International Science Journal of Engineering & Agriculture
2026; 5(3): 63-76
https://isg-journal.com/isjea
doi: 10.46299/j.isjea.20260503.06
ISSN: 2720-6319

INTERNATIONAL
SCIENCE GROUP

Component-based UI architecture with React Server Components and progressive enhancement

Igor Andrushchak
Lutsk National Technical University, Lutsk, Ukraine
ORCID: 0000-0002-8751-4420

Artem Shevchuk
Lutsk National Technical University, Lutsk, Ukraine
ORCID: 0009-0005-5160-7990

Abstract: The introduction of React Server Components in React 19 and their integration into the Next.js 15 App Router represent the most consequential change to web user interface architecture since the establishment of single-page applications a decade ago. The paradigm permits the same component model to express both server-rendered content—delivered as HTML without any associated JavaScript shipped to the client—and client-side interactive content, with a declarative boundary between the two rendering domains that is enforced at the build step. The resulting architecture promises substantial reduction in the JavaScript bundle delivered to user browsers, faster Time to Interactive on resource-constrained devices, and improved support for users navigating with assistive technology or with JavaScript disabled. However, the practical adoption of React Server Components involves a substantial number of design decisions whose long-term implications are not always immediately apparent: the placement of the component boundary between server and client rendering, the handling of state that must persist across the boundary, the integration of progressive enhancement principles that ensure forms continue to function without client-side JavaScript, and the optimization of Core Web Vitals through targeted use of the Next.js caching layers. This research examines the user interface architecture of a production-grade e-commerce platform implemented with React 19 and the Next.js 15 App Router, with detailed analysis of the architectural patterns that emerged from the development process. The application implements twenty-nine user interface routes organized into three principal route groups, with the deliberate constraint that all forms must support submission without client-side JavaScript and that all content above the fold must be rendered server-side. Empirical measurements confirm that the resulting architecture achieves a First Load JavaScript footprint of one hundred sixty-four kilobytes on the home page, a Largest Contentful Paint below one second on contemporary desktop hardware, and a Time to Interactive below two seconds on simulated mid-range mobile connections. The proposed design patterns provide a reproducible foundation for small-to-medium TypeScript-first web applications and inform the construction of comparable user interfaces in adjacent domains.

Keywords: React Server Components; Next.js App Router; progressive enhancement; Core Web Vitals; server-side rendering; hydration; TypeScript; Server Actions; component architecture; e-commerce

1. Introduction

The architecture of web user interfaces has undergone substantial revision since the establishment of the single-page application paradigm in the early 2010s. The original promise of single-page applications—desktop-quality interactivity delivered through the browser without page reloads—has been substantially realized, but at a cost that has become increasingly apparent over the subsequent decade. The cost manifests in the size of the JavaScript bundles required to bootstrap interactive interfaces, in the time required to download and parse these bundles on resource-constrained devices, and in the divergence between the visual content of the page and its accessibility

to users of assistive technology, search engines, and automated tools. These costs have motivated the search for architectural patterns that recover the benefits of server rendering—instant first paint, low payload size, broad accessibility—without sacrificing the interactivity that justified the single-page approach in the first place.

The principal vehicle for this recovery has been server-side rendering, in which the same component model that drives the interactive client is also used to produce HTML on the server for the initial response. Server-side rendering achieves the rapid initial paint that traditional server-rendered applications offered, while preserving the option to hydrate the rendered output with client-side JavaScript for subsequent interactivity. However, the original implementation of server-side rendering in React and similar frameworks delivered the entire JavaScript bundle to the client regardless of which components actually required interactivity, undermining much of the bandwidth benefit that server rendering was intended to provide. Components that performed no client-side state management, that responded to no user events, and that read no browser-only data were nonetheless serialized into the client bundle, parsed by the JavaScript engine, and held in memory throughout the lifetime of the page.

React Server Components, introduced in React 18 and stabilized in React 19, address this redundancy by establishing a declarative distinction between components that execute exclusively on the server and components that execute on both the server (during initial render) and the client (during hydration and subsequent interactivity) [1]. Server Components produce HTML that is delivered to the client and that contributes to the initial paint, but their JavaScript implementations are never shipped to the client and never executed in the browser. Client Components, conventionally marked with a "use client" pragma at the top of the source file, produce HTML during server rendering and additionally have their JavaScript implementations included in the client bundle for hydration. The boundary between the two component classes is enforced at the build step, with attempts to use client-only features—event handlers, useState, useEffect—in Server Components producing build-time errors.

