LEARN · THE STACK

Multilingual SEO at Static-Generation Scale

Most teams hit a wall around 500 pages in three languages: translations drift, hreflang breaks, schema rots, build times balloon. Pillar runs 14,000+ HTML files across four languages on a one-minute build-and-deploy loop. The discipline is in the data layer and the templates, not in any clever runtime.

---

10 min read

Last updated: June 10, 2026

> *Shipping 14,000+ HTML files in four languages with full hreflang quads, JSON-LD schemas, and mobile-first design at zero runtime cost requires discipline at three layers: data, templates, and generation. Get those right, and the volume becomes a feature, not a liability.*

The thesis

- Why a static site at 14,000+ pages outperforms a CMS-backed site on the metrics search engines actually weigh
- How to structure inventory data so adding a fifth language is a config change, not a rewrite
- How to enforce hreflang correctness in the template instead of policing it in audits
- How to generate JSON-LD schema programmatically from the same data that renders the page body
- How to orchestrate one-script builds that complete in under 30 seconds at this scale

01 — The Static-Multilingual Discipline

The Static-Multilingual Discipline

The Static-Multilingual Discipline is the operating principle behind Pillar's 14,000-page footprint. It says: every page on every language variant is a deterministic function of inventory data plus a template plus a translation dictionary. There is no runtime, no database query, no dynamic rendering. The discipline holds across four pillars, and breaking any one of them re-introduces the failure modes a static architecture exists to eliminate.

1

Inventory as source of truth

A single CSV (for tabular data like the domain portfolio) or JSON (for structured content like Learn pieces) holds the canonical inventory. Every page generator reads from it. When a domain is added or a Learn piece is published, the inventory file is the only place that changes. The generators do the rest.

2

Templates that enforce correctness

Page templates are not just layout; they are the enforcement layer. Hreflang quads, canonical tags, JSON-LD schema, Open Graph metadata, and mobile-first viewport declarations are all rendered by the template, not by the author. If a field is missing, the template either fails the build or fills a sane default. Authors cannot ship a page with broken hreflang because they never write the hreflang.

3

Translations as a keyed dictionary

Every translatable string lives in a JSON dictionary keyed by a stable identifier. Templates reference strings by key, never by literal value. Adding a new language is a matter of producing a new dictionary file. The English template and the Portuguese template are the same template, fed different dictionaries.

4

One-script orchestration

A single build script invokes every page generator in dependency order, then writes the sitemap index, the per-language sitemaps, and llms.txt. There is no manual step. Full regeneration of 14,000+ files takes roughly 30 seconds; deploy via `wrangler pages deploy` takes another 30. From inventory change to globally-live is under a minute.

5

Static delivery, zero runtime

The output is plain HTML, served from a CDN edge. There is no server to scale, no database to back up, no application framework to patch. The runtime cost of every request is a file lookup. This is what makes the architecture sustainable at five-figure page counts and four-figure daily edits.

02 — The data.

14,000+

Total HTML files generated

PILLAR BUILD INVENTORY, 2026

4

Launch languages: English, Spanish, French, Portuguese (Brazilian)

PILLAR INTERNATIONALIZATION PLAN

6,608

Per-domain landing pages across the portfolio

PILLAR DOMAIN INVENTORY, 2026

5

Hreflang entries per page (4 language alternates + 1 x-default)

PILLAR TEMPLATE SPECIFICATION

~30 sec	~1 min
Full-site regeneration time, from inventory to dist directory	End-to-end: regenerate plus `wrangler pages deploy` to live global edge
PILLAR BUILD TELEMETRY, 2026	PILLAR DEPLOY TELEMETRY, 2026

Why static beats dynamic at this scale

Conventional wisdom says a 14,000-page site needs a CMS. That advice is correct for sites where content changes hour by hour and editors need WYSIWYG tooling. It is exactly wrong for sites where content is structured, the schema is stable, and the bottleneck is search engine indexation rather than authoring throughput. Pillar's content is structured: a domain, a Learn piece, a category landing, a hub page. Each type has a defined data shape. There is no reason to pay the latency and complexity tax of a database lookup on every request when the same HTML will be served to every visitor.

