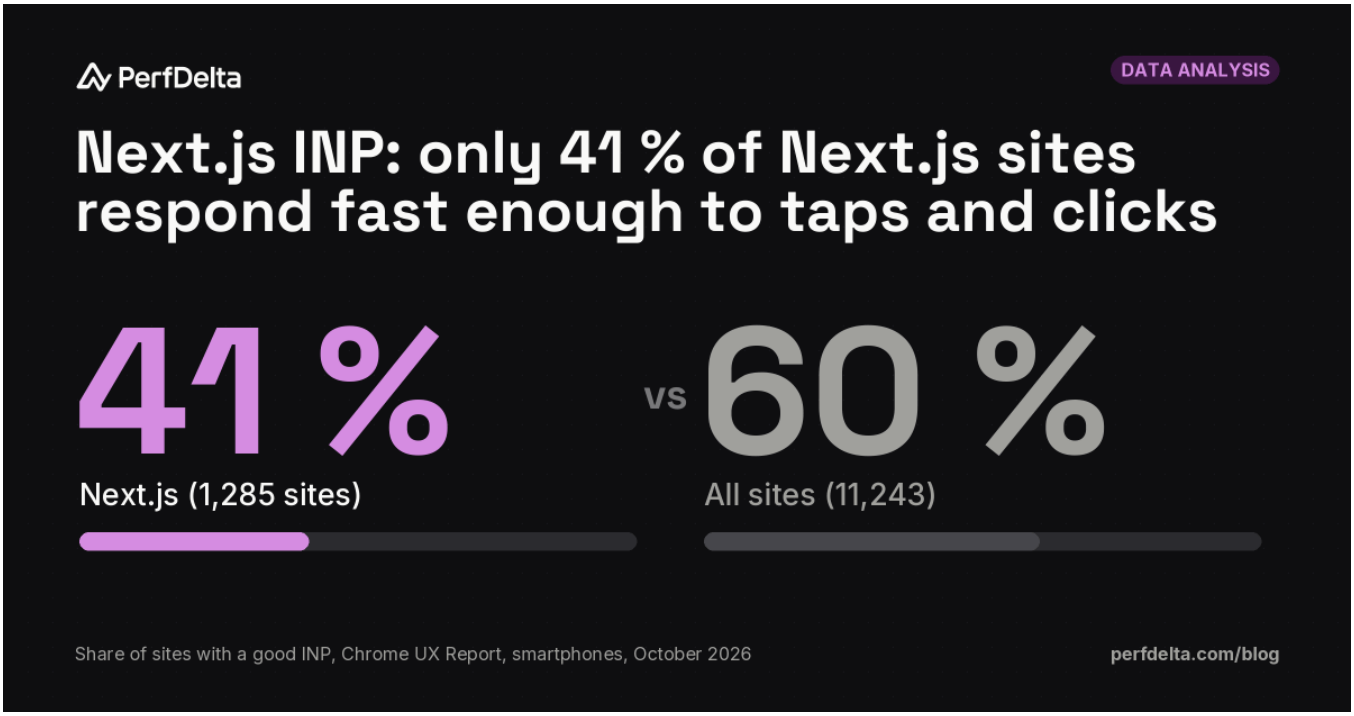
Next.js INP: Only Four in Ten Next.js Sites Have a Good Interaction to Next Paint

Published November 6, 2026 by Christian Schreiber



Next.js is chosen for speed: server rendering, image optimization and code splitting are built in. In our field data, Next.js websites load about as fast as the average website. Where they fall behind is responsiveness. Of the 1,285 Next.js websites with INP data, only 41.1 percent had a good Interaction to Next Paint (INP) on smartphones. Across all 11,243 websites in our sample with INP data, it was 60.2 percent.

INP is the Core Web Vital with the largest gap. 26.1 percent of the Next.js websites passed all three Core Web Vitals, compared with 36.4 percent of all websites. If every website had a good INP, the pass rates would be 45.9 and 50.1 percent: the gap would shrink from ten to four percentage points.

INP measures how quickly a page responds to taps and clicks

The Interaction to Next Paint records every tap, click and key press during a page view and measures the time until the browser shows the next frame. The reported value is close to the slowest interaction of the page view. Google rates 200 milliseconds or less as good and more than 500 milliseconds as poor. Our guide to the Interaction to Next Paint explains how the value is calculated.

A slow INP feels like a light switch with a delay: you press it, nothing happens, and you press again. The cause is usually JavaScript that occupies the main thread of the browser at the moment of the interaction or has to do a lot of work in response to it. On large pages, the style and layout work for the next frame adds to that.

Next.js websites fall behind on INP, not on LCP

The table compares the Next.js websites with all websites in our sample. The values are based on the 75th percentile of real smartphone visits from the Chrome UX Report. Shares refer to the websites with data for the respective metric:

Smartphone field data	Next.js	All websites
Good INP (200 ms or less)	41.1 %	60.2 %
Good LCP (2.5 s or less)	56.9 %	59.0 %
Good CLS (0.1 or less)	70.5 %	76.6 %
All three Core Web Vitals good	26.1 %	36.4 %
Good LCP and CLS, but poor INP	19.8 %	13.7 %
TTFB, median of the p75 values	875 ms	841 ms
bfcache hit rate, median	35 %	56 %

Loading speed (LCP) and server response (TTFB) are close to the average. One in five Next.js websites has good loading and layout values and fails the Core Web Vitals only because of INP. Layout stability is also somewhat weaker, and the bfcache hit rate is much lower, which we look at further below.

