FROM WEBFLOW TO ASTRO IN 6 DAYS

— WITH CLAUDE CODE

A practical migration playbook: implementation prompts, architecture decisions, tests and lessons from rebuilding the Localyzer website.

DAY 1 · first crawl | DAY 2 | DAY 3 | DAY 4 | DAY 5 | DAY 6 · go-live

6
calendar days from first crawl to go-live

2–3
estimated days of net hands-on work

103
live URLs across English and German

Based on our November 2026 migration of **www.localyzer.io**.

CONTENTS

The story behind the migration 02
What we migrated 03
Who this playbook is for 04

1 Phase 1: Preparation 05
2 Phase 2: Migration 12
3 Phase 3: Deployment 24

After the migration: running the site with AI 28
Lessons learned 31
Final perspective 32
Technical reference documentation 33

THE STORY BEHIND THE MIGRATION

Our biggest problem with Webflow wasn't the technology. It was the time it took to make changes.

A new page, a layout adjustment, a small bug fix. Almost every change went through our agency.

Briefing. Coordination. Implementation. Feedback. Another iteration.

We work with an excellent agency, RNGS, and continue to do so. Their expertise in branding, creative design and brand strategy remains essential to us.

But even with a great partner, this workflow takes time. And our team never really became comfortable working directly in Webflow.

AI coding gave us a different option.

Rather than introducing AI into the existing website builder, we decided to move to a code-first setup where Claude Code could work directly with our source code.

We migrated the entire website from Webflow to Astro in six calendar days, with approximately two to three days of hands-on work. One person led the project, combining AI-assisted implementation with hands-on architecture decisions, testing and deployment.

The real benefit came afterwards.

Today, we can implement a new page, modify an animation or fix a bug directly. Test it, review it and deploy it. Without starting an external project or waiting for another round of coordination.

The biggest gain wasn't lower costs. It was a shorter path from idea to implementation.

WHAT WE MIGRATED

103
live URLs, EN + DE

~60
blog posts

32
redirects

3
forms

4
header themes

6 days
2–3 days net hands-on

AREA	SCOPE
URLs and languages	103 live URLs across English and German, including translated slugs
Content	Blog (about 60 posts across both languages), jobs, legal pages and six smaller CMS-backed lists
Design and interaction	Scroll animations, sliders, Lottie, marquees, four header themes and a demo modal
Integrations	Three forms, Google Tag Manager, cookie consent (Consent Mode v2) and 32 redirects
Quality target	Preserve URLs, content and SEO metadata; match desktop and mobile layouts
Delivery	Six calendar days; estimated 2–3 days net hands-on time; one person using Claude Code

WHO THIS PLAYBOOK IS FOR

This is for founders, developers and technical marketing teams who can review code, work with Git, evaluate deployment decisions and take responsibility for operating a website. You do **not** need to be an Astro specialist. But you do need enough technical judgment to recognize when an AI-generated solution is incomplete or wrong.

Webflow remains a good choice for teams that prefer visual editing, and it offers AI features and integrations with external agents. For companies with technical expertise, a code-first approach can offer greater flexibility, portability and control over **code, content and deployment**.

What you take away: an inventory-and-mirror workflow, an example `CLAUDE.md`, migration prompts, content-export strategy, animation replacements, serverless forms architecture, automated verification and a deployment/rollback checklist.

THE THREE PHASES

1 PREPARATION:

capture the live site as a frozen reference and write the rules the AI will work by.

2 MIGRATION:

set up the project and rebuild the site page by page, then replace Webflow's built-in services (forms, redirects, consent).

3 DEPLOYMENT:

choose a host, test on staging, switch DNS, keep a rollback ready.

PHASE 1 OF 3

PHASE 1: PREPARATION

Capture the live site, export its content and document the implementation rules before rebuilding pages. The mirror and CMS export become the source of truth for AI-generated code and for later comparisons.

STEPS IN THIS PHASE

1.1 Get the list of every URL

1.2 Mirror the whole site

1.3 Export what isn't in the HTML

1.4 Write the rules for the AI

1.5 Decide the target URL scheme and stack

STEP 1.1: GET THE LIST OF EVERY URL

The live sitemap is a useful starting inventory, but not a complete source of truth for every live URL.

A single `curl | grep` is insufficient when the sitemap contains nested indexes, XML entities or compressed child sitemaps. This Python 3 script resolves those cases and collects the page URLs:

PHASE 1 · STEP 1.1

PYTHON

```python
# scripts/sitemap_urls.py
import gzip
import sys
import urllib.request
import xml.etree.ElementTree as ET
from urllib.parse import urlparse

seen = set()
urls = set()

def collect(url):
    if url in seen:
        return
    seen.add(url)
    with urllib.request.urlopen(url, timeout=30) as response:
        body = response.read()
        encoding = response.headers.get("Content-Encoding", "")
    if urlparse(url).path.endswith(".gz") or encoding == "gzip":
        body = gzip.decompress(body)
    try:
        root = ET.fromstring(body)
    except ET.ParseError as exc:
        raise ValueError(f"Invalid sitemap XML at {url}: {exc}") from exc
    kind = root.tag.rsplit("}", 1)[-1]
    locations = [node.text.strip() for node in root.iter()
                 if node.tag.rsplit("}", 1)[-1] == "loc" and node.text]
    if kind == "sitemapindex":
        for child in locations:
            collect(child)
    elif kind == "urlset":
        urls.update(locations)
    else:
        raise ValueError(f"Unexpected sitemap format at {url}: {kind}")

if __name__ == "__main__":
    if len(sys.argv) != 2:
        sys.exit("Usage: python3 scripts/sitemap_urls.py https://www.example.com/sitemap.xml")
    try:
        collect(sys.argv[1])
    except (ValueError, OSError) as exc:
        sys.exit(f"Sitemap retrieval failed: {exc}. Supply an XML sitemap URL, not the homepage.")
    for url in sorted(urls):
        print(url)
```

