NEWS · JULY 2, 2026

Vier Stunden für die Migration einer sechs Jahre alten Delphi-Multi-Tier-Anwendung

By Dr. Holger Flick



Die Behauptung, präzise formuliert

Ich möchte die Behauptung im Titel sehr genau fassen, denn sie klingt nach der Art von Aussage, mit der man einen Kurs verkaufen will. Eine Produktivanwendung — ein Delphi-**TMS-WEB-Core**-Frontend, ein **TMS-XData**-REST-Backend und eine **Firebird**-Datenbank mit Zehntausenden Zeilen echter Benutzerhistorie — wurde auf einem komplett modernen Stack neu aufgebaut: **Next.js 16, React 19, Prisma 7, PostgreSQL**. Neues Schema. Neue Authentifizierung. Neues Backend. Jedes Chart neu erstellt. Dark Mode. Ein Live-Datenmigrationstool mit Fortschritts-UI. VPS-Deployment-Skripte. Dokumentation. Von Anfang bis Ende dauerte das **etwa vier Stunden.** Dies von Hand zu erledigen — sorgfältig, so wie man es bei echten Benutzerdaten tun müsste — ist eine Aufgabe von **mehreren Wochen**. Ich habe solche Migrationen schon auf die langsame Art gemacht. Das hier war etwas anderes.

Wenn Sie nur eine Sache aus diesem Beitrag mitnehmen, dann diese: **Die vier Stunden sind nicht die Geschichte. Die sechs Monate vor den vier Stunden sind die Geschichte.** Die Geschwindigkeit am Ende ist die *Folge* einer Fähigkeit, die ich gezielt aufgebaut habe — der Fähigkeit, KI als echtes Engineering-Werkzeug zu führen statt als Spielerei. Dieser Beitrag ist der detaillierte Bericht darüber, wie das konkret aussieht, wenn es funktioniert.

Ich gehe jede Phase durch. Ich zeige Ihnen die Architektur, die Entscheidungen, die Diagramme. Und — weil ein Erfahrungsbericht, der nur die Highlights zeigt, wertlos ist — zeige ich Ihnen auch die Stellen, an denen die KI etwas *falsch* oder *hässlich* gemacht hat, ich es bemerkt habe und wir den Kurs korrigiert haben. Diese Schleife aus Erkennen und Korrigieren ist heute der eigentliche Job.

Los geht's.

Die Ausgangslage (oder: worüber ich bewusst vage bleibe)

Hier die ehrliche Einordnung.

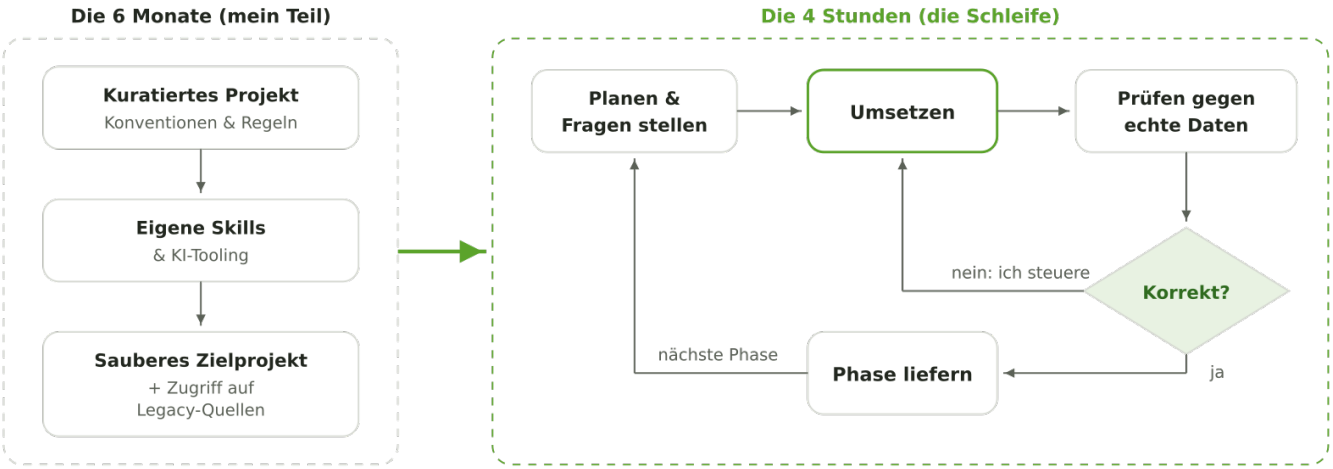
Ich habe mich **nicht** hingesetzt, „migriere meine Delphi-App" getippt und bin weggegangen. Was ich tatsächlich getan habe, bevor auch nur eine Zeile neuen Codes geschrieben wurde, ist der Teil, in dem ich ein halbes Jahr lang besser geworden bin — und der Teil, den ich ein wenig für mich behalte, denn er ist der Unterschied zwischen einer Demo und einer Lieferung.

Was ich Ihnen gerne verrate:

- Ich habe ein **sauberes, leeres Next.js-Projekt** als Ziel vorbereitet. Eine leere Leinwand, konfiguriert auf die Art, von der ich weiß, dass sie gute Ergebnisse liefert.
- Ich habe der KI **kontrollierten Zugriff auf die alten Quellen** gegeben — die Delphi-Units, die XData-Service-Schicht, das Firebird-Schema — als reines Referenzmaterial.
- Ich habe einen sorgfältig abgestimmten Satz an **Projektanweisungen, Hausregeln, Skills und KI-Tooling** bereitgestellt, den ich über Monate verfeinert habe. Darin ist kodiert, wie ich Code geschrieben haben will: Konventionen, Framework-Versionen, Designregeln, das ganze Programm. Das ist meine „Magie", und sie leistet im Hintergrund von allem, was folgt, eine *Menge* stiller Arbeit.

Diese Vorbereitung ist der Hebel. Das Modell ist stark, aber ein starkes Modell, das auf ein vages Ziel gerichtet wird, produziert plausiblen Müll. Ein starkes Modell, das auf ein **scharfes Ziel mit scharfen Leitplanken** gerichtet und **von jemandem beaufsichtigt wird, der die Ausgabe lesen kann**, produziert Produktionscode.

Die Lücke zwischen diesen beiden Ergebnissen ist Expertise, und die gibt es nicht geschenkt.



