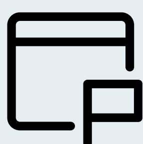
FIX CORE WEB VITALS: NEXT.JS OPTIMIZATION GUIDE FOR HIGH-TRAFFIC WEBSITES

NEXT.JS OPTIMIZATION FOR HIGH-TRAFFIC WEBSITES

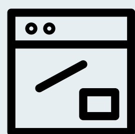
At Pagepro, we specialize in optimizing Next.js for high-traffic, large-scale applications. Our hands-on experience with diverse projects has equipped us with the technical expertise to enhance speed, scalability, and stability—factors critical for today's digital ecosystems.

Through strategic optimization techniques, we've consistently delivered results for high-demand platforms across industries:



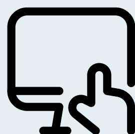
65% FASTER LOAD TIMES

Interactive Media Platform
Optimized asset delivery and implemented code splitting, resulting in a 65% reduction in load times, from 4.2s to 1.5s, significantly improving user retention.



2.5S LCP REDUCTION

E-Learning Platform for Artists
Reduced Largest Contentful Paint (LCP) from 3.8s to 1.3s by optimizing image rendering and leveraging server-side caching boosting SEO and conversion rates.



370MS INP DECREASE

Educational Resources Portal
Improved responsiveness by optimizing JavaScript execution, deferring non-critical scripts, reducing INP from 550ms to 180ms and boosting user engagement rate.

Each project challenged us to push Next.js performance to its limits, and in this playbook, we share the insights and techniques that have consistently delivered measurable results for high-traffic applications.

OUR SERVICES

STEP 1: PRIORITIZE HIGH-IMPACT, LOW-EFFORT OPTIMIZATIONS

The temptation to dive into every optimization technique can be strong, but focusing on high-impact, low-effort changes first is essential for efficient results. Start by running a complete performance audit using tools like Lighthouse and WebPageTest. Identify bottlenecks, especially those impacting Core Web Vitals metrics: Largest Contentful Paint (LCP) and Interaction to Next Paint (INP), and Cumulative Layout Shift (CLS).

ACTION PLAN

1. Conduct a Performance Audit: Use Lighthouse, WebPageTest, or Chrome DevTools to identify the top performance bottlenecks on your site.
2. Create a Prioritized List: Categorize optimizations based on impact and effort. Focus on changes that maximize improvements with minimal resources.
3. Remove Unused JavaScript: Use Chrome DevTools' Coverage tab to detect and eliminate any unused code, such as third-party scripts that don't impact critical functionality.
4. Apply CSS Content Visibility: Use the content-visibility CSS property for off-screen content. This defers rendering for content below the fold, especially beneficial on long pages. However, use progressive enhancement since not all browsers support this feature.
5. Iterative Measurement: After implementing each improvement, re-run performance metrics to gauge impact. Refine your approach with every iteration.

EXPECTED WIN
Immediate reduction in page weight, improved load speeds, and optimized rendering for above-the-fold content.

