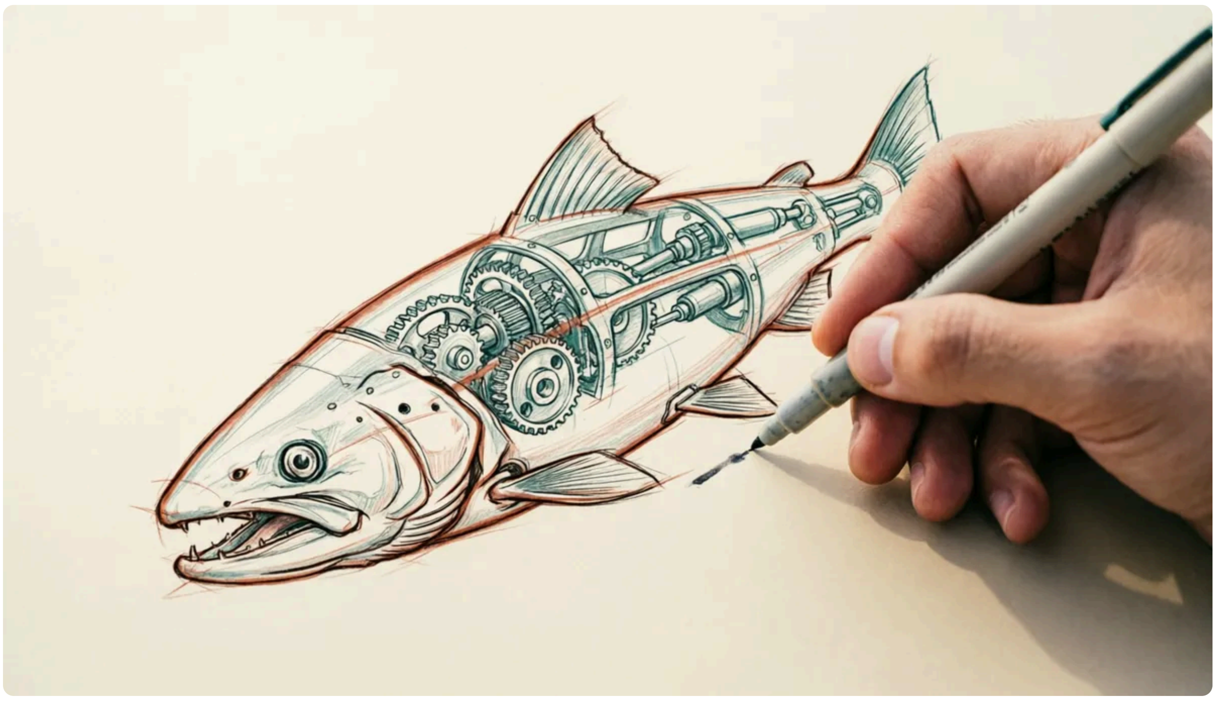
Images on a static site: tradeoffs, limits, and a 150-line Rust optimizer

Emil Lindfors | February 11, 2026



Every blog post I've written so far has been pure text. Code blocks, citations, math – but no images. Now that I'm writing about sensor rigs and aquaculture sites, I need photographs. That raised a question I hadn't needed to answer yet: where should the images live?

The options

There are three reasonable approaches for a static site deployed to Cloudflare Pages.

Co-located in git. Images sit next to the markdown file in the same directory. Zola copies them to the output at build time. Simple, version-controlled, no external dependencies.

Object storage (Cloudflare R2, S3, etc.). Images live in a bucket, referenced by URL. The git repo stays lean. Adds operational complexity: a separate upload step, URL management, another service to configure.

Inline as base64. Encode images directly into the HTML. No extra files to serve. Terrible idea in practice: 33% size overhead from the encoding, defeats browser caching (the image is re-downloaded with every page load), and bloats the HTML the browser has to parse before rendering anything.

