NEWS · JULY 6, 2026

Verpassen Sie Ihrer Delphi-App ein Gehirn — ohne ein einziges Byte in die Cloud zu schicken

By Dr. Holger Flick

Ihre Delphi-Anwendung funktioniert bereits. Sie enthält fünfzehn Jahre gewachsene Geschäftslogik, Formulare, die Ihre Anwender im Schlaf bedienen, und ein Datenbankschema, das drei Windows-Versionen überlebt hat. Das Letzte, was Sie hören wollen, ist: *„Schreiben Sie alles neu."*

Das müssen Sie auch nicht.

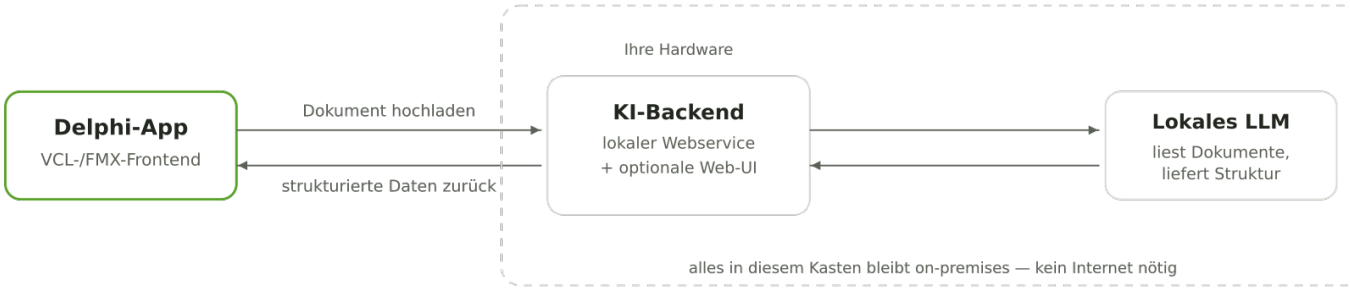
Es gibt ein Muster, auf das ich bei Kunden immer wieder zurückkomme: Lassen Sie das kampferprobte Delphi-Frontend genau da, wo es ist, und stellen Sie ihm ein kleines, modernes **KI-Backend** zur Seite, das es aufrufen kann. Das Backend erledigt die schwere, unscharfe „Versteh dieses Dokument"-Arbeit, die kein noch so ausgefeiltes `TStringList`-Parsing jemals sauber hinbekommt — und liefert Ihrer VCL-Anwendung ein sauberes, strukturiertes Ergebnis zurück, das sie längst anzuzeigen weiß. Als Bonus kann dasselbe Backend eine Web-UI mitbringen, sodass Sie eine Verwaltungskonsole gratis dazubekommen.

Und der Teil, bei dem die Buchhaltung aufatmet: **Die KI läuft vollständig auf Ihrer eigenen Hardware.** Keine Cloud. Keine Abrechnung pro Token. Keine Rechnung, kein Beleg, kein Gehaltsreport verlässt das Haus.

Ich zeige Ihnen das Ganze an einem realen Beispiel — eine hochgeladene Rechnung wird zu geprüften Ausgabendatensätzen — und beweise, dass eine zwanzig Jahre alte Delphi-Anwendung sich mit ungefähr so viel Code anbinden lässt, wie ein REST-Aufruf braucht.

Die Grundidee

Der Trick besteht darin, KI nicht länger als etwas zu betrachten, das man *einbettet*, sondern als etwas, das man *aufruft*. Ihre Delphi-Anwendung führt das Modell nicht aus. Sie schickt ein Dokument an einen kleinen lokalen Webservice, und dieser Service liefert strukturierte Daten zurück.



Ihr Delphi-Frontend ruft ein lokales KI-Backend auf — nichts verlässt die Maschine

Dieses Backend ist in meinem Beispiel eine Next.js-Anwendung. Aber halten Sie das nicht zu fest — das wichtige Wort ist nicht „Next.js", sondern **Webservice**. Ihre Delphi-Anwendung spricht mit ihm über schlichtes HTTP mit einem JSON-Body. Mit einem Backend in Go, Python oder C# würde sie auf exakt dieselbe Weise sprechen.