Run from the repository root:

BASH

```bash
python3 scripts/sitemap_urls.py https://www.example.com/sitemap.xml > urls.txt
```

PHASE 1 · STEP 1.1

A sitemap only contains URLs that were included when it was generated. Add missing campaign pages, thank-you pages and other unlisted URLs to `urls.txt`; cross-check against CMS exports, redirects and analytics.

This file becomes the **baseline acceptance list**: every existing live URL must still work. Account separately for deliberately added pages (such as new thank-you or error pages), and record any approved URL changes. Check the build against that manifest throughout the migration.

STEP 1.2: MIRROR THE WHOLE SITE

BASH

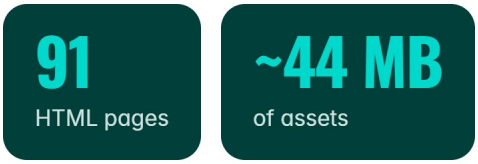
```
wget -i urls.txt --page-requisites --adjust-extension --convert-links \
  --span-hosts --domains=www.example.com,cdn.prod.website-files.com \
  -e robots=off -P mirror/
```

FLAG	WHY
`-i urls.txt`	Crawl exactly the URLs in the list instead of following links
`--page-requisites`	Also download the CSS, JS, images and fonts each page needs
`--adjust-extension`	Save `/about` as `about.html`
`--convert-links`	Rewrite links to local paths so the mirror can be browsed offline
`--span-hosts --domains=…`	Webflow serves all assets from `cdn.prod.website-files.com`; allow that host but no others
`-e robots=off`	Ignore `robots.txt` for the crawl

PHASE 1 · STEP 1.2

Only crawl sites you control or are authorized to copy. Disabling robots checks is appropriate for your own migration reference, not for third-party sites. The example assumes the Webflow asset CDN hostname shown; inspect the live site's network requests and extend the allowed domains for your own site.

The mirror is the AI's reference for **exact text, HTML attributes, SEO tags, image sizes, Webflow's generated CSS** (where the design tokens come from) and inline scripts (tracking, consent, form actions). For us it was 91 HTML pages and about 44 MB of assets.



Known gaps of a wget mirror, and how we handled them:

- **The mirrored CSS doesn't load.** Webflow's `<link>` tags carry `integrity` (SRI) hashes; wget rewrites URLs inside the CSS, so the hashes no longer match and the browser blocks the stylesheet. To view the mirror, serve a *copy* with the `integrity` attributes stripped.
- **Assets loaded by JavaScript are missing.** Lottie files and anything else fetched at runtime aren't seen by wget. Download them individually.
- **Custom scripts hosted elsewhere.** Our animations came from custom JS files hosted on a separate domain (referenced from the page). Find every `<script src>` that isn't Webflow's and download those too; they're the spec for the animations.
- **Fonts with spaces in their filenames** may not load from the mirror, which changes text wrapping. Compare visually against the live site while it's still online.

PHASE 1 · PREPARATION

STEP 1.3: EXPORT WHAT ISN'T IN THE HTML

WHAT	WHERE TO GET IT	WHY
Redirects	Webflow → Site settings → Publishing → 301 redirects. Save as `redirects.csv` (`source,target`)	They have to keep working after the move
robots.txt and sitemap.xml	Download from the live site	Originals to compare against
CMS content	**Webflow Data API v2**: create a site API token with the required read scopes; enumerate collections and their localized items	The HTML shows only rendered content. The API exposes collection fields, locale associations and item status; production publication must be verified against the live-items endpoint
Form setup	Webflow → Forms, plus any third-party form (newsletter tool, CRM)	You'll rebuild these in Phase 2
Tracking and consent	GTM container ID, cookie banner tool, consent settings	Same

Put everything into a `webflow/` folder in the new repository and treat it as **read-only**.

CMS export checklist: enumerate collections; page through *all* results (do not assume a single response is complete); use each collection's field schema; preserve CMS item IDs, per-locale slugs, translation associations, ordering and asset references; compare staged vs live items and use live publication as the authority for which pages appear on the public website. Validate the exported record counts before converting rich text.

PHASE 1 · PREPARATION

STEP 1.4: WRITE THE RULES FOR THE AI (`CLAUDE.md`)

Put architecture constraints, file locations and acceptance criteria in `CLAUDE.md`. Claude Code loads these project instructions as context, allowing shorter task-specific prompts and more consistent implementation.

