NEWS · JULY 6, 2026

Give Your Delphi App a Brain — Without Sending a Single Byte to the Cloud

By Dr. Holger Flick



Your Delphi application already works. It has fifteen years of business logic baked in, forms your users know in their sleep, and a database schema that has survived three Windows versions. The last thing you want to hear is *"rewrite it."*

You don't have to.

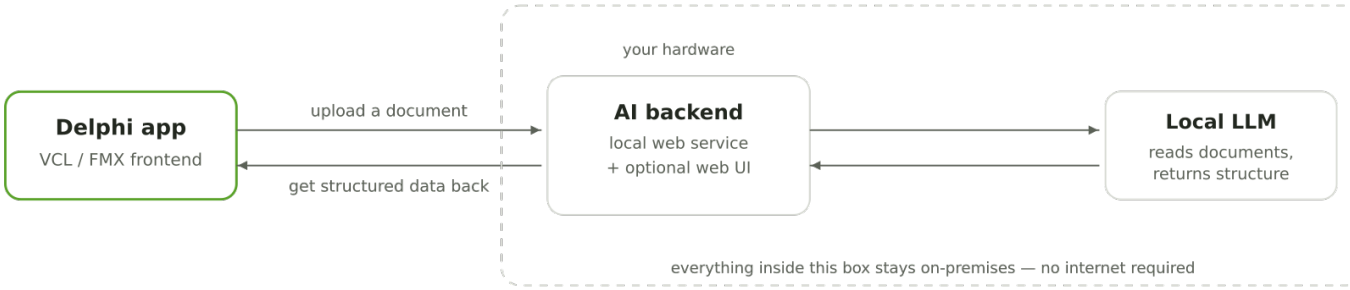
There's a pattern I keep coming back to with clients: leave the battle-tested Delphi frontend exactly where it is, and give it a small, modern **AI backend** to call. The backend does the heavy, fuzzy, "understand this document" work that no amount of `TStringList` parsing will ever do cleanly — and it hands your VCL app back a clean, structured result it already knows how to display. As a bonus, that same backend can ship a web UI, so you get a management console for free.

And the part that makes accounting stop sweating: **the AI runs entirely on your own hardware.** No cloud. No per-token bill. No invoice, receipt, or salary report leaving the building.

Let me show you the whole thing with a real example — turning an uploaded invoice into reviewed expense records — and prove that a twenty-year-old Delphi app can plug into it in about as much code as a REST call.

The shape of the idea

The trick is to stop thinking of AI as something you *embed* and start thinking of it as something you *call*. Your Delphi app doesn't run the model. It sends a document to a small local web service, and that service returns structured data.



Your Delphi frontend calls a local AI backend — nothing leaves the machine

That backend, in my example, is a Next.js app. But hold onto that lightly — the important word isn't "Next.js," it's **web service**. Your Delphi app talks to it over plain HTTP with a JSON body. It would talk to a backend written in Go, Python, or C# in exactly the same way.

> **The one-sentence version**
> Delphi sends a document to a local backend ’ the backend asks a locally-running AI model to read it ’ the model returns structured data ’ Delphi displays it in a review screen your users already understand. No cloud, no subscription, no data leaving the machine.

Why local, and why it matters to *your* customers

I run this on a Mac Studio and a MacBook Pro, both with plenty of unified memory. A typical invoice or receipt comes back as structured data in a **few seconds** — often faster, depending on the model. That's not a lab benchmark; that's the working experience.

But speed isn't the headline. **Privacy is.**

Think about what your business customers actually feed a document scanner: invoices with supplier pricing, expense reports with employee names, medical claims, payroll, contracts. Now imagine telling them every one of those documents gets uploaded to a third-party cloud API to be "read." For a lot of industries that's an instant no — legal, healthcare, finance, government, defense. It's the entire reason the deal doesn't close.

Local AI removes that objection completely.

> **The uncomfortable question that closes deals**
> *"Who wants to share their personal expenses, salary reports, and supplier contracts with a cloud provider — if there's a perfectly good option that never does?"* When a document contains something confidential, "it never leaves your server" isn't a feature. It's the whole product.

