NEWS · OCTOBER 20, 2026

Not Vibe Coding: 58,588 Lines, 313 Tests and 48 Releases in 12 Days

Twelve days, 58,588 lines, 313 automated tests, 34 isolation checks, 52 pull requests and 48 releases -- the numbers and the guardrails behind Hearth, and why fast AI-assisted development and careful engineering are not opposites.

By Dr. Holger Flick

Whenever I show people what I build with AI, somebody asks the question politely, and somebody else asks it less politely: "Isn't that just vibe coding?" It's a fair question. The term comes from Andrej Karpathy, who described in February 2025 a way of programming where you "fully give in to the vibes, embrace exponentials, and forget that the code even exists" -- accepting every change without reading the diffs (Wikipedia collects the original post and the debate since). For a weekend prototype, that's a perfectly reasonable way to have fun.

For software that holds a family's insurance policies, medical bills and passwords, it is not. In my previous post, I showed Hearth, the household system I built for my wife in twelve days. This post is the other half of that story: the numbers, and the guardrails that produced them. My claim is simple, and I'll back it with the repository itself. Fast and careful aren't opposites -- AI made the careful parts cheaper too.

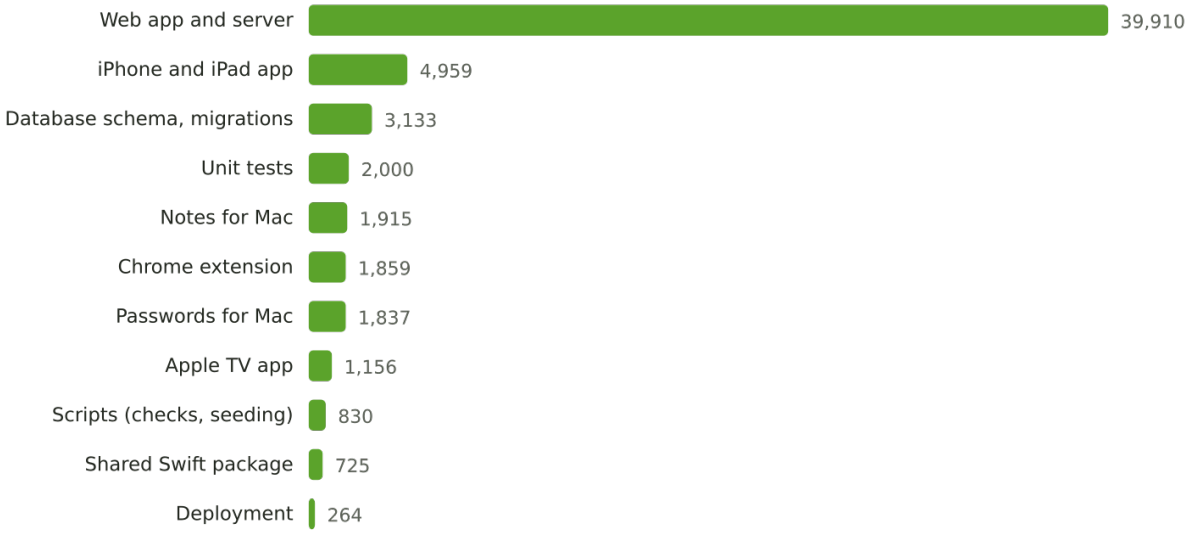
By the end, you'll know exactly what was measured, how the tests and reviews were set up, how the data is protected, and what I did versus what the AI did.

The numbers

Everything in this table comes straight from the Git repository, the test runner and my Claude Code session logs. The line counts leave out blank lines and generated code.

