Building a Home for Twenty Thousand Photographs

April 2026

13 min read • 3,028 words

A photograph without a home is a memory without a body.

Three days ago I wrote about sixty thousand images and nowhere to put them. A meditation on creative work without a platform, on the death of photo-sharing communities, on the particular ache of possessing a body of work that exists in a vacuum. That essay ended with resignation: The moments are still out there. I just can't catch them anymore. I meant it when I wrote it. I also couldn't leave it alone.

Something about writing the lament cracked open the energy to build the answer. Maybe articulating the problem clearly enough is itself a form of compilation. The act of naming what's wrong makes the architecture of what's right suddenly visible. I've noticed this pattern in programming as spiritual practice: the debugging is the understanding. You don't debug and then understand. The careful attention to what's broken reveals what wholeness looks like.

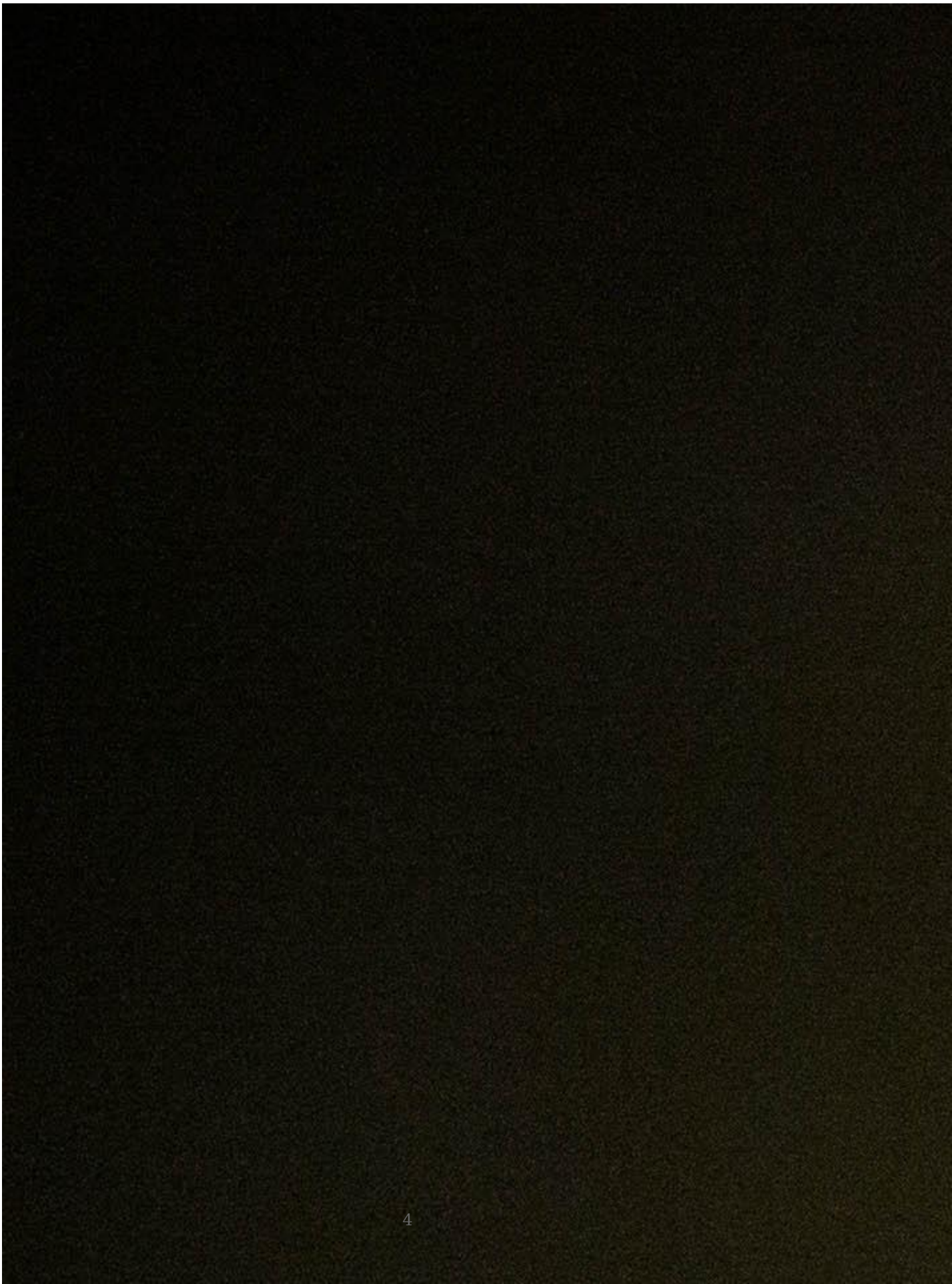
So I sat down with Claude and built the place.

photos.kennethreitz.org is live. Twenty thousand photographs, organized by camera, lens, city, year, and AI-generated tags. Every image has EXIF metadata extracted, GPT-4o-generated titles and descriptions, and is discoverable through

the gear that made it. The whole thing was built in a single extended session. From empty Django project to deployed production site with infinite scroll, a photo manager, and a word cloud of three thousand tags.

