Thanks! I’ll compile a complete, updated version of the performance guide with 15 total patterns covering frontend, backend/server, and developer experience optimizations. I’ll rewrite all examples using modern TypeScript conventions and include both bad and good snippets to illustrate best practices. I’ll share the full updated version with you shortly.

**Next.js 14+ Performance Best Practices**

Next.js 14 (App Router) introduces powerful new features (React Server Components, streaming, Server Actions, Turbopack, etc.) that help make apps fast by default. Below are **15 best-practice patterns** for frontend performance, server architecture, and developer experience. Each pattern includes a brief explanation (often quoting the official docs) and both “Bad” and “Good” TypeScript code examples illustrating the practice.

**1. Optimize Images with next/image**

Use Next.js’s <Image> component for all images. It automatically optimizes images (lazy-loading, responsive sizes, modern formats) and prevents layout shifts. For example, Next.js notes the Image component "optimizes images, prevents layout shift, and serves them in modern formats like WebP".

// Bad: Using a plain <img> element (no optimization)

export default function Logo(): JSX.Element {

return <img src="/logo.png" alt="Logo" />;

}

import Image from 'next/image';

// Good: Using next/image with width/height for optimization

export default function Logo(): JSX.Element {

return (

<Image

src="/logo.png"

alt="Logo"

width={100}

height={100}

/>

);

}

**2. Optimize Fonts with next/font**

Use Next.js’s built-in font optimization (e.g. next/font/google or the local Font Module). The framework “automatically hosts your font files with other static assets, removes external network requests, and reduces layout shift”. This inlines critical font CSS and preload links under the hood, improving LCP and avoiding FOUT.

// Bad: Importing fonts via CSS or external link (extra network requests)

export default function Page(): JSX.Element {

return <p style={{ fontFamily: 'Open Sans' }}>Hello World</p>;

}

import { Inter } from 'next/font/google';

// Good: Using next/font for automatic optimization

const inter = Inter({ subsets: ['latin'] });

export default function Page(): JSX.Element {

return <p className={inter.className}>Hello World</p>;

}

**3. Lazy-Load Third-Party Scripts with <Script>**

For external scripts (analytics, ads, tag managers), use Next.js’s <Script> component. It defers loading to avoid blocking the main thread. The docs say <Script> “automatically defers scripts and prevents them from blocking the main thread”. This way, third-party code doesn’t slow down first contentful paint.

// Bad: Directly placing a <script> tag (blocking)

export default function Home(): JSX.Element {

return (

<>

<h1>Welcome</h1>

<script src="https://example.com/analytics.js" />

</>

);

}

import Script from 'next/script';

// Good: Using next/script with lazy loading strategy

export default function Home(): JSX.Element {

return (

<>

<h1>Welcome</h1>

<Script

src="https://example.com/analytics.js"

strategy="lazyOnload"

/>

</>

);

}

**4. Code Splitting with Dynamic Imports**

Next.js automatically splits code by route, but you can further split large components using dynamic imports. The framework documentation notes it “automatically code-splits your application code by pages”. To reduce the initial bundle, use next/dynamic for heavy components or libraries.

// Bad: Importing a heavy component upfront

import HeavyChart from '@/components/HeavyChart';

export default function Dashboard(): JSX.Element {

return <HeavyChart />;

}

import dynamic from 'next/dynamic';

// Good: Lazy-loading the heavy component

const HeavyChart = dynamic(() => import('@/components/HeavyChart'), {

loading: () => <p>Loading chart...</p>,

});

export default function Dashboard(): JSX.Element {

return <HeavyChart />;

}

**5. Leverage React Server Components (Minimize Client JS)**

Next.js’s App Router renders by default as React Server Components, sending zero JS to the client unless use client is specified. This **minimizes client-side JavaScript** and speeds up hydration. As noted in the docs, Next.js 13+ “supports streaming SSR, Server Components, and Suspense” so that “by pushing more code to the server, we can reduce the client-side JavaScript and speed up hydration”. Only mark components with use client when they truly need client-side interactivity.

