The 2025 Recursive React PRD: An Architectural Standard for AI-Driven Development

1. Executive Summary and System Meta-Protocol

The rapid evolution of frontend technologies, specifically the transition to React 19 and Next.js 15, has necessitated a fundamental reimagining of application architecture. Concurrently, the rise of AI-assisted software development—powered by coding agents like Cursor, Windsurf, and Copilot—demands a new type of specification: one that is not merely readable by humans but optimized for recursive ingestion by Large Language Models (LLMs). This document serves that dual purpose. It is a comprehensive Product Requirements Document (PRD) defining the "Gold Standard" for React development in 2025, and simultaneously, it is a recursive instruction set for the very AI agents tasked with building it.

This architecture moves beyond the component-centric models of the past decade into a **Server-First, Signal-Driven, and AI-Native** methodology. We leverage the React Compiler to eliminate manual memoization, Feature-Sliced Design (FSD) to enforce strict architectural boundaries that prevent "context pollution" in LLM reasoning, and a "Zero-Config" styling engine via Tailwind CSS v4. Furthermore, this specification mandates a strict separation of concerns regarding state management—isolating Server State (TanStack Query) from Client State (Zustand)—and enforces rigorous type safety through strict TypeScript configurations that specifically target common AI hallucination patterns, such as assuming array access safety.

1.1 Recursive Meta-Protocol for AI Agents

Instruction to the AI Agent: You are to treat this document as the immutable source of truth for all code generation tasks. This PRD is recursive; it contains the logic required for you to understand your own operating parameters. When tasked with generating a React application or feature, you must:

1. **Ingest:** Read this entire document to establish the architectural context.
2. **Constraint Checking:** Before generating any code, cross-reference your plan against the "Prohibited Patterns" defined in each section.
3. **Structure:** Adopt the folder structure strictly defined in the Feature-Sliced Design section. Do not hallucinate arbitrary folder structures like src/components or src/hooks.
4. **Self-Correction:** After generating code, perform a virtual "lint" pass against the rules defined herein (e.g., "Did I use useEffect for data fetching? If yes, refactor to TanStack Query").

This document is exhaustive. It anticipates the edge cases where AI agents typically fail—such as circular dependencies, state management confusion, and accessibility violations—and provides rigid guardrails to prevent them. By adhering to this 2025 standard, we ensure that the resulting codebase is robust, maintainable, and aligned with the cutting edge of the React ecosystem.

2. The Core Runtime Architecture: React 19 and Next.js 15

The foundation of our 2025 stack is built upon the deep integration between React 19's concurrent rendering engine and Next.js 15's application framework. This combination fundamentally alters the lifecycle of a React application, shifting heavy computation to the server and streamlining the client-side interactivity model to reduce bundle sizes and improve Time-to-Interactive (TTI).

2.1 The React 19 Paradigm Shift

React 19 introduces the most significant changes to the library since the introduction of hooks in 2018. The explicit goal is to lower the "floor" of complexity for developers and AI agents alike by automating performance optimizations and unifying data fetching patterns. The manual management of reactivity is replaced by compiler-driven optimization.

2.1.1 The React Compiler and Automatic Memoization

Historically, React developers struggled with the manual application of useMemo, useCallback, and React.memo. This manual dependency management was a primary source of bugs and performance issues. Misuse led to memory overhead, while omission led to unnecessary re-renders. For AI agents, correctly determining the dependency array for complex effects was a frequent source of hallucinations.[1]

React 19 integrates the **React Compiler**, which automatically memoizes components and hooks at build time. This compiler understands the semantic flow of data through the application and applies fine-grained memoization strategies that are often superior to manual implementation.

Feature	Pre-React 19 (Manual)	React 19 (Compiler)	AI Implication
Component	React.memo(Com	Automatic	Do not generate

Memoization	ponent)		React.memo.
Hook Memoization	useCallback(() => {}, [deps])	Automatic	Do not generate useCallback.
Calculation Caching	useMemo(() => a + b, [a, b])	Automatic	Do not generate useMemo.
Dependency Arrays	Manual verification required	Inferred by Compiler	Reduced risk of stale closures.

