NEWS · JULY 2, 2026

Cuatro horas para migrar una aplicación Delphi multicapa de 6 años

By Dr. Holger Flick



La afirmación, formulada con precisión

Quiero ser muy preciso con la afirmación de ese título, porque suena al tipo de cosa que la gente dice para vender un curso. Una aplicación en producción — un frontend Delphi **TMS WEB Core**, un backend REST **TMS XData** y una base de datos **Firebird** con decenas de miles de filas de historial real de usuarios — fue reconstruida sobre un stack completamente moderno: **Next.js 16, React 19, Prisma 7, PostgreSQL**. Nuevo esquema. Nueva autenticación. Nuevo backend. Cada gráfica recreada. Dark mode. Una herramienta de migración de datos en vivo con interfaz de progreso. Scripts de deploy para VPS. Documentación. De principio a fin, tomó **unas cuatro horas.** Hacer esto a mano — con cuidado, como hay que hacerlo con datos reales de usuarios — es un trabajo de **varias semanas**. He hecho migraciones así a la manera lenta. Esto no fue eso.

Si solo te llevas una cosa de este post, que sea esta: **las cuatro horas no son la historia. Los seis meses anteriores a las cuatro horas son la historia.** La velocidad del final es una *consecuencia* de una habilidad que he estado construyendo deliberadamente: la habilidad de manejar la IA como una herramienta de ingeniería genuina y no como una novedad. Este post es el relato detallado de cómo se ve eso en la práctica cuando funciona.

Voy a recorrer cada fase. Te mostraré la arquitectura, las decisiones, los diagramas. Y — porque una carta de recomendación que solo muestra los momentos estelares no vale nada — te mostraré los varios puntos en los que la IA hizo algo *mal* o *feo*, yo lo detecté y corregimos el rumbo. Ese ciclo de detectar y corregir es todo el trabajo ahora.

Vamos.

El punto de partida (o: sobre qué soy deliberadamente vago)

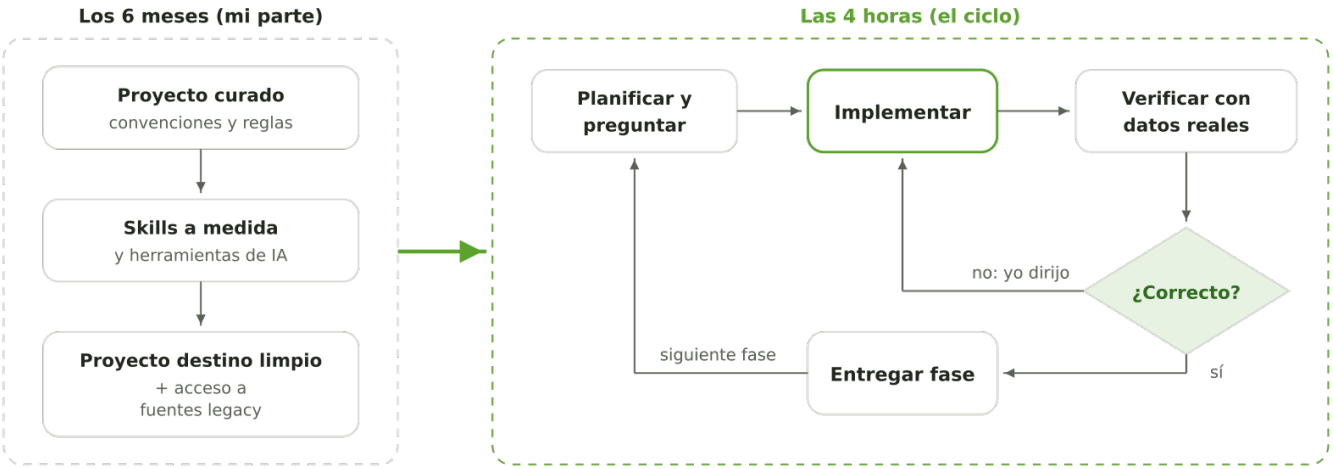
Este es el planteamiento honesto.

No me senté a escribir "migra mi aplicación Delphi" para luego irme a pasear. Lo que realmente hice antes de que se escribiera una sola línea de código nuevo es la parte en la que llevo medio año volviéndome bueno, y es la parte que voy a guardarme un poco — porque es la diferencia entre una demo y una entrega.

Lo que sí puedo contarte:

- Preparé un **proyecto Next.js limpio y vacío** como destino. Un lienzo en blanco, configurado de la manera que sé que produce buenos resultados.
- Le di a la IA **acceso controlado a las fuentes antiguas** — las unidades Delphi, la capa de servicios XData, el esquema de Firebird — como material de referencia de solo lectura.
- Aporté un conjunto cuidadosamente afinado de **instrucciones de proyecto, reglas de la casa, skills y herramientas de IA** que he refinado durante meses. Estas codifican cómo quiero que se escriba el código: convenciones, versiones de frameworks, reglas de diseño, todo. Esta es mi "magia", y hace *mucho* trabajo silencioso en el trasfondo de todo lo que viene a continuación.

Esa preparación es la palanca. El modelo es potente, pero un modelo potente apuntado a un objetivo difuso produce basura plausible. Un modelo potente apuntado a un **objetivo nítido, con restricciones nítidas, supervisado por alguien que sabe leer el resultado** produce código de producción. La brecha entre esos dos desenlaces es la experiencia, y no viene gratis.



