About a second, no eval, no browser

Running a PDF engine inside a single Cloudflare Worker request means giving up a lot of things you'd normally reach for — `eval`, headless Chromium, Node's filesystem, most of the npm ecosystem that assumes any of those exist. The payoff is rendering typical documents in about a second — no browser boot, no cold-start penalty — and hosting costs that round to zero at idle.

This is the fourth and last of the series. Post 1 is the long history — four product attempts going back to 2011. Post 2 is about the AI pivot. Post 3 is about the catalog / DSL / skill-file architecture. This one is the runtime deep-dive.

If you've read my 2017 post about writing a PDF writer from scratch, some of this will feel familiar. The problem is the same problem it was nine years ago. The constraints are different, which is why the shape of the answer is different.

What Workers take away

Cloudflare Workers run V8 isolates, not Node. No `fs`. No `Buffer` (you use `Uint8Array` instead). No native addons. No child processes. Critically for a code-execution service, CSP blocks `eval` and `new Function` by default — and even if you could turn them off, you wouldn't want to, because they're the biggest single vector for tenant-escape in a multi-tenant runtime.

You also don't get headless Chromium. The vast majority of PDF-from-HTML services I've seen boot Puppeteer or Playwright, load the page, print to PDF, and tear the browser down. That round-trip is *hundreds of milliseconds on a warm browser*, seconds on a cold one, and Workers don't give you the browser in the first place. So: no HTML-to-PDF shortcut.

What you *do* get: V8 itself, fast cold starts, cheap per-request billing, global distribution, and the ability to ship a single bundle of pure JavaScript that Just Runs.

The trick is making "pure JavaScript that Just Runs" include a PDF engine.

The three-stage pipeline

makesPDF's rendering pipeline is three packages stitched together:

1. `@pdf/template-engine` — resolves `{{variable}}` expressions, loops (`template:each`), conditionals (`template:if`). Input: a `DocumentDefinition` + a data object. Output: a `DocumentDefinition` with variables filled in.
2. `@pdf/layout-engine` — the measurement and positioning pass. Sixteen steps: style cascade, text bundling, font measurement, width computation, height computation, positioning, page splitting, page-number resolution, render-command extraction. Widths are resolved **top-down** — each parent portions out its available width among its children — and then heights are computed **bottom-up** from the leaves back, where a column's height is the sum of its children's and a row's is its tallest child's. Took a while to get right; nice once it clicked. Input: the resolved `DocumentDefinition` plus a `TextMeasurer`. Output: a `LayoutResult` with per-page render commands.
3. `@pdf/pdf-writer` — low-level PDF binary assembly. CID TrueType font subsetting, image embedding (JPEG and PNG with SMask transparency), content streams, XMP metadata, the structure tree for tagged PDFs. Produces PDF/A-2A + PDF/UA-1 compliant output.

All three are pure TypeScript. No native dependencies. The only runtime deps that aren't written in-house are `opentype.js` (for parsing TTF tables during subsetting), `fflate` (for gzip/deflate), `acorn` (more on that in a moment), and `elkjs` (for Mermaid diagram layout — more on that too).

Coordinate handling is the kind of detail that bites you if you don't think about it. The layout engine uses top-left origin — makes sense if you think in terms of HTML-like boxes. PDF uses bottom-left origin — makes sense if you think in terms of mathematical coordinates. The bridge between the two lives in `apps/web/src/server/layout-renderer.ts`, which takes the layout engine's render commands and emits PDF content stream operations with the y-axis flipped. This is the sort of thing you only notice once, at 2am, when your header is mysteriously rendering at the bottom of the page.

Fonts, without a filesystem

In a normal Node PDF service you'd ship some TTF files in a `fonts/` directory and have the PDF library load them from disk. Workers don't have a disk.

