Developer Handbook

Stylized Edition with Bookmarks + Clickable TOC

Table of Contents

Introduction

Developer Handbook — Introduction

Welcome to the **Recruiting Ecosystem Developer Handbook**.

This handbook provides the technical guidance needed to build, maintain, and extend the system. It is written for engineers working on the web application, APIs, database, and AI services that power the ecosystem.

System Purpose

The Recruiting Ecosystem is a unified platform designed to:

- elevate high school coaching quality

- modernize college recruiting workflows

- centralize athlete development data

- provide AI-assisted decision-making

- integrate billing, roles, and organizational hierarchies

The system spans several domains:

- Recruiting CRM and pipeline intelligence

- Program, roster, and team management

- Athlete development and performance analytics

- High school coaching tools and knowledge base

- AI intelligence layer

- Stripe-based billing and subscriptions

Core Principles

1. **Reliability** – Features must support real-world, time-sensitive coaching workflows.

2. **Security** – Data access is enforced via Supabase RLS and strict API auth.

3. **Simplicity** – Prefer predictable patterns and clear abstractions over cleverness.

4. **Scalability** – Architecture must support many orgs, teams, and athletes.

5. **Extensibility** – Design modules that can extend to new sports and tools.

Contributing Workflow

1. **Create a branch** for your feature or bugfix.

2. **Implement** changes following the patterns in this handbook.

3. **Run tests and linting** locally.

4. **Open a pull request** with a clear description and screenshots for UI changes.

5. **Request review**, address feedback, and merge only when tests pass.

6. **Update docs** (`/docs/*`) when schema, APIs, or major flows change.

This handbook, together with `system-architecture.md`, `schema.md`, and `api-routes.md`, forms the core of our technical documentation.

Tech Stack Overview

Tech Stack Overview

This document summarizes the technologies used in the Recruiting Ecosystem.

Frontend

Next.js (App Router)

- React Server Components

- Nested layouts and route groups

- Server Actions for safe DB interaction

- Deployed on Vercel

UI Layer

- **ShadCN UI** for consistent, accessible primitives

- **Tailwind CSS** for utility-first styling

- Mild, purposeful animations for perceived quality

- Responsive layouts optimized for desktop first, then mobile

---

Backend

Supabase

- Postgres with Row Level Security (RLS)

- Auth (JWT-based, cookie-backed)

- SQL migrations and functions

- Storage for media assets (videos, images)

Stripe

- Checkout Sessions for org and athlete billing

- Customer Portal for subscription management

- Webhooks to synchronize subscription state into Postgres

AI Stack

- OpenAI models orchestrated via server-side endpoints

- Optional use of LangChain-style patterns for prompt composition

- AI used for:

- scout scores

- commit probability estimates

- athlete summaries

- practice plan generation

- HS coaching assistance

Tooling

- **TypeScript** for static typing

- **ESLint + Prettier** for linting and formatting

- **Git + GitHub** for version control and reviews

- **Vercel** for deployments and preview environments

- **Supabase CLI** for migrations and local testing

- **Stripe CLI** for webhook and billing testing

Use this stack consistently; new dependencies should be justified and documented.

Project Structure

Project Structure

This document describes the high-level file and folder layout of the project.

```text

/

■■ app/

■ ■■ (site)/

■ ■■ api/

■ ■ ■■ me/

■ ■ ■■ athletes/

■ ■ ■■ recruiting/

■ ■ ■■ coach-tools/

■ ■ ■■ ai/

■ ■ ■■ stripe/

■ ■■ billing/

■ ■■ dashboard/

■

■■ components/

■■ lib/

■ ■■ supabase/

■ ■■ stripe/

■ ■■ ai/

■ ■■ auth/

■ ■■ utils/
```

■

■■ types/

■■ docs/

■ ■■ schema.md

■ ■■ api-routes.md

■ ■■ system-architecture.md

■ ■■ developer-handbook/

■

■■ public/

■■ .env.local

```

---

## Module Boundaries

- **`app/`**

- Contains Next.js routes, layouts, and page logic.

- All API routes live under `app/api/*/route.ts`.

- UI-facing pages live under route groups such as `/dashboard`, `/billing`, etc.

- **`components/`**

- Reusable UI components (forms, tables, cards, dialogs).

- Avoid business logic; keep components presentational or thin.

- **`lib/`**

- Cross-cutting libraries, helpers, and service wrappers:

- `lib/supabase/` – server and client Supabase clients

- `lib/stripe/` – Stripe client and helpers
```

- `lib/ai/` – AI orchestration logic

- `lib/auth/` – auth helpers and middleware

- `lib/utils/` – generic helpers

- **`types/`**

- Shared TypeScript interfaces and types.