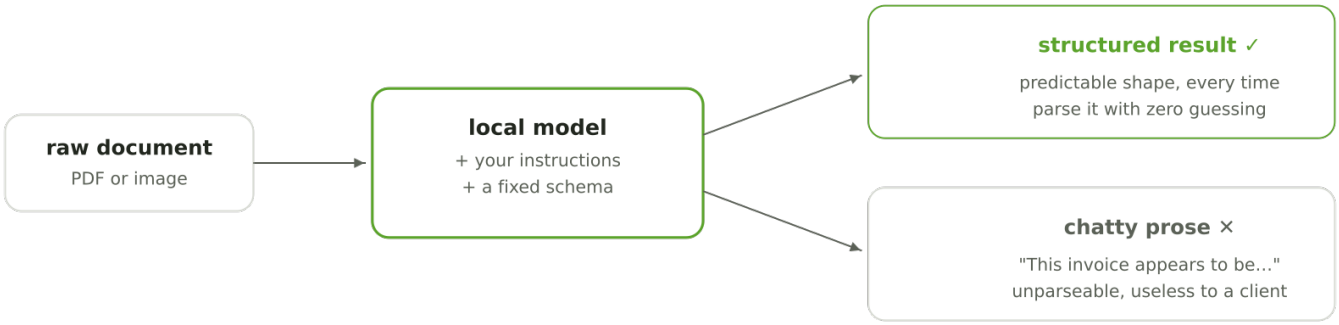
And here's the elegant part: because the backend speaks a **standard web-service protocol**, going local versus cloud is a *configuration choice, not an architecture choice*. If a particular customer genuinely doesn't care — say, they're scanning public product brochures — you point the same backend at a cloud provider by changing one setting. Same code, same Delphi frontend. You decide per deployment where the compute lives.

How the two halves talk

Any capable local inference runtime today exposes a small HTTP API. You send it a chat-style request with your document attached; it sends back the model's answer. The critical design decision on top of that is this:

> **Don't ask the model for prose. Ask it for a* contract *your client can parse.**

The model is fully capable of writing you a friendly paragraph describing an invoice. That paragraph is useless to a Delphi app. What you want is a **strict, predictable structure** — the same shape every single time — so your client can consume it without heuristics.



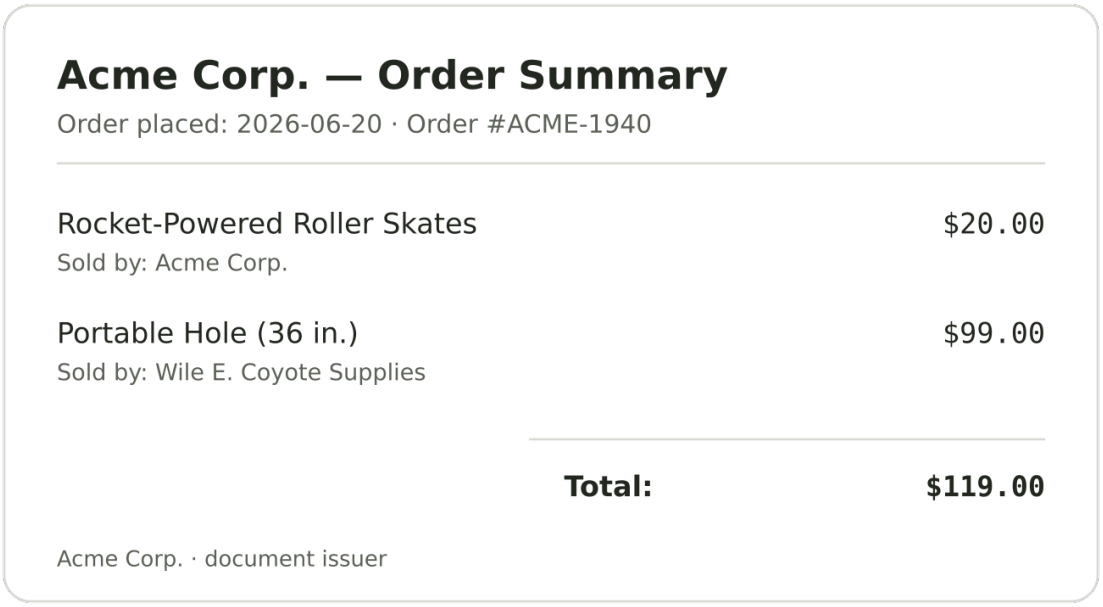
Constrain the model to a fixed structure so any client can parse it blindly

