LEARN · THE STACK

Cloudflare Pages + Workers: The Static-but-Dynamic Stack

Cloudflare Pages handles your static surface. A small fleet of Workers handles the dynamic surface payments, forms, webhooks, redirects. Together they cost almost nothing, serve from 330+ edge cities, and require no servers to patch. This is the architecture we run Pillar on, explained for engineers who are choosing between Vercel, Netlify, AWS, and Cloudflare for a content-heavy business.

---

11 min read

Last updated: June 10, 2026

> *Most businesses do not need a server. They need fast static HTML for 99% of traffic and a thin layer of serverless logic for the 1% that actually moves money or data. Cloudflare Pages plus Workers is the cleanest expression of that split in 2026.*

The thesis

- Why the static-but-dynamic split is the right mental model for content-heavy sites in the AI search era
- How Cloudflare's global edge network changes the latency math for SEO and conversion
- The exact role of Pages (static surface) versus Workers (dynamic surface) and where to draw the line
- Concrete setup steps using the wrangler CLI to deploy a project from zero
- Why this stack is cheaper, faster, and lower-maintenance than Vercel, Netlify, or AWS S3 + CloudFront for the same workload

01 — The framework: The Static-but-Dynamic

Sandwich

The Static-but-Dynamic Sandwich

Think of your site as a sandwich. The bread is static HTML cached at every edge city in the world. The filling is a thin slice of dynamic logic that runs only when it has to: a payment redirect, a form handler, a webhook receiver. Pages is the bread. Workers are the filling. You do not need a kitchen running 24/7 just to keep a sandwich on the counter.

1

Top slice: pre-rendered HTML at the edge

Every page on your site is generated at build time, pushed to Cloudflare Pages, and replicated across 330+ cities. A visitor in São Paulo, Lagos, or Seoul receives your HTML from a Point of Presence near them, not from your origin. There is no server-side rendering, no runtime database call, no cold start. The static layer is the bread that makes the whole sandwich cheap and fast.

2

The filling: Workers for the dynamic 1%

Anything that genuinely needs to run code on demand a Stripe redirect, a form submission, an A/B test, a webhook from a payment processor goes into a single-file Worker. Workers run on V8 isolates with no cold starts and scale globally without configuration. You do not need an EC2 instance, a Heroku dyno, or a Lambda warm-up trick. You write a `fetch` handler and bind it to a route.

3

Bottom slice: edge cache as the database for most reads

Most reads in a content business are not really dynamic. They are the same article, the same pricing page, the same case study served to thousands of visitors per day. By treating the edge cache as your read layer and Workers only as the write layer, you collapse the cost of running the business by an order of magnitude. The bread holds the sandwich together.

4

The crust: Cloudflare's DNS, certs, and security envelope

Cloudflare provides DNS, automatic TLS certificate provisioning, DDoS protection, and bot management as part of the same control plane. You do not run Let's Encrypt cron jobs. You do not configure a CDN behind your CDN. The platform handles the boring infrastructure crust so you can focus on the content and the conversion logic.

5

The plate: Git as the only state that matters

Pages integrates with Git so every push to your main branch triggers a global deploy. Combined with a separate domain registrar (we use Dynadot for Pillar's portfolio) and an external repo, your business has exactly three durable artifacts: the source code, the domain ownership, and the Worker code. Lose your Cloudflare account tomorrow and you can rebuild the entire stack in an afternoon from those three things.

02 — The data.

330+

Cloudflare Points of Presence worldwide, putting your static HTML within ~50ms of approximately 95% of internet users

CLOUDFLARE NETWORK STATEMENT, 2025

100+

Countries with at least one Cloudflare edge city, which materially affects Time to First Byte in emerging markets

CLOUDFLARE NETWORK STATEMENT, 2025

Unlimited

Bandwidth included on the Cloudflare Pages free tier no surprise overage bill at scale, unlike most competitors

CLOUDFLARE PAGES PRICING, 2025

500 / month

Builds included on the Pages free tier, which is enough headroom for most teams to ship daily without paying

CLOUDFLARE PAGES PRICING, 2025

10ms CPU

Per-request CPU budget on the Workers free tier sufficient for redirects, form handlers, and webhook receivers

CLOUDFLARE WORKERS PRICING, 2025

14,000+

HTML files Pillar deploys to Cloudflare Pages, served by a single Worker for dynamic payment redirects

PILLAR INTERNAL ARCHITECTURE, 2026

Why the static-but-dynamic split is the right shape in 2026

For most of the last decade, the default architecture for a content-heavy business was a server: a Node app behind a load balancer, a Rails monolith on Heroku, a WordPress install on a managed host. The server rendered HTML, talked to a database, and answered every request. That made sense when content was small and personalized. It stopped making sense as content libraries grew into thousands of pages and as AI search engines started crawling those pages constantly.

