LEARN · THE STACK

The Wrangler Deploy Loop: Near-Live Edge Deploys

Deploy latency is a hidden tax on content velocity. When the cycle from save to live runs 30 seconds instead of 5 minutes, iteration stops feeling like deployment and starts feeling like editing. This piece walks through the loop, the tooling, and the architecture that makes near-live edge deploys the default at Pillar.

---

10 min read

Last updated: June 10, 2026

> *When deploys take 5 minutes, content velocity drops because operators batch changes to avoid the wait. When deploys take 30 seconds, iteration becomes continuous and revision cost approaches zero. Wrangler, Cloudflare's edge deploy CLI, delivers the latter shape, and once that shape is locked in, everything downstream, from copy testing to design polish, gets faster.*

Why deploy latency is the silent killer of iteration

- Why a 30-second deploy loop is not just faster, it changes which work feels reasonable to do
- How `wrangler pages deploy` compares to GitHub Actions, AWS CodePipeline, and traditional PaaS deploys
- The mechanics of edge propagation and why Cloudflare's anycast network makes global rollout feel instant
- A concrete deploy script you can adapt, including preview branches, cache-busting verification, and staging directories
- How the Wrangler loop fits inside a broader zero-secret, owned-asset content stack

01 — The 30-Second Iteration Loop: a mental

model for content velocity

The 30-Second Iteration Loop

Every content system has the same four-step loop: edit, deploy, verify, revise. The total time of that loop is the real bottleneck on quality, because below a certain threshold operators stop batching changes and start treating deploys like saves. The 30-Second Iteration Loop names the four phases and the latency budget each one earns when the stack is tuned correctly. The math is unforgiving. At 30 seconds per loop, ten iterations cost five minutes. At five minutes per loop, ten iterations cost fifty minutes. The same work, different clock time, vastly different willingness to refine.

1

Edit (under 5 seconds to commit)

The first phase is purely local. A markdown change, a CSS tweak, a copy revision, a new HTML page generated from a template. The constraint here is keystroke-to-saved-file latency and local build time if any. Static-site stacks win this phase because there is no compile step heavier than a templating pass, and most edits ship as plain file overwrites.

2

Deploy (under 30 seconds to live)

This is the phase Wrangler collapses. `wrangler pages deploy` uploads only changed files via a content-addressed manifest, parallelizes the transfer, and triggers atomic activation at the edge. For sites under 10,000 files the typical deploy completes in 15 to 45 seconds. At Pillar, with 14,000+ HTML files in the index, real-world deploys land in 25 to 50 seconds.

3

Propagate (30 to 90 seconds to global edge)

Cloudflare's anycast network propagates the new deploy to its global PoP fleet within roughly 30 to 90 seconds after the deploy completes. Because activation is atomic, there is no half-deployed state, you are either on the old version or the new one. Cache invalidation is automatic for the deployed paths, which removes a step that traditionally adds minutes.

4

Verify (under 10 seconds with cache-busting)

Verification happens by hitting the URL with a cache-buster, typically `?cb=$(date +%s)` in a curl command or a hard-reload in the browser. The cache buster ensures you are looking at the freshly deployed version and not a stale browser or intermediary cache. When this is automated as the final step of the deploy script, the loop closes cleanly and feedback arrives immediately.

5

Revise (back to Edit)

The revise phase is the human one. It is where the operator decides what to change next based on what they just saw live. The faster the previous four phases, the more revisions a human will choose to make in a working session. Operators visibly try more variations when the loop is fast, which is the entire compounding mechanism.

02 — The data.

25–50s

Typical Pillar deploy time across 14,000+ HTML files using `wrangler pages deploy`

PILLAR INTERNAL DEPLOY LOGS, 2026

15–45s

Standard `wrangler pages deploy` latency for sites under 10,000 files

CLOUDFLARE PAGES DOCUMENTATION, 2024

30–90s

Cloudflare anycast edge propagation window after deploy completion

CLOUDFLARE NETWORK OPERATIONS, 2024

10x

Clock-time difference between a 30-second loop and a 5-minute loop across ten iterations

30-SECOND ITERATION LOOP MATH, THIS PIECE

2–10 min	60–180s
Typical AWS CodePipeline deploy time for comparable static workloads	GitHub Actions build + deploy time for typical Pages-style workflows
AWS CODEPIPELINE BENCHMARKS, 2024	GITHUB ACTIONS RUNNER TELEMETRY, 2024

