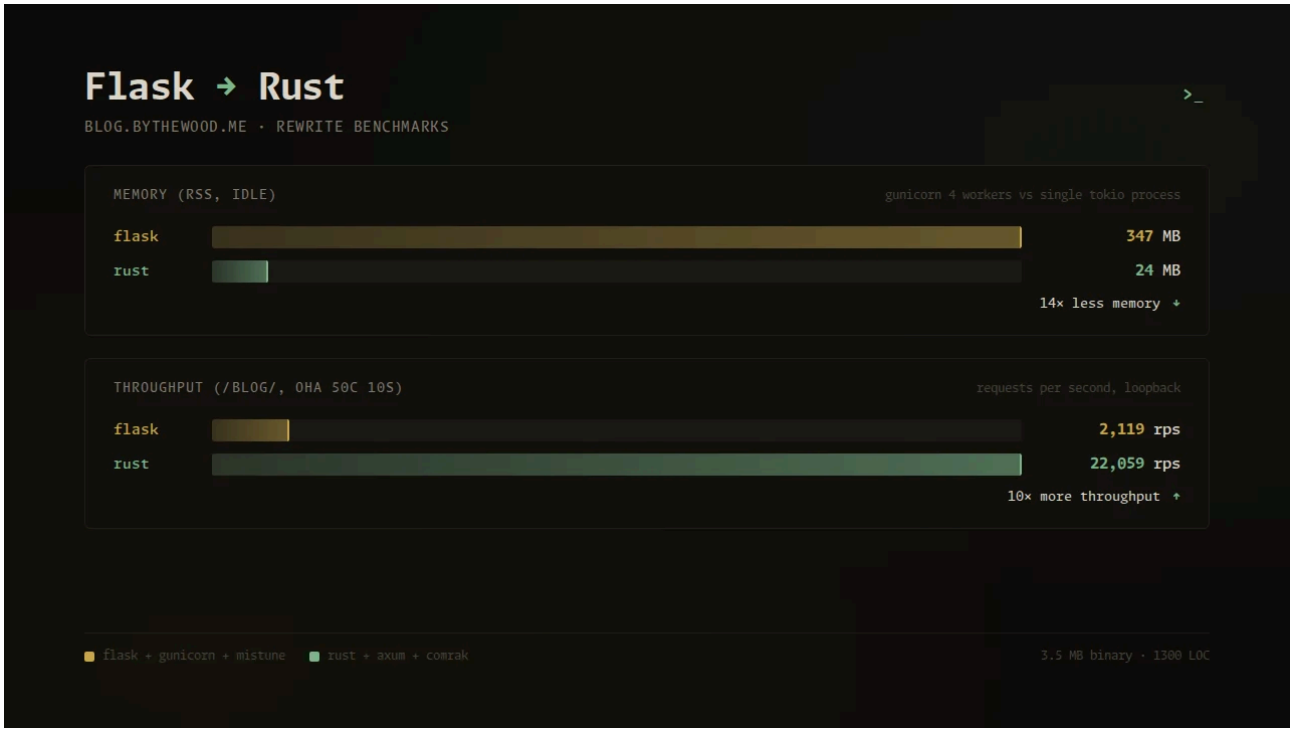
Rewriting my blog in Rust

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 used to be a Wagtail site. Then a Django site. Then a Flask app. As of today it's a single Rust binary, 3.5 MB, 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. This is the follow-up: 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, replacing WeasyPrint. Same idea, different tool.

Posts are read from disk and parsed once at startup. After that every request is a hashmap lookup and a template render. No filesystem traffic on the hot path, no markdown work per request.

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 is a real number, not a typo. Under load there’s a long tail you’d feel in production. The Rust version’s p99 on the same route is 17.8 ms.

Memory

This is the part I wasn’t expecting 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 because Python is wasteful in a deep philosophical sense, it's because Gunicorn forks four workers and each one carries its own copy of Flask, Jinja2, Mistune, WeasyPrint, and a few hundred lines of imported modules. That's 80-something MB per worker before the app does any work. Tokio runs everything on one heap with a work-stealing pool, so there's nothing to multiply.

For a $5 VPS this is the difference between "the blog is a rounding error on the box" and "the blog is a tenant I have to think about." The blog is now a rounding error.

What didn't get smaller

The container image got bigger, not smaller. The Flask image was 854 MB. The Rust image is 1.16 GB. The whole 800 MB delta is `chromium-headless-shell`, which is required 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. I want PDFs more than I want a small image, so chromium stays.

Iteration speed also got worse. Flask reloads on save. 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 this most while tweaking templates, until I remembered minijinja reloads templates from disk in debug mode without recompiling the binary. That covers most of what I was iterating on anyway.

Why a static-site generator wasn't the answer

The obvious response to "your blog is too slow" is "stop running a server, generate HTML at build time." I considered it. The reason I didn't is the same reason I rewrote rather than retired the app: I have a few endpoints that are dynamic by design. PDF export, server-rendered search, OG image generation per post, redirect handling for the old `/blog/<slug>/` URLs. A static site would need a sidecar for all of that, and the sidecar would itself need a runtime. At which point you have two deploy targets to worry about, not one.

The Rust binary handles everything in one process, parses every post at boot in single-digit milliseconds, and idles at 24 MB. That's close enough to "static" for me.

Was it worth it

The honest answer is "the speedup is more than I needed." This blog gets nowhere near 1,000 RPS in real traffic. The actual wins were the ones I didn't go in looking for: the memory drop, the disappearance of the long-tail latency on the sitemap, the fact that the binary starts in under fifty milliseconds and refuses to start at all if any post has a malformed frontmatter. The compiler 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, axum plus minijinja plus 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. The Dockerfile got shorter. The deploy is the same `git push server master`. The blog is faster than it has any right to be.