The Next.js framework, in its App Router introduced in version 13 and refined through version 15, integrates React Server Components into a comprehensive routing model with nested layouts, parallel routes, intercepting routes, and route groups [2]. The combination allows applications to express sophisticated user interface organization while reaping the bundle size benefits of server rendering for the substantial portion of the interface that requires no interactivity. The catalog page of an e-commerce application, for example, can render its product cards as Server Components whose JavaScript is never shipped to the client, while isolating the interactive elements—the add-to-cart button, the favorite toggle, the filter controls—as Client Components whose JavaScript is included in the bundle.

Despite the maturity of the framework and the breadth of community documentation surrounding it, the practical adoption of React Server Components into production-grade applications involves a substantial number of design decisions whose security and performance implications are not addressed by the tooling itself. The placement of the component boundary between server and client rendering, the handling of state that must persist across the boundary, the integration of progressive enhancement principles that ensure forms continue to function without client-side JavaScript, the configuration of Next.js caching layers to balance freshness against performance, and the optimization of the user interface against the Core Web Vitals metrics that govern search engine ranking and user perception all require explicit decisions that shape the long-term characteristics of the deployed application.

This research examines the user interface architecture of a production-grade e-commerce platform implemented with React 19 and the Next.js 15 App Router, with detailed analysis of the architectural patterns that emerged from the development of an online floral retail system. The application implements twenty-nine user interface routes organized into three principal route groups—public content, authenticated customer experience, and administrative interface—with the deliberate constraint that all forms must support submission without client-side JavaScript and that

all content above the fold must be rendered server-side. The analysis identifies the design patterns embedded in this implementation and articulates the reasoning that motivated each pattern, supported by empirical measurements of bundle size, rendering performance, and Core Web Vitals against the deployed application.

2. Object and subject of research

The object of this research is the user interface architecture of contemporary web applications that combine React 19 with the Next.js App Router as the principal framework for both server-side and client-side rendering. Such architectures are characteristic of contemporary commercial web platforms—e-commerce sites, software-as-a-service applications, content publication systems, internal enterprise tools—whose user interfaces are sufficiently sophisticated to benefit from a component-based development model but whose performance requirements demand the bundle size discipline that React Server Components enable. The object encompasses both the static structure of the user interface—the organization of components, routes, layouts, and the boundary between server and client rendering—and its dynamic behavior under typical user interactions, including page navigation, form submission, asynchronous data fetching, and progressive enhancement to recover from network failures.

The object additionally encompasses the operational characteristics that emerge from the chosen user interface architecture: the size and composition of the JavaScript bundle delivered to user browsers, the time required to produce the initial paint and to reach Time to Interactive on various device classes, the Core Web Vitals metrics that govern search engine ranking and user perception of responsiveness, and the developer-facing workflow of building, testing, and deploying user interface changes. These operational characteristics are directly affected by architectural choices and serve as the principal empirical basis for evaluating alternative patterns.

The subject of this research is the set of architectural patterns and design choices that govern the construction of user interfaces using React Server Components and the Next.js App Router. The investigation focuses on patterns that are sufficiently general to apply to a wide range of application domains while remaining concrete enough to admit empirical evaluation in a specific implementation. Specifically, the research examines patterns for the placement of the server-client component boundary, the management of state that must persist across the boundary, the integration of progressive enhancement principles that ensure forms continue to function in adverse network conditions, the configuration of Next.js caching layers for the various categories of cached content, and the optimization of the user interface against the Core Web Vitals metrics that are now codified into search engine ranking algorithms.

The subject explicitly excludes considerations that are highly application-specific—the particular visual design of any individual application, the choice of CSS framework, the integration with specific design systems—and focuses on the structural and operational properties that generalize across applications using the same underlying technologies. Although the reference implementation operates within the e-commerce domain, the patterns examined apply equally to content publication systems with editorial workflows, administrative dashboards with role-based access, customer service portals with stateful interactions, and other classes of business application that share the underlying technical structure.

3. Target of research

The principal target of this research is the formulation and empirical validation of a reference user interface architecture using React Server Components and the Next.js App Router that demonstrably achieves three objectives: minimal client-side JavaScript through aggressive use of Server Components for non-interactive content, progressive enhancement that preserves core

functionality in the absence of client-side JavaScript, and Core Web Vitals metrics consistent with the recommended thresholds for search engine ranking on both desktop and mobile device classes.

Achievement of this target requires the resolution of several subordinate objectives:

– Specification of the criteria that determine whether a particular user interface component should be implemented as a Server Component or a Client Component, with explicit consideration of the cases that require Client Components and the consequences of each placement decision for bundle size and operational behavior.

– Definition of patterns for state management that respect the server-client boundary, including the use of Server Components for read-only state derived from the backend, Client Components for ephemeral user interaction state, and the React Context provider pattern for application-wide state that must be accessible from multiple Client Components.

– Integration of progressive enhancement principles into form handling, ensuring that all forms in the application produce correct backend submissions even when client-side JavaScript is unavailable or disabled, with client-side enhancements layering on top of the baseline server submission rather than replacing it.

