NEWS · OCTOBER 20, 2026

Pas du vibe coding : 58 588 lignes, 313 tests et 48 versions en 12 jours

Douze jours, 58 588 lignes, 313 tests automatisés, 34 contrôles d'isolation, 52 pull requests et 48 versions -- les chiffres et les garde-fous derrière Hearth, et pourquoi développement rapide assisté par IA et ingénierie rigoureuse ne s'opposent pas.

By Dr. Holger Flick

Chaque fois que je montre ce que je construis avec l'IA, quelqu'un pose la question poliment, et quelqu'un d'autre la pose moins poliment : « Ce n'est pas simplement du vibe coding ? » La question est légitime. Le terme vient d'Andrej Karpathy, qui a décrit en février 2025 une façon de programmer où l'on « se laisse complètement porter par les vibes, embrasse les exponentielles et oublie jusqu'à l'existence du code » -- en acceptant chaque modification sans lire les diffs (Wikipédia rassemble le message d'origine et le débat qui a suivi). Pour un prototype de week-end, c'est une manière parfaitement raisonnable de s'amuser.

Pour un logiciel qui contient les contrats d'assurance, les factures médicales et les mots de passe d'une famille, ce n'en est pas une. Dans mon article précédent, j'ai présenté Hearth, le système domestique que j'ai construit pour ma femme en douze jours. Cet article est l'autre moitié de l'histoire : les chiffres, et les garde-fous qui les ont produits. Ma thèse est simple, et je vais l'étayer avec le dépôt lui-même. Rapide et rigoureux ne sont pas des contraires -- l'IA a aussi rendu les parties rigoureuses moins coûteuses.

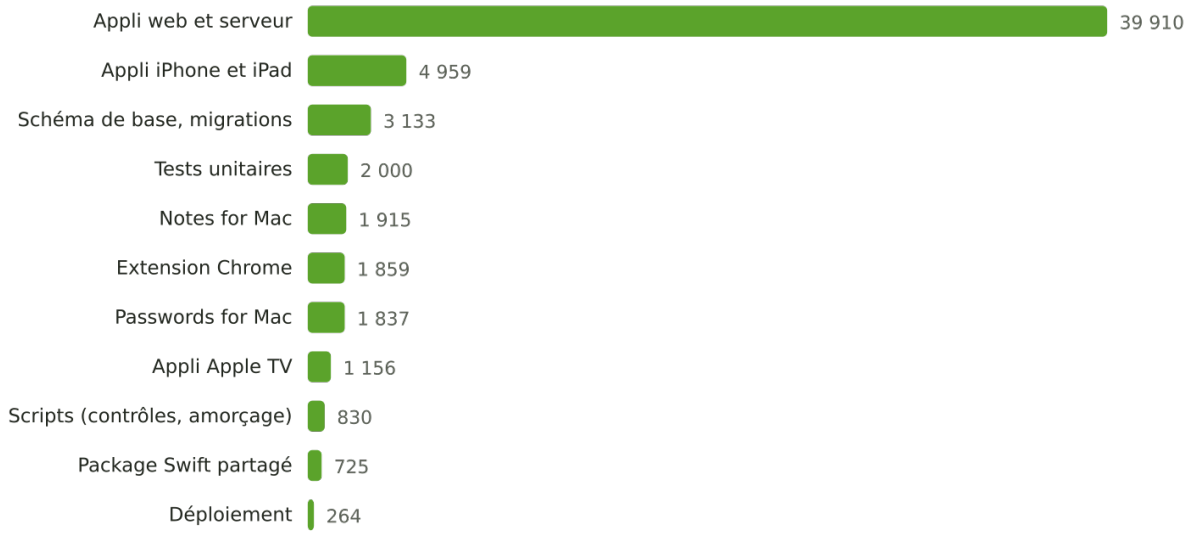
À la fin, vous saurez exactement ce qui a été mesuré, comment les tests et les revues ont été organisés, comment les données sont protégées, et ce qui revient à moi et ce qui revient à l'IA.