What we put in it (shortened):

CLAUDE.MD · MARKDOWN

```
# example.com — Webflow → Astro migration

Rebuild www.example.com (EN/DE) as a static Astro site. Content and URLs must
match the live Webflow site 1:1. Design is reproduced from the mirror, not
improved unless asked.

## Layout
- `site/` — the Astro app. All code goes here.
- `webflow/` — read-only reference. Never edit, never import from it at runtime.
  - `scraping/mirror/` — crawled HTML/CSS/assets of the live site
  - `scraping/urls.txt` — expected legacy URLs = the acceptance baseline
  - `redirects.csv` — existing 301s

## Stack rules
- `.astro` components by default. React only for interactive islands
  (demo form, cookie consent, filters), hydrated with client:visible or client:idle.
- Tailwind for styling. No Webflow class names, no webflow.js, no Webflow CSS.
- TypeScript strict. No new dependencies without asking.

## URLs & i18n
- EN at root (/about), DE under /de (/de/about). No trailing slashes.
- Every legacy URL in urls.txt must keep resolving. Approved new pages are allowed
  and tracked in a separate additions manifest.
- DE slugs are translated and stored per item — never derive them from EN slugs.
- hreflang: en, de-DE, x-default → EN. Only emit an alternate if the translation exists.

## Content model
- Markdown, one file per item and locale: blog, legal, jobs.
- YAML, one file per collection, translatable fields as { en, de }.
- draft: true items stay in the repo but never produce pages, list entries or sitemap entries.

## Assets
- All images local, rendered via astro:assets. Nothing may load from
  website-files.com or any Webflow domain — grep the build to verify.

## Definition of done for any page
1. Visually matches the mirror page (desktop + mobile).
2. Same title, meta description, canonical, OG tags as the mirror.
3. `npm run build` passes and the URL check reports no missing URLs.

## Don'ts
- Don't touch webflow/.
- Don't rewrite copy or translate content yourself — copy it from the mirror/CMS data.
- Don't change URLs or slugs.
```

PHASE 1 · PREPARATION

> For this migration, we kept `git commit`, `git push`, package installation and cloud deployment commands under human control: Claude Code proposed commands, and we executed them after review. Document that execution policy alongside the project rules.

Also add a short `BASIC_INFORMATION.md` about your company (what you do, products, audience, tone). The AI uses it whenever it needs context, for example when you later ask it to write a blog post.

STEP 1.5: DECIDE THE TARGET URL SCHEME AND STACK

Write these decisions down before the first prompt; changing them later means touching every page:

DECISION	OUR CHOICE	NOTE
Framework	Astro, static output	No server, no adapter
URL format	`/about`, no trailing slash, served from `about.html`	Match what Webflow did
Languages	EN at `/`, DE at `/de/`	Astro's built-in i18n routing
Styling	Tailwind	Tokens extracted from Webflow's CSS
Content	Astro content collections (Markdown + YAML)	Validated by schemas at build time
Interactivity	Small React "islands" + plain scripts	Only where something is interactive

PHASE 1 · PREPARATION

CHECKLIST: PHASE 1

- [ ] `urls.txt` from the sitemap, plus unlinked pages added by hand
- [ ] Full wget mirror, plus JS-loaded assets and external custom scripts
- [ ] `redirects.csv`, `robots.txt`, `sitemap.xml`
- [ ] Webflow API token (stored in a git-ignored `.env`)
- [ ] Inventory of forms, tracking, consent and third-party scripts
- [ ] `CLAUDE.md` with rules and a definition of done
- [ ] `BASIC_INFORMATION.md` about the company

PHASE 2 OF 3

PHASE 2: MIGRATION

STEP 2.1: SET UP THE PROJECT

1. **Editor:** VS Code with the Claude Code extension (the terminal CLI works just as well).
2. **Repository:** create it with the structure from Step 1.4 (`site/`, `webflow/`, `CLAUDE.md`, `docs/`).
3. **Astro app** in `site/`, then Tailwind. You run the install commands yourself:

BASH

```
npm create astro@latest site
cd site && npx astro add tailwind react sitemap
```

PHASE 2 · STEP 2.1

4 **Base config** in `astro.config.mjs`: site URL, `output: 'static'`, `build.format: 'file'`, `trailingSlash: 'never'`, i18n with `prefixDefaultLocale: false`. Example (adapt the locales, domain and integrations):

ASTRO.CONFIG.MJS · JS

```
import { defineConfig } from 'astro/config';
import sitemap from '@astrojs/sitemap';
import react from '@astrojs/react';

export default defineConfig({
  site: 'https://www.example.com',
  output: 'static',
  build: { format: 'file' },
  trailingSlash: 'never',
  i18n: {
    defaultLocale: 'en',
    locales: ['en', 'de'],
    routing: { prefixDefaultLocale: false },
  },
  integrations: [react(), sitemap()],
});
```

`build.format: 'file'` generates `about.html` for a route such as `/about`. Whether the bare path `/about` works, and whether `/about/` redirects, is **hosting configuration**, not something `trailingSlash: 'never'` guarantees for prerendered pages.

5 **Create a baseline commit.** Commit after each verified page type so regressions can be isolated or reverted.

PHASE 2 · MIGRATION

STEP 2.2: THE WORKING PATTERN

We used the same implementation-and-review cycle for every page type:

1. **One prompt per page type** (homepage, solutions, about, ...), in both languages at once.
2. **The AI builds it and runs the URL check.**
3. **You compare it with the live site** in the browser, desktop and mobile.
4. **For every difference, ask first:** *"The images in the timeline are missing on the new about page. Why is that? Do not change code yet."* You get an analysis and usually several options. You decide, then say *"go for your recommendation"* or pick another option.
5. **Commit**, next page.

REPEAT FOR EVERY PAGE TYPE

Requesting a diagnosis before an implementation change avoided premature fixes and unnecessary rewrites. We used this especially when a discrepancy might require an architectural choice rather than a CSS adjustment.

