The builder I didn't build

The previous three versions of this product all assumed a template-authoring UI was the hard thing to build. The 2026 version doesn't have one. An AI reads a sample document and produces a template directly.

This is the second of four posts. Post 1 is the long history — four product attempts going back to 2011, plus two decades of PDF-generation work alongside it. Post 3 is the technical architecture that makes AI output reliable. Post 4 is about the runtime constraints.

What Docca had, and what I kept starting on

Docca's authoring layer was a JSX component library that rendered to Docca's DML markup — the `docca-pdf-react-builder` repo. You wrote a template as Preact components, POSTed data to an Express route, and got DML back; the DML then went as a separate HTTP call to the PDF server, which returned the PDF.

None of the four product versions ever had a drag-and-drop visual builder. The closest I got was a couple of side projects in 2024 and 2025 — `bildar` and then `bildar-monorepo` — aimed at drag-and-drop website content editing, not PDF templates specifically. Neither reached a finished state. What they did give me was a first-hand sense of how much work a drag-and-drop builder actually is once you get past the easy parts.

So when I started looking at reviving the PDF product in early 2026, I already knew that "build the visual template editor" was not a few weekends of work.

What changed

Talking to a couple of work friends in late March 2026, this is how I described where I'd got to:

> Originally the intent was that you wrote templates in an HTML type language. Ideally you would have had a drag-and-drop builder or whatever. And I sort of thought, okay, AI is the thing today. So, let's get an AI to build it.

That's the pivot in one sentence. The product isn't *"a PDF service with a drag-and-drop builder"* any more. It's *"a PDF service where the AI builds the template."*

The mechanism, again in my own words from the same conversation:

> Got it to the point where it's got all these samples, just screenshots, and it's using Visual AI to grab that and create a template and then generate the PDF from it. And then it's looking at the two, identifying differences, and going back and fixing them. And that's where I can leave it for an hour at a time or whatever and it'll just keep going through and it's doing a really good job.

The "samples" are sample invoices, receipts, quotes and similar documents. The tool takes a screenshot, asks a vision model for a template in the makesPDF schema, renders the PDF, compares the result against the source, and iterates.

Why this isn't just a prompt wrapper

A valid objection: *so it's ChatGPT-with-a-prompt that outputs JSON?* No, and this is the part that matters.

Asking a vision model for a template *directly* gives you results that look right but don't render. Invented tags. Wrong property names. Column widths that don't add up. Footers in places footers aren't allowed. The model's intent is fine; the output is malformed in ways you only catch when the renderer blows up.

What makes the scan loop work isn't the model. It's that makesPDF has three things sitting behind it that constrain and validate the output:

1. A **template catalog** — one machine-readable source of truth that defines every valid tag, style, component, and nesting rule. When the model returns a template, we validate it against the catalog before trying to render.
2. A **compact builder DSL** — the model writes ~75–95% fewer tokens than it would for the equivalent raw JSON, which means faster responses, cheaper requests, and fewer places to get things wrong.
3. A **skill file** at a stable URL — one markdown file that teaches any AI assistant how to use the service, so the knowledge the model needs isn't locked inside one prompt.

Those three pieces are the subject of post 3. The short version: without them, AI-generated templates are a demo. With them, they're a product.

The other front door

For developers who don't want to send a screenshot and let the service figure it out, the DSL is a direct front door. You write the template yourself — or you ask *your own* AI assistant to write it using the published skill file — and POST it to `/api/v1/preview` or save it and render via `/api/v1/render`. The DSL looks like this:

```
const template = doc(
  { size: "A4", title: "Invoice {{invoiceNumber}}" },
  page(bold("Invoice", 18), gap(8), r(s("Bill to:"), s("{{customer.name}}")))
);
const sampleData = {
  invoiceNumber: "INV-001",
  customer: { name: "Acme Pty Ltd" },
};
```

That's executable JavaScript. Readable, small, and precisely what the service expects. No visual builder needed; no proprietary markup to learn.

What didn't change

One thing worth being honest about. Vision-scanned templates are a very good starting point, not a finished deliverable. Most of them benefit from a human pass — to rename fields, tighten spacing, or correct a misread. What's different from the old Docca workflow isn't that the human step is gone; it's that the human step starts from something mostly correct instead of a blank canvas or a builder you have to wrestle into shape.

A proper visual editor for touching up templates is still a reasonable thing to build, and it might get built. But it's no longer the *first* thing that needs to exist, and when it does get built, its job will be narrower and better-defined — *edit the template the AI produced* — rather than the much larger *compose a template from scratch.*

The broader point

If you strip away the AI framing, the claim this post is making is about shape. The standard shape for a PDF generation service in 2026 is: a headless browser, some HTML templates, and a queue. That works, and lots of products are built that way, and the result is PDFs that take several seconds to render because a browser has to boot every time. makesPDF is shaped differently. There's no browser. The template isn't HTML. The render path is a deterministic layout engine that produces a PDF/A-2A + PDF/UA-1 compliant document in sub-second wall-clock time for typical inputs — no browser boot, no cold-start penalty. The AI-generated-template bit sits on top of that engine — it's a nice front door, not the entire building.

That matters because the performance, cost, and reliability properties all come from the runtime shape, not from the AI. The AI makes the service easier to use. The runtime makes it fast and cheap. Both posts 3 and 4 cover that side in more detail.

Next

Post 3 is the architecture post: catalog, DSL, skill file, and how they fit together. Then post 4 is the runtime post: Cloudflare Workers, no eval, no browser, and the AST-walking sandbox I originally wrote for Docca in 2017 and dusted off nine years later for a completely unrelated reason.