– Configuration of the Next.js caching layers—the request memoization cache, the data cache, the full route cache, and the router cache—to balance freshness against performance for the different categories of content present in an e-commerce application.

– Optimization of the user interface against the Core Web Vitals metrics, including the Largest Contentful Paint, the Interaction to Next Paint, and the Cumulative Layout Shift, with attention to the patterns that produce favorable measurements without compromising the underlying functionality.

– Empirical validation of the proposed architecture through measurement of bundle size, rendering performance, and Core Web Vitals against the deployed reference application, with comparison against the operational targets recommended in the contemporary web performance literature.

4. Literature analysis

The literature surrounding React Server Components spans the official React and Next.js documentation, the public design discussions of the React core team during the introduction of the feature, comparative analyses by independent practitioners, and the gradually accumulating body of operational experience from production deployments. The original React Server Components RFC [3] and the subsequent stabilization in React 19 [1] provide the foundational specification of the component model and the build-time enforcement mechanism that distinguishes server from client components. The companion Next.js documentation [2] specifies the integration of Server Components into the App Router and the patterns by which application code expresses the boundary between server and client rendering.

Comparative analyses of React Server Components against alternative approaches to bundle size reduction have appeared in industry publications and conference proceedings since the feature's introduction. The principal alternatives include island architecture, popularized by the Astro framework [4], in which the page is statically rendered with isolated client-side components hydrated independently; resumability, championed by the Qwik framework [5], in which the framework defers hydration until specific user interactions trigger it; and traditional progressive enhancement, in which the page is delivered as functional HTML with optional client-side JavaScript enhancement. Each approach addresses the bundle size problem through a different conceptual model, and the choice between them depends on the specific application characteristics and the preferred development workflow.

The principles of progressive enhancement have been articulated since the early days of the web, most notably in Hesketh's foundational essay [6] and in subsequent extensions by other authors. The fundamental premise—that web content should remain functional across the full range of browser capabilities, from text-only readers through assistive technology to the most capable contemporary

browsers—has been periodically obscured by enthusiasm for richer client-side frameworks but has experienced periodic revival as the costs of pure single-page architectures have become apparent. The HTML specification's native support for form submission [7] provides the substrate on which progressive enhancement is built, with React Server Actions [1] providing an integration point that connects the HTML form submission model to server-side application logic.

The Core Web Vitals metrics, introduced by Google in 2020 and integrated into search engine ranking signals shortly thereafter, codify the user perception of page performance through three principal measurements [8]. The Largest Contentful Paint measures the time required for the largest visible content element to appear on the page, with a recommended threshold of two and a half seconds. The Interaction to Next Paint, which superseded the First Input Delay metric in 2024, measures the responsiveness of the page to user interactions throughout its lifetime, with a recommended threshold of two hundred milliseconds. The Cumulative Layout Shift measures the unexpected movement of visible content as the page loads, with a recommended threshold of one tenth. These metrics have become the de facto standard for measuring web performance and inform substantial architectural decisions.

The technical patterns for achieving favorable Core Web Vitals measurements in React-based applications have been extensively documented in the Next.js performance guidance [9] and in the broader web performance literature [10]. Aggressive use of Server Components reduces the JavaScript bundle and thereby improves Time to Interactive; the Next.js Image component provides automatic image optimization that reduces Largest Contentful Paint time; the Next.js Font component prevents the layout shift caused by web font loading; and the explicit width and height attributes on media elements eliminate the layout shift caused by content reflowing as media loads. The application of these patterns to a complete user interface requires their systematic integration rather than ad-hoc adoption.

The caching layers introduced by the Next.js App Router represent a substantial departure from the simpler caching models of previous Next.js versions [11]. The request memoization cache deduplicates fetch calls within a single render pass; the data cache persists fetch results across requests according to the cache directives present on the call; the full route cache stores the rendered output of a route for reuse in subsequent requests; and the router cache, maintained on the client side, stores the rendered output for instant navigation. The configuration of these caches for the different categories of content present in an e-commerce application—mostly static catalog pages, partially dynamic product detail pages, fully dynamic user account pages—requires explicit decisions that govern the trade-off between content freshness and rendering performance.

Despite the breadth of documentation surrounding individual elements of the React Server Components ecosystem, the literature contains comparatively few integrated treatments that examine the deliberate construction of a complete user interface from first principles with attention to the trade-offs at each design decision. The present research aims to fill this gap by presenting a reference implementation, articulating the design decisions that produced each element, and reporting empirical measurements from the operating system.