What Wrangler actually does that makes it fast

Wrangler is the official Cloudflare CLI for managing Workers and Pages deployments. The speed advantage is not a single optimization, it is the absence of several slow steps that other deploy systems still perform. There is no container image to build, no virtualized runtime to spin up, no centralized region to route through, and no separate cache purge step. Files are uploaded in parallel as a content-addressed manifest, meaning only the bytes that have actually changed since the previous deploy traverse the wire. For an incremental copy edit on a 14,000-file site, the actual upload is often a single small batch even though the project is large.

The activation step is atomic. Cloudflare assigns the new deployment a unique URL, then swaps the production alias to point at it. There is no half-deployed window where some visitors see one version and others see another, and there is no manual cutover. If something is wrong, rollback is also atomic via the dashboard or `wrangler pages deployment` commands. This atomicity is part of why the loop can be tight without being risky.

Compare this to a traditional PaaS like Heroku or a buildpack-style platform: the deploy must build a slug, push it to a registry, spin up new dynos, run health checks, and only then swap traffic. Each step adds latency, and the total budget is two to five minutes even for small apps. AWS CodePipeline can be even slower because it chains discrete stages. GitHub Actions plus a separate edge target is faster but still bottlenecked by the runner cold start and the queue. Wrangler skips all of this by treating deploys as content uploads rather than application releases.

The architecture that lets Pillar deploy 14,000+ files in under a minute

Pillar's content stack is pre-rendered static HTML. There is no runtime templating, no server-side rendering, no database round trip on request. Every page, every learn library piece, every multilingual variant exists as a flat HTML file in an output directory before deploy time. This is the single most important architectural choice for fast deploys: the cost of an additional page is one file write, not one extra request to a CMS or database.

The build step that produces those files runs locally or in CI, separate from the deploy. By the time `wrangler pages deploy` runs, the output directory is already complete and stable. Wrangler then compares the local file manifest against the previously deployed manifest and uploads only what changed. On a typical copy edit touching one or two pages, the actual transfer is kilobytes, not megabytes. This is why deploy time scales with the size of the change, not the size of the site.

A small but useful pattern: for content-heavy sites, pre-rsync the build output to a clean staging directory before invoking Wrangler. This gives you a controlled snapshot of exactly what will ship, separates the build artifacts from the deploy artifacts, and makes it trivial to inspect or diff the deploy contents before they go live. It also keeps the Wrangler invocation pointed at a predictable path that does not move with build tooling changes.

Preview branches, verification, and the closed-loop deploy script

Cloudflare Pages supports preview deployments out of the box. Passing `--branch=feature-name` to `wrangler pages deploy` creates a unique subdomain for that branch, independent of production. This is the right place to review a non-trivial change before promoting it, and it removes the temptation to ship straight to main just to see the change live. For trivial copy or design edits, deploying directly to main with `--branch=main` is the fast path. For anything that needs review, branch previews give you a real URL on a real edge without polluting production analytics or search indexing.

Verification is the easy-to-skip step that should not be skipped. The simplest pattern is a curl with a cache-buster: `curl -s https://<YOUR_DOMAIN>/path?cb=$(date +%s) | grep 'expected string'`. The cache buster forces the edge to serve the fresh version even if an aggressive intermediate cache is in play. For browser verification, a hard reload (Cmd+Shift+R or Ctrl+Shift+R) does the equivalent. Building this verification into the deploy script itself means the loop closes automatically and you do not have to remember to check.

A complete deploy script looks something like this: build, rsync to a clean staging directory, run `wrangler pages deploy ./staging --project-name=<YOUR_PROJECT> --branch=main`, then curl the canonical URL with a cache buster and exit nonzero if the expected content is not present. That single script is the entire deploy loop, and it runs in well under a minute on a typical change. For more on keeping deploy credentials out of source while still automating this, see Zero-Secret Architecture (/learn/en/the-stack/zero-secret-architecture/).