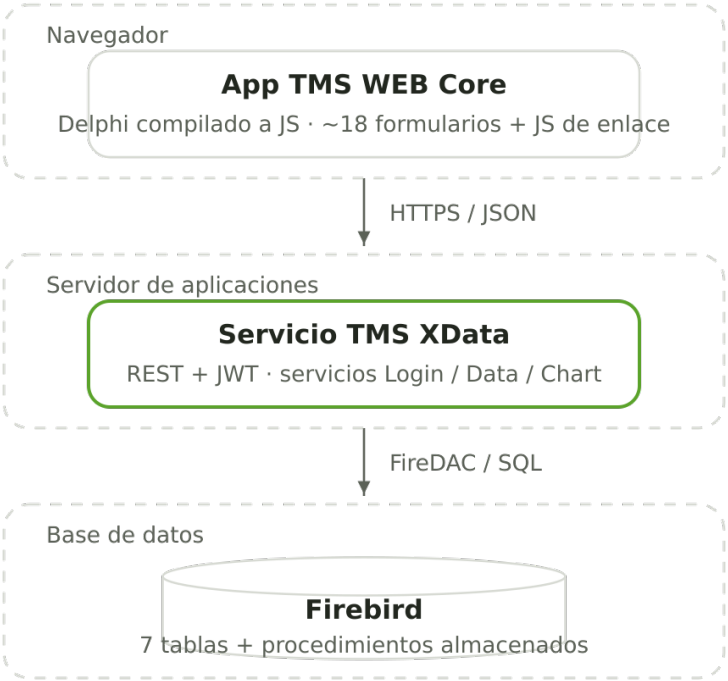
Seis meses de preparación alimentan un ciclo rápido de planificar–implementar–verificar

Todo lo que viene a partir de aquí son las cuatro horas.

El sistema legacy que íbamos a reemplazar

Seamos justos con la aplicación antigua: funcionaba, y llevaba años haciéndolo. Es una aplicación de registro de comidas y datos corporales — anotas lo que comes a lo largo del día, registras tu peso, tus pasos y tu sueño, y te grafica el progreso en el tiempo. Personas reales la venían usando y alimentando con datos desde 2023.

Arquitectónicamente, era una configuración de tres capas muy típica de la era Delphi:



La arquitectura legacy de tres capas: cliente WEB Core, servidor REST XData, base de datos Firebird

Las piezas:

- **Frontend:** TMS WEB Core — Object Pascal compilado a JavaScript — con unos dieciocho formularios (login, menú principal, cuadrícula de datos diarios, editores por comida, lista de alimentos, vista de nutrición, gráficas) y una buena cantidad de JavaScript de enlace escrito a mano.
- **Backend:** Un servidor TMS XData que exponía servicios REST tipados — un `LoginService`, un `DataService` y un `ChartService` — con autenticación JWT y FireDAC hablando con la base de datos.
- **Base de datos:** Firebird, con siete tablas y un puñado de procedimientos almacenados que hacían la agregación por día (calorías consumidas, sumas de nutrición diaria, acumulados de peso/pasos).

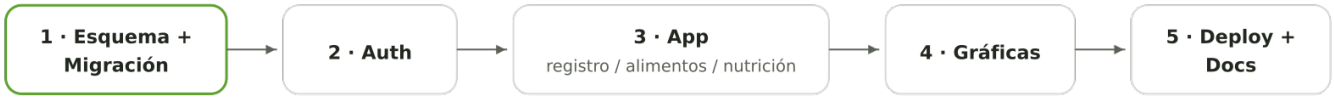
Nada malo en ello. Pero es un stack con una reserva de talento cada vez menor, una historia de despliegue pensada para escritorio y una interfaz que aparenta su edad. El cliente la quería sobre algo que un desarrollador web moderno pueda tomar, extender y desplegar en cualquier parte. Justo.

Mi trabajo: **reconstruirla, con fidelidad, sobre un stack de 2026** — sin perder ni una sola fila de esos tres años de historial de datos.

El plan general

Antes de tocar código, la IA y yo acordamos un enfoque por fases. Este es el primer lugar donde la supervisión importa: yo no quería una manguera de archivos, quería un **plan que pudiera aprobar.** Acordamos:

1. **Cimientos** — esquema Prisma, base de datos PostgreSQL y una herramienta de migración de datos de verdad para traer el historial de Firebird.
2. **Autenticación** — porque no puedes probar nada como usuario real hasta que puedes iniciar sesión como uno.
3. **La aplicación** — registro diario, lista de alimentos, vista de nutrición.
4. **Las gráficas** — señaladas explícitamente como funcionalidad de primera clase, porque las gráficas eran el corazón de la aplicación antigua.
5. **Deploy y documentación** — llevarla a un VPS con scripts repetibles, y documentarlo todo.



Las cinco fases acordadas, en orden

De manera crucial, al inicio de cada fase la IA **se detenía y me hacía preguntas** — preguntas reales, con forma de decisión, no relleno del tipo "¿continúo?". ¿Qué estrategia de hashing para las contraseñas legacy? ¿Dónde debe vivir el estado de la migración? ¿Cómo manejo una rareza en los datos? Esas preguntas son el punto donde mi experiencia se inyecta en el resultado de la máquina. Las iré señalando sobre la marcha.

