NEWS · JULY 2, 2026

Quatro Horas para Migrar um App Delphi Multicamadas de 6 Anos

By Dr. Holger Flick

A afirmação, dita com precisão

Quero ser muito preciso sobre a afirmação nesse título, porque ela soa como o tipo de coisa que as pessoas dizem para vender um curso. Uma aplicação em produção — um frontend Delphi **TMS WEB Core**, um backend REST **TMS XData** e um banco de dados **Firebird** com dezenas de milhares de linhas de histórico real de usuários — foi reconstruída sobre uma stack completamente moderna: **Next.js 16, React 19, Prisma 7, PostgreSQL**. Novo schema. Nova autenticação. Novo backend. Cada gráfico recriado. Dark mode. Uma ferramenta de migração de dados ao vivo com UI de progresso. Scripts de deploy para VPS. Documentação. Do início ao fim, levou **cerca de quatro horas.** Fazer isso à mão — com cuidado, do jeito que você teria que fazer com dados reais de usuários — é um trabalho de **várias semanas**. Já fiz migrações assim do jeito lento. Isto não foi isso.

Se você levar uma única coisa deste post, leve esta: **as quatro horas não são a história. Os seis meses antes das quatro horas são a história.** A velocidade no final é uma *consequência* de uma habilidade que venho construindo deliberadamente — a habilidade de conduzir a IA como uma ferramenta de engenharia genuína, e não como uma novidade. Este post é o relato detalhado de como isso realmente se parece quando funciona.

Vou percorrer cada fase. Vou mostrar a arquitetura, as decisões, os diagramas. E — porque uma carta de referência que só mostra os melhores momentos não vale nada — vou mostrar os vários pontos em que a IA fez algo *errado* ou *feio*, eu percebi, e corrigimos o rumo. Esse ciclo de perceber-e-corrigir é o trabalho inteiro agora.

Vamos lá.

O cenário (ou: sobre o que estou sendo deliberadamente vago)

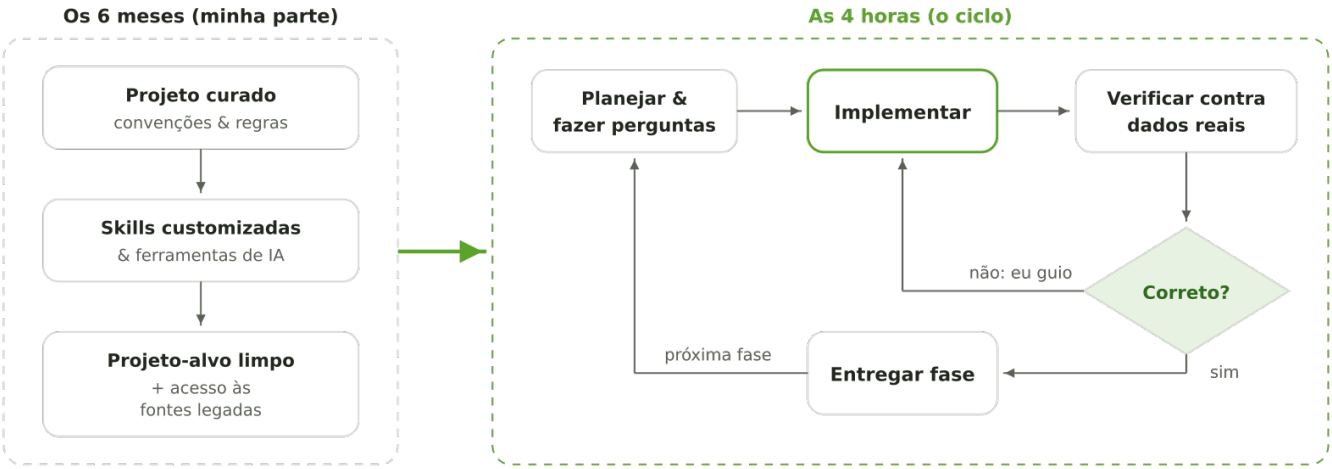
Aqui vai o enquadramento honesto.

Eu **não** sentei, digitei "migre meu app Delphi" e fui embora. O que eu realmente fiz antes de uma única linha de código novo ser escrita é a parte em que passei meio ano ficando bom, e é a parte que vou manter um pouco guardada — porque é a diferença entre uma demo e uma entrega.

O que me sinto confortável em contar:

- Preparei um **projeto Next.js limpo e vazio** como alvo. Uma tela em branco, configurada do jeito que eu sei que produz bons resultados.
- Dei à IA **acesso controlado às fontes antigas** — as units Delphi, a camada de serviços XData, o schema Firebird — como material de referência somente leitura.
- Forneci um conjunto cuidadosamente calibrado de **instruções de projeto, regras da casa, skills e ferramentas de IA** que venho refinando há meses. Elas codificam como eu quero que o código seja escrito: convenções, versões de frameworks, regras de design, tudo. Essa é a minha "mágica", e ela faz *muito* trabalho silencioso nos bastidores de tudo o que vem abaixo.

Essa preparação é a alavanca. O modelo é forte, mas um modelo forte apontado para um alvo vago produz lixo plausível. Um modelo forte apontado para um **alvo nítido, com restrições nítidas, supervisionado por alguém que sabe ler o resultado** produz código de produção. A distância entre esses dois desfechos é expertise, e ela não vem de graça.