- DB row types, domain models, and API payloads.

- **`docs/`**

- Architecture, schema, API routes, and this handbook.

- Keep in sync with actual implementation.

Naming Conventions

- Use `camelCase` for variables and functions.

- Use `PascalCase` for React components and TypeScript types.

- Use `kebab-case` for filenames (except React components where appropriate).

- API route directories mirror path segments, e.g. `/api/recruiting/board` → `app/api/recruiting/board/route.ts`.

Keeping this structure clean and consistent makes it easier for new developers to onboard.

Environment Setup

Environment Setup

Follow these steps to set up a local development environment.

Prerequisites

- Node.js (LTS)

- Git

- A Supabase project

- A Stripe account with test mode enabled

- Vercel account (for deployment, optional at first)

---

macOS Setup

```bash

# Install Homebrew if needed

/bin/bash -c "$(curl -fsSL
https://raw.githubusercontent.com/Homebrew/install/HEAD/install.sh)"

brew install node

brew install --cask visual-studio-code

brew install supabase/tap/supabase

brew install stripe/stripe-cli/stripe

```

Windows Setup

- Install Node.js from the official installer.

- Install VS Code.

- Install Supabase CLI via npm:

```bash

npm install supabase --global

```

- Install Stripe CLI via the Stripe docs instructions.

Clone the Repository

```bash

git clone <repo-url> xc-ecosystem

cd xc-ecosystem

```

Install Dependencies

```bash

npm install

# or

pnpm install

```

Configure Environment Variables

Create `.env.local` in the project root and populate it as described in `environment-variables.md`.

Run the App

```bash
npm run dev
```

The app runs at `http://localhost:3000`.

Supabase Setup

```bash
supabase login
supabase link --project-ref <your-project-ref>
```

Stripe CLI

```bash
stripe login
stripe listen --forward-to localhost:3000/api/stripe/webhook
```

You now have a functioning local environment ready for development.

Environment Variables

Environment Variables

This document lists required environment variables and their purposes.

All secrets must be stored in `.env.local` (local) and in the Vercel dashboard (production/preview), never committed to Git.

Supabase

```env

NEXT_PUBLIC_SUPABASE_URL=

NEXT_PUBLIC_SUPABASE_ANON_KEY=

SUPABASE_SERVICE_ROLE_KEY=

SUPABASE_JWT_SECRET=

```

- `NEXT_PUBLIC_SUPABASE_URL` – Public URL for Supabase project.

- `NEXT_PUBLIC_SUPABASE_ANON_KEY` – Public anon key used in browser.

- `SUPABASE_SERVICE_ROLE_KEY` – Secret key for server-side operations (e.g., webhooks).

- `SUPABASE_JWT_SECRET` – Auth JWT secret (from Supabase settings).

Stripe

```env

STRIPE_SECRET_KEY=

STRIPE_WEBHOOK_SECRET=

STRIPE_PRICE_HS_ATHLETE_BASIC=

STRIPE_PRICE_HS_ATHLETE_PRO=
```

STRIPE_PRICE_HS_ATHLETE_ELITE=

STRIPE_PRICE_HS_STARTER=

STRIPE_PRICE_HS_PRO=

STRIPE_PRICE_HS_ELITE=

STRIPE_PRICE_COLLEGE_STARTER=

STRIPE_PRICE_COLLEGE_PRO=

STRIPE_PRICE_COLLEGE_ELITE=

```

- `STRIPE_SECRET_KEY` – Secret API key for making Stripe calls.

- `STRIPE_WEBHOOK_SECRET` – Signing secret used to verify incoming webhooks.

- `STRIPE_PRICE_*` – Price IDs for each product plan in Stripe.

---

## General Rules

- Never expose `SUPABASE_SERVICE_ROLE_KEY` or `STRIPE_SECRET_KEY` in client components.

- Only server routes, server actions, or API handlers may use secret keys.

- When deploying to Vercel, set environment variables via the dashboard or `vercel env` CLI.
```

Supabase Architecture & RLS

Supabase Architecture & RLS

Supabase provides our Postgres database, authentication, storage, and Row Level Security (RLS).

This document summarizes the high-level design; see `schema.md` for detailed tables.

Key Domains

- **Users & Orgs**

- **Programs & Teams**

- **Athletes & Performance**

- **Recruiting & Pipeline**

- **High School Coaching Tools**

- **AI Intelligence Layer**

- **Billing & Subscriptions**

Auth Model

- Supabase Auth manages email/password or OAuth logins.