// Bad: Using a client component for static data (no need for client JS)

'use client';

export default function Dashboard(): JSX.Element {

const data = fetchData(); // Synchronous fetch (wrong in client)

return <div>{data.title}</div>;

}

// Good: Async server component fetches data on the server

export default async function Dashboard(): Promise<JSX.Element> {

const data = await fetchData();

return <div>{data.title}</div>;

}

**6. Use Static Generation and ISR (Incremental Static Regeneration)**

Where possible, use static generation or ISR to cache pages at build time. In App Router, this means adding export const revalidate = N or using fetch() with next: { revalidate: N }. This lets Next.js “update static pages after they’ve been built” without a full rebuild. Cached pages are then served globally from the CDN for fastest load times, only revalidating in the background as needed.

export default async function Page(): Promise<JSX.Element> {

// Bad: Forcing fresh fetch on every request (no caching)

const res = await fetch('https://api.example.com/data', { cache: 'no-store' });

const data = await res.json();

return <div>{data.name}</div>;

}

export const revalidate = 60; // Revalidate at most once per minute

export default async function Page(): Promise<JSX.Element> {

// Good: Fetch with revalidation (stale-while-revalidate caching)

const res = await fetch('https://api.example.com/data', { next: { revalidate: 60 } });

const data = await res.json();

return <div>{data.name}</div>;

}

**7. Data Fetching and Caching Strategies**

Ensure your data fetching uses caching and the new Route Handlers as appropriate. Next.js recommends using Route Handlers (App Router route.ts) for backend logic and “verify whether your data requests are being cached”. In client components, prefer caching libraries like SWR or React Query over un-cached fetch loops.

'use client';

import { useEffect, useState } from 'react';

function Profile(): JSX.Element {

// Bad: Fetching on every mount, no cache

const [user, setUser] = useState<{name:string} | null>(null);

useEffect(() => {

fetch('/api/user')

.then(res => res.json())

.then(data => setUser(data));

}, []);

return <div>{user?.name}</div>;

}

import useSWR from 'swr';

function Profile(): JSX.Element {

// Good: Using SWR for caching data on the client

const { data: user } = useSWR('/api/user', url => fetch(url).then(res => res.json()));

return <div>{user?.name}</div>;

}

**8. Use Edge Middleware for Personalization**

Next.js Middleware runs at the CDN edge, enabling fast, personalized routing without hitting your origin. By “server[ing] rich personalized experiences entirely from the edge (CDN)” you get huge speed-ups. For example, you can rewrite visitors transparently to /marketing/women or /marketing/men based on a cookie or header, while still serving pages from the CDN.

// Bad: Personalization with getServerSideProps (slower, server-only)

export default async function Page(): Promise<JSX.Element> {

const cookie = /\* read cookie on server \*/;

const page = cookie === 'male' ? '/marketing/men' : '/marketing/women';

const res = await fetch(page);

const html = await res.text();

return <div dangerouslySetInnerHTML={{ \_\_html: html }} />;

}

import { NextResponse } from 'next/server';

import type { NextRequest } from 'next/server';

// Good: Middleware routes at the edge before CDN cache

export function middleware(req: NextRequest) {

const gender = req.cookies.get('gender')?.value;

const url = req.nextUrl.clone();

url.pathname = gender === 'male' ? '/marketing/men' : '/marketing/women';

return NextResponse.rewrite(url);

}

**9. Use Server Actions for Form/Data Mutations**

Next.js 14+ introduces Server Actions (server functions) for handling form submissions and mutations without needing an API route. A function marked with 'use server' can be called directly from a form’s action (or button’s formAction) prop. This integrates with Next’s caching and requires only one round-trip. It’s more efficient than client-side fetch to /api.

// Bad: Client handles form and POSTs to an API route

export default function CreatePost(): JSX.Element {

const handleSubmit = async (event: React.FormEvent) => {

event.preventDefault();

const form = event.currentTarget as HTMLFormElement;

const formData = new FormData(form);

await fetch('/api/posts', { method: 'POST', body: formData });

};

return (

<form onSubmit={handleSubmit}>

<input name="title" />

<button type="submit">Create</button>

</form>

);

}

// Good: Using a Server Action directly in the form

// app/actions.ts

'use server';

export async function createPost(formData: FormData) {

const title = formData.get('title') as string;

// ... (e.g. save to database)

}

// app/page.tsx

import { createPost } from './actions';

export default function Page(): JSX.Element {

return (

<form action={createPost}>

<input name="title" />

<button type="submit">Create</button>

</form>

);

}

**10. Optimize React 18 Concurrent Updates**

Use React 18’s concurrent features for heavy UI updates. For example, wrap large state updates in startTransition (via useTransition) so they don’t block user interactions. This keeps the UI responsive: React can pause or defer low-priority renders. (No explicit citation for this widely known practice, but it complements Next.js’s concurrency.)

import { useState } from 'react';

function Search(): JSX.Element {

const [query, setQuery] = useState('');

const [results, setResults] = useState<string[]>([]);

// Bad: Synchronous update (can block UI on large lists)

const handleChange = (e: React.ChangeEvent<HTMLInputElement>) => {

setQuery(e.target.value);

const filtered = heavyFilter(e.target.value);

setResults(filtered);

};

return <input value={query} onChange={handleChange} />;