I want to talk about how it works, because the architecture reveals something I keep circling back to: the recursive loop between tools and consciousness. We build tools that change how we see, and then we see differently, and then we build different tools. This project sits right at the center of that loop — software built to see my own seeing.

The Stack



The foundation is Django 6 on Python 3.14, deployed on Fly.io. PostgreSQL for data and as the Celery broker (no Redis in production, one less service to manage). Tigris for S3-compatible object storage. HTMX for interactivity. Vanilla JavaScript only where absolutely necessary: drag-and-drop upload, the photo manager's multi-select.

No frontend framework. No React, no Vue, no build step. Django templates render HTML, HTMX handles infinite scroll, the browser does the rest.

This is a deliberate choice rooted in the same philosophy that drove Requests: match the mental model. A photography site should feel like looking at photographs, not like operating a web application. The server renders pages. The browser displays them. Scroll down, more images appear. Click one, see it large with its EXIF data. That's it.

> The "for humans" principle keeps showing up everywhere I build. HTTP should feel like language. Photography browsing should feel like looking at photographs. The interface should disappear until what remains is the experience itself. When you have to think about the tool, the tool has failed.

The complexity lives in the pipeline, not the presentation. This distinction matters more than most architects realize. The user's experience of simplicity should be proportional to the system's hidden sophistication, never the reverse.

The Pipeline

Every image that enters the system passes through a ten-step pipeline:

1. Validate format and size.
2. Extract EXIF metadata.
3. Normalize camera and lens names, because EXIF strings are chaos. "NIKON CORPORATION NIKON D850" and "Nikon D850" need to resolve to the same canonical record.
4. Compute a perceptual hash for visual deduplication.
5. Generate three thumbnail sizes.
6. Create the EXIF data record with the raw JSON preserved.
7. Reverse geocode GPS coordinates to a city (offline, using a local dataset, no API rate limits).

8. Apply cleanup rules: delete images from certain dates, clear incorrect EXIF timestamps, block GPS coordinates from countries I've never visited.
9. Mark as processed.
10. Dispatch an async AI description task.

Steps 1 through 9 happen synchronously during upload. Step 10 fires off a Celery task that sends the thumbnail to GPT-4o-mini and gets back a structured JSON response: an artistic title, a two-sentence description, and five to ten tags.

> OpenAI's structured output with JSON schema enforcement means the response is always valid JSON with exactly the fields you need. No parsing hacks, no "please respond in JSON" prompting. The schema is the contract. This is the kind of human-machine interface design I admire — declare your expectations clearly, and the system meets them.

There's something meditative about designing a pipeline like this. Each step has exactly one job. Each step trusts the steps before it. The whole thing is a chain of small certainties building toward a larger one — that any image entering the system will emerge on the other side understood, categorized, and findable. It's the same principle behind good API design: make the common case effortless, handle the edge cases gracefully, never surprise the caller.

The cleanup rules are the part I'm most proud of architecturally. They're defined in two places: a management command for batch operations, and inline in the pipeline for real-time enforcement. Delete rules remove images from years with bad data. Fix rules clear incorrect timestamps. Privacy rules make certain dates private. City rules block GPS coordinates that geocode to countries I've never been to (bad EXIF data from camera clock drift, not actual travel). The same rules apply whether you're importing ten thousand images from a hard drive or uploading one from the web interface. One truth, enforced everywhere. The system has opinions, and they're consistent.

The AI Layer

Every photograph gets an AI-generated title, description, and set of tags. The titles are short and evocative. "Golden Hour Over the Valley", "Urban Geometry in Shadow", "Whispers of the Waterfront". The descriptions are two to three sentences. The tags are single lowercase words: architecture, shadow, reflection, monochrome, street.

This matters more than it might seem. My Flickr exports had filenames like "Pro Photos - 7025 of 11810.jpeg". The original Leica files were "L1006175.jpg". Neither tells you anything about what's in the image. The AI layer transforms an archive of opaque filenames into a searchable, browsable, discoverable collection. It gives language to the visual. It makes the silent speak.

The tag cloud alone (three thousand tags, sized by frequency, filterable as you type) turns the archive from a chronological dump into something you can explore by subject. Click cobblestone and see every street I ever walked on with interesting ground. Click silhouette and see a decade of figures against light. Click reflection and watch patterns emerge that I never consciously intended but clearly couldn't stop making.

