

Launch Readiness Review

Moveslot

Class booking and class-pack payments for independent fitness studios.

Verdict

Not ready to take payments yet

Three critical issues to fix first. None needs a rewrite: about three days of work with your AI tool, or two for a developer.

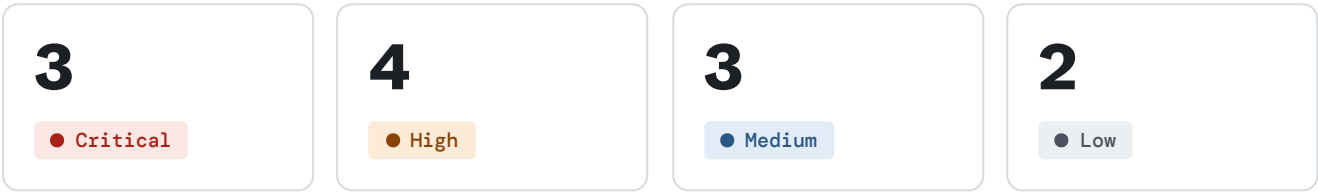
Prepared for	Moveslot founders
Review period	5 to 9 October 2026
Scope	<code>moveslot-web</code> at commit <code>a41c9e2</code> , Supabase project, Stripe account, production hosting
Built with	Lovable, React, Supabase, Stripe Checkout, Resend, OpenAI
Reviewer	Matthew Starkey, vibereview.io

Example report. Moveslot is a fictional app, and its code, data and findings are invented. This report shows the format and depth of a real Launch Readiness Review. Real reports are confidential and never published.

Summary

Moveslot is in better shape than most apps I review at this stage. The booking flow works, the data model is sensible, and using Stripe Checkout keeps card details off your servers entirely.

But three issues would let a stranger read every customer's details, change anything in your database, or buy a 20-class pack for \$1. All three are quick to fix. Fix the five "before launch" items on the next page and you're in a good position to go live.



Scorecard

Area	Rating	Findings	In short
Security and data	Needs work	2 2 1	Supabase is set up as if every visitor can be trusted. Locking down one key and switching on row-level security closes the biggest gaps.
Payments and vendors	Needs work	1 1 1	Stripe Checkout is the right choice. The code around it trusts the browser too much. Lock-in risk is low: the stack is portable.
Architecture and scale	Fair	1 1	Comfortably handles launch traffic. Overbooking popular classes is the one real risk, and nobody would know if something broke.
Code and delivery	Fair	1 1	Readable TypeScript with a clear structure. Nothing stops a broken change from going straight to customers.

Fix plan

The order I'd tackle things in. Effort assumes you're making the changes with your AI tool and testing each one; a developer would be faster.

Before launch

About 3 days. Don't take a payment until these are done.

VR-01	● Critical	Your database admin key is shipped to every visitor's browser	2 to 3 hours
VR-02	● Critical	Any signed-in user can read every customer's details	3 to 4 hours
VR-03	● Critical	Customers can choose their own price at checkout	2 hours
VR-04	● High	Class credits are granted by the success page, not by Stripe	Half a day
VR-07	● High	Two people can book the last place in a class	2 to 3 hours

First 30 days

About 3 days. Set the OpenAI budget limit in VR-05 today: it takes five minutes.

VR-05	● High	Nothing stops someone running up your OpenAI bill or spamming sign-ups	2 to 3 hours
VR-06	● High	Your production database can't be rebuilt from your code	Half a day
VR-08	● Medium	You'll hear about problems from customers, not from your app	1 to 2 hours
VR-09	● Medium	Every change goes straight to production, untested	Half a day
VR-10	● Medium	You collect personal data you don't need, with no way to delete it	Half a day

When you have time

About 2 hours.

VR-11	● Low	The <code>.env</code> file is committed to the repository	30 minutes
VR-12	● Low	AI recommendations are regenerated on every dashboard visit	1 hour

Findings

VR-01	● Critical	Security and data	Effort: 2 to 3 hours
-------	------------	-------------------	----------------------

Your database admin key is shipped to every visitor's browser

What's wrong

The admin dashboard creates its Supabase connection with the *service role* key. Because the dashboard is part of the same React app as the customer site, that key is bundled into the JavaScript that every visitor downloads, whether or not they ever see the dashboard.