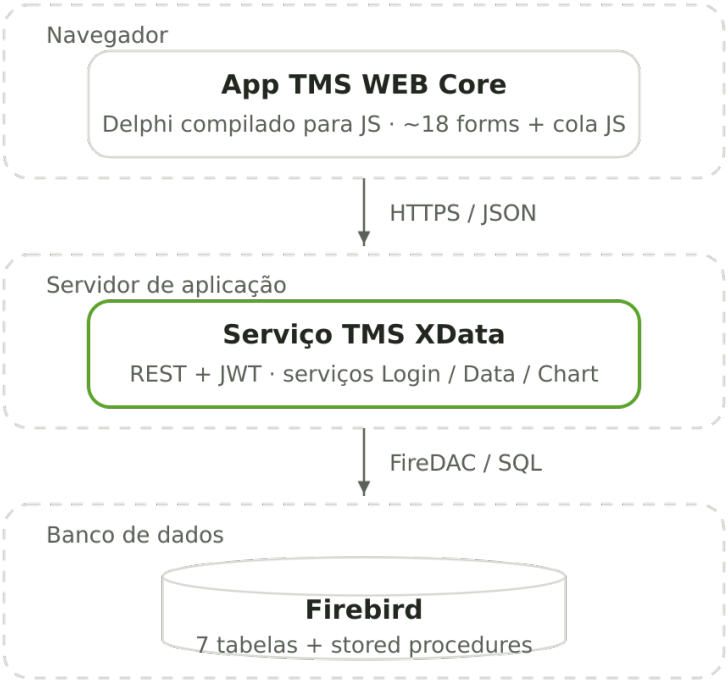
Seis meses de preparação alimentam um ciclo rápido de planejar–implementar–verificar

Tudo daqui em diante são as quatro horas.

O sistema legado que estávamos substituindo

Sejamos justos com o app antigo: ele funcionava, e funcionava há anos. É uma aplicação de acompanhamento de refeições e do corpo — você registra o que come ao longo do dia, anota peso, passos e sono, e ela mostra seu progresso em gráficos ao longo do tempo. Pessoas reais vinham usando e alimentando o sistema com dados desde 2023.

Arquiteturalmente, era um arranjo de três camadas muito típico da era Delphi:



A arquitetura legada de três camadas: cliente WEB Core, servidor REST XData, banco Firebird

As peças:

- **Frontend:** TMS WEB Core — Object Pascal compilado para JavaScript — com cerca de dezoito forms (login, menu principal, grade de dados diários, editores por refeição, lista de alimentos, visão de nutrição, gráficos) e uma boa quantidade de JavaScript de cola escrito à mão.
- **Backend:** Um servidor TMS XData expondo serviços REST tipados — um `LoginService`, um `DataService` e um `ChartService` — com autenticação JWT e FireDAC conversando com o banco.
- **Banco de dados:** Firebird, com sete tabelas e um punhado de stored procedures que faziam agregação por dia (calorias consumidas, somas nutricionais diárias, consolidação de peso/passos).

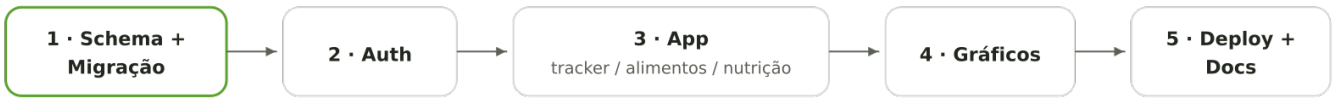
Nada de errado com isso. Mas é uma stack com um pool de talentos encolhendo, uma história de deployment que pensa primeiro em desktop e uma UI que mostra a idade. O cliente queria o sistema em algo que um desenvolvedor web moderno consiga pegar, estender e fazer deploy em qualquer lugar. Justo.

Meu trabalho: **reconstruí-lo, fielmente, sobre uma stack de 2026** — sem perder uma única linha daquele histórico de três anos de dados.

O plano geral

Antes de tocar em código, a IA e eu concordamos em uma abordagem por fases. Este é o primeiro lugar onde a supervisão importa: eu não queria uma mangueira de incêndio de arquivos, eu queria um **plano que eu pudesse aprovar.** Fechamos em:

1. **Fundação** — schema Prisma, banco PostgreSQL e uma ferramenta de migração de dados de verdade para trazer o histórico do Firebird.
2. **Autenticação** — porque você não consegue testar nada como um usuário real até conseguir fazer login como um.
3. **A aplicação** — tracker diário, lista de alimentos, visão de nutrição.
4. **Os gráficos** — explicitamente destacados como recurso de primeira classe, porque os gráficos eram o coração do app antigo.
5. **Deploy e docs** — colocar em uma VPS com scripts repetíveis e documentar tudo.



As cinco fases acordadas, em ordem

Crucialmente, no início de cada fase a IA **pausava e me fazia perguntas** — perguntas reais, em formato de decisão, não enchimento do tipo "devo continuar?". Qual estratégia de hashing para as senhas legadas? Onde o estado da migração deve viver? Como tratar uma peculiaridade dos dados? Essas perguntas são onde a minha experiência é injetada no resultado da máquina. Vou sinalizá-las conforme avançamos.

