NEWS · OCTOBER 20, 2026

Geen vibe coding: 58.588 regels, 313 tests en 48 releases in 12 dagen

Twaalf dagen, 58.588 regels, 313 geautomatiseerde tests, 34 isolatiecontroles, 52 pull requests en 48 releases -- de cijfers en de vangrails achter Hearth, en waarom snelle AI-ondersteunde ontwikkeling en zorgvuldige engineering geen tegenpolen zijn.

By Dr. Holger Flick



Telkens als ik mensen laat zien wat ik met AI bouw, stelt iemand de vraag beleefd, en iemand anders wat minder beleefd: "Is dat niet gewoon vibe coding?" Een terechte vraag. De term komt van Andrej Karpathy, die in februari 2025 een manier van programmeren beschreef waarbij je je "volledig overgeeft aan de vibes, exponentiële groei omarmt en vergeet dat de code überhaupt bestaat" -- en elke wijziging accepteert zonder de diffs te lezen (Wikipedia verzamelt het oorspronkelijke bericht en het debat dat sindsdien volgde). Voor een weekendprototype is dat een prima manier om plezier te hebben.

Voor software die de verzekeringspolissen, medische rekeningen en wachtwoorden van een gezin bewaart, is het dat niet. In mijn vorige post liet ik Hearth zien, het huishoudsysteem dat ik in twaalf dagen voor mijn vrouw bouwde. Deze post is de andere helft van dat verhaal: de cijfers, en de vangrails die ze hebben opgeleverd. Mijn stelling is eenvoudig, en ik onderbouw haar met de repository zelf. Snel en zorgvuldig zijn geen tegenpolen -- AI maakte ook de zorgvuldige delen goedkoper.

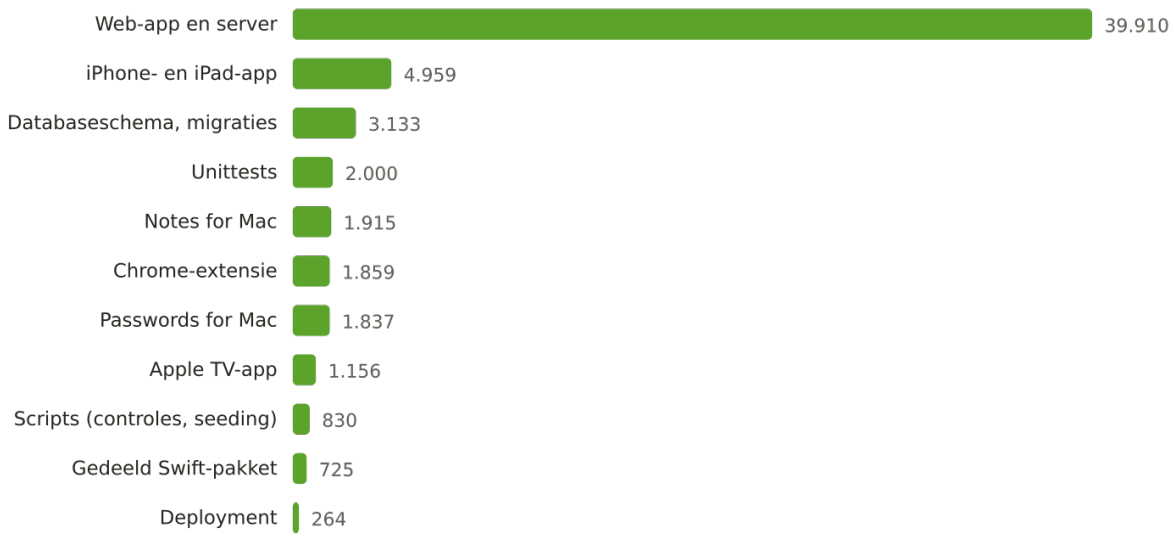
Aan het eind weet je precies wat er is gemeten, hoe de tests en reviews waren ingericht, hoe de data wordt beschermd, en wat ik deed en wat de AI deed.

De cijfers

Alles in deze tabel komt rechtstreeks uit de Git-repository, de testrunner en mijn Claude Code-sessielogs. De regeltellingen laten lege regels en gegenereerde code buiten beschouwing.