Static delivery also collapses the security surface. There is no admin login to harden, no SQL injection vector, no plugin ecosystem to patch. The build pipeline runs on a developer machine or a CI runner; the output is plain files. This is the foundation we describe in the Zero-Secret Architecture (/learn/en/the-stack/zero-secret-architecture/) piece, and it is the reason a one-person ops loop is viable at this page count.

Finally, static wins on Core Web Vitals by default. Largest Contentful Paint is bounded by file size and edge latency, not by application server response time. Cumulative Layout Shift is determined by the rendered HTML, which is deterministic. First Input Delay is near zero because there is no hydration step. These are the metrics search engines actually weigh, and a static site starts the race ten meters ahead.

The data layer: CSV for tabular, JSON for structured

Pillar uses two inventory formats and the choice is deliberate. The domain portfolio is tabular: every row has the same columns (domain, category, price, status, language availability). A CSV is the right tool. It opens in any spreadsheet, diffs cleanly in version control, and parses in three lines of Python. The build script reads the CSV, iterates rows, and emits one HTML file per domain per language. At 6,608 domains across four languages, that single CSV drives over 26,000 generated files if every domain has every language; in practice the count is lower because not every domain ships in every language at launch.

Structured content, Learn pieces and product pages, lives in JSON. A Learn piece has a title, a subhead, a thesis, a framework with named pillars, a tactical checklist, an FAQ. That is not a flat row; it is a nested object. JSON expresses the nesting, validates with a schema, and parses directly into the dict the template expects. The 88 Learn library pages are each one JSON file plus a shared template. Edit the JSON, rerun the generator, the page reflects the change.

What does not belong in either format is a database. A SQL or NoSQL store for a static site is pure overhead: it adds a query latency, a backup obligation, a schema migration burden, and a credential to manage. None of those costs buy you anything the file system does not already provide. The inventory lives in the repo, alongside the templates, under version control. The diff for a content change is a diff to a JSON or CSV file. That is the entire change-management story.

The template layer: enforce hreflang and schema, do not police them

Hreflang correctness is where most multilingual sites fail. The spec requires that every language variant of a page link to every other variant, including itself, plus an x-default. For Pillar's four languages, that is five `<link rel="alternate">` tags on every single page. Across 14,000+ files, that is over 70,000 hreflang declarations. Hand-writing them, or relying on an editor to remember them, is not a strategy. The template renders them from the inventory's URL slug plus the language list. Authors cannot omit hreflang because they never touch hreflang.

The same discipline applies to JSON-LD schema. Every page type has a defined schema set: Organization plus WebPage plus BreadcrumbList for most pages; FAQPage added on Learn pieces with FAQs; ItemList on category landings. The template builds the schema object programmatically from the data dict and emits it as a single `<script type="application/ld+json">` block. The author writes prose; the template emits machine-readable structured data. Neither side of that division is asked to do the other side's job.

Canonicals, Open Graph tags, Twitter cards, mobile viewport, and font preloads round out the head block. All of it is template-rendered. The author's surface area is the JSON or CSV row. That separation is what makes the system scale: adding a 14,001st page is exactly as hard as adding the first.

The generation layer: one script per page type, one orchestrator

Each page type has a dedicated generator: one for domain pages, one for Learn pieces, one for product trees (Studio, Authority, Institute and their subpages), one for category landings, one for the hub pages. Each generator follows the same contract: read inventory, iterate items, render template, write file. A generator is rarely more than 200 lines of Python. Because the contract is uniform, adding a new page type is mechanical: copy the closest existing generator, swap in the new template, point at the new inventory section.

A single orchestration script runs the generators in dependency order: product pages first because hub pages link to them, hub pages next, then Learn pieces, then domain pages, then the sitemap index and the four per-language sitemaps, then llms.txt last because it references everything. The orchestrator is the only entry point anyone needs to know. Run it, and the entire dist directory is regenerated in roughly 30 seconds. Deploy with `wrangler pages deploy` and the new content is on Cloudflare's edge in about another 30. That one-minute loop is what makes daily content updates frictionless.