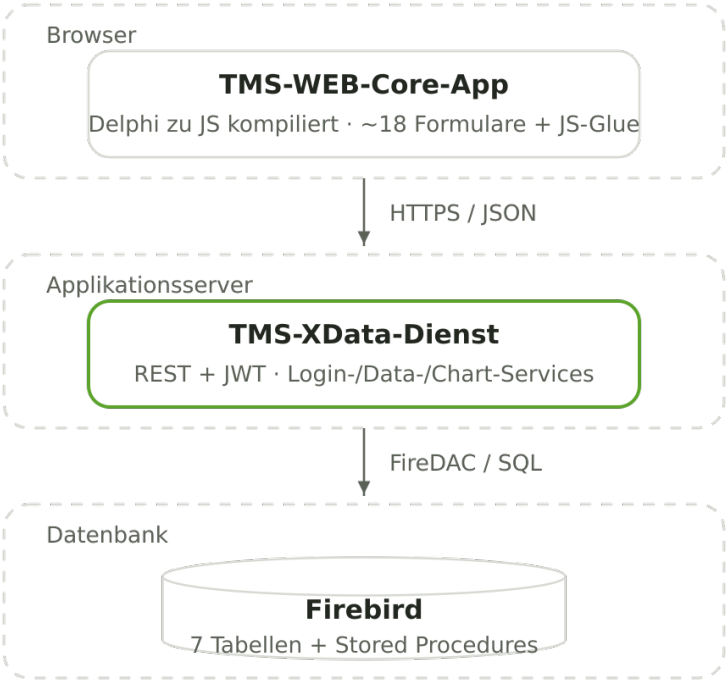
Sechs Monate Vorbereitung speisen eine schnelle Planen–Umsetzen–Prüfen-Schleife

Alles ab hier sind die vier Stunden.

Das Legacy-System, das wir abgelöst haben

Seien wir fair zur alten App: Sie funktionierte, und das seit Jahren. Es handelt sich um eine Anwendung zur Ernährungs- und Körperdatenerfassung — Sie protokollieren, was Sie über den Tag essen, erfassen Gewicht, Schritte und Schlaf, und die App stellt Ihren Fortschritt über die Zeit in Charts dar. Echte Menschen nutzten sie und fütterten sie seit 2023 mit Daten.

Architektonisch war es ein sehr typischer Drei-Schichten-Aufbau aus der Delphi-Ära:



Die Legacy-Drei-Schichten-Architektur: WEB-Core-Client, XData-REST-Server, Firebird-Datenbank

Die Bestandteile:

- **Frontend:** TMS WEB Core — Object Pascal, kompiliert nach JavaScript — mit rund achtzehn Formularen (Login, Hauptmenü, Tagesdaten-Grid, Editoren pro Mahlzeit, Lebensmittelliste, Nährwertansicht, Charts) und einer ordentlichen Portion handgeschriebenem JavaScript-Glue-Code.
- **Backend:** Ein TMS-XData-Server mit typisierten REST-Services — ein `LoginService`, ein `DataService` und ein `ChartService` — mit JWT-Authentifizierung und FireDAC als Datenbankzugriff.
- **Datenbank:** Firebird, mit sieben Tabellen und einer Handvoll Stored Procedures für die Tagesaggregation (gegessene Kalorien, tägliche Nährwertsummen, Gewichts-/Schritte-Rollups).

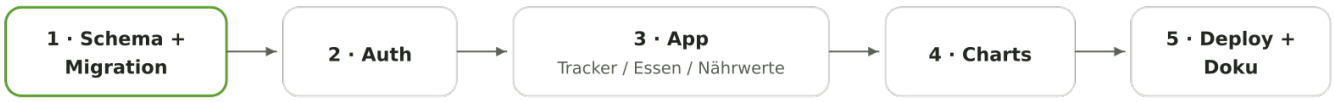
Daran war nichts falsch. Aber es ist ein Stack mit schrumpfendem Talentpool, einer Desktop-first-Deployment-Story und einer UI, der man ihr Alter ansieht. Der Kunde wollte die Anwendung auf etwas, das ein moderner Webentwickler übernehmen, erweitern und überall deployen kann. Fair.

Meine Aufgabe: **die App originalgetreu auf einem Stack von 2026 neu bauen** — ohne eine einzige Zeile dieser drei Jahre Datenhistorie zu verlieren.

Der Gesamtplan

Bevor Code angefasst wurde, einigten die KI und ich uns auf ein phasenweises Vorgehen. Das ist die erste Stelle, an der Aufsicht zählt: Ich wollte keinen Feuerwehrschlauch voller Dateien, ich wollte einen **Plan, den ich abnehmen kann.** Wir einigten uns auf:

1. **Fundament** — Prisma-Schema, PostgreSQL-Datenbank und ein echtes Datenmigrationstool, um die Firebird-Historie herüberzuholen.
2. **Authentifizierung** — denn Sie können nichts als echter Benutzer testen, solange Sie sich nicht als einer anmelden können.
3. **Die Anwendung** — Tages-Tracker, Lebensmittelliste, Nährwertansicht.
4. **Die Charts** — explizit als First-Class-Feature ausgewiesen, denn die Charts waren das Herz der alten App.
5. **Deployment & Doku** — die App mit wiederholbaren Skripten auf einen VPS bringen und alles dokumentieren.



