keys) would be set as needed but not exposed publicly. Vercel provides an easy UI to add these, and they will be injected at build time.
- Vercel automatically serves the app over **HTTPS**, which is required for PWA features like service workers and for security. We might use a custom domain for the app (like `rcs.yourgolfclub.com`), which we can configure in Vercel's settings, and Vercel will handle the SSL certificate via Let's Encrypt.
- **CI/CD:** By integrating with Git, every commit can trigger a deployment. We can protect the main branch so that changes must pass tests (we can set up GitHub Actions to run our Vitest/Playwright tests on pull requests). Optionally, we can use Vercel's GitHub integration which provides **Preview Deployments** for each PR (so stakeholders can test features before merging). Once merged, Vercel will deploy to production. This continuous deployment pipeline ensures that updates go live quickly and reliably, with the safety net of tests and previews.

- We should monitor the Vercel deployment logs after going live for any errors (for instance, missing env vars or SSR issues). Vercel offers serverless function logs which will show if any error happens during page loads.

- **Backend Deployment (Supabase):**
  Since we chose Supabase's cloud service, much of the "deployment" of the backend is simply the creation of the project which we did earlier. However, managing changes to the database schema and functions should be done in a controlled way:

- **Database Migrations:** As the project evolves, if we change the schema (add a table, modify a column, etc.), we will use **Supabase CLI migrations** to apply these changes. Supabase CLI allows us to capture schema changes into migration files (SQL scripts) that can be committed to our repo [23]. This ensures we have version control over the database structure. When deploying, we can either manually run these migrations via the Supabase CLI (`supabase db push`) or automate it with GitHub Actions. For example, on each push to main, run `supabase db diff` to generate migration and `supabase db push` to apply to the remote dev/prod database. This prevents drift between environments and documents schema changes over time [24].
- **Supabase Environment:** Ensure the **Supabase project's environment variables** (like the JWT secret) remain configured. By default, Supabase handles JWT secret internally. If we wrote any edge functions or gotrue settings (for example, if we needed to allow a custom email domain for username login hack), those should be configured in Supabase Settings. For instance, Supabase might require that the domain we use in fake emails (`@rcs.app`) is added as an allowed domain (or since email confirmations off, it might not matter). We will double-check auth settings before launch.
- **Row Level Security & Anon Key:** We will ensure RLS is fully enabled on all necessary tables in production. The **anon public key** will be used by the frontend for non-auth calls or initial requests; the **service role key** (which is extremely sensitive and full-power) will **never** be exposed to the frontend. We might use the service role key only in secure server environments if needed (like a Node script or Cloud function for admin tasks not covered by our app).

- Supabase provides daily **backups**, but we might also set up our own backup schedule or use the Supabase export if needed. This is part of deployment readiness – ensuring we can recover data if needed.

- **Continuous Integration/Deployment:**
  As mentioned, we will integrate testing and deployment. A possible workflow: use GitHub Actions to run tests on each push; if tests pass, automatically deploy to a staging environment (maybe a separate Supabase project and Vercel project). Once changes are approved, merge to main which triggers deployment to production. We could also use Supabase **branching** (a newer feature) to spin up a temporary DB for preview environments if needed, but that might be overkill. The aim is that deployment is not a manual, error-prone process but rather a smooth, repeatable pipeline. This reduces risk when releasing updates.

- **Monitoring Post Deployment:**
  After deployment, we will monitor both frontend and backend. Vercel's Analytics can give us performance metrics on the frontend. Supabase provides some monitoring on query performance and logs of errors (like if a query is denied by RLS, it shows in logs). We'll keep an eye on these, especially during the first live tournament usage, to catch any performance bottlenecks (e.g. a heavy query that could be optimized with an index).
  We also set up alerts – for example, if the Supabase CPU or memory usage spikes or if a specific error appears frequently in logs, we want to know.

Finally, ensure that the app's versioning is clear. We might tag releases (v1.0, etc.) in Git. If we need to roll back, with Vercel it's as easy as redeploying the previous commit. Having a migration strategy for the database is also important – if a migration fails, we should be ready to apply fixes (or in worst case restore backup).

Deployment is considered done when end-users (tournament admins and players) can reliably access the application at the public URL, and all services (auth, data, realtime) are working together. At that point, we transition into the **maintenance** phase, continually monitoring and improving the system.

6. Maintenance and Monitoring

The maintenance phase of the SDLC is about keeping the software running smoothly, fixing issues, and making iterative improvements once the system is in use [25]. For the RCS PWA, we will establish procedures for monitoring the app's health, handling bug fixes or security updates, updating features, and collecting user feedback for future enhancements. Maintenance ensures the software adapts to changing needs and continues to meet user expectations [26].

Key maintenance and monitoring practices include:

- **Application Monitoring & Error Tracking:** Set up tools to catch runtime errors and performance issues in production. We can integrate a service like **Sentry** or use Vercel's built-in logging to track exceptions on the frontend and backend. For example, if a player's device encounters a JavaScript error (perhaps a corner-case bug), Sentry can log the stacktrace and context, allowing us to diagnose it. Likewise, we can monitor Supabase for any error responses or permission denials that occur

unexpectedly. This proactive error monitoring means we don't have to wait for a user to report an issue; we may get alerted immediately. Performance monitoring (like slow page loads or slow database queries) can also be done. Supabase's dashboard might highlight slow queries if our leaderboard query is inefficient at scale – we would then optimize (maybe adding indexes or caching results). Regularly reviewing logs and performance metrics is an important maintenance task, especially during active tournament usage.