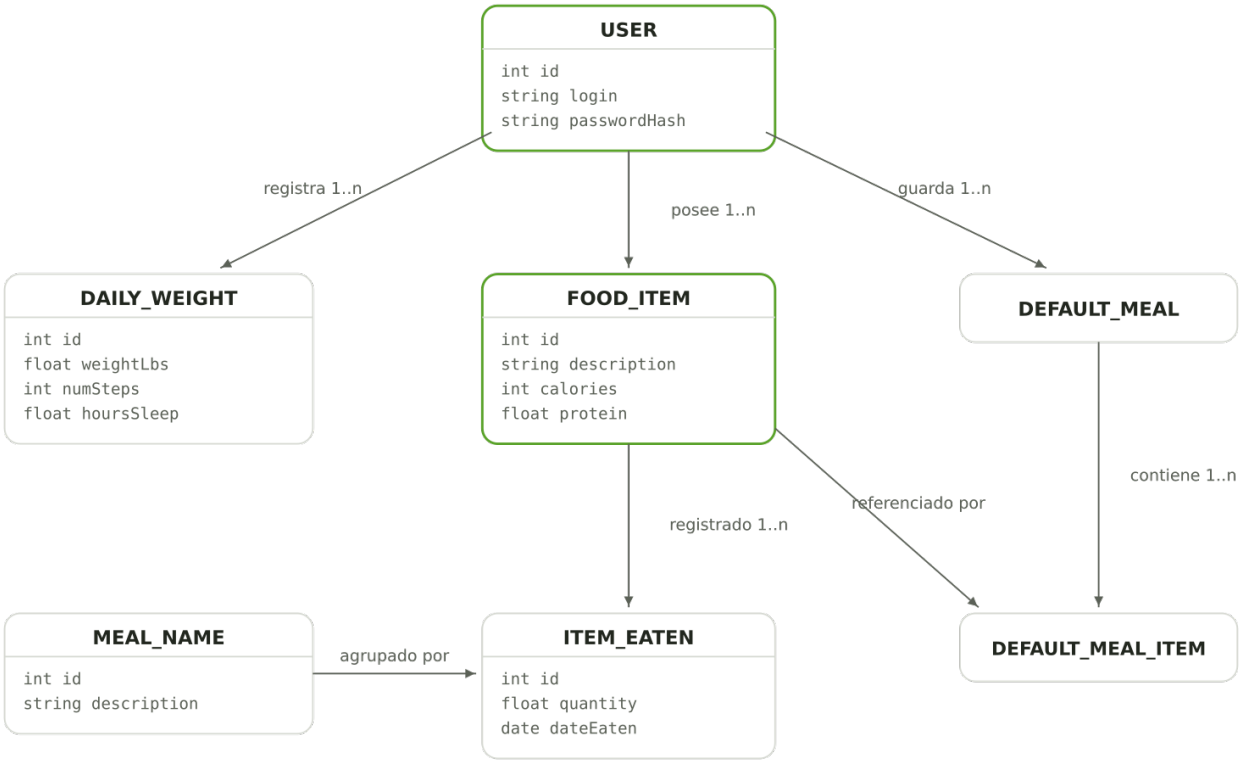
Fase 1 — El esquema y la gran migración de datos

Leer el mundo antiguo

Lo primero que hizo la IA fue *leer*. No solo el script `CREATE` de Firebird, sino las implementaciones de los servicios

XData en Pascal — porque el esquema por sí solo no te cuenta la lógica de negocio. Los procedimientos almacenados, el SQL dentro de los servicios Delphi, la manera en que realmente se calculaban las "calorías de un día" — todo eso vive en el código del backend, y todo tenía que entenderse antes de escribir un solo modelo Prisma.

Siete tablas cruzaron conceptualmente:



Las siete tablas y sus relaciones, tal como pasaron al nuevo esquema

Corrección n.º 1: "no te limites a poner las columnas en mayúsculas"

Aquí fue el primer lugar donde intervine. El esquema legacy usaba la convención a gritos de Firebird — `PERMISSION_STRING`, `DATE_EATEN`, `NUM_STEPS`, `DESCR`. La primera pasada del esquema Prisma estaba trasladando esos nombres fielmente.

Lo detuve: **usa nombres idiomáticos de TypeScript.** `DESCR` → `description`. `LBS` → `weightLbs`. `NUM_STEPS` → `numSteps`. El esquema debe leerse como código moderno, no como un volcado de base de datos de 2010. La IA regeneró el esquema completo con nombres de campo limpios en camelCase y relaciones bien definidas. Cosa pequeña, pero es la diferencia entre un código que un desarrollador nuevo disfruta y uno que tolera.

El problema de las contraseñas (y una decisión que solo debe tomar un humano)

Luego vino una pregunta genuinamente interesante. El backend XData antiguo verificaba las contraseñas con la función de hash incorporada de Firebird:

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

¿Podíamos reproducir eso en Node — es decir, **cero restablecimientos de contraseña para los usuarios existentes** — o teníamos que borrar las contraseñas y forzar a todos a restablecerlas?

La IA no adivinó. **Se conectó a la base de datos Firebird en vivo, extrajo el hash de contraseña almacenado de un usuario real y demostró el algoritmo** hasheando una contraseña de prueba conocida en Node y comparándola byte a byte:

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

// SHA-512, hex, mayúsculas — exactamente lo que emite CRYPT_HASH de Firebird.
const candidate = createHash("sha512").update(password).digest("hex").toUpperCase();
const matches = candidate === storedHashFromFirebird;
```

Confirmado: era un SHA-512 plano, sin sal. Totalmente reproducible.

Eso convirtió una pregunta aterradora en una decisión limpia, que después me planteó a mí:

> **La propuesta que aprobé**
> Verificar contra el hash SHA-512 legacy en el login y **actualizar de forma transparente a cada usuario a bcrypt moderno en su primer inicio de sesión exitoso.** Sin restablecimientos. Sin interrupciones. El hashing antiguo y débil se retira solo, un login a la vez.

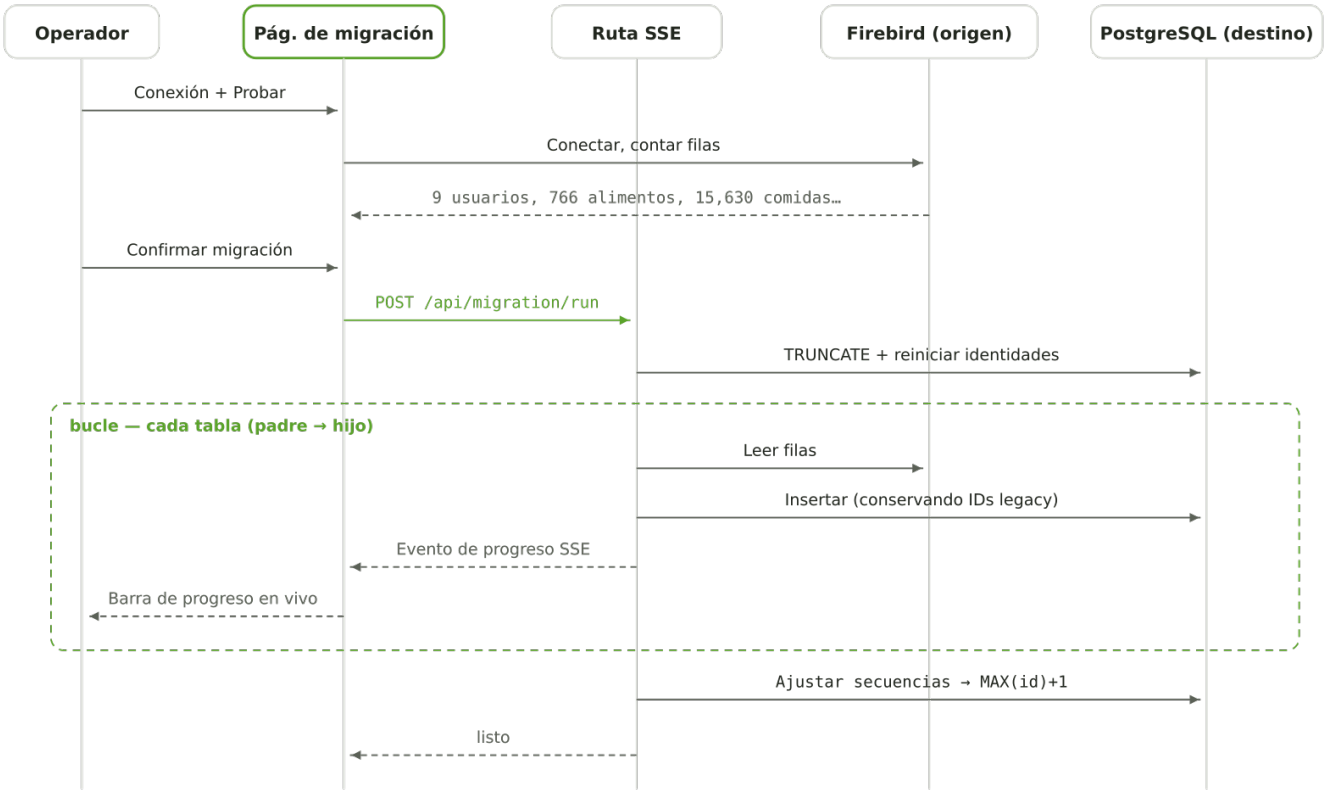
Lo aprobé. Ese es el patrón que quiero en una migración: nadie recibe un correo diciendo "por favor restablece tu contraseña porque cambiamos de backend". Simplemente inician sesión y, tras bambalinas, su seguridad mejora en silencio. La IA lo propuso; mi trabajo fue reconocerlo como la decisión correcta y darle luz verde.

La herramienta de migración en sí

Esta es la parte de la que estoy más orgulloso, porque es donde "la IA escribió algo de código" se convierte en "la IA construyó una herramienta".

En lugar de un script desechable, construimos una **página de migración** de verdad — fuera de la autenticación (tiene que estarlo; es la que *crea* los primeros usuarios) — con:

- Un formulario para introducir los datos de conexión de la Firebird de origen, con un botón **"Probar conexión"** que reporta recuentos de filas en vivo antes de comprometerte.
- Un **historial** de conexiones para que no vuelvas a teclear credenciales.
- Un diálogo de confirmación estricto, porque esto borra y recarga el destino.
- **Server-Sent Events transmitiendo el progreso en vivo** — una barra por tabla — mientras miles de filas cruzan.



La ejecución de la migración: probar, confirmar y transmitir el progreso por tabla vía SSE

Por debajo hay siete migradores ordenados (padres antes que hijos, para que las claves foráneas nunca se rompan), cada uno reportando su propio recuento:

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

Todo el proceso corrió contra la base de datos en vivo y movió **9 usuarios, 766 alimentos, 15.630 comidas registradas y 3.409 entradas de peso con cero filas omitidas.**

Corrección n.º 2: el ruido de coma flotante

Después de la migración, miré la aplicación en marcha y algo me molestó: una cantidad que debía leerse `1.1` aparecía como `1.100000023841858`. Un peso mostraba `252.60000610351562`. Feo.

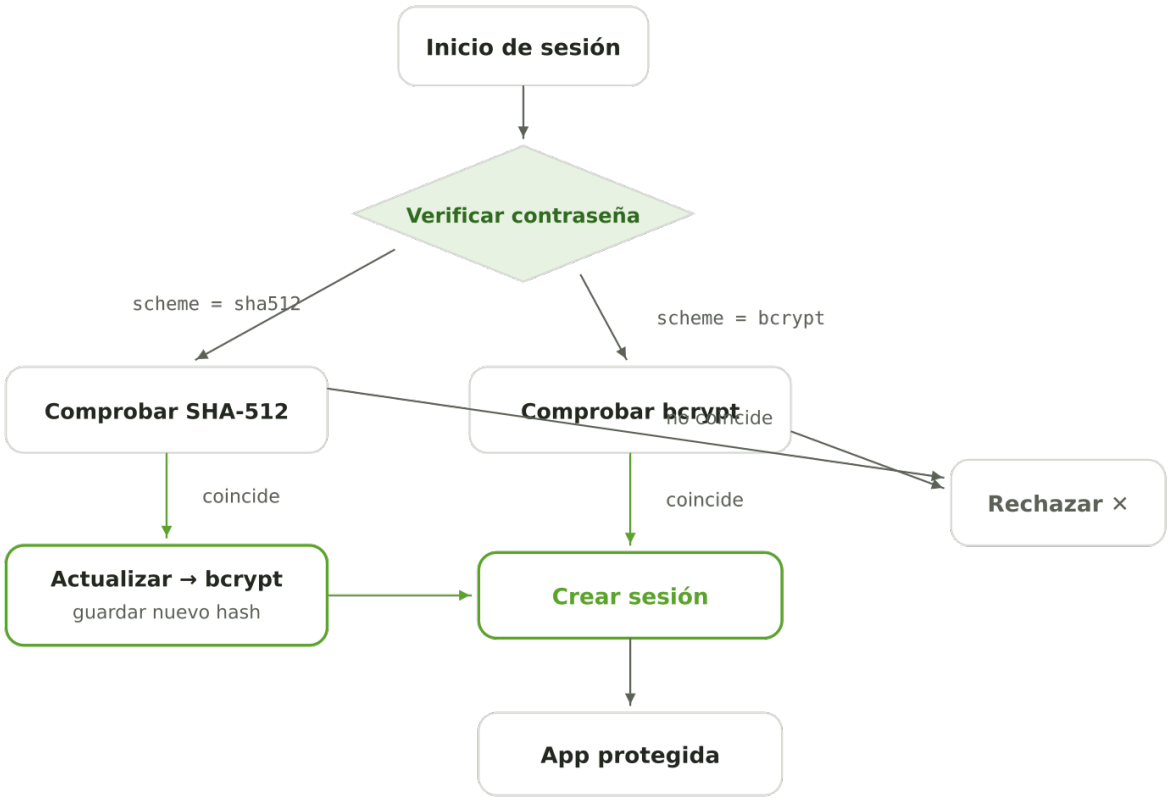
Lo señalé — *"la precisión parece rara, ¿es redondeo o almacenamiento?"* — y la IA lo diagnosticó correctamente: las columnas antiguas de Firebird eran floats de 32 bits; promovidas a los dobles de 64 bits de PostgreSQL, el ruido de precisión simple se vuelve visible. Ofreció tres arreglos y elegí el más limpio: **redondear los valores en el momento de la migración**, de modo que los datos almacenados queden limpios en sí mismos y el arreglo sea reproducible en cualquier re-ejecución futura.

Ya que estábamos ahí, señalé un segundo problema, más sutil: los totales de calorías por comida no siempre cuadraban con la suma de las líneas visibles (redondear la suma bruta vs. sumar los elementos redondeados — una unidad de diferencia, y *parece* incorrecto aunque matemáticamente esté bien). Arreglamos los totales para que se reconcilien con lo que el ojo suma. Ese es el tipo de bug que una herramienta automatizada pasa de largo y que un humano que *usa el producto* detecta al instante.

Cimientos listos