Les chiffres

Tout ce qui figure dans ce tableau provient directement du dépôt Git, de l'exécuteur de tests et des journaux de mes sessions Claude Code. Le décompte des lignes exclut les lignes vides et le code généré.

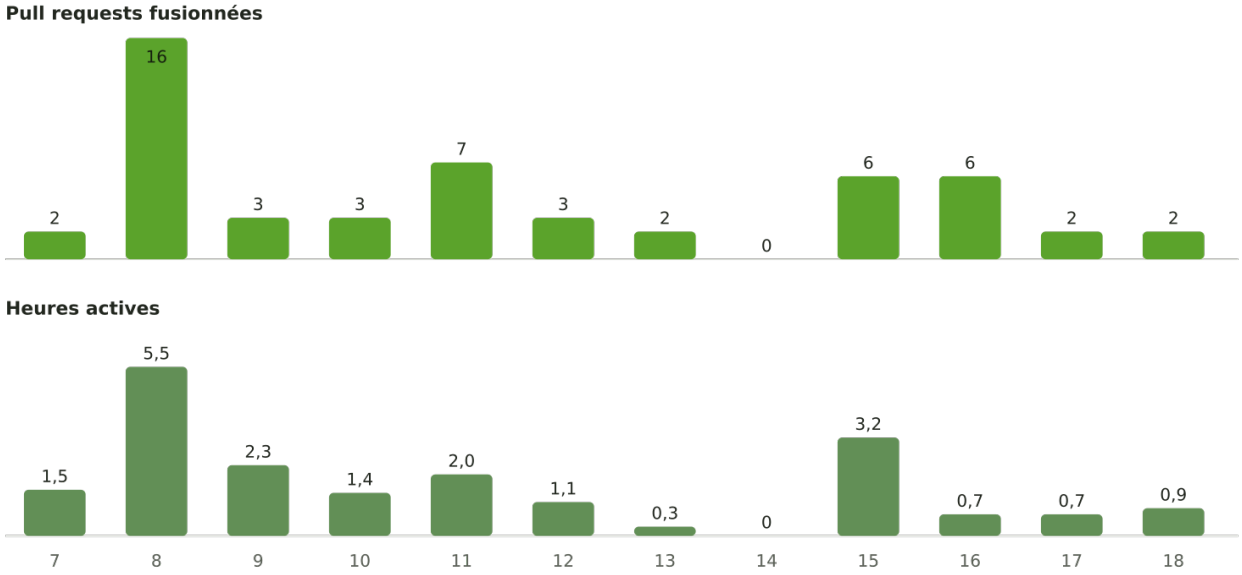
Quoi	Nombre
Jours calendaires, du premier prompt à la dernière version	12 (du 7 au 18 octobre)
Temps de session actif avec l'IA	environ 20 heures
Mes messages à l'IA	environ 110
Lignes de code non vides (TypeScript, Swift, SQL, JavaScript)	58 588
Tables / migrations de la base de données	44 / 30
Pull requests fusionnées / versions taguées	52 / 48
Tests automatisés (Vitest, extension Chrome, package Swift)	294 + 15 + 4 = 313
Contrôles du script d'isolation des familles	34
Triggers de base de données qui garantissent qui possède quoi	49
Applications sur un seul backend	7

Voici comment ces lignes se répartissent entre les sept applications et leurs parties communes. L'application web et son serveur portent l'essentiel du poids, et c'est normal : c'est là que vivent toutes les règles, et toutes les autres applis dialoguent avec elle.



Lignes de code non vides par partie de Hearth (58 588 au total)

Le détail jour par jour est plus intéressant que les totaux, parce qu'il montre le rythme. Les pull requests sont arrivées par vagues, et les heures actives sont restées modestes. Le jour le plus chargé, le 8 octobre, a vu seize pull requests fusionnées en cinq heures et demie de session environ -- de l'isolation des familles et de l'API Claude jusqu'à une liste de documents plus apaisée --, après que j'ai utilisé la première version pendant une journée et que j'ai su ce que je voulais changer.



