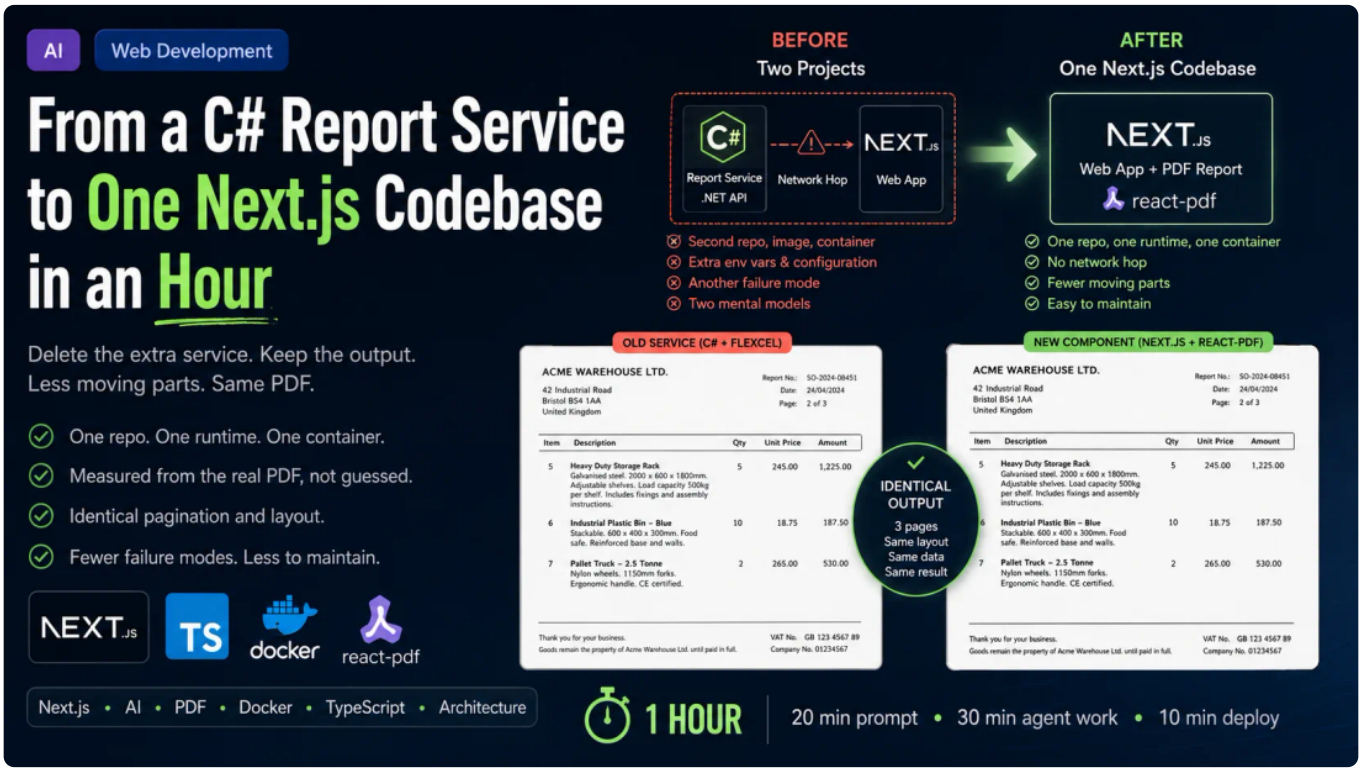
NEWS · AUGUST 17, 2026

Von einem C#-Berichtsservice zu einer einzigen Next.js-Codebasis in einer Stunde

Ein PDF-Bericht lebte in einem eigenen C#-Service mit Excel-Vorlage, weil ein handgebautes PDF vor einem Jahr die teure Variante war — was dieses zweite Projekt wirklich gekostet hat, und wie ein Nachmittag mit einer KI es in die Next.js-Anwendung geholt hat, in die es gehörte.

By Dr. Holger Flick



Es gibt einen Deployment-Fehler, den man nur einmal macht. Man rollt die Anwendung aus, sieht alles grün werden, klappt den Laptop zu — und am nächsten Morgen sagt jemand, der Drucken-Knopf melde, der Berichtsserver sei nicht verfügbar. Die Anwendung war in Ordnung. Das *andere* Ding nicht.

