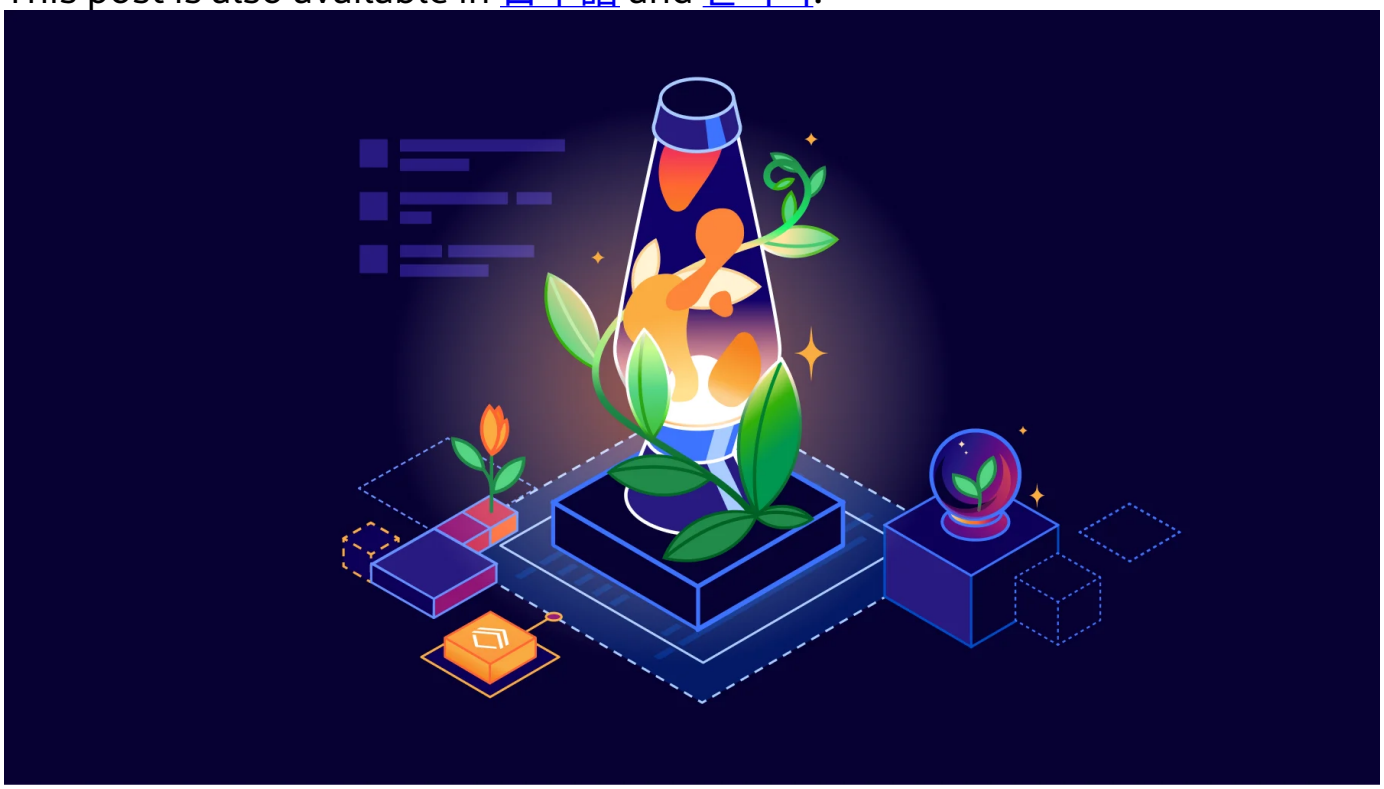
How we rebuilt Next.js with AI in one week

2026-02-24

Steve Faulkner

10 min read

This post is also available in 日本語 and 한국어.



**This post was updated at 12:35 pm PT to fix a typo in the build time benchmarks.*

Last week, one engineer and an AI model rebuilt the most popular front-end framework from scratch. The result, vinext (pronounced "vee-next"), is a

drop-in replacement for Next.js, built on Vite, that deploys to Cloudflare Workers with a single command. In early benchmarks, it builds production apps up to 4x faster and produces client bundles up to 57% smaller. And we already have customers running it in production.

The whole thing cost about $1,100 in tokens.