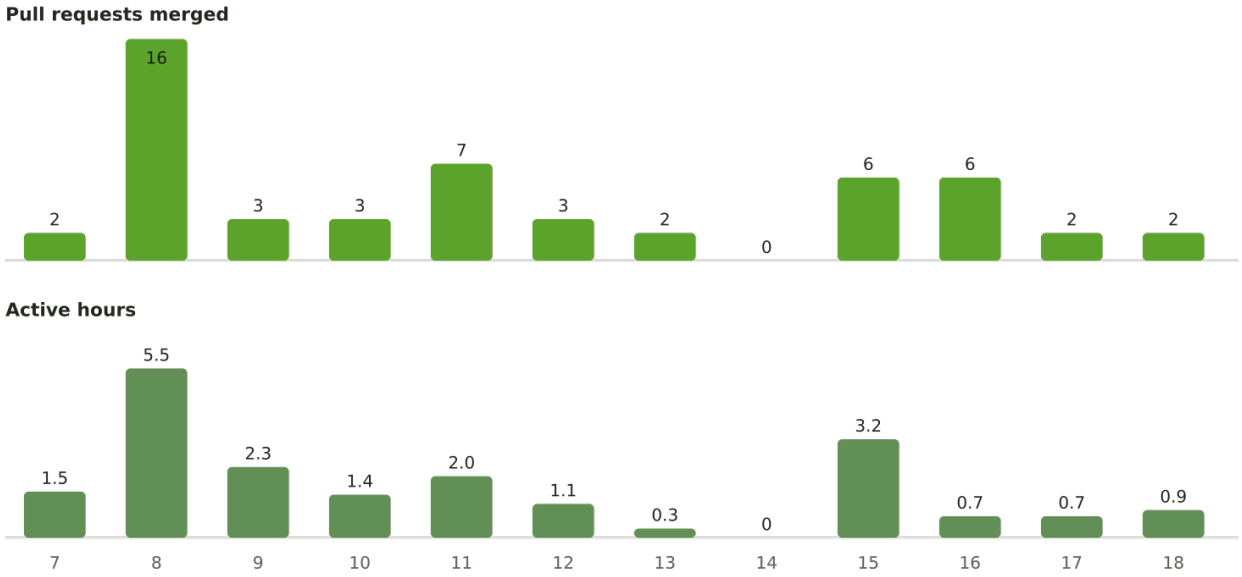
What	Count
Calendar days, first prompt to last release	12 (Oct 7 to Oct 18)
Active session time with the AI	about 20 hours
My messages to the AI	about 110
Non-blank lines of code (TypeScript, Swift, SQL, JavaScript)	58,588
Database tables / migrations	44 / 30
Merged pull requests / tagged releases	52 / 48
Automated tests (Vitest, Chrome extension, Swift package)	294 + 15 + 4 = 313
Checks in the family isolation script	34
Database triggers that enforce who owns what	49
Applications on one backend	7

Here is how those lines spread across the seven applications and their shared parts. The web application and its server carry most of the weight, as they should: that is where every rule lives, and every other app talks to it.



Non-blank lines of code per part of Hearth (58,588 in total)

The day-by-day picture is more interesting than the totals, because it shows the rhythm. Pull requests came in bursts, and the active hours stayed modest. The busiest day, October 8, saw sixteen merged pull requests in about five and a half hours of session time -- from family isolation and the Claude API to a calmer document list -- after I had used the first version for a day and knew what I wanted changed.



Merged pull requests (top) and active session hours (bottom) per day, October 7 to 18

"Active hours" counts the time a session was busy, with any break longer than twenty minutes left out. Several sessions ran in parallel at times, and those overlaps are counted once.

> **What the numbers don't say**
>
> Twenty hours of session time is not twenty hours of my life. Thinking, testing on the iPad and the Apple TV, and living with the app for a day before deciding what to change all happened around it. Lines of code don't measure value either; Dijkstra argued they should be booked as "lines spent". I use them here for one reason only: they give a sense of scale that the classic estimation models can compare against, which I did in the previous post.

Guardrail 1: every change is a pull request

Vibe coding has no history; you accept, you run, you move on. Hearth has 52 merged pull requests, and every one of them carries a written description of what changed and why. The bigger ones also say how the change was verified -- the security review's description, for instance, ends with "tsc, biome, `pnpm build` and `pnpm check:isolation` (26/26) pass", plus a database backup restored into a scratch database and compared with the live one.

