Mohit Chimote

Product Manager & Architect — UK Regulated Financial Services

13 years at TCS building digital mortgage origination platforms for UK banks and building societies. This document collects six personal engineering projects — case studies in architecture, not just descriptions of what they do.

mohit.chimote@gmail.com linkedin.com/in/mohitchimote github.com/mohitchimote

mohitchimote.com

CONTENTS

InventPro

Multi-tenant SaaS inventory and billing platform — built first for a family member's paint shop, general enough for any small-retail trade managing stock and billing, running serverless end-to-end on Cloudflare's edge network.

• Live github.com/mohitchimote/InventPro

PROBLEM

My uncle runs a paint retail shop in India, and like most small retailers he was running it on paper ledgers and spreadsheets that didn't talk to each other — stock counts drifted from what was actually on the shelf, billing was manual, and there was no single place to see what was owed, what was low, or what a customer bought last time. Off-the-shelf inventory SaaS either assumes a much larger operation (multi-warehouse, barcode hardware, dedicated IT) or charges per-seat in a way that doesn't make sense for a single-counter shop.

InventPro was built to fit the actual shape of that business: one shop, a handful of staff, and a need to track stock, purchases, and billing without hiring anyone to run it. The brief was personal, but the build wasn't scaled down for it — multi-tenant from day one, with the same isolation and auth model a paying SaaS customer would expect. Nothing in the data model is paint-specific, so the same tenant shape has since picked up shops in other trades too.

INTENDED USERS

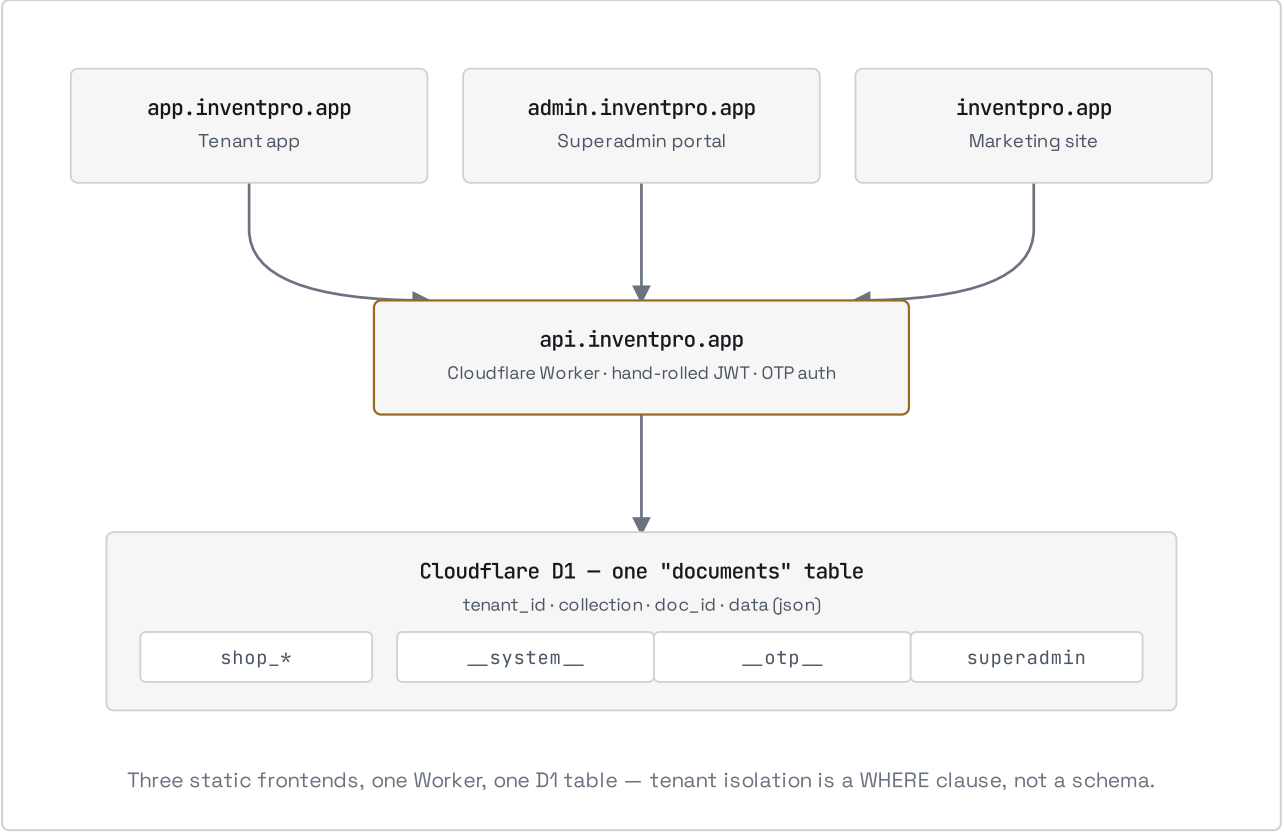
Independent small-retail shop owners and their counter staff, with a superadmin role for platform operations. Built first for my uncle's paint shop, but the tenant model isn't paint-specific — SKUs, stock movements, and billing work the same whether the shop sells paint, computer hardware, or anything else counted and sold over a counter.

MY ROLE

Solo engineering team, built and maintained alongside a full-time product role at TCS — product design, full-stack architecture and build (worker API, tenant app, superadmin portal, marketing site, CI/CD).

ARCHITECTURE & STACK

`JavaScript` `Cloudflare Workers` `Cloudflare D1` `Cloudflare Pages` `Web Crypto API` `Resend` `GitHub Actions`



Three static frontends, one Worker, one D1 table — tenant isolation is a WHERE clause, not a schema.

InventPro is deliberately edge-native — every part of it runs on Cloudflare's serverless platform (Workers, D1, Pages), with no server to provision or patch and infrastructure that scales with usage rather than needing to be sized upfront.

Three standalone frontends, one API. `app/` (tenant-facing), `admin/` (superadmin portal), and `website/` (marketing) are each a single HTML file with no build step — no bundler, no framework, deployed to Cloudflare Pages as-is. All three talk to one Cloudflare Worker (`worker/src/index.js`) that fronts a single Cloudflare D1 database.

One table, JSON documents. Rather than a normalized schema per feature, every tenant's data — SKUs, inward/outward stock movements, bills, customers, purchase orders, credit notes, audit log, settings — lives in one `documents` table (`tenant_id`, `collection`, `doc_id`, `data`), with `data` as a JSON blob. The D1 helpers (`dbFind`, `dbFindOne`, `dbUpsert`, `dbBatchUpsert`) are the entire data layer — no ORM. It trades some query flexibility for a schema that never needs a migration when a new collection shows up.

Tenant isolation is a WHERE clause. Every row is scoped by `tenant_id`, with reserved IDs for platform-level concerns: `__system__` holds the tenant registry and platform settings, `__otp__` holds short-lived OTP sessions, and each onboarded business gets its own `shop_*` id. A superadmin role can cross tenants by passing `?tenantId=` explicitly — there's no ambient "view everything" mode.

Auth is hand-rolled, on purpose. Cloudflare Workers don't have Node's `crypto` module, so JWT signing and verification are implemented directly against the Web Crypto API (`crypto.subtle`) — see the snippet below. Login is OTP-based: a 6-digit code is generated, AES-256-GCM encrypted at rest, emailed via Resend, and exchanged for a 24-hour HS256 JWT on verification (10-minute OTP TTL, 5 attempts max).

Tenant lifecycle is a small state machine. Self-registration lands a business in a pending queue; a superadmin approval creates the tenant and its owner user. Deletion is two-step — archive (soft delete, data retained) before a hard delete wipes every row for that `tenant_id`. Plan limits (`trial` / `free` / `pro` / `enterprise`) are enforced server-side at the points that matter — data sync and user creation — not just in the UI.

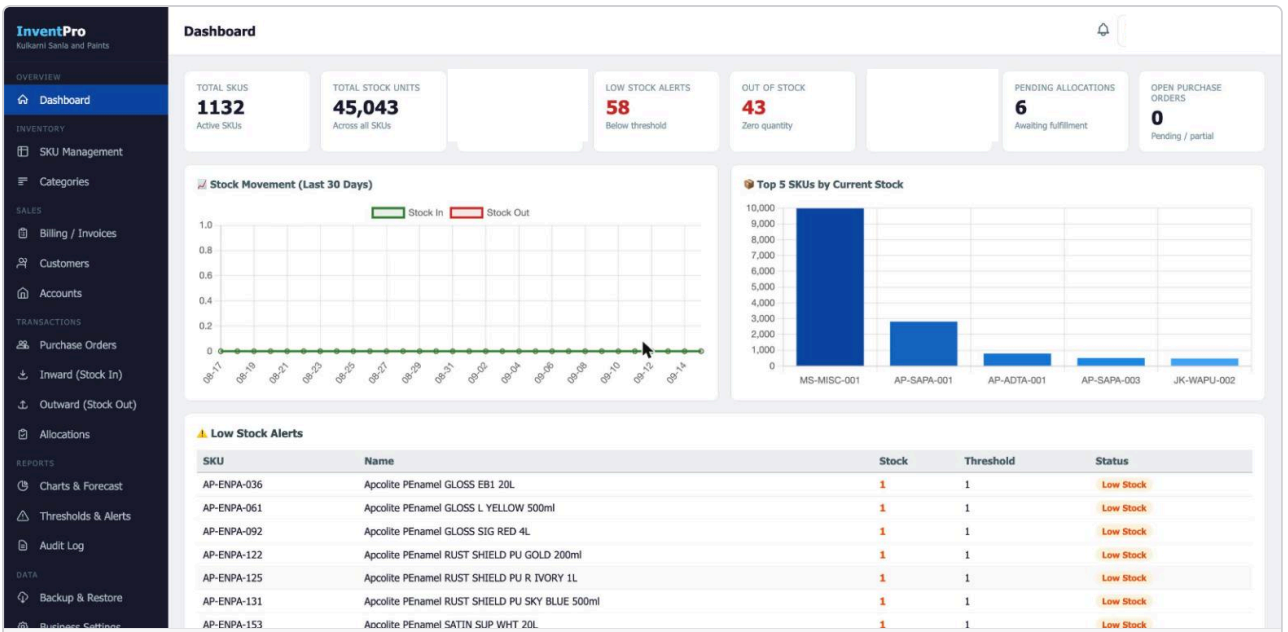
Shipping. GitHub Actions handles CI/CD to Cloudflare; secrets (`JWT_SECRET`, `RESEND_API_KEY`, `OTP_ENCRYPT_KEY`) are set via `wrangler secret put`, never committed.