The Next.js deployment problem

Next.js is the most popular React framework. Millions of developers use it. It powers a huge chunk of the production web, and for good reason. The developer experience is top-notch.

But Next.js has a deployment problem when used in the broader serverless ecosystem. The tooling is entirely bespoke: Next.js has invested heavily in Turbopack but if you want to deploy it to Cloudflare, Netlify, or AWS Lambda, you have to take that build output and reshape it into something the target platform can actually run.

If you're thinking: "Isn't that what OpenNext does?", you are correct.

That is indeed the problem OpenNext was built to solve. And a lot of engineering effort has gone into OpenNext from multiple providers, including us at Cloudflare. It works, but quickly runs into limitations and becomes a game of whack-a-mole.

Building on top of Next.js output as a foundation has proven to be a difficult and fragile approach. Because OpenNext has to reverse-engineer Next.js's build output, this results in unpredictable changes between versions that take a lot of work to correct.

Next.js has been working on a first-class adapters API, and we've been

collaborating with them on it. It's still an early effort but even with adapters, you're still building on the bespoke Turbopack toolchain. And adapters only cover build and deploy. During development, next dev runs exclusively in Node.js with no way to plug in a different runtime. If your application uses platform-specific APIs like Durable Objects, KV, or AI bindings, you can't test that code in dev without workarounds.

Introducing vinext

What if instead of adapting Next.js output, we reimplemented the Next.js API surface on Vite directly? Vite is the build tool used by most of the front-end ecosystem outside of Next.js, powering frameworks like Astro, SvelteKit, Nuxt, and Remix. A clean reimplementation, not merely a wrapper or adapter. We honestly didn't think it would work. But it's 2026, and the cost of building software has completely changed.

We got a lot further than we expected.

```
npm install vinext
```

Replace `next` with `vinext` in your scripts and everything else stays the same. Your existing `app/`, `pages/`, and `next.config.js` work as-is.

```
vinext dev          # Development server with HMR
vinext build        # Production build
```

```
vinext deploy       # Build and deploy to Cloudflare Workers
```

This is not a wrapper around Next.js and Turbopack output. It's an alternative implementation of the API surface: routing, server rendering, React Server Components, server actions, caching, middleware. All of it built on top of Vite as a plugin. Most importantly Vite output runs on any platform thanks to the Vite Environment API.

The numbers

Early benchmarks are promising. We compared vinext against Next.js 16 using a shared 33-route App Router application. Both frameworks are doing the same work: compiling, bundling, and preparing server-rendered routes. We disabled TypeScript type checking and ESLint in Next.js's build (Vite doesn't run these during builds), and used force-dynamic so Next.js doesn't spend extra time pre-rendering static routes, which would unfairly slow down its numbers. The goal was to measure only bundler and compilation speed, nothing else. Benchmarks run on GitHub CI on every merge to main.

Production build time:

Framework	Mean	vs Next.js
Next.js 16.1.6 (Turbopack)	7.38s	baseline
vinext (Vite 7 / Rollup)	4.64s	1.6x faster
vinext (Vite 8 / Rolldown)	1.67s	4.4x faster

Client bundle size (gzipped):

Framework	Gzipped	vs Next.js
Next.js 16.1.6	168.9 KB	baseline

Framework	Gzipped	vs Next.js
vinext (Rollup)	74.0 KB	56% smaller
vinext (Rolldown)	72.9 KB	57% smaller

These benchmarks measure compilation and bundling speed, not production serving performance. The test fixture is a single 33-route app, not a representative sample of all production applications. We expect these numbers to evolve as three projects continue to develop. The full methodology and historical results are public. Take them as directional, not definitive.

The direction is encouraging, though. Vite's architecture, and especially Rolldown (the Rust-based bundler coming in Vite 8), has structural advantages for build performance that show up clearly here.

Deploying to Cloudflare Workers

vinext is built with Cloudflare Workers as the first deployment target. A single command takes you from source code to a running Worker:

```
vinext deploy
```

This handles everything: builds the application, auto-generates the Worker configuration, and deploys. Both the App Router and Pages Router work on Workers, with full client-side hydration, interactive components, client-side navigation, React state.