Rund ein Jahr lang lebte ein PDF-Bericht einer Lagerverwaltung, die ich betreue, in einem eigenen Service: eine kleine C#-Web-API, die ein JSON-Paket entgegennahm, es in eine Excel-Vorlage goss und ein PDF zurückgab.

Sie funktionierte. Sie ist nie abgestürzt, hat nie eine Seite zerschossen, brauchte nie einen Fix. Und ich habe sie gerade gelöscht — weil korrekt zu funktionieren die kleinere Hälfte dessen ist, was Software einen kostet.

Möglich gemacht hat das Löschen keine neue Bibliothek. Die Bibliothek, die ich benutze, gibt es seit Jahren, und ich hatte sie für genau diese Aufgabe schon einmal verworfen — aus damals guten Gründen. Geändert hat sich, dass der teure Teil daran — eine Seite auf den Punkt genau auszumessen — heute etwas ist, das ich zusammen mit einem Referenz-PDF an eine KI übergeben und nach einer halben Stunde zurückbekommen kann.

Warum es überhaupt zwei Projekte gab

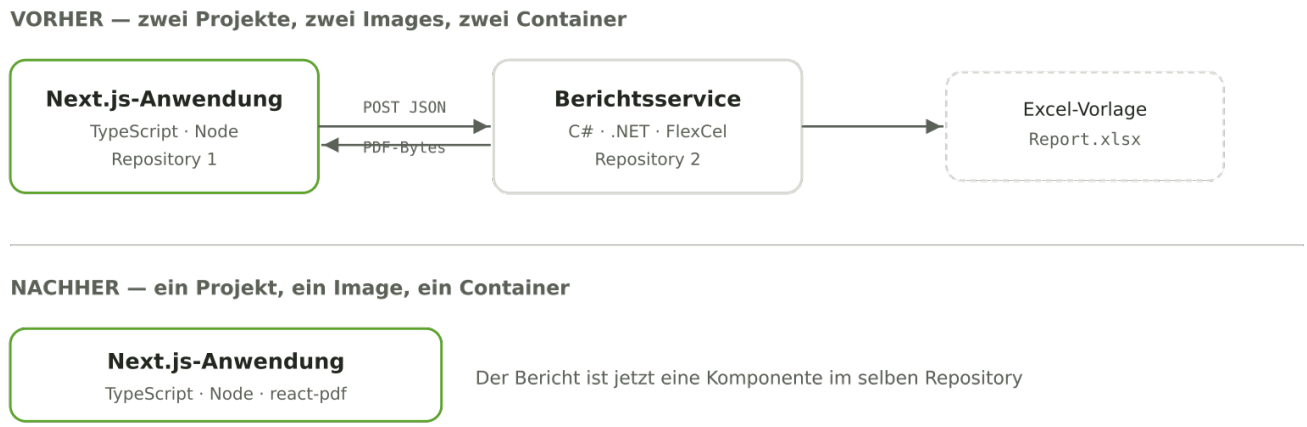
Die ehrliche Antwort: vor einem Jahr war die Trennung die günstigere Entscheidung, und ich würde sie mit demselben Wissensstand wieder so treffen.

Der Bericht ist ein dichtes Dokument: Briefkopf, Adressblock, ein Raster mit Kopfdaten und dann eine Tabelle von Positionen, deren Beschreibungen je zwanzig Zeilen lang werden und über so viele Seiten laufen, wie sie eben brauchen. Das in einem Web-Stack zu bauen hiess eines von zwei Dingen. Entweder ich bezahle eine kommerzielle Reporting-Engine, oder ich baue das Layout von Hand — und in der JavaScript-Welt heisst von Hand react-pdf, ein MIT-lizenzierter React-Renderer, der PDFs mit flexbox-ähnlichen Primitiven zeichnet statt mit HTML.

react-pdf ist eine wirklich schöne Bibliothek. Aber darin zu *entwerfen* ist etwas anderes, als sie zu *benutzen*. Man positioniert Kästen in Punkt auf einer Fläche von 595 × 842, und jede Spaltenbreite, jeder Zeilenabstand, jeder Rand ist eine Zahl, auf die man irgendwie kommen muss. Es gibt keinen Designer. Es gibt kein Lineal. Es gibt dich, eine `pnpm dev`-Schleife und viel Blinzeln auf eine Seite, die *fast* stimmt.