> **Die Ein-Satz-Version**
> Delphi schickt ein Dokument an ein lokales Backend ’ das Backend bittet ein lokal laufendes KI-Modell, es zu lesen ’ das Modell liefert strukturierte Daten ’ Delphi zeigt sie in einem Prüfdialog an, den Ihre Anwender bereits verstehen. Keine Cloud, kein Abo, keine Daten verlassen die Maschine.

Warum lokal — und warum das *Ihren* Kunden wichtig ist

Ich betreibe das auf einem Mac Studio und einem MacBook Pro, beide mit reichlich Unified Memory. Eine typische Rechnung oder ein Beleg kommt in **wenigen Sekunden** als strukturierte Daten zurück — je nach Modell oft schneller. Das ist kein Laborbenchmark, das ist die tägliche Arbeitserfahrung.

Aber die Geschwindigkeit ist nicht die Schlagzeile. **Die Privatsphäre ist es.**

Überlegen Sie, was Ihre Geschäftskunden einem Dokumentenscanner tatsächlich vorsetzen: Rechnungen mit Lieferantenpreisen, Spesenabrechnungen mit Mitarbeiternamen, Arztabrechnungen, Gehälter, Verträge. Und jetzt stellen Sie sich vor, Sie müssten ihnen sagen, dass jedes einzelne dieser Dokumente zu einer Cloud-API eines Drittanbieters hochgeladen wird, um „gelesen" zu werden. Für viele Branchen ist das ein sofortiges Nein — Recht, Gesundheitswesen, Finanzen, Behörden, Verteidigung. Es ist der ganze Grund, warum der Deal nicht zustande kommt.

Lokale KI räumt diesen Einwand vollständig aus.

> **Die unbequeme Frage, die Deals abschließt**
> *„Wer möchte seine persönlichen Ausgaben, Gehaltsreports und Lieferantenverträge mit einem Cloud-Anbieter teilen — wenn es eine vollwertige Option gibt, die das niemals tut?"* Sobald ein Dokument etwas Vertrauliches enthält, ist „es verlässt Ihren Server nie" kein Feature mehr. Es ist das ganze Produkt.

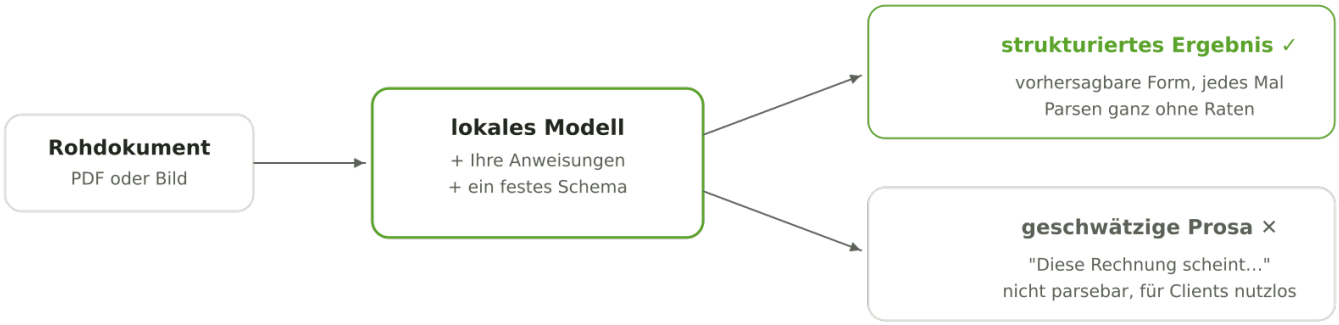
Und hier kommt der elegante Teil: Weil das Backend ein **standardisiertes Webservice-Protokoll** spricht, ist „lokal versus Cloud" eine *Konfigurationsentscheidung, keine Architekturentscheidung*. Wenn es einem bestimmten Kunden wirklich egal ist — etwa, weil er öffentliche Produktbroschüren scannt — richten Sie dasselbe Backend mit einer einzigen Einstellung auf einen Cloud-Anbieter aus. Gleicher Code, gleiches Delphi-Frontend. Sie entscheiden pro Installation, wo die Rechenleistung lebt.