- Each auth user maps to a row in the `users` table via `auth_id = auth.uid()`.

- Application roles (`coach` vs `athlete`) are stored in the `users` table.

RLS Principles

1. Users can only select and update their own row in `users`.

2. Coaches can only access `orgs`, `teams`, `athletes`, and recruiting data where they have membership via `org_memberships`.

3. Athletes can only access their own athlete profile and training data.

4. Billing tables (`org_subscriptions`, `athlete_subscriptions`) are readable only by owners and internal admin contexts.

5. AI output tables reference the same RLS logic as their underlying entities (athlete/team/org).

Supabase Clients

Server-side

Use a server client that reads auth from cookies and respects RLS:

```ts
import { createSupabaseServerClient } from "@/lib/supabase/server";

const supabase = createSupabaseServerClient();

const { data: { user } } = await supabase.auth.getUser();
```

Client-side

Use browser helpers only when necessary:

```ts
import { createBrowserClient } from "@supabase/auth-helpers-nextjs";
```

Prefer server components for data fetching whenever possible to keep logic secure and centralized.

Migrations

- Use Supabase CLI migrations (`supabase db diff`, `supabase db push`) to evolve schema.

- Do not hand-edit tables via the Supabase web UI without also updating migration scripts and `schema.md`.

API Design Guide

API Design Guide

This document defines patterns and conventions for building API routes under `app/api/*`.

General Principles

- Use REST-style endpoints grouped by domain:

- `/api/me`, `/api/auth/*`

- `/api/athletes/*`

- `/api/recruiting/*`

- `/api/coach-tools/*`

- `/api/ai/*`

- `/api/stripe/*`

- All responses use JSON.

- All input must be validated on the server (e.g., with Zod).

- Do not trust client-side validation.

Error Handling

Always return structured errors:

```ts

import { NextResponse } from "next/server";

return NextResponse.json(

{ error: "Invalid request" },

{ status: 400 }

);
```

```

Avoid throwing unhandled exceptions; catch errors, log them on the server, and return a clear error message to clients.

---

## Auth Context

Within API routes, authenticate via Supabase:

```ts

const supabase = createSupabaseServerClient();

const { data: { user } } = await supabase.auth.getUser();

if (!user) {

return NextResponse.json({ error: "Unauthorized" }, { status: 401 });

}

```

Use `user.id` or `user.email` to look up application-level records (e.g., `users` table).

Request Validation

Use `zod` or similar for request bodies:

```ts

import { z } from "zod";

const BodySchema = z.object({

scope: z.enum(["org", "athlete"]),

ownerId: z.string().uuid(),

planCode: z.string(),

});

const json = await request.json();

const body = BodySchema.parse(json);

```

This ensures runtime safety and clear error messaging.

Stripe Webhook Special Case

The Stripe webhook route must:

- Read the raw request body.

- Validate the signature with `STRIPE_WEBHOOK_SECRET`.

- Use a Supabase service-role client for writing subscription rows.

All other routes should rely on RLS-aware clients, not the service role.

Follow these patterns to keep the API consistent and secure.
```

Billing & Subscriptions

Billing & Subscriptions

Billing is handled via Stripe. The application supports **org-level** and **athlete-level** subscriptions.

Data Model

Two main tables:

- `org_subscriptions`

- `athlete_subscriptions`

Each record stores:

- Stripe customer ID

- Stripe subscription ID

- plan code

- status

- current period end

- cancel at period end flag

See `schema.md` for full column definitions.

Flow: Org or Athlete Upgrade

1. User clicks **Upgrade** in the UI.

2. Frontend calls `POST /billing/create-checkout-session` with:

- `scope` (`"org"` or `"athlete"`)

- `ownerId` (org_id or user_id)

- `planCode`

3. Route validates input and creates a Stripe Checkout Session with metadata (scope, ownerId, planCode).

4. User completes checkout on Stripe.

5. Stripe sends `checkout.session.completed` and subscription events to `/api/stripe/webhook`.

6. Webhook verifies signature, reads metadata, and upserts a row in:

- `org_subscriptions` (if scope = `org`)

- `athlete_subscriptions` (if scope = `athlete`)

7. `/api/me` reads subscription tables to determine plan and feature access.

Webhook Events

We handle:

- `checkout.session.completed`

- `customer.subscription.created`

- `customer.subscription.updated`

- `customer.subscription.deleted`

The webhook is the **only** place that writes subscription state. UI must never “guess” from Stripe directly.