Dagegen war die Alternative fast unfair. TMS FlexCel ist eine Komponenten-Suite, deren Funktionsliste eine "Report Engine that allows to create complex reports using Excel as your report designer" und "100% Native PDF report generation from .XLS/.XLSX files" nennt — und neben der Delphi-Variante gibt es eine vollständig verwaltete .NET-Ausgabe. Das Entwurfswerkzeug war also Excel. Ich habe den Bericht in einer Tabelle gezeichnet, Platzhalter wie `<#Title>` und `<#I.Text>` in die Zellen geschrieben, und FlexCel hat sie aus einer Objektliste gefüllt und das Ergebnis als PDF exportiert. Ein Nachmittag Arbeit, und das Layout war Drag-and-drop.

Der Haken: FlexCel ist .NET, und die Anwendung ist Next.js — das React-Framework, das deine Komponenten auf dem Server ausführt. Zwei Laufzeitumgebungen, also zwei Projekte.



Der Bericht brauchte eine zweite Laufzeitumgebung — also bekam er einen zweiten Container, ein zweites Image und ein zweites Deployment

Was das zweite Projekt tatsächlich gekostet hat

Die laufenden Kosten dieses Services lagen nahe null — und genau deshalb habe ich ein Jahr gebraucht, um zu merken, was er wirklich in Rechnung stellte.

Nichts auf dieser Liste ist ein *Fehler*. Jeder einzelne Punkt ist eine kleine, dauerhafte Steuer auf Aufmerksamkeit:

- Ein **zweites Repository** zum Klonen, Branchen und Zugangsdaten-Pflegen.
- Eine **zweite Sprache samt Werkzeugkette** — das .NET-SDK auf einer Maschine, die sonst nur Node braucht.
- Ein **zweites Container-Image**, für zwei Architekturen zu bauen und in die Registry zu schieben.
- Ein **zweiter Service** in der `docker-compose.yaml`, mit `depends_on` und eigener Restart-Policy.
- Eine **Umgebungsvariable**, die das eine auf das andere zeigen lässt und in jedem frischen Checkout erst einmal falsch ist.
- Ein **Fehlerfall, der mit dem Bericht nichts zu tun hat** — der Netzwerksprung dazwischen — plus der Zweig in der Oberfläche, der ihn erklärt: *Der Berichtsserver ist nicht verfügbar, bitte versuchen Sie es später erneut.*
- Und das, was sich nicht aufzählen lässt: **zwei mentale Modelle**, von denen um 23 Uhr nur eines geladen ist.

Das Letzte ist die eigentliche Rechnung. Der Bericht war nie schwer. Sich zu merken, dass der Bericht ein *separates Ding* ist, war schwer.

> **Die andere Seite**
> Ein eigener Service ist kein Fehler, und viele Teams sollten ihren behalten. Er skaliert unabhängig, lässt sich neu ausrollen, ohne die Anwendung anzufassen, und ein Absturz im Renderer kann die Web-Anwendung nicht mitreissen. Wenn deine Berichte CPU-lastig sind, in Massen erzeugt werden oder von mehreren Anwendungen genutzt werden, ist diese Isolation bares Geld wert. Mein Fall war das Gegenteil: ein Bericht, ein Aufrufer, ein paarmal am Tag erzeugt, von einem Entwickler. Die Naht hat mir nichts gebracht und täglich etwas gekostet.

Was ich der KI gegeben habe

Der ganze Job begann mit drei Artefakten und ohne jede Erklärung, wie sie zusammengehören.

Ich habe Claude Code — Anthropics Coding-Agent, der im Terminal, in der IDE und im Browser läuft — genau das gegeben:

1. Den **C#-Quellcode**, alles in allem etwa siebzig relevante Zeilen: den Request-Handler, die Klasse, die die Berichtswerte gesetzt hat, und den Aufruf, der die Arbeitsmappe als PDF exportierte.
2. Die **Excel-Vorlage**, die `.xlsx`-Datei selbst, mitsamt Platzhaltern.
3. **Ein PDF**, das der alte Service tatsächlich erzeugt hatte, für einen echten Auftrag.

Die dritte Datei ist die entscheidende, und das ist der Teil, den ich jedem zum Nachmachen empfehlen würde. Der Quellcode verrät die *Daten*. Die Vorlage verrät die *Absicht*. Nur das erzeugte PDF verrät die *Wahrheit* — was die Engine aus der Vorlage gemacht hat, nach all ihren eigenen Rundungs-, Skalierungs- und Seitenumbruch-Entscheidungen.