The four fonts makesPDF ships — Inter, NotoSans, Cousine (a monospace for code blocks), and a subset of NotoSansSymbols2 (for checkbox glyphs, bullets, and arrows used in markdown rendering) — are embedded directly in the bundle. The raw TTFs get gzip-compressed, then base64-encoded, and dropped into a TypeScript module as string constants. At runtime, when the `FontManager` needs one, it base64-decodes and gunzips the font data on demand, parses it with `opentype.js`, and hands the glyph tables to the subsetter.

Subsetting matters because a full TTF is not small, and a PDF that embeds the whole font for one short heading is wasteful. The pipeline works in two passes. First pass: walk the document, collect every character that actually gets rendered, per font variant. Second pass: the PDF writer emits each font as a CID TrueType subset containing only those characters. A heading that uses twenty characters produces a font object sized to those twenty characters, not the whole font. Output PDFs end up roughly the size you'd expect them to be.

The character-collection pass also has a nice property I didn't appreciate until I'd built it: it makes adding new symbol glyphs cheap. If someone adds a new emoji or arrow to their markdown, the measurer notices it needs a glyph, the accumulator adds it to the used-characters set, the subsetter includes it. No manual font configuration.

Scriptable — a sandboxed JS interpreter from 2017

The template DSL is JavaScript. Users can send the service something like:

```
const template = doc({ size: "A4" }, page(col(s("Hello {{name}}"))));
const sampleData = { name: "World" };
```

That's a real JavaScript script. To render it, I need to execute it and pull out the `template` and `sampleData` variables. `eval` or `new Function` would give you that — if you trust the input. With user-supplied code you don't, and on Workers the CSP blocks both anyway.

What I have instead is a little package called scriptable, which I originally wrote in August 2017. The first commit message is *"evaluate babylon AST"*. It's a safe, sandboxed JavaScript interpreter that walks the AST node-by-node — parses the source with a real parser, then handles each node type explicitly. Only whitelisted functions are callable. There's no way to reach out to the host, no way to define new free-form functions, no way to loop forever without hitting a step limit, no way to allocate arbitrarily large strings or arrays.

I wrote it originally for Docca. The reason back then was that Docca let users write little data-transform scripts to massage input data before rendering, and running untrusted user JavaScript with `eval` in a Node process that had database credentials in its environment felt like a bad idea. Even on the server side. The sandbox was a defence-in-depth thing.

I put it away when Docca wound down and didn't think about it for about eight years. Then in 2026 I was setting up makesPDF on Workers, hit the *no eval, no new Function* wall, and realised the thing I needed already existed on my drive. I dusted it off, swapped the old `babylon` parser for `acorn`, added step limits and allocation limits and a wall-clock deadline, published it to npm, and that's what runs the DSL now. It's at version 0.2.1 as I write this.

The underlying problem in both eras is the same — running untrusted user-supplied JavaScript without trusting it. The 2017 environment was a Node server where `eval` was available and a bad idea; the 2026 environment is a Worker where `eval` isn't available and would still be a bad idea. Same implementation solves both. Sometimes old code stays useful.

SVG, Mermaid, and avoiding a headless browser

Markdown input can include images, and some of those images are SVG — logos, badges, and any `<img>` tag pointing at an SVG URL. Embedding an SVG into a PDF is a choice. You could rasterise it to a PNG and drop the PNG in (loses fidelity, bloats the file). Or you can translate the SVG directly to PDF vector operators and get a crisp, scalable, small image.

makesPDF does the second thing. `@pdf/svg-renderer` parses the SVG, resolves styles, walks the tree, and emits PDF content stream operators directly — `m`, `l`, `c`, `re`, `S`, `f` and friends. Fill colours get looked up from the 148 CSS named colour table (plus rgb/rgba/hsl/hex). Opacity becomes an `ExtGState`. Transforms become matrix operations. Arcs get converted to cubic Béziers because PDF doesn't have a native arc primitive. Text inside SVG gets fed through the same font pipeline as body text, so the CID subsetter picks up the characters automatically.