For production caching, vinext includes a Cloudflare KV cache handler that gives you ISR (Incremental Static Regeneration) out of the box:

```
import { KVCacheHandler } from "vinext/cloudflare";
import { setCacheHandler } from "next/cache";

setCacheHandler(new KVCacheHandler(env.MY_KV_NAMESPACE));
```

KV is a good default for most applications, but the caching layer is designed to be pluggable. That setCacheHandler call means you can swap in whatever backend makes sense. R2 might be a better fit for apps with large cached payloads or different access patterns. We're also working on improvements to our Cache API that should provide a strong caching layer with less configuration. The goal is flexibility: pick the caching strategy that fits your app.

Live examples running right now:

- App Router Playground
- Hacker News clone
- App Router minimal
- Pages Router minimal

We also have a live example of Cloudflare Agents running in a Next.js app, without the need for workarounds like getPlatformProxy, since the entire app now runs in workerd, during both dev and deploy phases. This means being able to use Durable Objects, AI bindings, and every other Cloudflare-specific service without compromise. Have a look here.

Frameworks are a team sport

The current deployment target is Cloudflare Workers, but that's a small part of the picture. Something like 95% of vinext is pure Vite. The routing, the

module shims, the SSR pipeline, the RSC integration: none of it is Cloudflare-specific.

Cloudflare is looking to work with other hosting providers about adopting this toolchain for their customers (the lift is minimal — we got a proof-of-concept working on Vercel in less than 30 minutes!). This is an open-source project, and for its long term success, we believe it's important we work with partners across the ecosystem to ensure ongoing investment. PRs from other platforms are welcome. If you're interested in adding a deployment target, open an issue or reach out.

Status: Experimental

We want to be clear: vinext is experimental. It's not even one week old, and it has not yet been battle-tested with any meaningful traffic at scale. If you're evaluating it for a production application, proceed with appropriate caution.

That said, the test suite is extensive: over 1,700 Vitest tests and 380 Playwright E2E tests, including tests ported directly from the Next.js test suite and OpenNext's Cloudflare conformance suite. We've verified it against the Next.js App Router Playground. Coverage sits at 94% of the Next.js 16 API surface. Early results from real-world customers are encouraging. We've been working with National Design Studio, a team that's aiming to modernize every government interface, on one of their beta sites, CIO.gov. They're already running vinext in production, with meaningful improvements in build times and bundle sizes.

The README is honest about what's not supported and won't be, and about known limitations. We want to be upfront rather than overpromise.

What about pre-rendering?

vinext already supports Incremental Static Regeneration (ISR) out of the box. After the first request to any page, it's cached and revalidated in the background, just like Next.js. That part works today.

vinext does not yet support static pre-rendering at build time. In Next.js, pages without dynamic data get rendered during `next build` and served as static HTML. If you have dynamic routes, you use `generateStaticParams()` to enumerate which pages to build ahead of time. vinext doesn't do that… yet.

This was an intentional design decision for launch. It's on the roadmap, but if your site is 100% prebuilt HTML with static content, you probably won't see much benefit from vinext today. That said, if one engineer can spend $1,100 in tokens and rebuild Next.js, you can probably spend $10 and migrate to a Vite-based framework designed specifically for static content, like Astro (which also deploys to Cloudflare Workers).

For sites that aren't purely static, though, we think we can do something better than pre-rendering everything at build time.

Introducing Traffic-aware Pre-Rendering

Next.js pre-renders every page listed in `generateStaticParams()` during the build. A site with 10,000 product pages means 10,000 renders at build time, even though 99% of those pages may never receive a request. Builds scale linearly with page count. This is why large Next.js sites end up with 30-minute builds.

So we built **Traffic-aware Pre-Rendering** (TPR). It's experimental today, and we plan to make it the default once we have more real-world testing behind it.

The idea is simple. Cloudflare is already the reverse proxy for your site. We

have your traffic data. We know which pages actually get visited. So instead of pre-rendering everything or pre-rendering nothing, vinext queries Cloudflare's zone analytics at deploy time and pre-renders only the pages that matter.

```
vinext deploy --experimental-tpr

  Building...
  Build complete (4.2s)

  TPR (experimental): Analyzing traffic for my-store.com (last 24h)
  TPR: 12,847 unique paths — 184 pages cover 90% of traffic
  TPR: Pre-rendering 184 pages...
  TPR: Pre-rendered 184 pages in 8.3s → KV cache

  Deploying to Cloudflare Workers...
```

For a site with 100,000 product pages, the power law means 90% of traffic usually goes to 50 to 200 pages. Those get pre-rendered in seconds. Everything else falls back to on-demand SSR and gets cached via ISR after the first request. Every new deploy refreshes the set based on current traffic patterns. Pages that go viral get picked up automatically. All of this works without `generateStaticParams()` and without coupling your build to your production database.