Pull requests fusionnées (en haut) et heures de session actives (en bas) par jour, du 7 au 18 octobre

Les « heures actives » comptent le temps pendant lequel une session était occupée, toute pause de plus de vingt minutes étant exclue. Plusieurs sessions ont parfois tourné en parallèle ; ces chevauchements ne sont comptés qu'une fois.

> **Ce que les chiffres ne disent pas**
>
> Vingt heures de session, ce n'est pas vingt heures de ma vie. La réflexion, les tests sur l'iPad et l'Apple TV, et le fait de vivre une journée avec l'appli avant de décider quoi changer, tout cela s'est passé autour. Le nombre de lignes de code ne mesure pas non plus la valeur ; Dijkstra soutenait qu'il faudrait les comptabiliser comme « lignes dépensées ». Je ne les utilise ici que pour une seule raison : elles donnent un ordre de grandeur auquel les modèles d'estimation classiques peuvent se mesurer, ce que j'ai fait dans l'article précédent.

Garde-fou 1 : chaque modification est une pull request

Le vibe coding n'a pas d'historique ; on accepte, on lance, on passe à autre chose. Hearth compte 52 pull requests fusionnées, et chacune porte une description écrite de ce qui a changé et pourquoi. Les plus importantes indiquent aussi comment la modification a été vérifiée -- la description de la revue de sécurité, par exemple, se termine par « tsc, biome, `pnpm build` et `pnpm check:isolation` (26/26) passent », plus une sauvegarde de la base de données restaurée dans une base de travail et comparée à la base de production.

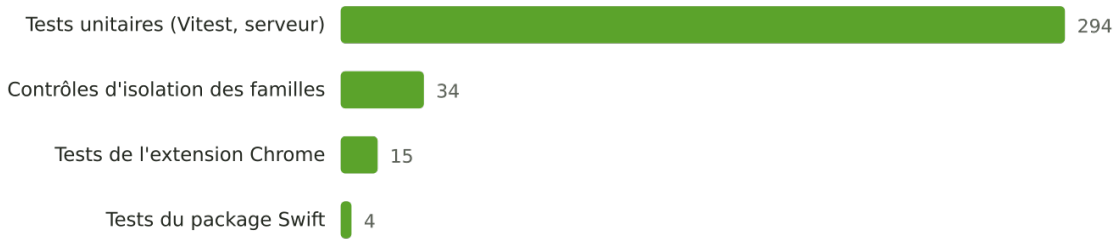
Les petites pull requests ne relèvent pas de la bureaucratie. Ce sont elles qui permettent de retrouver la modification qui a cassé quelque chose, de lire un diff en cinq minutes au lieu d'une heure, et d'annuler exactement une seule chose. Le programme de recherche DORA répète depuis des années que des lots de petite taille et des tests approfondis comptent, et son rapport 2024 a constaté que l'adoption de l'IA allait de pair avec une stabilité des livraisons *moindre*. À mon avis, ce constat n'est pas un argument contre l'IA. C'est un argument pour garder des lots petits quand l'IA donne envie de les faire énormes.

Garde-fou 2 : des tests qui ont trouvé de vrais bugs dès le premier jour

Le 12 octobre, le projet a reçu ses tests unitaires : 186 au départ, 294 aujourd'hui, écrits avec Vitest pour les fonctions pures de `src/lib` -- calculs de dates, échéanciers, règles métier, parseurs et gardes de requêtes. Ils s'exécutent en une demi-seconde environ, si bien que personne n'a de raison de les sauter. Un détail auquel je tiens particulièrement : la machine de test fait comme si elle se trouvait à Honolulu, qui n'est ni en UTC (le serveur) ni à New York (le foyer). Le code qui oublie de préciser un fuseau horaire échoue là-bas plutôt qu'en production.

La première exécution a trouvé quatre vrais bugs, chacun corrigé avec un test qui échouait d'abord.