Suggested Solution	Impact on the CWV	Time requirement	Comments / Next Steps
Removal of dynamic chunks preloading (can possibly cause CLS?)	High	Low	Implement
Investigate the usage of content-visibility https://web.dev/articles/content-visibility	High	Low	Implement
Try to reduce or delay code being executed on load of a clinical page	High	Medium	Do the investigation and estimate how much time will be required for this
Prepare a dedicated "indian" version with limited functionalities	High	Medium	turn off ADS on global locale first - we should try that but after the 6th
Disable the ADS on global locale	High	Low	We need to test it for a week on global locale/ there is a risk of UK users using global page instead of UK and wont see ads
See if there's any use for web workers (execute code off of the main thread)	High	High	Do the investigation and estimate how much time will be required for this
Remove all css variables	High	High	Do the investigation and estimate how much time will be required for this
Remove unused sections from the homepage (unused = rarely visited) (global locale only)	Medium	Low	related to the no5 - we should try that but after the 6th
Use Cloudflare Zaraz for GTM/GA/Google Ads/FB etc	Medium	Low	We may get rid of cloudflare
Check if DNS configuration is fully compliant with Vercel recomendations	Medium	Low	We may get rid of cloudflare
Optimize Header component (it's being re-rendered multiple times and has an expensive useMemo)	Medium	Low	
Consider removing ReactSVG (bigger HTML, but less reflows)	Medium	Medium	
Prepare a dedicated version with limited functionalities for Africa and Asia countries	Medium	Medium	Similar to no 5 - we should be able to test hypothesis with ads on this one
Go through each section and try reducint its bundle and optimizing it (starting with the big ones like registration, login and listing)	Medium	Medium	
The homepage has way more forced layout recalculations, we can investigate and fix them	Medium	Medium	
Go for Vercel enteprise plan (projects optimization, more regions for functions)	Medium	Medium	
Remove fetching on client side where possible (page updates count, popular, latest)	Medium	High	There is a ticket for that already
Reduce component re-renders (for example UserContextProvider, EditorialDetails, ClinicalReference)	Low	Low	
Get rid of cloudflare DNS (hypothesis with Vercel DNS)	Low	Low	This is something we will go in the future
Some of css variables can maybe be removed or set only when needed (search-bar-height, hcp-container-visible-height, navigation-top)	Low	Medium	We are implementing 8 and this one is a part of it.
Replace some of the libraries with ones with smaller bundle size/lower page speed impact (select, datepicker, date-fns)	Low	Medium	Implement
Update some of the libraries that has newer major version and may have positive impact on performance	Low	Medium	Implement

“The above task list represents a prioritized approach to improving Core Web Vitals (CWV). Each task targets specific performance issues, such as removing unused code, disabling ads in certain locales, or creating region-specific versions with limited functionalities.
Tasks are marked by their expected impact on CWV, focusing on metrics like Largest Contentful Paint (LCP) and Interaction to Next Paint (INP). High-impact tasks are prioritized for faster, more visible improvements.
Each task’s estimated effort—low, medium, or high—helps identify quick wins with high returns. Low-effort, high-impact tasks are tackled first, ensuring efficient resource use.”

Michał Widga
CDO at Pagepro

STEP 2: OPTIMIZE THIRD-PARTY SCRIPTS AND ADS

Third-party integrations, can be a significant drain on performance. Start by auditing each third-party script and determining its value to the user and the bottom line. For those that stay, consider optimization techniques to reduce their impact.

ACTION PLAN

1. Audit Third-Party Scripts: Identify all external scripts with Chrome DevTools or specialized analyzers. Document each script's purpose and frequency of use.
2. Eliminate Redundant Scripts: Review if any script is truly necessary; eliminate those that are non-essential.
3. Optimize Loading Strategies:
   a. Use NextScript with the Correct Strategy: When adding scripts in Next.js, use the built-in NextScript component with an appropriate strategy (lazyOnload, afterInteractive, or beforeInteractive) to ensure third-party scripts don't block rendering and load efficiently.
   b. Lazy load non-critical scripts to optimize first paint performance.
   c. Implement preconnect for third-party domains that need quicker initial connection times.
   d. Replace resource-heavy widgets, like help chats, with lightweight placeholder icons. Load the full widget only when the user interacts with the placeholder, such as clicking on it. This approach keeps the page lightweight while maintaining user functionality.
4. Ad-Specific Optimizations:
   a. Consider deferring ads for non-revenue-critical markets or user segments.
   b. Implement lazy-loading for ads below the fold to reduce initial load.
   c. Collaborate with ad partners to secure lighter ad units that don't compromise speed.
5. Explore Edge Solutions: Offload third-party scripts with tools like Cloudflare Workers or Vercel Edge Functions, allowing cached delivery closer to users for a faster experience.

EXPECTED WIN
Shrinked load times with reduced render-blocking, faster time-to-interactive, and lower resource consumption on the client side.

