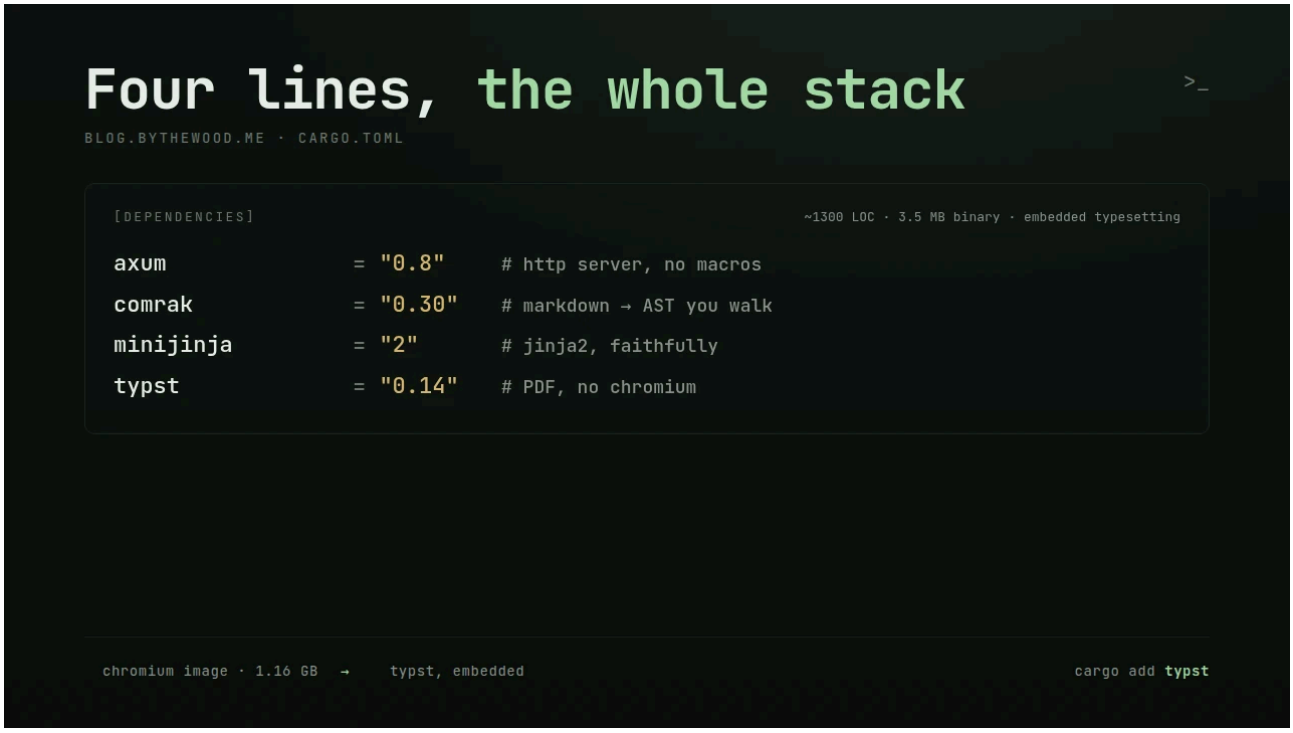
The Rust ecosystem is unreasonably good

Isaac Bythewood · 2026-05-09 · 5 min read

rust webdev performance



A second pass on the Rust port of my blog. I deleted the chromium PDF subprocess and replaced it with embedded Typst. Notes on axum, comrak, minijinja, and a typesetting compiler that ships as a crate.

A few days ago I rewrote this blog from Flask to Rust. The benchmarks were the headline. What I didn't write up: a day later I deleted `chrome-headless-shell` from the runtime image and replaced it with Typst, embedded as a library. The Docker image lost most of a gigabyte. The PDF route stayed the same shape.

This is the follow-up. A closer look at the four crates the blog actually runs on.

axum

axum is small. A handler is an async function, its arguments are extractors, and its return type implements `IntoResponse`.

```
pub async fn show(
    State(s): State<AppState>,
    Path(slug): Path<String>,
) → impl IntoResponse {
    // ...
}
```

State is a clone-cheap struct, `Arc`'d once at startup. Routers compose with `merge`, so I keep one router file per feature (`routes/post.rs`, `routes/blog.rs`, `routes/search.rs`, `routes/seo.rs`) and stitch them together in `app.rs`:

```
Router::new()
    .merge(routes::home::router())
    .merge(routes::blog::router())
    .merge(routes::post::router())
    .merge(routes::search::router())
    .merge(routes::seo::router())
    .nest_service("/static", static_files)
    .nest_service("/content/images", images)
    .fallback(routes::errors::not_found)
    .layer(axum_middleware::from_fn(log_requests))
    .with_state(state)
```

