PORTFOLIO PIECE · 2026

I built a multi-tenant SaaS without writing code by hand.

SchemaForge — an AI search visibility platform.

Designed, architected and shipped by me using AI-native tooling. The story below is about the decisions I made, the trade-offs I navigated, and what this kind of build proves about how I work.

What I shipped

30+	8	4	1
DATABASE TABLES	BACKEND SERVICES	USER ROLES	PERSON

Why I built it

AI search engines are quietly reshaping how customers find businesses, and most SMBs have no visibility into how they're being represented in those answers. I saw a gap: a productised service that combines automated audits, structured-data fixes, and continuous monitoring — sold as a subscription with a white-label tier for agencies. SchemaForge is that product.

Why this case study exists

I wanted to test a hypothesis: with the right product instincts and AI-native tooling, one person can deliver something that would normally need a small engineering team. Not a prototype — a real platform with auth, multi-tenancy, billing, role-based access, automated workflows, and a white-label hierarchy. I built it. This document is the proof.

The decisions that mattered

AI tooling will happily generate whatever you ask for. The job — my job — was to know what to ask for, and to recognise when the answer was wrong. Here are the calls I made that shaped the product.

Multi-tenant from day one, not bolted on later

Every table that holds tenant data has an organisation ID and Row-Level Security from the first migration. This isn't glamorous, but it's the difference between a product that scales and one that has to be rewritten when the second customer signs.

A separate roles table — not roles on the user record

Storing roles on the profile is the most common security mistake I see in early-stage SaaS. I used a dedicated user_roles table with a SECURITY DEFINER function for checks. It closes off privilege-escalation attacks that most builders never think about.

Parent–child organisations for white-label

Rather than cloning the platform per agency, I modelled agencies as parent organisations with their own client roster and branded portal. One codebase, infinite resellers. This was a commercial decision as much as a technical one — it makes the agency programme genuinely scalable.

Two-pass webhook lifecycle for the audit pipeline

Audits run on an external VPS and stream results in via a signed webhook. I designed a two-pass lifecycle (create-then-enrich) so partial failures don't leave the database in a broken state. Boring infrastructure work, but it's why the product is reliable.

Score weights as data, not code

The visibility scoring algorithm (Technical 25% / Content 35% / Authority 25% / Local 15%) lives in a versioned config table. I can re-tune the model for different verticals or A/B-test weightings without redeploying. Small decision, big optionality.

GDPR retention modelled into the schema

Two-year retention with cascading purges, baked into the data model — not a TODO for later. Selling to UK SMBs means UK compliance has to be a feature, not a footnote.

How I work

> *I treat AI tooling the way a senior developer treats a strong junior: clear instructions, tight feedback loops, and a refusal to merge anything I haven't read. The output is mine. So is the responsibility.*

My loop

Specify intent in plain English

User stories, edge cases, success criteria. No tickets, no Jira — just clear prose that anyone on the business side could read and challenge.

Generate, then review like an engineer

I read every change. Security boundaries, RLS policies, role checks, error handling. If it doesn't pass my own code review, it doesn't ship.

Steer with architecture in mind

I'm constantly thinking three steps ahead. Will this scale to a hundred tenants? Does this play nicely with the white-label hierarchy? Am I painting myself into a corner?

Persist context

Every architectural decision goes into project memory so it survives across sessions. This is the discipline that stops AI-assisted projects from drifting into chaos at month three.

Ship, observe, iterate

Push to preview, click through it, find the rough edges, refine. The fastest feedback loop I've ever worked with.

What this says about me

I'm a product person who thinks in systems. I'm comfortable in databases, RLS policies, edge functions, payment flows, multi-tenancy and agency economics — not because I memorised any of it, but because I've shipped enough product to know which decisions are load-bearing and which are cosmetic.

I move fast because I cut the right corners. I don't cut the ones that matter — security, data integrity, tenant isolation, compliance. Those I get right the first time, because rework is the most expensive thing in software.