Small pull requests are not bureaucracy. They are what lets you find the change that broke something, read a diff in five minutes instead of an hour, and roll back exactly one thing. The DORA research program has said for years that small batch sizes and thorough testing matter, and its 2024 report found that AI adoption went along with *lower* delivery stability. In my opinion, that finding is not an argument against AI. It is an argument for keeping the batches small when the AI makes it tempting to make them huge.

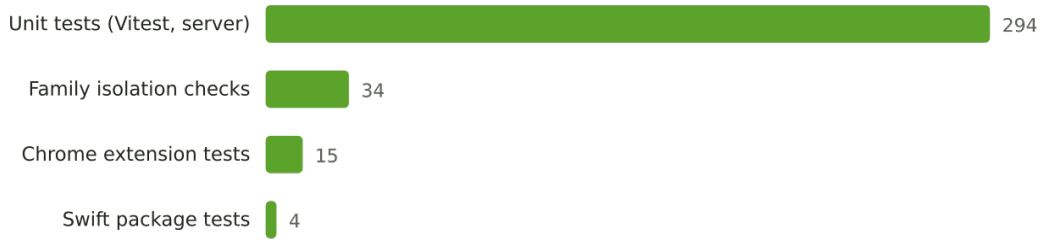
Guardrail 2: tests that found real bugs on day one

On October 12, the project got its unit tests: 186 at first, 294 today, written with Vitest for the pure functions in `src/lib` -- date math, schedules, business rules, parsers and request guards. They run in about half a second, so nobody has a reason to skip them. One detail I'm rather fond of: the test machine pretends to be in Honolulu,

which is neither UTC (the server) nor New York (the household). Code that forgets to name a time zone fails there instead of in production.

The first run found four real bugs, each one fixed with a test that failed first.

1. **Quotes on the wrong day.** Two quotes sent at 3 PM and 9:30 PM New York time landed in different "rounds", because 9:30 PM is already tomorrow in UTC. The cheaper afternoon quote counted as replaced.
2. **An open redirect.** The function that guards "return to this page after sign-in" accepted `/\t/evil.example`. Browsers drop tabs and line breaks, so that address led to another site.
3. **Inherited names.** A parser accepted names that every JavaScript object inherits, such as `toString`.
4. **Days that don't exist.** The date check accepted `2026-02-30`.