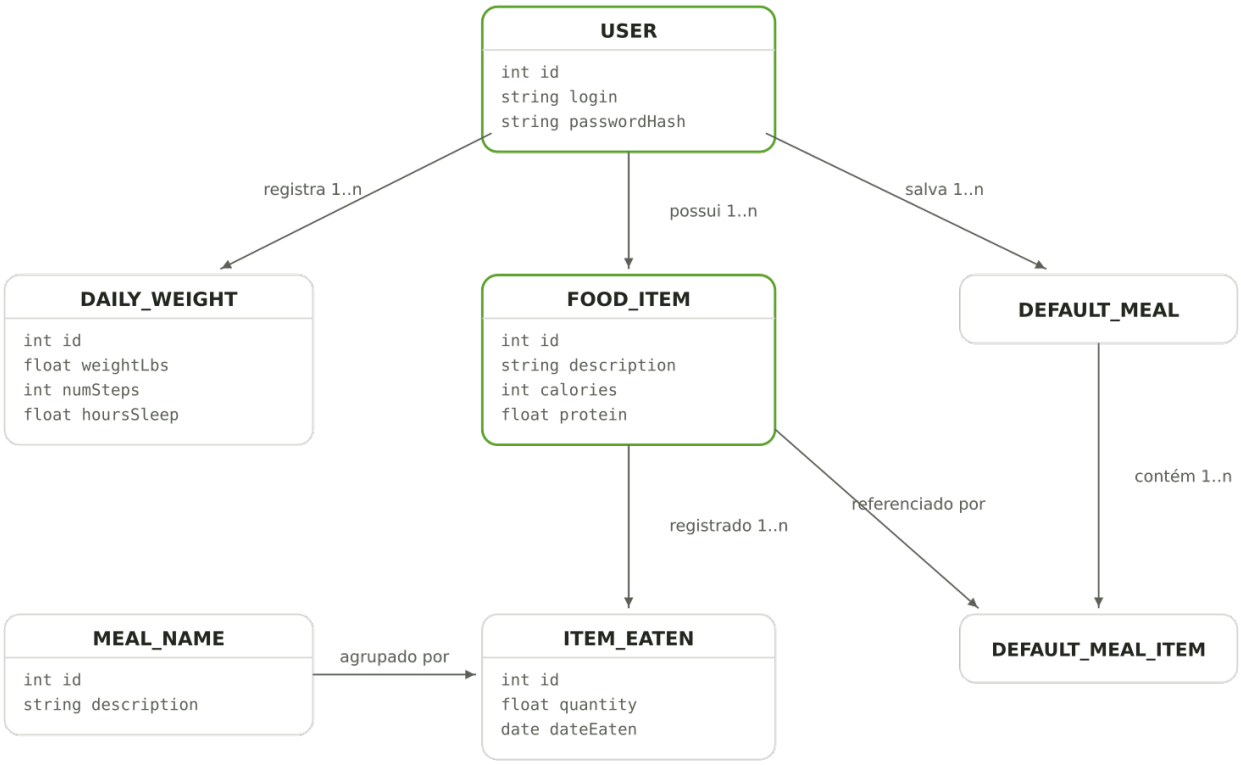
Fase 1 — O schema e a grande migração de dados

Lendo o mundo antigo

A primeira coisa que a IA fez foi *ler*. Não só o script `CREATE` do Firebird, mas as implementações dos serviços

XData em Pascal — porque o schema sozinho não conta a lógica de negócio. As stored procedures, o SQL dentro dos serviços Delphi, a forma como "calorias de um dia" era de fato calculada — tudo isso vive no código do backend, e tudo isso tinha que ser entendido antes de um único model Prisma ser escrito.

Sete tabelas atravessaram conceitualmente:



As sete tabelas e seus relacionamentos, como levadas para o novo schema

Correção #1: "não simplesmente carregue as colunas em maiúsculas"

Aqui foi o primeiro lugar onde eu intervi. O schema legado usava a convenção gritada do Firebird — `PERMISSION_STRING`, `DATE_EATEN`, `NUM_STEPS`, `DESCR`. A primeira versão do schema Prisma estava carregando fielmente esses nomes para o novo mundo.

Eu parei o processo: **use nomes idiomáticos de TypeScript.** `DESCR` ’`description`. `LBS` ’`weightLbs`. `NUM_STEPS` ’`numSteps`. O schema deve se ler como código moderno, não como um dump de banco de dados de 2010. A IA regenerou o schema inteiro com nomes de campos limpos em camelCase e relações adequadas. Coisa pequena, mas é a diferença entre uma base de código que um desenvolvedor novo curte e uma que ele tolera.

O problema das senhas (e uma decisão que só um humano deve tomar)

Então veio uma pergunta genuinamente interessante. O backend XData antigo verificava senhas com a função de hash embutida do Firebird:

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

Conseguiríamos reproduzir isso em Node — significando **zero redefinições de senha para os usuários existentes** — ou teríamos que apagar as senhas e forçar todo mundo a redefinir?

A IA não chutou. Ela **conectou no banco Firebird ao vivo, puxou o hash de senha armazenado de um usuário real e provou o algoritmo**, fazendo o hash de uma senha de teste conhecida em Node e batendo byte a byte:

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

// SHA-512, hex, maiúsculas — exatamente o que o CRYPT_HASH do Firebird emite.
const candidate = createHash("sha512").update(password).digest("hex").toUpperCase();
const matches = candidate === storedHashFromFirebird;
```

Confirmado: era um SHA-512 puro, sem salt. Totalmente reproduzível.

Isso transformou uma pergunta assustadora em uma decisão limpa, que ela então colocou para mim:

> **A proposta que eu aprovei**
> Verificar contra o hash SHA-512 legado no login e **atualizar cada usuário de forma transparente para bcrypt moderno no primeiro login bem-sucedido.** Sem redefinições. Sem interrupção. O hashing antigo e fraco se aposenta sozinho, um login de cada vez.

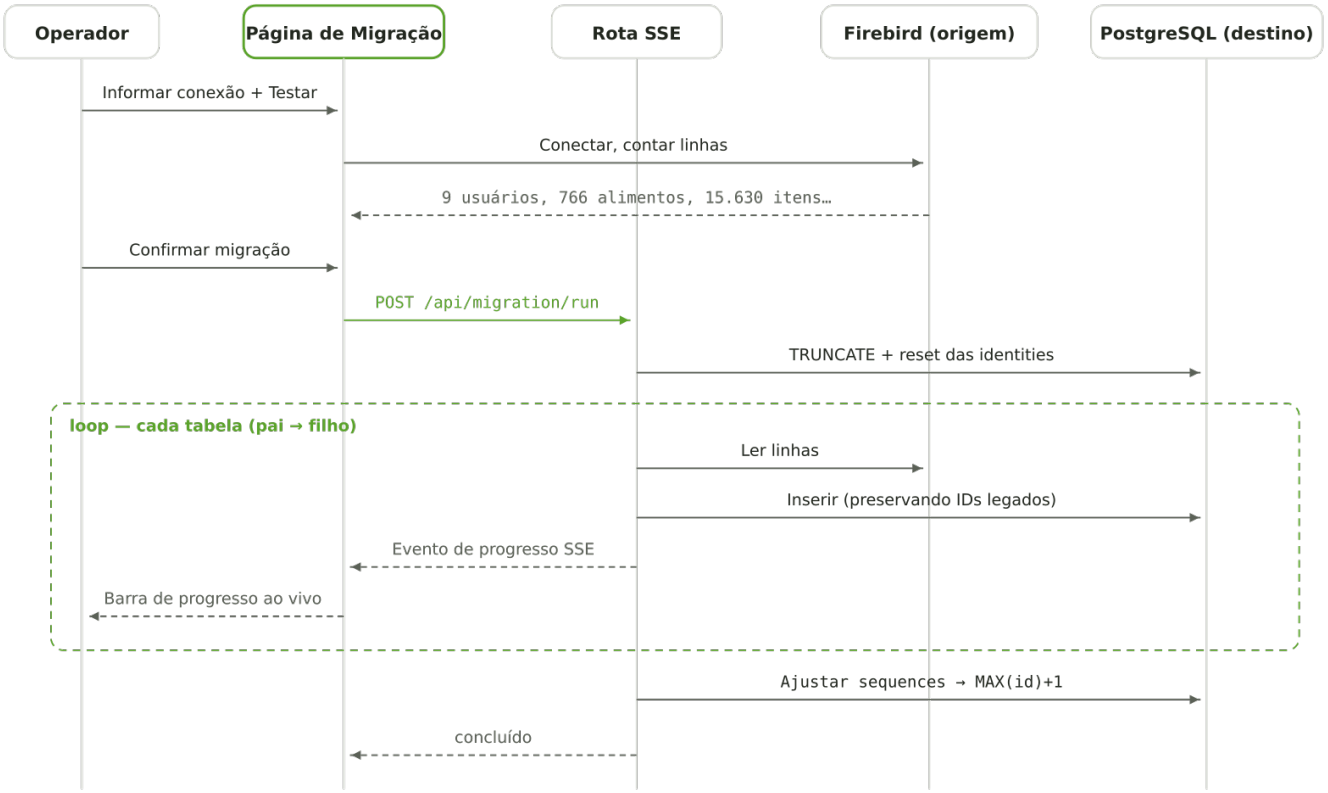
Aprovei. É esse o padrão que eu quero em uma migração: ninguém recebe um e-mail dizendo "por favor, redefina sua senha porque trocamos de backend." As pessoas simplesmente fazem login e, nos bastidores, sua segurança melhora silenciosamente. A IA propôs; meu trabalho foi reconhecer que era a decisão certa e dar sinal verde.

A ferramenta de migração em si

Esta é a parte de que mais me orgulho, porque é onde "a IA escreveu um código" vira "a IA construiu uma ferramenta."

Em vez de um script descartável, construímos uma **página de migração** de verdade — fora da autenticação (tem que ser; é ela que *cria* os primeiros usuários) — com:

- Um formulário para informar os detalhes de conexão do Firebird de origem, com um botão **"Testar Conexão"** que reporta contagens de linhas ao vivo antes de você se comprometer.
- Um **histórico** de conexões para você não redigitar credenciais.
- Um diálogo de confirmação rigoroso, porque isso apaga e recarrega o destino.
- **Server-Sent Events transmitindo o progresso ao vivo** — uma barra por tabela — enquanto milhares de linhas atravessam.



A execução da migração: testar, confirmar e então transmitir o progresso por tabela via SSE

Por baixo dos panos há sete migradores ordenados (pais antes de filhos, para que chaves estrangeiras nunca quebrem), cada um reportando seu próprio placar:

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

A coisa toda rodou contra o banco de dados ao vivo e moveu **9 usuários, 766 alimentos, 15.630 refeições registradas e 3.409 registros de peso com zero itens pulados.**

Correção #2: o ruído de ponto flutuante