CODE SNIPPET

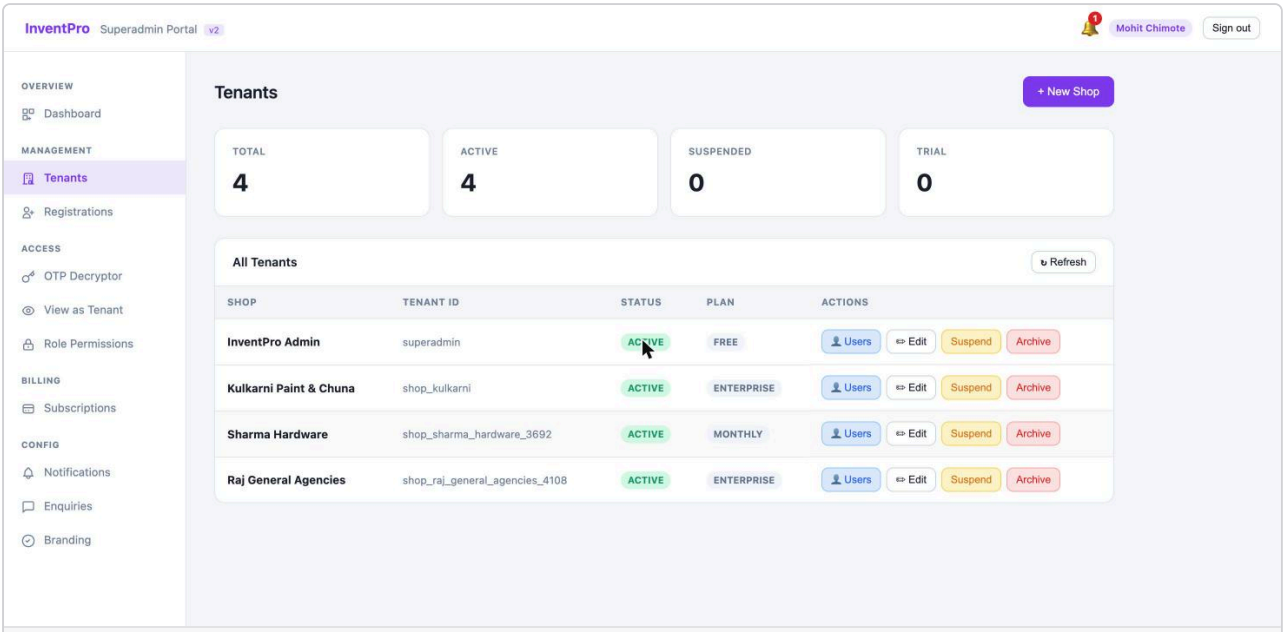
worker/src/index.js — javascript

```javascript
// ── JWT ──────────────────────────────────────────────────────────
async function signJWT(payload, secret) {
  const h = btoa(JSON.stringify({ alg: 'HS256', typ: 'JWT' }));
  const b = btoa(JSON.stringify({ ...payload, iat: Date.now() }));
  const key = await crypto.subtle.importKey('raw', new TextEncoder().encode(secret),
    { name: 'HMAC', hash: 'SHA-256' }, false, ['sign']);
  const sig = await crypto.subtle.sign('HMAC', key, new TextEncoder().encode(`${h}.${b}`));
  return `${h}.${b}.${btoa(String.fromCharCode(...new Uint8Array(sig)))}`;
}
async function verifyJWT(token, secret) {
  try {
    const [h, b, s] = token.split('.');
    const key = await crypto.subtle.importKey('raw', new TextEncoder().encode(secret),
      { name: 'HMAC', hash: 'SHA-256' }, false, ['verify']);
    const valid = await crypto.subtle.verify('HMAC', key,
      Uint8Array.from(atob(s), c => c.charCodeAt(0)),
      new TextEncoder().encode(`${h}.${b}`));
    if (!valid) return null;
    const p = JSON.parse(atob(b));
    if (Date.now() - p.iat > 86400000) return null;
    return p;
  } catch { return null; }
}
```

SCREENSHOTS



Tenant dashboard — a live paint-shop tenant (1,132 SKUs, 45k+ units tracked). Inventory value and receivables figures blurred out for privacy.



Superadmin portal — the tenant registry, cross-tenant: four live shops across free, monthly, and enterprise plans

OUTCOME

Live and in daily use, starting with my uncle's paint shop as the first tenant and since onboarded further tenants in other retail trades — evidence the multi-tenant model generalizes beyond the shop it was originally built for.

The serverless architecture means there's been no server to provision or patch since launch, and the single-table document model plus three-static-HTML-files-and-one-Worker-script shape keeps operational overhead low as tenant count grows — very little surface area to maintain per additional shop, regardless of what it sells.

Journey LOS

Config-first, multi-tenant Loan Origination System for UK building societies and specialist lenders — a from-scratch RBAC engine (65-table schema, three-layer nav → section → field permissions) built so a new lender is onboarded by data entry, not a code deployment.

• Proof of Concept los-platform.pages.dev/ github.com/mohitchimote/los-platform

---

PROBLEM

Loan origination platforms I've worked on professionally tend to bake one lender's org chart into the product — role names like "underwriter" or "senior underwriter" hardcoded, along with which fields each role can see. That works until the next lender has a different structure: a small building society might have a flat 12-person team; a larger one has managers, regional directors, and a formal approval hierarchy. Shipping code to support every new lender's org chart doesn't scale.

Journey LOS is a personal exploration of the alternative: what if the entire permission model — roles, workflow stages, which fields are visible or editable at each stage — was configuration a lender's admin enters, not code a development team ships?

INTENDED USERS

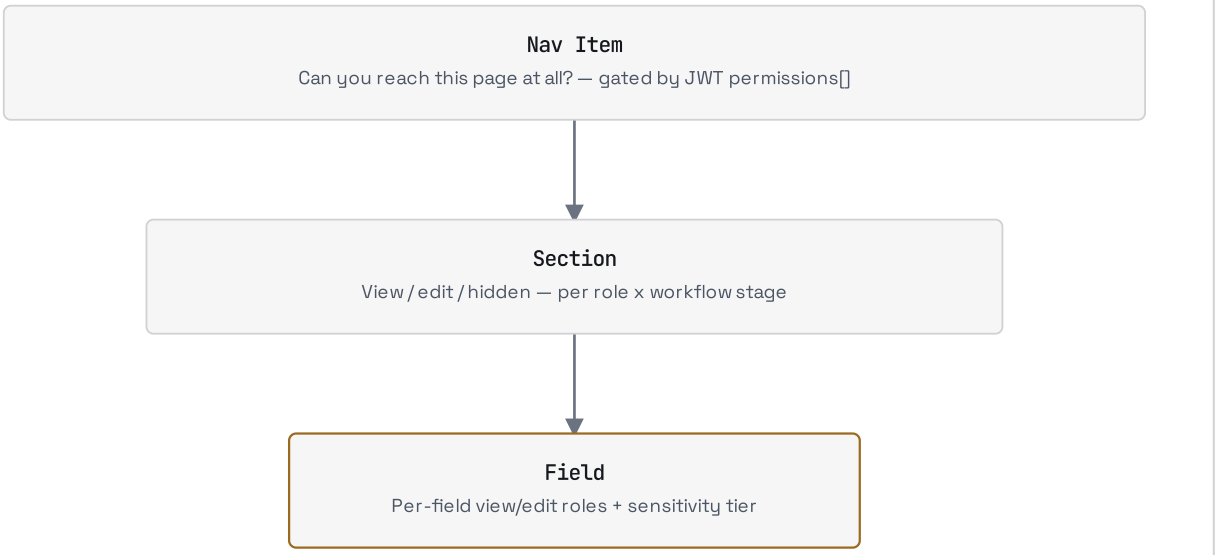
UK building societies and specialist lenders — the institution is the tenant. Within one case, internal staff (advisors, underwriters, processors, compliance), broker firms, solicitors, valuers, and the customer themselves all work the same case through a role-appropriate view, enforced by permissions rather than separate apps.

MY ROLE

Solo engineering team, alongside a full-time product role at TCS. This is the same domain I work in professionally — UK mortgage and loan origination — built here as a from-scratch personal exploration of how far a fully configurable permission model can go before it needs custom code per lender.