Die fünf vereinbarten Phasen, in Reihenfolge

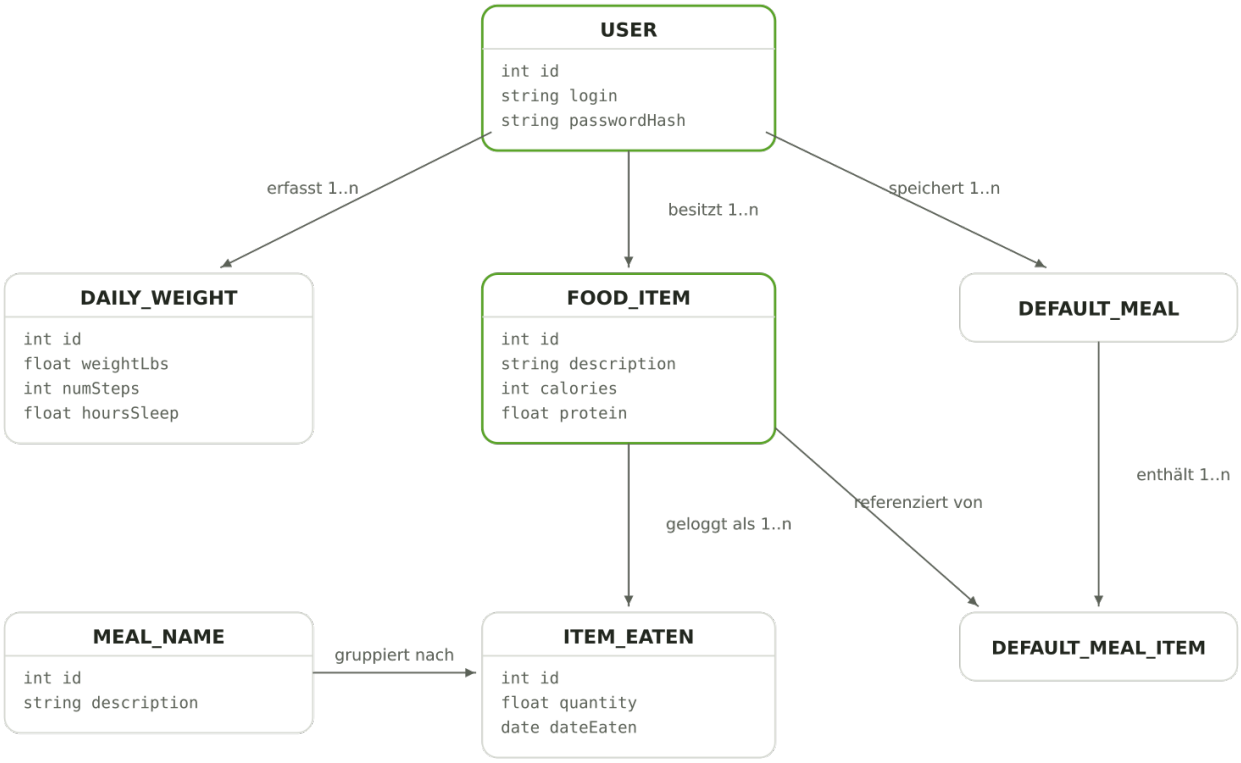
Entscheidend: Zu Beginn jeder Phase hat die KI **innegehalten und mir Fragen gestellt** — echte, entscheidungsförmige Fragen, kein „soll ich weitermachen?"-Füllmaterial. Welche Hashing-Strategie für die Legacy-Passwörter? Wo soll der Migrationsstatus leben? Wie soll eine Daten-Eigenheit behandelt werden? In diesen Fragen wird meine Erfahrung in die Ausgabe der Maschine injiziert. Ich werde sie unterwegs markieren.

Phase 1 — Das Schema und die große Datenmigration

Die alte Welt lesen

Das Erste, was die KI tat, war *lesen*. Nicht nur das `CREATE`-Skript von Firebird, sondern auch die Pascal-Implementierungen der XData-Services — denn das Schema allein verrät die Geschäftslogik nicht. Die Stored Procedures, das SQL in den Delphi-Services, die Art, wie „Kalorien eines Tages" tatsächlich berechnet wurde — all das lebt im Backend-Code, und all das musste verstanden sein, bevor ein einziges Prisma-Modell geschrieben wurde.

Sieben Tabellen kamen konzeptionell herüber:



Die sieben Tabellen und ihre Beziehungen, wie sie ins neue Schema übernommen wurden

Korrektur Nr. 1: „nicht einfach die Spalten in Großbuchstaben übernehmen"

Hier bin ich zum ersten Mal eingeschritten. Das Legacy-Schema folgte Firebirds Schrei-Konvention — `PERMISSION_STRING`, `DATE_EATEN`, `NUM_STEPS`, `DESCR`. Der erste Entwurf des Prisma-Schemas übernahm diese Namen brav eins zu eins.

Ich habe gestoppt: **Verwende idiomatische TypeScript-Namen.** `DESCR` ’`description`. `LBS` ’`weightLbs-`

. `NUM_STEPS` ’`numSteps`. Das Schema soll sich lesen wie moderner Code, nicht wie ein Datenbank-Dump von 2010. Die KI hat das gesamte Schema mit sauberen camelCase-Feldnamen und ordentlichen Relationen neu generiert. Eine Kleinigkeit — aber es ist der Unterschied zwischen einer Codebasis, an der ein neuer Entwickler Freude hat, und einer, die er nur erträgt.

Das Passwort-Problem (und eine Entscheidung, die nur ein Mensch treffen sollte)

Dann kam eine wirklich interessante Frage. Das alte XData-Backend verifizierte Passwörter mit Firebirds eingebauter Hash-Funktion:

```
CRYPT_HASH(password USING SHA512)
```

Konnten wir das in Node reproduzieren — also **null Passwort-Resets für Bestandsnutzer** — oder mussten wir die Passwörter löschen und alle zum Zurücksetzen zwingen?

Die KI hat nicht geraten. Sie hat sich **mit der Live-Firebird-Datenbank verbunden, den gespeicherten Passwort-Hash eines echten Benutzers gezogen und den Algorithmus bewiesen**, indem sie ein bekanntes Testpasswort in Node hashte und Byte für Byte verglich:

```
import { createHash } from "node:crypto";

// SHA-512, hex, Großbuchstaben – exakt das, was Firebirds CRYPT_HASH liefert.
const candidate = createHash("sha512").update(password).digest("hex").toUpperCase();
const matches = candidate === storedHashFromFirebird;
```

Bestätigt: ein schlichtes, ungesalzenes SHA-512. Vollständig reproduzierbar.

Das verwandelte eine beängstigende Frage in eine saubere Entscheidung, die sie mir dann vorlegte:

> **Der Vorschlag, den ich freigegeben habe**
> Beim Login gegen den Legacy-SHA-512-Hash verifizieren und **jeden Benutzer bei der ersten erfolgreichen Anmeldung transparent auf modernes bcrypt upgraden.** Keine Resets. Keine Störung. Das schwache alte Hashing schafft sich selbst ab, ein Login nach dem anderen.