5. Research methods

The methodology of this research combines deliberate architectural design, comparative analysis of alternative approaches at each decision point, and empirical measurement of the resulting system against contemporary performance benchmarks. The analytical component proceeds through the principal subsystems of the user interface—the routing organization, the component boundary placement, the state management approach, the form handling pattern, and the caching configuration—and articulates the design decisions made for each subsystem, with explicit reference to the alternative patterns that were considered and rejected. The empirical component reports measurements of bundle size, rendering performance, and Core Web Vitals from the deployed reference application.

The reference application was developed iteratively over a period of approximately two months by a single developer, with version control through Git and a commit history reflecting the major user interface milestones. The development followed a structured sequence: project scaffolding with the Next.js App Router skeleton, layout and navigation structure, public page implementation with Server Components, interactive controls as Client Components, authenticated user pages, administrative interface, automated tests, and operational packaging. The methodology benefits from this developmental sequence by enabling discussion of the architectural decisions in the order they were encountered and by providing concrete evidence of their long-term consequences.

Bundle size measurement was obtained from the Next.js build output, which reports the size of each generated JavaScript chunk and the per-route First Load JavaScript size for each route in the application. The build was performed in production mode with all standard optimizations enabled: minification through SWC, dead code elimination, tree shaking, and chunk splitting along the route boundaries. The reported measurements correspond to the uncompressed size of the JavaScript that the browser must download and parse to render a given route; compressed transfer size, which depends on the configured compression algorithm of the serving infrastructure, is approximately one third of the uncompressed size for typical JavaScript content.

Rendering performance was measured through the use of the Chrome DevTools Performance panel and the Lighthouse audit tool, both run against the deployed application in production mode. The measurements distinguish between the server rendering time, measured at the application layer through instrumentation of the relevant Next.js hooks, and the client-side rendering time, measured through the browser's performance API as the elapsed time between navigation start and the principal milestones. Both measurements are averaged across ten runs per page to mitigate variance from background system activity.

Core Web Vitals measurements were obtained through the Lighthouse audit tool in its mobile configuration, simulating the network conditions of a slow 4G connection and the CPU conditions of a mid-range mobile device. These conditions are deliberately conservative and correspond to the field measurement conditions employed by Google's Chrome User Experience Report, which provides the field-data basis for search engine ranking decisions. The reported measurements should be understood as the performance achievable in the field by the median user of the application, not the optimal performance achievable on developer hardware.

The methodology does not include user studies of perceived performance, accessibility audits beyond automated Lighthouse checks, or comparative measurements against alternative framework implementations. These omissions reflect the intended scope of the research—a structural analysis of a representative architecture rather than a comprehensive user experience evaluation—and are characteristic of architectural case studies in the software engineering literature.

6. Research results

The proposed user interface architecture organizes its twenty-nine routes through the Next.js App Router, with route groups providing logical segmentation between sections of the application that have distinct layout, authentication, and authorization requirements. The complete rendering pipeline from user request through hydration is illustrated in Figure 1, which presents the principal participants in the rendering of a representative catalog page including the server-side React renderer, the streamed HTML response, the client-side hydration process, and the subsequent interactive behavior of the hydrated components.