- **Architectural Mandate:** Explicit manual memoization is now considered an anti-pattern unless profiling demonstrates a specific need that the compiler misses. The code generated should be "naive" in its structure, focusing purely on business logic and render purity, trusting the compiler to optimize the execution graph.
- **Optimization Strategy:** The AI must focus on keeping components small and focused. The compiler is most effective when the abstract syntax tree (AST) of a component is not overly complex. Large, monolithic components can confuse the compiler's static analysis capabilities.[1]

2.1.2 Actions and Form Handling

Data mutation has historically been a fragmented experience in React, relying on onSubmit handlers, manual fetch calls, and complex loading state management. React 19 standardizes this with **Actions**—functions that handle data submission and update state automatically.[1]

Server Actions allow functions to run exclusively on the server, callable directly from client components or form action attributes. This eliminates the need for distinct API route files for simple CRUD operations, reducing the boilerplate that AI agents need to manage.

- **Mechanism:** When a form with a Server Action is submitted, Next.js handles the POST request, serializes the FormData, executes the function on the server, and returns the result, all without the developer writing a single line of API routing code.
- **Progressive Enhancement:** Forms using Actions function even if JavaScript has not yet loaded, providing a robust baseline experience.
- **Feedback Loops:** The useActionState and useFormStatus hooks allow the UI to respond to the state of the action (pending, success, error) without manual state variables. The

useOptimistic hook enables immediate UI updates before the server response is received, a critical pattern for perceived performance.[3]

AI Instruction: Prefer native <form action={serverAction}> patterns over onClick handlers with fetch. When generating mutations, always scaffold the useActionState hook to handle the response cycle.

2.1.3 The use API and Async Context

The new use API permits the reading of resources (Promises and Context) within the render phase, even conditionally. This breaks the rigid "Rules of Hooks" constraint regarding conditional execution for this specific API, allowing for more flexible component composition.[1]

- **Async Context:** The use(Context) pattern is safe for use in async boundaries and loops, enabling patterns where themes or user preferences are applied conditionally without extracting logical sub-components.
- **Promise Unwrapping:** const data = use(promise) suspends the component until the promise resolves, integrating seamlessly with Suspense boundaries. This replaces the need for useEffect data fetching in Client Components.

2.2 Next.js 15: The Application Orchestrator

Next.js 15 acts as the production orchestrator for these React 19 features, introducing stricter caching defaults and deeper integration with the new primitives. It serves as the "operating system" for the application.

2.2.1 Caching Semantics and fetch Behavior

One of the most critical changes in Next.js 15 is the shift in caching strategy. In previous versions (Next.js 13/14), fetch requests were cached by default, often leading to stale data confusion. In Next.js 15, fetch requests, GET Route Handlers, and client navigations are **no longer cached by default**.[5] This "fetch-defaults-to-dynamic" approach aligns better with the dynamic nature of modern AI-driven applications where data freshness is paramount.

- **Optimization Strategy:** AI agents must now explicitly opt-in to caching via cache: 'force-cache' or next: { revalidate: 3600 } when data is known to be static. This prevents the common "stale data" bugs that plagued previous versions.
- **Revalidation:** We strictly utilize revalidateTag for precise cache invalidation (e.g., clearing the 'products' cache tag after an admin updates a product), rather than time-based revalidation, to ensure immediate consistency.[6]

2.2.2 Turbopack and Dev Experience

Next.js 15 stabilizes Turbopack for development, providing sub-second refresh times.[5] While this is an infrastructural benefit, it implies that the feedback loop for AI-generated code (write -> test -> fix) is significantly faster, enabling more iterative "self-healing" workflows for coding

agents.

2.2.3 Async Request APIs

Next.js 15 transitions APIs like headers(), cookies(), and params to be asynchronous.[5] This is a breaking change that frequently causes AI hallucinations if the model is trained on older data.

- **Breaking Change Rule:** AI models must never access props.params.slug directly.
- **Correction Rule:** All access to dynamic request data in Server Components must be awaited: const { slug } = await params; or const cookieStore = await cookies();. Failure to await these promises will result in runtime errors in Next.js 15.

2.3 Server Components vs. Client Components Strategy

The 2025 architecture enforces a **"Server by Default"** mentality. This reduces the JavaScript bundle size sent to the client, improving performance on low-end devices.