Esquema, base de datos y una herramienta de migración de verdad — con los datos verificados contra la fuente de la verdad.

Fase 2 — Autenticación

Fase corta, fase importante. Usando `iron-session` para sesiones con cookies cifradas, construimos el flujo de login alrededor de la estrategia verificar-SHA-512-y-actualizar-a-bcrypt de la Fase 1, un layout de rutas protegidas que redirige a los visitantes no autenticados y una página de inicio de sesión limpia.

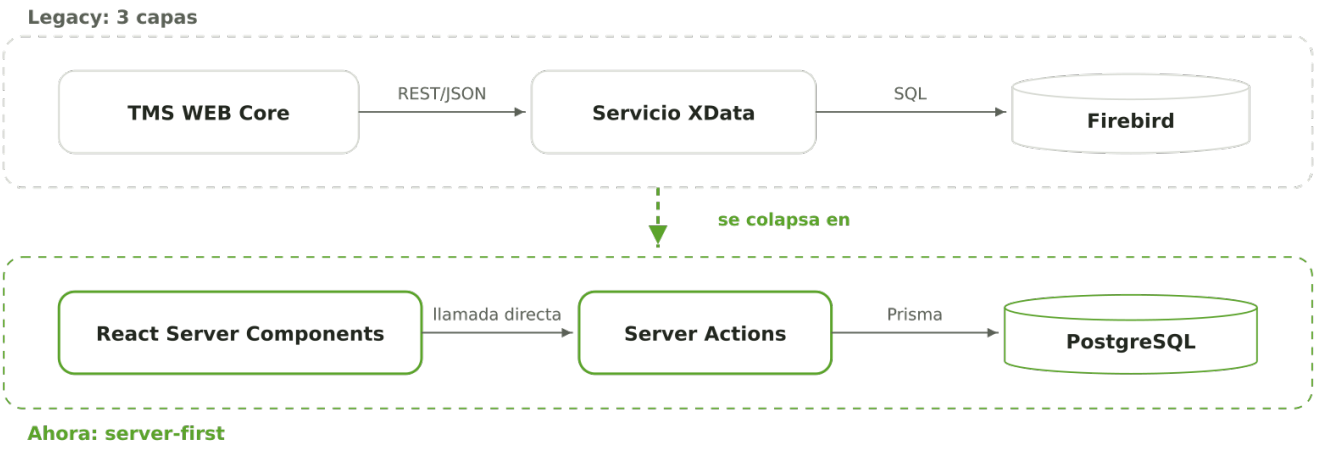


Flujo de login: verificar SHA-512 legacy o bcrypt, actualizando en silencio los hashes antiguos al acertar

La verificación que importó aquí no fue visual — fue demostrar que el login *realmente funcionaba* contra una cuenta migrada. La IA selló una sesión real, inició sesión como un usuario migrado genuino, confirmó que la actualización a bcrypt se disparó en el primer login y confirmó que una contraseña incorrecta era rechazada. Después restauró la cuenta a su estado recién migrado para no dejar nada alterado.

Fase 3 — Reconstruir la aplicación

Aquí es donde la vieja arquitectura de tres capas se colapsó en algo dramáticamente más simple. Sin backend REST separado. Sin capa de API que diseñar, versionar y asegurar. Los **Server Actions de Next.js** permiten que la interfaz llame directamente a funciones del lado del servidor, con el acceso a la base de datos, las comprobaciones de propiedad y la validación viviendo todos en el servidor.



Tres capas legacy se colapsan en una arquitectura Next.js centrada en el servidor

El registro diario

La pieza central. Una vista de día con navegador de fechas, una tarjeta por comida (desayuno, almuerzo, cena, los snacks), los elementos registrados en cada una con el cálculo de calorías en vivo, y un panel de peso/pasos/sueño/composición corporal.

Aquí tomé una decisión de diseño explícita que la IA luego ejecutó: **modernizar el modelo de interacción.** La aplicación antigua era una página por formulario — navegar a una pantalla separada para editar una comida, otra para editar el peso. Reemplazamos todo eso con **diálogos modales y edición en línea** desde una única vista de día. Añades un elemento con un selector de alimentos con búsqueda, respaldado por el servidor, con vista previa de calorías en vivo. Editas una cantidad en el sitio. Ajustas el peso sin salir de la página. Los mismos datos, una sensación de 2026.

Cada mutación lleva una **guarda de propiedad** en el servidor — un usuario solo puede tocar sus propios datos — reemplazando las comprobaciones de claims JWT que hacían los viejos servicios XData.

La lista de alimentos

CRUD completo sobre los alimentos con toda su información nutricional, como tabla responsive en escritorio y tarjetas en móvil (la paridad móvil fue regla estricta — nada de funcionalidad oculta en pantallas pequeñas). Con búsqueda, paginada. Y una **guarda de borrado** genuinamente considerada: no puedes eliminar un alimento referenciado por comidas registradas — la app te dice que se usa en *N* comidas en lugar de fallar crípticamente o dejar huérfanos tres años de historial.

La vista de nutrición

Un desglose de macros por día — proteína, carbohidratos, fibra, carbohidratos netos, azúcares, azúcares añadidos, grasa, grasa saturada, colesterol, sodio, agua — sobre un rango de fechas seleccionable, con promedios diarios, reproduciendo fielmente el viejo procedimiento almacenado `GET_DAILY_NUTRITION_STATS` como una agregación SQL limpia.

Durante toda esta fase la verificación fue continua: la IA inició sesión como un usuario migrado real y confirmó que el total de un día renderizado coincidía con el agregado de la propia base de datos **hasta la caloría exacta**:

```
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 — Las gráficas

Las gráficas no eran negociables. En la aplicación antigua eran la recompensa — la prueba visual de meses de esfuerzo. Los datos del cliente cuentan una historia genuinamente buena (el viaje de peso de varios años de un usuario), y había que contarla bien.

Nos quedamos con **Chart.js**, deliberadamente, porque se porta limpiamente a React y da control real. La capa de datos reprodujo la lógica legacy con exactitud, incluido un detalle que insistí en respetar fielmente: las medias móviles de "14 días" y "50 días" de la app antigua en realidad se calculaban como **ventanas basadas en filas** sobre pesajes consecutivos, no en días de calendario. Lo igualamos, para que los usuarios que regresan vean las mismas curvas que siempre vieron. Gráfica de peso, gráfica de calorías y las tarjetas de estadísticas de pérdida de peso (inicial vs. actual, cambio total, ritmo por semana/mes).

Correcciones n.º 3 y n.º 4: las gráficas que se resistieron

Las gráficas no salieron bien a la primera. Vale la pena mostrarlo, porque es la ilustración más honesta del ciclo.

> **Bug #1 — las gráficas vacías**
> Las gráficas se renderizaban *vacías* — el eje X mostraba "12AM, 1AM, 2AM…" en lugar de fechas que abarcaran 2023–2026. Envié una captura de pantalla: *"¿¡las gráficas están vacías!?"* La IA lo diagnosticó de inmediato: estaba alimentando **cadenas** de fecha a un eje temporal que, con el parseo desactivado, esperaba timestamps numéricos. Los 1.173 puntos habían colapsado en un solo día.

