Using AI to Build Internal Tooling at Speed

Praan Sanctuary — Device Monitoring & MQTT Debugging System

Context

Praan Sanctuary is a home automation platform for automated air quality and experience control, currently securing developer commitments for 500+ home deployments.

I owned the core architecture myself : backend APIs, database schema, parsing layers, and Docker deployment configs, along with the main client-facing app, since these needed precise control given they were customer-facing.

But shipping to 500+ homes meant I also needed a way to monitor device health, add/remove devices, and debug MQTT connectivity issues at scale, fast, without slowing down the client-facing work.

Decision Making : Where to Use AI, and Where Not To

Rather than defaulting to AI everywhere, I made a deliberate split based on stakes and precision needs:

Hand-built by me	Built with Claude Code
Backend APIs, DB schema, parsing layers, Docker configs, client-facing app	Internal monitoring dashboard, device add/remove & mode controls, MQTTX-like live packet inspector
Precision-critical, customer-facing, high cost of error	Velocity-critical, internal-only, high leverage on my time

This wasn't about trusting AI less for 'important' work — it was about matching the tool to the stakes. Customer-facing systems needed my full judgment on every line; internal tooling needed to exist fast so I could see how the fleet was actually behaving.

The Workflow

- Scoped the internal tool set myself: a dashboard to add/remove Sanctuary devices and change their modes, plus an MQTTX-like tool to record and inspect live MQTT packets for debugging and monitoring.
- Used Claude Code as the default build environment rather than writing this by hand, described the spec and data flow, let it scaffold the dashboard and packet inspector, then reviewed and iterated.
- Verified against real device traffic — checked that the packet inspector correctly surfaced live MQTT messages and that dashboard actions (mode changes, add/remove) reflected against actual device state.
- Iterated quickly on edge cases (e.g., malformed packets, offline devices) by describing the failure mode to Claude Code rather than debugging line-by-line myself first.

Outcome

- Built the entire monitoring + debugging system solo in about a week, roughly half the time a fully manual build would have taken, going from zero fleet visibility to a working system I could rely on for the 500+ home rollout.
- Gave me direct, live insight into device behavior across the fleet, without pulling time away from the client-facing app or core backend.
- The underlying Sanctuary platform is now part of the company's investor fundraise demos.

Reflection

As a builder, this is the split I default to: I stay hands-on for customer-facing, high-stakes work where precision and judgment matter most, and delegate lower-risk, internal-only surfaces — monitoring tools, internal dashboards — to Claude Code. That delegation isn't about doing less; it's about running things in parallel so the important work gets my full attention while the supporting infrastructure still gets built, instead of sitting in a backlog. The outcome is that I can optimize for quality where it counts and for speed everywhere else, without one coming at the cost of the other.