1. **Des devis au mauvais jour.** Deux devis envoyés à 15 h et à 21 h 30, heure de New York, se sont retrouvés dans des « tours » différents, parce qu'à 21 h 30, il est déjà le lendemain en UTC. Le devis de l'après-midi, moins cher, était considéré comme remplacé.
2. **Une redirection ouverte.** La fonction qui protège le « retour à cette page après la connexion » acceptait `/\t/evil.example`. Les navigateurs suppriment les tabulations et les sauts de ligne, si bien que cette adresse menait vers un autre site.
3. **Des noms hérités.** Un parseur acceptait des noms dont hérite tout objet JavaScript, comme `toString`.
4. **Des jours qui n'existent pas.** Le contrôle de date acceptait `2026-02-30`.



Contrôles automatisés dans Hearth : tests unitaires, contrôles d'isolation, tests de l'extension et tests Swift

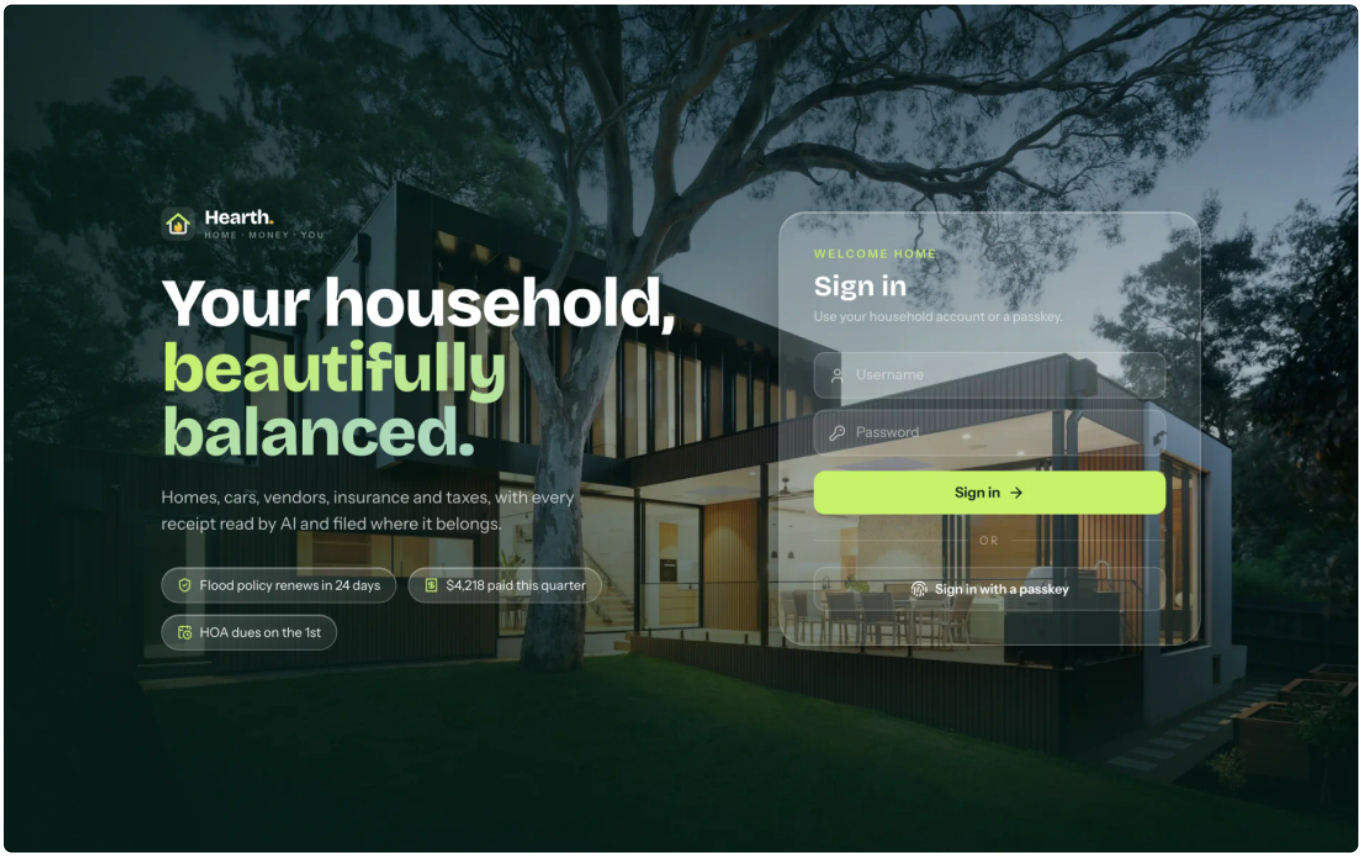
Le même jour, les tests sont devenus la loi. Le script de déploiement les exécute avant de construire quoi que ce soit et refuse de livrer si l'un d'eux échoue, peu importe qui déploie. Pour prouver que la barrière fonctionne vraiment, nous avons cassé exprès la fonction `relativeDays` et regardé le déploiement s'arrêter sur « Unit tests failed: not deploying. » Puis nous l'avons réparée. Les instructions de mise en production interdisent aussi de sauter, de supprimer ou d'assouplir un test pour faire passer une version -- une règle écrite autant pour l'IA que pour moi.