Search spans titles, AI descriptions, and tags simultaneously. Type "rain" and you get every image the AI recognized rain in, across every camera and city and year. That's not something I could have built by hand with twenty thousand images. This is what human-AI collaboration looks like when it's working well. Not AI replacing human judgment, but AI providing the infrastructure for human discovery. The machine sees the surface. I see the meaning. Together we make something neither could alone.

The Import

The bulk import command (`import_folder`) was the workhorse. Point it at a directory, it recurses through subdirectories, hashes each file against the database to skip duplicates, uploads originals to Tigris, and dispatches processing. A single worker doing one image at a time, because the remote Postgres can't handle concurrent connections over a Fly proxy without exhausting the connection pool.

The auto-skip is smarter than it looks. It preloads all existing filenames from the database in a single query, normalizes them (spaces to underscores, case-insensitive), and skips matches instantly. No disk I/O required for known images. Only genuinely new files get hashed. For a re-run against twenty thousand existing images and a few hundred new ones, the skip phase takes seconds.

Perceptual deduplication catches what content hashing misses. Two different exports of the same photograph (one from Flickr at 1600px, one from the original at full resolution) have different SHA-256 hashes but nearly identical perceptual hashes. The deduplication pass compares hamming distances and removes the duplicate while preserving collection memberships, tags, and AI metadata on the surviving copy. Identity isn't about byte-level equality. It's about what the image is. The perceptual hash understands that in a way the cryptographic hash never will.

The Geography

GPS coordinates in EXIF data are a gift and a curse. A gift because they let you build a browsable map of everywhere you've ever photographed. A curse because camera clocks drift, and suddenly you have sixty photographs allegedly taken in western China when you were actually in Virginia.

The solution is a four-level filter. The `City.from_coordinates()` method rejects coordinates that geocode to countries on an exclusion list. The processing pipeline checks the same list. The geocode management command skips them during batch operations. And the cleanup command catches anything that slipped through. Four independent checks, same list, same result. Belt, suspenders, and two extra belts.

India required special handling. I've actually been to Bangalore and Mysore, so the filter allows those cities while rejecting everything else in the country. Substring matching, because the reverse geocoder returns "Bangalore Urban" as the admin2 region, not "Bangalore" as the city name. Reality is messy. Good systems acknowledge the mess and route around it rather than pretending it doesn't exist.

The cities page groups locations by continent, country, and state for the US, with an interactive dark-themed map showing gold markers sized by image count. Click a marker, see the city name and a link to browse its photos. It's the kind of feature that makes the archive feel alive. Not just a collection of images but a map of a life. A geography of attention. Fourteen years of choosing where to point a camera, rendered as data, revealing patterns of presence I didn't know I had.

The Architecture That Evolved

This is the part of the story I keep coming back to, because it taught me something I should have already known.

The original vision wasn't a personal photography site. It was a platform. Multi-tenant architecture. User registration, per-user storage quotas, shared tag taxonomies, the works. I was going to build the photography platform that didn't exist anymore, the one I'd been mourning in the lament essay. If Flickr was dead and Instagram had been hollowed out by engagement optimization, surely the answer was to build something new. Something for photographers. Something for humans.

It took a surprisingly difficult moment of honesty to abandon that vision. Multi-tenant architecture touches everything. Database queries need tenant scoping. Storage needs per-user isolation. Every feature has to consider permissions, visibility, abuse vectors. The complexity isn't in any single decision; it's in the way every decision compounds. I was building infrastructure for a community that didn't exist yet, instead of building a home for photographs that did.

So I ripped out multi-tenancy. Single user. Single tenant. My photographs, my site, my rules. It felt like failure at first, like giving up on the bigger idea. But it was actually the first honest architectural decision of the project. Build for the reality you have, not the platform you imagine. The community can come later, if it comes at all. The photographs needed a home now.

That decision unlocked everything else. Without multi-tenancy, the data model simplified dramatically. Without user registration, the auth story collapsed to one admin account. Without shared taxonomies, the AI could generate tags freely without worrying about namespace collisions. Every layer of complexity I removed made the next feature easier to build.

The site started as an API-first application. django-bolt, an async Django framework, served a full REST API with JWT authentication, and Alpine.js powered the frontend. Every page loaded data from API endpoints, rendered it client-side, stored tokens in localStorage. The modern way. The way you're supposed to do it.

I hated it within hours.

Alpine kept not working in Safari. The JWT dance (store the token, check it on every page load, redirect to login if it fails, clear it on logout) was fragile ceremony for a single-user site. Every page required JavaScript to render anything. The upload page's drag-and-drop didn't work because Alpine wasn't loading. The dashboard was a blank white page until three API calls resolved.

> This is the complexity trap I write about in the context of the Algorithm Eats series, but applied to ourselves. We build systems that optimize for imagined future requirements rather than present human experience. The same pattern that makes platforms hostile to users can make our own tools hostile to us.