ARCHITECTURE & STACK

React | Vite | Hono.js | Cloudflare Workers | Cloudflare D1 | Cloudflare KV | Cloudflare R2 | TypeScript | Tailwind CSS | WebCrypto API (PBKDF2 + HS256 JWT)



One case, three permission layers — nav, section, and field each decide what a role sees before a query runs.

Journey LOS is a hub-and-spoke multi-tenant platform: one hub, multiple lender institutions, with every party on a case (internal staff, brokers, solicitors, valuers, customers) accessing the same case data through a role-appropriate view rather than separate applications.

Auth was rebuilt from scratch, on purpose. It originally used Clerk. I pulled it out — a third-party service sitting on the critical auth path, a JWT claim payload too thin for rich RBAC data, per-MAU pricing at scale, and SSO integration being harder with Clerk as an intermediary. In its place: PBKDF2-SHA256 password hashing (100,000 iterations, 16-byte salt, pure WebCrypto, no npm dependency) and hand-rolled HS256 JWTs — the same "no external auth library" instinct as InventPro, but carrying a much richer payload here. Every token encodes the caller's institution, role, role level, permission list, and visibility scope directly, so authorization needs zero database lookups per request.

Permission is layered three ways. The same case view renders differently per role and per workflow stage, checked at three levels: can you reach this page at all (nav), can you see this section and is it editable (section, per workflow stage), can you see this specific field and edit it (field, down to a per-field sensitivity tier). See the diagram below.

Config-first RBAC, not fixed roles. `role_definitions` stores each institution's org chart as data — role label, level, approval limit, visibility scope — checked against a ~60-entry `permission_catalog`. Permission checks support wildcard matching (`entity.*` grants `entity.case.create`, `entity.case.edit`, etc.) via a single `hasPermission()` function,

wired into every protected route through a `requirePermission()` Hono middleware — see the code snippet.

A genuinely large domain model. 65 tables across 38 migrations: cases carry an array of case types so a Porting + Further Advance can share one case, one ESIS, and one customer set; facilities model Part A/Part B loans independently; collateral has its own lifecycle (`proposed → owned → pledged → charged → released`) and can be flagged, not blocked, when re-pledged across active cases; customers carry versioned financial profiles with a configurable staleness window that gates workflow progression.

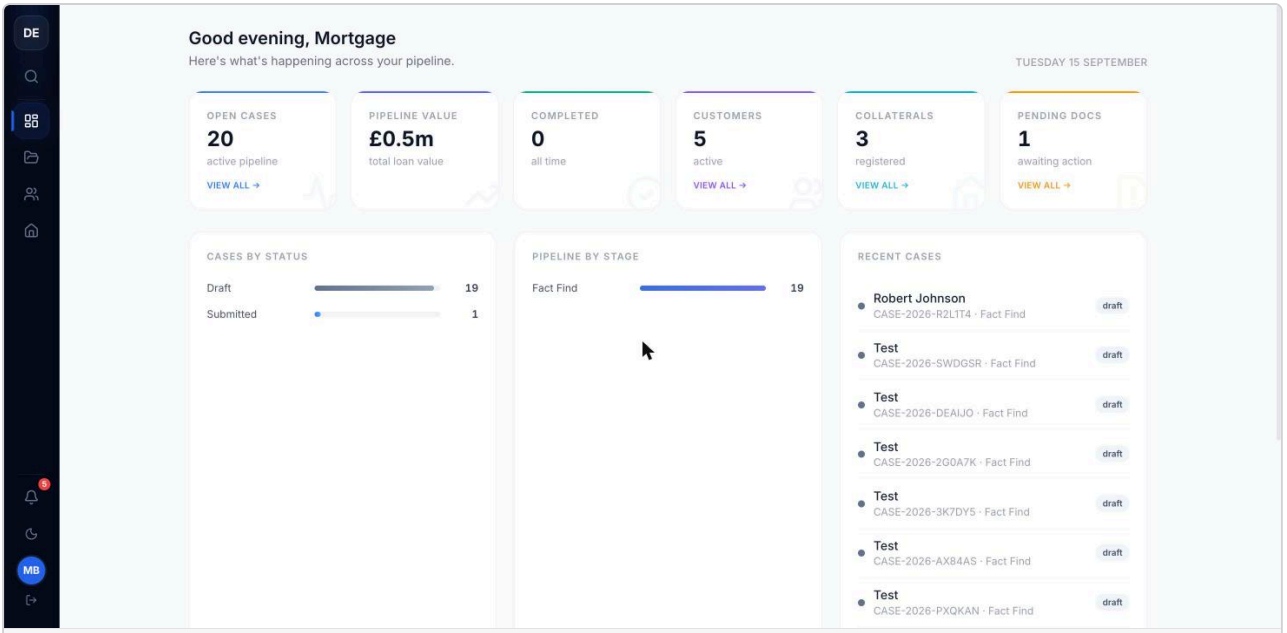
Honest state. This is proof-of-concept stage. The RBAC middleware and schema described above are real and wired into all 28 API route files — that part works. Some of the design document's more ambitious enforcement patterns, like query-level Chinese-wall separation between broker-sourced and internally-sourced cases, are specified in the role schema (`case_source_filter`) but not yet enforced in every case query. I'm building this in the open about what's actually implemented versus designed.

CODE SNIPPET

apps/api/src/middleware/auth.ts — typescript

```typescript
// Checks that the user holds at least one of the required permissions.
// Supports wildcard suffix matching: 'entity.*' grants 'entity.case.create', etc.
export function hasPermission(userPermissions: string[], required: string): boolean {
  return userPermissions.some(p => {
    if (p === '*') return true;
    if (p === required) return true;
    if (p.endsWith('.*')) {
      const prefix = p.slice(0, -2);
      return required === prefix || required.startsWith(prefix + '.');
    }
    return false;
  });
}

export const requirePermission = (permission: string) =>
  createMiddleware<{ Bindings: Env }>(async (c, next) => {
    const user = c.get('user');
    if (!hasPermission(user.permissions, permission)) {
      return c.json({ success: false, error: 'Forbidden: insufficient permissions' }, 403);
    }
    await next();
  });
```

SCREENSHOTS



Broker dashboard — pipeline by stage, case status, and recent cases for the logged-in role.

OUTCOME

Proof-of-concept stage — not deployed anywhere live, and not trying to look otherwise. What's real: a 65-table schema across 38 migrations, a three-layer permission model wired into all 28 API route files, and a working decision to rip out a third-party auth provider mid-build once it stopped fitting the RBAC model, rather than bend the model to fit the provider.

The open question this was built to answer — whether a fully config-driven permission model can absorb a new lender's org chart without a code change — is answered for the parts that are wired up. The parts still marked as designed-but-not-enforced (like the Chinese-wall query filtering) are the honest boundary of what's actually done versus what's specified.

LetWise UK

A landlord wealth-management platform — mortgage, tenancy, and cashflow tracking with an automatic remortgage-savings finder, plus a role-gated admin console. Built for a friend managing a multi-property portfolio.

• Live letwise-uk.lovable.app/ github.com/mohitchimote/letwise-uk

---

PROBLEM

A friend of mine manages a portfolio of rental properties in the UK, and was tracking mortgages, tenancies, and rental income the way most landlords do — spreadsheets, plus whatever each mortgage lender's portal shows in isolation. The two things that actually matter for a landlord's return — whether any mortgage's fixed-rate period is ending soon, and whether a better rate is now available elsewhere — weren't connected anywhere. Compliance deadlines (EPC ratings, gas safety certificates) added another set of dates to track by hand.

LetWise UK was built to put portfolio equity, cashflow, and compliance deadlines in one dashboard, and — the more novel part — to automatically flag when a property's current mortgage rate is worse than what's newly available, without the landlord having to go shopping for rates themselves.

INTENDED USERS

My friend, a portfolio landlord managing multiple UK rental properties, is the primary user — tracking property valuations, mortgages, tenancies, and cashflow in one place. A second, role-gated admin interface is for platform operations: managing the market-rate data that powers the savings comparisons, monitoring subscriptions, and handling remortgage referrals.

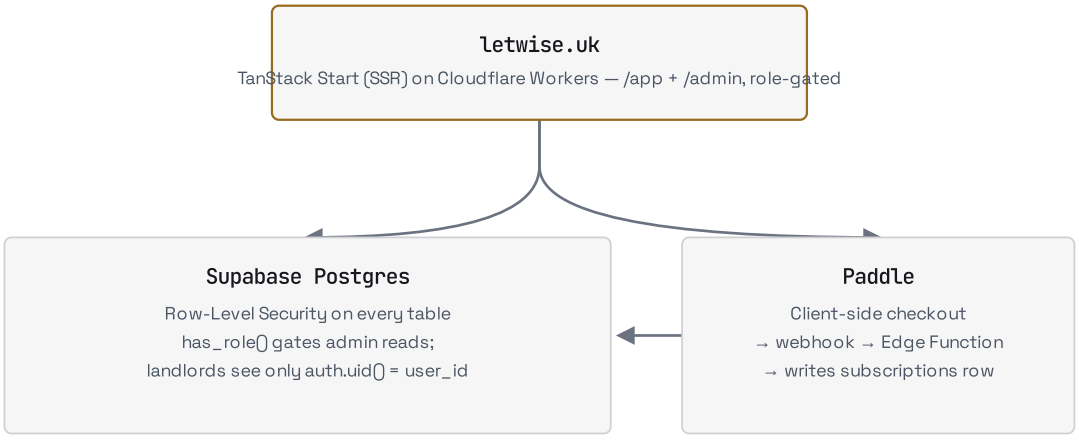
MY ROLE

Solo engineering team, alongside a full-time product role at TCS — product design, data model, and full-stack build (TanStack Start frontend, Supabase schema and RLS policies, Cloudflare Workers deployment, Paddle billing integration).