PHASE 2 · MIGRATION

STEP 2.3: THE MAIN PROMPTS

The following prompts follow our migration sequence. We refined their wording to incorporate the issues and checks identified during the project.

1. EXTRACT THE CMS

PROMPT

Write site/scripts/extract-webflow.ts: enumerate Webflow Data API v2 CMS collections and locales using a token from .env. Handle pagination and API errors. Fetch staged and live items separately, preserve item IDs, locale IDs, translated slugs, sort order and asset references, and determine public visibility from the live-items endpoint (not isDraft alone). Download referenced assets and generate content files and content schemas per CLAUDE.md. Never overwrite hand-edited content without explicit confirmation. Report item counts and mismatches per collection and locale.

2. BUILD THE ACCEPTANCE TEST

PROMPT

Write site/scripts/check-urls.ts: compare the legacy URL manifest and approved new-page manifest with Astro's built dist/ files, respecting build.format=file, localized slugs, percent encoding and the homepage. List missing legacy routes, approved additions and unexpected outputs. Separately generate a deployed-site HTTP test for status codes, trailing-slash redirects and preserved query strings. Add the local check as npm run check.

3. DESIGN TOKENS, LAYOUT, HOMEPAGE

PROMPT

Build the base layout (head/SEO/hreflang, nav with language switch, footer, cookie banner placeholder) and the homepage for EN and DE, matching the mirror. Extract design tokens into Tailwind first.

PHASE 2 · STEP 2.3

4. ONE PAGE TYPE AT A TIME

PROMPT

Build /solutions and /de/solutions from webflow/scraping/mirror/www.example.com/solutions.html and de/solutions.html. Reuse the existing layout and components; only create new components for sections the homepage doesn't have. Copy all text verbatim from the mirror. Match the SEO tags. Run npm run check.

PROMPT

Build /about and /de/about from the mirror. Render team members, timeline, about-strip images and logos from their YAML collections, not hardcoded. Copy text verbatim, match the SEO tags, and run npm run check.

PROMPT

Build /contact and /de/contact from the mirror. Render the form as a static, visually identical placeholder (no submit logic yet; that comes in the islands step). Match the SEO tags and run npm run check.

PROMPT

Build /careers and /de/careers with the job list from the jobs collection (excluding drafts), plus the detail template /open-positions/[slug] and /de/open-positions/[slug] using the per-locale slugs. Match the mirror for list and detail. Emit hreflang only when a translationKey partner exists. Run npm run check.

PROMPT

Build the blog list (/blog, /de/blog), the post template (/blog/[slug], /de/blog/[slug]) and the category pages (/category/[slug], /de/category/[slug]) from the blog and categories collections, excluding drafts. Use per-locale slugs. Emit hreflang only when a translationKey partner exists, and add no fallback pages for missing translations. Match the mirror's list, post and category layouts, including dates, categories and related posts if present. Add the Article JSON-LD only if the mirror has it. Run npm run check.

PROMPT

Build /legal/[slug] and /de/legal/[slug] from the legal collection. Render the content exactly as extracted, with no rewording or reformatting. Diff the rendered text against the mirror pages and report any differences. Run npm run check.