Today, the workload is upside down from the old assumption. A site like Pillar serves more than 14,000 HTML files, most of which change at most once a week. A small fraction handle anything that needs to run real code: the buy-now redirect, the contact form, the occasional webhook from a payment processor. If you keep the entire site server-rendered to handle that 1%, you pay for 100% server time, accept cold starts on every region you do not have a warm instance in, and patch a database for the rest of your career.

The static-but-dynamic split flips it: pre-render everything you can, run code only when you must, and let the edge absorb the rest. Cloudflare Pages and Workers are not the only way to express this pattern Vercel's static output plus Edge Functions does it too, and Netlify has its own version but they are the cleanest and cheapest expression of it for a content business that does not need server-side personalization.

Pages handles the static surface and that is more dynamic than it

sounds

Cloudflare Pages is, at its core, a build-and-deploy system for static sites. You point it at a Git repository or push directly with the wrangler CLI, and Cloudflare runs your build command, takes the resulting directory of HTML, CSS, JS, and assets, and replicates it across the global edge network. The deploy is atomic: every URL flips to the new version at once, and old versions are kept around as preview deployments you can roll back to from the dashboard.

Where Pages gets interesting is the configuration that lives alongside the static files. A `_headers` file lets you set custom HTTP headers per path: cache-control rules, security headers, CORS. A `_redirects` file handles 301s and 302s without writing code, which is enough for most legacy-URL migrations. Pages Functions let you drop a JavaScript file into a special directory and have it deployed as a Worker bound to that route. You do not need to run a build for most of these features they are static configuration files that Cloudflare reads at deploy time.

The deploy flow we use on Pillar is straightforward: `wrangler login` once to authenticate, `wrangler pages project create <YOUR_PROJECT>` to register the project, and `wrangler pages deploy ./build` to push the contents of the build directory. From a clean machine to a live global deployment is under five minutes. Custom domains are added in the dashboard, Cloudflare provisions the TLS certificate automatically through Universal SSL, and DNS records point at the project hostname.

Workers handle the dynamic 1% and one Worker is often enough

A Worker is a single JavaScript file that exports a `fetch` handler. When a request hits a route bound to that Worker, Cloudflare spins up a V8 isolate (not a container, not a VM), runs your handler, and returns the response. Isolates start in single-digit milliseconds, so there is no cold-start tax of the kind you see on AWS Lambda when a function has been idle. You write code as if it were a tiny HTTP server and Cloudflare runs it everywhere at once.

The mental shift that matters is that you do not need a Worker per feature. On Pillar we run a single Worker whose job is to receive a request to a buy URL, look up the corresponding Stripe Payment Link, and 302-redirect the visitor to the checkout. Twenty lines of code, one route binding, no maintenance. The same pattern handles form submissions (POST to a Worker, validate, forward to Formspree or your email provider), A/B tests (read a cookie, rewrite the response, set the cookie), and webhook receivers (POST from Stripe or another provider, verify signature, write to a queue).

When you do need more, Cloudflare gives you primitives that compose with Workers: KV for low-write key-value storage, D1 for SQLite at the edge, R2 for object storage with no egress fees, Queues for async work. You can build a real backend with these, but the discipline of the static-but-dynamic sandwich is to avoid building one until you actually need it. Every primitive you adopt is a new thing to monitor.

Why this beats Vercel, Netlify, and AWS for a content business

The honest comparison: all three competitors are fine platforms. They differ on price, defaults, and where the surprises hide. Vercel is excellent if you are running Next.js with server-side rendering, but for a static site at scale you start paying $20+/mo on the Pro tier and bandwidth costs become real once you cross a few terabytes. Netlify is similar in price and has historically charged for build minutes, which means a chatty CI pipeline can quietly become expensive. AWS S3 plus CloudFront is the cheapest at the raw infrastructure layer but requires you to assemble the pieces yourself: bucket policies, origin access identity, CloudFront behaviors, ACM certificates, Route 53. You can do it well, but it is half a week of setup and a permanent surface area you maintain.

Cloudflare Pages plus Workers wins on three axes for a content business. First, unlimited bandwidth on the free tier means your cost does not scale with viral spikes, which removes a category of anxiety. Second, the unified control plane DNS, certs, edge, Workers, R2, D1 means you configure one platform instead of stitching together five. Third, the developer experience of `wrangler` is consistently better than the equivalent AWS CLI workflow and roughly on par with Vercel's CLI, with the added advantage that you are not betting on a single vendor's framework opinions.