Why it matters

The service role key skips all of Supabase's security rules. Anyone who opens their browser's developer tools can copy it and read, change or delete every row in your database: customers, bookings, credits and payment records. Bots scan public sites for exactly this, so assume it would be found within days of launch.

Evidence

src/integrations/supabase/admin.ts, lines 3 to 6

```
import { createClient } from "@supabase/supabase-js";

export const supabaseAdmin = createClient(
  "https://qkzrmvwtplxhsdfa.supabase.co",
  "eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXZWQ0LXB5YWRkb2UiLCJ1bmMiOiJkaWkiLCJ1dHUiOiJ1cm9udGVudC5yZWFkdjEwMDEyMDA0NDAxOTg5LnVzbnVlcnR5cy5jb20ifQ==" // admin key, bundled for every visitor
);
```

Also present in the production build: `assets/index-8f2c1d.js`

The fix

1. Rotate the service role key now in Supabase (Project Settings, API). Treat the current one as public.
2. Delete `admin.ts`. Move the three admin actions (cancel a class, refund credits, edit studio details) into Supabase Edge Functions. Each one checks the caller is an admin before doing anything. The service role key lives only in the functions' secrets.
3. Rebuild the site and search the output for `service_role` to confirm it's gone.

Prompt for your AI tool

Remove `src/integrations/supabase/admin.ts` and every use of `supabaseAdmin`. For each admin action (`cancelClass`, `refundCredits`, `updateStudio`), create a Supabase Edge Function that reads the service role key from `Deno.env`, verifies the caller's JWT, checks that `profiles.role` is 'admin', and only then performs the action. Update the admin dashboard to call these functions with `supabase.functions.invoke`. No file under `src/` may import or contain the service role key.

Any signed-in user can read every customer's details

What's wrong

Row-level security (RLS) is switched off on the `bookings` and `profiles` tables. RLS is what tells Supabase which rows each user may see. Without it, the public key that's meant to be in the browser can query the whole table.

Why it matters

I signed up on staging as an ordinary customer and, with one line in the browser console, downloaded all 386 customer profiles (names, emails and phone numbers) and all 1,240 bookings. In production, that's a personal data breach you'd have to report to the regulator and to your customers.

Evidence

supabase/migrations/20260812_bookings.sql

```
create table public.bookings ( ... );  
alter table public.bookings disable row level security;
```

Run in the browser console while signed in as a regular customer

```
await supabase.from("profiles").select("full_name, email, phone")  
// → 386 rows
```

The fix

Enable RLS on every table in the `public` schema, then add policies: customers can see and create only their own bookings, studio staff can see bookings for their own studio's classes, and each profile is readable only by its owner and that studio's staff. Afterwards, Supabase's Security Advisor (Database, Advisors) should show no RLS warnings.



Prompt for your AI tool

Write a Supabase migration that enables row level security on every table in the public schema. Add policies so that: a user can select, insert and delete only bookings where `user_id = auth.uid()`; a user can select and update only their own row in profiles; staff (profiles.role = 'staff' with a matching studio_id) can select bookings and profiles for their own studio. Don't add any policy that uses (true). Then list every table with its policies so I can check them.

Customers can choose their own price at checkout

What's wrong

The `create-checkout` function builds the Stripe payment from an amount sent by the browser, instead of looking up the pack's price itself.

Why it matters

Anyone can edit the request in their browser and buy the 20-class pack (\$240) for \$1. Stripe charges whatever it's told, and the full 20 credits are granted. You'd only notice when your payouts didn't add up.

Evidence

supabase/functions/create-checkout/index.ts, lines 12 to 21

```
const { packId, amount } = await req.json();

const session = await stripe.checkout.sessions.create({
  mode: "payment",
  line_items: [{
    price_data: {
      currency: "usd",
      unit_amount: amount, // set by the browser
      product_data: { name: packId },
    },
    quantity: 1,
  }],
});
```

The fix

Create a Product and Price in Stripe for each class pack and store the Stripe price ID against each pack in your `packs` table. The function should accept only `packId`, look up the price ID on the server, and ignore any amount the browser sends.