The stack I chose

These weren't defaults. Each pick reflects a trade-off I thought through.

Choice	Why I picked it
React + Vite + TypeScript	Type-safe end-to-end. Catches a class of bugs at compile time that would otherwise eat support h
Tailwind + shadcn/ui	Design-system-first. Semantic tokens mean rebrands and white-labelling are configuration, not ref
Supabase (Postgres + Auth + Storage)	RLS-native, managed, and the right shape for multi-tenant work. Plays well with my security mode
Deno Edge Functions	Lightweight server-side logic for webhooks, AI calls, and automation. Scales without me thinking a
External VPS for heavy audits	Audits are CPU-heavy and long-running — wrong fit for serverless. Splitting them onto a VPS kee
Lovable as build environment	AI-native code generation with project memory and live preview. The right tool for the job — used
Stripe for billing	Industry standard. Subscription model, affiliate payouts, and customer portal all out of the box.

What I'm looking for

Roles or projects where the bottleneck isn't engineering capacity — it's knowing what to build, in what order, with which trade-offs. Founder-shaped work. Internal tools that never make the roadmap. Products that need to exist next month, not next year. If that's you, let's talk.

Lee Hartley

Product, build, and the bit in between.

leehartley.co.uk

Appendix: Product Walkthrough

Representative screens from the SchemaForge admin and client portal. Mockups shown to avoid exposing live customer data.

Admin Dashboard

Operations overview

Top-level admin view: live KPIs across MRR, active orgs, scans in flight, and a real-time activity feed. The sidebar reflects the full admin surface (Leads, Clients, Scans Queue, Reports Builder, Tickets, Billing). Built as a single React route reading aggregated views from Postgres so the page stays responsive even as raw event volume grows.