Feature	Server Component (Default)	Client Component ('use client')
Data Fetching	Direct DB access, internal API calls	via Route Handlers or Server Actions
Interactivity	None (Static HTML)	onClick, onChange, useEffect, useState
Bundle Cost	Zero (Code stays on server)	Adds to client JS bundle
Access to Browser APIs	No (window, document are undefined)	Yes
Context	Can read via use (limited)	Can provide and consume Context

- **Leaf-Node Pattern:** Interactivity should be pushed to the "leaves" of the component

tree. An entire page should rarely be a Client Component. Instead, a Server Component page should fetch data and pass it as props to small, interactive Client Component "islands".[7]

- **Serialization Constraints:** Data passed from Server to Client components must be serializable. React 19 supports passing Promises, Dates, Maps, and Sets more transparently, but circular references and functions remain forbidden. The AI must ensure that any props passed across the network boundary are clean data structures.[8]

3. Architectural Design Patterns: Feature-Sliced Design (FSD)

To manage the complexity of large-scale, AI-generated codebases, we adopt **Feature-Sliced Design (FSD)**. Unlike traditional "layer-based" (components/hooks/utils) or "route-based" structures, FSD organizes code by **business domain** and **architectural responsibility**. This provides clear "shelves" for AI agents to place files, minimizing circular dependencies and preventing the "spaghetti code" phenomenon often seen in long-context AI generations.[9]

3.1 The FSD Hierarchy and Dependency Rules

The structure imposes a strict unidirectional dependency flow: **App > Processes > Pages > Widgets > Features > Entities > Shared**. A layer can only import from layers *below* it. This rule is absolute and must be enforced by linting rules and AI prompt constraints.

3.1.1 Layer Analysis

1. **app/ (Root of FSD in src/app):** This layer initializes the application. It contains global providers, styles, and the entry point logic. In the context of Next.js, this also houses the "App Router" specific files, but these are strictly thin wrappers around the logic defined in lower layers.[11]
2. **processes/ (Optional):** This layer handles complex, multi-step scenarios that span multiple pages, such as a multi-stage checkout or a complex user registration wizard. It orchestrates widgets and features but does not contain UI primitives itself.
3. **pages/:** This layer is for composition. A "Page" in FSD assembles Widgets, Features, and Entities into a coherent view. It should contain minimal logic, focusing instead on layout and data injection.
4. **widgets/:** These are self-contained UI blocks that provide specific functionality, such as a Header, NewsFeed, or ProductList. Widgets combine Features and Entities into reusable units.
5. **features/:** This layer contains the user interactions that bring business value. Examples include addToCart, likePost, loginByEmail. A feature contains the UI for the interaction (e.g., a button), the logic (event handlers), and the API calls (server actions).
6. **entities/:** This layer represents the business domain models (e.g., User, Product, Order, Payment). It contains the visual representation of the entity (e.g., UserCard), the

TypeScript interfaces, and data transformation logic. Entities generally do *not* contain direct business logic for mutations; they are the "nouns" of the system.

7. **shared/:** This is the infrastructure layer. It contains reusable code that is not specific to the business domain, such as the UI kit (buttons, inputs), generic helper functions, API clients, and third-party library configurations.[9]

3.2 Adapting FSD for Next.js App Router

A well-known conflict exists between Next.js's file-system routing (which enforces a nested folder structure in the app directory) and FSD's flat slice structure. We resolve this by treating the Next.js app directory purely as a **Router Layer**—an implementation detail of the framework rather than a structural component of the business logic.[11]

Standardized Folder Structure for 2025:

```
root/
├──.cursor/
│ └── rules/ # AI Governance Rules (Recursive PRD enforcement)
├── src/
│ ├── app/ # Next.js App Router (Entry Point & Routing)
│ │ ├── (auth)/ # Route Group
│ │ │ └── login/
│ │ │ └── page.tsx -> Imports { LoginPage } from '@/pages/login'
│ │ ├── layout.tsx # Global Layout
│ │ └── page.tsx # Landing Page
│ ├── pages/ # FSD Pages Layer
│ │ ├── home/
│ │ │ └── ui/
│ │ │ └── Page.tsx
│ │ └── login/
│ ├── widgets/ # FSD Widgets Layer
│ │ ├── header/
│ │ └── sidebar/
│ ├── features/ # FSD Features Layer
│ │ ├── auth-login/ # Slice: Authentication Logic
│ │ │ ├── ui/ # LoginForm Component
│ │ │ ├── model/ # Stores (Zustand) & Hooks
│ │ │ └── api/ # Server Actions & DTOs
│ │ └── cart-add/
│ ├── entities/ # FSD Entities Layer
│ │ ├── user/ # Slice: User Domain
│ │ │ ├── ui/ # UserCard, UserAvatar
│ │ │ └── model/ # Types, Validation Schemas (Zod)
```

```
│ │ └── product/
│ └── shared/ # FSD Shared Layer
│ ├── ui/ # Radix/Shadcn Primitives (Button, Input)
│ ├── api/ # Base Fetch Client / Query Client
│ └── lib/ # Utils (cn, formatters)
├── tailwind.config.ts # Configuration
└── tsconfig.json # Strict Types
```