> **Là où les tests s'arrêtent**
> Les tests unitaires couvrent la logique, pas les écrans. Il n'y a pas encore de tests de bout en bout dans le navigateur, et les applis Swift n'ont qu'une poignée de tests dans leur package commun. Chaque

> modification d'écran a été vérifiée en lançant l'appli -- dans le navigateur, dans les simulateurs, sur le vrai iPad -- et les pull requests le mentionnent, mais c'est une vérification manuelle, pas de l'automatisation. Pour un produit avec de nombreux clients au lieu d'un seul foyer, les tests de bout en bout seraient la prochaine chose que j'ajouterais.

Garde-fou 3 : une connexion dans les règles de l'art

La connexion est un bon endroit pour distinguer un travail professionnel d'une démo, parce qu'une démo s'arrête à « ça me laisse entrer ». Hearth propose des mots de passe et des passkeys, le standard de connexion résistant au phishing de la FIDO Alliance, bâti sur WebAuthn du W3C (implémenté avec SimpleWebAuthn). Les mots de passe sont stockés avec bcrypt ; un nom d'utilisateur inconnu est vérifié contre un hash factice de même coût, si bien que la réponse prend exactement autant de temps que pour un vrai compte et ne révèle pas quels comptes existent.

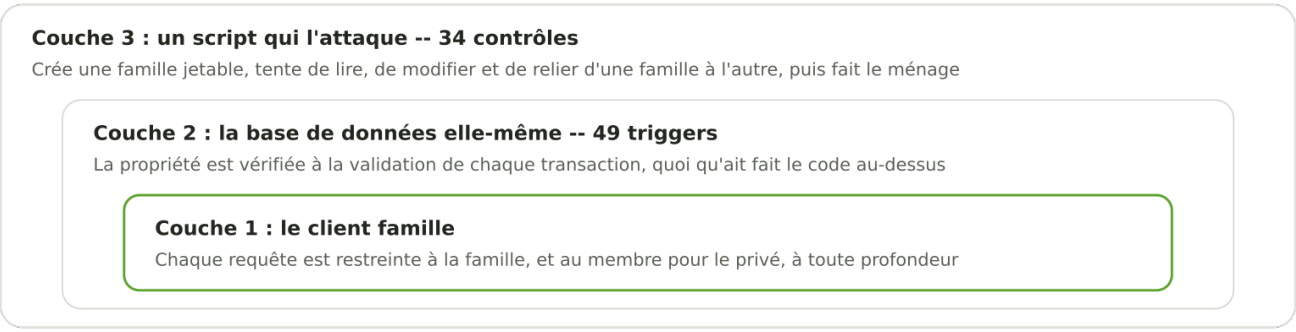


La page de connexion : un mot de passe ou une passkey, avec une limitation des tentatives derrière les deux (instance de démonstration).

C'est dans les détails que se trouve le travail. Les tentatives échouées sont limitées par nom d'utilisateur et par adresse, et les compteurs vivent dans la base de données, si bien qu'un redémarrage ne les remet pas à zéro. Les jetons de session sont aléatoires et stockés uniquement sous forme de HMAC ; une copie de la base de données ne connecte donc personne, et le cookie est `httpOnly`, `secure` et `sameSite`. Chaque route handler vérifie l'origine de la requête. Changer son mot de passe déconnecte ses autres sessions et toutes les applis connectées, l'ajout d'une passkey demande d'abord le mot de passe, et la réinitialisation d'un mot de passe par un administrateur supprime aussi les passkeys. Enfin, chaque server action et chaque route d'API vérifie l'utilisateur elle-même au lieu de supposer qu'un middleware l'a déjà fait.