ARCHITECTURE & STACK

`TanStack Start` `React` `TypeScript` `Tailwind CSS` `shadcn/ui`

`Supabase (Postgres, Auth, Edge Functions)` `Cloudflare Workers` `Paddle`



One deployment, two role-gated interfaces — authorization enforced in Postgres RLS, not in the app.

LetWise UK is a single TanStack Start application — React 19 with file-based routing, server-rendered and deployed to Cloudflare Workers — rather than a separate frontend and API. Role-gated route groups (`/_authenticated/*` for the landlord app, `/_authenticated/_admin/*` for the admin console) mean one deployment serves two distinct interfaces without a second app to build or ship.

Access control lives in the database, not the app. Every table — properties, mortgages, tenancies, transactions, subscriptions — has Row-Level Security enabled in Postgres via Supabase. Rather than each policy re-deriving "is this user an admin," a single `has_role()` security-definer function is called from every policy that needs it (see the snippet below). Landlords see only rows where `auth.uid() = user_id`; admins pass a role check instead. The authorization logic exists in exactly one place, not duplicated across a dozen policies or re-implemented in application code.

The savings finder is a straight comparison, not a black box. Each mortgage is matched against the latest `market_rates` row for its LTV band and term type, and the difference against the mortgage's current rate becomes a "potential monthly saving" shown on the landlord's dashboard. Acting on it logs a row to `referrals`, which shows up in the admin console's referral queue — the whole feature is a join and a subtraction, not a model.

Billing is Paddle, wired through Supabase Edge Functions. Checkout runs client-side via Paddle.js; a Supabase Edge Function verifies and handles the webhook, updating the `subscriptions` table server-side so subscription status is never trusted from the client.

Admin is a first-class interface, not an afterthought. The admin console (visually distinct — emerald rather than the landlord app's indigo) covers rate management, user

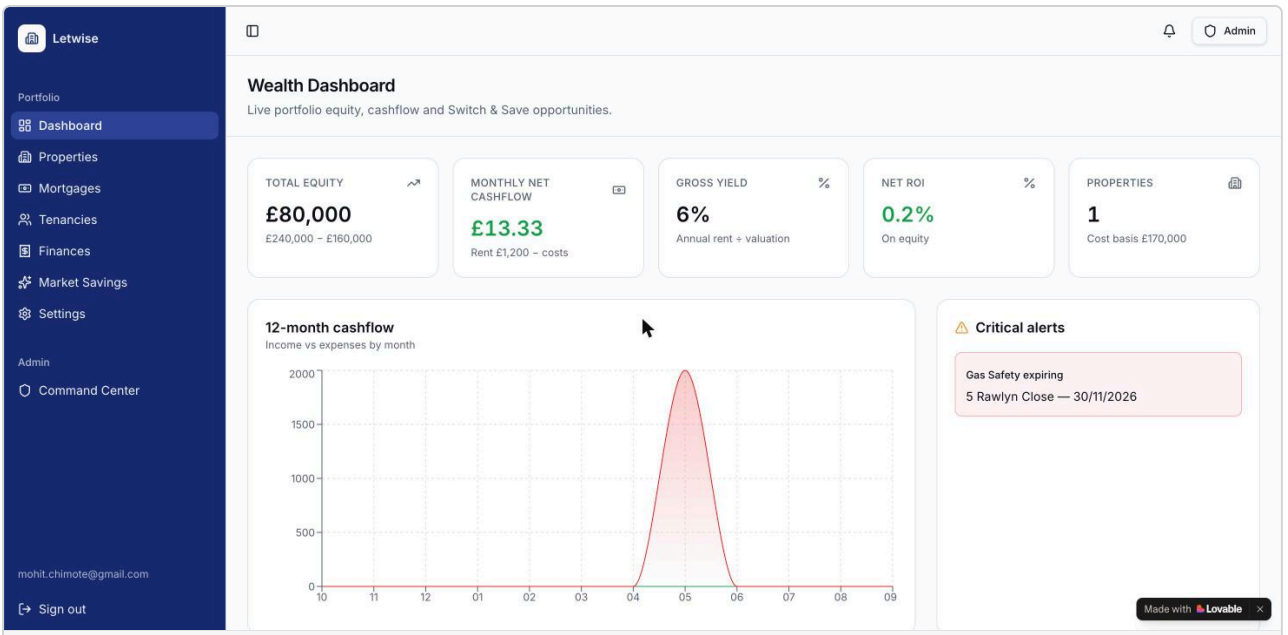
administration, a broadcast tool for announcements, the referral queue, and a read-only audit log that every admin write goes through.

CODE SNIPPET

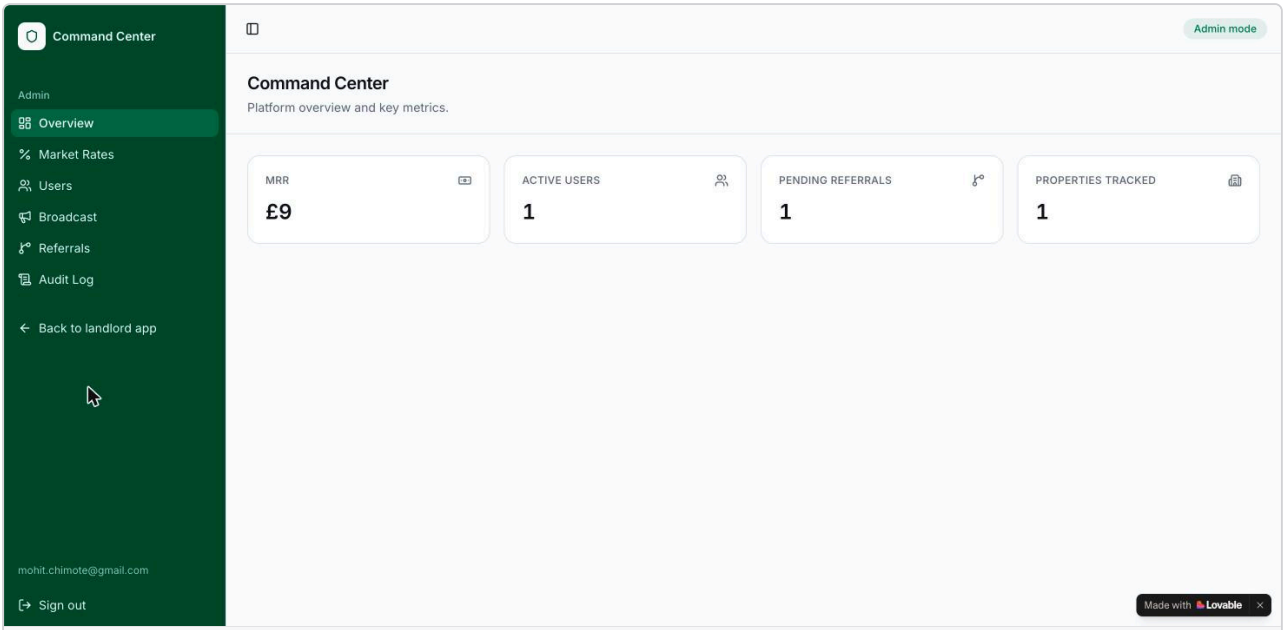
supabase/migrations/20260419221700_403e120a-61b7-4927-bed1-b532dcefa40e.sql sql

```sql
-- one security-definer function, called from every RLS policy that needs a role check
create or replace function public.has_role(_user_id uuid, _role public.app_role)
returns boolean language sql stable security definer set search_path = public as $$
  select exists (select 1 from public.user_roles where user_id = _user_id and role = _role)
$$;

-- used directly in policies, e.g. (properties table):
create policy "properties_admin_select" on public.properties
  for select using (public.has_role(auth.uid(), 'admin'));
```