Wie die beiden Hälften miteinander sprechen

Jede fähige lokale Inferenz-Runtime stellt heute eine kleine HTTP-API bereit. Sie schicken ihr einen Chat-artigen Request mit angehängtem Dokument; sie schickt die Antwort des Modells zurück. Die entscheidende Designentscheidung darüber ist diese:

> **Bitten Sie das Modell nicht um Prosa. Bitten Sie es um einen* Vertrag*, den Ihr Client parsen kann.**

Das Modell kann Ihnen problemlos einen freundlichen Absatz schreiben, der eine Rechnung beschreibt. Dieser Absatz ist für eine Delphi-Anwendung nutzlos. Was Sie wollen, ist eine **strikte, vorhersagbare Struktur** — jedes einzelne Mal dieselbe Form —, damit Ihr Client sie ohne Heuristiken konsumieren kann.



Zwingen Sie das Modell in eine feste Struktur, die jeder Client blind parsen kann

Wie Sie diese Struktur bekommen — genau dort steckt das Handwerk: die Formulierung der Anweisungen, die Beschreibung der Felder, die Regeln, die Sie einbauen, um die zehn vorhersehbaren Fehler zu verhindern, die das Modell machen möchte. Meine exakten Prompts reiche ich hier nicht herum (ein Zauberer behält *manche* Dinge im Ärmel), aber das Prinzip ist übertragbar und lohnt sich zu verinnerlichen:

> **JSON ist bequem, nicht Pflicht**
> Ein JSON-Schema ist die natürliche Wahl, weil jede Sprache es kostenlos parst — Delphi eingeschlossen, via `System.JSON`. Aber Sie sind nicht mit JSON verheiratet. Wenn Sie lieber eine kleine pipe-getrennte Tabelle, einen festen Satz von `KEY=VALUE`-Zeilen oder irgendetwas anderes hätten, das Ihr Client bequemer

> parst: **Prompten Sie einfach genau diese Form.** Das Modell produziert jeden Vertrag, den Sie spezifizieren. Wählen Sie das Format, das *Ihren* Client-Code am einfachsten macht, und zwingen Sie das Modell darauf.

Ein durchgerechnetes Beispiel: Aus einer Rechnung werden Ausgabendatensätze

Machen wir es konkret. Ein Benutzer lädt eine Marktplatz-Bestellung hoch — zwei Produkte von zwei verschiedenen Verkäufern, jede Zeile „verkauft von" einem anderen Anbieter. So kommt das Dokument an:

Die hochgeladene Rechnung: ein Dokument, zwei Verkäufer

Der Vertrag, den wir anfordern

Wir weisen das Modell an, exakt diese Struktur zurückzugeben. Jedes Feld ist definiert; das Modell füllt es aus — und sonst nichts.

```
{
  "expenseDate": "YYYY-MM-DD" | null,   // document date
  "company":     string | null,         // issuer of the whole document
  "items": [
    {
      "description": string,
      "company":     string | null,     // per-item seller (may differ per line)
      "category":    string,
      "amount":      number             // line price
    }
  ]
}
```

Eine Entscheidung in dieser Form zahlt sich später aus: `company` **existiert auf zwei Ebenen.** Das Dokument hat einen Aussteller, aber jede Zeile kann ihren eigenen Verkäufer nennen — auf einem Marktplatz kann jede