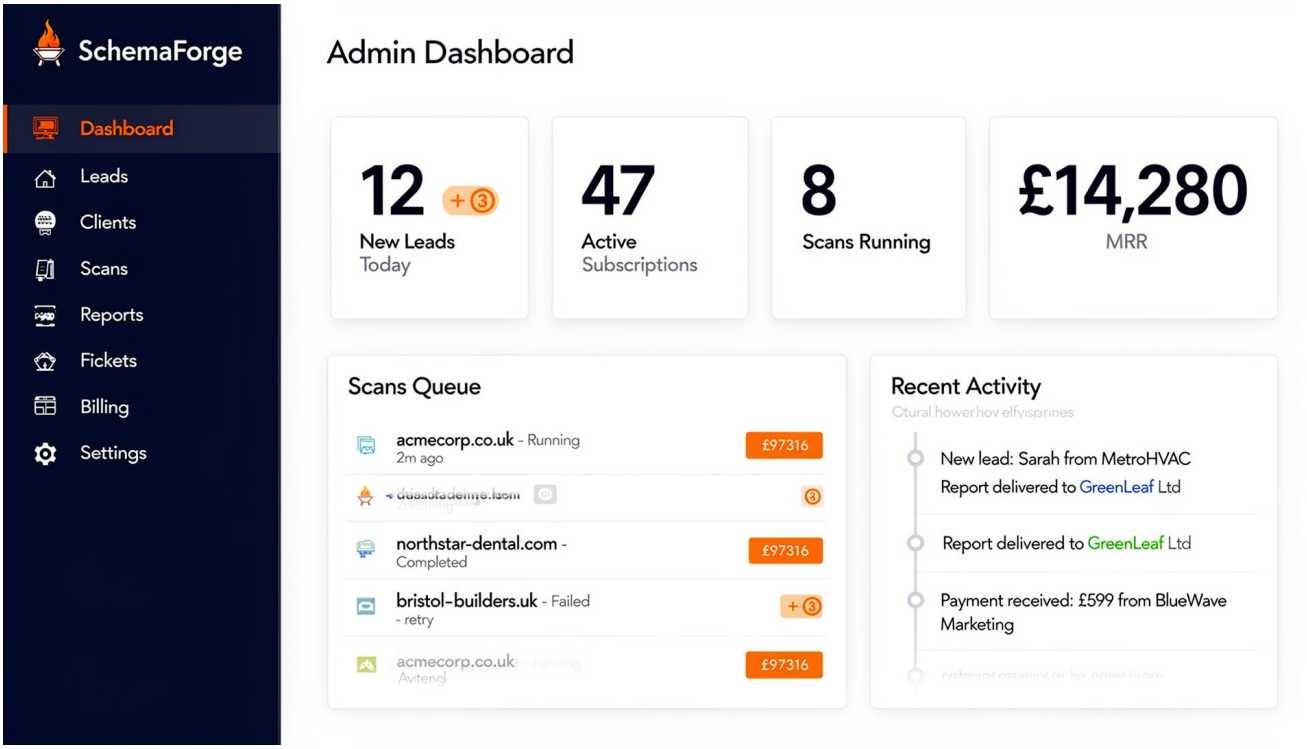

Fig. 1 — Admin Dashboard

Leads CRM

Sales pipeline

Six-stage Kanban (New → Contacted → Qualified → Trial → Won → Lost) backed by the leads table and updated via drag-and-drop. Lead capture flows in from the public marketing form via the submit-lead Edge Function (validation, rate-limiting, audit logging) — so the CRM is the single source of truth from first touch through conversion.