El arreglo fue entregarle al eje timestamps reales en lugar de cadenas:

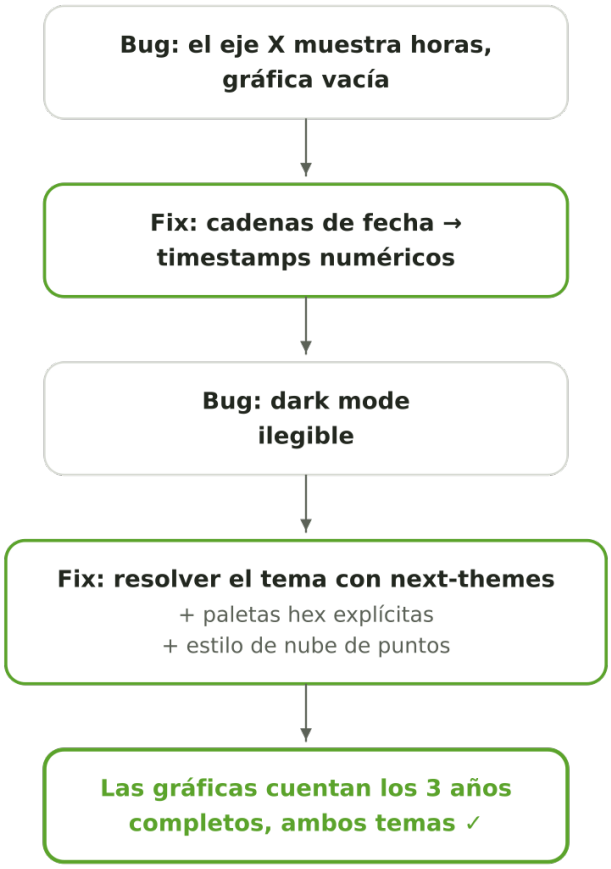
```
// Antes: las cadenas de fecha colapsan en un solo punto con el parseo desactivado.
const data = rows.map((r) => ({ x: r.dateEaten, y: r.weightLbs }));

// Después: los milisegundos epoch numéricos son lo que un eje temporal realmente espera.
const data = rows.map((r) => ({ x: new Date(r.dateEaten).getTime(), y: r.weightLbs }));
```

> **Bug #2 — el dark mode se veía terrible**
> Con los datos por fin visibles, el dark mode era ilegible — el texto de la leyenda casi invisible, las líneas de la cuadrícula turbias. Lo dije sin rodeos: *"las gráficas en dark mode se ven como una m**… resuelve el tema y luego decide los colores."*

La causa raíz era sutil, y un buen ejemplo de algo que un humano nota al instante y una máquina no: los colores del tema de la app se almacenan como variables CSS `oklch()` y expresiones `color-mix()` que un `<canvas>` sencillamente no puede resolver. El arreglo fue detectar correctamente el tema activo y entregarle a la gráfica **paletas de color explícitas y legibles por modo** — además de renderizar la serie diaria cruda como una nube

de puntos translúcida con las medias móviles como líneas limpias encima, lo que quitó el ruido a la ajetreada gráfica de calorías.



Dos bugs en las gráficas, dos correcciones de raíz y, al fin, gráficas que cuentan la historia

Dos iteraciones, ambas iniciadas por mí mirando la pantalla realmente renderizada y diciendo "no". Ese es el trabajo. La IA es asombrosamente rápida produciendo y arreglando; *no* es un sustituto de alguien con criterio que mire el resultado y decida si es lo bastante bueno. Ese alguien sigue siendo, muy claramente, una persona.

La ronda extra: cero dependencia de terceros, dark mode, deploy, docs

Alrededor y después de las fases principales pasaron algunas cosas que merecen mención, porque son donde "funciona en mi máquina" se convierte en "es un producto de verdad".

- Se introdujeron el **modo oscuro y claro** limpiamente en toda la aplicación con un conmutador de tema — el tipo de cosa que toca cada componente y que es un suplicio adaptar a mano después, hecho de forma consistente.
- **Eliminamos la dependencia de servicios de UI de terceros** — la interfaz está construida sobre componentes que poseemos y controlamos, no sobre un sistema de diseño alquilado. Sin sorpresas de precios, sin dependencia externa en tiempo de ejecución para la interfaz.
- Construimos el **despliegue en VPS con scripts de build y deploy en condiciones** — la app no solo corre en local, se envía a un servidor real de forma repetible.
-

Y para rematar, **documentación** — documentación genuinamente buena, de la que agradeces seis meses después, cuando ya olvidaste cómo funciona el historial de conexiones de la herramienta de migración.