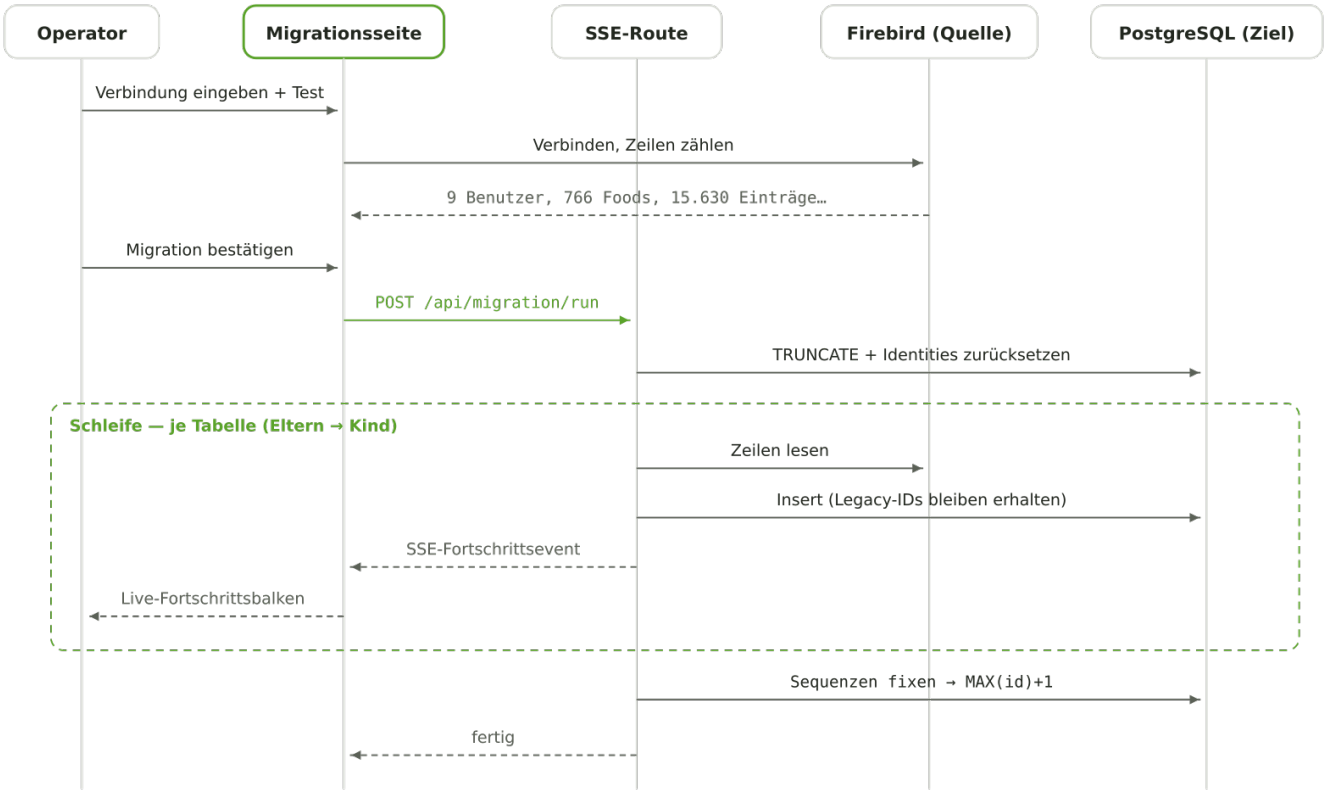
Ich habe zugestimmt. Genau dieses Muster will ich in einer Migration: Niemand bekommt eine E-Mail mit „bitte setzen Sie Ihr Passwort zurück, weil wir das Backend gewechselt haben". Die Nutzer melden sich einfach an, und hinter den Kulissen wird ihre Sicherheit still und leise besser. Die KI hat es vorgeschlagen; mein Job war, es als die richtige Entscheidung zu erkennen und freizugeben.

Das Migrationstool selbst

Auf diesen Teil bin ich am stolzesten, denn hier wird aus „die KI hat Code geschrieben" ein „die KI hat ein Werkzeug gebaut".

Statt eines Wegwerf-Skripts haben wir eine richtige **Migrationsseite** gebaut — außerhalb der Authentifizierung (das muss sie sein; sie *erzeugt* ja erst die ersten Benutzer) — mit:

- Einem Formular für die Verbindungsdaten der Firebird-Quelle, mit einem **„Test Connection"**-Button, der vor dem Start live die Zeilenanzahlen meldet.
- Einer Verbindungs-**Historie**, damit Sie Zugangsdaten nicht neu eintippen müssen.
- Einem strengen Bestätigungsdialog, denn der Vorgang leert das Ziel und lädt es neu.
- **Server-Sent Events, die den Fortschritt live streamen** — ein Balken pro Tabelle — während Tausende Zeilen hinüberwandern.



Der Migrationslauf: testen, bestätigen, dann Fortschritt pro Tabelle per SSE streamen

Unter der Haube arbeiten sieben geordnete Migratoren (Eltern vor Kindern, damit Fremdschlüssel nie brechen), von denen jeder seine eigene Bilanz meldet:

```
type MigrationResult = { migrated: number; skipped: number };
```

Das Ganze lief gegen die Live-Datenbank und bewegte **9 Benutzer, 766 Lebensmittel, 15.630 protokollierte Mahlzeiten und 3.409 Gewichtseinträge — ohne einen einzigen übersprungenen Datensatz.**

Korrektur Nr. 2: das Fließkomma-Rauschen

Nach der Migration schaute ich mir die laufende App an, und etwas störte mich: Eine Menge, die `1.1` lauten sollte, wurde als `1.100000023841858` angezeigt. Ein Gewicht zeigte `252.60000610351562`. Hässlich.

Ich habe es gemeldet — *„die Präzision wirkt falsch — Rundung oder Speicherung?"* — und die KI diagnostizierte es korrekt: Die alten Firebird-Spalten waren 32-Bit-Floats; in PostgreSQLs 64-Bit-Doubles befördert, wird das Single-Precision-Rauschen sichtbar. Sie bot drei Lösungen an, und ich wählte die sauberste: **die Werte zum Migrationszeitpunkt runden**, sodass die gespeicherten Daten selbst sauber sind und der Fix bei jedem künftigen Neu-Lauf reproduzierbar ist.

