Rewriting my blog in Rust

Isaac Bythewood · 2026-05-06 · 6 min read

rust webdev performance



I rewrote this blog from Flask to Rust over an afternoon. The result is a single 3.5 MB binary that uses 14x less memory and serves 10x more requests per second.

This blog started out as a Wagtail site, then a Django site, then a Flask app, and as of today it's a single 3.5 MB Rust binary that loads every post into memory at startup and serves the whole thing from a tokio runtime. I wrote about the URL design mistake from the Flask port last week, so this one is about what the Rust port actually changed.

The stack

The Flask version was about 540 lines in a single `app.py`, with Gunicorn out front, Mistune for markdown, Jinja2 for templates, and WeasyPrint for PDF export. Posts were loaded from `content/posts/*.md` on each request, parsed, then rendered.

The Rust version is roughly 1300 lines split across five files (`main.rs`, `posts.rs`, `markdown.rs`, `templates.rs`, `pdf.rs`). The pieces:

- axum for HTTP routing. Handlers are plain async functions, no macros.
- comrak for markdown. It has a `create_formatter!` macro that lets you override the HTML for individual node types, which is how I kept the same `div.block-*` wrapping the Mistune custom renderer produced.
- minijinja for templates. It accepts Jinja2 syntax verbatim, so the `templates/` directory came over without rewrites. Two whitespace tweaks and a parens fix on a ternary, that was it.
- A `chrome-headless-shell` subprocess for PDF export in place of WeasyPrint, which is the same idea with a different tool.

Posts are read from disk and parsed once at startup, so after that every request is just a hashmap lookup and a template render with no filesystem traffic or markdown work involved.

Benchmarks

I ran oha against both versions, ten-second bursts at fifty concurrent connections, with the production stack on each side (Gunicorn with four workers for Flask, release build for Rust). Both apps shared the same templates, the same content directory, and the same Vite-built static assets.

Loopback inside the same container, just framework overhead:

Route	Flask RPS	Rust RPS	Speedup
`/`	4,485	31,634	7.1x
`/blog/`	2,119	22,059	10.4x
`/posts/<slug>/`	4,134	43,732	10.6x
`/sitemap.xml`	6,190	67,257	10.9x
`/search/live/`	9,026	116,416	12.9x
`/og/<slug>.svg`	8,144	170,492	20.9x

p99 latency on `/blog/` dropped from 30.3 ms to 7.5 ms. The OG SVG endpoint sees the biggest gain because it's pure template rendering with no markdown work, so framework overhead dominates and Python pays it on every byte.

Container-to-container with Caddy in front was less dramatic but still real:

Route	Flask RPS	Rust RPS	p99 Flask	p99 Rust
`/`	1,616	2,511	45.9 ms	36.5 ms
`/blog/`	601	1,266	129.0 ms	69.3 ms
`/posts/<slug>/`	1,162	2,569	71.0 ms	34.9 ms
`/sitemap.xml`	1,966	5,178	870.5 ms	17.8 ms
`/search/live/`	3,002	8,149	29.6 ms	12.0 ms
`/og/<slug>.svg`	2,831	29,353	27.5 ms	3.9 ms

The Flask sitemap p99 of 870 ms isn't a typo. Under load there's a long tail you'd actually feel in production, where the Rust version comes in at 17.8 ms on the same route.

Memory

I wasn't expecting this part to be quite so lopsided.

Idle, post-warmup RSS:

- Flask, 4 Gunicorn workers: **347 MB**
- Rust, single process: **24 MB**

Under 50 concurrent sustained load:

- Flask: **147 MiB**
- Rust: **4 MiB**

The Flask number isn't really about Python being wasteful, it's that Gunicorn forks four workers and each one carries its own copy of Flask, Jinja2, Mistune, WeasyPrint, and a few hundred lines of imported modules, which is 80-something MB per worker before the app does any work at all. Tokio runs everything on one heap with a work-stealing pool so there's nothing to multiply.

On a $5 VPS that's the difference between the blog being something you have to budget for and something you can forget about.

What didn't get smaller

The container image got bigger rather than smaller, going from 854 MB on Flask to 1.16 GB on Rust, and the whole 800 MB of that is `chromium-headless-shell` for PDF export. Without PDFs the runtime image would be about 180 MB on alpine, or under 25 MB if I went static-musl on scratch, but I want PDFs more than I want a small image so chromium stays.

Iteration speed also got worse, since Flask reloads on save and Rust is `cargo build` every time, which is two to five seconds for an incremental debug build and around thirty seconds for a release build with LTO. `cargo watch -x run` makes it tolerable but you don't get the instant feedback loop you do in Python. I noticed it most while tweaking templates until I remembered that minijinja reloads templates from disk in debug mode without recompiling the binary, which covers most of what I was iterating on anyway.

Why a static-site generator wasn't the answer

The obvious answer to a slow blog is to stop running a server and generate HTML at build time, and I did think about it. The reason I didn't is that a few endpoints here have to be dynamic, like PDF export, server-rendered search, OG image generation per post, and redirect handling for the old `/blog/<slug>/` URLs. A static site would need a sidecar for all of that and the sidecar needs a runtime of its own, so I'd end up with two deploy targets instead of one.

The Rust binary handles all of it in one process, parses every post at boot in single-digit milliseconds, and idles at 24 MB, which is close enough to static for me.

Was it worth it

The speedup is honestly more than I needed, since this blog gets nowhere near 1,000 RPS in real traffic. The things I ended up appreciating were the ones I wasn't looking for, like the memory drop, the long tail on the sitemap going away, and the binary starting in under fifty milliseconds and refusing to start at all if a post has malformed frontmatter. That last one caught two stale fields in old posts I'd forgotten about.

If you've got a small Flask or Django app whose footprint annoys you more than its speed does then axum with minijinja and comrak is a surprisingly direct port. The Jinja2 templates carried over almost untouched, the custom Mistune renderer mapped one to one onto a comrak formatter, and the deploy didn't change at all.