The not-supported list is short — no gradients (`<linearGradient>`, `<radialGradient>`) and no filters. Both are doable; neither has come up often enough to prioritise. Everything else I've tested has come through looking like the source.

Mermaid is where this gets interesting. Mermaid is the library that turns ```` ```mermaid ```` code blocks into diagrams. Its normal rendering path is: parse diagram syntax → build a virtual DOM → render SVG via browser DOM APIs. Workers don't have a DOM.

An earlier version of the makesPDF approach was `@pdf/mermaid-shim` — a package that used `linkedom` to fake a DOM just well enough to get Mermaid to run. That worked, but it dragged in 76.5MB of installed dependencies (Mermaid itself plus its transitive deps plus linkedom), which is a lot of bundle to justify for a feature that most users don't even use.

The current version is `@pdf/mermaid-render`: a clean reimplementation of the parse → layout → render pipeline for all eight diagram types Mermaid supports. It vendors the jison-generated parsers from Mermaid's source, uses `elkjs` for graph layout where one's needed, and emits SVG XML directly — no DOM involved. Installed size is 7.7MB, roughly a tenth of the old version. SVG output is about 80% smaller too, because it's not trying to emulate a browser's SVG serialisation. And it's deterministic, which matters for caching.

I'd like to claim I planned this. I didn't. The first version was *get it working*; the second version came later when I'd used the feature enough to see what the actual shape of the problem was. Every time I've done this on a side project, it's worked that way. Get it working badly, understand the problem, rewrite cleanly. Don't do the clean version first — you don't know enough yet.

The whole request

Stepping back, here's what a `/api/v1/render` request looks like, end to end:

1. HTTP request arrives at the Worker. Auth check: bearer token or session cookie. (Every v1 endpoint is authed — see the API docs for the reasoning.)
2. Load the template from D1 (Cloudflare's SQLite). Deduct one credit per ten pages from the user's balance — but only on success.
3. Execute the DSL through scriptable. Get back a `DocumentDefinition` and some sample data.
4. Merge in the user-supplied data. Run the template engine to resolve variables, loops, conditionals.
5. Validate the resulting document against the catalog. If it's broken, return a 400 with specific issues.
6. Run the layout engine. 16 passes. Measures, positions, paginates, builds the render command list.
7. Hand the result to the PDF writer. Subset fonts. Embed images. Emit the content streams, the cross-reference table, the XMP metadata, the structure tree. Produce bytes.
8. Return the PDF.

All of that, end-to-end against production, measures well under a second for a small document and a couple of seconds for a 50KB markdown file. No container to wait on. No browser to boot. No filesystem reads. No external services in the hot path (D1 is local to the Worker's region). Just JavaScript doing what JavaScript does, in an isolate that started cold maybe a few milliseconds ago.

The honest version of "why fast" is mostly "because there's nothing slow in the way." It's not that any individual piece is particularly optimised — though some are. It's that the things that normally take the time in a PDF service (starting a browser, reading from disk, network round-trips to auxiliary services) aren't there.

Back to where we started

Post 1 ended with a list of reasons the fourth attempt at this product might be different. The first one on that list was *"running costs are near-zero at idle."* This post is the explanation of how that works.

When PDFUnicorn was running in 2014, it had nginx, Hypnotoad, MongoDB, and the app itself, all on a VPS that cost me money every month whether anyone used it or not. When Docca was running in 2018, it was four containers behind nginx, a Docker Compose setup, a database, and the Node process — more moving parts, more to monitor, more to keep patched, and the same monthly bleed.

makesPDF in 2026 is a single Worker bundle, a D1 database, an R2 bucket for PDFs, and a KV namespace for caching. When nobody's using it, it costs me basically nothing. When someone uses it, I pay for that request and nothing else. I don't have to shut it down if it doesn't grow fast enough. It can just sit there, available, not costing anything.

I couldn't say that about the 2014 or 2018 versions.

Thanks for reading. If any of it's useful, give the service a go — the free tier covers most small uses, and if you hit something that breaks, I'd like to know. 8)