Build determinism matters as much as build speed. The same inventory plus the same templates produces byte-identical output. That means a CI check can regenerate the site and diff against the committed dist directory to catch drift. It also means rolling back is a git revert away. There is no database state to reconcile, no cache to invalidate, no eventual-consistency window to wait through. The repo is the system of record.

03 — Watch: a real walkthrough

04 — Tactical checklist: shipping a multilingual

static site

If you are setting up an architecture like this from scratch, this is the order of operations that tends to work. Doing these in sequence saves rework. Doing them out of order tends to require a refactor about three months in.

1. Decide your inventory format per content type before you write a single template. CSV for tabular, JSON for structured, nothing in a database for a static site.
2. Pick your launch languages and create a translation dictionary file per language with the same key set. An empty value is fine at this stage; an inconsistent key set is not.
3. Write the head-block template first. Canonical, hreflang quad plus x-default, Open Graph, Twitter card, mobile viewport, JSON-LD schema. Make it impossible to render a page without these.
4. Write one generator per page type. Keep generators short, side-effect-free, and uniform in signature: `generate(item, language, output_dir)`.
5. Build the orchestrator as a single script with explicit dependency order. Resist the urge to make it clever. A linear sequence of generator calls is easier to debug than a DAG.
6. Add a sitemap index plus per-language sitemaps as the final step of the build. Add llms.txt after sitemaps because it can reference them.
7. Deploy via `wrangler pages deploy` or your CDN's equivalent. Measure full-build-to-live time. If it exceeds two minutes, you have a bottleneck worth fixing before the page count grows.

Frequently asked questions.

Why not use a headless CMS with a static export step?

Headless CMS plus static export is a defensible choice if your bottleneck is non-technical editors who need a WYSIWYG. Pillar's bottleneck is not authoring, it is correctness at scale: hreflang quads, schema, canonical, sitemap entries. A CMS adds an editorial UI, a hosted service, an account to manage, and a webhook to debug. It does not improve hreflang correctness. For structured content with a stable schema, an inventory file in the repo gives you everything a headless CMS gives you, minus the operational surface.

How do you keep translations from drifting across four languages?

Two mechanisms. First, every translatable string is keyed in the per-language JSON dictionary, so adding a string in English without a corresponding entry in Spanish, French, or Portuguese is detectable by a simple key-set diff. Second, the build fails if a referenced key is missing in the active language. That means a translation gap surfaces at build time, not in production. You can still ship a stub translation deliberately, but you cannot ship one accidentally.

What does the build actually do for 14,000 files in 30 seconds?

Read the inventory CSV and JSON files into memory once. Iterate each page type in dependency order: for each item, format the template with the data dict, write the resulting HTML to the dist directory. Python's string formatting plus file I/O against an SSD comfortably hits this throughput at this scale. The orchestrator is single-process and single-threaded because the workload is I/O bound, not CPU bound. Parallelizing tends to save less than it costs in complexity until you are well past 100,000 pages.

Why Cloudflare Pages specifically, instead of Vercel or Netlify or AWS?

Cloudflare Pages offers free static hosting with a global edge, generous bandwidth allowances, and a CLI deploy (`wrangler pages deploy`) that matches the one-minute loop the build is optimized for. Vercel, Netlify, AWS S3 plus CloudFront, and similar services would all work; the architecture is portable. The cost story is what tips the choice for a bootstrapped operation: a 14,000-file site on Pages costs effectively nothing at typical traffic, which preserves capital for the parts of the business that actually need it.

What is the migration path if we already have a database-backed multilingual site?

Export every record to CSV or JSON, grouped by content type. Write one generator per content type that reads your export and emits HTML. Run the generators against a staging directory and diff the output against a crawl of your current site, paying particular attention to hreflang, canonical tags, and JSON-LD schema. Cut DNS over once the diff is acceptable. The migration is usually less work than teams expect because the database schema you already have maps cleanly to the inventory file the static build wants. The hardest part tends to be retiring the admin UI people are used to, not the technical conversion.