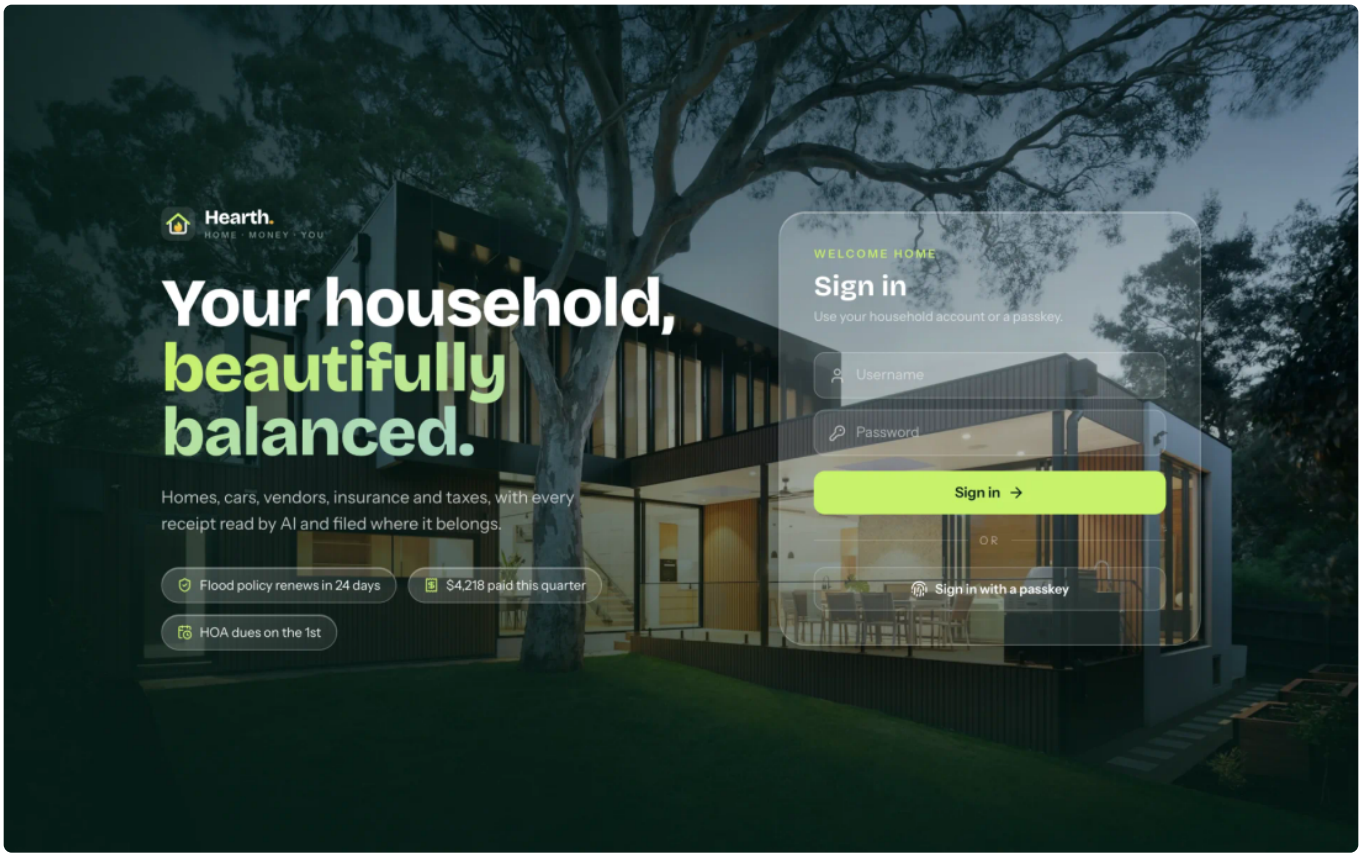
Automated checks in Hearth: unit tests, isolation checks, extension and Swift tests

The same day, the tests became a law. The deploy script runs them before it builds anything and refuses to ship when one fails, no matter who deploys. To prove the gate actually works, we broke the `relativeDays` function on purpose and watched the deploy stop with "Unit tests failed: not deploying." Then we fixed it again. The release instructions also forbid skipping, deleting or loosening a test to get a release through -- a rule written as much for the AI as for me.

> **Where the tests stop**
> The unit tests cover logic, not screens. There are no browser end-to-end tests yet, and the Swift apps have only a handful of tests in their shared package. Every screen change was checked by running the app -- in the browser, in the simulators, on the real iPad -- and the pull requests say so, but that is manual verification, not automation. For a product with many customers instead of one household, end-to-end tests would be the next thing I add.

Guardrail 3: sign-in, done properly

The sign-in is a good place to tell professional work from a demo, because a demo stops at "it lets me in". Hearth offers passwords and passkeys, the phishing-resistant sign-in standard from the FIDO Alliance built on the W3C's WebAuthn (implemented with SimpleWebAuthn). Passwords are stored with bcrypt; an unknown user name is checked against a dummy hash of the same cost, so the answer takes exactly as long as for a real one and doesn't reveal which accounts exist.