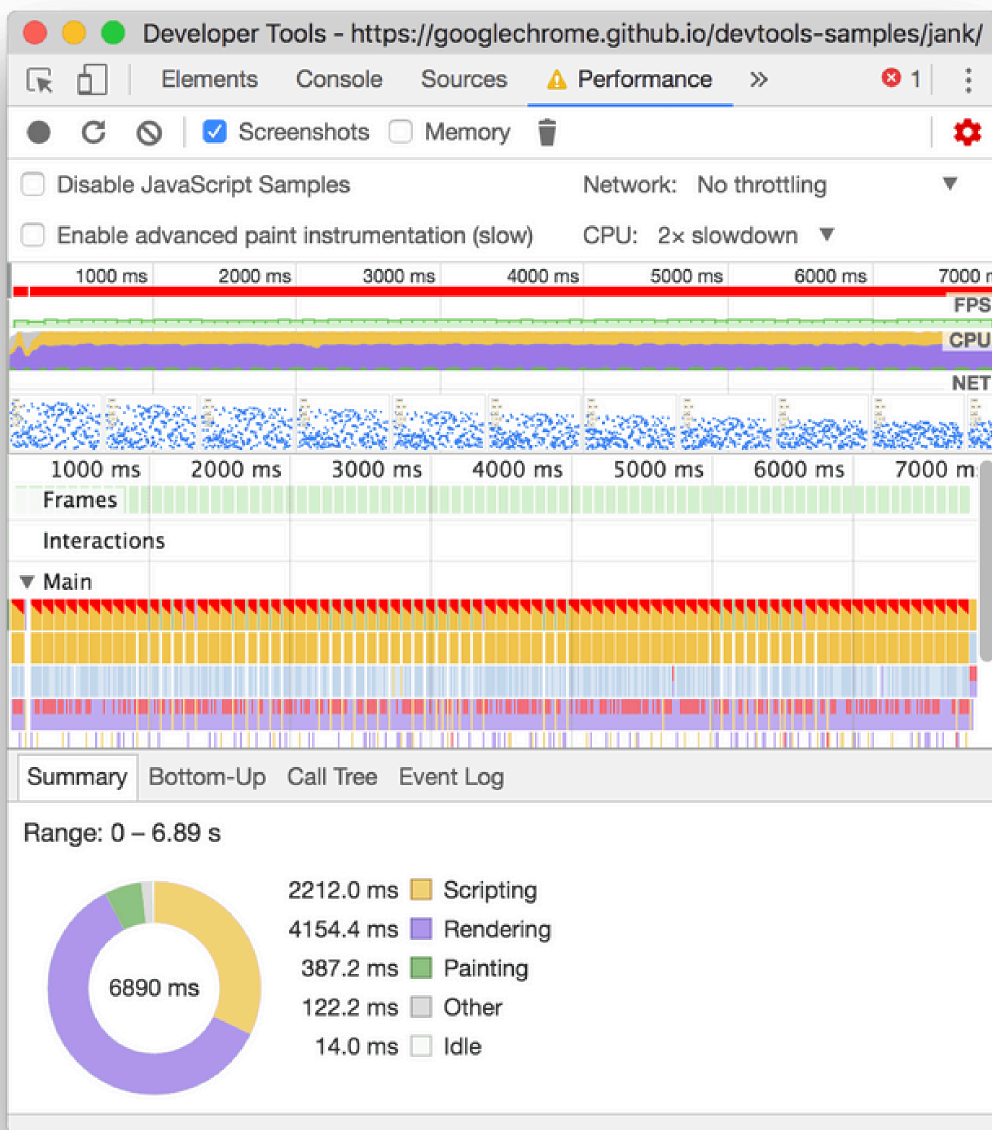
STEP 3: FOCUS ON CRITICAL PAGES

For high-traffic sites, focusing optimizations on the most-visited pages can yield significant returns. Not every page needs full-scale optimization efforts, but business-critical pages do. Use analytics to identify these pages, then dive deep with Chrome DevTools Performance Tab to find areas for improvement.