Taking on the Next.js challenge, but this time with AI

A project like this would normally take a team of engineers months, if not years. Several teams at various companies have attempted it, and the scope is just enormous. We tried once at Cloudflare! Two routers, 33+ module shims, server rendering pipelines, RSC streaming, file-system routing, middleware,

caching, static export. There's a reason nobody has pulled it off.

This time we did it in under a week. One engineer (technically engineering manager) directing AI.

The first commit landed on February 13. By the end of that same evening, both the Pages Router and App Router had basic SSR working, along with middleware, server actions, and streaming. By the next afternoon, App Router Playground was rendering 10 of 11 routes. By day three, `vinext deploy` was shipping apps to Cloudflare Workers with full client hydration. The rest of the week was hardening: fixing edge cases, expanding the test suite, bringing API coverage to 94%.

What changed from those earlier attempts? AI got better. Way better.

Why this problem is made for AI

Not every project would go this way. This one did because a few things happened to line up at the right time.

Next.js is well-specified. It has extensive documentation, a massive user base, and years of Stack Overflow answers and tutorials. The API surface is all over the training data. When you ask Claude to implement `getServerSideProps` or explain how `useRouter` works, it doesn't hallucinate. It knows how Next works.

Next.js has an elaborate test suite. The Next.js repo contains thousands of E2E tests covering every feature and edge case. We ported tests directly from their suite (you can see the attribution in the code). This gave us a specification we could verify against mechanically.

Vite is an excellent foundation. Vite handles the hard parts of front-end

tooling: fast HMR, native ESM, a clean plugin API, production bundling. We didn't have to build a bundler. We just had to teach it to speak Next.js. `@vitejs/plugin-rsc` is still early, but it gave us React Server Components support without having to build an RSC implementation from scratch.

The models caught up. We don't think this would have been possible even a few months ago. Earlier models couldn't sustain coherence across a codebase this size. New models can hold the full architecture in context, reason about how modules interact, and produce correct code often enough to keep momentum going. At times, I saw it go into Next, Vite, and React internals to figure out a bug. The state-of-the-art models are impressive, and they seem to keep getting better.

All of those things had to be true at the same time. Well-documented target API, comprehensive test suite, solid build tool underneath, and a model that could actually handle the complexity. Take any one of them away and this doesn't work nearly as well.

How we actually built it

Almost every line of code in vinext was written by AI. But here's the thing that matters more: every line passes the same quality gates you'd expect from human-written code. The project has 1,700+ Vitest tests, 380 Playwright E2E tests, full TypeScript type checking via tsgo, and linting via oxlint. Continuous integration runs all of it on every pull request. Establishing a set of good guardrails is critical to making AI productive in a codebase.

The process started with a plan. I spent a couple of hours going back and forth with Claude in OpenCode to define the architecture: what to build, in what order, which abstractions to use. That plan became the north star. From there, the workflow was straightforward:

1. Define a task ("implement the `next/navigation` shim with usePathname, `useSearchParams`, `useRouter`").
2. Let the AI write the implementation and tests.
3. Run the test suite.
4. If tests pass, merge. If not, give the AI the error output and let it iterate.
5. Repeat.

We wired up AI agents for code review too. When a PR was opened, an agent reviewed it. When review comments came back, another agent addressed them. The feedback loop was mostly automated.

It didn't work perfectly every time. There were PRs that were just wrong. The AI would confidently implement something that seemed right but didn't match actual Next.js behavior. I had to course-correct regularly. Architecture decisions, prioritization, knowing when the AI was headed down a dead end: that was all me. When you give AI good direction, good context, and good guardrails, it can be very productive. But the human still has to steer.

For browser-level testing, I used agent-browser to verify actual rendered output, client-side navigation, and hydration behavior. Unit tests miss a lot of subtle browser issues. This caught them.

Over the course of the project, we ran over 800 sessions in OpenCode. Total cost: roughly $1,100 in Claude API tokens.

What this means for software

Why do we have so many layers in the stack? This project forced me to think deeply about this question. And to consider how AI impacts the answer.