Prompt for your AI tool

In `supabase/functions/create-checkout`, stop reading `amount` from the request. Accept only `packId`, look the pack up in the `packs` table (add a `stripe_price_id` column in a migration), return a 400 error if it doesn't exist or isn't active, and create the Checkout Session with `line_items: [{ price: pack.stripe_price_id, quantity: 1 }]`. Add metadata with the `user_id` and `pack_id` to the session.

VR-04 ● High Payments and vendors Effort: half a day

Class credits are granted by the success page, not by Stripe

What's wrong

After checkout, the `/success` page calls the `add-credits` function with the session ID from the URL. That function adds credits without checking that the payment went through, and it will happily credit the same session twice. A `stripe-webhook` function exists, but it doesn't verify that requests really come from Stripe, and it isn't used to grant credits.

Why it matters

Reloading the success page adds the credits again. Someone who abandons checkout can still open `/success` with their session ID and get the credits free. And anyone who finds the webhook URL can send fake "payment succeeded" events.

Evidence

src/pages/Success.tsx, lines 14 to 18

```
useEffect(() => {  
  supabase.functions.invoke("add-credits", {  
    body: { sessionId: params.get("session_id") },  
  });  
}, []);
```

supabase/functions/stripe-webhook/index.ts, line 9

```
const event = await req.json(); // signature never checked
```

The fix

Grant credits in exactly one place: the webhook, on `checkout.session.completed` when `payment_status` is `paid`. Verify every event with Stripe's signing secret. Record each session ID in a table with a unique constraint so a payment can only be credited once. Make the success page read-only: it shows the new balance and grants nothing.



Prompt for your AI tool

Delete the add-credits function and remove its call from Success.tsx; the success page should only display the user's current credit balance. In supabase/functions/stripe-webhook, verify every request with `stripe.webhooks.constructEventAsync` using the `STRIPE_WEBHOOK_SECRET` env var, and reject anything that fails. On `checkout.session.completed` with `payment_status` 'paid', insert the session id into a new `processed_payments` table (unique on `session_id`) and add the pack's credits to the user from the session metadata, both in one database transaction. If the session id already exists, do nothing.

VR-05 ● High Security and data Effort: 2 to 3 hours

Nothing stops someone running up your OpenAI bill or spamming sign-ups

What's wrong

The `recommendations` function calls OpenAI on every request. It doesn't check who's asking or how often, and the sign-up form has no bot protection.

Why it matters

A simple script calling the function in a loop costs you money on every call, and OpenAI won't stop it unless you've set a spending limit. Scripted sign-ups fill your database with junk accounts, and the confirmation emails can get your sending domain flagged as spam, which stops real customers receiving their booking emails.

Evidence

supabase/functions/recommendations/index.ts, lines 6 to 9

```
serve(async (req) => {  
  const { goals } = await req.json(); // no auth check, no limit  
  const completion = await openai.chat.completions.create({ ... });
```

The fix

1. Today: set a monthly budget limit in your OpenAI account. It takes five minutes.
2. Require a signed-in user for the recommendations function and allow each user a few calls per hour.
3. Turn on bot protection for sign-up in Supabase (Authentication, Attack Protection) using Cloudflare Turnstile.



Prompt for your AI tool

In supabase/functions/recommendations, reject requests without a valid user JWT. Add a rate_limits table and allow each user at most 5 calls per hour, returning HTTP 429 with a friendly message when they exceed it. Then add Cloudflare Turnstile to the sign-up form and pass the captcha token to supabase.auth.signUp in options.captchaToken.

VR-06 ● High Security and data Effort: half a day

Your production database can't be rebuilt from your code

What's wrong

Most database changes were made by clicking in the Supabase dashboard. The `supabase/migrations` folder holds three migrations, the newest six weeks old, while production has four tables and two functions that aren't in any of them. The project's plan has no point-in-time recovery, and a restore has never been tested.

Why it matters

If production is damaged by a bad change, an accidental delete or a leaked key (see VR-01), you can't reliably rebuild it. Your test and production databases will also keep drifting apart, so changes that work in testing break in production.