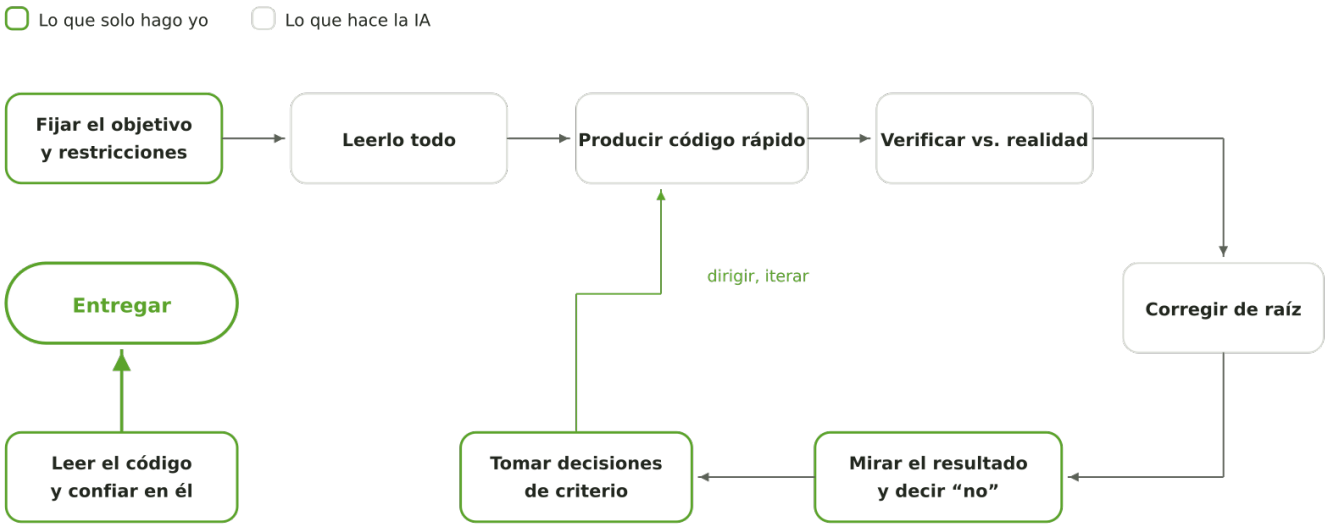
Cada una de estas cosas es, por sí sola, una tarde de trabajo engorroso en un proyecto normal. Aquí fueron incrementos.

Qué hizo esto rápido en realidad

Déjame ser directo sobre de dónde salió la velocidad, porque "lo hizo la IA" es la lección equivocada y no quiero que nadie se la lleve.

La IA escribió el código. Pero **la IA escribió el código* correcto *porque operaba dentro de un sistema que yo construí.** Considera lo que realmente estaba ocurriendo en el trasfondo de cada uno de los pasos anteriores:

- **Leyó las fuentes legacy reales y la base de datos real** antes de proponer nada. No alucinó la vieja lógica de negocio — fue y la confirmó. (Literalmente se conectó a Firebird y cotejó un hash de contraseña para demostrar una suposición.)
- **Me hizo preguntas con forma de decisión** en cada frontera de fase y dejó que *mi* juicio marcara la dirección: convenciones de nombres, estrategia de hashing, dónde modernizar la UX, cómo manejar rarezas en los datos.
- **Verificó su propio trabajo contra la fuente de la verdad** constantemente — totales renderizados cotejados con un `SELECT SUM(...)`, viajes de ida y vuelta CRUD demostrados, logins realmente ejecutados — no "a mí me parece que está listo".
- Y cuando produjo algo incorrecto o feo — los nombres en mayúsculas, el ruido de los floats, las gráficas vacías, el dark mode ilegible — **yo lo detecté y lo corregí,** y arregló la verdadera causa raíz en lugar de tapar el síntoma.



La división del trabajo: el juicio humano dirige un ciclo rápido de la IA de producir y verificar

Nada de eso funciona sin un operador que sepa leer Prisma, detectar un descuadre de una unidad en un total de calorías, reconocer a simple vista un artefacto de almacenamiento en coma flotante y saber que un SHA-512 sin sal debe actualizarse en silencio en lugar de conservarse. **La IA es un multiplicador fenomenal**

de la experiencia. No es un sustituto de ella. Apúntala a un problema difícil sin esa experiencia y obtienes resultados seguros de sí mismos, plausibles y sutilmente erróneos a una velocidad aterradora.

Esa es la habilidad que llevo seis meses construyendo. No "prompting". Juicio, a la velocidad de la generación.

El resumen que de verdad quiero que recuerdes

Una aplicación real, en producción — frontend Delphi TMS WEB Core, backend REST TMS XData, base de datos Firebird, tres años de datos vivos de usuarios — fue reconstruida sobre Next.js 16, React 19, Prisma 7 y PostgreSQL. Nuevo esquema con nombres limpios e idiomáticos. Una herramienta de migración de datos en vivo, con streaming, que movió cada fila sin pérdida alguna y sin restablecer contraseñas. Autenticación de sesión moderna. Un backend de server actions sin capa REST separada. Un registro diario, una lista de alimentos y una vista de nutrición, todo responsive y basado en diálogos. Cada gráfica recreada en Chart.js con la matemática legacy preservada. Modo oscuro y claro. Sin dependencia de UI de terceros. Scripts de deploy para VPS. Y documentación sólida.

Semanas de trabajo — realistamente un mes o más hecho a mano con cuidado — entregadas en unas cuatro horas.

Pero escucha la lección de verdad, porque es la que importa para los próximos años de esta profesión: **las cuatro horas solo son posibles gracias a los seis meses de aprender y acumular experiencia.** La productividad es real y es apabullante, pero la *desbloquea* un experto capaz de manejar las herramientas, evaluar el código generado con ojo crítico, tomar las decisiones de juicio que una máquina no debería tomar y dirigir los prompts directo al objetivo. Saca al experto de ese ciclo y la velocidad no produce una migración — produce un desastre muy rápido.

Estamos, genuinamente, en un lugar nuevo. Por primera vez, la *experiencia y el criterio* de un desarrollador pueden aplicarse a la velocidad del pensamiento en lugar de a la velocidad de la escritura. El cuello de botella se movió de "qué tan rápido puedo escribirlo" a "qué tan bien puedo dirigirlo y juzgarlo". Eso no es una amenaza para los buenos ingenieros. Es el mejor día que los buenos ingenieros han tenido jamás.

Estoy más entusiasmado con construir software de lo que he estado en años. Cuatro horas para un mes de trabajo te hacen eso.

A 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 —