- **AI Instruction:** When asked to "create a login page," do not put the logic in src/app/login/page.tsx. Create the logic in src/features/auth-login (for the form) and src/pages/login (for the page layout), then simply export the page component to the Next.js App Router. This keeps the routing layer thin and the business logic portable and testable.

3.3 Public API Pattern and Encapsulation

Each slice (folder within a layer) must define a index.ts file, known as the Public API.

- **Rule:** Only exports defined in index.ts are accessible to upper layers. Internal implementation details (ui/InternalComponent.tsx, model/internalStore.ts) are encapsulated.
- **AI Enforcement:** If an AI agent attempts to import deeply (e.g., import... from '@/entities/user/ui/UserCard'), it must be corrected to import from the slice root (import { UserCard } from '@/entities/user'). This ensures that refactoring internal file structures does not break the entire application dependency tree.[13]

4. State Management Strategy: The Separation of Concerns

In the 2025 ecosystem, the "global store" pattern (putting everything in Redux or a massive Context) is considered obsolete and harmful. We strictly enforce the separation of **Server State** and **Client State**. This distinction is critical for performance and code maintainability.[14]

4.1 Server State: TanStack Query

Server state is asynchronous, owned by a remote source, and potentially outdated. It does not belong in a synchronous client store. We use **TanStack Query (v5+)** to manage this data.

- **Role:** TanStack Query handles caching, deduping requests, background refetching, and managing loading/error states. It is the single source of truth for all data fetched from the backend.
- **Integration with Server Components:**
  - **Prefetching:** In Next.js Server Components, we use the QueryClient to prefetch data on the server. We then dehydrate this state and pass it to the client. This ensures the user sees data immediately (SSR) while the client takes over for subsequent updates.

    - **Hydration:** Pass initial data to Client Components via the HydrationBoundary pattern. This avoids "prop drilling" initial data down the tree.[16]
- **Query Keys:** We utilize a strict **Query Key Factory** pattern to prevent key collisions and ensure type safety for cache invalidation. AI agents must generate these factories for every Entity.

```TypeScript
// src/entities/product/api/query-keys.ts
export const productKeys = {
  all: ['products'] as const,
  lists: () => [...productKeys.all, 'list'] as const,
  list: (filters: string) => [...productKeys.lists(), { filters }] as const,
  details: () => [...productKeys.all, 'detail'] as const,
  detail: (id: string) => [...productKeys.details(), id] as const,
};
```

- **Anti-Pattern:** Do not store API responses in a Zustand store. If the data comes from the server, it stays in the Query Cache.

4.2 Client State: Zustand

Client state is synchronous and owned by the browser (UI state, form inputs, session settings). We use **Zustand** for its minimal boilerplate and unopinionated structure.

- **Role:** Managing isSidebarOpen, theme, complex wizard form steps, or interactive visual states that do not persist to the backend.
- **Structure:** Create small, atomic stores per feature slice (e.g., useCartStore, useUIStore) rather than one monolithic store. This supports code splitting and reduces bundle size.
- **Selector Pattern:** Components should only subscribe to the specific slice of state they need to avoid unnecessary re-renders.

```TypeScript
const bears = useStore((state) => state.bears) // Good: Re-renders only when bears change
const state = useStore() // Bad: Re-renders on any store change
```

- **AI Instruction:** If a variable is persisted in a database, use TanStack Query. If it resets on page refresh (or is persisted only in localStorage), use Zustand.[14]

4.3 Handling Federated State

In micro-frontend or complex modular architectures, the "multiple instance" problem can occur where different bundles instantiate their own stores. We ensure that the QueryClient and Zustand stores are initialized as singletons. This guarantees that state is shared correctly across module boundaries, preventing issues like authentication tokens being valid in one part of the app but missing in another.[18]

5. The Design System Engine: Tailwind CSS v4

Tailwind CSS v4 represents a complete rewrite of the framework, focused on performance and leveraging modern CSS features. It is the mandatory styling engine for this architecture, replacing CSS-in-JS libraries which often add runtime overhead.