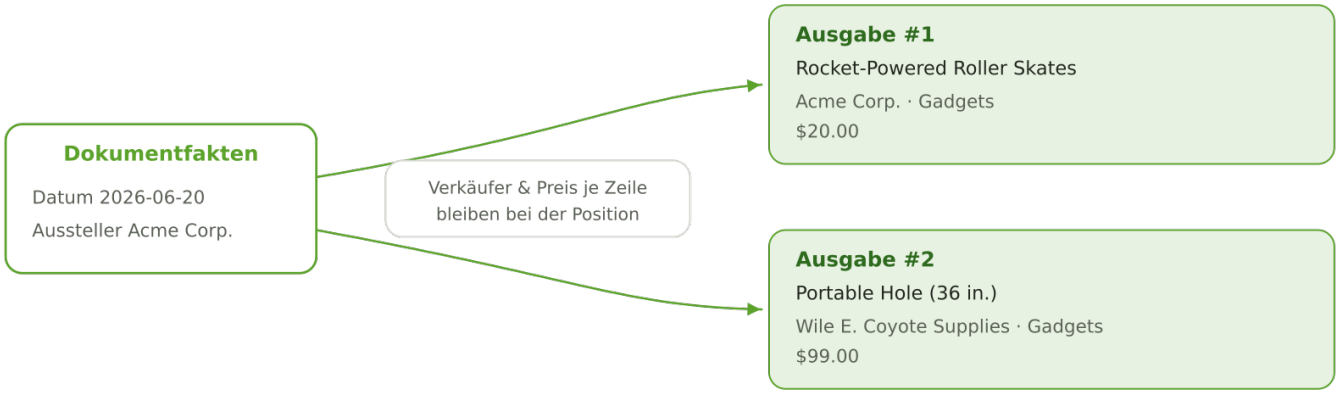
Zeile von einem anderen Anbieter „verkauft" werden. Eine Position mit `null` als Company erbt den Aussteller des Dokuments.

Was zurückkommt

Für unsere Rechnung liefert das Modell exakt dies — das eine Dokument aufgefächert in zwei Positionen, jede mit ihrem eigenen Verkäufer:

```
{
  "expenseDate": "2026-06-20",
  "company": "Acme Corp.",
  "items": [
    { "description": "Rocket-Powered Roller Skates",
      "company": "Acme Corp.",              "category": "Gadgets", "amount": 20.00 },
    { "description": "Portable Hole (36 in.)",
      "company": "Wile E. Coyote Supplies", "category": "Gadgets", "amount": 99.00 }
  ]
}
```

Die Summe und etwaige Versandzeilen sind schlicht nicht enthalten — die Extraktionsregeln schließen Summenzeilen aus der Positionsliste aus. Die Auffächerung sieht so aus:



Gemeinsame Dokumentfakten fließen in jede Position; Zeilenfakten bleiben, wo sie sind

Das echte Beispiel, das ich in Produktion betreibe, verarbeitet die ganze unordentliche Realität, die ein Beleg mit sich bringt — Rabatte, Steuern, Geschenkkarten, Versand und den Rest —, aber ich habe diesen Beitrag auf das Wesentliche eingegrenzt, damit das Muster unter der Buchhaltung sichtbar bleibt.

Ein Prinzip lohnt es sich hier herauszuziehen: **Das Modell liest, Ihr Code rechnet.** Bitten Sie das Modell nur um das, worin Code schlecht ist — ein unordentliches Dokument zu verstehen —, und behalten Sie alles, was exakt und prüfbar sein muss, in gewöhnlicher Arithmetik, in Delphi oder im Backend, wo Sie es testen können.

Jetzt der Delphi-Teil

Hier verdient sich die zwanzig Jahre alte VCL-Anwendung ihr Geld. Für Ihr Delphi-Frontend ist das *nur ein HTTP-Aufruf*. HTTP-Aufrufe können Sie längst. Sie posten ein paar Bytes, Sie bekommen JSON, Sie parsen es in Objekte, Sie zeigen sie in einem Grid oder einem Prüfdialog an, der Benutzer bestätigt, Sie speichern.

Ich zeige die Client-Seite des Roundtrips in beiden Sprachen, damit offensichtlich wird, wie wenig dahintersteckt.