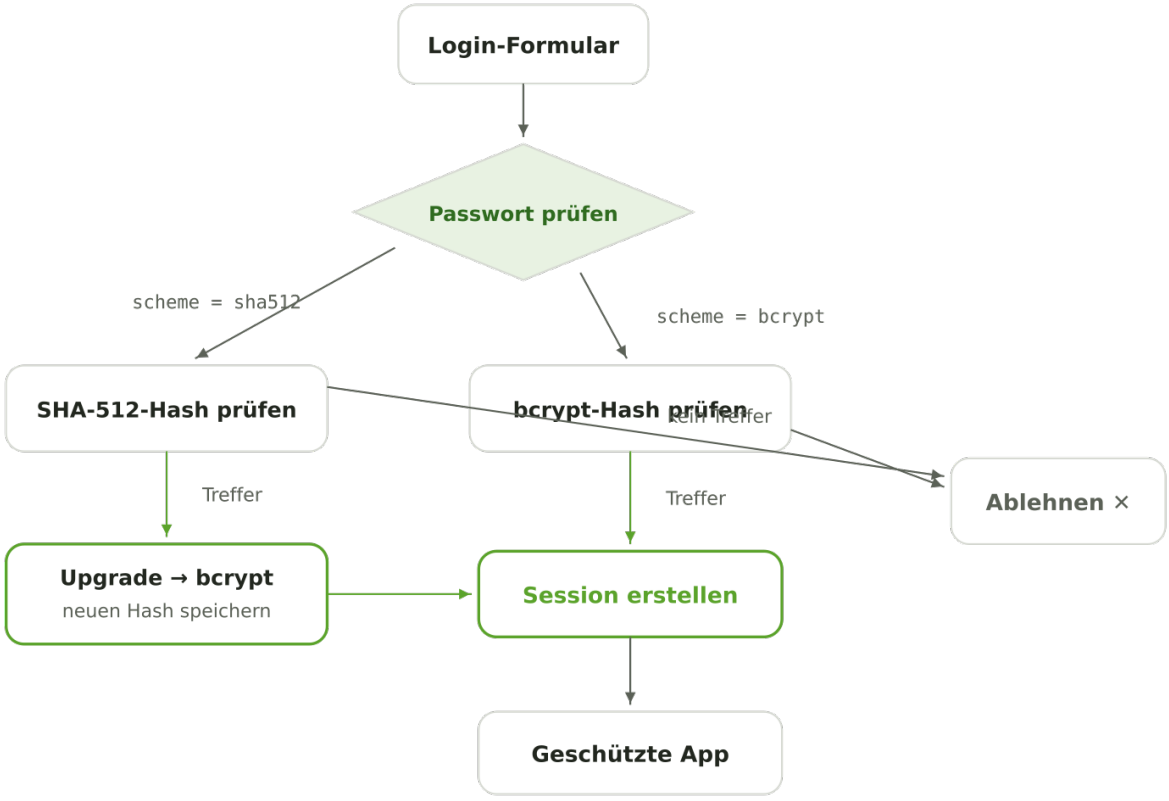
Wo wir schon dabei waren, wies ich auf ein zweites, subtileres Problem hin: Die Kaloriensummen pro Mahlzeit stimmten nicht immer mit der Summe der sichtbaren Einzelposten überein (die rohe Summe runden vs. die gerundeten Posten summieren — um eins daneben, und es *sieht* falsch aus, selbst wenn es mathematisch in Ordnung ist). Wir haben die Summen so korrigiert, dass sie mit dem übereinstimmen, was das Auge

zusammenzählt. Das ist die Art von Bug, an der ein automatisiertes Werkzeug glatt vorbeisegelt und die ein Mensch, der *das Produkt benutzt*, sofort erwischt.

> **Fundament fertig**
> Schema, Datenbank und ein echtes Migrationstool — mit Daten, die gegen die Ground Truth verifiziert wurden.

Phase 2 — Authentifizierung

Kurze Phase, wichtige Phase. Mit `iron-session` für verschlüsselte Cookie-Sessions bauten wir den Login-Flow rund um die SHA-512-verifizieren-dann-auf-bcrypt-upgraden-Strategie aus Phase 1, ein Protected-Route-Layout, das nicht angemeldete Besucher umleitet, und eine saubere Anmeldeseite.



Login-Flow: Legacy-SHA-512 oder bcrypt prüfen, alte Hashes bei Erfolg still upgraden

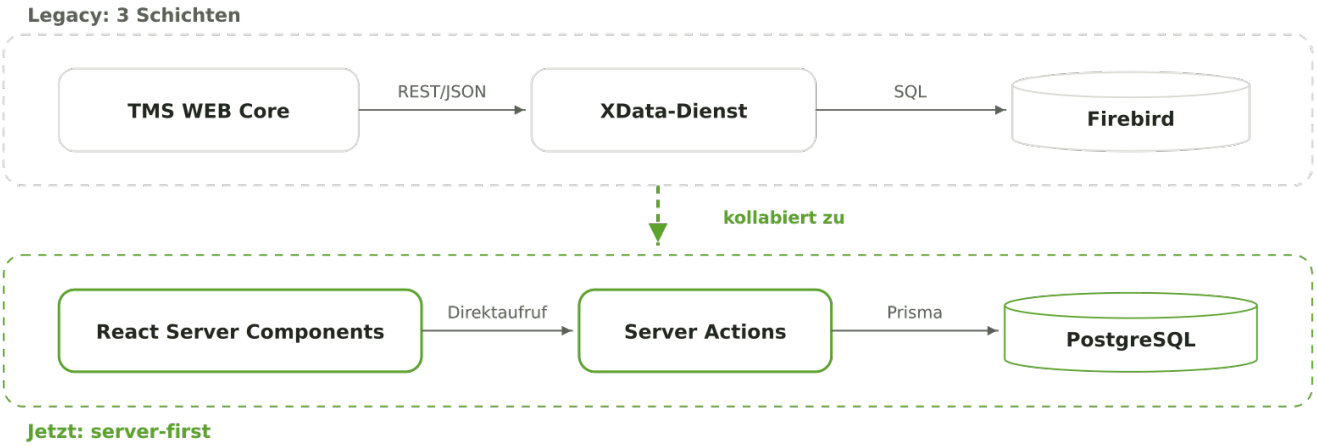
Die Verifikation, auf die es hier ankam, war nicht visuell — es ging darum zu beweisen, dass der Login gegen ein migriertes Konto *tatsächlich funktionierte*. Die KI versiegelte eine echte Session, meldete sich als echter

migrierter Benutzer an, bestätigte, dass das bcrypt-Upgrade beim ersten Login griff, und bestätigte, dass ein falsches Passwort abgelehnt wurde. Anschließend setzte sie das Konto in seinen migrierten Ausgangszustand zurück, sodass nichts verändert blieb.