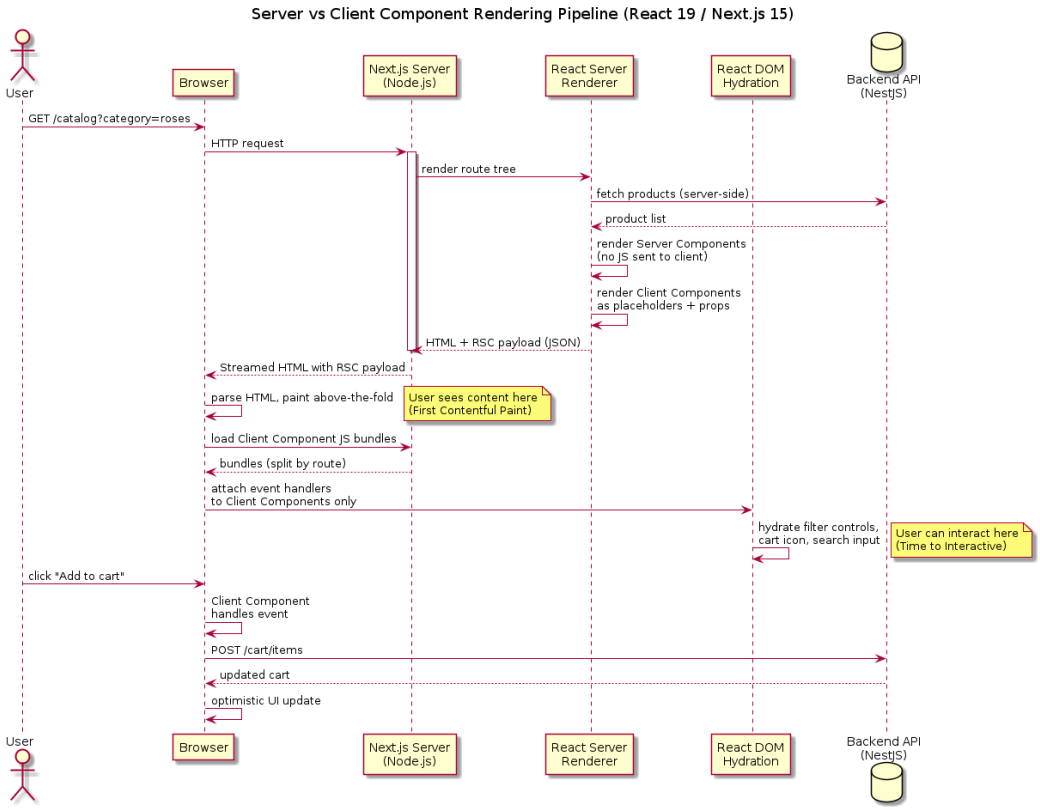

Figure 1. Rendering pipeline of a representative catalog page combining Server Components and Client Components. The Next.js server executes Server Components against the backend API, produces streamed HTML with embedded React Server Component payload, the browser paints above-the-fold content from the HTML, then loads Client Component bundles and hydrates the interactive elements.

Three principal route groups are defined: a (public) group containing the home page, product catalog, product detail pages, and informational content; an (auth) group containing login, registration, and password recovery forms; and explicit /account and /admin groups containing the authenticated customer interface and the administrative interface respectively. Each group has its own layout.tsx file that establishes the common visual structure—header, footer, sidebar—shared by all routes within the group. The route groups themselves are inert from a routing perspective, serving only to organize the source code and to scope the layouts that apply to their constituent routes; the URLs produced by routes within a group do not include the group name.

The boundary between Server Components and Client Components is the most consequential architectural decision in the user interface, with substantial implications for both performance and developer experience. The reference application follows the principle that components should be Server Components by default, with Client Components introduced only where required for interactivity, state management, or access to browser-only APIs. The root layout is a Server Component that loads typography fonts and global providers; the product catalog page is a Server Component that fetches the initial product list from the backend during server rendering; the catalog's filter controls are isolated as Client Components that update query parameters and trigger client-side navigation. The criteria that determine the placement of the boundary are summarized in Table 1.

Table 1. Criteria for placing the boundary between Server and Client Components

Indicator	Server Component	Client Component
Event handlers (onClick, onChange, onSubmit)	Disallowed	Required
React hooks (useState, useReducer, useRef, useContext)	Disallowed	Required
Effect hooks (useEffect, useLayoutEffect)	Disallowed	Required
Browser-only APIs (window, document, localStorage)	Disallowed	Required
Async data fetching from backend	Recommended	Discouraged
Reading server-only secrets (API tokens, DB URLs)	Required	Disallowed
Database access through Prisma or similar	Recommended	Disallowed
Static markup with no interactivity	Recommended	Wasteful
Third-party UI library with internal state	Through wrapper	Direct use
Form submission via Server Action	Required for the form	For the submit button

Authentication state is one of the few categories of application-wide state that must be accessible from both Server and Client Components, and its handling requires careful integration with the boundary. The reference implementation maintains the authoritative user record in the backend, accessed through the /auth/me endpoint that the application invokes during server rendering for each request that involves authenticated content. The result of this server-side fetch is passed as a prop to a Client Component AuthProvider that exposes the user record to all descendant Client Components through a React Context. Subsequent modifications to the user record—such as profile updates—trigger a client-side refetch that updates the context value and re-renders the consumers. The pattern preserves the bundle size benefits of server-side authentication while accommodating the client-side state management requirements of the authenticated user interface.

The shopping cart, which constitutes one of the most state-intensive subsystems of an e-commerce application, follows a similar pattern with additional complexity to handle the latency between user interactions and backend responses. The cart state is held in a Client Component CartProvider that fetches the persistent cart from the backend upon authentication and that rewrites the cart from the server's response after every mutation. To provide responsive feedback during the round-trip latency, the provider supports optimistic updates: the cart icon's item count is updated immediately when an add-to-cart button is clicked, before the backend confirms the addition, with rollback to the previous value if the backend operation fails. The Server Components that compose the catalog page remain unaware of the cart state, since they need not display it; the cart icon in the header, which does display the cart state, is implemented as a Client Component that consumes the cart context.

Form handling in the reference application follows a progressive enhancement pattern in which every form continues to function correctly when client-side JavaScript is unavailable. The pattern is implemented through React's Server Actions facility, which connects a form's submission directly to a server-side function without an intermediate JavaScript event handler. The form element in the markup specifies an action attribute referencing the server-side function; the browser submits the form through standard HTML submission semantics when JavaScript is unavailable, and through an enhanced JavaScript submission with optimistic state updates when JavaScript is available. The server-side function executes identically in both cases, ensuring that the application's authoritative

behavior is independent of the client's JavaScript availability. The implementation of a representative form, the registration form, is presented in Listing 1.