}

import { useState, useTransition } from 'react';

function Search(): JSX.Element {

const [query, setQuery] = useState('');

const [results, setResults] = useState<string[]>([]);

const [isPending, startTransition] = useTransition();

// Good: Use startTransition for non-urgent update

const handleChange = (e: React.ChangeEvent<HTMLInputElement>) => {

setQuery(e.target.value);

startTransition(() => {

const filtered = heavyFilter(e.target.value);

setResults(filtered);

});

};

return (

<div>

<input value={query} onChange={handleChange} />

{isPending && <p>Loading...</p>}

</div>

);

}

**11. Use React Suspense and Streaming SSR**

Wrap slow-loading components in <Suspense> with a fallback. Next.js supports streaming server-side rendering, so parts of the UI can appear immediately while other parts load. For example, put a busy component inside <Suspense> so its data fetch won’t block the rest of the page from rendering.

import HeavyChart from '@/components/HeavyChart';

export default async function Dashboard(): Promise<JSX.Element> {

// Bad: Fetching data and rendering HeavyChart blocks the whole page

const data = await fetchChartData();

return <HeavyChart data={data} />;

}

import { Suspense } from 'react';

const HeavyChart = async () => {

const data = await fetchChartData();

return <Chart data={data} />;

};

export default function Dashboard(): JSX.Element {

return (

<Suspense fallback={<div>Loading chart...</div>}>

<HeavyChart />

</Suspense>

);

}

**12. Use <Link> for Client-Side Navigation**

Use Next.js’s <Link> for internal links to enable prefetching of pages. The docs advise using <Link> for client-side navigation (it prefetches routes in viewport). This makes page transitions nearly instantaneous. Always prefer <Link href="/about">About</Link> instead of a plain <a>.

// Bad: Regular anchor (no prefetching)

export default function Nav(): JSX.Element {

return (

<nav>

<a href="/about">About</a>

</nav>

);

}

import Link from 'next/link';

// Good: Using Link component

export default function Nav(): JSX.Element {

return (

<nav>

<Link href="/about">About</Link>

</nav>

);

}

**13. Profile and Analyze Bundles (Turbopack, Analyzer)**

Take advantage of Next.js’s new tooling for faster builds and analysis. Enable the Rust-based Turbopack bundler (use next dev --turbo) to dramatically speed up compilation (Next.js saw up to ~96% faster updates). Use @next/bundle-analyzer to visualize bundle contents and spot large dependencies. These tools let you identify and eliminate performance bottlenecks in development.

// Bad (next.config.js): default Webpack config

/\*\* @type {import('next').NextConfig} \*/

const nextConfig = {};

module.exports = nextConfig;

// Good: Enable Turbopack and bundle analyzer

/\*\* @type {import('next').NextConfig} \*/

const nextConfig = {

experimental: { turbo: true },

};

const withBundleAnalyzer = require('@next/bundle-analyzer')({

enabled: process.env.ANALYZE === 'true',

});

module.exports = withBundleAnalyzer(nextConfig);

**14. Monitor Web Vitals and Performance Metrics**

Instrument your app to measure real user metrics. Next.js has built-in support via the useReportWebVitals hook. Use it (or third-party analytics) to capture Core Web Vitals (LCP, FID, CLS, etc.) and Next.js’s custom metrics (like hydration time). This lets you detect regressions and focus optimizations where they matter most.

// Bad: No instrumentation, metrics are blind spots

export default function App({ Component, pageProps }) {

return <Component {...pageProps} />;

}

import { useReportWebVitals } from 'next/web-vitals';

// Good: Hook into Web Vitals for reporting

export default function App({ Component, pageProps }): JSX.Element {

useReportWebVitals(console.log); // log metrics or send to analytics

return <Component {...pageProps} />;

}

**15. Use TypeScript and Linting for Safety**

Adopt strict typing and linting for better DX and fewer runtime issues. Next.js recommends using TypeScript and its ESLint setup (including eslint-plugin-jsx-a11y) for catching errors early. Type annotations don’t directly speed up runtime, but they improve code quality and maintainability, indirectly aiding performance by avoiding costly mistakes.

// Bad: No TypeScript (plain JS, no type safety)

/\*\* @type {any} \*/

let count = 0;

export default function Page() {

return <div>{count}</div>;

}

// Good: Using TypeScript with explicit types

export default function Page(): JSX.Element {

const count: number = 0;

return <div>{count}</div>;

}

**Sources:** Official Next.js docs and related resources have informed these patterns. For example, the Next.js production checklist highlights Image, Script, and Font optimizations; the App Router docs describe caching and Server Actions; and recent blog posts cover Streaming, Middleware, and Turbopack. Each code example follows modern TypeScript conventions for clarity and type safety.