Garde-fou 4 : une famille ne peut jamais voir les données d'une autre

Hearth est multi-tenant : plusieurs familles et, au sein d'une famille, des données partagées et des données privées propres à un membre. Une erreur ici n'est pas un bug ; c'est un incident de confidentialité. L'isolation est donc assurée trois fois, à trois niveaux différents.



Trois couches indépendantes tiennent les données d'une famille à l'écart des autres

La couche intérieure est un client de base de données qui ajoute automatiquement les règles de famille et de membre à chaque requête, y compris aux relations imbriquées et aux comptages, si bien qu'un développeur -- ou une IA -- ne peut pas les oublier. La couche du milieu ne fait pas confiance à la couche intérieure : quarante-neuf triggers dans PostgreSQL vérifient la propriété au moment où une transaction est validée. La

couche extérieure ne fait confiance à aucune des deux : un script crée une famille jetable, tente 34 façons de lire, de modifier ou de relier des données par-delà la frontière, puis la supprime.

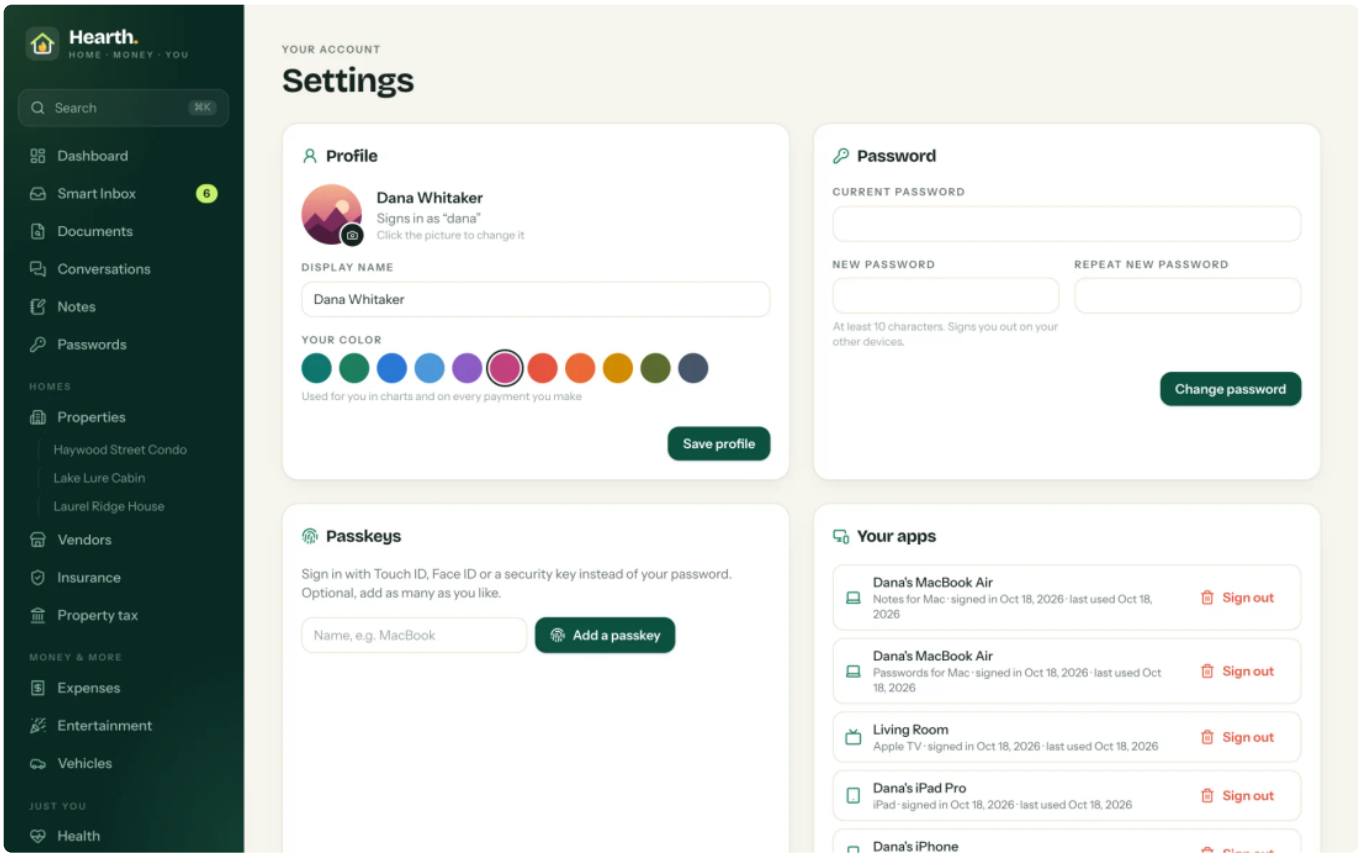
À mon avis, c'est la différence la plus importante entre un prototype et un produit. N'importe qui peut écrire `where familyId = ?`. La question est de savoir ce qui se passe le jour où quelqu'un oublie.