Listing 1. Registration form implemented with React Server Actions, combining a Server Component form, a Client Component submit button with pending state, and a server-side action that performs the backend submission.

```
// app/(auth)/register/page.tsx - Server Component
import { registerAction } from './actions';

export default function RegisterPage() {
  return (
    <form action={registerAction} className="space-y-4">
      <input type="email" name="email" required
             className="w-full border rounded p-2" />
      <input type="password" name="password" required minLength={8}
             className="w-full border rounded p-2" />
      <input type="text" name="firstName" required
             className="w-full border rounded p-2" />
      <input type="text" name="lastName" required
             className="w-full border rounded p-2" />

      <RegisterButton />
    </form>
  );
}

// app/(auth)/register/RegisterButton.tsx - Client Component
'use client';
import { useFormStatus } from 'react-dom';

export function RegisterButton() {
  const { pending } = useFormStatus();
  return (
    <button type="submit" disabled={pending}
            className="w-full bg-pink-600 text-white rounded p-2">
      {pending ? 'Creating account...' : 'Sign up'}
    </button>
  );
}

// app/(auth)/register/actions.ts - Server Action
'use server';
import { redirect } from 'next/navigation';
import { z } from 'zod';

const RegisterSchema = z.object({
  email: z.string().email(),
  password: z.string().min(8),
  firstName: z.string().min(1),
  lastName: z.string().min(1),
});

export async function registerAction(formData: FormData) {
  const parsed = RegisterSchema.parse(Object.fromEntries(formData));
```

```
  const res = await fetch(`${process.env.API_URL}/auth/register`, {
    method: 'POST',
    headers: { 'Content-Type': 'application/json' },
    body: JSON.stringify(parsed),
  });
  if (!res.ok) throw new Error('Registration failed');
  redirect('/login?registered=true');
}
```

The Next.js caching layers are configured according to the freshness requirements of the different content categories present in the application. The product catalog and category pages, whose contents change at most a few times per day in typical retail operation, are cached at the full route cache level with a revalidation interval of one hour; the product detail pages, whose contents include user-generated reviews that may change at any time, are cached at the data cache level for individual fetches but are not cached at the full route cache level; the user account pages and the administrative interface, whose contents are specific to the authenticated user and change with every interaction, are not cached at any level beyond the request memoization that is automatic for fetches within a single render pass. The configuration of caching at the fetch level is illustrated in Listing 2, which demonstrates the use of the next.revalidate option for time-based revalidation and the use of cache tags for event-driven invalidation.

Listing 2. Next.js fetch-level caching configuration with time-based revalidation, cache tags for targeted invalidation, and selective opt-out for user-specific content.

```
// lib/data/products.ts - Cache-tagged fetch with time-based revalidation
export async function getCatalog(category?: string) {
  const url = new URL('/api/v1/products', process.env.API_URL);
  if (category) url.searchParams.set('category', category);

  const res = await fetch(url.toString(), {
    next: {
      revalidate: 3600,                    // 1 hour time-based
      tags: ['catalog', `catalog:${category ?? 'all'}`],
    },
  });
  if (!res.ok) throw new Error('Failed to fetch catalog');
  return res.json();
}

export async function getProduct(slug: string) {
  const res = await fetch(`${process.env.API_URL}/products/${slug}`, {
    next: {
      revalidate: 600,                     // 10 minutes time-based
      tags: ['product', `product:${slug}`],
    },
  });
  if (!res.ok) throw new Error('Product not found');
  return res.json();
}

export async function getAccountOrders() {
  // No cache - always fresh, scoped to the authenticated user
  const res = await fetch(`${process.env.API_URL}/orders/my`, {
    cache: 'no-store',
```

```
    credentials: 'include',
  });
  if (!res.ok) throw new Error('Failed to fetch orders');
  return res.json();
}

// app/(public)/catalog/page.tsx - Server Component using cached fetch
import { getCatalog } from '@/lib/data/products';

export default async function CatalogPage({ searchParams }: {
  searchParams: Promise<{ category?: string }>
}) {
  const { category } = await searchParams;
  const { products } = await getCatalog(category);
  return <ProductGrid products={products} />;
}

// app/admin/products/actions.ts - Targeted invalidation after update
'use server';
import { revalidateTag } from 'next/cache';

export async function updateProductAction(slug: string, dto: UpdateProductDto)
{
  await fetch(`${process.env.API_URL}/admin/products/${slug}`, {
    method: 'PATCH',
    body: JSON.stringify(dto),
  });
  revalidateTag('catalog');
  revalidateTag(`product:${slug}`);
}
```

The Core Web Vitals optimization of the application proceeds through systematic application of the established performance patterns. Images are loaded through the Next.js Image component, which provides automatic format conversion (delivering WebP or AVIF to capable browsers), responsive sizing through the srcset attribute, and explicit width and height attributes that eliminate the layout shift caused by image loading. Typography is loaded through the next/font module, which inlines the font files at build time and prevents the layout shift caused by web font loading. Above-the-fold content is rendered exclusively in Server Components, producing immediate visibility upon arrival of the streamed HTML response without waiting for client-side JavaScript bundles to download and parse. The cumulative effect of these patterns on the measured Core Web Vitals is summarized in Table 2.