So we ripped it out. Replaced Alpine with HTMX. Replaced JWT with Django's built-in session authentication. Replaced client-side rendering with server-rendered templates. The login page went from a JavaScript form that POST'd to an API endpoint and stored a token to a Django `LoginView` with a `<form method="post">`. The nav went from a JavaScript auth check that runs on every page to `{% if user.is_authenticated %}`.

The entire frontend got simpler by an order of magnitude.

And here's the thing. Bolt is still in there. The ASGI server, the API endpoints, the router infrastructure. It serves the application. It's the process that runs in production. The bolt API endpoints still exist and still work. You can hit `/api/cameras` and get JSON. Nothing calls them anymore, but they're there.

I'm proud of that. Not because it's good architecture. It's objectively unnecessary, a full API framework serving an application that doesn't use the API. But seeing bolt run in production on a real site with twenty thousand images feels good in a way that's hard to explain. It's like keeping a hand-built engine in a car you mostly drive to the grocery store. Overengineered for the task. Beautiful in its own right. And it works.

There's a deeper lesson here, and it connects to something I've been thinking about in the context of programming as spiritual practice. The architecture that evolved wasn't the one I planned. The architecture I planned was the "right" one: API-first, client-side rendering, token-based auth, separation of concerns. The architecture that actually works is the one that emerged from honest engagement with what the project needed. Templates. Sessions. Server-rendered HTML. The technologies of 2005, proven and boring and correct.

The lesson, if there is one: start with the simplest thing that works. Django templates and session auth are the simplest thing. HTMX is the simplest interactivity. Vanilla JavaScript is the simplest client-side logic. Every layer of abstraction you add (API endpoints, JWT tokens, client-side rendering frameworks) is a layer you have to debug, maintain, and explain. Sometimes you add those layers because you need them. Sometimes you add them because you think you're supposed to. The difference matters. And the willingness to rip out what isn't working, even features you spent hours building, is a form of honesty that code demands and ego resists.

> I've watched engineers defend architectural decisions long after they've proven wrong, because admitting the mistake feels like admitting incompetence. But the opposite is true. The willingness to evolve is the competence. The code doesn't care about your feelings. It cares about working.

What the Mirror Shows

The most interesting thing about this project isn't the technology. It's what happens when you give twenty thousand photographs AI-generated metadata and make them searchable.

Patterns emerge that I never saw in fourteen years of shooting. The AI notices things I didn't. Recurring compositional tendencies, subjects I'm drawn to unconsciously, a bias toward certain kinds of light that I wasn't aware of until I could search for it. The tag cloud is a mirror. It shows you what you see when you look at the world.

I keep clicking through tags and discovering myself. There's an overwhelming bias toward shadow and silhouette and contrast. The interplay between presence and absence, the visible defined by what's hidden. There's a persistent pull toward geometry and pattern, the mathematical structure underlying the organic world. There's more solitude than I expected and less crowd. The tags don't just describe photographs. They describe a consciousness — the particular way I've been carving attention out of the visual chaos for fourteen years.

This is the recursive loop made visible. I built a tool. The tool shows me how I see. Seeing how I see changes how I see. I'll shoot differently now, knowing what the tag cloud reveals. And the new photographs will change the tag cloud, which will change what I notice, which will change what I shoot. Code shapes consciousness shapes code. The loop doesn't end. It deepens.

Photography is already a consciousness practice. The discipline of noticing, of being present enough to see what's actually in front of you rather than what you expect to see. Every photographer knows this. What I didn't expect was that building software around photography would intensify that practice. The AI layer doesn't just organize images. It provides feedback on the act of seeing itself. It's a mirror for attention.

From Lament to Architecture

Three days between writing the lament and deploying the solution. That's not a brag about speed. It's an observation about what becomes possible when you stop mourning the platform that doesn't exist and start building the one you need. The platforms that understood photography as art have been hollowed out by the same engagement machinery that eats everything beautiful. The response isn't nostalgia. It's architecture.

The sixty thousand images have a home now. Not all of them. Cleanup rules and curation brought the public count to around twenty thousand. But the ones that matter. Browsable by camera, lens, city, year, and tag. Every image discoverable. Every photograph findable by the gear that made it or the subject it captured.

The platform I was looking for didn't exist. So I built it. And in building it, I discovered something I wasn't looking for. A tool that doesn't just store photographs but reflects the photographer back to himself. A tag cloud that maps not just subjects but the shape of a consciousness. An architecture that evolved from overengineered ambition to honest simplicity, which is maybe the only architectural evolution that ever matters.

We sit at the center of the feedback loop between code and consciousness. What we build to see with changes what we see. What we see changes what we build. The loop is the practice. The practice is the point.

The site is live at photos.kennethreitz.org. The code is at github.com/kennethreitz/photos.kennethreitz.org.