Phase 3 — Die Anwendung neu bauen

Hier wurde die alte Drei-Schichten-Architektur zu etwas dramatisch Einfacherem zusammengefaltet. Kein separates REST-Backend. Keine API-Schicht, die entworfen, versioniert und abgesichert werden muss. **Next.js**

Server Actions lassen die UI serverseitige Funktionen direkt aufrufen — Datenbankzugriff, Ownership-Checks und Validierung leben komplett auf dem Server.



Drei Legacy-Schichten kollabieren zu einer server-first Next.js-Architektur

Der Tages-Tracker

Das Herzstück. Eine Tagesansicht mit Datumsnavigation, einer Karte pro Mahlzeit (Frühstück, Mittagessen, Abendessen, die Snacks), den jeweils protokollierten Einträgen mit Live-Kalorienrechnung und einem Panel für Gewicht/Schritte/Schlaf/Körperzusammensetzung.

Hier traf ich eine explizite Designentscheidung, die die KI dann umsetzte: **das Interaktionsmodell modernisieren.** Die alte App war Seite-pro-Formular — man klickte sich zu einem separaten Screen, um eine Mahlzeit zu bearbeiten, zu einem weiteren fürs Gewicht. Wir haben all das durch **modale Dialoge und Inline-Editing** aus einer einzigen Tagesansicht ersetzt. Einen Eintrag über einen durchsuchbaren, servergestützten Lebensmittel-Picker mit Live-Kalorienvorschau hinzufügen. Eine Menge direkt an Ort und Stelle ändern. Das Gewicht anpassen, ohne die Seite zu verlassen. Dieselben Daten, ein Gefühl von 2026.

Jede Mutation trägt einen serverseitigen **Ownership-Guard** — ein Benutzer kann immer nur seine eigenen Daten anfassen — und ersetzt damit die JWT-Claim-Prüfungen der alten XData-Services.

Die Lebensmittelliste

Volles CRUD über Lebensmittel mit ihren kompletten Nährwertangaben, als responsive Tabelle auf dem Desktop und als Karten auf dem Smartphone (mobile Gleichwertigkeit war eine harte Regel — keine versteckte Funktionalität auf kleinen Bildschirmen). Durchsuchbar, paginiert. Und ein wirklich durchdachter **Lösch-Guard**:

Ein Lebensmittel, das von protokollierten Mahlzeiten referenziert wird, lässt sich nicht löschen — die App sagt Ihnen, dass es in *N* Mahlzeiten verwendet wird, statt kryptisch zu scheitern oder drei Jahre Historie zu verwaisen.

Die Nährwertansicht

Eine Makro-Aufschlüsselung pro Tag — Protein, Kohlenhydrate, Ballaststoffe, Netto-Kohlenhydrate, Zucker, zugesetzter Zucker, Fett, gesättigtes Fett, Cholesterin, Natrium, Wasser — über einen wählbaren Datumsbereich, mit Tagesdurchschnitten, die die alte Stored Procedure `GET_DAILY_NUTRITION_STATS` originalgetreu als saubere SQL-Aggregation reproduziert.

Während dieser gesamten Phase lief die Verifikation kontinuierlich: Die KI meldete sich als echter migrierter Benutzer an und bestätigte, dass die gerenderte Tagessumme mit dem eigenen Aggregat der Datenbank **auf die Kalorie genau** übereinstimmte:

```
SELECT SUM(quantity * calories) AS total_calories
FROM item_eaten
JOIN food_item ON food_item.id = item_eaten.food_item_id
WHERE item_eaten.user_id = $1 AND item_eaten.date_eaten = $2;
```

Phase 4 — Die Charts

Die Charts waren nicht verhandelbar. In der alten App waren sie die Belohnung — der sichtbare Beweis für Monate der Anstrengung. Die Daten des Kunden erzählen eine wirklich großartige Geschichte (die mehrjährige Gewichtsreise eines Benutzers), und die musste gut erzählt werden.

Wir blieben bewusst bei **Chart.js**, weil es sich sauber nach React portieren lässt und echte Kontrolle bietet. Die Datenschicht reproduzierte die Legacy-Logik exakt, inklusive eines Details, auf dessen originalgetreue Umsetzung ich bestand: Die „14-Tage"- und „50-Tage"-gleitenden Durchschnitte der alten App waren in Wahrheit **zeilenbasierte Fenster** über aufeinanderfolgende Wiegungen, keine Kalendertage. Wir haben das nachgebildet, damit wiederkehrende Benutzer dieselben Kurven sehen wie immer. Gewichts-Chart, Kalorien-Chart und die Abnehm-Statistikkarten (Start- vs. aktuelles Gewicht, Gesamtveränderung, Rate pro Woche/Monat).

Korrektur Nr. 3 und Nr. 4: die Charts, die sich wehrten

Die Charts kamen beim ersten Mal nicht richtig heraus. Das ist es wert, gezeigt zu werden, denn es ist die ehrlichste Illustration der Schleife.

> **Bug Nr. 1 — die leeren Charts**
> Die Charts wurden *leer* gerendert — die X-Achse zeigte „12AM, 1AM, 2AM…" statt Daten von 2023 bis 2026. Ich schickte einen Screenshot: *„Charts sind leer?!"* Die KI diagnostizierte es sofort: Sie fütterte Datums-**Strings** an eine Zeitachse, die bei deaktiviertem Parsing numerische Timestamps erwartete. Alle 1.173 Punkte waren auf einen einzigen Tag kollabiert.