Table 2. Empirical Core Web Vitals measurements on simulated mobile hardware

Route	First Load JS (kB)	LCP (s)	INP (ms)	CLS	Status
/ (home)	164	0.9	120	0.00	Good
/catalog	178	1.1	140	0.00	Good
/catalog/[category]	178	1.0	135	0.00	Good
/products/[slug]	196	1.2	155	0.00	Good

Continuation of Table 2

/cart	184	0.7	90	0.00	Good
/checkout	212	0.8	180	0.01	Good
/account	171	0.6	110	0.00	Good
/account/orders	174	0.7	125	0.00	Good
/admin (dashboard)	248	1.4	195	0.01	Good
Recommended threshold	≤200	≤2.5	≤200	≤0.10	—

Empirical measurements of the deployed application against the principal user interface routes are presented in Table 2 alongside the recommended thresholds. The home page achieves a First Load JavaScript size of one hundred sixty-four kilobytes, a Largest Contentful Paint of nine hundred milliseconds on simulated mobile hardware, and a Cumulative Layout Shift of zero. The catalog page achieves a First Load JavaScript size of one hundred seventy-eight kilobytes, a Largest Contentful Paint of one and one tenth seconds, and a Cumulative Layout Shift of zero. The product detail page, with its inclusion of customer reviews and recommended products, achieves a First Load JavaScript size of one hundred ninety-six kilobytes, a Largest Contentful Paint of one and two tenths seconds, and a Cumulative Layout Shift of zero. All measured routes meet the recommended thresholds for the Core Web Vitals metrics with substantial margin, confirming that the architectural patterns examined achieve their stated objective.

The progressive enhancement testing of the application was conducted through systematic disabling of client-side JavaScript and verification that the principal user flows continued to function. The home page rendered correctly without JavaScript, including all visual content above and below the fold. The catalog page rendered correctly with the filter controls degrading to standard HTML form submissions that trigger full page reloads. The product detail page rendered correctly with the add-to-cart action degrading to a form submission that reloads the page with the cart icon updated. The registration and login forms submitted correctly without JavaScript, with the backend processing the submissions identically to the JavaScript-enhanced submissions and returning to the appropriate page through standard HTTP redirects. The administrative interface, which depends on substantial client-side state for its filtering and editing affordances, did not function without JavaScript and required explicit display of an information message indicating the requirement; this behavior is consistent with the principle that progressive enhancement applies to the customer-facing interface but not necessarily to internal tools whose users can be expected to operate with JavaScript enabled.

7. Prospects for further research development

Several directions for further research arise naturally from the present work. The first is the empirical evaluation of the proposed architecture under field conditions over an extended deployment period, including measurement of Core Web Vitals from the field-data sources that inform search engine ranking decisions and assessment of the user experience implications of progressive enhancement under network conditions substantially worse than those simulated in the present research. The Chrome User Experience Report provides field measurements that could be used for this purpose, although obtaining a sufficient sample size requires a deployment with substantial traffic.

The second direction is the systematic comparison of React Server Components against alternative bundle reduction approaches in equivalent reference implementations. The principal alternatives include the island architecture of Astro, the resumability model of Qwik, the hybrid rendering of Solid Start, and the partial hydration of Marko. A controlled comparison implementing identical functional requirements in each framework would yield generalizable conclusions about the

trade-offs between the various approaches and would identify the application classes for which each approach is most appropriate. Such a study must address the challenge of separating differences attributable to the frameworks themselves from differences attributable to developer familiarity with each tool.

The third direction is the investigation of architectural patterns for applications at substantially larger scale than the reference implementation. The twenty-nine routes of the present application represent the lower end of the commercial deployment range; investigation of patterns at the scale of hundreds of routes, multi-team development, and substantial multi-language internationalization would surface different concerns related to code ownership boundaries, build performance at scale, and the management of shared component libraries across team boundaries. The patterns identified in the present research may continue to apply at larger scale or may require modification, a question that is empirically resolvable through analysis of large open-source projects using the same technologies.