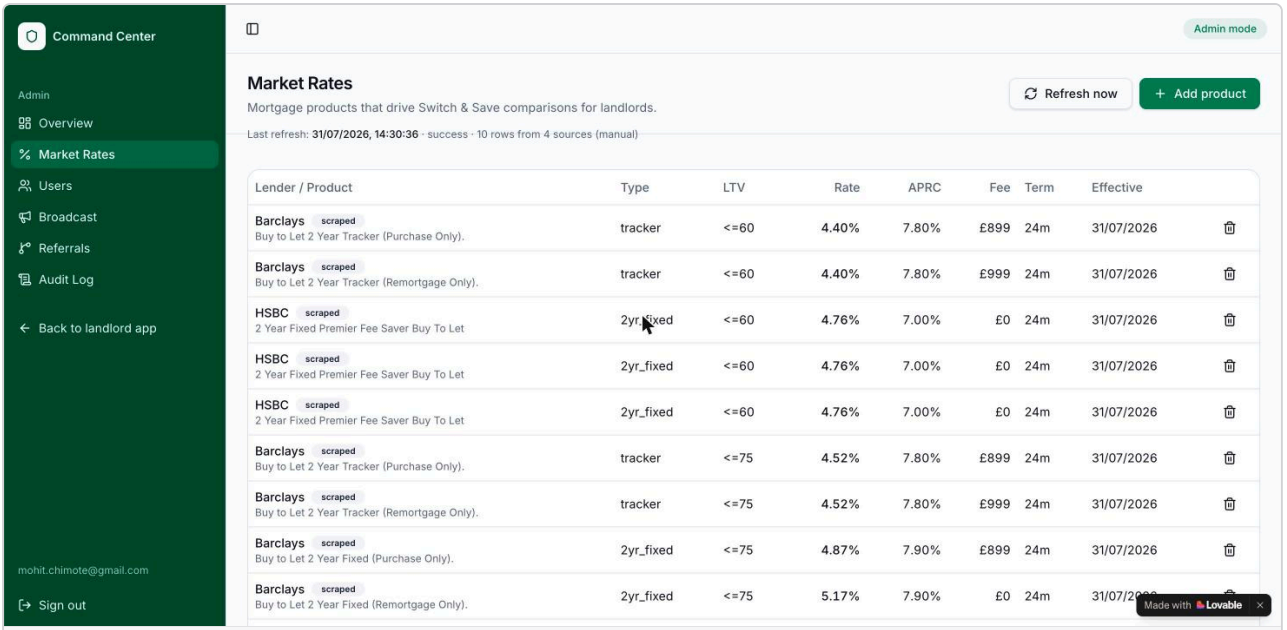
SCREENSHOTS



Landlord wealth dashboard — portfolio equity, net cashflow, and compliance alerts in one view



Command Center — platform metrics, visually distinct (emerald vs. the landlord app's indigo) from the landlord interface



Market Rates — the scraped product data every landlord's Switch & Save comparison is matched against

OUTCOME

In active use by the friend it was built for, tracking their property portfolio day to day — mortgages, tenancies, cashflow, and compliance deadlines in one dashboard instead of scattered across lender portals and spreadsheets.

The architecture decisions travel well beyond a single user, though: role-gated routing in one deployment, RLS-enforced multi-tenancy through a single `has_role()` function, and Paddle billing wired through server-verified webhooks are the same shape a genuine multi-landlord SaaS product would need — nothing here was built assuming there'd only ever be one user.

Clinic Pay Tracker

A pay-tracking and invoicing tool for a self-employed dental specialist working across two clinics with two different pay structures.

• Live paytracker.drmonica.in github.com/mohitchimote/clinic-tracker

---

PROBLEM

My wife is self-employed across two dental clinics that pay her in completely different ways — one on an hourly rate plus a two-tier treatment commission and a set of flat daily bonuses, the other on a per-treatment rate table with no hourly component at all. She was tracking both by memory and a notes app, then reconstructing the month from scratch to invoice each clinic — easy to under-bill, easy to lose a day entirely.

She needed something she could log on her phone in under a minute after each shift, that would do the maths itself for whichever clinic she'd worked that day, and hand her a ready-to-send invoice at the end of the month.

INTENDED USERS

My wife — a self-employed dental specialist working across two clinics (a London practice and one in Southend-on-Sea) with different pay structures at each. She's the only user, logging her own pay on her phone after each shift.

MY ROLE

Solo engineering team, alongside a full-time role at TCS — built and maintain the whole thing: the frontend, the Worker API, and the D1 schema.

ARCHITECTURE & STACK

`HTML` `JavaScript` `Cloudflare Workers` `Cloudflare D1` `GitHub Pages`

It's deliberately the simplest stack that could do the job. `index.html` is the entire frontend — no framework, no build step — hosted on GitHub Pages behind a custom domain. It talks to a small Cloudflare Worker API backed by Cloudflare D1.

Two clinics, two pay formulas, one app. Each clinic has its own calculation function. The Harley clinic is `hourly + tiered treatment commission + flat daily bonuses` (see the snippet below); the Southend clinic is an entirely different item-and-rate-table model. Switching clinics in the UI swaps which formula and which invoice renderer is used — there's no attempt to unify them into one generic model, because they aren't actually the same shape of problem.

One table in a shared database. The D1 database (`los-db`) is shared with another one of my projects — the Worker only ever reads or writes the single `clinic_days` table that belongs to this app, identified by a `YYYY-MM-DD` key per day worked.

Auth is a shared password, on purpose. There's exactly one real user, so a single app-wide password gating both the frontend and the Worker API is enough — no accounts, no sessions, no OAuth. It's the right amount of engineering for who actually uses this.

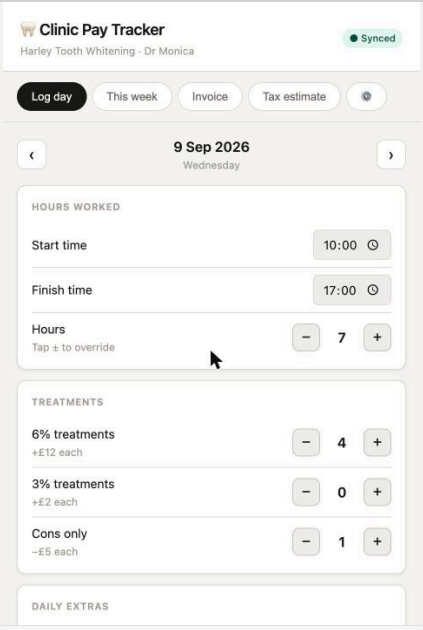
Invoicing is a copy-paste, not a PDF. At month end, the invoice tab totals up hours and treatments for the period and renders a plain-text summary she copies straight into a message to the clinic — matching how she actually sends invoices today rather than generating a document nobody asked for.

CODE SNIPPET

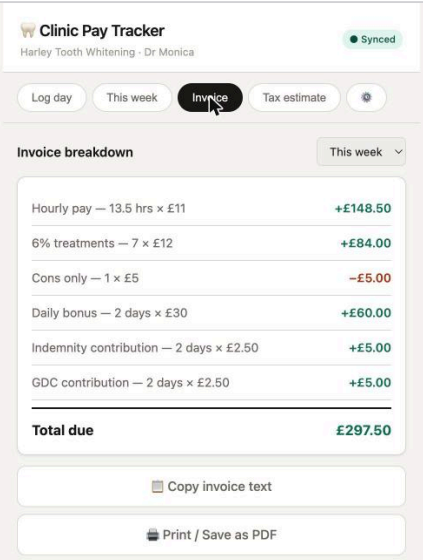
index.html javascript

```javascript
function calcDay(s) {
  const hourly = s.hours * 11;
  const treats = (s.t6*12)+(s.t3*2)-(s.cons*5);
  const extras = (s.bonus?30:0)+(s.indem?2.5:0)+(s.gdc?2.5:0);
  return { hourly, treats, extras, total: hourly+treats+extras };
}
```

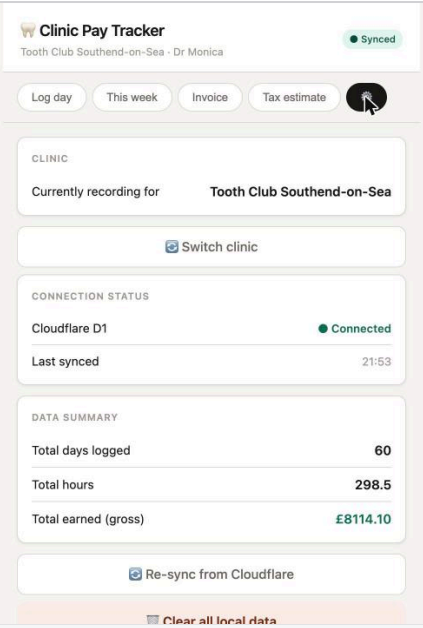
SCREENSHOTS



Daily log — Harley clinic, a real logged shift (7 hrs, 4× 6% treatments, 1 cons-only) — the exact inputs to the calcDay() snippet above



Invoice breakdown — hourly pay, treatment commission, and daily extras itemised to a real total due, ready to copy or export as PDF



Settings — the clinic switcher (she works across two, each with its own pay formula) and real cumulative usage since launch

OUTCOME

In daily use since it was built: 60 days logged, 298.5 hours, £8,114.10 tracked in gross fees — real numbers pulled straight from the app's own data summary, not an estimate. Logging a shift now takes under a minute on her phone, and month-end invoicing is copy-paste instead of reconstructing the month from memory.

It's the smallest of these projects by far, and it hasn't needed to change since — which, for a tool that has to survive being used by someone other than the person who built it, is the actual measure of whether it works.

Life Partner Seek

Matchmaking platform built for a proposed marriage bureau — complete and deployable, with a role-based visibility model built to the same standard as an enterprise RBAC system.

• Built · Awaiting operator life-partner-seek.lovable.app/ github.com/mohitchimote/life-partner-seek

---

PROBLEM

A family member was planning to launch a matrimonial matchmaking bureau — the traditional model where bride and groom candidates, often with a parent managing the profile on their behalf, are matched by staff who mediate between families. That staffing model creates a real access-control problem: members need to browse and message each other, parents need visibility into a profile they manage without holding that person's login, and bureau staff (relationship managers, admins) need broad cross-profile access to do their job — but a member shouldn't see everyone's full profile, and even staff identity shouldn't be exposed to a member who has no active relationship with them.

Life Partner Seek was built to solve that access problem in the data layer, not paper over it with UI-level hiding that a direct API call could bypass.