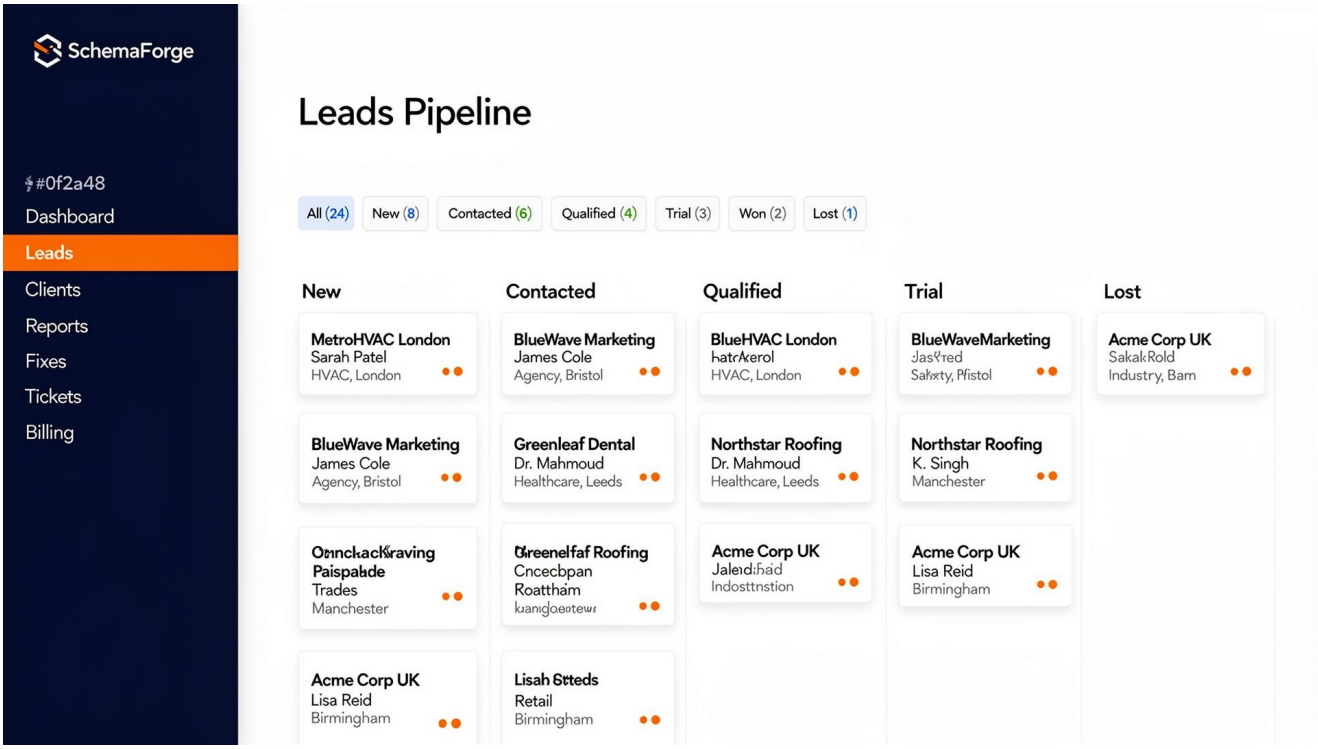

Fig. 2 — Leads CRM

Client Dashboard

Tenant-scoped portal

What a paying customer sees after login: visibility score, schema health, crawlability, and trend lines pulled from their own audit history. Every query is gated by org_id under RLS, so multi-tenant isolation is enforced at the database level rather than relying on application-side filtering.