5.1 The v4 Engine & Zero Configuration

Tailwind v4 abandons the JavaScript-heavy configuration (tailwind.config.js) for a CSS-first approach.

- **Configuration:** Theme configuration is done directly in CSS variables within the main CSS file using the @theme directive.
- **Performance:** The Rust-based engine provides incremental builds measured in microseconds, significantly speeding up the dev loop.[19]
- **Automatic Content Detection:** The engine automatically scans the project graph to find class usage. We no longer need to manually configure the content array in a config file, reducing setup errors.[19]

5.2 Modern CSS Features

- **OKLCH Color Space:** We mandate the use of the oklch() color function. This perceptual color space provides consistent lightness across different hues and enables vibrant, accessible colors that interpolate correctly in gradients, avoiding the "gray dead zone" seen in sRGB gradients.[19]
- **@property:** We leverage registered custom properties (@property) for animating complex values like gradients, which was previously impossible with standard CSS.
- **Dynamic Spacing:** We utilize container queries and fluid typography natively supported in v4 to build resilient layouts that adapt to any device context.

5.3 Component Composition: Class Variance Authority (CVA)

While Tailwind is utility-first, we use **Class Variance Authority (CVA)** in the shared/ui layer to create reusable, type-safe component variants. This bridges the gap between utility classes and component design systems.

- **Pattern:**

```
TypeScript
import { cva } from "class-variance-authority";
const buttonVariants = cva("px-4 py-2 rounded font-semibold", {
  variants: {
    intent: {
      primary: "bg-blue-500 text-white hover:bg-blue-600",
      secondary: "bg-gray-200 text-gray-900 hover:bg-gray-300",
    },
```

```
    size: {
      sm: "text-sm",
      lg: "text-lg",
    },
  },
  defaultVariants: {
    intent: "primary",
    size: "sm",
  },
});
```

- **Rule:** Utility classes (text-center p-4) belong in the HTML/JSX. Complex conditional class logic belongs in CVA recipes within the shared layer.

6. Type Safety and Developer Experience: Strict TypeScript

TypeScript in 2025 is not just about types; it is about verifying the structural integrity of the architecture. We configure TypeScript to be as strict as possible to force AI agents to handle edge cases explicitly.

6.1 Strict Configuration Constraints

We adhere to the strictest possible tsconfig.json settings. This is a non-negotiable requirement for all projects generated under this PRD.[21]

JSON

```
{
  "compilerOptions": {
    "strict": true,
    "noUncheckedIndexedAccess": true,
    "exactOptionalPropertyTypes": true,
    "noImplicitOverride": true,
    "forceConsistentCasingInFileNames": true,
    "moduleResolution": "Bundler",
    "jsx": "preserve",
    "plugins": [{ "name": "next" }]
  }
```

```
}
```

- **noUncheckedIndexedAccess:** This is the most critical setting. Accessing an array arr or an object index obj[key] will return T | undefined instead of T.
  - **Impact:** The developer (or AI) *must* write a check (if (item)...) to narrow the type. This single setting eliminates a vast class of runtime undefined errors (the "White Screen of Death") that frequently occur when assuming data exists.[21]
- **reset.d.ts:** We recommend including a Total TypeScript reset file in the project. This improves default typings for global objects like JSON.parse (returning unknown instead of any) and fetch.

6.2 End-to-End Type Safety

With Next.js Server Actions, we can share types directly between the server-side logic and the client-side invocation site.

- **Validation:** We use **Zod** for runtime validation of all inputs in Server Actions. While TypeScript ensures static correctness, it cannot validate runtime data from user input. Zod ensures that the data actually matches the type.
- **AI Instruction:** Every Server Action generated must begin with ZodSchema.parse(data). Never trust client input, even if it is typed in the code.