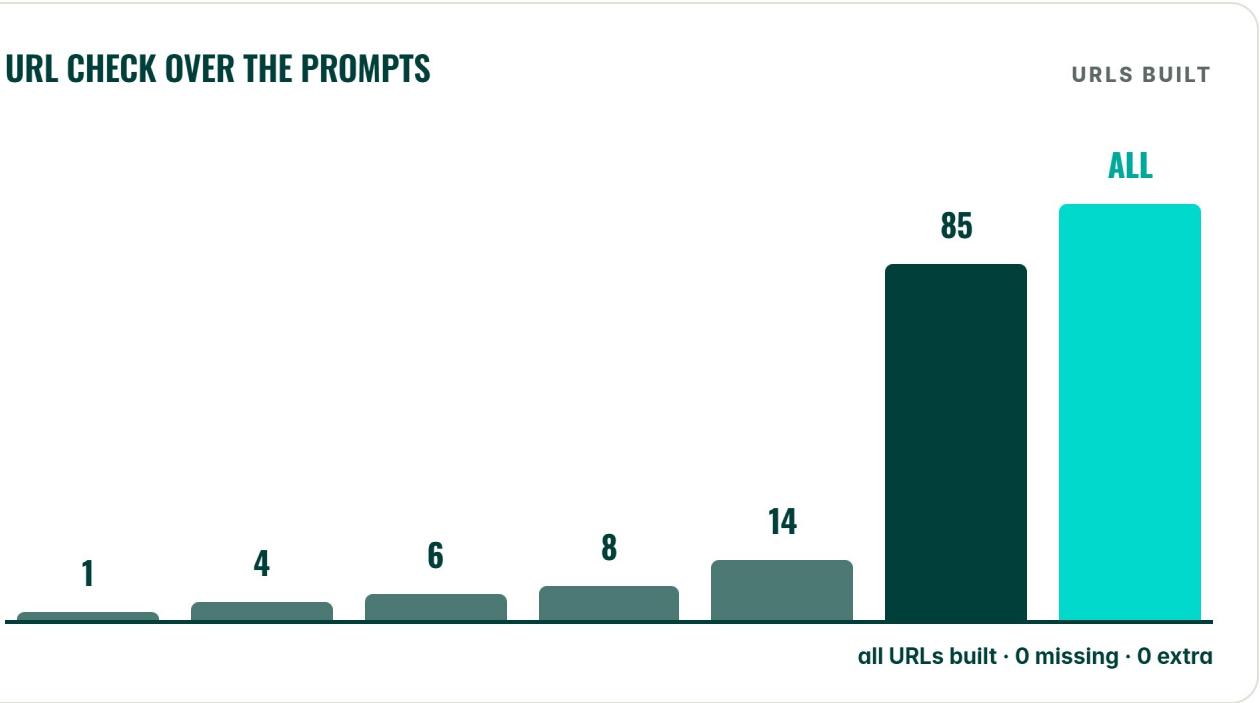
PHASE 2 · STEP 2.3

5. DOCUMENT AS YOU GO

PROMPT

Please write multiple documentations (markdown) in docs folder: the complete history/approach of our migration, how GTM is handled, how the cookie banner works (Consent Mode v2 etc.), how forms work, and a TODO list of all open tasks.

The URL check went from 1 to 4, 6, 8, 14, 85 and finally **all URLs built, 0 missing, 0 extra** over these prompts. By the end of day one, every page type existed in both languages. The remaining days went into details, animations, forms and hosting.



PHASE 2 · MIGRATION

STEP 2.4: CMS CONTENT: WHAT TO WATCH FOR

The extractor turns Webflow's CMS into Markdown and YAML files in the repo. Things that needed attention:

- **Don't infer public visibility from the draft flag alone.** In our export, one `isDraft` flag contradicted what was live. Use Webflow's *live items* endpoint as the publication check, and validate localized items independently.
- **Rich text → Markdown** needs custom rules to render identically: keep non-breaking spaces, keep empty paragraphs (Webflow uses them as spacers), stop plain-text URLs from becoming links, keep embeds (charts, videos) as raw HTML.
- **Images inside rich text** must be downloaded and rewritten to local paths, so Astro can optimize them. Make the script fail if any Webflow URL is left over.
- **Order isn't preserved** by Astro's `getCollection`. Store the CMS position as an `order` field.
- **Translations:** link EN and DE versions with a shared `translationKey`, store each slug separately, and create no fallback pages for missing translations.
- **Once you edit content in the repo, stop running the extractor.** It overwrites the files.

STEP 2.5: ANIMATIONS AND INTERACTIONS

Webflow interactions (IX2) don't export. Rebuild them with standard libraries:

WEBFLOW FEATURE	REPLACEMENT
Scroll animations, pinning, counters	GSAP + ScrollTrigger
Sliders, carousels	Swiper
Lottie animations	dotLottie web player
Marquees, hovers, transitions	Plain CSS (Tailwind utilities)
Modals, menus, dropdowns	Small scripts or React islands

PHASE 2 · STEP 2.5

- **If the site uses custom JavaScript**, give the AI the original script as the spec: *"port the hero animation from homeCopy.js"*. That's how we got the exact offsets and timings.
- **Load heavy libraries only where needed.** Astro bundles a page's script only for that page; we also load the Lottie player only when its section approaches the viewport.
- **Everything must render without JavaScript.** The animation only adds movement to a static layout.

STEP 2.6: REDIRECTS

- Convert `redirects.csv` into explicit `301` rules in your host's format (`_redirects` for Netlify/Cloudflare, `.htaccess` for Apache, generated JSON rules for AWS Amplify). Keep them out of the framework config, so they work on the server without JavaScript.
- **Check every target.** We found one redirect pointing to a page that didn't exist and one with the wrong language slug.
- **Trailing slashes:** Webflow redirected `/about/` → `/about`. Astro's `trailingSlash` option does **not** enforce this on a static host; configure it at the hosting layer and test the actual HTTP status code.
- If the sitemap address changes (Astro's integration writes `/sitemap-index.xml`), add a redirect from the old `/sitemap.xml`.

STEP 2.7: SEO PARITY

- **Meta tags:** title, description, canonical, Open Graph, Twitter and hreflang should be identical to the old site. Have the AI compare them programmatically for every page in the mirror, not just spot-check.
- **Sitemap:** the official `@astrojs/sitemap` integration picks up new pages automatically. We tried a custom sitemap that reproduced Webflow's exactly and dropped it: every new page type would have needed manual code, and per-page hreflang tags are enough for Google.
- **Fix only what's clearly broken.** Six Webflow posts had an empty `<title>`; we fell back to the post title. Everything else we copied 1:1, quirks included, and documented every deliberate difference.

PHASE 2 · MIGRATION

STEP 2.8: FORMS

Webflow's form backend goes away with Webflow. Decide where submissions go:

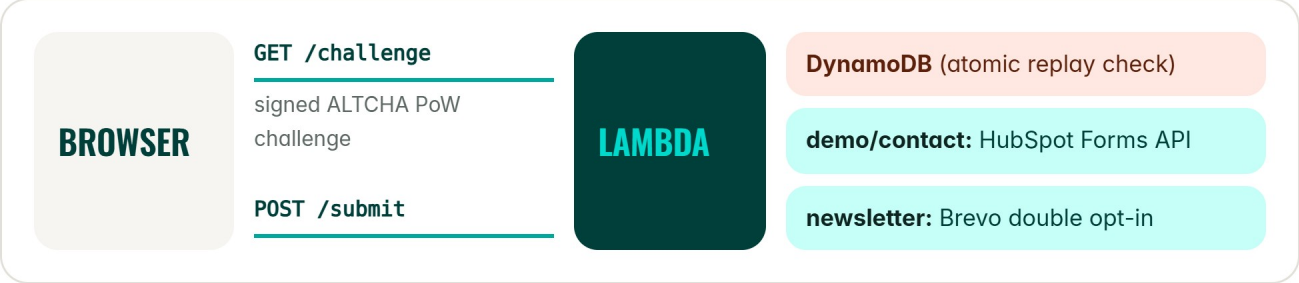
OPTION	EFFORT	GOOD FOR
Your CRM's form API directly from the browser (e.g. HubSpot Forms API)	Low	Quick start; but no server-side spam protection, and the endpoint is public
A form service (Formspree, Basin, Netlify Forms, ...)	Low	Small sites without a CRM; another subscription
Your own serverless function (AWS Lambda, Cloudflare Worker, Netlify Function)	Medium	Full control: spam protection, routing to several targets, no secrets in the site

What we did: one AWS Lambda behind an API Gateway, in the EU (Frankfurt). The site stays fully static and posts to it:

FLOW

```
browser ──GET /challenge──▶ Lambda  (signed ALTCHA PoW challenge)
browser ──POST /submit────▶ Lambda ──▶ DynamoDB (atomic replay check)
                                   └─▶ demo/contact: HubSpot Forms API
                                       newsletter: Brevo double opt-in
```

THE SAME FLOW AS A DIAGRAM



PHASE 2 · STEP 2.8

- **Spam protection without a third-party CAPTCHA:** an invisible proof-of-work challenge (ALTCHA, self-hosted), honeypot, origin validation and rate limiting. Localyzer **stores challenge-use records in Amazon DynamoDB to block replay**. ALTCHA's cryptographic verification proves that the solution matches the challenge; it does not, by itself, enforce single use.
- **DynamoDB replay prevention:** derive a stable identifier from the verified challenge payload (for example, a hash of its canonical signed challenge identity). After signature/solution/expiry validation and **before** forwarding the form, atomically claim that identifier with a DynamoDB conditional `PutItem` (`attribute_not_exists(pk)`). Exactly one concurrent request can acquire the claim; reject `ConditionalCheckFailedException` as a replay. Do **not** implement this as `GetItem` followed by an unconditional write: that races under concurrent submissions.
- **TTL and expiry:** store an expiration timestamp to age out challenge records, but never treat DynamoDB TTL deletion as immediate. Enforce the challenge expiration cryptographically and use a replay-record retention window sufficiently long to prevent an expired-but-replayed payload from becoming acceptable. Log failed claims without logging form contents or secrets.
- **Failure behavior:** fail closed if challenge verification or the DynamoDB claim fails unexpectedly. Decide whether a downstream CRM failure should issue a fresh challenge for a retry or use idempotent queued delivery; never reopen a previously claimed challenge.
- **No keys in the site.** API keys live in AWS Systems Manager; form IDs in the Lambda config.
- **Alarms** by email when HubSpot or Brevo reject a submission.
- **Preserve the form UX:** submitting state, success confirmation, error feedback and thank-you pages.

PHASE 2 · STEP 2.8

Claude Code generated the Lambda, tests, infrastructure template and deployment scripts. We reviewed them and executed deployments ourselves.

Security checklist: supply the ALTCHA signing secret and upstream service credentials from a secret manager, configure scoped IAM permissions for the DynamoDB table and upstream API access, bound request-body sizes and timeouts, validate allowed origins (origin checks are not authentication), validate fields server-side, and rate-limit requests at the edge or API. Test malformed signatures, expired challenges, a repeated submission, **two parallel submissions of the same challenge**, rejected CRM requests and upstream timeouts.

STEP 2.9: TRACKING, CONSENT AND LEGAL

- **Google Tag Manager:** copy the snippet as is; the container keeps working.
- **Cookie banner:** if it was a Webflow add-on (we had Finsweet), rebuild it as a small React island with the same texts and the same Consent Mode v2 signals to GTM. Test with GTM's preview mode that the tags respect the consent.
- **Add a "Cookie preferences" link** in the footer to reopen the banner.
- **Privacy policy:** update it for the new hosting provider, the form processing and any new services. Have it reviewed.

PHASE 2 · MIGRATION

STEP 2.10: VERIFY EVERYTHING

Before deployment, the AI ran these checks; we repeated them after every larger change:

CHECK	HOW
Build passes	`npm run build`
Every URL exists, nothing extra	`npm run check` against `urls.txt`
Meta tags identical	Script comparing title, description, canonical, OG and Twitter tags of every page with the mirror
Visible text identical	Script comparing body text character by character, including non-breaking spaces
Layout matches	Headless Chromium screenshots and page heights at 1440, 800 and 390px against the live site
No Webflow left	`grep -r "website-files.com" dist/` returns nothing
Interactions	Forms, modal, menus, language switch, "load more", cookie banner tested in a browser

Measure, don't eyeball. Most layout bugs (a missing image `width` attribute, a container forcing full width, a wrong font size on an icon) only showed up as a few pixels of height difference against the live page.