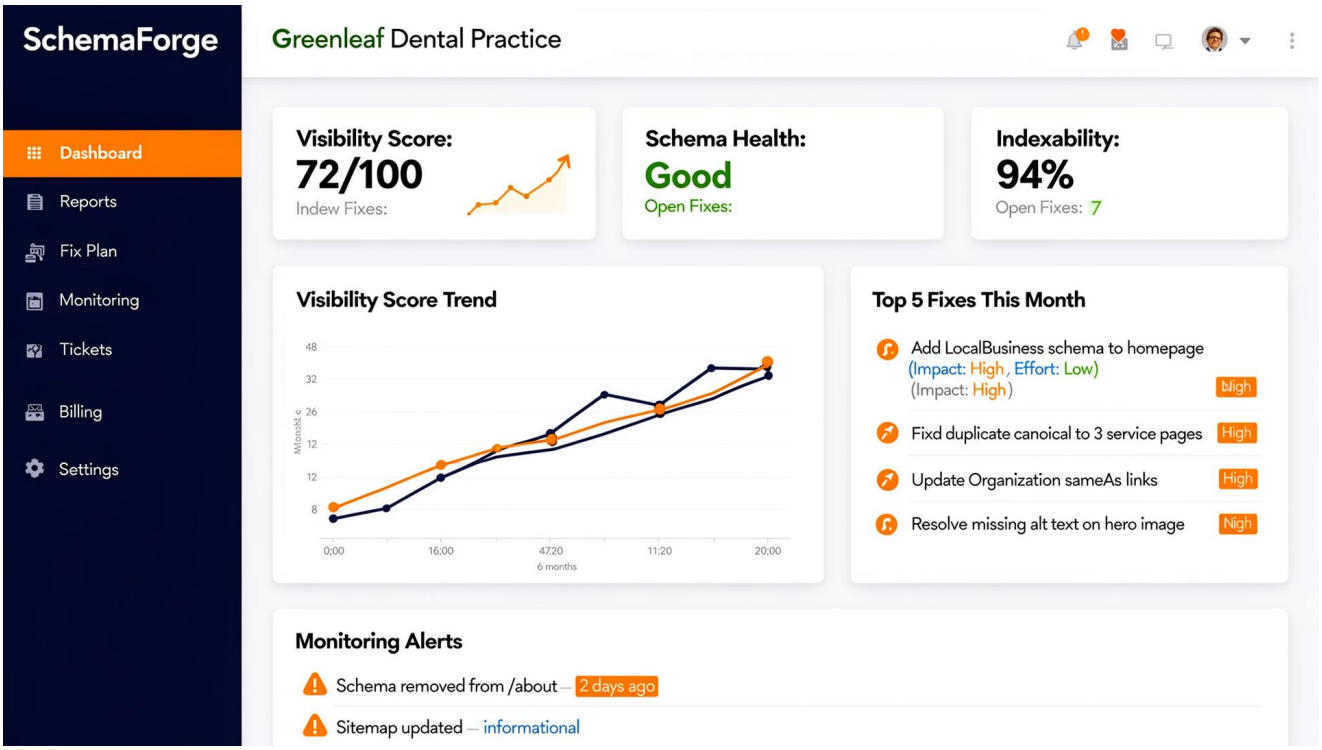

Fig. 3 — Client Dashboard

Fix Plan

Unified remediation board

Kanban merging fix_items and audit_issues into a single board (Recommended → Approved → In Progress → Deployed → Verified). Items carry impact/effort scores and an owner (client vs. SchemaForge), so the same surface drives both client self-service and our managed-service workflow.

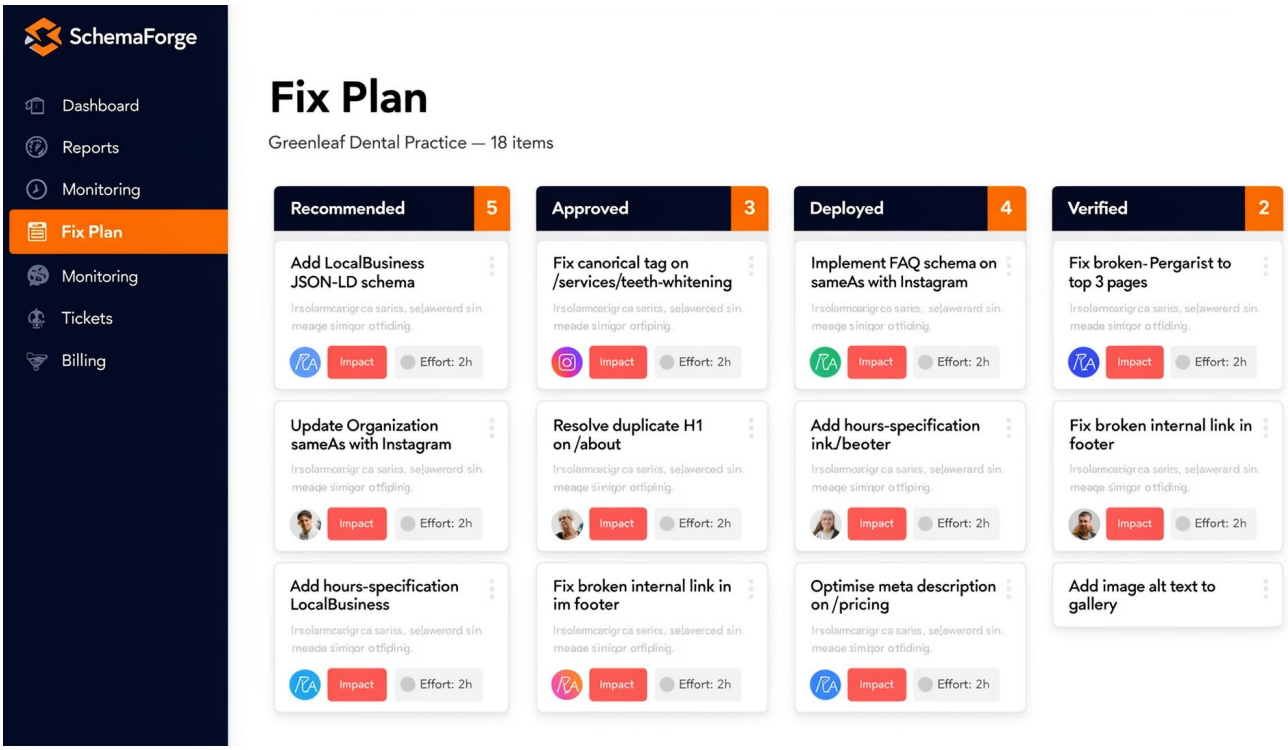

Fig. 4 — Fix Plan