Drei Eingaben — und wofür jede davon wirklich gut war

Messen statt raten

Das ist der Schritt, den ich von Hand nicht an einem Tag geschafft hätte — und der ganze Grund, warum die Migration eine Stunde gedauert hat und nicht zwei Wochen.

Der Agent hat die `.xlsx` auseinandergenommen — eine Excel-Datei ist ein ZIP-Archiv voller XML — und die Geometrie des Arbeitsblatts direkt gelesen. Die Vorlage entpuppte sich als gleichmässiges Raster: zweiund-dreissig Spalten, jede exakt 2,5625 Einheiten breit, mit der Standardschrift der Arbeitsmappe in 11pt. Jede ver-bundene Zelle im Layout liess sich damit als Spanne von Rasterspalten ausdrücken — genau die Information, die ein Flexbox-artiger Renderer braucht.

Dann kam der Teil, den ich wirklich schön finde. Der Agent hat das Referenz-PDF mit pdfplumber geöffnet, einer

Python-Bibliothek, die die Position jedes Wortes und jeder Linie auf einer Seite meldet, und die Koordinaten schlicht *vom fertigen Dokument abgelesen*:

```
x0=51.8  top=57.6   size=8.51   Absenderzeile über der Adresse
x0=51.8  top=162.2  size=10.40  Adressblock, erste Zeile
x0=51.8  top=231.4  size=13.24  Überschrift des Berichts
         top=245.3            Trennlinie, x 50.4 ' 540.1
x0=51.8  top=261.1  size=10.40  Kopfdatenzeile
x0=51.8  top=312.6  size=10.40  Tabellenkopf (wiederholt sich auf jeder Seite)
```

Aus diesen Zahlen wurde eine Konstantendatei. Keine Schätzung, die man mit zwanzig Screenshot-Vergleichen nachjustiert — die tatsächlichen gedruckten Positionen des Dokuments, das ich ersetzt habe, abgeschrieben.

Die eigentliche Überraschung in diesem Auszug sind die Schriftgrössen. Es sind keine runden Zahlen: 13,24 — 11,35 — 10,40 — 8,51 — 7,57. Die Reporting-Engine hatte das gesamte Blatt um etwa 94,5 % skaliert, damit es in den Druckbereich passt, und niemand hatte das je wissen müssen. Es aus der Ausgabe abzulesen machte es kostenlos.

Der Schrifttrick, der die Zeilenumbrüche gerettet hat

Ein Schriftwechsel hätte jeden Zeilenumbruch im Dokument zerstören müssen — und die Lösung ist ein einziges Verhältnis.

Die Vorlage war in Consolas gesetzt, einer Microsoft-Schrift, und das Repository des Berichtsservices trug vier Consolas-`.ttf`-Dateien neben dem Code mit sich, damit der Renderer sie unter Linux einbetten konnte. Microsofts eigene FAQ zur Weitergabe von Schriften ist dazu unmissverständlich — *"Apart from the document embedding rights described previously, you may not redistribute the Windows fonts. You may not copy them to*

other computers or servers" —, womit das Kopieren auf einen Server genau die Sache ist, die dort ausdrücklich benannt wird. Das war mein Versäumnis, ein Jahr lang mitgeschleppt, und ich hätte nie danach gesucht. Erst das Neuschreiben hat mich hinschauen lassen.

Der Ersatz ist JetBrains Mono, veröffentlicht unter der SIL Open Font License 1.1, und die Schrift deckt jedes Zeichen ab, das die Daten hergeben — die Umlaute, das ®, das ø, das eAber eine dicktengleiche Schrift ist über eine einzige Zahl definiert, ihre Dickte, und die beiden sind sich nicht einig: Consolas rückt 0,55 em pro Zeichen vor, JetBrains Mono 0,60 em. Gleiche Punktgrösse, neun Prozent breiter. Jede umbrochene Zeile in jeder Beschreibung wäre woanders gebrochen, und die Seitenzahl wäre verrutscht.