PHASE 2 · MIGRATION

CHECKLIST: PHASE 2

- [ ] Project set up, first commit
- [ ] URL check script, passing
- [ ] Animations rebuilt
- [ ] SEO tags identical, sitemap and robots.txt in place
- [ ] GTM, cookie banner and privacy policy updated
- [ ] Docs: how the site works, how to add a post, open tasks
- [ ] CMS extracted, drafts handled, images local
- [ ] Design tokens, base layout, every page type in every language
- [ ] Redirects converted and checked
- [ ] Forms working end to end, with spam protection
- [ ] All verification checks pass

PHASE 3 OF 3

PHASE 3: DEPLOYMENT

3.1 Choose a host
3.2 Set up hosting as code
3.3 Test on staging
3.4 Prepare DNS
3.5 Switch day
3.6 After go-live

STEP 3.1: CHOOSE A HOST

An Astro static build is a folder of HTML, CSS, JS and images. Almost any host works; the differences are in redirects, previews and where your other infrastructure lives.

PHASE 3 · STEP 3.1

HOST	REDIRECTS	BRANCH PREVIEWS	NOTE
Netlify	`_redirects` file	Yes	Very easy; built-in form handling
Cloudflare Pages	`_redirects` file	Yes	Generous free tier, global CDN
Vercel	`vercel.json`	Yes	Very easy
AWS Amplify Hosting	Rules via API/console (JSON)	Yes, with password protection	Good fit if you already use AWS
Classic web hosting (Apache)	`.htaccess`	No	Works; deploy by upload

We chose AWS Amplify, because the forms Lambda already runs in the same AWS account in the EU, staging branches can be password-protected, and the build and redirect configuration can be maintained as code.

Hosting and serverless request costs can be low for a modest-traffic static marketing site, but vary with traffic, build minutes, bandwidth and provider pricing. Compare total cost of ownership, including AI tools, internal maintenance, monitoring and any external support—not just hosting subscriptions.

STEP 3.2: SET UP HOSTING AS CODE

- **Build spec** in the repo: app root `site/`, Node version, `npm ci`, `npm run build`, output `site/dist`.
- **Redirect rules generated from** `_redirects`, so there's one source of truth: 32 redirects, one trailing-slash redirect per page, and 404 handling per language. Ensure the generator produces Amplify's JSON rule format and checks that specific rules precede broad wildcards. A catch-all rule can shadow every later rule.
- **Query strings:** Amplify generally preserves query parameters in 301/302 redirects, but documents exceptions when conditions or destination query parameters are specified. Include query-string assertions in your redirect tests.
- **404 behavior:** test an unknown English and German URL. A custom error page should return HTTP **404**, not an SPA-style 200 rewrite.
- **Cache headers:** hashed assets (`/_astro/*`) and fonts cached for a year; HTML not cached.
- **Branches:** `main` → production, `integration` → staging.

PHASE 3 · DEPLOYMENT

STEP 3.3: TEST ON STAGING

Connect a staging subdomain (`staging.example.com`), protected with a password so it isn't indexed. Then test with a script, not by hand:

- [ ] every page in the URL list returns 200
- [ ] every redirect returns 301 to the right target, query string kept
- [ ] `/page/` → 301 → `/page`
- [ ] unknown URLs return a 404 with the right language's 404 page
- [ ] `robots.txt` and the sitemap return 200
- [ ] forms submit end to end (add the staging domain to the allowed origins first)
- [ ] ALTCHA valid/invalid/expired/replayed challenges, including concurrent reuse, are correctly rejected or accepted

Then by hand: menu, footer, language switch, blog pagination and categories, thank-you pages, OG images, fonts, favicon, and DevTools → Network shows no request to a Webflow domain.

STEP 3.4: PREPARE DNS

- **Don't touch MX and TXT records** (email, SPF, domain verifications). Write them down before you change anything.
- **Lower the TTL** of the `www` and root records ahead of the switch (we used 60 seconds). This reduces caching delays for resolvers that honor the new TTL; cached records and registrar forwarding can still delay cutover or rollback.
- **Connect `www` to the new host first** and validate the TLS certificate before changing the traffic record (we used DNS validation). Check the certificate independently of the DNS cutover.
- **The root domain (`example.com`):** many registrars can't point the root to a CDN (no ALIAS/ANAME records). Use the registrar's HTTP forwarding to `https://www.example.com` instead. Test it on a throwaway subdomain first: we checked HTTPS, that path and query are kept, and the status code (our registrar only offers 302, which we accepted).

PHASE 3 · DEPLOYMENT

STEP 3.5: SWITCH DAY (ABOUT 30 MINUTES)

1. Make the production branch public (remove the password protection).
2. Point the `www` CNAME from Webflow to your new host.
3. Set up the root domain forwarding (this replaces Webflow's A record).
4. After 1–2 minutes: run the staging check script against the production domain, test `example.com/about` → `www.example.com/about`, submit every form in every language, send and receive one email (MX unchanged).
5. Google Search Console: submit the new sitemap and remove the old one.

Rollback: restore the previous `www` DNS record and root-domain forwarding, and re-run the URL and form checks. Keep the Webflow site published until the new deployment is stable. A low TTL can accelerate rollback, but propagation is not guaranteed to finish within minutes.

STEP 3.6: AFTER GO-LIVE

- [ ] Raise the DNS TTLs back to normal (e.g. 1 hour)
- [ ] Watch Search Console coverage and the form alarms for the first weeks
- [ ] Unpublish the Webflow site after a few days without problems, then cancel the Webflow plan
- [ ] End data processing agreements you no longer need (Webflow) and add the new ones (your host)
- [ ] Remove test contacts from the CRM and newsletter tool

AFTER THE MIGRATION: RUNNING THE SITE WITH AI

The migration is the one-off part. **The real gain is afterwards:** the site is now maintained in a repository, and an AI agent can implement much of the work that previously required the Webflow designer or an agency—subject to human review and testing.