7. Quality Assurance: Testing and Accessibility

Reliability is paramount. We employ a "Testing Trophy" strategy: heavy on integration and static analysis, lighter on unit tests for UI, and heavy on E2E for critical user flows.

7.1 Unit and Integration Testing: Vitest

We use **Vitest** for unit testing due to its speed and seamless compatibility with Vite/Next.js workflows.[24]

- **Server Components:** Testing async Server Components in Vitest is complex because they run in a Node environment but render JSX.
  - **Strategy:** We prefer testing the *logic* (utility functions, data transformers) via unit tests and the *rendering* via E2E tests. We do not attempt to shallow render Server Components in JSDOM.
- **Client Components:** We use **React Testing Library**. Tests must focus on user interactions (userEvent.click) rather than implementation details (state.value === 'active'). This ensures that tests do not break when we refactor the internal code structure.

7.2 End-to-End Testing: Playwright

Playwright is the standard for E2E testing. It is used to verify the integration of Server Components, Database connectivity, and Client interactivity.

- **Critical Flows:** We mandate E2E tests for "Critical Paths" (Login, Checkout, Core Feature usage).
- **Mocking:** We use Playwright's network interception to mock external third-party APIs (like Stripe or Auth0) during tests to ensure determinism.

7.3 Accessibility (WCAG 2.2)

The application must meet **WCAG 2.2 Level AA** standards. Accessibility is a compile-time requirement, not a post-launch cleanup task.[26]

- **Focus Appearance:** WCAG 2.2 introduces strict rules for focus indicators. We rely on Tailwind's focus-visible: utilities to ensure high-contrast focus rings that are visible but not obtrusive to mouse users.
- **Target Size:** All interactive targets must be at least 24x24 CSS pixels.
- **Linting:** eslint-plugin-jsx-a11y must be enabled and set to strict mode. This plugin catches common errors like missing alt text or invalid ARIA attributes during the build process.[28]
- **AI Instruction:** Always generate aria-label for icon-only buttons. Ensure sufficient color contrast using OKLCH tools. Use semantic HTML (<main>, <nav>, <article>, <aside>) by default instead of .

8. Recursive Context Engineering and Agent Configuration

This section defines the **"Meta-Protocol"** for AI agents. It explains specifically how the AI should read this document to generate *new* PRDs or Code, ensuring the recursive nature of the system.

8.1 The Recursive PRD Structure

A Recursive PRD is designed to be self-replicating and self-correcting.

1. **Context Loading:** The AI agent reads this master document to understand the "Laws of Physics" for the project.
2. **Task Analysis:** The agent breaks down a user request (e.g., "Build a dashboard") against the constraints here (FSD structure, TanStack Query, etc.).
3. **Prompt Generation:** The agent generates a specific "Implementation Plan" (a mini-PRD) that references these sections.
4. **Code Generation:** The agent writes code that strictly adheres to the generated plan.
5. **Self-Correction:** The agent validates the code against the "Prohibited Patterns" list (e.g., "Did I use useEffect for data fetching? If yes, refactor to TanStack Query").

8.2 System Prompts for Agents (Cursor/Windsurf)

To enforce this standard, specific rule files must be placed in the project root. The AI agent

must verify the existence of these files.[30]

8.2.1 .cursor/rules/react-2025-master.mdc

This file acts as the enforcement layer for Cursor. It is a condensed version of this PRD.

---

description: Master rules for React 2025 Architecture. Apply to all React/Next.js files. globs: *.tsx, *.ts, *.jsx, *.js

React 2025 Master Architecture

You are an expert React Architect. Follow these rules strictly.

1. Architecture & Structure

- Use **Feature-Sliced Design (FSD)**.
- Do not place business logic in src/app. Use src/features or src/entities.
- Import dependencies only from layers *below*.

2. React 19 & Next.js 15

- **Server Components by Default**. Use 'use client' only for interactivity.
- Use **Server Actions** for mutations. Do not use API routes for simple CRUD.
- **NO manual memoization** (useMemo, useCallback) unless specifically requested. Trust the React Compiler.
- Use use() for Context and Promises.

3. State Management

- **Server State**: Must use TanStack Query.
- **Client State**: Must use Zustand.
- **NO** useEffect for data fetching.

4. Styling (Tailwind v4)

- Use CSS variables for theme config.
- Use oklch colors.
- Zero runtime CSS-in-JS.