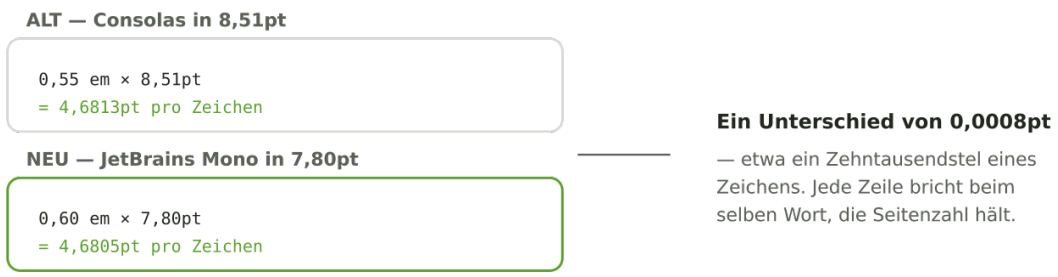
Also behält man nicht die Punktgrösse, sondern die *physische* Zeichenbreite — indem man die Grössen mit dem Kehrwert des Verhältnisses skaliert:

```
// Consolas rückt 0,55em pro Zeichen vor, JetBrains Mono 0,60em. Skaliert man
// die Punktgrössen mit dem Kehrwert, bleibt die Anzahl Zeichen pro Zeile
// gleich – die Beschreibungen brechen wie bisher, und der Bericht paginiert
// identisch.
const CONSOLAS_TO_JBM = 0.55 / 0.6;

const consolas = (size: number) => size * CONSOLAS_TO_JBM;

export const SIZE_HEADING = consolas(13.24); // Überschrift des Berichts
export const SIZE_BODY = consolas(10.4); // Adresse, Kopfdaten, Tabellenkopf
export const SIZE_TABLE = consolas(8.51); // Positionszeilen
```

Nebeneinandergestellt kommen die beiden Schriften von entgegengesetzten Seiten auf dieselbe Zeichenbreite:



Andere Schrift, andere Punktgrösse, identische Zeichenbreite — und davon hängt die Paginierung tatsächlich ab

Das Ergebnis, mit demselben Werkzeug wieder aus dem neuen PDF herausgemessen, deckt sich fast überall mit dem Original:

	Alter Service	Neue Komponente
Seiten	3	3
Positionen pro Seite	1–4 / 5–9 / 10	1–4 / 5–9 / 10
Seitenzahl, von oben	23,4pt	23,4pt
Adressblock	162,2pt	162,2pt
Tabellenkopf	312,6pt	312,4pt
Fusszeilen	803,1 / 812,7pt	803,1 / 812,7pt
Vorschub pro Zeichen	4,6813pt	4,6805pt

Diese Tabelle habe ich nicht angefordert. Ich habe gefragt, ob die Ausgabe übereinstimmt, und statt einer Meinung den Beleg bekommen — und das ist genau die Antwort, die man haben will, wenn man gleich einen Service löscht, der ein Jahr lang funktioniert hat.

Was das Neuschreiben in den Daten zutage förderte

Zwei Dinge kamen ans Licht, die ein Jahr lang wahr und ein Jahr lang unsichtbar waren — beide, weil die KI auf echte Daten geschaut hat statt auf das Schema.

Die Positionsbeschreibungen sind kein reiner Text. Alle 3.011 Zeilen in der Datenbank enthalten HTML, weil der Editor des ERP das so speichert und die alte Engine einen "HTML-Modus" gesetzt hatte, der das stillschweigend geschluckt hat. Eine Auswertung der tatsächlichen Spalte fand ein kleines, geschlossenes Vokabular und eine Menge Unordnung:

Tag	Vorkommen	
` `	14.571	Zeilenumbrüche
`<strong>`	3.727	Produktnamen und Zwischenüberschriften
`<span style=…>`	157	aus Word eingefügt — `mso-*`, Calibri, Farben
`<em>`	128	Kursivsatz
`<li>` / `<ul>`	73 / 15	Aufzählungen

Dazu HTML-Entities im Tausenderbereich: `&uuml;` 5.890-mal, `&auml;` 2.591-mal, `&reg;` 1.385-mal. Hätte ich das von Hand aus dem C#-Quellcode portiert, hätte ich `text` in ein `<Text>`-Element geschrieben, ausgeliefert — und von einem Benutzer erfahren, dass jede Beschreibung im System jetzt ihr eigenes Markup druckt. Stattdessen bekam das Markup einen richtigen kleinen Parser — verschachtelte Tags, leere Tags, verirrte Leerzeichen inklusive —, bevor eine einzige Zeile Layout-Code geschrieben war.