What that looks like day to day:

PROMPT

Create a new blog post in English and German from the attached draft. Category: Report. Use the attached image as header image. Pick slugs, set the date to today.

The AI creates both Markdown files, links them as translations, adds the image, runs the build and shows you the result locally. You review it and commit; the host deploys automatically.

Other tasks we've handed over since go-live:

- new pages (an AI product page, thank-you pages, a 404 page in both languages)
- splitting the privacy policy into website and platform versions
- adding a "Cookie preferences" link to every footer variant
- performance work (lazy-loading animations, cache headers)
- Consent Mode updates

The rules in `CLAUDE.md` still apply to every one of these: no new dependencies without asking, no copy invented, no URL changes, nothing committed or deployed without you.

AFTER THE MIGRATION

WHAT CHANGED OPERATIONALLY?

Our old bottleneck was not the amount of HTML a website needs. It was that small changes required coordination with an external specialist. Now a developer or technically experienced owner can ask an agent to implement a change in the repository, inspect the diff, run the tests and deploy through a controlled pipeline.

That does **not** make all editing effortless. Nontechnical colleagues still need a suitable content workflow; Git and Markdown are not automatic substitutes for a visual CMS. And the team now owns dependency updates, integration maintenance, security, hosting and release quality.

DOES IT SAVE MONEY?

COST OR RESPONSIBILITY	WEBFLOW + AGENCY WORKFLOW	ASTRO + AI CODING WORKFLOW
Platform and hosting	Webflow site/CMS subscription	Static hosting, builds and bandwidth
Content editing	Visual CMS and editor access	Repository content or an additional editorial UI
Technical changes	Agency or Webflow specialist, depending on team skills	AI-assisted implementation plus human review
Forms and integrations	Platform features and third parties	Independently maintained services and integrations
Ongoing responsibility	Shared with Webflow and service providers	More directly owned by your team
Platform dependency	Webflow designer/runtime and export constraints	Lower builder dependency; normal framework/provider dependencies remain

AFTER THE MIGRATION

The ongoing savings come from handling technical changes internally instead of commissioning an agency for every update. We now own the corresponding engineering work, including dependency maintenance, infrastructure and release testing.

WHEN STAYING ON WEBFLOW MAY BE THE BETTER CHOICE

If nontechnical editors routinely create and change visual layouts, an integrated visual workflow can be more valuable than source-code control. The same applies if there is no one willing to own production releases, integrations and dependency maintenance. In those circumstances, migrating solely because AI coding exists is likely the wrong decision.

Our case was different: Webflow had not given our colleagues the independence we wanted, while our team had the technical capability to operate a code-first site. AI coding made that switch practical.

LESSONS LEARNED

1. **Freeze the reference.** Save the HTML mirror, assets, CMS export, redirects and URL manifest before rebuilding. Use the same snapshot for acceptance tests throughout.
2. **Put reusable constraints in `CLAUDE.md`.** Define the stack, file layout, translated slugs, asset rules and page-level completion checks once rather than repeating them in every prompt.
3. **Ask before you change.** "Do not change code yet" turned many would-be wrong turns into a quick decision. We once let the AI build a custom sitemap generator and then reverted to the standard integration; asking first would have saved that detour.
4. **Never let text be retyped.** Copy must come from the mirror or the CMS export. The few times text was "completed from memory", it was wrong, and only the automated tag comparison caught it.
5. **Audit source data rather than assuming it is perfect.** In our export we encountered a draft-status inconsistency, invalid CSS values and a broken redirect. Decide case by case whether to copy or fix, and write the decision down.
6. **Compare screenshots and geometry.** Our browser tests at 1440, 800 and 390px exposed spacing and height regressions that were hard to notice in a casual review.
7. **Review commands that change state.** We manually approved commits, dependency installs, deployments and DNS changes after inspecting AI-generated code and proposed commands.
8. **Document while you go.** We asked the AI to write the docs during the project: how forms work, how to add a post, what's still open. They're also what makes the next AI session productive.

FINAL PERSPECTIVE

The surprising part was not that Astro could serve a static website. It was that AI coding changed the **make-or-buy decision**. A migration we would previously have commissioned externally became manageable internally, without discarding the existing design, content or SEO structure.

TECHNICAL REFERENCE DOCUMENTATION

Astro configuration: build format, trailing slash and i18n
https://docs.astro.build/en/reference/configuration-reference/

Webflow Data API: CMS collection items and localization
https://developers.webflow.com/data/reference

AWS Amplify Hosting: redirects, precedence and query strings
https://docs.aws.amazon.com/amplify/latest/userguide/redirects.html

AWS Amplify Hosting: rewrite and custom 404 examples
https://docs.aws.amazon.com/amplify/latest/userguide/redirect-rewrite-examples.html

Amazon DynamoDB: conditional writes
https://docs.aws.amazon.com/amazondynamodb/latest/developerguide/Expressions.ConditionExpressions.html

Amazon DynamoDB: TTL expiration behavior
https://docs.aws.amazon.com/amazondynamodb/latest/developerguide/TTL.html

ALTCHA documentation
https://altcha.org/docs/