Depois da migração, olhei o app rodando e algo me incomodou: uma quantidade que deveria mostrar `1.1` estava exibindo `1.100000023841858`. Um peso mostrava `252.60000610351562`. Feio.

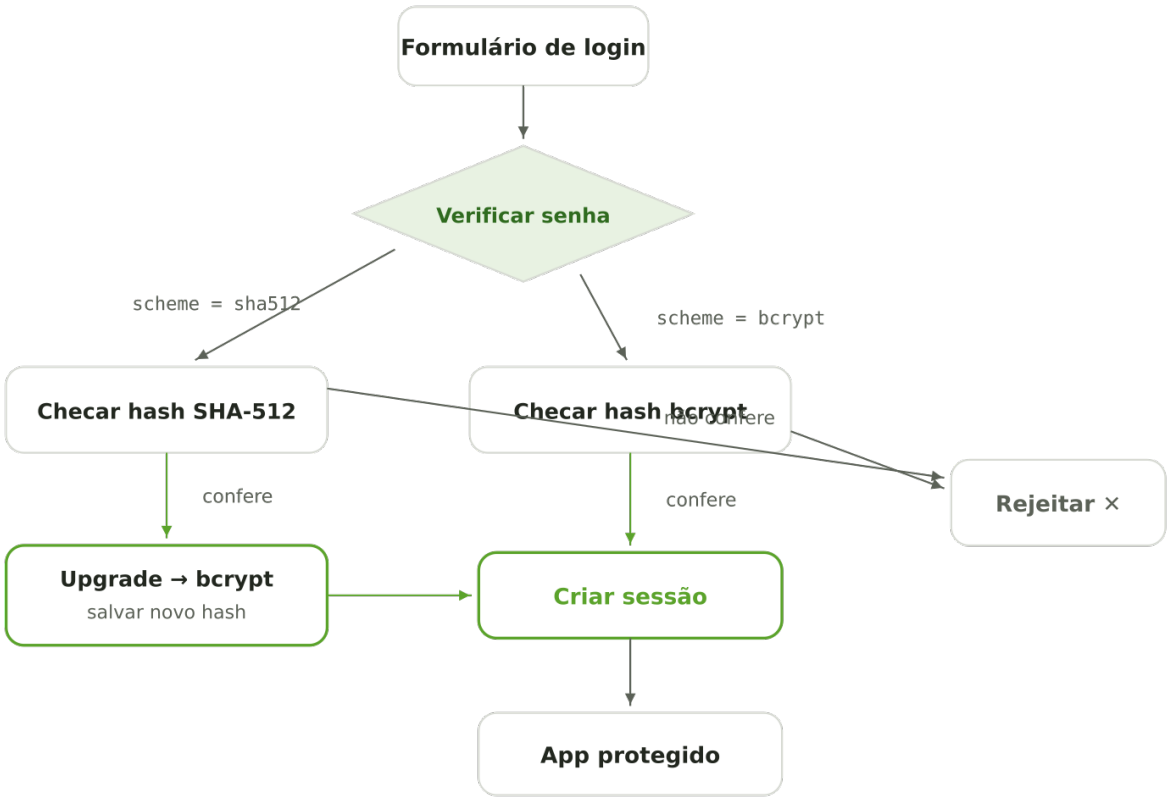
Eu sinalizei — *"a precisão parece errada, é arredondamento ou armazenamento?"* — e a IA diagnosticou corretamente: as colunas antigas do Firebird eram floats de 32 bits; promovidas para os doubles de 64 bits do PostgreSQL, o ruído de precisão simples fica visível. Ela ofereceu três correções e eu escolhi a mais limpa: **arredondar os valores no momento da migração**, para que os próprios dados armazenados fiquem arrumados e a correção seja reproduzível em qualquer nova execução futura.

Já que estávamos ali, apontei um segundo problema, mais sutil: os totais de calorias por refeição nem sempre batiam com a soma dos itens visíveis (arredondar a soma bruta vs. somar os itens arredondados — diferença de um, e *parece* errado mesmo quando é matematicamente correto). Corrigimos os totais para reconciliarem com o que o olho soma. Esse é o tipo de bug pelo qual uma ferramenta automatizada passa reto e que um humano que *usa o produto* pega na hora.

Fundação concluída
Schema, banco de dados e uma ferramenta de migração de verdade — com os dados verificados contra a fonte da verdade.

Fase 2 — Autenticação

Fase curta, fase importante. Usando `iron-session` para sessões em cookies criptografados, construímos o fluxo de login em torno da estratégia verificar-SHA-512-e-atualizar-para-bcrypt da Fase 1, um layout de rotas protegidas que redireciona visitantes não autenticados e uma página de login limpa.

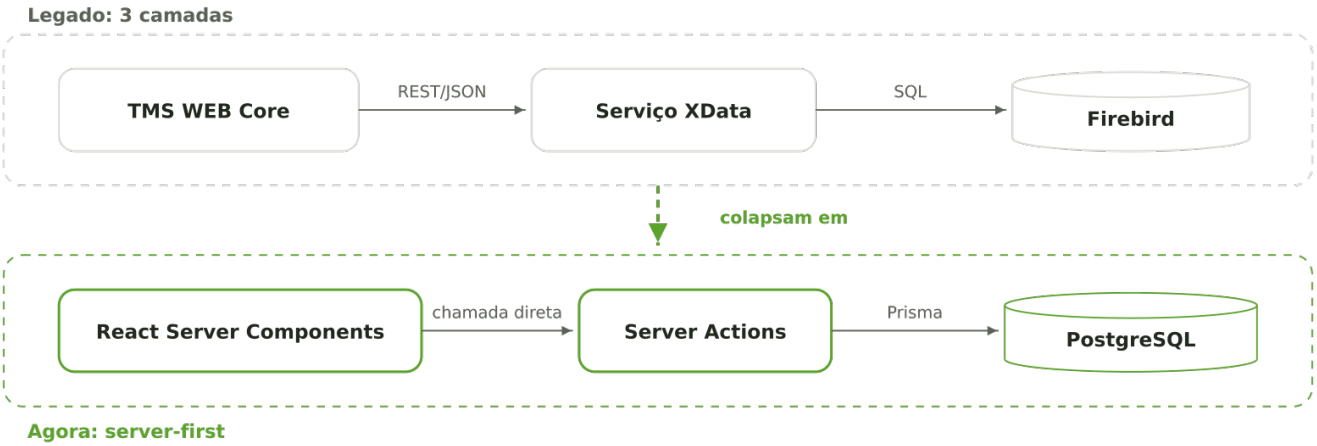


Fluxo de login: verificar SHA-512 legado ou bcrypt, atualizando hashes antigos silenciosamente no sucesso

A verificação que importava aqui não era visual — era provar que o login *realmente funcionava* com uma conta migrada. A IA selou uma sessão real, fez login como um usuário migrado genuíno, confirmou que o upgrade para bcrypt disparou no primeiro login e confirmou que uma senha errada era rejeitada. Depois restaurou a conta ao estado pós-migração, para que nada ficasse alterado.

Fase 3 — Reconstruindo a aplicação