The pattern is not unique to Next.js. Nuxt, which also renders on the server and then hydrates the page with JavaScript, shows a similar value in our sample (227 websites with INP data, 46.7 percent good INP). That suggests the shared architecture plays a role. It is not the only factor: some shop platforms without such a framework also have low INP values in our data, so the type of website and the number of third-party scripts matter as well.

Hydration and large client bundles block the main thread

A server-rendered Next.js page looks ready as soon as the HTML arrives. It is only interactive once React has loaded the JavaScript for the client components and attached it to the HTML, a process called hydration. A tap during hydration often has to wait until the main thread is free. That is input delay, the first of the three parts of INP. Suspense boundaries let React hydrate large pages in smaller pieces and prioritize the part the visitor interacts with.

After hydration, three things typically make interactions slow:

- **Large client bundles.** Every component marked with `"use client"` and every module that file imports is shipped to the browser, parsed and executed. A `"use client"` high up in the tree therefore pulls a lot of code into the bundle. Server components passed into it as `children` stay on the server.
- **Expensive re-renders.** A state change in a context or a component near the root can re-render large parts of the page in a single long task before the browser can paint the response.
- **Third-party scripts.** Tag managers, consent tools, chat widgets and A/B testing share the same main thread and compete with React for it.

Six measures improve the INP of Next.js websites

1. **Keep client components small.** In the App Router, components are server components unless marked otherwise. Move `"use client"` to the interactive leaves of the tree, such as a button or a filter, instead of whole sections. The bundle analyzer shows what ends up in the browser: `next analyze` for Turbopack builds since Next.js 16.1, the `@next/bundle-analyzer` plugin for webpack builds (Optimizing package bundling, Next.js documentation).
2. **Mark non-urgent updates as transitions.** With `useTransition` or `useDeferredValue`, React renders an expensive update, such as filtering a long list, in interruptible steps on the main thread and handles new input first (useTransition, React documentation). This only helps with React rendering, not with slow synchronous code in the event handler itself.
3. **Give immediate feedback.** Show a pressed state, a spinner or a `loading.js` skeleton in the first frame after the interaction, and do the heavy work afterwards.
4. **Break up long tasks.** Split work that runs longer than 50 milliseconds and yield to the browser in between, for example with `scheduler.yield()`, available in Chrome since version 129, with `setTimeout` as a fallback in other browsers (Optimize long tasks, web.dev, December 2024).
5. **Reduce and schedule third-party scripts.** Remove tags nobody uses. With `next/script` and the strategy `lazyOnload`, scripts that are not needed for the first interaction load when the browser is idle. Their work still runs on the main thread later, so this only shifts the load and does not replace removing tags.
6. **Render rarely used components only when needed.** Split dialogs, carousels and maps out with `next/dynamic` and render them only when they are opened. A dynamically imported component that is rendered on page load is still loaded and hydrated immediately.

The guide to Next.js Core Web Vitals also covers images, fonts and rendering strategies.

Dynamic Next.js pages often miss the bfcache

The second finding concerns the back button. On the median Next.js website, only 35 percent of back and forward navigations were restored from the bfcache. On 42.0 percent of the 1,023 Next.js websites with enough history navigations, the rate was below 25 percent. Across all websites, that applied to 28.8 percent.

A likely reason is a default of the framework. According to the Next.js documentation, dynamically rendered pages send the header `Cache-Control: private, no-cache, no-store, max-age=0, must-revalidate` (Self-Hosting, Next.js documentation). According to Chrome, `no-store` on the HTML was the largest single reason for missed bfcache restores (Enabling bfcache for Cache-Control: no-store, Chrome for Developers, September 2025). Chrome now allows such pages into the bfcache only under certain conditions. Pages without personal content can often be rendered statically or revalidated instead. Our analysis of the bfcache hit rate explains the other blockers.

Single-page navigations change the picture only slightly

A Next.js app switches pages on the client. For the Chrome UX Report, a page view therefore often spans several route changes, and INP covers more interactions than on a classic website. Chrome's experimental measurement of soft navigations splits these page views. For the median Next.js website, it changed INP by only four percent, so this effect does not explain the gap.

Back navigations within the app also run on the client and do not appear as back navigations in the Chrome UX Report. The bfcache rate above therefore describes full page loads, for example when a visitor returns from another website.

Check the INP of your Next.js website

The page speed test shows the INP of your website from real smartphone visits, next to your competitors and week by week. Up to five websites can be compared directly in the Core Web Vitals benchmark. How Next.js websites perform overall is shown in the Next.js speed statistics.

Method and limits of this analysis

- **Data:** 11,396 smartphone records from the Chrome UX Report, each origin counted once, data period up to October 31, 2026, retrieved by PerfDelta. Each record covers a rolling 28-day window of real Chrome visits. Shares of good values refer to the origins with data for that metric, the pass rate to the origins with all three values.
- **Platform:** detected from the technology stack of the homepage in our latest measurement. 1,293 origins were identified as Next.js. Sites where a CMS was detected, including a headless CMS with a Next.js front end, are counted under that CMS. The field data covers the whole origin.
- **bfcache:** only origins where back and forward navigations made up at least two percent of page views (1,023 Next.js websites, 8,868 in total).

- **Soft navigations:** ratio of the experimental INP with soft navigations to the regular INP, median over 1,282 Next.js websites.
- **Correlation, not cause:** Next.js websites differ from the rest of the sample in more than the framework. Many are shops, web apps or marketing sites with numerous third-party scripts. The data shows how Next.js websites perform in practice, not how fast Next.js can be.
- **Sample:** the websites come from PerfDelta tests and studies and lean towards larger brands. The results are not representative of the entire web. More figures are in the website speed statistics.