ACTION PLAN

1. Identify Business-Critical Pages: Use Google Analytics to track your top-performing pages in terms of traffic and user engagement.
2. Start with a Lighthouse Audit: Conduct a Lighthouse audit to identify easy wins, such as unoptimized images or render-blocking resources. Use these insights as a starting point to guide further analysis in Chrome DevTools.
3. Conduct a Detailed Analysis: For each high-traffic page, use Chrome DevTools' Performance tab to create profiles and analyze load times, TTFB, and render performance.
4. Refine Data Fetching: In Next.js, the approach to data fetching depends on the router being used:
    a. Pages Router: Utilize getServerSideProps or getStaticProps to fetch data server-side, reducing client-side loading times and improving initial render performance.
    b. App Router: Avoid using the "use client" directive unless necessary, as it forces client-side rendering.
5. Optimize Caching for Data: For data frequently accessed across sessions, implement smart caching strategies to reduce unnecessary re-fetching.
6. Streamline Component Rendering: Use React Profiler to detect unnecessary re-renders and apply React hooks like React.memo, useMemo, and useCallback to ensure optimized component behavior.
7. Create Simplified Versions: For users on lower-powered devices or slower networks, create stripped-down versions of key pages by removing non-critical features and implementing server-side rendering (SSR) where possible.

EXPECTED WIN
Reduced load times and faster interaction on critical pages, leading to better retention and lower bounce rates.



"At first, I tried to convince myself that there was nothing useful in the Performance Tab, but I was simply intimidated by the data I didn't understand at the time. Now, I see how much valuable information it provides about your app,
and I use it every day. I highly recommend it to everyone."

Jakub Dakowicz
CTO at Pagepro

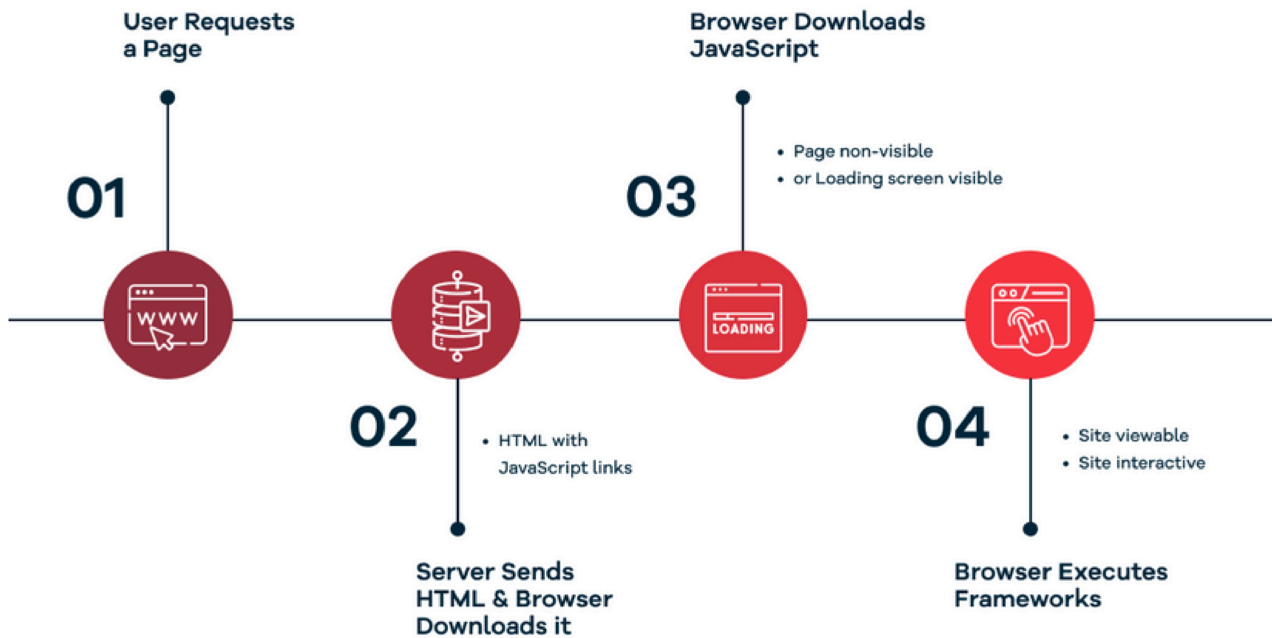
STEP 4: USE NEXT.JS'S BUILT-IN PERFORMANCE FEATURES

One of the advantages of using Next.js is its out-of-the-box support for performance enhancements. Developers can use these built-in features to make the most of Next.js's native power. Code-splitting, image optimization, and font loading optimization are just a few ways Next.js natively supports faster websites without extensive configuration.