Middleware is a tower layer, so request logging, cache headers on static files, and the 404 fallback all use the same shape. The whole request logger is twenty lines:

```
pub async fn log_requests(req: Request, next: Next) → Response {
    let method = req.method().clone();
    let path = req.uri().path_and_query()
        .map(|p| p.as_str().to_string())
        .unwrap_or_default();
    let start = Instant::now();
    let response = next.run(req).await;
    let elapsed_ms = start.elapsed().as_secs_f64() * 1000.0;
    let status = response.status().as_u16();
    eprintln!("{method:<5} {status} {elapsed_ms:>7.2}ms  {path}");
    response
}
```

I haven't pulled in `tracing` yet and I don't expect to.

comrak

comrak parses CommonMark + GFM into an AST. Most markdown libraries either render straight to HTML or hand back an event stream, both of which make non-trivial customization annoying. comrak gives you the tree.

I render every post twice from the same source. Once to HTML for `/posts/<slug>/`, once to Typst markup for `/posts/<slug>/pdf/`. Both walks read the same arena, so a typo in markdown fails both renders identically.

For HTML, comrak's `create_formatter!` macro overrides individual node types and inherits the rest. I use it to wrap blocks in `div.block-*` classes the CSS hooks into, the same shape the Mistune custom renderer in the Flask version produced. The Typst pass is a hand-written walker, about 250 lines in `src/pdf.rs`.

minijinja

I came in expecting to rewrite my templates. I didn't have to. minijinja, by Armin Ronacher (who also wrote Jinja2), is faithful enough that the entire `templates/` directory came over with two whitespace tweaks and a parens fix on a ternary.

Two things to know:

- Jinja2 escapes `/` in URLs to `&#x2f;`. minijinja doesn't, which is technically more correct, but it broke the OG image template and a couple of expected-string snapshots. Thirty lines of formatter to match Jinja2 fixed it.

- In debug builds, minijinja re-reads templates from disk on every render. Gate the loader on `cfg(debug_assertions)` and you get template hot-reload without restarting `cargo run`.

Typst, embedded

Typst is a typesetting system. The reason I care: the entire compiler is on crates.io. typst, typst-pdf, and typst-kit for font discovery. No binary to ship alongside the app, no LaTeX install, no subprocess. You call `typst::compile(&world)` and get back a `PagedDocument`. You call `typst_pdf::pdf(&doc, ...)` and get bytes.

End to end:

```
let main = Source::new(
    FileId::new(None, VirtualPath::new("/main.typ")),
    source,
);
let world = PdfWorld { library, book, fonts, root, main };
let document = typst::compile::<PagedDocument>(&world).output?;
let bytes = typst_pdf::pdf(&document, &PdfOptions::default())?;
```

The interesting type is `World`. It's the trait Typst uses to ask "give me the source for this file id, give me the bytes for this asset, give me a font by index, give me today's date." You implement it once. Mine resolves Typst paths against the project root, so a snippet like:

```
#image("/content/images/cover.webp")
```

reads `content/images/cover.webp` from the running binary's working directory. Same on macOS, alpine, and CI. No bind mounts, no file URIs, no temp files.

Fonts are found once at startup with `typst-kit`'s `FontSearcher`. The runtime alpine image installs `font-jetbrains-mono`, `ttf-dejavu`, and `ttf-liberation` so there's always a sans, mono, and fallback available.

The size delta is what got me. The chromium runtime image was 1.16 GB. The Typst image is a few hundred MB, most of it the font packages themselves. The PDF route used to spawn a process and write a temp file; now it's a function call.

The other PDF tools I've shipped (WeasyPrint, wkhtmltopdf, headless Chromium) all add a binary to the runtime image and a process boundary at request time. Typst doesn't.

What I keep noticing

Coming from uv on the Python side, `cargo add` and `Cargo.lock` felt familiar. uv already does the single-tool, single-lockfile thing for Python; on the Rust side it's the same loop. The trade is at build time. A Docker image for this blog takes tens of seconds to build incrementally and a couple of minutes cold, where the Flask + uv version was a few seconds either way. In exchange I get a binary that idles at 24 MB and serves an order of magnitude more traffic. I'll take that.

Underneath axum, comrak, minijinja, and typst, the project pulls in tokio for the runtime, tower-http for middleware and static files, serde for frontmatter parsing, chrono for dates, and anyhow for error handling. The whole `Cargo.toml` fits on a screen.

Every time I've gone looking for something in Rust, the ecosystem has had a good answer ready. So far it hasn't missed.