INTENDED USERS

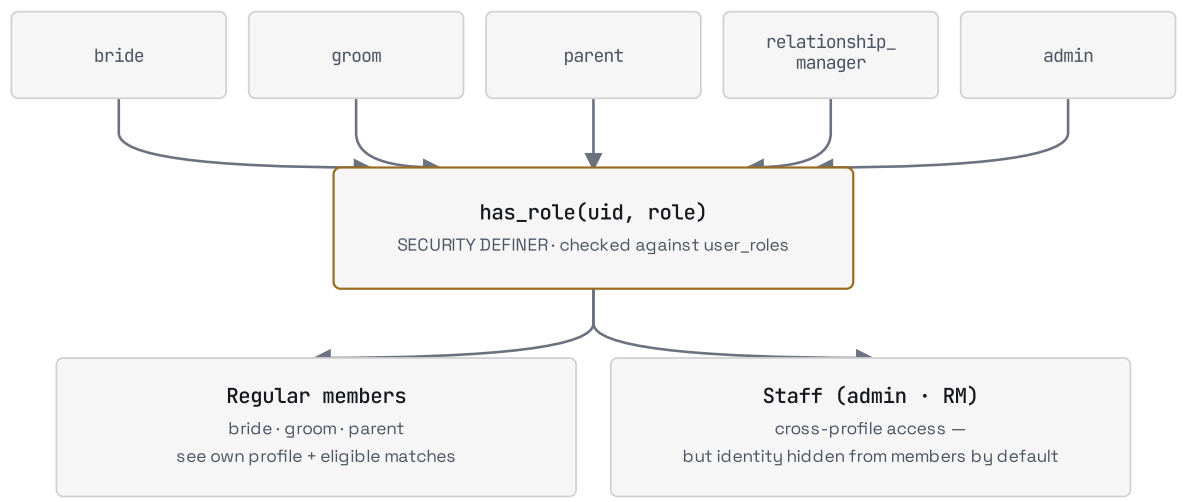
Bride, groom, and parent members browsing and managing matrimonial profiles, plus relationship-manager and admin roles staffing the bureau. Built for a family member who's planning to launch this as a matchmaking bureau — it hasn't launched yet, so there are no real members using it, and this write-up doesn't claim otherwise.

MY ROLE

Solo engineering team, alongside a full-time role at TCS — product design, data model, and full-stack build (React/TypeScript frontend, Supabase schema and RLS policies).

ARCHITECTURE & STACK

React | TypeScript | Supabase | PLpgSQL | Row-Level Security | shadcn/ui | Tailwind CSS



Role membership alone doesn't grant visibility — a RESTRICTIVE policy checks the actual relationship (self, admin, RM, a managed-profile link, or an active message thread) before a staff member's identity is shown to a regular member.

Five roles, one enum, one lookup table. `app_role` — `bride`, `groom`, `parent`, `relationship_manager`, `admin` — is a Postgres enum, and role membership lives in a dedicated `user_roles` table rather than as a column on `profiles`. That separation is deliberate: it's the standard defense against a user editing their own profile row to grant themselves a role they shouldn't have.

One function gates every decision. `has_role(user_id, role)` is a `SECURITY DEFINER` SQL function that checks `user_roles`, and it's the only thing every RLS policy calls to make an access decision — no role logic duplicated across policies or reimplemented in the frontend.

Staff access is real, but staff identity isn't free. Admins and relationship managers can see across member profiles to do their job. But a `RESTRICTIVE` policy on `profiles` (the snippet below) means a member can only see *who* a given admin or RM actually is if they have a genuine relationship with that staff member — themselves, an active `managed_profiles` link, or an existing message thread. Role membership alone doesn't unlock visibility in either direction by default.

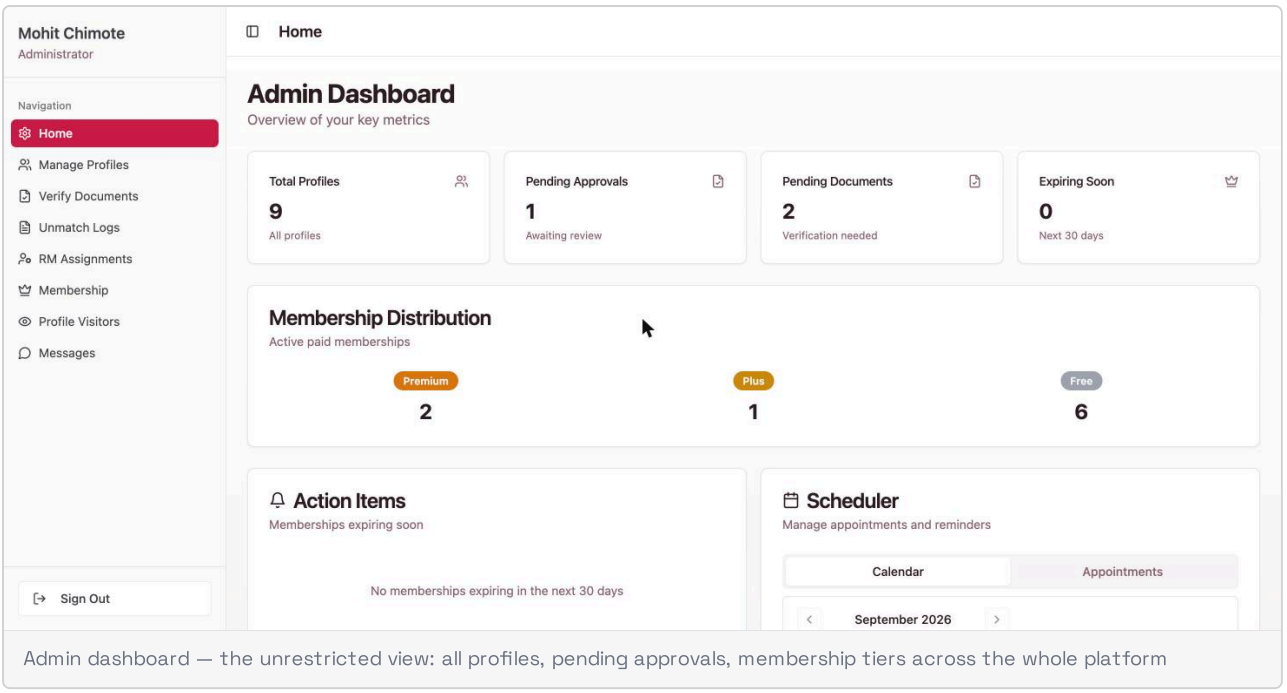
Parents get delegated access, not shared logins. A `managed_profiles` table links a `manager_id` to a `managed_profile_id` with an explicit `relationship` (`parent` or `relationship_manager`) — so a parent managing their child's profile is a real, auditable row in the data model, not a shared password.

CODE SNIPPET

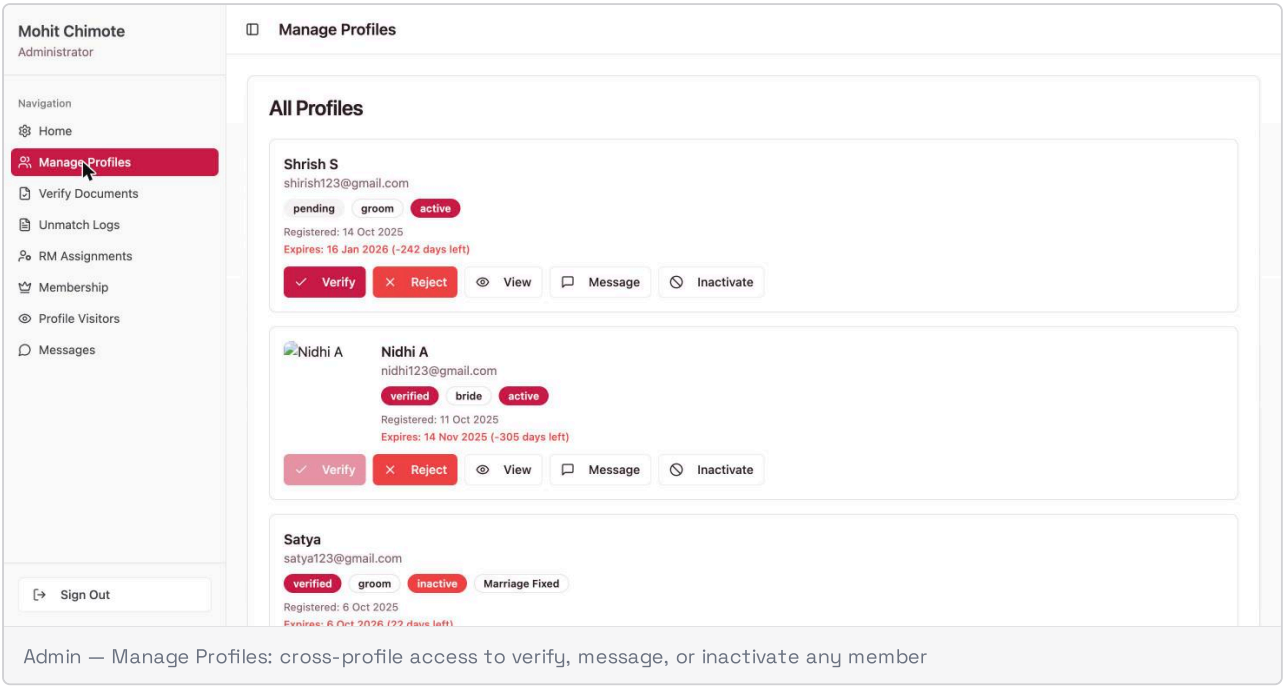
supabase/migrations/20251020212633_23ff1f4c-f63b-4f2f-9c6d-b52789f4746e.sql sql

```sql
-- Allow staff names (Admin/RM) to be visible to users who have a direct message relationship with them
-- This updates the restrictive profiles visibility policy to include has_message_relationship()

DROP POLICY IF EXISTS "Restrict staff visibility" ON public.profiles;

CREATE POLICY "Restrict staff visibility"
ON public.profiles
AS RESTRICTIVE
FOR SELECT
USING (
  -- Non-staff profiles are generally visible subject to other policies
  (
    NOT has_role(id, 'admin'::app_role)
    AND NOT has_role(id, 'relationship_manager'::app_role)
  )
  OR
  -- Staff profiles (admin/RM) are visible if one of these conditions holds
  (
    (has_role(id, 'admin'::app_role) OR has_role(id, 'relationship_manager'::app_role))
    AND (
      auth.uid() = id                                    -- self
      OR has_role(auth.uid(), 'admin'::app_role)         -- admin viewing
      OR has_role(auth.uid(), 'relationship_manager'::app_role) -- RM viewing
      OR EXISTS (
        SELECT 1
        FROM public.managed_profiles mp
        WHERE mp.manager_id = profiles.id
          AND mp.managed_profile_id = auth.uid()
      )
      OR has_message_relationship(auth.uid(), id)        -- has DM relationship with this staff
    )
  )
);
```