Das Dokument senden

Das Backend nimmt die rohe Datei samt Name und MIME-Type entgegen. Aus Delphi mit `System.Net.HttpClient` und `System.Net.Mime`:

```
uses
  System.Net.HttpClient, System.Net.Mime, System.JSON, System.SysUtils;

function AnalyzeDocument(const AFileName: string): string;
var
  LClient: THTTPClient;
  LForm: TMultipartFormData;
  LResp: IHTTPResponse;
begin
  LClient := THTTPClient.Create;
  LForm := TMultipartFormData.Create;
  try
    // Die ganze Integration ist das hier: Datei anhängen, per POST senden, Body lesen.
    LForm.AddFile('file', AFileName);
    LResp := LClient.Post('http://127.0.0.1:8000/api/analyze-expense', LForm);

    if LResp.StatusCode <> 200 then
      raise Exception.CreateFmt('Backend returned %d', [LResp.StatusCode]);

    Result := LResp.ContentAsString;  // der strukturierte JSON-Vertrag
  finally
    LForm.Free;
    LClient.Free;
  end;
end;
```

Das Äquivalent aus dem Web-Frontend, damit Sie sehen, dass beide dasselbe tun:

```
async function analyzeDocument(file: File): Promise<string> {
  const form = new FormData();
  form.append("file", file);

  const res = await fetch("http://127.0.0.1:8000/api/analyze-expense", {
    method: "POST",
    body: form,
  });
  if (!res.ok) throw new Error(`Backend returned ${res.status}`);

  return res.text(); // der strukturierte JSON-Vertrag
}
```

Gleicher Request, gleiche Response, gleicher Vertrag. Die Delphi-Anwendung und die Web-Anwendung sind austauschbare Clients eines Backends — und ebenso wären es eine Mobile-App, ein Python-Skript oder das C#-Tool, das jemand in der Ecke noch pflegt.

Den Vertrag in Objekte parsen

Weil wir auf einer festen Struktur bestanden haben, ist das Parsen langweilig — und genau das wollen Sie. In Delphi wandert `System.JSON` direkt in Ihren eigenen Record oder Ihre Klasse:

```
type
  TExpenseItem = record
    Description: string;
    Company: string;
    Category: string;
    Amount: Double;
  end;

  TAnalysis = record
    ExpenseDate: string;
    Issuer: string;
    Items: TArray<TExpenseItem>;
  end;

function ParseAnalysis(const AJson: string): TAnalysis;
var
  LRoot, LItem: TJSONObject;
  LItems: TJSONArray;
  I: Integer;
begin
  LRoot := TJSONObject.ParseJSONValue(AJson) as TJSONObject;
  try
    Result.ExpenseDate := LRoot.GetValue<string>('expenseDate', '');
    Result.Issuer      := LRoot.GetValue<string>('company', '');

    LItems := LRoot.GetValue<TJSONArray>('items');
    SetLength(Result.Items, LItems.Count);
    for I := 0 to LItems.Count - 1 do
    begin
      LItem := LItems.Items[I] as TJSONObject;
      Result.Items[I].Description := LItem.GetValue<string>('description', '');
      // Eine Position mit null als Company erbt den Aussteller des Dokuments.
      Result.Items[I].Company     := LItem.GetValue<string>('company', Result.Issuer);
      Result.Items[I].Category    := LItem.GetValue<string>('category', '');
      Result.Items[I].Amount      := LItem.GetValue<Double>('amount', 0);
    end;
  finally
    LRoot.Free;
  end;
end;
```

Die TypeScript-Seite ist dieselbe Idee mit weniger Zeremonie:

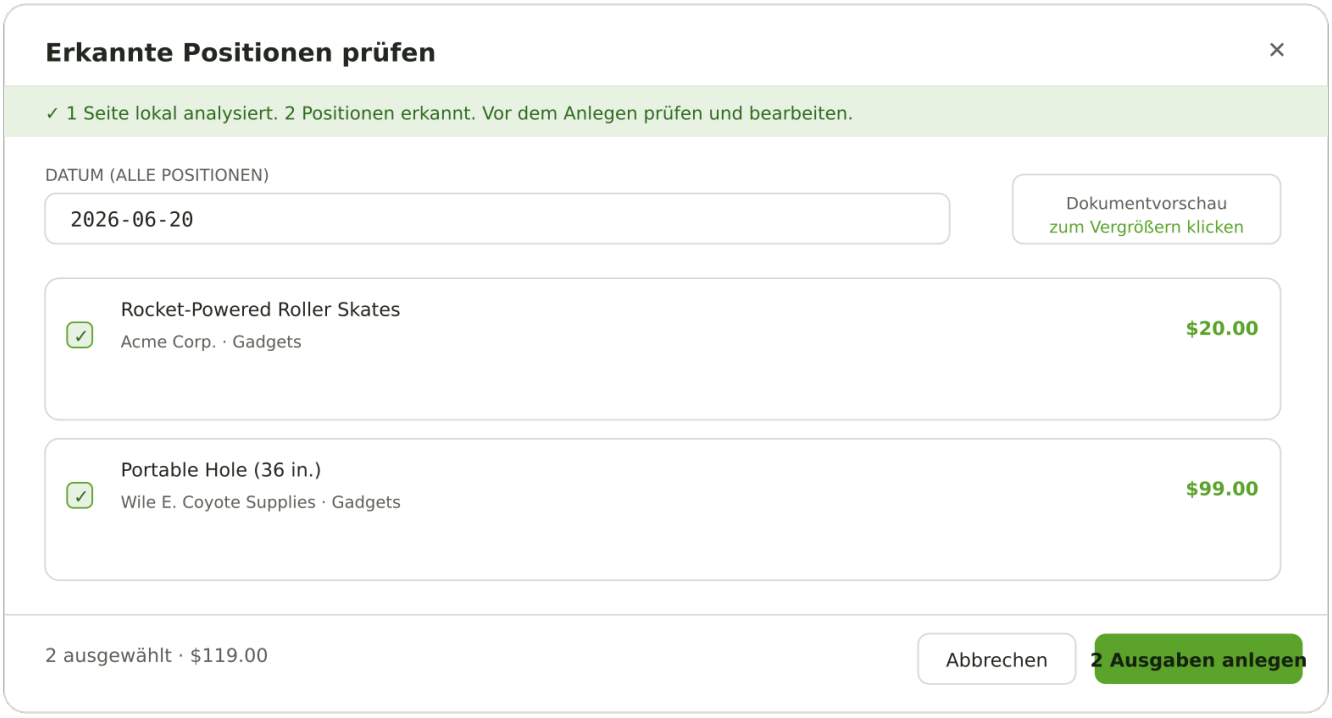
```
type ExpenseItem = {
  description: string;
  company: string;      // null erbt den Aussteller des Dokuments
  category: string;
  amount: number;
};

function parseAnalysis(json: string) {
  const raw = JSON.parse(json);
  const issuer: string = raw.company ?? "";
  const items: ExpenseItem[] = (raw.items ?? []).map((it: Record<string, unknown>) => ({
    description: String(it.description ?? ""),
    company: (it.company as string) ?? issuer,   // bei null den Aussteller erben
    category: String(it.category ?? ""),
    amount: Number(it.amount ?? 0),
  }));
  return { expenseDate: raw.expenseDate ?? "", issuer, items };
}
```

An diesem Punkt hat Ihre Delphi-Anwendung ein Array stark typisierter Records. Alles danach — die Anzeige in einem `TDBGrid` oder einer `TListView`, das Bearbeiten durch den Benutzer, das Schreiben in Ihre Datenbank

— ist Code, den Sie seit Delphi 7 schreiben. **Die KI liegt bereits hinter Ihnen.** Sie hat ihren einen Job erledigt (aus dem Bild einer Rechnung Daten zu machen) und ist aus dem Weg gegangen.

Behalten Sie einen Menschen in der Schleife

Eine Regel, an die ich mich in jedem dieser Projekte halte: **Das Modell produziert Vorschläge, niemals Verbindlichkeiten.** Nichts wird allein auf das Wort des Modells hin in die Datenbank geschrieben. Die extrahierten Daten landen in einem Prüfdialog, in dem der Benutzer kontrolliert, korrigiert und bestätigt — eine falsche Vermutung kostet so einen Tastendruck, keinen fehlerhaften Buchungssatz.



Der Prüfdialog: alles ist ein editierbarer Vorschlag, bis der Benutzer bestätigt

In Ihrer Delphi-Anwendung ist dieser Prüfschritt einfach ein Formular — ein Grid mit den geparsten Positionen, editierbaren Zellen und einem **Bestätigen**-Button. Das gemeinsame Datum sitzt oben, weil es für jede Zeile gilt; Beschreibung, Verkäufer, Kategorie und Preis sind je Position. Wenn der Benutzer auf Bestätigen klickt, *dann* schreiben Sie die Zeilen — und erst dann.