The architecture is also easy to leave. Because your static output is just HTML and your dynamic logic is a single Worker per concern, migrating to another edge platform is a matter of changing the deploy command and rewriting the Worker against a different runtime. There is no lock-in to a proprietary framework, no database export ritual, no infrastructure-as-code drift. If you outgrow Cloudflare in five years, you walk away with your repo and your domain intact.

03 — Watch: a real walkthrough

04 — Setup checklist: deploy a static-but-dynamic stack in under an hour

If you already have a static site, this is roughly the steps to move it to Cloudflare Pages and add one Worker for the dynamic surface. If you are starting from scratch, the same flow works the only addition is producing the static output first.

1. Install the Cloudflare CLI: `npm install -g wrangler`. Authenticate with `wrangler login`, which opens a browser to OAuth your Cloudflare account.
2. Create a Pages project: `wrangler pages project create <YOUR_PROJECT>`. Pick a project name that matches the eventual hostname for clarity, but it is internal-only.
3. Deploy your build directory: `wrangler pages deploy ./build --project-name <YOUR_PROJECT>`. You get a preview URL immediately at `<your-project>.pages.dev`; visit it to confirm everything renders.
4. Add your custom domain in the Cloudflare dashboard under Pages → Custom domains. Cloudflare provisions a TLS certificate automatically; if your DNS is on Cloudflare it points the record for you, otherwise you add a CNAME at your registrar.
5. Configure caching and headers with a `_headers` file at the root of your build directory. Start with sensible defaults: long cache on hashed assets, short cache on HTML, security headers across the board.
6. Write your first Worker: a single `index.js` exporting a `fetch` handler. Use it for the most obvious dynamic concern (payment redirect, form handler, or webhook receiver) and bind it to a route via the dashboard or `wrangler.toml`.
7. Wire Git to Pages so every push to main triggers a deploy. From here, the static-but-dynamic stack runs itself the only things you maintain are your source code, your domain, and your one or two Worker files.

Frequently asked questions.

Do I lose SEO by going fully static instead of server-side rendered?

No, and arguably the opposite. Static HTML is the cleanest signal you can send to crawlers: there is no JavaScript-rendered content to misinterpret, no hydration mismatch, no slow Time to First Byte from a runtime server. Both classical search engines and AI overview systems prefer fast, well-structured static HTML. The only SEO case for SSR is personalized content that varies per user, which is not what content-heavy authority sites publish. See our piece on what AI overviews reward (/learn/en/web-foundations/what-ai-overviews-reward/) for the deeper argument.

What if I need a database? Does Cloudflare cover that?

Yes, with three options. Cloudflare KV is a global key-value store, ideal for configuration and low-write data. D1 is SQLite at the edge, ideal for app-shaped relational data. R2 is S3-compatible object storage with no egress fees. All three are bound directly to Workers, so you do not maintain a connection pool or a separate database service. The discipline is to stay static-first and reach for these primitives only when the use case truly requires writes.

How does Cloudflare compare to Vercel if I am already running Next.js?

Vercel is the path of least resistance for Next.js SSR and SSG together, especially if you use Next.js features like Image Optimization or middleware heavily. Cloudflare Pages also supports Next.js but is most compelling when your output is fully static. If your Next.js app is mostly static export with a few API routes, Cloudflare Pages plus Workers for those routes is typically cheaper and faster at scale. If your app relies on Vercel-specific runtime features, the migration is more involved.

Is one Worker really enough for a real business?

Often yes. On Pillar, one Worker handles the entire dynamic surface a redirect from buy URLs to Stripe Payment Links. Forms route to a third-party endpoint (we use Formspree for the public site). Analytics route to Cloudflare's built-in analytics and a separate provider. The instinct to spin up a Worker per feature comes from a microservices reflex that does not apply at this scale. Start with one Worker and split only when a real boundary forces it.

What is the failure mode? What happens if Cloudflare has an outage?

Cloudflare's edge has had a small number of notable outages over the years, and during those windows your site and any Workers go down. The mitigation is architectural: keep your source code in a separate Git host (GitHub, GitLab), keep your domain at an independent registrar (we use Dynadot), and document the redeploy path. Because the static output is just HTML, you can push it to any other static host in minutes during an emergency. The static-but-dynamic stack is intentionally easy to leave, which is itself a form of resilience.