- **Database Maintenance and Versioning:** As new features are requested (say we want to add a new field or support a new tournament format), database changes will happen. We will use a version-controlled migration process (as discussed in Deployment) so that every schema change is captured. Over time, we'll maintain a **changelog** of database versions. Maintenance also includes ensuring data integrity – e.g., if a bug allowed a wrong score entry, an admin might need to correct the database. We can build admin tools for certain maintenance tasks (like "reset a match" or "recalculate a leaderboard") to avoid direct DB edits. Also schedule periodic **backups** and test restoring them. Supabase does automatic backups, but we should know how to restore data quickly in case of a critical failure. In terms of DB performance, we should update indexes or queries as data grows; maintenance might involve adding an index if we notice a slowdown (Supabase allows adding indexes easily if needed). We should also consider cleaning up data: for example, archiving old tournaments (moving them to an "archive" table or a storage backup after a couple years) to keep the main DB lean.

- **Security Updates:** Keep an eye on SvelteKit, Supabase, and any other libraries for updates or patches (especially security fixes). Since our app is on a modern stack, updates come often. We should periodically upgrade dependencies (after tournaments or in a maintenance window) and run our test suite to ensure nothing breaks. Supabase platform itself is managed, so they will handle patching the database and auth services, but we should read their release notes for any breaking changes or new features that we might want to adopt.

- **User Support and Feedback Collection:** Once actual users (admins, players) start using RCS, we need a channel for them to report issues or request features. This could be as simple as a feedback button in the app that links to a Google Form, or an email/contact page. We plan to integrate a feedback mechanism so that maintenance can be guided by real user input. For example, players might suggest "It would be great if the app could also show weather info for the course" or report "I had trouble logging in on my phone." We will log these feedback items and incorporate them into the next development iteration. Maintenance often transitions into a new requirements gathering for the next version of the software, based on this feedback.
  Additionally, analytics (respecting privacy) can be collected to see how users are using the app – e.g., if we notice many players are not completing the score submission process, that could indicate a usability issue that we need to fix in an update.

- **Regular Updates and New Features:** Treat maintenance as an ongoing cycle. Plan minor releases or updates at regular intervals. Minor updates could include UI improvements (maybe after seeing how people use it in the field, we might adjust font sizes or contrast for sunlight readability), adding a small feature (like exporting results to PDF), etc. Ensure each update goes through the SDLC stages in brief – we gather the requirement or bug description, design the fix or feature, implement it, test it, deploy it. Because the app is live, we may use a staging environment to test maintenance releases

with sample data before updating production. Users should be informed of updates (perhaps via a changelog or notifications, especially if an update requires them to refresh the PWA or re-login).

- **Maintaining the PWA Aspects:** Over time, we must keep the PWA assets (like the manifest and icons) up to date, especially if branding changes. Also monitor the service worker – if we ever change caching strategy, ensure old service workers are properly updated on users' devices (there's a lifecycle to manage to not break the app for returning users). If users report that the app is showing old data, it might be a caching issue – we would address that in a maintenance patch (maybe tweak cache headers or use the service worker update mechanism).

- **Scalability and Infrastructure Monitoring:** If the app grows in usage (more tournaments, more concurrent users), we should keep an eye on Supabase usage quotas (e.g. max concurrent connections, bandwidth). If needed, we upgrade our Supabase plan or add caching. Similarly for Vercel, monitor the function invocation count and if we're nearing any limits. It's part of maintenance to ensure the infrastructure can handle the load. We can use Supabase's reports to see how heavy the real-time traffic is, etc., and optimize accordingly (for example, if real-time data usage becomes high, perhaps we switch to more granular channels or partially disable some subscriptions).

In conclusion, maintenance is an ongoing effort of **"constant assistance and improvement"**, as one SDLC source puts it [25]. By responding to user feedback, fixing issues promptly, and continuously polishing the application, we ensure RCS remains a reliable tool for managing golf tournaments over the long term. Each maintenance cycle feeds back into the requirements analysis for the next iteration, thereby continuing the SDLC loop and keeping the software aligned with user needs and technological advancements.

---

[1] [7] Software Development Lifecycle (SDLC) - Requirement Analysis
https://teachingagile.com/sdlc/requirement-analysis

[2] r/Supabase - Building User Authentication with Username ... - Reddit
https://www.reddit.com/r/Supabase/comments/16mrmrz/building_user_authentication_with_username_and/

[3] [12] SvelteKit PWA: Installable App with Offline Access | Rodney Lab
https://rodneylab.com/sveltekit-pwa/

[4] [5] [6] [10] Row Level Security | Supabase Docs
https://supabase.com/docs/guides/database/postgres/row-level-security

[8] [9] SvelteKit on Vercel
https://vercel.com/docs/frameworks/sveltekit

[11] [13] Svelte & Supabase Real-time Chat app - DEV Community
https://dev.to/ahmdtalat/svelte-supabase-chats-2l14

[14] Use Supabase with SvelteKit | Supabase Docs
https://supabase.com/docs/guides/getting-started/quickstarts/sveltekit

[15] How do I auth user with username or email? · supabase - GitHub
https://github.com/orgs/supabase/discussions/16619

[16] Build a User Management App with React | Supabase Docs
https://supabase.com/docs/guides/getting-started/tutorials/with-react

17 18 Testing • Docs • Svelte
https://svelte.dev/docs/svelte/testing

19 22 Testing Overview | Supabase Docs
https://supabase.com/docs/guides/local-development/testing/overview

20 21 Testing Supabase Row Level Security - DEV Community
https://dev.to/davepar/testing-supabase-row-level-security-4h32

23 24 Database Migrations | Supabase Docs
https://supabase.com/docs/guides/deployment/database-migrations

25 26 The Seven Phases of the Software Development Life Cycle
https://www.harness.io/blog/software-development-life-cycle-phases