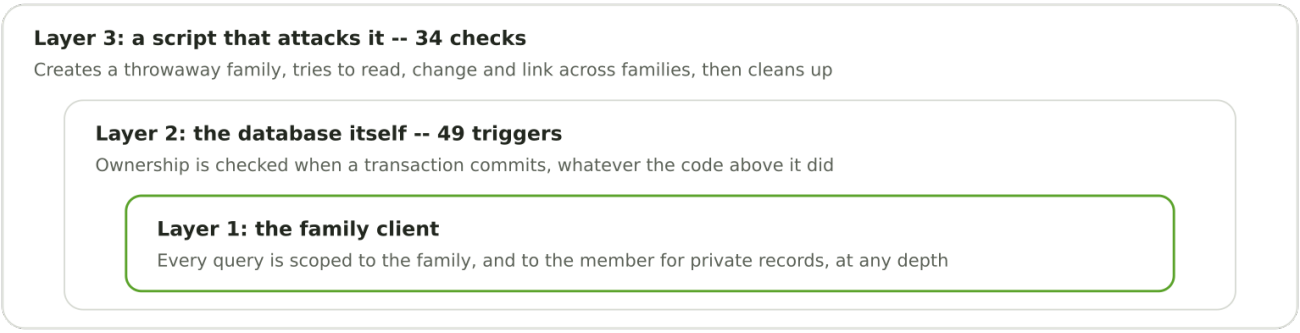


The sign-in page: a password or a passkey, with throttling behind both (demo instance).

The details are where the work is. Failed attempts are throttled per user name and per address, and the counters live in the database so a restart doesn't reset them. Session tokens are random and stored only as an HMAC, so a copy of the database signs nobody in, and the cookie is `httpOnly`, `secure` and `sameSite`. Every route handler checks the request's origin. Changing your password signs out your other sessions and every connected app, adding a passkey asks for the password first, and an administrator's password reset removes the passkeys too. And every server action and API route checks the user itself instead of trusting that some middleware already did.

Guardrail 4: one family can never see another

Hearth is multi-tenant: several families, and within a family, records that are shared and records that are private to one member. A mistake here isn't a bug; it's a privacy incident. So isolation is enforced three times, at three different levels.



