A catalog, a tiny DSL, and a skill file

Making AI reliably produce a working PDF template needs three things working together — a machine-readable catalog of everything that's valid, a compact DSL that costs roughly 75–95% fewer tokens than the raw JSON equivalent, and a skill file the AI can fetch once and remember.

This is the third of four posts. Post 1 is the long history — four product attempts going back to 2011. Post 2 is about the pivot from "AI helps me build a builder" to "AI builds the document." This one is the technical one. It's dev-heavy but I'll try to keep it readable if you're not one.

The problem with asking an AI for JSON

Imagine you ask Claude or GPT-4o to produce an invoice template. The naïve approach is: describe your schema in a giant prompt, show some examples, hope for the best. You get JSON back. Sometimes it's valid. Often it invents a tag you don't support. Sometimes it uses `font_size` when your schema wants `font-size`. Occasionally it produces a row with column widths that don't add up. You render it, something explodes, you bolt on a catch block, you tell the user "sorry, try again", they try again, same thing happens with a different part.

The issue isn't the AI — it's that there's no single source of truth for *what valid looks like*. Your schema lives in one place, your documentation lives in another, your prompt has a hand-curated summary that drifts out of sync. The AI sees the prompt, which is a lossy compression of the schema, which is itself a lossy description of what the renderer can actually handle. Three sources, three chances to disagree.

The catalog

The fix is a single machine-readable catalog that every other source is generated from. In makesPDF, the catalog has roughly this shape:

```
catalog/
  tags            — 20 tag definitions (14 AI-visible + 6 internal)
  styles          — 33 style property definitions (19 AI-visible + 14 internal)
  components      — 26 component patterns (atoms, molecules, organisms)
  recipes         — 8 document recipes (invoice, receipt, quote, etc.)
  prose           — design principles and common mistakes, as prose
  generate-prompt — turns catalog data into an AI system prompt
  validate        — validates a template against catalog rules
```

The shape is deliberately boring. Each tag definition says what it's called, what children it can have, what styles it accepts, whether it's AI-visible or internal. Each style property says what values it accepts and what the default is. Components are named patterns — "header row", "totals block", "footer with page number" — and recipes stitch components together into whole documents.

The AI-visible / internal split matters. Some tags exist for the renderer's internal use and would only confuse a model if it saw them. You mark those `aiVisible: false` and they never appear in the generated prompt. Same for styles — there are 33 properties in total but only 19 need to exist in the AI's mental model.

What the catalog gives you:

- **The AI system prompt is generated from it.** `generateSystemPrompt(catalog)` walks the tag and style definitions, stitches in the prose principles, and returns a ready-to-use prompt. Change a tag, the prompt updates next build. No drift.
- **Templates are validated against it.** `validateWithCatalog(doc, catalog)` checks tags exist, nesting is legal, style properties are real, row widths add up, footers are in the right place. Returns `{ valid, issues[] }`. The preview endpoint logs any warnings before rendering, and the save endpoint rejects anything with errors.
- **Humans read the same thing the AI reads.** The skill file published at `/skills/pdf-template-author.md` is also generated from the catalog. One source of truth, multiple render targets.

Before this split, the same information lived in three separate places — a hand-written prompt file, Zod schemas, and some defaults baked into the layout engine. Three places, three chances for them to disagree. Collapsing them into one source noticeably reduced the rate of invalid output.

The DSL

The other half of the token problem is verbosity. Here's a simple document as raw `DocumentDefinition` JSON:

```
{
  "attr": { "size": "A4", "title": "Hello" },
  "kids": [
    {
      "tag": "page",
      "kids": [
        {
          "tag": "column",
          "kids": [
            {
              "tag": "text",
              "attr": { "style": { "font-size": 14 } },
              "kids": [{ "tag": "span", "kids": ["Hello world"] }]
            }
          ]
        }
      ]
    }
  ]
}
```

And the same document in the builder DSL:

```
doc({ size: "A4", title: "Hello" }, page(col(text({ "font-size": 14 }, s("Hello world")))));
```

Roughly 75–95% fewer tokens, depending on the document. For a simple case like this it's about the same, but for a full invoice with twenty line items, headers, a totals block, a footer, tagged for accessibility — the saving is substantial. And AI output costs money per token, so token count matters.