Most abstractions in software exist because humans need help. We couldn't hold the whole system in our heads, so we built layers to manage the complexity for us. Each layer made the next person's job easier. That's how you end up with frameworks on top of frameworks, wrapper libraries, thousands of lines of glue code.

AI doesn't have the same limitation. It can hold the whole system in context and just write the code. It doesn't need an intermediate framework to stay organized. It just needs a spec and a foundation to build on.

It's not clear yet which abstractions are truly foundational and which ones were just crutches for human cognition. That line is going to shift a lot over the next few years. But vinext is a data point. We took an API contract, a build tool, and an AI model, and the AI wrote everything in between. No intermediate framework needed. We think this pattern will repeat across a lot of software. The layers we've built up over the years aren't all going to make it.

Acknowledgments

Thanks to the Vite team. Vite is the foundation this whole thing stands on. `@vitejs/plugin-rsc` is still early days, but it gave me RSC support without having to build that from scratch, which would have been a dealbreaker. The Vite maintainers were responsive and helpful as I pushed the plugin into territory it hadn't been tested in before.

We also want to acknowledge the Next.js team. They've spent years building a framework that raised the bar for what React development could look like. The fact that their API surface is so well-documented and their test suite so comprehensive is a big part of what made this project possible. vinext wouldn't exist without the standard they set.

Try it

vinext includes an Agent Skill that handles migration for you. It works with Claude Code, OpenCode, Cursor, Codex, and dozens of other AI coding tools. Install it, open your Next.js project, and tell the AI to migrate:

```
npx skills add cloudflare/vinext
```

Then open your Next.js project in any supported tool and say:

```
migrate this project to vinext
```

The skill handles compatibility checking, dependency installation, config generation, and dev server startup. It knows what vinext supports and will flag anything that needs manual attention.

Or if you prefer doing it by hand:

```
npx vinext init    # Migrate an existing Next.js project
npx vinext dev     # Start the dev server
npx vinext deploy  # Ship to Cloudflare Workers
```

The source is at github.com/cloudflare/vinext. Issues, PRs, and feedback are welcome.

Visit 1.1.1.1 from any device to get started with our free app that makes your Internet faster and safer.

To learn more about our mission to help build a better Internet, start here. If you're looking for a new career direction, check out our open positions.

Discuss on Hacker News

AI Cloudflare Workers Workers AI Developers Developer Platform JavaScript Open Source Performance

Follow on X

Cloudflare | @cloudflare

RELATED POSTS

March 24, 2026 2:00 PM

Sandboxing AI agents, 100x faster

We're introducing Dynamic Workers, which allow you to execute AI-generated code in secure, lightweight isolates. This approach is 100 times faster than traditional containers,

enabling millisecond startup times for AI agent sandboxing....

By Kenton Varda, Sunil Pai, Ketan Gupta

MCP, Workers AI, AI, Agents, Developer Platform, Developers

March 23, 2026 2:00 PM

Launching Cloudflare's Gen 13 servers: trading cache for cores for 2x edge compute performance

Cloudflare's Gen 13 servers double our compute throughput by rethinking the balance between cache and cores. Moving to high-core-count AMD EPYC ™ Turin CPUs, we traded large L3 cache for raw compute density. By running our new Rust-based FL2 stack, we completely mitigated the latency penalty to unlock twice the performance....

By Syona Sarma, JQ Lau, Jesse Brandeburg

Hardware, Performance, Infrastructure, Rust, AMD, Engineering

March 19, 2026 8:53 PM

Powering the agents: Workers AI now runs large models, starting with Kimi K2.5

Kimi K2.5 is now on Workers AI, helping you power agents entirely on Cloudflare's Developer Platform. Learn how we optimized our inference stack and reduced inference costs for internal agent use cases. ...

By Michelle Chen, Kevin Flansburg, Ashish Datta, Kevin Jain

Developer Platform, Developers, Workers AI, Cloudflare Workers, AI, Agents

March 11, 2026 2:05 PM

Slashing agent token costs by 98% with RFC 9457-compliant error responses

Cloudflare now returns RFC 9457-compliant structured Markdown and JSON error payloads to AI agents, replacing heavyweight HTML pages with machine-readable instructions. This reduces token usage by over 98%, turning brittle parsing into efficient control flow....

By Sam Marsh

AI, Developer Platform, Developers, WAF, Edge Computing