Garde-fou 5 : les secrets restent secrets

Le coffre-fort de mots de passe chiffre chaque champ d'une entrée -- titre, adresse, nom d'utilisateur, mot de passe et notes -- avec AES-256-GCM, à l'aide d'une clé dérivée d'un secret du serveur. Chaque valeur est liée

à sa famille, à son entrée et à son champ en tant que données authentifiées additionnelles. Copiez un mot de passe chiffré d'une ligne vers une autre, et il ne s'ouvre plus. Les anciens mots de passe sont conservés dans un historique, chiffrés de la même manière, si bien qu'un ancien mot de passe n'est jamais perdu.

Les applis suivent la même logique du moindre privilège. Chacune se connecte séparément avec un code confirmé sur le web, et son jeton est stocké uniquement sous forme de HMAC. Chaque type d'appli ne peut faire que ce pour quoi il a été conçu : l'appli Apple TV et les applis iPhone et iPad ne font que lire et ne modifient jamais rien, Notes for Mac ne voit que les notes, et l'extension Chrome ne voit que les mots de passe. Dans l'extension, seul le worker d'arrière-plan détient le jeton, et il ne répond à une page web qu'à propos du site de cette page, après un vrai clic.



Réglages ' Vos applis : chaque appli connectée avec son type, la date de sa connexion et celle de sa dernière utilisation -- chacune peut être déconnectée séparément.

Garde-fou 6 : une revue qui a trouvé les erreurs de l'IA

Le 11 octobre, j'ai demandé à l'IA de passer en revue sa propre base de code sous l'angle des autorisations, de la confidentialité entre membres et de la sécurité -- non pas pour la forme, mais avec la mission explicite de trouver des problèmes. Elle en a trouvé de vrais, et c'est tout l'intérêt de l'exercice. Parmi eux, une faille de **cross-site scripting stocké** : un fichier téléversé pouvait déclarer son propre type de contenu, et un fichier forgé aurait pu exécuter du script dans l'appli. Elle a aussi trouvé des endroits où la modification d'une donnée privée pouvait la rendre visible au reste de la famille, ainsi qu'un rapport d'utilisation de l'IA qui montrait les titres des documents à l'administrateur.

Tout cela a été corrigé dans la même pull request, avec la limitation des tentatives dans la base de données, la déconnexion des autres sessions lors d'un changement de mot de passe et les sauvegardes nocturnes. Je veux être honnête sur ce que cela signifie. L'IA a écrit ces bugs, et l'IA les a trouvés -- parce que j'ai demandé une revue et que je l'ai traitée comme une barrière, pas comme un rituel. Ce n'est pas différent de la façon dont une bonne équipe travaille avec des développeurs juniors. Personne ne livre sans un deuxième regard.