Evidence

Comparing production with the migrations folder

```
$ supabase db diff --linked  
+ create table public.waitlist (...)  
+ create table public.studio_settings (...)  
+ create table public.promo_codes (...)  
+ create table public.audit_log (...)  
+ create function public.get_studio_stats (...)  
+ create function public.expire_credits (...)
```

The fix

Run `supabase db pull` to capture production as a migration and commit it. From then on, make every database change as a migration that goes through code review. Before launch, move to a plan with daily backups and point-in-time recovery, and restore a backup into a new project once, so you know it works before you need it.



Prompt for your AI tool

From now on, never change the database through the Supabase dashboard. Every schema change must be a new SQL file in `supabase/migrations` with a timestamped name, and must include any RLS policies for new tables. Before writing a migration, read the existing migrations to understand the current schema.

VR-07 ● High Architecture and scale Effort: 2 to 3 hours

Two people can book the last place in a class

What's wrong

The booking button first asks how many places are left, then creates the booking in a separate request. In between, someone else can take the same place. Credits are deducted in a third request, so a failure part-way through can leave a booking with no credit taken, or a credit taken with no booking.

Why it matters

Popular classes will be oversold the moment two people book at once, which happens exactly when demand peaks: a new timetable goes live, or it's 5:59 for the 6pm class. The studio then has to turn away someone who has already paid.

Evidence

`src/hooks/useBookClass.ts`, lines 18 to 26

```
const { count } = await supabase.from("bookings")
  .select("*, { count: 'exact', head: true }")
  .eq("class_id", classId);
if (count < cls.capacity) { // someone else can book here
  await supabase.from("bookings").insert({ class_id: classId, user_id });
  await supabase.rpc("deduct_credit", { user_id });
}
```

The fix

Do the whole booking in one Postgres function that runs as a single transaction: lock the class row, check capacity and credits, deduct the credit and insert the booking. The app calls it once. Add a unique constraint on `(class_id, user_id)` so nobody can book the same class twice.



Prompt for your AI tool

Create a Postgres function `book_class(p_class_id uuid)` in a new migration. It should use `auth.uid()` as the user, lock the class row with `SELECT ... FOR UPDATE`, check the class isn't full and the user has at least one credit, then deduct one credit and insert the booking, raising a clear error otherwise. Add a unique constraint on bookings (`class_id`, `user_id`). Replace the logic in `useBookClass.ts` with a single `supabase.rpc('book_class')` call and show the error message to the user.

VR-08 ● Medium Architecture and scale Effort: 1 to 2 hours

You'll hear about problems from customers, not from your app

What's wrong

There's no error tracking, no uptime monitoring and no alerts. Errors in Edge Functions only reach Supabase's logs, which nobody watches. Errors in the browser aren't recorded anywhere: 14 places in the code catch an error and only log it to the console.

Why it matters

When bookings or payments start failing, you won't know until a customer complains, and you won't have the details you need to fix it quickly.

Evidence

`src/hooks/useBookClass.ts`, line 31 (one of 14 similar)

```
} catch (e) { console.log(e) } // the user sees nothing, you see nothing
```

The fix

Add Sentry to the React app and the Edge Functions (the free tier is plenty for now). Add an uptime check on the home page and the checkout function with UptimeRobot or Better Stack, with alerts to your phone. Replace silent catches with errors that reach Sentry and show the user a message they can act on.

VR-09 ● Medium Code and delivery Effort: half a day

Every change goes straight to production, untested

What's wrong

Lovable publishes changes straight to the live site. There's no review step, no automated tests and no type check, and the main branch isn't protected. The TypeScript compiler currently reports 23 errors, including three in the payment code.

Why it matters

A prompt that "tidies up" the booking hook can break payments for every customer, and the first test is a real customer trying to pay.

Evidence

```
$ ls .github/workflows      → not found
$ find src -name "*.test.*" → 0 files
$ npx tsc --noEmit          → 23 errors in 9 files
```

The fix

Protect the main branch so changes arrive through pull requests. Add a GitHub Actions workflow that runs the type check, the linter and a handful of tests covering the three flows that make you money: signing up, buying a pack and booking a class. Use a Supabase branch or a separate staging project so changes can be tried before they go live.