Aqui é onde a velha arquitetura de três camadas foi colapsada em algo dramaticamente mais simples. Sem backend REST separado. Sem camada de API para projetar, versionar e proteger. As **Server Actions do Next.js** deixam a UI chamar funções do lado do servidor diretamente, com o acesso ao banco, as checagens de propriedade e a validação todos vivendo no servidor.



Três camadas legadas colapsam em uma arquitetura Next.js server-first

O tracker diário

A peça central. Uma visão do dia com um navegador de datas, um card por refeição (café da manhã, almoço, jantar, os lanches), os itens registrados em cada uma com a matemática de calorias ao vivo, e um painel de peso/passos/sono/composição corporal.

Aqui eu fiz uma decisão explícita de design que a IA então executou: **modernizar o modelo de interação.**

O app antigo era página-por-formulário — clique para uma tela separada para editar uma refeição, outra para editar o peso. Substituímos tudo isso por **diálogos modais e edição inline** a partir de uma única visão do dia.

Adicione um item por um seletor de alimentos pesquisável, com backend no servidor e prévia de calorias ao vivo. Edite uma quantidade no lugar. Ajuste o peso sem sair da página. São os mesmos dados, com uma cara de 2026.

Cada mutação carrega uma **guarda de propriedade** no servidor — um usuário só pode tocar nos próprios dados — substituindo as checagens de claims JWT que os serviços XData antigos faziam.

A lista de alimentos

CRUD completo sobre os alimentos com sua tabela nutricional completa, como uma tabela responsiva no desktop e cards no mobile (paridade mobile era regra dura — nenhuma funcionalidade escondida em telas pequenas). Pesquisável, paginada. E uma **guarda de exclusão** genuinamente bem pensada: você não pode excluir um alimento referenciado por refeições registradas — o app avisa que ele é usado em *N* refeições, em vez de falhar cripticamente ou de deixar órfãos três anos de histórico.

A visão de nutrição

Um detalhamento diário de macros — proteína, carboidratos, fibras, carboidratos líquidos, açúcares, açúcares adicionados, gordura, gordura saturada, colesterol, sódio, água — em um intervalo de datas selecionável, com médias diárias, reproduzindo fielmente a antiga stored procedure `GET_DAILY_NUTRITION_STATS` como uma agregação SQL limpa.

Ao longo desta fase, a verificação foi contínua: a IA fez login como um usuário migrado real e confirmou que o total de um dia renderizado batia com o agregado do próprio banco **na caloria exata**:

```
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;
```

Fase 4 — Os gráficos

Os gráficos não eram negociáveis. No app antigo, eles eram a recompensa — a prova visual de meses de esforço. Os dados do cliente contam uma história genuinamente ótima (a jornada de peso de vários anos de um usuário), e ela tinha que ser bem contada.

Ficamos com o **Chart.js**, deliberadamente, porque ele porta de forma limpa para o React e dá controle real. A camada de dados reproduziu a lógica legada exatamente, incluindo um detalhe que eu insisti que honrássemos com fidelidade: as médias móveis de "14 dias" e "50 dias" do app antigo eram, na verdade, computadas como **janelas baseadas em linhas** sobre pesagens consecutivas, não em dias de calendário. Reproduzimos isso, para que os usuários de longa data vejam as mesmas curvas que sempre viram. Gráfico de peso, gráfico de calorias e os cards de estatísticas de perda de peso (inicial vs. atual, variação total, taxa por semana/mês).

Correções #3 e #4: os gráficos que resistiram

Os gráficos não saíram certos de primeira. Vale a pena mostrar, porque é a ilustração mais honesta do ciclo.

> **Bug #1 — os gráficos vazios**
> Os gráficos renderizavam *vazios* — o eixo X mostrava "12AM, 1AM, 2AM…" em vez de datas cobrindo 2023–2026. Mandei um screenshot: *"os gráficos estão vazios?!"* A IA diagnosticou na hora: ela estava alimentando **strings** de data a um eixo de tempo que, com o parsing desabilitado, esperava timestamps numéricos. Todos os 1.173 pontos tinham colapsado em um único dia.

A correção foi entregar ao eixo timestamps de verdade em vez de strings:

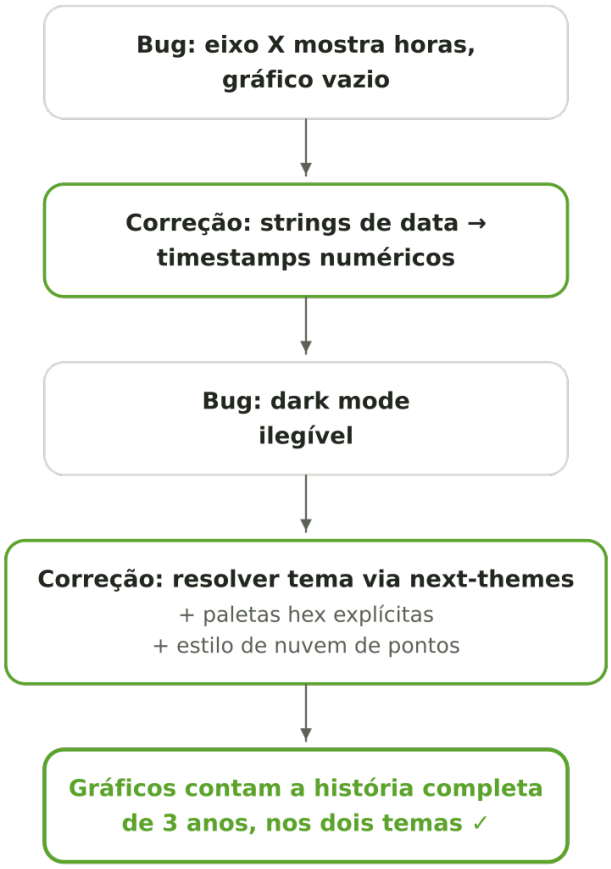
```
// Antes: strings de data colapsam em um único ponto com o parsing desabilitado.
const data = rows.map((r) => ({ x: r.dateEaten, y: r.weightLbs }));