Plan Codes

Plan codes map to Stripe Price IDs by env vars:

- `hs_athlete_basic`, `hs_athlete_pro`, `hs_athlete_elite`

- `hs_starter`, `hs_pro`, `hs_elite`

- `college_starter`, `college_pro`, `college_elite`

These codes are used across:

- UI feature gating

- `/billing/create-checkout-session`

- Webhook subscription updates

Always keep plan code definitions in sync between Stripe, env vars, and application logic.

AI Development Guide

AI Development Guide

The AI Intelligence Layer provides context-aware intelligence for recruiting, athlete development, and coaching.

Supported AI Features

- **Athlete summaries** – Quick natural-language overviews of an athlete.

- **Scout scores** – AI-generated evaluation score for a recruit.

- **Commit probability** – Estimated likelihood an athlete commits to a program.

- **Practice plans** – Generated suggestions for workouts and sessions.

- **HS coaching assistance** – Training, drill, and intervention suggestions.

Design Principles

1. **Server-only** – All AI calls must be made server-side; never expose API keys to the browser.

2. **Deterministic interfaces** – AI endpoints should behave like normal APIs, returning typed outputs.

3. **Explainability** – Include explanations text where possible (e.g., why a scout score or probability was chosen).

4. **Safety & alignment** – Avoid generating inappropriate content; enforce domain limits (sports coaching, training, performance).

Example Pattern

```ts

// app/api/ai/athlete-summary/route.ts

import { NextResponse } from "next/server";
```

```
import { createSupabaseServerClient } from "@/lib/supabase/server";
import { generateAthleteSummary } from "@/lib/ai/athlete";
export async function POST(request: Request) {
const supabase = createSupabaseServerClient();
const { data: { user } } = await supabase.auth.getUser();
if (!user) {
return NextResponse.json({ error: "Unauthorized" }, { status: 401 });
}
const { athleteId } = await request.json();
// fetch athlete data...
const summary = await generateAthleteSummary(athleteData);
return NextResponse.json({ summary });
}
```

Prompting

Prompts should:

- Include sufficient context: athlete stats, history, goals.
- Provide clear instructions to the model.
- Specify the tone (professional, coach-facing).
- Limit output length to something usable in UI.

Store prompts and experiments in `lib/ai/` or separate notes so the system is reproducible.

UI/UX Guidelines

UI / UX Guidelines

This document provides design and UX standards for the Recruiting Ecosystem.

Design System

- **ShadCN UI** for base components

- **Tailwind CSS** for layout and styling

- **Rounded corners** and **soft shadows** for cards and modals

- Color palette aligned with neutral backgrounds and sport-accent colors

---

Layout

- Desktop-first with adaptable mobile layouts.

- Left-hand navigation for primary sections:

- Dashboard

- Recruiting

- Roster & Teams

- Athlete Database

- Coach Tools

- Analytics

- AI Assistant

- Settings

Use clear section headings and breadcrumbs where appropriate.

Drag & Drop

Drag-and-drop is central to:

- Recruiting boards

- Depth charts

- Practice plan blocks

Guidelines:

- Show visual feedback while dragging (card lift, shadow).

- Animate drop transitions smoothly.

- Avoid jarring animations or excessive motion.

- Allow undo or easy manual correction.

Animations

Use **mild animations** only:

- Fade/slide for modals and drawers

- Subtle hover transitions on cards and buttons

- Skeleton loaders while fetching data

Animations should reinforce clarity, not distract.

Accessibility

- Use semantic HTML where possible.

- Provide `aria-labels` for interactive elements.

- Maintain sufficient contrast for text and key UI components.

- Ensure forms and dialogs are keyboard-navigable.

Consistent UX improves coach adoption and reduces training time.

Testing & Quality Assurance

Testing & Quality Assurance

This document outlines the testing strategy for the Recruiting Ecosystem.

Types of Tests

1. **Unit Tests**

- Utility functions (e.g., plan mapping, score normalization)

- Zod schemas and helpers

2. **Integration Tests**

- API routes (e.g., `/api/me`, `/api/recruiting/board`)

- RLS behaviors using Supabase test users

3. **End-to-End (E2E) Tests (optional / phased)**

- Critical flows: login, upgrade plan, move recruits, view athlete profile

4. **Manual QA**

- Stripe test flows

- AI endpoint behavior sanity checks

Recommended Tools

- Jest or Vitest for unit/integration tests

- Playwright or Cypress for E2E tests

- Supabase CLI for local DB and RLS verification

Stripe Testing

Use the Stripe CLI:

```bash
stripe listen --forward-to localhost:3000/api/stripe/webhook
```

Use test cards provided by Stripe to simulate:

- Successful payments
- Failures
- Subscription cancellations

Verify that:

- `org_subscriptions` or `athlete_subscriptions` rows are updated.
- `/api/me` reflects new subscription state.