Why I chose git

This site's design principle is everything in one repo. The fonts are self-hosted. The newsletter runs on a Cloudflare Worker checked into the same repository. The PDF generation pipeline reads from the same markdown files. Adding a second deployment target for images would break that model.

The practical question is whether git can handle it. The answer is yes, easily, for a blog.

The math

A well-optimized blog image – WebP format, 1200 pixels wide, quality 80 – is typically 80-150 KB. Diagrams and charts as SVG are even smaller, often under 10 KB.

At 150 KB average, here's how far you can go:

Images	Repo size	Git performance
100	~15 MB	No impact
500	~75 MB	No impact
1,000	~150 MB	Fine
2,000	~300 MB	Still fine

Git starts feeling sluggish around 500 MB to 1 GB of binary content. That's somewhere between 3,000 and 7,000 optimized blog images. I'm not going to write 3,000 blog posts.

Cloudflare Pages itself has generous limits: 20,000 files per deployment, 25 MB max per file. Neither is a concern for blog images.

When object storage makes sense

If I start hosting video, large downloadable datasets, or high-resolution photo galleries with dozens of full-res images per post, then R2 with a custom subdomain would be the right move. The migration is straightforward since it's just changing image paths from relative to absolute URLs. But that's a bridge to cross when I'm actually at it.

The optimization tool

The key to making co-located images work is optimizing before committing. A 4 MB DSLR photo has no business being in a git repository or served to a browser. What you want is a WebP at a sensible resolution.

I built a small Rust CLI for this. Two crates: `image` for decoding and resizing, `webp` (which wraps libwebp) for lossy WebP encoding with quality control.

Cargo.toml

```
[package]
name = "img-optim"
version = "0.1.0"
edition = "2021"

[dependencies]
image = { version = "0.25", default-features = false, features = [
    "jpeg", "png", "gif", "bmp", "tiff"
] }
webp = "0.3"

[profile.release]
opt-level = 3
```

The `image` crate's default features pull in a pure-Rust WebP encoder that only supports lossless encoding. For lossy with quality control, you need the `webp` crate, which links against libwebp. On Ubuntu/Debian that's `apt install libwebp-dev`.

The core logic

The interesting part fits in three functions:

```
fn resize_to_width(img: image::DynamicImage, max_width: u32) -> image::DynamicImage {
    let (w, h) = img.dimensions();
    if w > max_width {
        let new_h = (max_width as f64 / w as f64 * h as f64) as u32;
        img.resize_exact(max_width, new_h, image::imageops::FilterType::Lanczos3)
    } else {
        img
    }
}

fn encode_webp(
    img: &image::DynamicImage,
    path: &Path,
    quality: f32,
) -> Result<(), Box<dyn std::error::Error>> {
    let encoder = webp::Encoder::from_image(img)
        .map_err(|e| format!("webp encode: {e}"))?;
    let data = encoder.encode(quality);
    fs::write(path, &*data)?;
    Ok(())
}
```

```
fn optimize(
    path: &Path,
    max_width: u32,
    quality: f32,
) -> Result<(u64, u64, PathBuf), Box<dyn std::error::Error>> {
    let before = fs::metadata(path)?.len();
    let img = image::open(path)?;
    let img = resize_to_width(img, max_width);

    let out_path = path.with_extension("webp");
    encode_webp(&img, &out_path, quality)?;

    let after = fs::metadata(&out_path)?.len();
    Ok((before, after, out_path))
}
```

`resize_to_width` uses Lanczos3 resampling, which is the right choice for downscaling photographs. It preserves sharpness better than bilinear or bicubic. The function is a no-op if the image is already within the target width.

`encode_webp` wraps the libwebp encoder. Quality 80 is a good default – visually indistinguishable from the original for blog-sized images, with significant size savings.