// Depois: epoch-millis numéricos são o que um eixo de tempo realmente quer.
const data = rows.map((r) => ({ x: new Date(r.dateEaten).getTime(), y: r.weightLbs }));
```

> **Bug #2 — o dark mode estava horrível**
> Com os dados finalmente aparecendo, o dark mode estava ilegível — o texto da legenda quase invisível, as linhas de grade lamacentas. Eu falei sem rodeios: *"os gráficos no dark mode estão uma m**… resolva o tema e depois decida as cores."*

A causa raiz era sutil, e um bom exemplo de algo que um humano nota instantaneamente e uma máquina não: as cores de tema do app são armazenadas como variáveis CSS `oklch()` e expressões `color-mix()` que um `<canvas>` simplesmente não consegue resolver. A correção foi detectar o tema ativo corretamente e entregar ao gráfico **paletas de cores explícitas e legíveis por modo** — além de renderizar a série diária bruta como

uma nuvem de pontos translúcida, com as médias móveis como linhas limpas por cima, o que tirou o ruído do gráfico de calorias.



Dois bugs nos gráficos, duas correções de causa raiz, e então gráficos que finalmente contam a história

Duas iterações, ambas iniciadas por mim olhando a tela realmente renderizada e dizendo "não." Esse é o trabalho. A IA é assombrosamente rápida em produzir e corrigir; ela *não* substitui alguém com bom gosto olhando o resultado e decidindo se está bom o suficiente. Esse alguém ainda é, muito claramente, uma pessoa.

A rodada bônus: zero lock-in de terceiros, dark mode, deploy, docs

Algumas coisas aconteceram em torno das fases principais, e depois delas, que merecem menção — porque são onde "funciona na minha máquina" vira "é um produto de verdade".

- **Dark e light mode** foram introduzidos de forma limpa em todo o app com um alternador de tema — o tipo de coisa que toca cada componente e é penoso de retrofitar à mão, feito de forma consistente.
- **Removemos a dependência de serviços de UI de terceiros** — a UI é construída sobre componentes que possuímos e controlamos, não um design system alugado. Sem surpresas de preço, sem dependência de runtime externo para a interface.
- Construímos o **deploy em VPS com scripts adequados de build e deploy** — o app não roda apenas localmente, ele é entregue em um servidor real, de forma repetível.
- E, para coroar, **documentação** — documentação genuinamente boa, do tipo que você agradece seis meses depois, quando já esqueceu como funciona o histórico de conexões da ferramenta de migração.

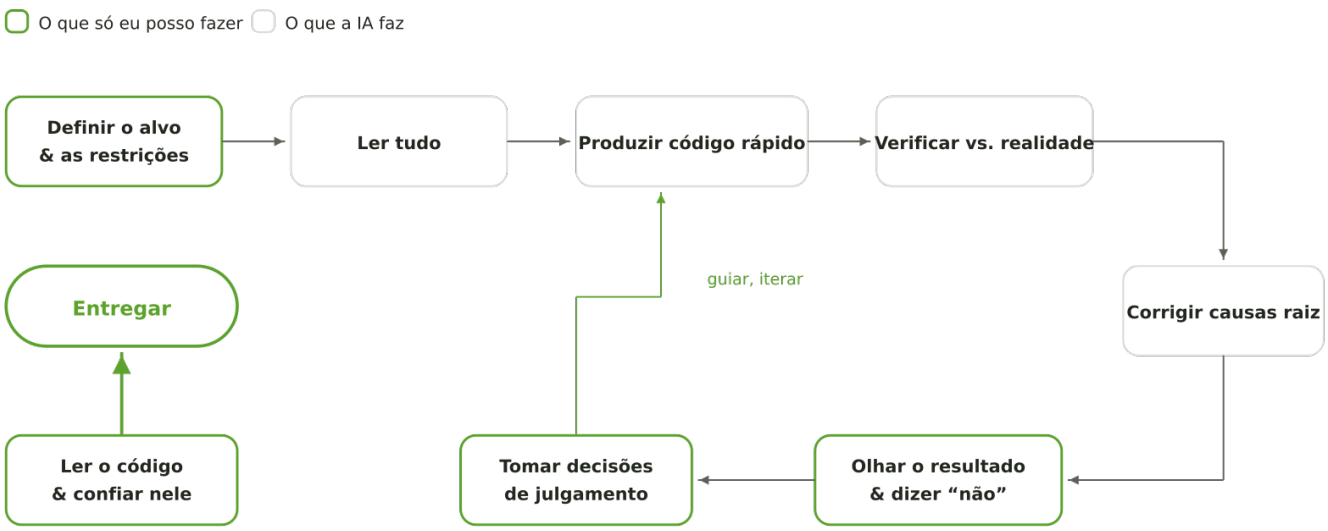
Cada um desses itens é, sozinho, uma tarde de trabalho chato em um projeto normal. Aqui foram incrementos.

O que realmente tornou isso rápido

Deixe-me ser direto sobre de onde veio a velocidade, porque "a IA fez" é a lição errada e eu não quero que ninguém a tire daqui.

A IA escreveu o código. Mas **a IA escreveu o código* certo *porque estava operando dentro de um sistema que eu construí.** Considere o que estava realmente acontecendo nos bastidores de cada passo acima:

- Ela **leu o código legado real e o banco de dados real** antes de propor qualquer coisa. Não alucinou a velha lógica de negócio — foi lá e confirmou. (Ela literalmente conectou no Firebird e bateu um hash de senha para provar uma suposição.)
- Ela **me fez perguntas em formato de decisão** em cada fronteira de fase e deixou o *meu* julgamento definir a direção: convenções de nomes, estratégia de hashing, onde modernizar a UX, como tratar peculiaridades dos dados.
- Ela **verificou o próprio trabalho contra a fonte da verdade** constantemente — totais renderizados conferidos contra um `SELECT SUM(...)`, ciclos de CRUD provados, logins realmente executados — nada de "parece pronto para mim."
- E quando produziu algo errado ou feio — os nomes em maiúsculas, o ruído de float, os gráficos vazios, o dark mode ilegível — **eu percebi e corrigi,** e ela consertou a verdadeira causa raiz em vez de maquiar o sintoma.



A divisão de trabalho: julgamento humano guiando um ciclo rápido de produzir-e-verificar da IA

Nada disso funciona sem um operador que saiba ler Prisma, perceber um off-by-one em um total de calorias, reconhecer um artefato de armazenamento de ponto flutuante à primeira vista e saber que um SHA-512 sem salt deve ser silenciosamente atualizado em vez de mantido. **A IA é um multiplicador de força fenomenal sobre a expertise. Ela não é um substituto para a expertise.** Aponte-a para um problema difícil sem essa expertise e você recebe um resultado confiante, plausível e sutilmente errado, a uma velocidade aterrorizante.

Essa é a habilidade que passei seis meses construindo. Não "fazer prompts". Julgamento, na velocidade da geração.

O resumo que eu realmente quero que você lembre

Uma aplicação real, em produção — frontend Delphi TMS WEB Core, backend REST TMS XData, banco Firebird, três anos de dados reais de usuários — foi reconstruída sobre Next.js 16, React 19, Prisma 7 e PostgreSQL. Novo schema com nomes limpos e idiomáticos. Uma ferramenta de migração de dados ao vivo, com streaming, que moveu cada linha com zero perdas e sem redefinições de senha. Autenticação moderna por sessão. Um backend de server actions sem camada REST separada. Um tracker diário, uma lista de alimentos e uma visão de nutrição, tudo responsivo e orientado a diálogos. Cada gráfico recriado em Chart.js com a matemática legada preservada. Dark e light mode. Sem lock-in de UI de terceiros. Scripts de deploy em VPS. E documentação sólida.

Semanas de trabalho — realisticamente um mês ou mais, feito com cuidado à mão — entregues em cerca de quatro horas.

Mas ouça a lição de verdade, porque é a que importa para os próximos anos desta profissão: **as quatro horas só são possíveis por causa dos seis meses de aprendizado e acúmulo de experiência.** A produtividade é real e é impressionante, mas ela é *destravada por* um especialista que sabe gerenciar o ferramental, avaliar o código gerado com olhar crítico, tomar as decisões de julgamento que uma máquina não deveria tomar e fazer prompts que vão direto ao alvo. Tire o especialista desse ciclo e a velocidade não produz uma migração — produz uma bagunça muito rápida.

Estamos, genuinamente, em um lugar novo. Pela primeira vez, a *experiência e o bom gosto* de um desenvolvedor podem ser aplicados na velocidade do pensamento, e não na velocidade da digitação. O gargalo se moveu de "quão rápido eu consigo escrever" para "quão bem eu consigo dirigir e julgar". Isso não é uma ameaça aos bons engenheiros. É o melhor dia que os bons engenheiros já tiveram.

Estou mais animado para construir software do que estive em anos. Quatro horas para um mês de trabalho fazem isso com você.

Vamos construir.

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 —