5. TypeScript

- Strict mode is ON.
- Handle undefined from array access (noUncheckedIndexedAccess).
- Use Zod for boundary validation.

8.2.2 Recursive Prompting Strategy (Meta-Prompting)

When the task is complex, the AI must use **Recursive Meta-Prompting**.[32]

- **Phase 1 (Architect):** "Analyze the user request. Based on the Master PRD, outline the FSD slices needed, the data model, and the server actions."
- **Phase 2 (Tech Lead):** "Review the Phase 1 plan. Check for circular dependencies in FSD. Ensure security in Server Actions. Output a refined checklist."
- **Phase 3 (Coder):** "Implement the checklist. Generate one file at a time. After every file, check against the Master PRD constraints."

9. Operational Standards and Error Handling

9.1 Error Handling

Next.js 15 provides granular error handling. We must utilize the full spectrum of error boundaries.

- **error.tsx:** Place in every major route segment (e.g., src/app/dashboard/error.tsx) to catch runtime errors without crashing the whole app.
- **global-error.tsx:** Mandatory for catching errors in the root layout. It must define its own <html> and <body> tags.
- **Recovery:** Always include a reset() button in error boundaries to allow users to retry the render.[34]

9.2 Security Best Practices

- **Server Actions:** Treat them as public endpoints. Even though they look like functions, they are exposed via POST requests.
  - **Authentication:** Check auth() session at the start of every action.
  - **Authorization:** Verify user permissions before mutation.
  - **CSRF:** Next.js handles this automatically for Server Actions, but ensure headers are not manipulated if using custom fetchers.

10. Conclusion

This PRD represents the convergence of modern tooling and disciplined architecture. By combining **React 19's performance**, **Next.js 15's infrastructure**, **FSD's scalability**, and **AI-native workflows**, we create a development environment that is robust, maintainable, and

ready for the future of automated software engineering.

Any AI agent reading this document is now authorized to act as a **Senior React Architect**. Proceed with implementation, adhering strictly to the constraints defined herein.

Document Status: FINAL
Version: 2025.1.0
Authority: Root Architect
Compliance: Mandatory for all AI Agents and Human Developers.

Works cited

1. React 19 Best Practices: Write Clean, Modern, and Efficient React Code - DEV Community, accessed January 1, 2026, https://dev.to/jay_sarvaiya_reactjs/react-19-best-practices-write-clean-modern-and-efficient-react-code-1beb
2. Unlocking the Power of React 19: 10 Best Practices for Modern Development, accessed January 1, 2026, https://javascript.plainenglish.io/unlocking-the-power-of-react-19-10-best-practices-for-modern-development-fcbc28a348a5
3. React 19 Features: Complete Guide to the Latest Updates | ItsIndianGuy, accessed January 1, 2026, https://www.itsindianguy.in/blog/react-19-features-complete-guide-2025
4. Getting Started: Error Handling - Next.js, accessed January 1, 2026, https://nextjs.org/docs/app/getting-started/error-handling
5. Next.js 15, accessed January 1, 2026, https://nextjs.org/blog/next-15
6. Getting Started: Updating Data - Next.js, accessed January 1, 2026, https://nextjs.org/docs/app/getting-started/updating-data
7. Next.js Best Practices in 2025 — Build Faster, Cleaner, Scalable Apps | by Goutam Singha, accessed January 1, 2026, https://medium.com/@GoutamSingha/next-js-best-practices-in-2025-build-faster-cleaner-scalable-apps-7efbad2c3820
8. Configuration: TypeScript - Next.js, accessed January 1, 2026, https://nextjs.org/docs/app/api-reference/config/typescript
9. Feature Sliced Design in Next JS. What is FSD and why is it needed ? | by Sriramanvellingiri, accessed January 1, 2026, https://medium.com/@sriramanvellingiri/feature-sliced-design-in-next-js-7d20be4338de
10. Let's Learn Feature-Sliced Design (FSD) - DEV Community, accessed January 1, 2026, https://dev.to/nyaomaru/lets-learn-feature-sliced-design-fsd-15bb
11. Usage with Next.js | Feature-Sliced Design, accessed January 1, 2026, https://feature-sliced.design/docs/guides/tech/with-nextjs
12. How to deal with NextJS App Router and FSD problem - DEV Community, accessed January 1, 2026,