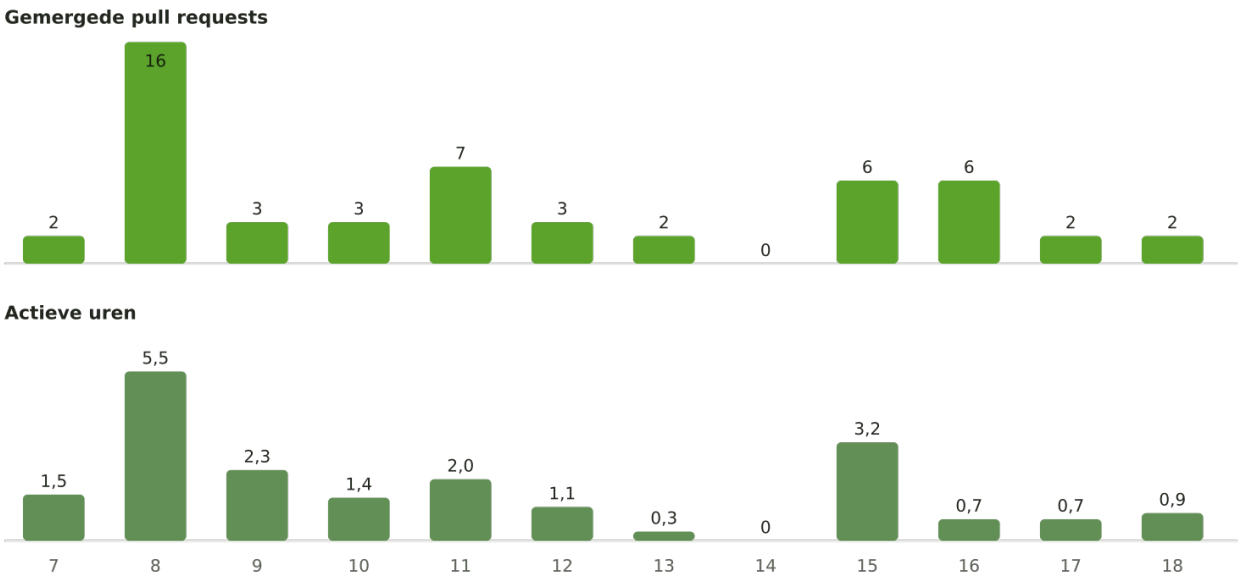
Wat	Aantal
Kalenderdagen, van eerste prompt tot laatste release	12 (7 t/m 18 okt.)
Actieve sessietijd met de AI	ongeveer 20 uur
Mijn berichten aan de AI	ongeveer 110
Niet-lege regels code (TypeScript, Swift, SQL, JavaScript)	58.588
Databasetabellen / migraties	44 / 30
Gemergede pull requests / getagde releases	52 / 48
Geautomatiseerde tests (Vitest, Chrome-extensie, Swift-pakket)	294 + 15 + 4 = 313
Controles in het script voor gezinsisolatie	34
Databasetriggers die afdwingen wie wat bezit	49
Applicaties op één backend	7

Zo verdelen die regels zich over de zeven applicaties en hun gedeelde onderdelen. De webapplicatie en haar server dragen het meeste gewicht, en zo hoort het ook: daar leeft elke regel, en elke andere app praat ermee.



Niet-lege regels code per onderdeel van Hearth (58.588 in totaal)

Het beeld per dag is interessanter dan de totalen, omdat het het ritme laat zien. Pull requests kwamen in golven, en de actieve uren bleven bescheiden. Op de drukste dag, 8 oktober, werden zestien pull requests gemerged in ongeveer vijfenhalf uur sessietijd -- van gezinsisolatie en de Claude API tot een rustigere documentenlijst -- nadat ik de eerste versie een dag had gebruikt en wist wat ik anders wilde.



Gemergede pull requests (boven) en actieve sessie-uren (onder) per dag, 7 t/m 18 oktober

"Actieve uren" telt de tijd waarin een sessie bezig was, zonder pauzes van meer dan twintig minuten. Soms liepen er meerdere sessies tegelijk, en die overlap is één keer geteld.

> **Wat de cijfers niet zeggen**
>
> Twintig uur sessietijd is geen twintig uur van mijn leven. Nadenken, testen op de iPad en de Apple TV, en een dag met de app leven voordat ik besloot wat er moest veranderen, gebeurde er allemaal omheen. Regels code meten ook geen waarde; Dijkstra betoogde dat je ze moet boeken als "verbruikte regels". Ik gebruik ze hier om maar één reden: ze geven een idee van de omvang waarmee de klassieke schattingsmodellen kunnen vergelijken, en dat heb ik gedaan in de vorige post.