> **Wo sich ein echter Parser auszahlt**
>
> Das Vokabular umfasst sechs Tags — genau die Grösse, bei der ein regulärer Ausdruck vernünftig wirkt und es nicht ist. In den Daten steckten `<strong>` in `<strong>` verschachtelt, leere `<strong></strong>`-Paare und Zeilenumbrüche zwischen Block-Tags. Ein geparster Baum verkraftet alle drei ohne einen einzigen Sonderfall; Mustererkennung verkraftet keinen davon.

Keine Engine — nur ein Bericht

Das ist die Verschiebung, die sich meiner Meinung nach über mein eines Dokument hinaus verallgemeinern lässt, und sie gehört klar ausgesprochen.

Eine Reporting-Engine existiert, weil massgeschneiderter Layout-Code früher teuer war. Das ist ihre gesamte wirtschaftliche Rechtfertigung: Man lernt *einmal* einen Vorlagendialekt, einen Designer und eine Platzhalter-

syntax, um nie wieder eine Koordinate ausrechnen zu müssen. Ein sehr guter Handel, solange der Handbau Tage kostet.

Wenn der Handbau eine Stunde kostet, dreht sich der Handel um. Was ich jetzt habe, ist kein Berichts*werkzeug*

— es ist ein Bericht, ausgedrückt als gewöhnliche React-Komponenten in derselben Codebasis wie alles andere, mit demselben TypeScript, denselben Lint-Regeln, demselben Review-Prozess, demselben Deployment. Es gibt keinen Dialekt. `Strg+Klick` auf `SIZE_TABLE` springt auf die Zeile, aus der die Zahl stammt, samt Kommentar, warum sie so lautet.

Und die Wiederverwendung wird besser, nicht schlechter. Braucht ein zweiter Bericht denselben Briefkopf, wird dieser Briefkopf eine Komponente und wird importiert — also über den Mechanismus, den das Framework ohnehin hat, statt über einen zweiten, den das Reporting-Werkzeug mitbringt. Komposition ist das, wofür React da ist.

> **Wo das nicht gilt**
>
> Ich habe etwas aufgegeben, und es wäre unehrlich, das zu überspringen. Mit der Excel-Vorlage konnte jemand ohne Entwicklerkenntnisse die Datei öffnen, eine Spalte verschieben, den Wortlaut der Fusszeile ändern — und der Bericht änderte sich. Kein Build, kein Deployment, kein ich. Das ist eine echte Fähigkeit, und sie ist weg; das Layout lebt jetzt in TypeScript, und nur eine Entwicklerin oder ein Entwickler kann es verschieben. Den häufigen Fall habe ich abgefedert, indem die redaktionellen Texte (Briefkopfzeilen, Schlussabsatz) in die Einstellungen der Anwendung gewandert sind — aber die *Geometrie* ist jetzt
>
> Code. Wenn eure Berichte regelmässig von den Leuten neu gesetzt werden, die sie benutzen, ist eine Vorlagen-Engine weiterhin die richtige Antwort, und ihr solltet sie behalten. Genauso gilt: Bei achtzig Berichten statt einem hört "bauen wir jeden einzeln" auf, ein Plan zu sein, und ein Designer fängt wieder an, sich zu rechnen.

> **Alternativen, die man kennen sollte**
>
> Nichts hier behauptet, du solltest jedes PDF von Hand bauen. Gotenberg ist eine MIT-lizenzierte Dock-
>
> er-API, die HTML, URLs, Markdown und über hundert Office-Formate via Headless-Chromium und Libre-Office zu PDF rendert — wenn du lieber in HTML und CSS entwirfst, ist das eine starke, kostenlose Option. jsreport ist eine vollständige Reporting-Plattform mit Designer und Vorlagensprache, AGPL-lizenziert mit
>
> einer ausdrücklichen Lizenz-Position für die Anbindung über HTTP. Und die kommerziellen Engines sind
>
> aus gutem Grund kommerziell: Sie bringen Designer, Diagrammbibliotheken und eine ausgereifte Semantik für paginierte Berichte mit, die eine handgebaute Komponente nicht hat. Wähle danach, wie viele Berichte du hast und wer sie bearbeitet — nicht danach, welcher Ansatz moderner klingt.