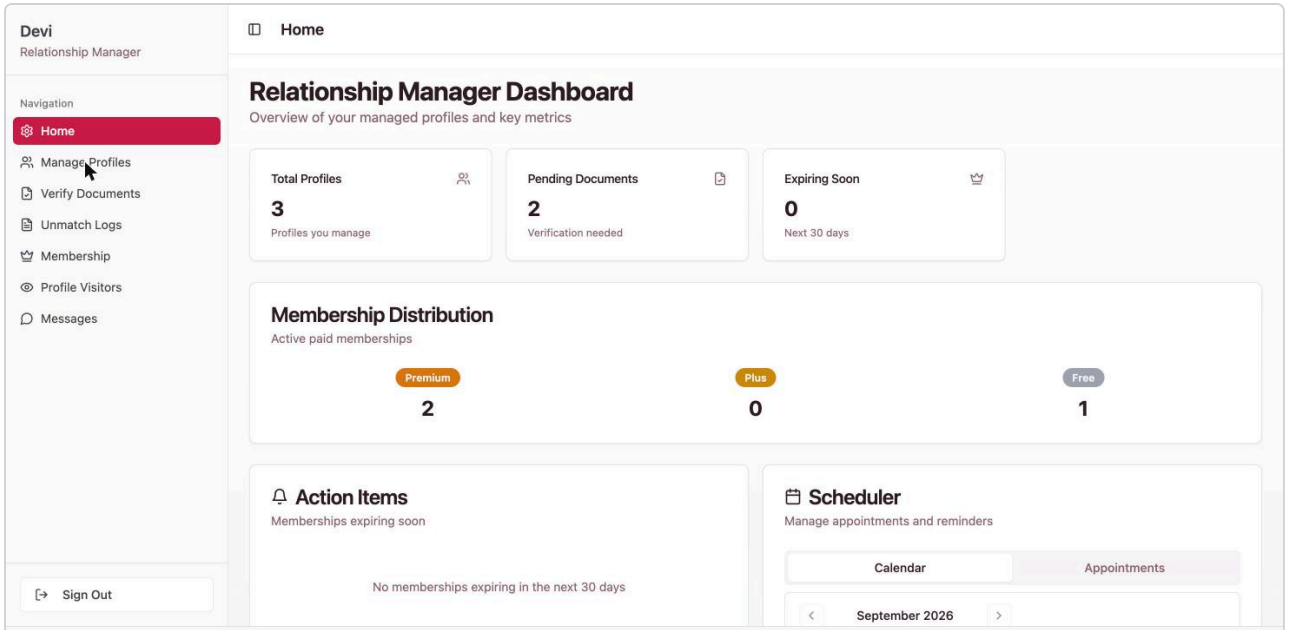
SCREENSHOTS



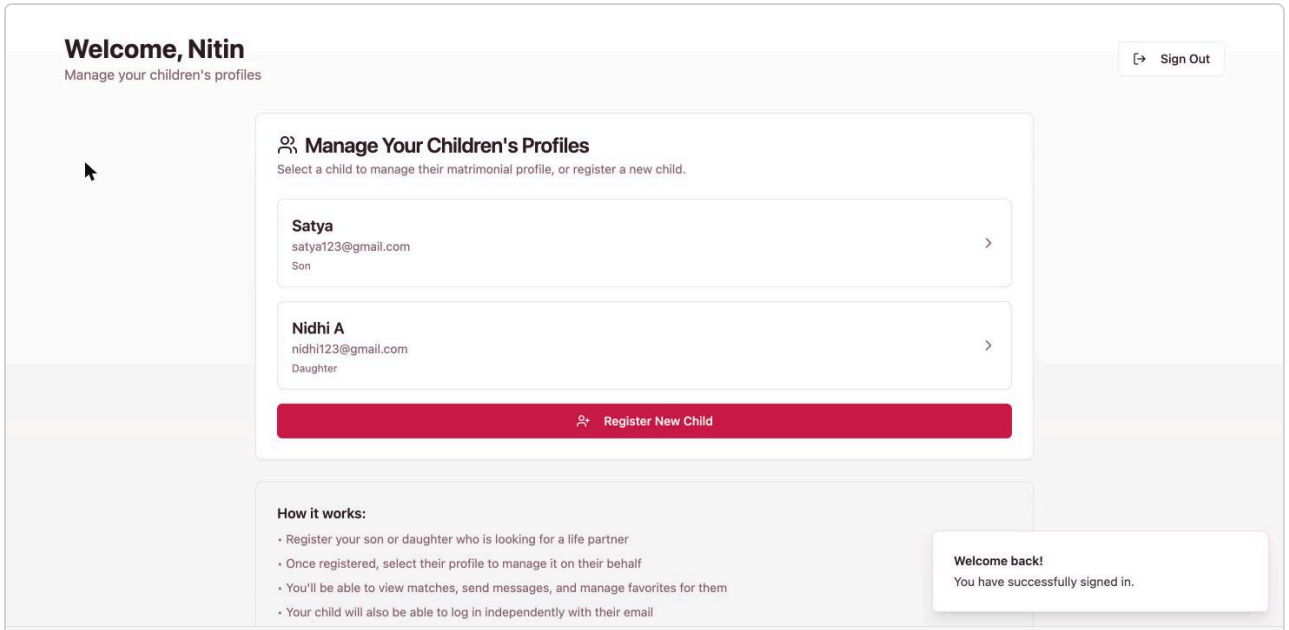
Admin dashboard — the unrestricted view: all profiles, pending approvals, membership tiers across the whole platform



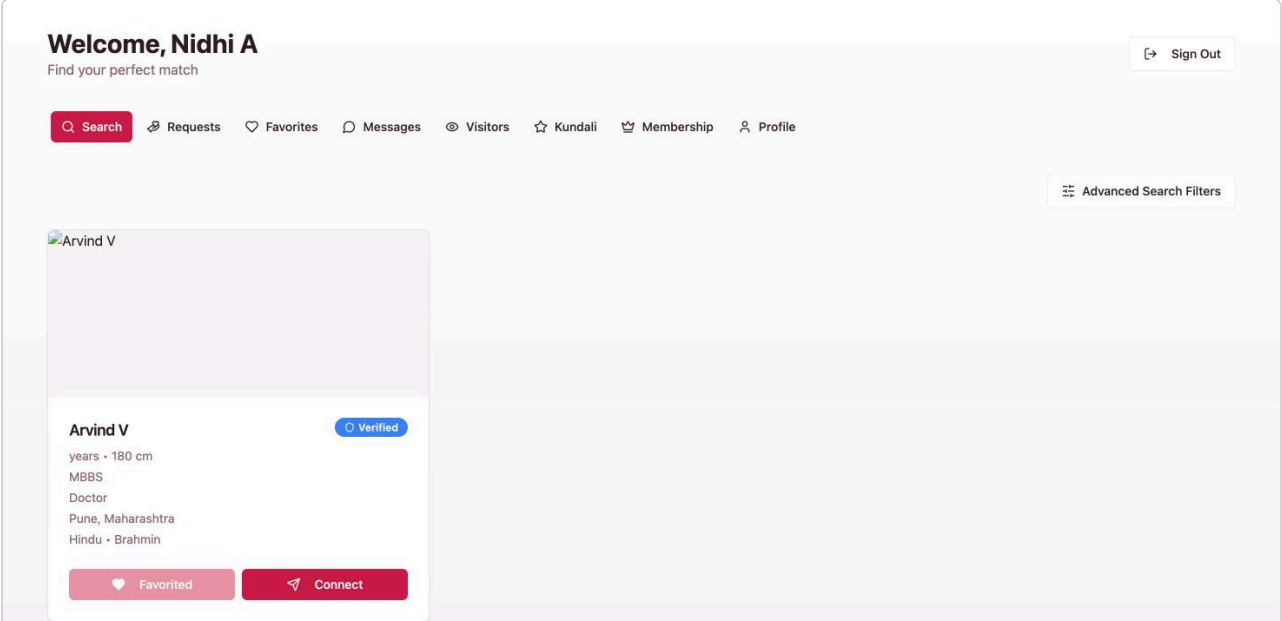
Admin — Manage Profiles: cross-profile access to verify, message, or inactivate any member