ACTION PLAN

1. Code Splitting via Dynamic Imports: Use Next.js's dynamic import function to code-split and minimize the initial bundle. Apply code-splitting strategically on components not needed right away.
2. Image Optimization: Use the Next.js Image component for automatic image optimization, lazy loading, and serving responsive images tailored to each device.
3. Font Optimization: Use Next.js's font optimization to preload fonts and reduce layout shifts. Implement font loading directly in _app.js to optimize site-wide font management.
4. Optimize Rendering Strategies: Choose server-side rendering (SSR) for pages requiring real-time data, static site generation (SSG) for less dynamic pages, and incremental static regeneration (ISR) for pages needing periodic updates.

EXPECTED WIN
Smoother initial page loading, reduced layout shifts, and enhanced Core Web Vitals scores.



“Remember, keep your code on the client to a minimum. Every extra line impacts load times, increases bundle size, and can slow down the user experience. Too much client-side code overwhelms the browser, creating performance bottlenecks and frustrating users.”

Jakub Dakowicz
CTO at Pagepro

STEP 5: OPTIMIZE IMAGES AND STATIC ASSETS

Static assets like images and CSS contribute heavily to page weight. Rather than defaulting to standard image and asset loading practices, focus on fine-tuning asset delivery to match user needs and device capabilities.

ACTION PLAN

1. Image Optimization: Use Next.js Image component for resizing, format conversion (to WebP/AVIF), and lazy loading. This minimizes image-related load time impacts.
2. Serve Responsive Images: Set up responsive images to serve differently sized images based on device screen sizes.
3. Modern File Formats: Use WebP or AVIF where supported to reduce file sizes without quality loss.
4. Minify CSS and JavaScript: Reduce the size of your static assets with minification tools to remove whitespace and comments.
5. Cache Static Assets: Use long-term caching strategies on static assets to enable quick retrieval for returning users.
6. Lazy Load Off-Screen Content: Implement loading="lazy" for below-the-fold images and iframes to improve initial page load speed.

EXPECTED WIN

Reduced page weight, faster load times, and improved visual stability.

STEP 6: OPTIMIZE HOSTING AND CDN CONFIGURATION

Your site's infrastructure and CDN configuration are foundational to performance. For global reach, optimizing these settings means your site's content is delivered quickly, regardless of user location.

ACTION PLAN

1. Analyze Current Hosting Setup: Use tools like Pingdom and GTmetrix to assess hosting performance across regions.
2. Optimize CDN Setup: Ensure your CDN is correctly configured for Next.js, and consider using Vercel's Edge Network optimized for Next.js applications.
3. Enable Edge Caching: Use serverless or edge computing platforms like Vercel Edge Functions or Cloudflare Workers to distribute computation closer to your users.
4. Resolve CDN Conflicts: If using multiple services (like Vercel and Cloudflare), make sure they're compatible to avoid performance conflicts.
5. Consolidate Services Where Possible: Minimize complexity by reducing the number of services used to improve coordination between hosting and CDN.

EXPECTED WIN
Faster content delivery across global regions, particularly for international users.

STEP 7: REDUCE YOUR BUNDLE SIZE TO MINIMUM

By default NextJS splits your bundle to chunks - you have a chunk for each page you created. There is also an app chunk that is a big one - in most scenarios it contains your layout and things shared across all of your pages. In most scenarios you can reduce it.

ACTION PLAN

1. Add bundle analyser to your project so you can anyone the modules used in each chunk.
2. Focus on reducing the app bundle size as it's shared across all of the pages.
3. Use direct imports in utility libraries to eliminate risk of incorrect tree shaking.
4. Lazy load all of the components that are visible on demand.
5. Extract shared logic to separate files - it will load less data into the chunks.

EXPECTED WIN
Faster initial load times and improved Core Web Vitals by reducing app bundle size, resulting in smoother navigation and enhanced user experience across all pages.