How you get that structure is where the craft lives — the wording of the instructions, how you describe the fields, the rules you bake in to prevent the model from making the ten predictable mistakes it wants to make. I'm not going to hand you my exact prompts here (a magician keeps *some* things up the sleeve), but the principle is transferable and worth internalizing:

> **JSON is convenient, not mandatory**
> A JSON schema is the natural fit because every language parses it for free — Delphi included, via `System.JSON`. But you are not married to JSON. If you'd rather have the model emit a tiny pipe-delimited table, a fixed set of `KEY=VALUE` lines, or anything else your client parses more comfortably, **just prompt for that shape.** The model produces whatever contract you specify. Pick the format that makes *your* client code simplest, then constrain the model to it.

A worked example: an invoice becomes expense records

Let's make it concrete. A user uploads a marketplace order — two products from two different sellers, each line "sold by" a different vendor. Here's the document as it arrives.



The uploaded invoice: one document, two sellers

The contract we ask for

We tell the model to return this exact structure. Every field is defined; the model fills it in and nothing else.

```
{
  "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
    }
  ]
}
```

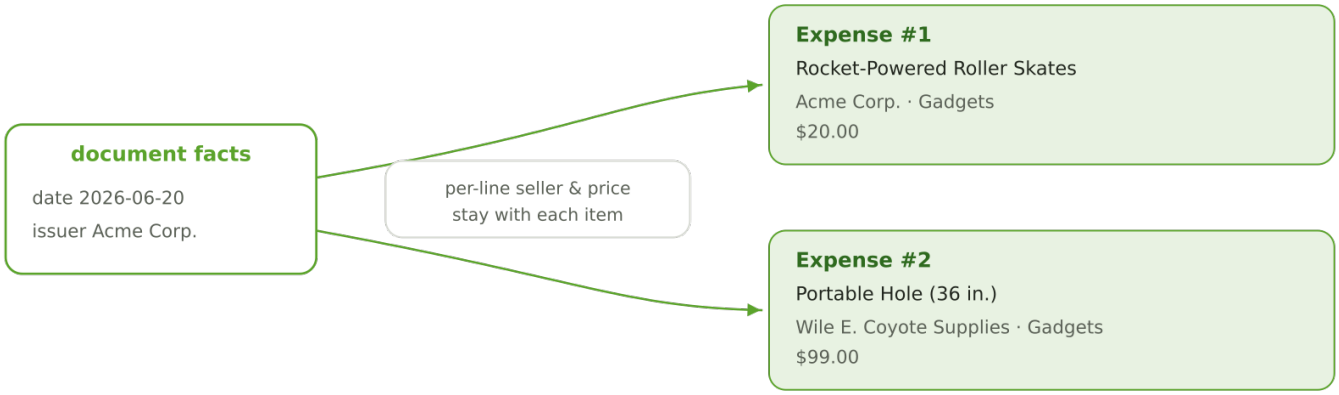
One decision in that shape pulls its weight later: **company exists at two levels.** The document has one issuer, but each line can name its own seller — on a marketplace every line may ship "sold by" a different vendor. An item with a `null` company inherits the document issuer.

What comes back

For our invoice, the model returns exactly this — the single document fanned out into two line items, each keeping its own seller:

```
{
  "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 }
  ]
}
```

The total and any shipping rows simply aren't there — the extraction rules exclude summary rows from the item list. The fan-out looks like this:



Shared document facts flow into every item; per-line facts stay put

The real example I run in production handles all the messy reality a receipt throws at you — discounts, taxes, gift cards, shipping, and the rest — but I've scoped this post to the essentials so the pattern stays visible under the accounting.