Quality Gates

Before merging:

- All tests must pass.
- Linting and formatting must succeed.
- Major schema or API changes must update:
- `schema.md`
- `api-routes.md`
- `system-architecture.md` (if relevant)

Deployment Guide

Deployment Guide

This document covers deploying the Recruiting Ecosystem using Vercel and Supabase.

Vercel

- Connect the GitHub repo to Vercel.

- Configure:

- Production branch (e.g., `main`)

- Preview deployments for feature branches

- Set environment variables in Vercel:

- Supabase keys

- Stripe keys

- Plan price IDs

Never commit `.env.local` to Git.

Supabase

- Use a single primary Supabase project for production.

- Optionally maintain a separate Supabase project for staging.

- Run migrations with:

```bash

supabase db push

```

- Keep `schema.md` updated to match production.

Deployment Steps

1. Merge PR into `main`.

2. Vercel builds and deploys automatically.

3. Verify:

- `/api/health` (if implemented)

- Key flows (login, `/api/me`, dashboard load)

4. Monitor logs in Vercel and Supabase.

Stripe Live Mode

When ready to go live:

1. Create live products and prices in Stripe.

2. Set live `STRIPE_SECRET_KEY` and `STRIPE_WEBHOOK_SECRET` in Vercel.

3. Update env variables for live price IDs.

4. Test carefully with real (small) transactions if appropriate.

Coordinate live mode changes with any marketing or onboarding for real users.

Troubleshooting

Troubleshooting

This document lists common issues and their likely fixes.

RLS: "new row violates row-level security policy"

Cause:

- Insert or update is blocked by a Row Level Security policy.

Fix:

- Ensure the Supabase client is using an authenticated user with the right role.

- Confirm the policy conditions match the insert/update (e.g., `new.auth_id = auth.uid()`).

- Only use the service-role key in trusted backend contexts like Stripe webhooks.

Stripe Webhook Errors (401 / 400)

Symptoms:

- Stripe dashboard shows webhook failures.

- Logs indicate signature verification failures or JSON parsing issues.

Fix:

- Read the raw request body (without JSON parsing first).

- Verify using the official Stripe library and `STRIPE_WEBHOOK_SECRET`.

- Ensure Vercel / Next.js route is configured to not consume the body prematurely.

"next: command not found"

Cause:

- Node modules not installed or PATH misconfigured.

Fix:

- Run `npm install`.

- Confirm `node_modules/.bin` is present.

- Use `npm run dev` (which calls the local `next` binary).

Supabase Connection Issues

Symptoms:

- API routes fail with connection errors.

- Local dev cannot reach Supabase.

Fix:

- Verify `NEXT_PUBLIC_SUPABASE_URL` and keys.

- Check network/firewall rules.

- Make sure Supabase project is running and not paused.

As you encounter new issues, add them here with clear symptoms and fixes.

Glossary

Glossary

A shared vocabulary for the Recruiting Ecosystem.

Core Domain Terms

Org

A school, club, or institution that houses one or more sports programs.

Program

A sport-specific unit within an org (e.g., "WCU Men's Track & Field").

Team

A sub-unit or event-group within a program (e.g., Sprints, Distance, Throws, Jumps).

Athlete

A student-athlete profile in the system. May be a high school or college athlete.

Recruit

An athlete being evaluated by a college program as a potential addition to the roster.

Board

A drag-and-drop recruiting board that shows recruits in stages.

Coaching & Training Terms

HS Coaching Tools

Season planning, practice planning, knowledge base, and other tools specifically aimed at improving high school coaching.

Knowledge Base

Collection of drills, articles, videos, and coaching resources provided within the system.

Practice Plan

A structured template or schedule for a single training session.

Season Plan

A macro-level view of training phases (pre-season, in-season, championship, etc.).

AI & Analytics Terms

AI Layer

The intelligence system that provides scout scores, commit probabilities, practice suggestions, and summaries.

Scout Score

An AI-assisted evaluation score for a recruit based on performance and qualitative data.

Commit Probability

The AI-estimated likelihood that a recruit will commit to a particular program.

Billing Terms

Subscription Scope

Whether a subscription applies to an entire org (`"org"`) or a single athlete (`"athlete"`).

Plan Code

Internal identifier for subscription plans (e.g., `hs_athlete_elite`, `college_pro`), mapped to Stripe prices.

Keeping terminology consistent across code, docs, and UI reduces confusion and speeds up development.