VR-10 ● Medium Security and data Effort: half a day, plus writing the policy

You collect personal data you don't need, with no way to delete it

What's wrong

Sign-up requires a date of birth and a phone number, and no feature uses either. There's no privacy policy and no way for customers to delete their account. Customers' names and fitness goals are sent to OpenAI, and this isn't mentioned anywhere.

Why it matters

Under UK and EU GDPR you need a reason for every piece of personal data you collect, must tell people who processes it, and must delete it on request. Studios will ask about this before they sign up, and it's an easy question to fail.

The fix

- Stop collecting date of birth. Make phone number optional, and say why you'd use it.
- Publish a privacy policy that names your processors: Supabase, Stripe, Resend and OpenAI.
- Add "Delete my account": remove the profile and anonymise past bookings so studio records still add up.
- Send OpenAI the customer's goals only, never their name or contact details.

VR-11 ● Low Code and delivery Effort: 30 minutes

The `.env` file is committed to the repository

What's wrong

`.env` isn't in `.gitignore` and is committed. Today it only holds the Supabase URL and the public anon key, which are designed to be public, so nothing is exposed. But the next secret anyone adds to it will be pushed too, and stays in the history even after it's deleted.

The fix

Add `.env` to `.gitignore`, remove it from the repository with `git rm --cached .env`, and commit a `.env.example` with empty values instead. Add gitleaks as a pre-commit hook and a CI step so secrets are caught before they leave your machine.

VR-12 ● Low Payments and vendors Effort: 1 hour

AI recommendations are regenerated on every dashboard visit

What's wrong

The dashboard asks for fresh class recommendations every time it loads, and the result is never stored. A customer's goals rarely change, but every visit pays for a new OpenAI call.

Why it matters

The cost is small today. But your AI bill grows with page views rather than with customers, so a busy month costs far more than it should, and it's the line item most likely to surprise you as you grow.

The fix

Store each customer's recommendations with a timestamp. Only generate new ones when their goals change or the stored ones are more than a week old.

What's already good

Worth knowing, so these don't get "fixed" by accident.

- ✓ **Card details never touch your servers.** Stripe Checkout handles payment entirely on Stripe's side, which puts you in the lightest PCI compliance category (SAQ A). Keep it that way.
 - ✓ **Supabase Auth instead of home-made login.** Password storage, reset emails and sessions are handled by a well-tested service rather than custom code.
 - ✓ **A sensible data model.** Studios, classes, sessions and bookings are separate tables linked by foreign keys. This will hold up as you add studios and features.
 - ✓ **Readable TypeScript with a clear structure.** Components, hooks and pages are organised predictably, so a developer you hire can find their way around quickly.
 - ✓ **Low lock-in.** Supabase is standard Postgres and the frontend is plain React and Vite. You can export the code from Lovable and host it anywhere, and move the database if you ever need to.
-

How this review was done

Scope	<code>moveslot-web</code> repository at commit <code>a41c9e2</code> (main, 6 October 2026). Read-only access to the Supabase project (schema, policies, Edge Functions, logs), the Stripe account and the hosting settings. Staging environment for hands-on testing.
Method	Manual review of all application code, database migrations, RLS policies and Edge Functions. Configuration review of Supabase, Stripe and hosting. Automated checks with gitleaks (secrets), npm audit (dependencies) and Supabase's Security and Performance Advisors. Hands-on testing as a signed-in customer and a studio admin on staging.
Not in scope	Load testing, penetration testing of Supabase or Stripe infrastructure, mobile apps, and the security of third-party services themselves.

Severity levels

● Critical	Can be exploited today with little skill, exposing customer data or money. Fix before launch.
● High	A likely path to data loss, financial loss or a serious outage. Fix before launch or soon after.
● Medium	Weakens your ability to prevent, notice or recover from problems. Fix in the first month.
● Low	Hygiene and efficiency. Fix when convenient.

Limits of this review

This review reflects the code and configuration at the commit above. It significantly reduces risk, but no review can guarantee there are no other issues, and new changes can introduce new ones. Re-check after major changes, or before a fundraiser or a large marketing push.



Next step: your walkthrough call. We'll go through this report together, agree the order of fixes, and I'll answer any questions about the changes. Matthew Starkey, hello@vibereview.io