Thumbnails

Featured images on the front page don't need to be 1200 pixels wide. A 600px thumbnail at slightly lower quality is enough for a card layout. The tool generates these with the `-t` flag:

```
const THUMB_WIDTH: u32 = 600;
const THUMB_QUALITY: f32 = 75.0;
const THUMB_SUFFIX: &str = "-thumb";

fn thumbnail(
    path: &Path,
    width: u32,
    quality: f32,
) -> Result<(u64, PathBuf), Box<dyn std::error::Error>> {
    let img = image::open(path)?;
    let img = resize_to_width(img, width);

    let stem = path.file_stem().unwrap().to_string_lossy();
    let out_path = path.with_file_name(format!("{stem}{THUMB_SUFFIX}.webp"));
    encode_webp(&img, &out_path, quality)?;

    let size = fs::metadata(&out_path)?.len();
    Ok((size, out_path))
}
```

The naming convention is deterministic: `hero.jpg` becomes `hero.webp` (full size) and `hero-thumb.webp` (thumbnail). The template derives the thumbnail path from the frontmatter `featured_image` field using a string replace.

Results

Running the tool against a couple of test images:

```
$ img-optim -t photo.jpg diagram.png
  photo.jpg -> photo.webp (47.7 KB -> 22.3 KB, -53%)
  photo.jpg -> photo-thumb.webp (17.3 KB)
  diagram.png -> diagram.webp (261.6 KB -> 7.5 KB, -97%)
  diagram.png -> diagram-thumb.webp (2.6 KB)

  Total: 309.3 KB -> 49.7 KB (-84%)
```

PNG diagrams compress spectacularly well. Photographs see significant reduction from the combined effect of format conversion, quality reduction, and resolution capping.

The workflow

The complete process for adding images to a blog post:

```
content/blog/my-post/
├── index.md
├── hero.webp          # featured image (1200px, q80)
├── hero-thumb.webp    # thumbnail for index page (600px, q75)
├── sensor-rig.webp    # inline image
└── results-chart.svg  # diagrams stay as SVG
```

The frontmatter:

```
[extra]
featured_image = "hero.webp"
```

In markdown, images use relative paths:

```
![The sensor rig mounted on the cage](sensor-rig.png)
```

The commands:

```
# Drop raw images into the post directory, then:
img-optim -t content/blog/my-post/hero.jpg
img-optim content/blog/my-post/sensor-rig.jpg

# Delete originals, commit the .webp files
rm content/blog/my-post/*.jpg
```

The `-t` flag is only needed for the featured image. Inline images don't need thumbnails.

What the template does

The `featured_image` field drives several things automatically:

- **Post header:** displays the full-resolution image below the title
- **Front page cards:** shows the thumbnail on featured post cards
- **OpenGraph / Twitter Card:** uses the image for social sharing previews
- **JSON-LD structured data:** includes the image for search engine rich results

If `featured_image` isn't set, the post renders exactly as before – text only. No fallback placeholder, no broken layout.

Format recommendations

WebP for photographs and screenshots. Lossy compression at quality 80 is the sweet spot. Virtually all browsers support it now.

SVG for diagrams, charts, architecture drawings, and anything with clean lines and text. Infinitely scalable, tiny file size, and they respect the site's light/dark theme if you use `currentColor`.

Avoid PNG for photographs – it's lossless, which means large files for no perceptible quality gain on a blog. PNG is fine for screenshots where you need pixel-perfect text, but WebP lossless handles that too at smaller sizes.

Avoid JPEG as the final format. WebP at equivalent visual quality is 25-35% smaller. Use JPEG as a source format that you convert from.

A note on the hero image

The featured image on this post was generated with Qwen-Image-2.0, Alibaba's open-source image generation model released in February 2026. It supports native 2K resolution and handles text rendering well, which makes it useful for generating decorative hero images when you don't have a photograph that fits.

For a technical blog, most images will be screenshots, diagrams, or photographs of actual hardware. But for posts like this one – where the topic is abstract – an AI-generated image works as a visual anchor without misrepresenting anything. The image went through the same `img-optim` pipeline as any other: generated at high resolution, then converted to a 1200px WebP with a 600px thumbnail.

This post is part of a series on the infrastructure behind this blog. See also: Site overview, Self-hosted newsletter, Citations, Typst PDF generation.