Three independent layers keep one family's records away from another

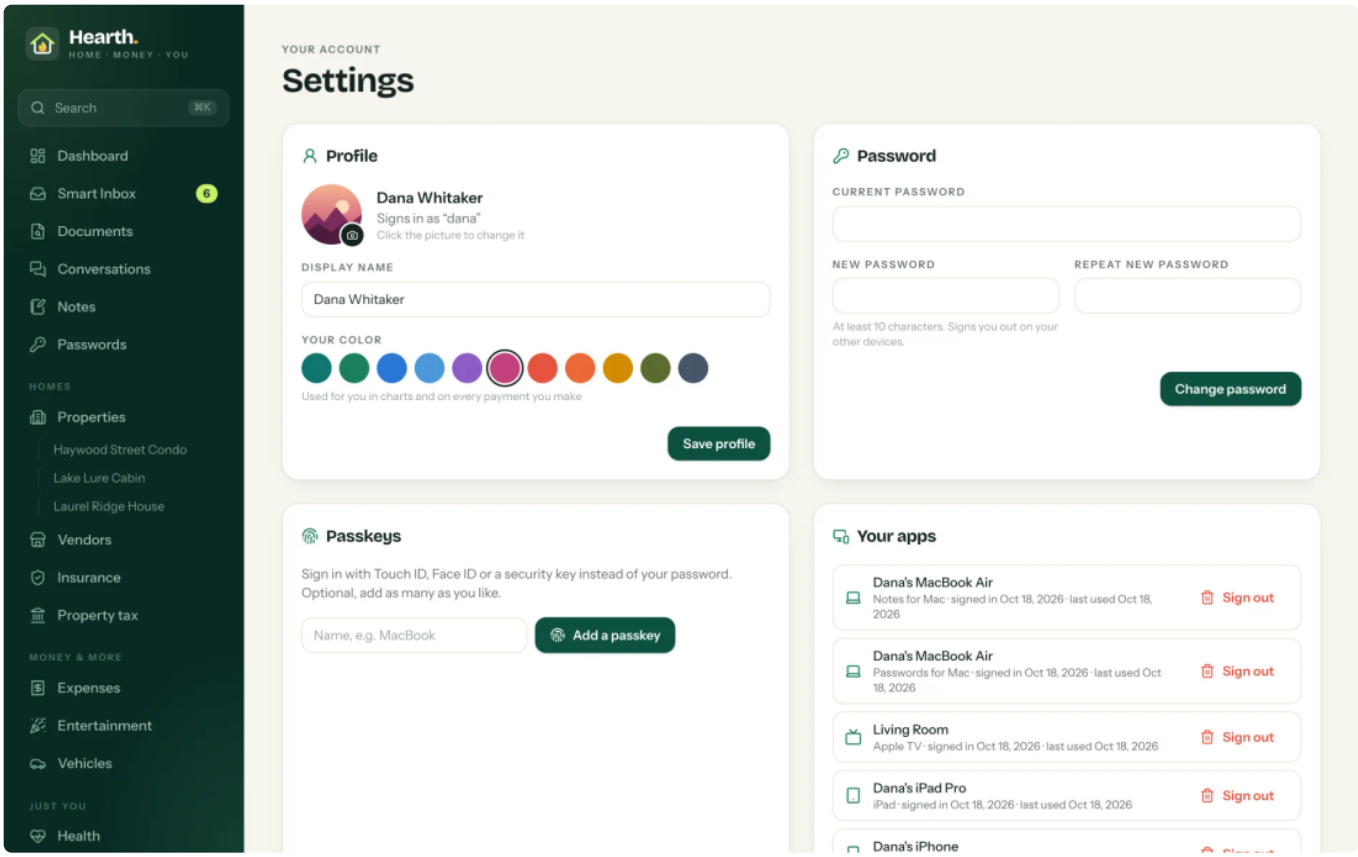
The inner layer is a database client that adds the family and member rules to every query automatically, including nested relations and counts, so a developer -- or an AI -- can't forget them. The middle layer doesn't trust the inner one: forty-nine triggers in PostgreSQL verify ownership when a transaction commits. The outer layer doesn't trust either of them: a script creates a throwaway family, tries 34 ways to read, change or link across the border, and then removes it again.

In my opinion, this is the single most important difference between a prototype and a product. Anybody can write `where familyId = ?`. The question is what happens on the day somebody forgets.

Guardrail 5: secrets stay secret

The password vault encrypts every field of an entry -- title, address, user name, password and notes -- with AES-256-GCM, with a key derived from a server secret. Each value is bound to its family, its entry and its field as additional authenticated data. Copy an encrypted password from one row into another, and it no longer opens. Earlier passwords stay in a history, encrypted the same way, so an old password is never lost.

The apps follow the same idea of least privilege. Each one signs in separately with a code confirmed on the web, and its token is stored only as an HMAC. Each kind of app may only do what it was made for: the Apple TV and the iPhone and iPad apps only read and never change anything, Notes for Mac sees only notes, and the Chrome extension sees only passwords. Inside the extension, only the background worker holds the token, and it answers a web page only about that page's own site, after a real click.



Settings ' Your apps: every signed-in app with its kind, when it signed in and when it was last used -- each one can be signed out on its own.

Guardrail 6: a review that found what the AI got wrong

On October 11, I had the AI review its own code base for authorization, privacy between members and security -- not as a formality, but with the explicit job to find problems. It found real ones, and that's the point of doing it. Among them was a **stored cross-site scripting** hole: an upload could choose its own content type, and a crafted file could have run script in the app. It also found places where editing a private record could make it visible to the rest of the family, and an AI usage report that showed document titles to the administrator.

All of those were fixed in the same pull request, together with the throttling in the database, the password-change sign-outs and nightly backups. I want to be honest about what this means. The AI wrote those bugs, and the AI found them -- because I asked for a review and treated it as a gate, not a ritual. That is no different from how a good team works with junior developers. Nobody ships without a second pair of eyes.