STEP 8: UPDATE DEPENDENCIES REGULARLY

Outdated dependencies often include inefficient code that can impact performance. Regular updates keep your tech stack lean and performant, minimizing the risk of compatibility issues or latent bugs that hurt load times.

ACTION PLAN

1. Schedule Routine Updates: Set up a monthly or quarterly task to review dependencies and update as needed. Focus on maintaining the core libraries like Next.js, React.
2. Use npm-check-updates: Identify outdated packages and prioritize updates that impact performance and security.
3. Testing and Monitoring: Use automated tests to validate performance after updates. Keep a close eye on metrics for any regressions.
4. Monitor Core Dependencies Closely: Pay extra attention to updates for critical packages like React, Next.js, and performance-related libraries.

EXPECTED WIN

Smoother updates with fewer disruptions, minimizing risks of performance regression.

STEP 9: IMPLEMENT CONTINUOUS MONITORING AND PERFORMANCE TRACKING

To maintain high performance, continuous monitoring is critical. By tracking metrics over time, you can catch and address issues early, keeping performance consistent even as you scale.

ACTION PLAN

1. Set Up Real User Monitoring (RUM): Implement Vercel Analytics or Google Analytics to get real-time data on user experiences.
2. Establish Performance Budgets: Define acceptable thresholds for metrics like LCP, CLS, and INP, and monitor regularly with Lighthouse CI.
3. Automate Performance Alerts: Set alerts for performance degradations to promptly address issues.
4. Review Performance Data: Conduct weekly or bi-weekly performance reviews, and implement fixes as part of a continuous improvement cycle.

EXPECTED WIN
Early detection and resolution of performance issues to maintain consistent user experience.

STEP 10: MARKET-SPECIFIC OPTIMIZATIONS FOR GLOBAL REACH

A single optimization approach rarely works worldwide. Global companies must account for differences in connectivity and device power across regions. Use analytics to understand how different regions perform and tailor site content and optimizations accordingly.

ACTION PLAN

1. Analyze Regional Performance Data: Use analytics to understand performance metrics for specific geographic regions.
2. Localize CDN Endpoints: Configure CDN endpoints close to high-traffic regions.
3. Develop Market-Specific Versions: Create lighter, more efficient versions of your site for markets with slower internet or older devices.
4. Implement Geolocation-Based Routing: Use geolocation to route users to region-specific versions of your site for faster load times.

EXPECTED WIN
Improved load times and user experience tailored to regional conditions.

CHECKLIST

1. Prioritize High-Impact, Low-Effort Optimizations

- [ ] Conduct a performance audit (e.g., Lighthouse, WebPageTest)
- [ ] Remove unused JavaScript (e.g., unnecessary third-party scripts)
- [ ] Defer rendering of off-screen content using content-visibility
- [ ] Create and refine a prioritized optimization list

2. Optimize Third-Party Scripts and Ads

- [ ] Audit and evaluate third-party scripts (Chrome DevTools)
- [ ] Optimize loading with async or defer
- [ ] Lazy load ads and non-essential third-party scripts
- [ ] Implement resource hints like preconnect for critical domains
- [ ] Explore server-side script handling (e.g., Cloudflare Workers, Vercel Edge Functions)

3. Investigate and Optimize Critical Pages

- [ ] Identify high-traffic, business-critical pages
- [ ] Analyze performance metrics (TTFB, INP) for each critical page
- [ ] Optimize data fetching (Next.js's getServerSideProps, getStaticProps)
- [ ] Implement caching and prefetching for frequently accessed data
- [ ] Optimize rendering with React.memo, useMemo, useCallback
- [ ] Create simplified versions for slower connections or devices

4. Leverage Next.js Built-in Optimizations

- [ ] Implement code-splitting with dynamic imports
- [ ] Use Next.js Image component for automatic image optimization
- [ ] Leverage Next.js font optimization to reduce layout shifts
- [ ] Optimize rendering strategies (SSR, SSG, ISR)
- [ ] Configure API Routes efficiently with timeouts and error handling