The DSL is just a set of functions: `doc`, `page`, `col`, `r` (row), `text`, `s` (span), `img`, `hdr`, `ftr`, `gap`, and a handful of composition helpers on top — `bold`, `italic`, `lv`, `addr`, `th`, `td`, `each`, `when`. Each function returns a plain `DocumentDefinition` node — there's no magic, no codegen, no intermediate representation. The renderer accepts either JSON or a DSL script. The DSL is a convenience layer for humans and AI, not a separate language.

The skill file

This is the part I think is underrated. makesPDF publishes a single markdown file at `https://makespdf.com/skills/pdf-template-author.md`. The file is about 1,400 lines long. It contains the full DSL reference, the style properties, the component patterns, a walkthrough of how to author a template, and the endpoints you'd call to render one. It's self-contained. You can drop the URL into Claude Code, Codex, Cursor, or any AI coding assistant as context, and that assistant now knows how to use makesPDF.

This matters because the interesting unit of distribution in 2026 isn't "API docs a human reads." It's "a file an agent fetches once and remembers." When someone's working with an agent and says *"make me a PDF of this invoice"*, the agent doesn't need to guess the API. It fetches the skill file, learns the shape, writes the DSL, calls the render endpoint. Done.

I stole — or more accurately, reused — this pattern after listening to a podcast with the Anthropic team early in 2026, where they were genuinely unexcited about MCP and very excited about skills. The shape they described was: forget trying to give agents tools via some structured protocol; let them write their own skill files as they learn, and load them on demand. The example that stuck with me was a browser-control skill: every time the agent hits a new cookie banner it doesn't know how to dismiss, it figures one out and writes the pattern to its own skill file. Next time, it reads the file, no re-figuring-out required.

There's a small story here that I like. I asked Claude Opus to read my first draft of the skill file and clean it up — reduce it, reorganise it, whatever looked right. It deleted about 95% of what I'd written. When I asked why, it said the information was already in the reference files and linking to them was better than duplicating them. The skill file became a short description plus pointers. The agent finds it, follows the pointers when it needs detail, and doesn't carry the whole reference in context every time.

This is the right shape for AI-facing documentation, and it's the opposite of how human docs work. Human docs want to be comprehensive-in-one-place because you don't want to make the reader click. Agent docs want to be *sparse with good pointers* because every token in context is a token taken away from the actual work. Same content, different packaging.

How the three pieces connect

Stepping back:

1. The **catalog** is the source of truth. Anything that changes about what a template can contain goes here.
2. The **DSL** is a compact surface the catalog describes. Changing the catalog doesn't change the DSL functions — they're stable — but it changes what's valid to put inside them.
3. The **skill file** is generated from the catalog plus hand-written prose, so agents see the same rules that the validator enforces.

When someone POSTs a DSL template to `/api/v1/preview`, the flow is:

1. Execute the DSL in a sandbox (more on that in the next post) to get a `DocumentDefinition` tree.
2. Validate the tree against the catalog.
3. If valid, render it through the layout engine → PDF writer pipeline.
4. Return the PDF, or on failure, return the list of catalog issues so the agent can fix them and retry.

Step 4 is important. When the agent gets a 400 back, the error message isn't "rendering failed" — it's "you used `font_size` but the valid property is `font-size`" or "your row children's widths sum to 110%". Specific, actionable, and exactly the shape the agent needs to fix it on the next try. Most of the feedback loops I mentioned in post 2 work because the validator returns the right kind of error, not because the AI is particularly smart.

Two things that worked for me

Of the things I've tried on makesPDF this year, two are the ones I'd keep if I were starting the build again:

- **One source of truth for what valid input looks like.** The human docs, the AI prompt, the schema, and the validator all come from the catalog now. When I had them in separate files, they drifted, and the drift showed up as flaky AI output. I don't know if this is the right shape for every AI-facing service, but for this one it's been the most useful single change.
- **A skill file at a stable URL, kept sparse.** The version that worked is the one Claude trimmed down to pointers — a table of contents plus links to the detail, not the detail itself. Human docs want to be comprehensive in-place; agent docs seem to want the opposite.

Next

The last post in this series is about the runtime: how you fit a PDF engine into a single Cloudflare Worker request with no `eval`, no browser, no Node `fs`, and still hit sub-second wall-clock renders for typical documents. That includes a little story about a sandboxed JS interpreter I wrote in 2017 for Docca and had to dust off in 2026 for a completely different reason.