Vangrail 1: elke wijziging is een pull request

Vibe coding heeft geen geschiedenis; je accepteert, je draait, je gaat door. Hearth heeft 52 gemergede pull requests, en elk daarvan heeft een geschreven beschrijving van wat er veranderde en waarom. De grotere vermelden ook hoe de wijziging is geverifieerd -- de beschrijving van de securityreview eindigt bijvoorbeeld met "tsc, biome, `pnpm build` and `pnpm check:isolation` (26/26) pass", plus een databaseback-up die in een kladdatabase werd teruggezet en met de live database vergeleken.

Kleine pull requests zijn geen bureaucratie. Ze zorgen ervoor dat je de wijziging vindt die iets kapotmaakte, dat je een diff in vijf minuten leest in plaats van in een uur, en dat je precies één ding kunt terugdraaien. Het DORA-onderzoeksprogramma zegt al jaren dat kleine batches en grondig testen ertoe doen, en het rapport van 2024 vond dat de inzet van AI samenging met een *lagere* stabiliteit van opleveringen. Volgens mij is die bevinding geen argument tegen AI. Het is een argument om de batches klein te houden juist wanneer de AI het verleidelijk maakt ze enorm te maken.

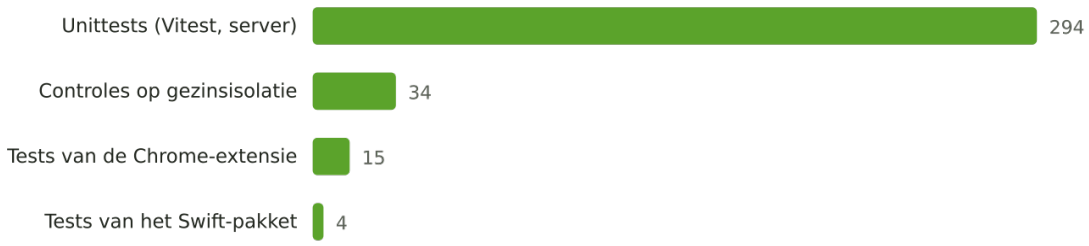
Vangrail 2: tests die op dag één echte bugs vonden

Op 12 oktober kreeg het project zijn unittests: eerst 186, nu 294, geschreven met Vitest voor de pure functies in `src/lib` -- datumrekenwerk, planningen, bedrijfsregels, parsers en beveiligingen van verzoeken. Ze draaien in ongeveer een halve seconde, dus niemand heeft een reden om ze over te slaan. Eén detail waar ik nogal

op gesteld ben: de testmachine doet alsof ze in Honolulu staat, en dat is noch UTC (de server) noch New York (het huishouden). Code die vergeet een tijdzone te noemen, faalt daar in plaats van in productie.

De eerste run vond vier echte bugs, en elk ervan werd opgelost met een test die eerst faalde.

1. **Offertes op de verkeerde dag.** Twee offertes, verstuurd om 15.00 uur en 21.30 uur New Yorkse tijd, belandden in verschillende "rondes", omdat het om 21.30 uur in UTC al de volgende dag is. De goedkopere offerte van 's middags telde als vervangen.
2. **Een open redirect.** De functie die "na het inloggen terug naar deze pagina" bewaakt, accepteerde `/\t/evil.example`. Browsers laten tabs en regeleinden weg, dus dat adres leidde naar een andere site.
3. **Geërfde namen.** Een parser accepteerde namen die elk JavaScript-object erft, zoals `toString`.
4. **Dagen die niet bestaan.** De datumcontrole accepteerde `2026-02-30`.