5. Optimize Images and Static Assets

- [ ] Use Next.js Image component for responsive, lazy-loaded images
- [ ] Adopt modern image formats (WebP, AVIF)
- [ ] Minify CSS and JavaScript files
- [ ] Implement caching strategy for static assets
- [ ] Lazy load below-the-fold content (e.g., loading="lazy")

6. Evaluate Infrastructure and CDN Options

- ☐ Assess current hosting and CDN setup
- ☐ Optimize CDN configuration for Next.js, such as Vercel's Edge Network
- ☐ Implement serverless or edge computing to move computations closer to users
- ☐ Ensure proper configuration if using multiple services (e.g., Vercel and Cloudflare)

7. Reduce Bundle Size to Minimum

- ☐ Add a bundle analyzer to identify modules in each chunk.
- ☐ Reduce shared dependencies to minimize the app bundle size.
- ☐ Use direct imports to improve tree-shaking in utility libraries.
- ☐ Lazy load non-critical components to reduce initial bundle size.
- ☐ Extract shared logic into separate files to optimize chunking.

7. Regularly Update Dependencies

- ☐ Establish a schedule for dependency updates
- ☐ Prioritize updates with known performance or security improvements
- ☐ Use automated testing to detect issues after updates
- ☐ Monitor performance post-update to avoid regressions

8. Implement Continuous Performance Monitoring

- ☐ Set up monitoring tools (e.g., Vercel Analytics, Google Analytics)
- ☐ Establish performance budgets for key metrics (INP, LCP, CLS)
- ☐ Schedule regular reviews of app performance data (weekly/bi-weekly)
- ☐ Automate alerts for significant performance regressions

9. Apply Market-Specific Optimizations

- ☐ Analyze geographic performance data for different regions
- ☐ Develop optimized versions for specific markets (e.g., lighter layouts)
- ☐ Implement intelligent routing using geolocation or user-agent detection
- ☐ Optimize CDN endpoints for local conditions

BONUS: 5 LESSONS WE'VE LEARNED OPTIMIZING NEXT.JS FOR LARGE PROJECTS

LESSON 1: USE THE POWER OF CHROME'S PERFORMANCE TAB

The Chrome DevTools Performance Tab is our go-to tool. Early on, we discovered that simply removing a heavy carousel component shaved off 1.8 seconds of load time. Don't skip this powerful diagnostic tool—it's essential for spotting hidden inefficiencies in your codebase.

LESSON 2: IMPLEMENT CONTINUOUS PERFORMANCE CHECKS

When every team member, from developers to designers, prioritizes performance, optimizations happen naturally. By embedding performance checks throughout the dev cycle, we reduced last-minute "firefighting" and kept sites running smoothly.

LESSON 3: REDUCE CODE ON THE CLIENT TO A MINIMUM

Heavy client-side JavaScript burdens users, particularly on mobile devices. We've learned that moving logic to the server or using Next.js's lazy loading capabilities has improved responsiveness and lightened load times.

LESSON 4: BUILD A PERFORMANCE-FIRST TEAM MINDSET

Regular workshops ensure our team knows the latest performance techniques. Sharing wins and lessons encourages everyone to contribute to a faster, more responsive site, keeping performance top of mind.

LESSON 5: LIMIT DATA TRANSFER TO THE MINIMUM

Over-fetching is a silent performance killer. By using GraphQL to control data transfer, we cut data load times by over 60% on a recent project, enhancing both speed and user experience.

PARTNER WITH PAGEPRO FOR NEXT-LEVEL PERFORMANCE OPTIMIZATION

We have extensive experience in building high-performing websites that drive real results. We know that optimizing for speed and stability translates to better engagement, improved SEO, and measurable benefits for your business.

How We Can Support Your Next.js Application:

TAILORED CONSULTATION AND STRATEGIC PLANNING:
We develop a customized, actionable plan focused on impactful performance improvements.

IMPLEMENTATION FOR RESULTS:
We don’t just plan - we execute. Our team implements each prioritized recommendation, ensuring that every change enhances speed, stability, and scalability.

COMPREHENSIVE PERFORMANCE REVIEW:
After implementation, we provide a detailed performance report, benchmarking improvements and outlining maintenance steps to keep your site running at peak performance.

BOOK A MEETING