Was der Betrieb jetzt kostet

Der Betriebsaufwand ist der Teil, den ein Kunde merkt, und er ist in jede Richtung kleiner geworden.

Das PDF wird **auf dem Server** gerendert — `renderToBuffer` liefert Node-Bytes, die direkt aus dem Route-Han-

dler als `application/pdf` hinausgehen. Im Browser wird nichts erzeugt, also gibt es überhaupt keine Brows-

er-Kompatibilitätsfläche: Das Tablet im Lager und die zwölf Jahre alte Maschine im Büro bekommen identische Bytes, und das Drucken verhält sich identisch, weil es dieselbe Datei ist.

Die gesamte Anwendung läuft weiterhin in einem einzigen Alpine-Linux-Container — das Basis-Image liegt bei rund 5 MB, daher der Ruf — auf einem kleinen VPS. Im Image steckt kein Headless-Browser, kein LibreOffice, keine .NET-Laufzeit und kein zweiter Container, der daneben eingeplant werden müsste. Und react-pdf brauchte im Server-Bundle von Next.js exakt null Konfiguration, weil Vercel es bereits in der eingebauten Liste externer Pakete des Frameworks führt.

Die Stunde, ehrlich aufgeschlüsselt

Die Schlagzeilenzahl stimmt, aber die Aufschlüsselung lohnt sich, weil die Verteilung das Interessante ist.

- **20 Minuten für das Schreiben des Prompts.** Der grösste Einzelblock, und das soll er auch sein. Hier entscheidet man, was die KI annehmen darf, was das Referenzartefakt ist und was "fertig" bedeutet.
- **30 Minuten Agentenarbeit.** Den C#-Service lesen, die Vorlage entpacken, die echte Datenbank nach echten Beschreibungen fragen, das Referenz-PDF ausmessen, die Komponenten schreiben und Testausgaben zum Vergleich rendern.
- **10 Minuten Deployment.** Ein Image statt zwei.

Zwanzig dieser sechzig Minuten war ich am Englisch-Tippen. Dieses Verhältnis ist die eigentliche Geschichte dieses Beitrags: Die knappe Ressource ist vom *Schreiben des Codes* zum *Beschreiben des Problems und Bereitstellen der Belege* gewandert. Über diese Verschiebung habe ich schon geschrieben, als eine KI aus einem Zweijahresprojekt einen Wochensprint machte und als ein natives Delphi-Werkzeug an einem Nachmittag entstand — hier ist es dasselbe Phänomen, angewandt auf das Löschen statt auf das Bauen.

Fazit

Der Bericht war nie das Problem, und darin liegt die ganze Lehre.

- **Rechne aus, was ein separates Projekt an Aufmerksamkeit kostet, nicht an CPU.** Ein zweites Repository, Image, Container, eine zweite Umgebungsvariable und ein zweiter Fehlerfall sind je fast gratis und zusammen teuer.
- **Eine Architekturentscheidung kann richtig bleiben, während ihre Voraussetzung verfällt.** Excel als Designer war die richtige Wahl, als Koordinatenrechnen Tage kostete. Prüfe die Abwägungen neu, wenn sich das Werkzeug ändert — nicht nur die, die wehtun.
- **Gib der KI die Ausgabe, nicht nur die Eingabe.** Quellcode und Vorlage beschreiben die Absicht; nur ein erzeugtes Dokument verrät, was tatsächlich gedruckt wurde. Dieses eine PDF ist der Grund, warum das neue identisch paginiert.
- **Nachweis schlägt Beruhigung.** "Sieht gleich aus" ist nichts wert neben einer Tabelle gemessener Koordinaten aus beiden Dokumenten.

> Die KI hat den Bericht nicht besser gemacht. Sie hat den Grund beseitigt, warum der Bericht woanders leben musste.

Wenn in deinem Stack ein kleines Satellitenprojekt existiert, nur weil irgendeine Aufgabe vor ein oder zwei Jahren in deiner Hauptsprache umständlich war, lohnt sich ein Nachmittag für die Frage, ob das noch stimmt.

Bei mir stimmte es nicht. Hab keine Angst vor dieser neuen Welt — das hier ist ein weiteres Beispiel dafür, dass eine Codebasis *einfacher* zu pflegen wird, nicht komplizierter.

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 —