> **Das Modell ist fehlbar — planen Sie das ein, statt es zu ignorieren**
> Manchmal findet es nichts oder liest eine Zeile falsch. Das ist in Ordnung, solange der Mensch die Quelle der Wahrheit bleibt. Behalten Sie einen Weg, „eine Zeile manuell hinzuzufügen", und eine klare Statusmeldung — dann degradiert eine schlechte Extraktion zu *„der Benutzer tippt es ein wie früher"*, niemals zu einem korrupten Datensatz. Die KI ist ein schneller Assistent, kein unbeaufsichtigter Sachbearbeiter.

Die Web-UI, die Sie gratis dazubekommen

Weil das Backend eine echte Webanwendung ist, ist es nicht *nur* eine API für Delphi. Derselbe Server kann eine browserbasierte Konsole hosten — zum Verwalten von Kategorien, zum Sichten des Verarbeiteten, zum Anpassen der Konfiguration, zum Beobachten des Durchsatzes. Sie bauen das Backend einmal und bekommen zwei Frontends: Ihre bestehende Delphi-Anwendung für die Menschen, die darin leben, und eine Web-UI für die Administration von überall im Netzwerk.

Das ist der stille strategische Gewinn. Sie ersetzen Ihre Delphi-Investition nicht. Sie **umgeben** sie mit modernen Fähigkeiten, die sie aufrufen kann — und jede einzelne dieser Fähigkeiten ist wiederverwendbar für alles, was Sie als Nächstes bauen.

Was das wirklich beweist

Treten Sie für einen Moment von der Rechnung zurück. Der Dokumentenscanner ist nur die Demo. Die eigentliche Behauptung lautet:

> Eine vor Jahren geschriebene Delphi-Anwendung kann echte, moderne KI-Funktionen gewinnen, indem sie ein kleines lokales Backend über HTTP aufruft — ohne Rewrite, ohne Cloud-Abhängigkeit und ohne dass vertrauliche Daten das Haus verlassen.

Alles Schwierige an der KI — das Modell, der Speicher, die Inferenz, das Prompt Engineering, das aus einem unordentlichen Dokument saubere Struktur herauskitzelt — lebt im Backend. Alles, was Ihre Delphi-Anwendung tun muss, ist das, was sie schon immer getan hat: HTTP, JSON, ein Grid, ein Bestätigen-Button, ein Datenbank-Write.

Als Delphi MVP ist das der Teil, den ich anderen Delphi-Entwicklern am dringendsten mitgeben möchte: **Ihre Plattform ist hier nicht die Grenze.** Die Vorstellung, „alte Delphi-Anwendungen können keine moderne KI", ist schlicht falsch. Sie können — und zwar so einfach —, sobald Sie aufhören, die KI *in* die Anwendung zu quetschen, und die Anwendung stattdessen die KI *aufrufen* lassen.

Wollen Sie das in Ihrem Produkt?

Das Muster ist geradlinig. Es *richtig* hinzubekommen — das Prompt-Design, das die Extraktion zuverlässig macht, das Schema, das sauber auf Ihre Domäne passt, das lokale Inferenz-Setup, das schnell und privat ist, der Prüf-Workflow, der Ihre Daten sauber hält — genau da zahlt sich Erfahrung aus, und genau da habe ich die Zeit investiert, damit meine Kunden es nicht müssen.

Wenn Sie eine Delphi-Anwendung haben, die mit einem lokalen KI-Backend im Rücken plötzlich viel mehr könnte — Dokumente lesen, klassifizieren, extrahieren, zusammenfassen, alles auf Ihrer eigenen Hardware — sprechen Sie mich an**.** Ich helfe Teams, genau diese Art von Fähigkeit in Software einzubauen, der sie bereits vertrauen — ohne die Firma auf einen Rewrite zu verwetten und ohne ihre Daten der Cloud zu übergeben.

Ihre Anwendung funktioniert bereits. Machen wir sie intelligent.

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 —