Why the loop compounds: the human side of fast deploys

Engineering teams sometimes treat deploy latency as a purely technical concern, but the real win is behavioral. When the loop is slow, operators batch changes. They make ten edits, deploy once, and discover that two of those edits broke something or missed the mark. They then have to remember which change caused which problem and either deploy a fix or roll everything back. The cost of being wrong is high, so people are conservative about what they try.

When the loop is fast, operators stop batching. They make one change, deploy, look at it live, and decide on the next change. The cost of being wrong is one undo and one more deploy, both measured in seconds. People try more variations, push more polish, and converge on better results because the experiment cost is negligible. This is the compounding effect, and it shows up most clearly in content quality over weeks, not in deploy logs.

There is a secondary effect for technical founders and small teams: a fast loop makes it reasonable to ship at unusual hours, in short bursts, between other tasks. A five-minute deploy is a context switch. A thirty-second deploy is a save. This shifts content work from scheduled blocks to opportunistic touches, which over months adds up to dramatically more total iteration.

03 — Watch: a real walkthrough

04 — Wire up your own 30-second deploy loop

These steps assume a static or pre-rendered output directory. If your stack still runs server-side rendering or hits a database on request, the loop will not collapse to thirty seconds no matter what deploy tool you use. The architecture has to be static-first.

1. Install Wrangler globally with `npm install -g wrangler`, or invoke it ad-hoc with `npx wrangler` if you prefer not to install globally
2. Authenticate once with `wrangler login`, which opens a browser flow and stores a long-lived token in your user profile so subsequent commands do not prompt
3. Create a project-level config: either a `wrangler.toml` at the repo root or a Pages project defined in the Cloudflare dashboard, with a stable project name you will reference in deploy commands
4. Write a deploy script that runs your build, rsyncs the output to a clean staging directory, then runs `wrangler pages deploy ./staging --project-name=<YOUR_PROJECT> --branch=main`
5. Add a verification step to the script: curl the canonical URL with `?cb=$(date +%s)` and grep for a known string from the deploy, exit nonzero if not present
6. Use `--branch=<feature-name>` for any change that needs review, which spins up a preview subdomain without touching production
7. Keep credentials out of source by relying on the long-lived token from `wrangler login` on dev machines, and project-scoped tokens stored as CI secrets for automated deploys

Frequently asked questions.

Why not just use Vercel or Netlify, which also have fast deploys?

Vercel and Netlify are excellent platforms and their deploy loops are also fast, often in the same 30 to 60 second range. The reason to choose Wrangler and Cloudflare Pages is not raw speed, it is the broader Cloudflare surface: Workers for any dynamic logic, R2 for object storage, anycast DNS, and a generous free tier that scales linearly. If you are already on Vercel or Netlify and happy, the iteration loop lesson still applies and the deploy tooling matters less than the static-first architecture underneath it.

What happens if a deploy fails partway through?

Wrangler deploys are atomic at activation time. If the upload fails, the production alias is not updated and the previous version stays live. If the upload completes but verification fails afterward, you can roll back to any prior deployment with one command or via the dashboard, and the rollback itself is atomic and fast. There is no broken-half-deployed state that requires manual cleanup.

How do I handle environment-specific config without leaking secrets?

For static sites, prefer compile-time substitution: read environment variables during the build step and write the resolved values into the output HTML or a separate config file, then deploy the result. For Workers that need runtime secrets, use `wrangler secret put` to store encrypted values bound to the worker. Never commit secrets to source. For the full pattern, see Zero-Secret Architecture (/learn/en/the-stack/zero-secret-architecture/).

Does the 30-second loop matter if I only deploy once a week?

Probably not, in the moment. But the deeper effect is that a fast loop changes what feels reasonable. Teams with five-minute deploys plan their content work in scheduled batches. Teams with thirty-second deploys treat content edits like file saves and ship continuously. Over months, the continuous shippers iterate more, refine more, and end up with measurably better content even if individual deploys were not in a hurry. The behavioral shift is the value, not the per-deploy seconds saved.

Can I use Wrangler with my existing GitHub Actions workflow?

Yes. Wrangler runs cleanly in CI: install it with `npm install -g wrangler` in the workflow, store a project-scoped Cloudflare API token as a GitHub Actions secret, and invoke `wrangler pages deploy` as the final step. This gives you automated production deploys on merge to main and preview deploys on pull requests, while still letting you deploy from your laptop for tighter loops when iterating live.