https://dev.to/m_midas/how-to-deal-with-nextjs-using-feature-sliced-design-4c67
13. The Battle-Tested NextJS Project Structure I Use in 2025. | by Burpdeepak - Medium, accessed January 1, 2026, https://medium.com/@burpdeepak96/the-battle-tested-nextjs-project-structure-i-use-in-2025-f84c4eb5f426
14. Zustand and TanStack Query: The Dynamic Duo That Simplified My React State Management | by Blueprintblog | JavaScript in Plain English, accessed January 1, 2026, https://javascript.plainenglish.io/zustand-and-tanstack-query-the-dynamic-duo-that-simplified-my-react-state-management-e71b924efb90
15. Zustand + React Query: A New Approach to State Management | by Freeyeon | Medium, accessed January 1, 2026, https://medium.com/@freeyeon96/zustand-react-query-new-state-management-7aad6090af56
16. Separating Concerns with Zustand and TanStack Query - Volodymyr Rudyi, accessed January 1, 2026, https://volodymyrrudyi.com/blog/separating-concerns-with-zustand-and-tanstack-query/
17. Zustand vs tanstack query : r/reactjs - Reddit, accessed January 1, 2026, https://www.reddit.com/r/reactjs/comments/1mugweq/zustand_vs_tanstack_query/
18. Federated State Done Right: Zustand, TanStack Query, and the Patterns That Actually Work, accessed January 1, 2026, https://dev.to/martinrojas/federated-state-done-right-zustand-tanstack-query-and-the-patterns-that-actually-work-27c0
19. Tailwind CSS v4.0, accessed January 1, 2026, https://tailwindcss.com/blog/tailwindcss-v4
20. What to expect from Tailwind CSS v4.0 | by Onix React - Medium, accessed January 1, 2026, https://medium.com/@onix_react/what-to-expect-from-tailwind-css-v4-0-9e8b4b98c6b4
21. TSConfig Reference - Docs on every TSConfig option - TypeScript, accessed January 1, 2026, https://www.typescriptlang.org/tsconfig/
22. The most strict TypeScript tsconfig mode - Gist - GitHub, accessed January 1, 2026, https://gist.github.com/dilame/32709f16e3f8d4d64b596f5b19d812e1
23. TSConfig Option: noUncheckedIndexedAccess - TypeScript, accessed January 1, 2026, https://www.typescriptlang.org/tsconfig/noUncheckedIndexedAccess.html
24. Testing: Vitest - Next.js, accessed January 1, 2026, https://nextjs.org/docs/app/guides/testing/vitest
25. Mastering React Testing with Vitest 2.0: A Comprehensive Guide Part 1 | Vivek Patel, accessed January 1, 2026, https://patelvivek.dev/blog/testing-react-vitest
26. WCAG in 2025: Trends, Pitfalls & Practical Implementation | by Alendennis - Medium, accessed January 1, 2026, https://medium.com/@alendennis77/wcag-in-2025-trends-pitfalls-practical-imple

mentation-8cdc2d6e38ad
27. Web Content Accessibility Guidelines (WCAG) 2.2 - W3C, accessed January 1, 2026, https://www.w3.org/TR/WCAG22/
28. Accessibility audit with react-axe and eslint-plugin-jsx-a11y | Articles - web.dev, accessed January 1, 2026, https://web.dev/articles/accessibility-auditing-react
29. eslint-plugin-jsx-a11y - NPM, accessed January 1, 2026, https://www.npmjs.com/package/eslint-plugin-jsx-a11y
30. Rules | Cursor Docs, accessed January 1, 2026, https://cursor.com/docs/context/rules
31. Windsurf AI Rules: A Guide to Windsurf.ai Prompting | UI Bakery Blog, accessed January 1, 2026, https://uibakery.io/blog/windsurf-ai-rules
32. What is Meta Prompting? - IBM, accessed January 1, 2026, https://www.ibm.com/think/topics/meta-prompting
33. Meta-Prompting: LLMs Crafting & Enhancing Their Own Prompts | IntuitionLabs, accessed January 1, 2026, https://intuitionlabs.ai/articles/meta-prompting-llm-self-optimization
34. Routing: Error Handling - Next.js, accessed January 1, 2026, https://nextjs.org/docs/14/app/building-your-application/routing/error-handling