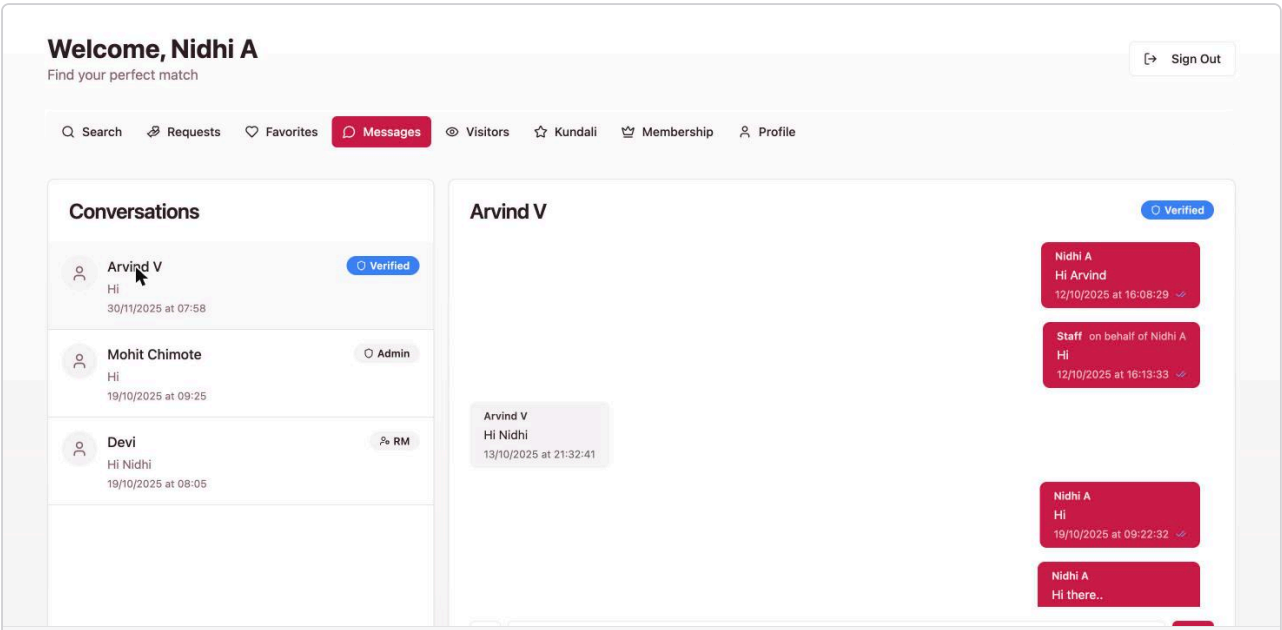
Relationship Manager dashboard — same layout as admin, but scoped to only the profiles this RM is assigned (3, not 9)



Parent view — managing two children's profiles via the managed_profiles delegation, not a shared login



Member — Search: browsing eligible matches, the ordinary member-facing view most of the RBAC model exists to protect



Member inbox — the RESTRICTIVE policy in action: staff (Admin, RM) show their real names here specifically because an active message thread exists — the exact condition in the code snippet below

OUTCOME

Complete and deployable, but not launched — it's waiting on its operator (the family member it was built for) to decide on go-to-market, so there are no real members or usage numbers to report yet.

It was built as though it were going into production from day one regardless: the role model wasn't bolted on after an MVP, it was there from the first schema migration, and the access-control decisions are enforced by Postgres RLS rather than trusted to the frontend.

AI Mortgage Tools

Two personal proof-of-concepts exploring AI-assisted mortgage processing with a locally-hosted Gemma model — document summarisation and affordability assessment, both built to run entirely offline.

• Proof of Concept AI Summariser: github.com/mohitchimote/ai-summariser

AI Affordability: github.com/mohitchimote/ai-affordability

PROBLEM

Mortgage underwriters spend a lot of time on two repetitive tasks: reading long documents — application packs, valuation reports, offer condition letters, payslips — to extract what actually matters, and running affordability calculations against a moving target of UK income tax bands, National Insurance thresholds, and cost-of-living data that shifts every tax year. AI summarisation and AI-assisted affordability checks are the obvious answer to both.

The complication is that UK regulated financial services (FCA, GDPR, Consumer Duty) makes it hard to casually send customer financial data to a third-party API. Both tools exist to test a narrower question: can a fully local, offline model do this work well enough to be useful, without that data ever leaving the network?

INTENDED USERS

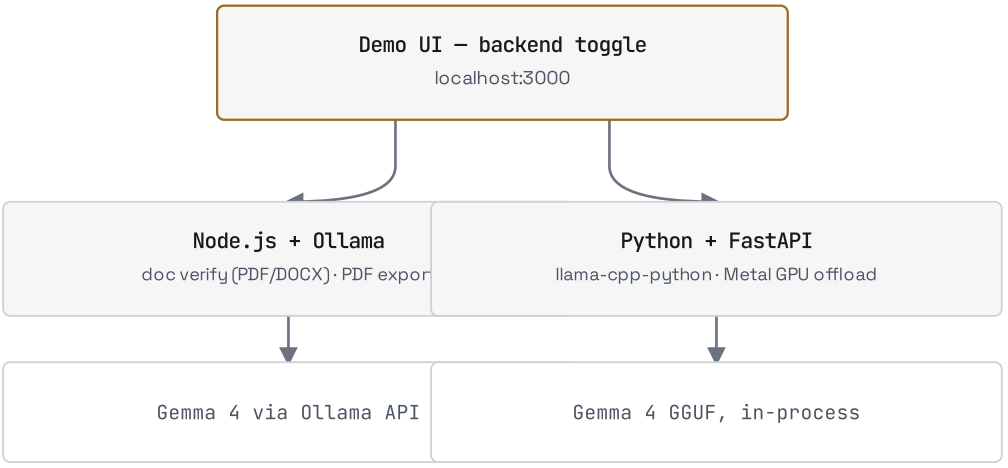
Mortgage underwriters and processors handling long document review and affordability checks — the personas these were built to test, not a live product with real users yet. Proof-of-concept stage; production path would depend on a real employer's model-governance sign-off, which is outside the scope of a personal project.

MY ROLE

Solo engineering team, alongside a full-time product role at TCS in UK mortgage technology — the day job gave me the problem context, but both of these are independent personal builds, not TCS deliverables.

ARCHITECTURE & STACK

Python | FastAPI | llama-cpp-python | Node.js | Express | Ollama | Gemma 4 | Docker



One UI, two interchangeable inference paths — both ultimately running the same Gemma 4 model.

These are two separate services sharing a theme, not one product — kept as one case study because the interesting parts (offline inference, structured domain data) are the same story told twice.

AI Summariser — one UI, two interchangeable backends. A shared demo UI toggles between a Node.js/Ollama backend (the easier path to run locally — adds document verification via PDF/DOCX upload and PDF export of summaries) and a Python/FastAPI backend using `llama-cpp-python` with Metal GPU offload on Apple Silicon for faster in-process inference, plus a mock mode for testing the API without loading the model at all. Both ultimately run Gemma 4 — the Node path calls it through the Ollama API, the Python path loads the GGUF weights (Q4_K_M, ~5GB) directly in-process. Prompts are domain-specific per document type: mortgage application, valuation report, offer conditions, payslip verification.

AI Affordability — rules as data, not as prompt strings. The Node.js/Express backend's `promptBuilder.js` detects application type (`residential` / `buy_to_let` / `holiday_let`) from the loan and property data, then filters a set of affordability rules down to the ones that actually apply before building the prompt. UK income tax bands (England & Wales and Scotland separately), employee and self-employed National Insurance thresholds, personal allowance tapering, and ONS household cost-of-living data all live as structured reference data pulled in via `referenceData.js`, not hardcoded into the prompt text. When a tax year changes, updating the tool is a data change, not a code change.

CODE SNIPPET

node-service/promptBuilder.js — javascript

```javascript
import { getAll } from './referenceData.js';

function fmt(n) { return Number(n).toLocaleString('en-GB'); }

function detectAppType(data) {
  const purpose = data?.loan?.purpose;
  const propType = data?.property?.type;
  if (purpose === 'holiday_let' || propType === 'holiday_let') return 'hl';
  if (purpose === 'buy_to_let'  || propType === 'buy_to_let')  return 'btl';
  return 'residential';
}

function filterRules(rules, appType) {
  return rules.filter(r => {
    const t = r.applicable_to || ['all'];
    return t.includes('all') || t.includes(appType);
  });
}

function formatRules(rules) {
  return rules.map(r => `[${r.category}] ${r.title}\n${r.text}`).join('\n\n');
}
```

OUTCOME

Both stayed at proof-of-concept — that decision belongs to whichever employer's model-governance process would need to sign off on production use, not something a personal project gets to decide on its own. What they did prove out: a Gemma 4 GGUF model with Metal GPU offload runs fast enough on a laptop for interactive document summarisation, and encoding UK tax and lending rules as structured reference data rather than prompt text makes the affordability tool's yearly maintenance trivial — swap a JSON file, not the code.