Der Fix bestand darin, der Achse echte Timestamps statt Strings zu geben:

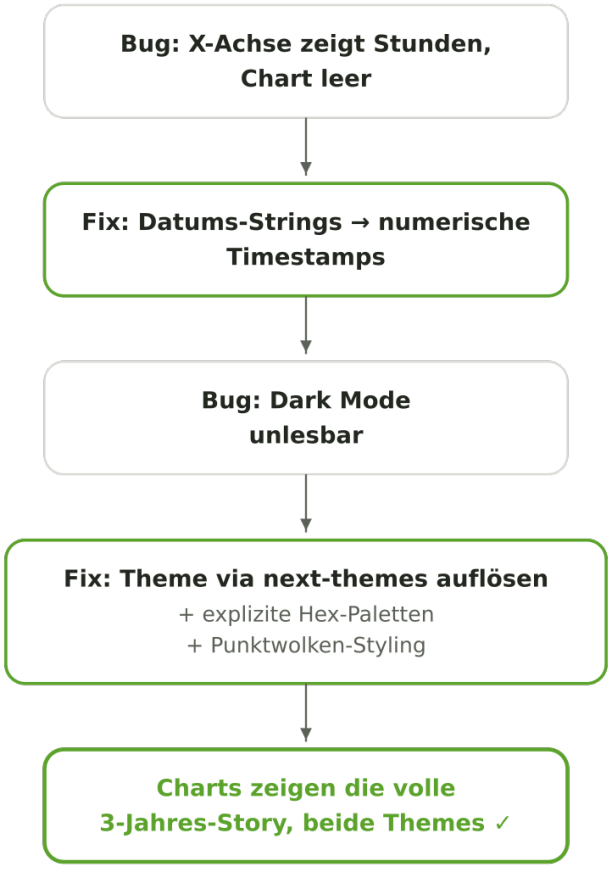
```
// Vorher: Datums-Strings kollabieren bei deaktiviertem Parsing auf einen einzigen Punkt.
const data = rows.map((r) => ({ x: r.dateEaten, y: r.weightLbs }));

// Nachher: numerische Epoch-Millis sind das, was eine Zeitachse wirklich will.
const data = rows.map((r) => ({ x: new Date(r.dateEaten).getTime(), y: r.weightLbs }));
```

> **Bug Nr. 2 — Dark Mode sah furchtbar aus**
> Als endlich Daten sichtbar waren, war der Dark Mode unlesbar — Legendentext fast unsichtbar, Gitterlinien matschig. Ich sagte es unverblümt: *„dark mode charts look s**… resolve the theme and then decide on colors."*

Die Ursache war subtil — und ein gutes Beispiel für etwas, das ein Mensch sofort bemerkt und eine Maschine nicht: Die Theme-Farben der App sind als `oklch()`-CSS-Variablen und `color-mix()`-Ausdrücke gespeichert, die ein `<canvas>` schlicht nicht auflösen kann. Der Fix bestand darin, das aktive Theme sauber zu erkennen und dem Chart **explizite, gut lesbare Farbpaletten pro Modus** zu geben — und außerdem die rohen Tagesserien

als transluzente Punktwolke zu rendern, mit den gleitenden Durchschnitten als klare Linien obendrauf, was das unruhige Kalorien-Chart entrauschte.



Zwei Chart-Bugs, zwei Root-Cause-Fixes, dann Charts, die endlich die Geschichte erzählen

Zwei Iterationen, beide angestoßen dadurch, dass ich auf den tatsächlich gerenderten Bildschirm schaute und „nein" sagte. Genau das ist der Job. Die KI ist erstaunlich schnell im Produzieren und Reparieren; sie ist *kein*

Ersatz für jemanden mit Geschmack, der sich das Ergebnis ansieht und entscheidet, ob es gut genug ist. Dieser Jemand ist nach wie vor, sehr eindeutig, ein Mensch.

Die Zugabe: kein Fremdanbieter-Lock-in, Dark Mode, Deployment, Doku

Rund um und nach den Kernphasen passierten ein paar Dinge, die Erwähnung verdienen, denn hier wird aus „läuft auf meinem Rechner" ein echtes Produkt.

- **Dark und Light Mode** wurden sauber über die gesamte App eingeführt, mit Theme-Toggle — die Sorte Aufgabe, die jede Komponente berührt und von Hand nachzurüsten eine Qual ist, hier konsistent erledigt.
-

Wir haben die **Abhängigkeit von Fremdanbieter-UI-Diensten entfernt** — die UI basiert auf Komponenten, die wir besitzen und kontrollieren, nicht auf einem gemieteten Designsystem. Keine Preisüberraschungen, keine externe Laufzeitabhängigkeit für die Oberfläche.

- Wir haben ein **VPS-Deployment mit ordentlichen Build- und Deploy-Skripten** gebaut — die App läuft nicht nur lokal, sie wird wiederholbar auf einen echten Server ausgeliefert.
- Und als Krönung: **Dokumentation** — richtig gute Dokumentation, die Sorte, für die man sechs Monate später dankbar ist, wenn man vergessen hat, wie die Verbindungshistorie des Migrationstools funktioniert.

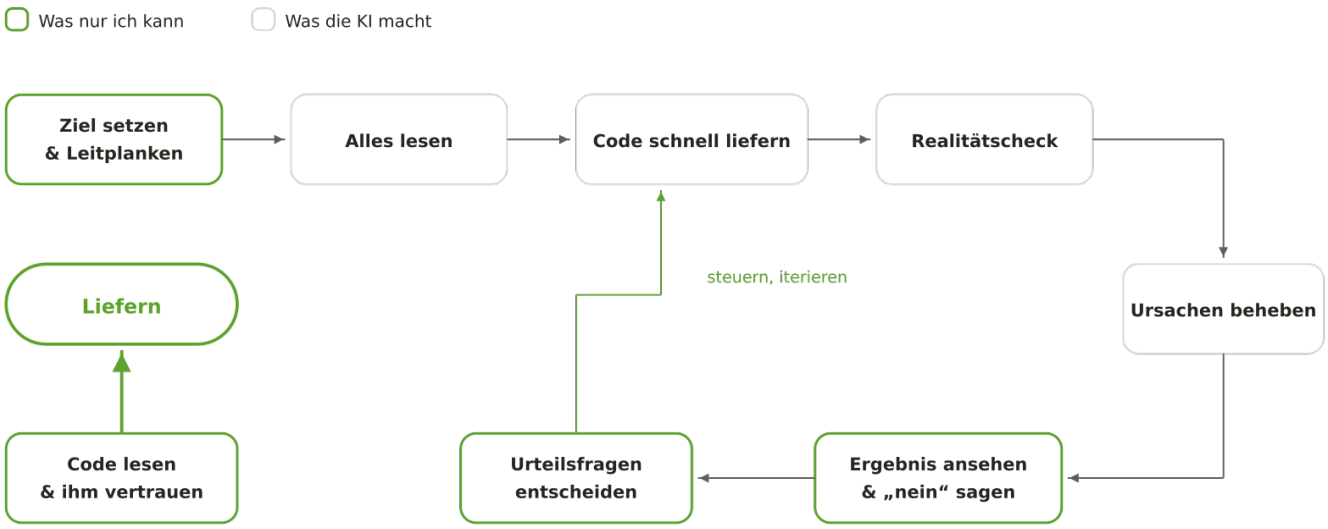
Jeder dieser Punkte ist für sich genommen ein Nachmittag kleinteiliger Arbeit in einem normalen Projekt. Hier waren es Inkremente.