There's a principle worth pulling out of this: **the model reads, your code computes.** Ask the model only to do the thing code is bad at — understanding a messy document — and keep anything that must be exact and auditable in ordinary arithmetic, in Delphi or the backend, where you can test it.

Now the Delphi part

Here's where the twenty-year-old VCL app earns its keep. To your Delphi frontend, this is *just an HTTP call*. You already know how to do HTTP calls. You post some bytes, you get JSON, you parse it into objects, you show them in a grid or a review dialog, the user confirms, you save.

Let me show the client side of the round trip in both languages so it's obvious how little there is to it.

Sending the document

The backend accepts the raw file plus its name and MIME type. From Delphi with `System.Net.HttpClient` and `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
    // The whole integration is this: attach the file, POST it, read the body.
    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;  // the structured JSON contract
  finally
    LForm.Free;
    LClient.Free;
  end;
end;
```

The equivalent from the web frontend, so you can see they're doing the same thing:

```
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(); // the structured JSON contract
}
```

Same request, same response, same contract. The Delphi app and the web app are interchangeable clients of one backend — and so would be a mobile app, a Python script, or that C# tool someone in the corner still maintains.

Parsing the contract into objects

Because we insisted on a fixed structure, parsing is boring — which is exactly what you want. In Delphi, `System.JSON` walks it directly into your own record or class:

```
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', '');
      // A null per-item company inherits the document issuer.
      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;
```

The TypeScript side is the same idea with less ceremony:

```
type ExpenseItem = {
  description: string;
  company: string;      // null inherits the document issuer
  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,   // inherit issuer on null
    category: String(it.category ?? ""),
    amount: Number(it.amount ?? 0),
  }));
  return { expenseDate: raw.expenseDate ?? "", issuer, items };
}
```

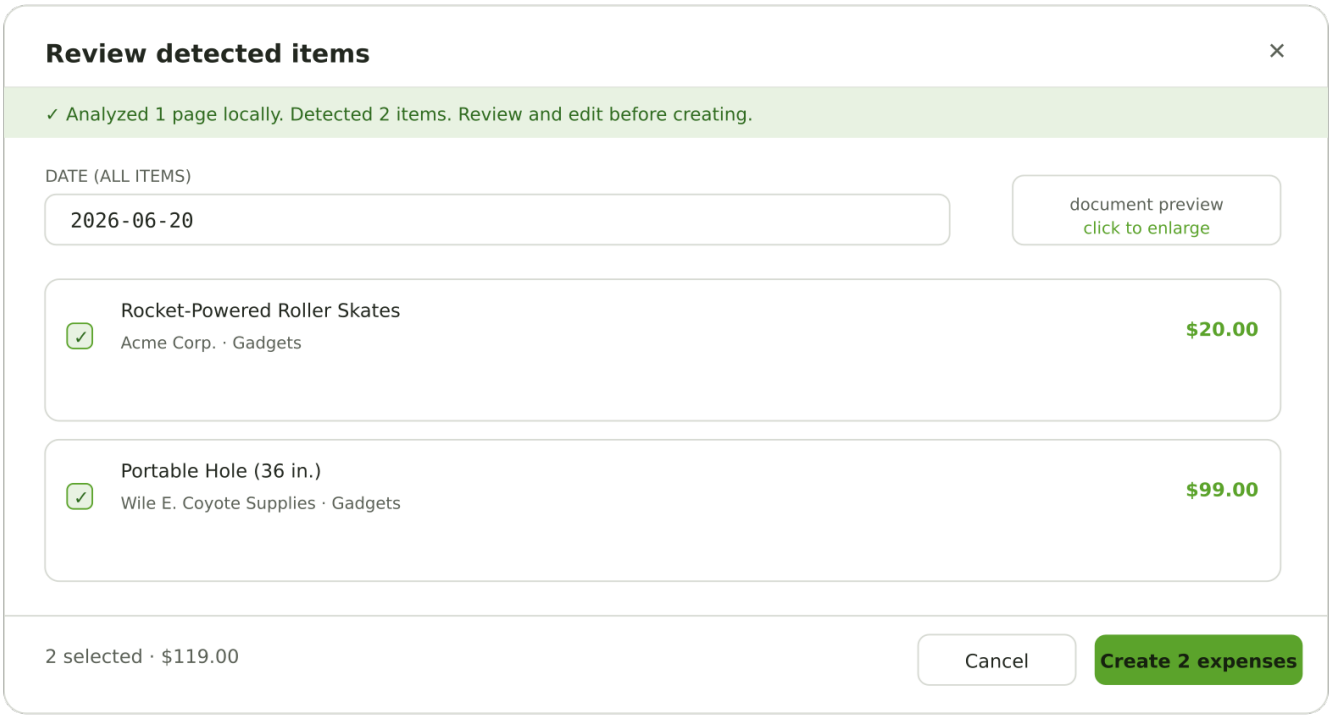
At this point your Delphi app has an array of strongly-typed records. Everything after this — showing them in a `TDBGrid` or a `TListView`, letting the user edit, writing them to your database — is code you've been writing

since Delphi 7. **The AI is already behind you.** It did its one job (turn a picture of an invoice into data) and got out of the way.

Keep a human in the loop

One rule I hold to in every one of these builds: **the model produces suggestions, never commitments.**

Nothing is written to the database off the model's word alone. The extracted data lands in a review screen where the user checks, edits, and confirms — so a wrong guess costs a keystroke, not a bad ledger entry.



The review screen: everything is an editable suggestion until the user confirms

In your Delphi app this review step is just a form — a grid of the parsed items with editable cells and a **Confirm** button. The shared date sits at the top because it applies to every row; description, seller, category, and price are per item. When the user clicks Confirm, *then* you write the rows — and only then.

> **The model is fallible — design for that, don't pretend otherwise**
> Sometimes it finds nothing, or misreads a line. That's fine, as long as the human is the source of truth. Keep an "add row manually" path and a clear status message, and a bad extraction degrades to *"the user types it in like they used to"* — never to a corrupt record. The AI is a fast assistant, not an unsupervised clerk.

The web UI you get for free

Because the backend is a real web application, it isn't *only* an API for Delphi. The same server can host a browser-based console — for managing categories, reviewing what's been processed, tweaking configuration, watching throughput. You build the backend once and you get two frontends: your existing Delphi app for the people who live in it, and a web UI for administration from anywhere on the network.

That's the quiet strategic win. You're not replacing your Delphi investment. You're **surrounding** it with modern capabilities it can call into — and every one of those capabilities is reusable by whatever you build next.

What this actually proves

Step back from the invoice for a second. The document scanner is just the demo. The real claim is this:

> A Delphi application written years ago can gain genuinely modern, AI-powered features by calling a small local backend over HTTP — with no rewrite, no cloud dependency, and no confidential data leaving the building.

Everything hard about AI — the model, the memory, the inference, the prompt engineering that coaxes clean structure out of a messy document — lives in the backend. Everything your Delphi app has to do is what it's always done: HTTP, JSON, a grid, a confirm button, a database write.

As a Delphi MVP, this is the part I most want other Delphi developers to hear: **your platform is not the limitation here.** The idea that "old Delphi apps can't do modern AI" is simply false. They can — this easily — the moment you stop trying to cram the AI *inside* the app and start letting the app *call* it.

Want this in your product?

The pattern is straightforward. Getting it *right* — the prompt design that makes extraction reliable, the schema that maps cleanly onto your domain, the local-inference setup that's fast and private, the human-review flow that keeps your data clean — is where the experience pays off, and where I've spent real time so my clients don't have to.

If you have a Delphi application that could suddenly do a lot more with a local AI backend behind it — reading documents, classifying, extracting, summarizing, all on your own hardware — let's talk**.** I help teams add exactly this kind of capability to software they already trust, without betting the business on a rewrite or handing their data to the cloud.

Your app already works. Let's make it smart.

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 —