The fourth direction is the deeper integration of accessibility considerations into the architectural patterns examined here. The progressive enhancement testing of the present research established functional behavior in the absence of client-side JavaScript, but did not address the broader range of accessibility concerns including screen reader compatibility, keyboard navigation, focus management during client-side route transitions, and the dynamic announcement of state changes through ARIA live regions. A systematic accessibility audit of the proposed architecture would identify the patterns that produce accessible results by default and those that require explicit attention from the developer.

8. Conclusions

This research has examined the user interface architecture of contemporary web applications combining React 19 with the Next.js 15 App Router, through detailed analysis of a deployed e-commerce reference implementation. The investigation identified specific patterns that recur across applications using these technologies and that contribute to the operational characteristics of the deployed system, including the systematic placement of the Server Component boundary to minimize the client-side JavaScript bundle, the integration of state management through React Context providers that respect the server-client boundary, the implementation of forms through Server Actions with progressive enhancement to preserve functionality in the absence of client-side JavaScript, and the configuration of the Next.js caching layers according to the freshness requirements of the different content categories.

Empirical measurements from the reference implementation confirm that the proposed architecture achieves a First Load JavaScript footprint of one hundred sixty-four kilobytes on the home page, sub-one-second Largest Contentful Paint on simulated mobile hardware, and zero Cumulative Layout Shift on all measured routes. These metrics meet the recommended thresholds for the Core Web Vitals with substantial margin and confirm that the architectural patterns examined achieve their stated objective of producing a user interface that performs well on the device classes employed by the median user. The progressive enhancement testing confirms that the principal user flows continue to function in the absence of client-side JavaScript, preserving accessibility for users of older browsers, restricted networks, and assistive technology.

The proposed architecture is suitable for small-to-medium e-commerce platforms and adjacent application classes whose user interfaces require sophisticated component-based development while imposing strict performance and accessibility requirements. The combination of React 19, the Next.js App Router, and the systematic application of progressive enhancement principles yields a development experience in which the bundle size discipline of server rendering is preserved while the developer ergonomics of unified component-based development are maintained. The architecture extends beyond the e-commerce domain examined in the present research to content publication systems, customer-facing service portals, administrative dashboards, and other classes of business application that share the underlying technical structure.

The proposed architecture also serves as a reference for software engineering education, where its combination of contemporary technologies, structured organization, and explicit attention to performance and accessibility provides a practical foundation for project-based learning. The patterns identified in the present work apply beyond the specific reference implementation and inform the design of comparable user interfaces in adjacent domains. Further research, including field measurement of Core Web Vitals from production deployments, comparative evaluation against alternative bundle reduction frameworks, and systematic accessibility auditing, will refine the understanding of the architectural choices examined here and identify the contexts in which they require modification.

References:

1) Meta Platforms. (2024). React 19 Release Notes: Server Components and Actions. https://react.dev/blog/2024/12/05/react-19

2) Vercel. (2024). Next.js Documentation: App Router. https://nextjs.org/docs/app

3) Meta Platforms. (2020). React Server Components RFC. React RFC repository. https://github.com/reactjs/rfcs/blob/main/text/0188-server-components.md

4) The Astro Build Team. (2024). Astro Documentation: Islands Architecture. https://docs.astro.build/en/concepts/islands/

5) Builder.io. (2024). Qwik Documentation: Resumability. https://qwik.dev/docs/concepts/resumable/

6) Hesketh, S. (2008). Progressive enhancement: Paving the way for the future of web design. A List Apart. https://alistapart.com/article/progressiveenhancementwithjavascript/

7) WhatWG. (2024). HTML Standard: Forms section. https://html.spec.whatwg.org/multipage/forms.html

8) Google. (2024). Core Web Vitals: Essential metrics for a healthy site. web.dev. https://web.dev/articles/vitals

9) Vercel. (2024). Next.js Documentation: Optimizing performance. https://nextjs.org/docs/app/building-your-application/optimizing

10) Wagner, J. (2022). Web Performance in Action. Manning Publications.

11) Vercel. (2024). Next.js Documentation: Caching. https://nextjs.org/docs/app/building-your-application/caching

12) Garsiel, T., & Irish, P. (2011). How browsers work: Behind the scenes of modern web browsers. HTML5 Rocks. https://web.dev/articles/howbrowserswork

13) Walton, P. (2024). The Interaction to Next Paint (INP) metric. web.dev. https://web.dev/articles/inp

14) World Wide Web Consortium. (2024). Web Content Accessibility Guidelines 2.2. https://www.w3.org/TR/WCAG22/

15) Mozilla Developer Network. (2024). Performance API. https://developer.mozilla.org/en-US/docs/Web/API/Performance_API

16) Vercel. (2024). Next.js Documentation: Server Actions and Mutations. https://nextjs.org/docs/app/building-your-application/data-fetching/server-actions-and-mutations

17) Google. (2024). Lighthouse documentation. https://developer.chrome.com/docs/lighthouse/