Using Codex to Rebuild My Blog and Import 15+ Years of Blogger Posts

March 28, 2026 / Gavin Jackson

codex ai blog php markdown docker automation

Over the last few weeks I used Codex to help me rebuild **gavinj.net** from scratch.

The goal was not to create a giant CMS or a complicated devops pipeline. I wanted something much simpler: a fast, lightweight blog that I could understand end to end, host cheaply, edit easily, and keep under my own control.

I also wanted to bring across a large archive of old Blogger posts without turning the migration into a months-long cleanup project.

Codex (https://openai.com/codex/) turned out to be very good at exactly this kind of work.

What I Wanted

My requirements were pretty straightforward:

- no database
- no heavy framework
- content stored as plain files
- local development that was easy to run and easy to throw away
- a design that felt clean and personal rather than looking like a default theme
- a practical migration path from Blogger into the new structure

That led to a simple architecture:

- PHP for rendering
- Markdown files with YAML frontmatter for content
- a `posts/` directory as the source of truth
- Docker Compose for local testing
- a minimal production deployment on Apache with PHP

None of those choices are exotic, and that was the point. I wanted boring, understandable technology.

Why Codex Worked Well

What impressed me most was not that Codex could generate code. Plenty of tools can do that now.

What mattered was that it worked well as a collaborator across the whole lifecycle:

- helping scaffold the initial PHP blog engine

- refining the layout and visual style over multiple iterations
- adding features like RSS, tags, source viewing, and draft support
- writing migration scripts for the old Blogger archive
- helping reason through local dev and deployment setup
- handling small but annoying infrastructure details that usually slow projects down

That combination is where tools like this become genuinely useful.

Building the New Blog

The first step was creating a minimal blog engine instead of dropping in WordPress or another full CMS.

Codex helped build a simple PHP application that:

- scans the `posts/` directory for Markdown files
- parses YAML frontmatter
- renders posts into HTML
- builds the homepage, post pages, RSS feed, and tag views
- keeps everything file-based and easy to inspect

Using Markdown for articles was one of the best choices in the whole rebuild. It means the content is portable, easy to diff in Git, and pleasant to edit without needing a browser-based admin interface.

Each post is just a file with metadata at the top:

```
---
title: "Example title"
date: "2026-03-28"
tags: ["php", "markdown"]
author: "Gavin Jackson"
excerpt: "Short summary for listings."
---
```

That gives me a format that is simple enough to maintain manually, but structured enough to support tagging, excerpts, draft handling, and publication dates.

Iterating on the Design Together

The design did not appear fully formed in one shot.

This was much more of a back-and-forth process, which is exactly how real development usually works. We started with a clean dark layout, then kept refining it:

- spacing
- typography
- post cards

- navigation
- accent colors
- mobile behaviour
- the tag cloud
- source view rendering
- the resume page

This was one of the more useful patterns with Codex. I could say things like "this feels too generic", "move that into the sidebar", or "make the hover accent match the site branding", and then iterate quickly from there.

It felt less like using a code generator and more like pairing with someone who can make changes quickly once the direction is clear.

Importing the Old Blogger Archive

The second major problem was content migration.

I have a lot of old posts going back many years, and I did not want to manually copy and paste them into the new system. Codex helped create an import script that pulled entries from the Blogger Atom feed and turned them into local post files.

The first pass imported legacy posts with frontmatter and preserved the original HTML body. That was a smart intermediate step because it got the archive into the new framework quickly without blocking on perfect conversion.

From there, I could progressively improve the content instead of trying to solve everything up front.

The import process included things like:

- extracting titles and publication dates
- generating slugs
- capturing categories as tags
- storing original publication timestamps
- preserving legacy URLs
- generating excerpts automatically

That meant the old archive became part of the new site structure almost immediately.

Converting Old HTML to Markdown

Once the import was done, the next step was improving the archive quality.

A lot of older Blogger posts were HTML-heavy, which is normal for content that has lived through multiple platforms and editors. Codex helped write a second migration script to convert those imported HTML posts into cleaner Markdown.

That script handled:

- headings
- paragraphs
- lists
- links
- blockquotes
- inline code and code blocks
- embedded images
- special handling for old monospace code spans

It also materialised image assets into the local repository so the new site was no longer dependent on external hosted content for every imported image.

That was a good example of where AI assistance saved real time. Writing and refining HTML-to-Markdown conversion logic is fiddly work. It is not impossible, but it is exactly the kind of task where having a fast coding partner is valuable.

Local Development with Docker Compose

I wanted local testing to be easy and disposable, so we added a local `docker-compose.yml` setup with `nginx` and `php-fpm`.

That gave me a repeatable dev environment that I could start with:

```
docker compose up --build
```

The setup also included:

- a custom PHP image
- Composer installed in the container
- automatic dependency installation on first start
- bind mounts for live editing
- a named volume for `vendor/` so dependencies persisted across restarts

That is exactly the kind of practical quality-of-life improvement I appreciate. It meant I could work on the site without having to remember local PHP versions, install steps, or one-off machine setup.

Deployment Help Matters Too

One thing I want to emphasize is that Codex was not only useful for writing the application.

It was also helpful in the deployment phase.

Once the site itself was in good shape, I still needed to get it running properly in production. That included things like:

- Apache virtual host configuration

- rewrite rules
- PHP module setup
- checking the right pieces were enabled
- getting Let's Encrypt SSL certificates in place
- generally closing the loop from "works on my machine" to "actually live"

That part is often underappreciated when people talk about AI coding tools. Real projects do not stop at code generation. There is always a layer of server configuration, packaging, permissions, services, and certificates that has to be sorted out before a site is genuinely done.

Having help there was just as valuable as having help with the PHP and CSS.

What I Still Liked About Being in Control

Using Codex did not mean handing the project over and hoping for the best.

I still made the architectural choices. I still reviewed the code, decided what to keep, adjusted the content, and steered the design. The value came from compressing the implementation and iteration time, not from removing human judgment.

That is probably the best way to think about tools like this.

They are strongest when:

- you know roughly what you want
- you can review the output
- you are willing to iterate
- you want to move faster through the boring or fiddly parts

For me, that was exactly the situation here.

The Result

The end result is a blog that feels much closer to what I wanted all along:

- lightweight
- easy to host
- Git-friendly
- based on plain files
- easy to extend
- easier to maintain than my old setup
- capable of carrying both new writing and old archive content

Just as importantly, the process was enjoyable.

Instead of getting bogged down in boilerplate, migration drudgery, or deployment trivia, I could keep making decisions and moving forward. Codex handled a lot of the implementation load, but still left me in control of the final shape of the site.

That is the part I think people sometimes miss. The best use of tools like this is not pressing a button and walking away.

It is collaborating with them to get better results faster.

And for this rebuild, that worked remarkably well.