Guardrail 7: releases and operations

Hearth runs on a Raspberry Pi 5 in the house, in a Docker container behind Caddy, which takes care of the HTTPS certificates. Every release follows the same pipeline, and the pipeline is written down as a skill the AI executes step by step, so a release on a tired evening looks exactly like one on a fresh morning.

From pull request to production: every release takes the same path

The last five release images stay on the Pi, so a rollback is one line in the compose file. The database is dumped every night to a separate storage bucket, with daily and monthly copies, and a backup was restored into a scratch database and compared with the live one before I trusted it. I learned that lesson the hard way, as I described in My Server Was Compromised: the architecture decides the blast radius long before anything goes wrong.

What the AI did, and what I did

The division of labor is the part people find hardest to believe, so let me be concrete. In twelve days, I wrote about 110 messages; the AI made roughly 3,900 tool calls in reply -- reading files, writing code, running tests, building, deploying. It wrote nearly all of the code, the tests and the pull request descriptions.

What it didn't do was decide. I chose the architecture and the stack: Next.js 16 with React 19, Prisma 7 on PostgreSQL, TypeScript 7 with its new native compiler, SwiftUI for the Apple apps, and a Manifest V3 Chrome extension. I decided that isolation is enforced in the database and not only in code, that passwords are bound to their row, that each app gets only its own part, and that tests gate every release. I tested on real devices, wrote down the rules the app should follow ("dates first, newest on top", "no plain dropdowns, always searchable"), and said no -- often. When the iPad sign-in looked like the Apple TV's code screen, it was a person who noticed.

> **The research says: it depends on the developer**
>
> I don't want to pretend the evidence is one-sided. In a randomized controlled trial by METR, sixteen experienced open-source developers were 19% slower with early-2025 AI tools on their own large codebases, while believing they were faster. METR itself lists the limits -- a small sample, one kind of work, tools that have since changed -- and has published newer data. The lesson I draw is not that AI is slow or fast. It is that the gain depends on the person steering it: on how precisely they specify, how strictly they review, and whether they treat tests and reviews as gates. Without those, you get vibes. With them, you get Hearth.

What this proves

Fast and careful are not opposites anymore. The same AI that wrote 58,588 lines in twelve days also wrote the 313 tests, the 34 isolation checks, the 49 triggers and the security review that kept those lines honest -- because it was asked to, and because nothing reached production without passing them. The speed came from the tooling. The care came from thirty years of knowing what goes wrong in production, and from refusing to skip the boring parts.

> AI didn't remove the engineering. It removed the excuse for skipping it.

If you want this kind of speed for your own product -- a Delphi application that needs a modern future, or a system that fits your business instead of the other way around -- have a look at what I built, what I offer, and then let's talk.

INDEPENDENT · AD-FREE · READER-SUPPORTED

Free to read. Not free to make.

Every tutorial, article, and line of example code on Holger's Code is published without a paywall and without ads. This page is about what keeps it that way.

Professional work, published free

The content here is held to the same standard as professional client work: every tutorial is built as a real project, tested against current versions, documented step by step, and revisited when the frameworks move on. A single in-depth tutorial routinely takes days — and that's before the bills behind it: servers, storage, build infrastructure, licenses, and the hardware everything is verified on.

If an article or tutorial here has saved you an afternoon of debugging, taught you a technique, or helped you ship, it created real value for you. Supporting it isn't charity — it's paying for professional work you found useful, at whatever price you choose. And if you can't, keep reading anyway. That's what the free part means.

Ways to support

KO-FI

Buy me a coffee

A one-time tip — no account, no subscription, any amount.

BOOKS & COURSES

Buy the paid work

The paid tier of everything published free here — and you get something lasting in return.

SPREAD THE WORD

Share an article

Send a tutorial to a colleague, cite an article, subscribe to the RSS feed.

Nothing here goes behind a paywall, whether you chip in or not. But if something on this site earned its keep today, this is where to say so.

Support on Ko-fi —