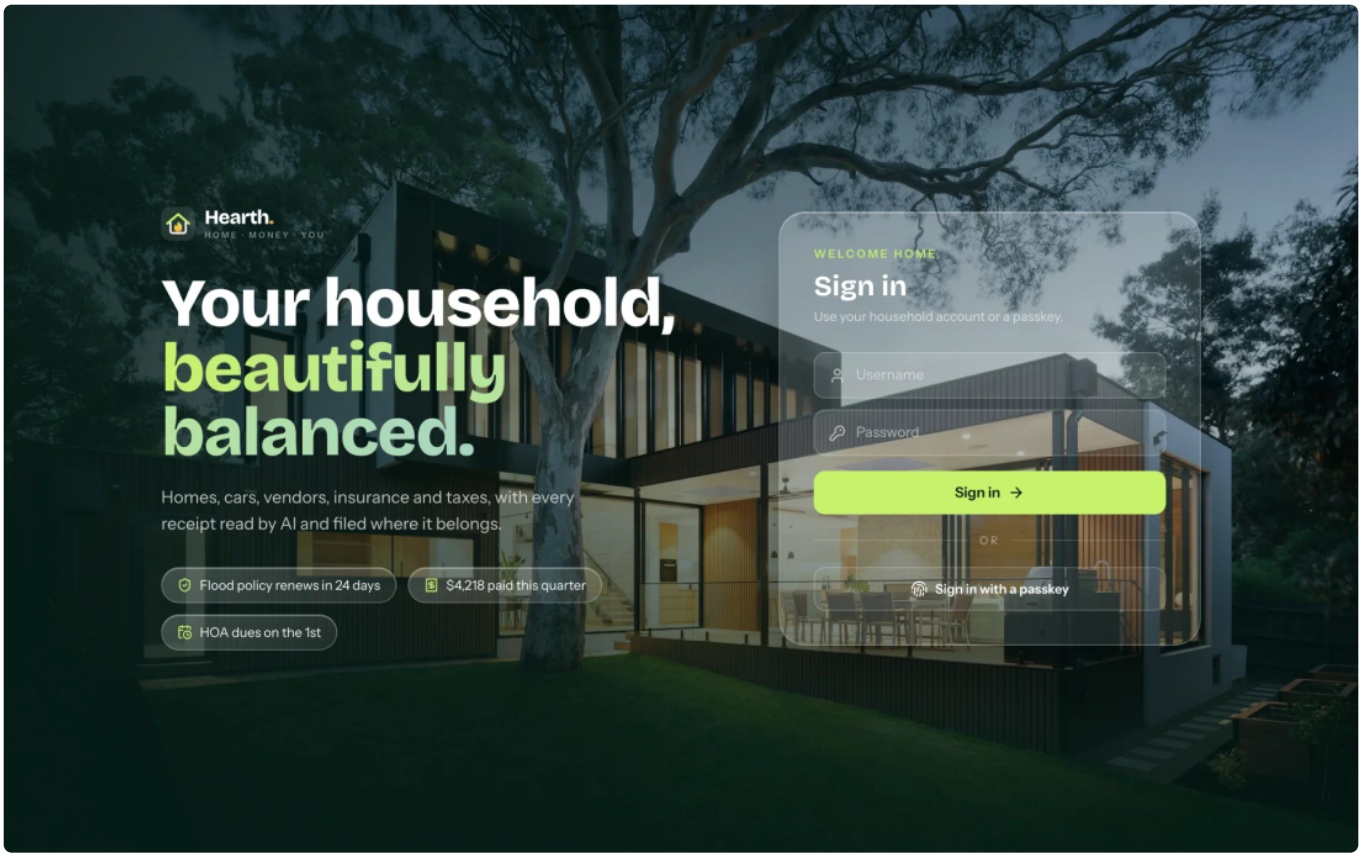
Geautomatiseerde controles in Hearth: unittests, isolatiecontroles, extensie- en Swift-tests

Diezelfde dag werden de tests wet. Het deployscript draait ze voordat het iets bouwt en weigert te shippen als er een faalt, ongeacht wie er deployt. Om te bewijzen dat de poort echt werkt, maakten we de functie `relativeDays` met opzet stuk en zagen we de deployment stoppen met "Unit tests failed: not deploying." Daarna repareerden we hem weer. De release-instructies verbieden bovendien om een test over te slaan, te verwijderen of te versoepelen om een release erdoor te krijgen -- een regel die net zo goed voor de AI is geschreven als voor mij.

> **Waar de tests ophouden**
> De unittests dekken logica af, geen schermen. Er zijn nog geen end-to-end-tests in de browser, en de Swift-apps hebben maar een handvol tests in hun gedeelde pakket. Elke schermwijziging is gecontroleerd door de app te draaien -- in de browser, in de simulators, op de echte iPad -- en de pull requests vermelden dat, maar dat is handmatige verificatie, geen automatisering. Voor een product met veel klanten in plaats van één huishouden zouden end-to-end-tests het eerste zijn wat ik toevoeg.

