Four goes at the same idea

I've been bumping into PDF generation as a problem since 1999. Four of my attempts to turn it into a product have reached the level of "you could use this." Three of those didn't find a market. This post is the history, honestly told.

This is the first of four posts about makesPDF. The second is about what's different about the 2026 version. The third is the technical architecture. The fourth is about how it all fits in a Cloudflare Worker.

Prologue 1: ezyDVD (1999–2011)

EzyDVD was an Australian online DVD retailer founded in March 1999. I was the sole developer on it for the first few years, starting on shared hosting with Perl CGI and ending on a dedicated server running Apache with `mod_perl`.

Somewhere in the pile of things I built for ezyDVD was the PDF bit: an automated order-processing system that charged cards and generated packing slips when pre-ordered titles shipped. To give a sense of the scale it was running at — the site cost about $3,000 to set up and was doing $3.5 million in sales by its second year, and the order system was the thing turning that revenue into physical parcels going out the door. I can't find the source any more, but it was Perl and almost certainly `PDF::API2` — it's what was on CPAN, and it's what I'd reach for again later.

The packing-slip problem wasn't glamorous. It was also never quite *finished*. Every few months something about the layout would break because a new product category had a longer-than-expected title, or a customer's address had an extra line, or the whole shipment label needed to move two millimetres to play nicely with a new sheet of pre-cut stock. That ongoing tax is the thing I remember most clearly about PDF generation from that era. It never quite sat down.

Prologue 2: SafetyCulture (2012–2019)

After ezyDVD I joined SafetyCulture in early 2012 as their senior software engineer. Another Australian startup, very early — at that point they were still in Townsville, and I'd drive up from Cairns every month or two and spend a couple of days working with the founders in the garage. Classic startup stuff.

One of the features I owned for most of my time at SafetyCulture was the server-side PDF audit export — replicating the iAuditor mobile app's PDF output, in Python, running on App Engine. The library was ReportLab, which in the early 2010s was the main option on Python.

The challenge wasn't ReportLab itself. iAuditor audits were generated from templates the users had built — custom templates producing custom audits with custom data, no consistency between them. We had to handle every possible variation a user could put in a template, plus images, plus everything else people answered with.

Over many years between ezyDVD and SafetyCulture, I'd wrangled a lot of PDF output.

Boxer (2011–2012)

PDF::Boxer was the first real attempt to fix what I'd been finding frustrating. A Perl module, published to CPAN in early 2012, built on top of `PDF::API2`. The idea was simple: describe the document as nested boxes — columns, rows, a grid — and let the library work out the measurements and positioning. A layout layer sitting on top of the low-level library I'd been using for years.

Boxer wasn't a product. It was a CPAN module, written in the overlap between the end of the ezyDVD engagement and the start of SafetyCulture. The POD is still up. It's honest about its limitations. The BUGS section reads:

> Sometimes leaves a small gap or overlap between boxes. Probably a rounding issue.

The TODO mentions paging, which at the time was another way of saying *I haven't done paging yet.*

Unicorn (2013–2015)

PDFUnicorn, briefly rebranded as EzyPDF, was the first attempt to turn the Boxer idea into a business. Perl again, but this time a full service: Mojolicious for the HTTP layer, Mango for MongoDB, Template::Alloy for template rendering, a Stripe plugin for billing, Hypnotoad as the app server, nginx out front. The cpanfile is still in the repo if you want to see the shape of it.

The core library was PDF::Grid — also mine, also on CPAN, the successor to Boxer with the rough edges filed down.

It worked. You could sign up, upload a template, POST JSON, get a PDF back. I had a landing page, pricing, the lot.

It didn't find customers. I think I nearly had one at one point. Hosting cost money every month and nobody was paying for it, so after a while I turned it off. That's the story. It wasn't a technical failure — the thing generated PDFs fine. It just didn't find a market, and I couldn't keep paying for servers indefinitely.

Docca (2016–2019)

Docca was the Node version. By 2016 I was deep into the JavaScript and React work at SafetyCulture, and going back to Perl for a side project didn't feel right.

The architecture was more ambitious than Unicorn. Four repos: `docca-pdf-writer` (the rendering library), `docca-pdf-server` (the HTTP API), `docca-pdf-react-builder` (a visual template builder), and `docca-docker` (the deployment setup). The server spoke a DML markup language (`<doc><page><row>…</row></page></doc>`) and supported Mustache templates:

```
http -f http://localhost:3999/render/template \
  template@invoice.mustache \
  data@invoice.json \
  image@logo.png \
  authorization:'Bearer apikey_e5721876...' \
  -d -o invoice.pdf
```

Four containers behind nginx. It ran on Node 6, which dates it nicely. This was all side-project work — my day job was SafetyCulture, and Docca was what I did in the evenings. There were no customers, no revenue. It wound down quietly when life got busy.

The code sat on my drive for about seven years.

makesPDF (2026)

In early 2026 I pointed Claude at the Docca code and asked for a review. Its verdict — actual phrase — was *"this is gold"*, which was nice to hear about code I'd forgotten about.

That was the nudge. Not *let me rewrite Docca*, but *what would this look like if I built it today?*

A few things had changed since 2019:

- **The runtime.** Cloudflare Workers run on V8 isolates with near-instant cold starts. No container to provision, no nginx to patch, no VPS paying for itself whether anyone uses the service or not. The slow cost bleed that killed Unicorn commercially basically doesn't exist on Workers. You pay per request.
- **The language.** TypeScript in 2026 is a pleasant place to work. The whole engine — layout, PDF writer, font subsetter — compiles to a single Worker bundle with no native dependencies.
- **The AI bit.** I'd always assumed the next version would need a drag-and-drop WYSIWYG builder. I'd started on one a couple of times and made decent progress both times. Then it turned out the AI could just build the document. Post 2 is about that specifically.

So that's makesPDF. A Workers-hosted service that turns Markdown or a compact JavaScript DSL into PDF/A-2A + PDF/UA-1 compliant documents — small docs render end-to-end in well under a second, with no browser boot and no cold-start penalty. Same underlying idea as Boxer in 2011, really: describe the document as nested boxes, let the engine do the measuring. The things around that core idea are new.

Why the fourth attempt might be different

makesPDF just launched. There are no customers yet. The previous three didn't find a market; the fourth might not either.

The honest list of what's changed:

1. **Running costs are near-zero at idle.** On Workers, a service that nobody uses costs me essentially nothing. The slow bleed that killed Unicorn isn't a factor.
2. **The default input isn't a template, it's an instruction.** Non-developers can describe a document in natural language or show a sample image and get something usable back. Developers can use the compact DSL directly. That's a wider front door than any previous version had.
3. **I've been hacking on this on and off for 25 years.** Pagination, font subsetting, the measurement pass — these aren't new problems to me. I'm solving them again with better tools.

None of that is a guarantee. It's a list of reasons the fourth attempt feels worth making.

Part of why I keep coming back to this is that every PDF library I've ever used has had rough edges. You see a PDF with blocks overlapping a page break, or a table running off the side, or a heading orphaned at the bottom of a page, and you think *surely this can be better.* Four product attempts and twenty-plus years of ongoing maintenance later, I still think it can.

Next

Post 2 is about the pivot from *AI helps me build a builder* to *AI is the builder*. Post 3 is the technical architecture that makes AI output reliable. Post 4 is the runtime story, including a sandboxed JS interpreter I originally wrote for Docca in 2017 that became relevant again in 2026 for an entirely different reason.