Was das Ganze wirklich schnell gemacht hat

Lassen Sie mich unverblümt sagen, woher die Geschwindigkeit kam, denn „die KI hat es gemacht" ist die falsche Lektion, und ich möchte nicht, dass irgendjemand sie zieht.

Die KI hat den Code geschrieben. Aber **die KI hat den* richtigen *Code geschrieben, weil sie innerhalb eines Systems arbeitete, das ich gebaut habe.** Bedenken Sie, was im Hintergrund jedes einzelnen Schritts oben tatsächlich ablief:

- Sie hat **den echten Legacy-Quellcode und die echte Datenbank gelesen**, bevor sie irgendetwas vorschlug. Sie hat die alte Geschäftslogik nicht halluziniert — sie hat sie nachgeprüft. (Sie hat sich buchstäblich mit Firebird verbunden und einen Passwort-Hash abgeglichen, um eine Annahme zu beweisen.)
- Sie hat mir an jeder Phasengrenze **entscheidungsförmige Fragen gestellt** und *mein* Urteil die Richtung setzen lassen: Namenskonventionen, Hashing-Strategie, wo die UX modernisiert wird, wie mit Daten-Eigenheiten umzugehen ist.
- Sie hat **ihre eigene Arbeit ständig gegen die Ground Truth verifiziert** — gerenderte Summen gegen ein `SELECT SUM(...)` geprüft, CRUD-Roundtrips bewiesen, Logins tatsächlich durchgeführt — nicht „sieht für mich fertig aus".
- Und wenn sie etwas Falsches oder Hässliches produzierte — die Großbuchstaben-Namen, das Float-Rauschen, die leeren Charts, den unlesbaren Dark Mode — **habe ich es erkannt und korrigiert,** und sie hat die eigentliche Ursache behoben, statt das Symptom zu übertünchen.



Die Arbeitsteilung: menschliches Urteilsvermögen steuert einen schnellen KI-Zyklus aus Produzieren und Prüfen

Nichts davon funktioniert ohne einen Operator, der Prisma lesen kann, ein Off-by-one in einer Kaloriensumme erkennt, ein Fließkomma-Speicherartefakt auf den ersten Blick identifiziert und weiß, dass ungesalzenes SHA-512 still upgegradet und nicht behalten gehört. **Die KI ist ein phänomenaler Kraftverstärker für Expertise. Sie ist kein Ersatz dafür.** Richten Sie sie ohne diese Expertise auf ein hartes Problem, und Sie bekommen selbstbewusste, plausible, subtil falsche Ausgaben in beängstigender Geschwindigkeit.

Das ist die Fähigkeit, die ich sechs Monate lang aufgebaut habe. Nicht „Prompting". Urteilsvermögen, in der Geschwindigkeit der Generierung.

Die Zusammenfassung, die Sie wirklich behalten sollen

Eine echte, produktive Anwendung — Delphi-TMS-WEB-Core-Frontend, TMS-XData-REST-Backend, Firebird-Datenbank, drei Jahre Live-Benutzerdaten — wurde auf Next.js 16, React 19, Prisma 7 und PostgreSQL neu aufgebaut. Neues Schema mit sauberer, idiomatischer Benennung. Ein live streamendes Datenmigrationstool, das jede Zeile verlustfrei und ohne Passwort-Resets bewegt hat. Moderne Session-Authentifizierung. Ein Server-Action-Backend ohne separate REST-Schicht. Ein Tages-Tracker, eine Lebensmittelliste und eine Nährwertansicht, alles responsiv und dialoggetrieben. Jedes Chart in Chart.js neu erstellt, mit erhaltener Legacy-Mathematik. Dark und Light Mode. Kein Fremdanbieter-UI-Lock-in. VPS-Deployment-Skripte. Und solide Dokumentation.

Wochen an Arbeit — realistisch ein Monat oder mehr, sorgfältig von Hand erledigt — geliefert in etwa vier Stunden.

Aber hören Sie die eigentliche Lektion, denn sie ist die, die für die nächsten Jahre dieses Berufs zählt: **Die vier Stunden sind nur möglich wegen der sechs Monate des Lernens und Erfahrungssammelns.** Der Produktivitätsgewinn ist real und er ist gewaltig, aber er wird *freigeschaltet durch* einen Experten, der das Tooling beherrscht, den generierten Code mit kritischem Blick bewertet, die Urteilsentscheidungen trifft, die eine Maschine nicht treffen sollte, und so promptet, dass es geradewegs aufs Ziel zugeht. Nehmen Sie den Experten aus dieser Schleife, und die Geschwindigkeit produziert keine Migration — sie produziert ein sehr schnelles Chaos.

Wir sind, ganz ehrlich, an einem neuen Punkt. Zum ersten Mal kann die *Erfahrung und der Geschmack* eines

Entwicklers in der Geschwindigkeit des Denkens angewendet werden statt in der Geschwindigkeit des Tippens. Der Engpass ist von „wie schnell kann ich es schreiben" zu „wie gut kann ich es lenken und beurteilen" gewandert. Das ist keine Bedrohung für gute Ingenieure. Das ist der beste Tag, den gute Ingenieure je hatten.

Ich bin so begeistert vom Software-Bauen wie seit Jahren nicht mehr. Vier Stunden für einen Monat Arbeit — das macht etwas mit einem.

Bauen wir was.

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 —