Vangrail 3: inloggen, goed gedaan

Het inloggen is een goede plek om professioneel werk van een demo te onderscheiden, want een demo houdt op bij "ik kom erin". Hearth biedt wachtwoorden en passkeys, de phishingbestendige inlogstandaard van de FIDO Alliance, gebouwd op WebAuthn van het W3C (geïmplementeerd met SimpleWebAuthn). Wachtwoorden worden opgeslagen met bcrypt; een onbekende gebruikersnaam wordt gecontroleerd tegen een dummyhash met dezelfde kosten, zodat het antwoord precies even lang duurt als bij een echte en niet verraadt welke accounts er bestaan.



De inlogpagina: een wachtwoord of een passkey, met throttling achter allebei (demo-instantie).

In de details zit het werk. Mislukte pogingen worden afgeremd per gebruikersnaam en per adres, en de tellers staan in de database, zodat een herstart ze niet terugzet. Sessietokens zijn willekeurig en worden alleen als HMAC opgeslagen, dus met een kopie van de database logt niemand in, en de cookie is `httpOnly`, `secure` en `sameSite`. Elke route handler controleert de herkomst van het verzoek. Als je je wachtwoord wijzigt, worden je andere sessies en elke gekoppelde app uitgelogd, een passkey toevoegen vraagt eerst om het wachtwoord, en als een beheerder een wachtwoord reset, verdwijnen ook de passkeys. En elke server action en API-route controleert de gebruiker zelf, in plaats van erop te vertrouwen dat een of andere middleware dat al deed.

Vangrail 4: het ene gezin kan nooit bij het andere

Hearth is multi-tenant: meerdere gezinnen, en binnen een gezin records die gedeeld zijn en records die privé zijn voor één gezinslid. Een fout hier is geen bug; het is een privacy-incident. Daarom wordt isolatie drie keer afgedwongen, op drie verschillende niveaus.



Drie onafhankelijke lagen houden de records van het ene gezin weg bij het andere

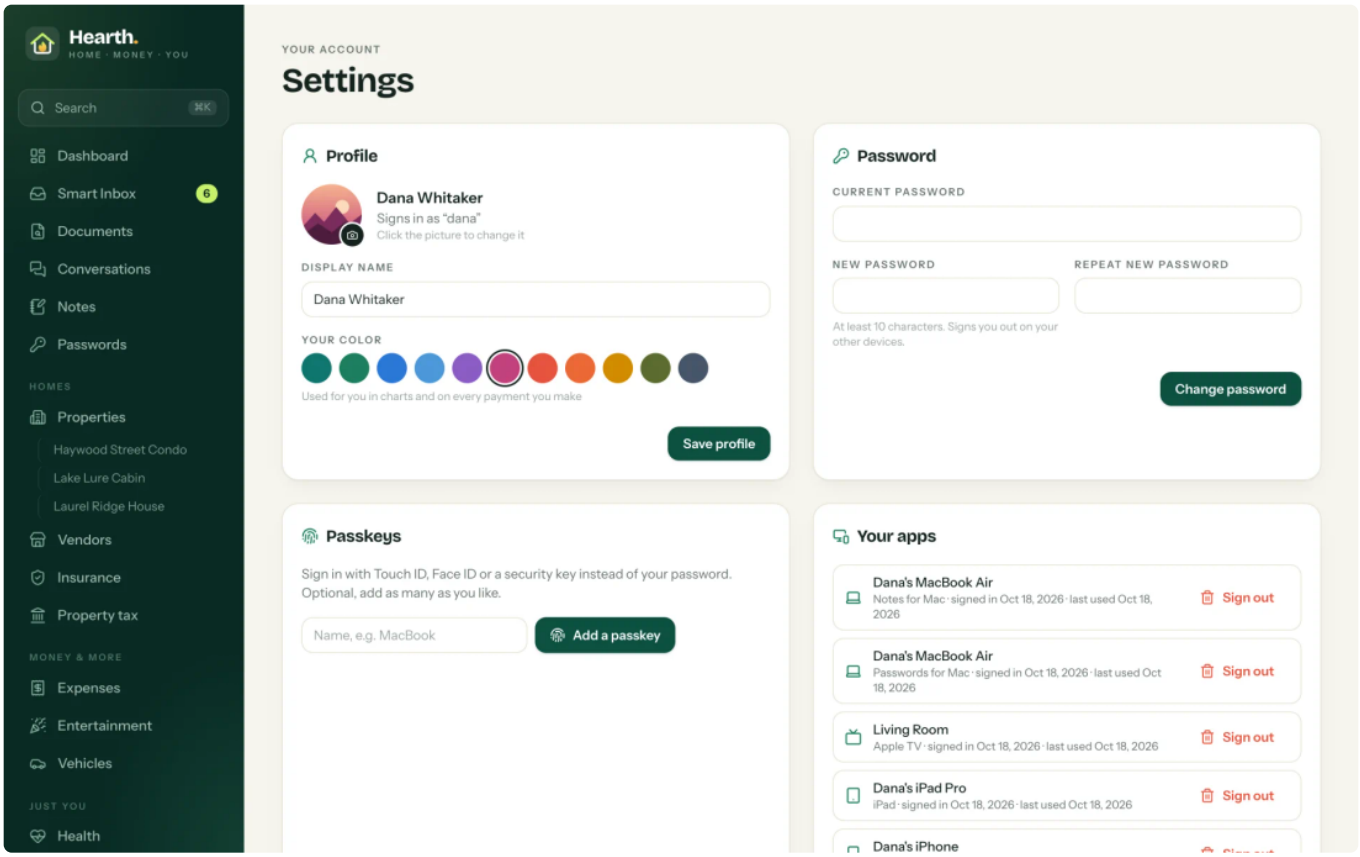
De binnenste laag is een databaseclient die de regels voor gezin en gezinslid automatisch aan elke query toevoegt, ook aan geneste relaties en tellingen, zodat een ontwikkelaar -- of een AI -- ze niet kan vergeten. De middelste laag vertrouwt de binnenste niet: negenenveertig triggers in PostgreSQL verifiëren het eigendom wanneer een transactie wordt gecommit. De buitenste laag vertrouwt geen van beide: een script maakt een wegwerpgezin aan, probeert op 34 manieren over de grens heen te lezen, te wijzigen of te koppelen, en verwijdert het daarna weer.