Garde-fou 7 : mises en production et exploitation

Hearth tourne sur un Raspberry Pi 5 à la maison, dans un conteneur Docker derrière Caddy, qui s'occupe des certificats HTTPS. Chaque mise en production suit le même pipeline, et ce pipeline est consigné sous forme de skill que l'IA exécute étape par étape ; une mise en production un soir de fatigue ressemble donc exactement à celle d'un matin frais et dispos.

De la pull request à la production : chaque version suit le même chemin

Les cinq dernières images de version restent sur le Pi ; un retour en arrière tient donc en une ligne dans le fichier compose. La base de données est sauvegardée chaque nuit dans un bucket de stockage séparé, avec des copies quotidiennes et mensuelles, et une sauvegarde a été restaurée dans une base de travail et comparée à la base de production avant que je lui fasse confiance. J'ai appris cette leçon à mes dépens, comme je l'ai raconté dans My Server Was Compromised : l'architecture détermine le rayon d'impact bien avant que quoi que ce soit ne tourne mal.

Ce qu'a fait l'IA, et ce que j'ai fait

La répartition des tâches est ce que les gens ont le plus de mal à croire, alors soyons concrets. En douze jours, j'ai écrit environ 110 messages ; l'IA a effectué en réponse quelque 3 900 appels d'outils -- lire des fichiers, écrire du code, lancer des tests, compiler, déployer. Elle a écrit presque tout le code, les tests et les descriptions des pull requests.

Ce qu'elle n'a pas fait, c'est décider. J'ai choisi l'architecture et la stack : Next.js 16 avec React 19, Prisma 7 sur PostgreSQL, TypeScript 7 avec son nouveau compilateur natif, SwiftUI pour les applis Apple, et une extension Chrome Manifest V3. J'ai décidé que l'isolation serait garantie dans la base de données et pas seulement dans le code, que les mots de passe seraient liés à leur ligne, que chaque appli n'aurait accès qu'à sa propre partie, et que les tests conditionneraient chaque mise en production. J'ai testé sur de vrais appareils, j'ai consigné les règles que l'appli devait suivre (« les dates d'abord, les plus récentes en haut », « pas de listes déroulantes simples, toujours avec recherche »), et j'ai dit non -- souvent. Quand l'écran de connexion de l'iPad ressemblait à l'écran de code de l'Apple TV, c'est une personne qui l'a remarqué.

> **Ce que disent les études : tout dépend du développeur**
> Je ne veux pas prétendre que les preuves vont toutes dans le même sens. Dans un essai contrôlé randomisé de METR, seize développeurs open source expérimentés ont été 19 % plus lents avec les outils d'IA du début 2025 sur leurs propres grandes bases de code, tout en se croyant plus rapides. METR énumère lui-même les limites -- un petit échantillon, un seul type de travail, des outils qui ont changé depuis -- et a publié des données plus récentes. La leçon que j'en tire n'est pas que l'IA est lente ou rapide. C'est que le gain dépend de la personne qui la pilote : de la précision de ses spécifications, de la rigueur de ses relectures, et du fait qu'elle traite ou non les tests et les revues comme des barrières. Sans cela, vous obtenez des vibes. Avec, vous obtenez Hearth.

Ce que cela prouve

Rapide et rigoureux ne sont plus des contraires. La même IA qui a écrit 58 588 lignes en douze jours a aussi écrit les 313 tests, les 34 contrôles d'isolation, les 49 triggers et la revue de sécurité qui ont gardé ces lignes honnêtes -- parce qu'on le lui a demandé, et parce que rien n'a atteint la production sans les passer. La vitesse

est venue des outils. La rigueur est venue de trente ans passés à apprendre ce qui tourne mal en production, et du refus de sauter les parties ennuyeuses.

> L'IA n'a pas supprimé l'ingénierie. Elle a supprimé l'excuse pour s'en passer.

Si vous voulez ce genre de vitesse pour votre propre produit -- une application Delphi qui a besoin d'un avenir moderne, ou un système qui s'adapte à votre entreprise et non l'inverse --, jetez un œil à ce que j'ai construit,

à ce que je propose, puis parlons-en.

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 —