Volgens mij is dit het allerbelangrijkste verschil tussen een prototype en een product. Iedereen kan `where familyId = ?` schrijven. De vraag is wat er gebeurt op de dag dat iemand het vergeet.

Vangrail 5: geheimen blijven geheim

De wachtwoordkluis versleutelt elk veld van een item -- titel, adres, gebruikersnaam, wachtwoord en notities -- met AES-256-GCM, met een sleutel die is afgeleid van een servergeheim. Elke waarde is als additional authenticated data gebonden aan haar gezin, haar item en haar veld. Kopieer je een versleuteld wachtwoord van de ene rij naar een andere, dan gaat het niet meer open. Eerdere wachtwoorden blijven bewaard in een geschiedenis, op dezelfde manier versleuteld, zodat een oud wachtwoord nooit verloren gaat.

De apps volgen hetzelfde principe van minimale rechten. Elke app logt apart in met een code die op het web wordt bevestigd, en zijn token wordt alleen als HMAC opgeslagen. Elke soort app mag alleen doen waarvoor hij gemaakt is: de Apple TV-app en de apps voor iPhone en iPad lezen alleen en wijzigen nooit iets, Notes for Mac ziet alleen notities, en de Chrome-extensie ziet alleen wachtwoorden. Binnen de extensie heeft alleen de background worker het token, en die geeft een webpagina alleen antwoord over de eigen site van die pagina, na een echte klik.



Instellingen ’ Je apps: elke ingelogde app met zijn soort, wanneer hij inlogde en wanneer hij voor het laatst werd gebruikt -- elke app kan afzonderlijk worden uitgelogd.

Vangrail 6: een review die vond wat de AI fout deed

Op 11 oktober liet ik de AI zijn eigen codebase reviewen op autorisatie, privacy tussen gezinsleden en security -- niet als formaliteit, maar met de uitdrukkelijke opdracht problemen te vinden. Hij vond echte problemen, en daar doe je het voor. Daaronder zat een **stored cross-site scripting**-lek: een upload kon zijn eigen content

type kiezen, en een gemanipuleerd bestand had script in de app kunnen uitvoeren. Hij vond ook plekken waar het bewerken van een privérecord dat record zichtbaar kon maken voor de rest van het gezin, en een rapport over AI-gebruik dat documenttitels aan de beheerder liet zien.

Al die problemen werden opgelost in dezelfde pull request, samen met de throttling in de database, het uitloggen bij een wachtwoordwijziging en nachtelijke back-ups. Ik wil eerlijk zijn over wat dit betekent. De AI schreef die bugs, en de AI vond ze -- omdat ik om een review vroeg en die als poort behandelde, niet als ritueel. Dat is niet anders dan hoe een goed team met junior ontwikkelaars werkt. Niemand shipt zonder dat een tweede paar ogen ernaar heeft gekeken.

Vangrail 7: releases en beheer

Hearth draait op een Raspberry Pi 5 in huis, in een Docker-container achter Caddy, dat voor de HTTPS-cer-

tificaten zorgt. Elke release volgt dezelfde pijplijn, en die pijplijn is vastgelegd als een skill die de AI stap voor stap uitvoert, zodat een release op een vermoeide avond er precies zo uitziet als een op een frisse ochtend.



Van pull request naar productie: elke release volgt hetzelfde pad

De laatste vijf release-images blijven op de Pi staan, dus terugrollen is één regel in het compose-bestand. De database wordt elke nacht gedumpt naar een aparte storage bucket, met dagelijkse en maandelijkse kopieën, en een back-up is teruggezet in een kladdatabase en vergeleken met de live database voordat ik erop vertrouwde. Die les leerde ik op de harde manier, zoals ik beschreef in My Server Was Compromised: de architectuur bepaalt de blast radius lang voordat er iets misgaat.

Wat de AI deed, en wat ik deed

De taakverdeling is het deel dat mensen het moeilijkst kunnen geloven, dus laat ik concreet zijn. In twaalf dagen schreef ik ongeveer 110 berichten; de AI deed als antwoord daarop zo'n 3.900 tool calls -- bestanden lezen, code schrijven, tests draaien, bouwen, deployen. Hij schreef bijna alle code, de tests en de beschrijvingen van de pull requests.

Wat hij niet deed, was beslissen. Ik koos de architectuur en de stack: Next.js 16 met React 19, Prisma 7 op PostgreSQL, TypeScript 7 met zijn nieuwe native compiler, SwiftUI voor de Apple-apps, en een Chrome-extensie op Manifest V3. Ik besliste dat isolatie in de database wordt afgedwongen en niet alleen in code, dat wachtwoorden aan hun rij gebonden zijn, dat elke app alleen zijn eigen deel krijgt, en dat tests elke release bewaken. Ik testte op echte apparaten, legde de regels vast die de app moet volgen ("datums eerst, nieuwste bovenaan", "geen kale dropdowns, altijd doorzoekbaar"), en zei nee -- vaak. Toen het inlogscherm van de iPad eruitzag als het codescherm van de Apple TV, was het een mens die dat opmerkte.

> **Het onderzoek zegt: het hangt af van de ontwikkelaar**
>
> Ik wil niet doen alsof het bewijs eenzijdig is. In een gerandomiseerde gecontroleerde studie van METR waren zestien ervaren opensource-ontwikkelaars met AI-tools van begin 2025 19% langzamer in hun eigen grote codebases, terwijl ze dachten dat ze sneller waren. METR noemt zelf de beperkingen -- een kleine steekproef, één soort werk, tools die inmiddels veranderd zijn -- en heeft nieuwere data gepubliceerd. De les die ik eruit trek, is niet dat AI traag of snel is. Het is dat de winst afhangt van wie het stuur in handen heeft: van hoe precies die specificeert, hoe streng die reviewt, en of die tests en reviews als poort behandelt. Zonder dat krijg je vibes. Met dat krijg je Hearth.

Wat dit bewijst

Snel en zorgvuldig zijn geen tegenpolen meer. Dezelfde AI die in twaalf dagen 58.588 regels schreef, schreef ook de 313 tests, de 34 isolatiecontroles, de 49 triggers en de securityreview die die regels eerlijk hielden -- omdat hem dat gevraagd werd, en omdat niets in productie kwam zonder ze te doorstaan. De snelheid kwam van de tools. De zorgvuldigheid kwam van dertig jaar weten wat er in productie misgaat, en van weigeren de saaie delen over te slaan.

> AI heeft de engineering niet weggenomen. Het heeft het excuus weggenomen om haar over te slaan.

Wil je dit soort snelheid voor je eigen product -- een Delphi-applicatie die een moderne toekomst nodig heeft, of een systeem dat bij je bedrijf past in plaats van andersom -- kijk dan eens naar wat ik heb gebouwd, wat ik aanbied, en dan: laten we praten.

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 —