Architectural Convergence: Strategic Frameworks for Transposing React Native Storybook Ecosystems to React Web

(Context-Aware Analysis for `av-ui-library`)

1. Executive Overview: The Cross-Platform Imperative

The software development landscape has witnessed a profound convergence between mobile and web ecosystems, driven by the necessity for operational efficiency and design consistency. The objective to translate an existing Storybook library architected for React Native (specifically utilizing the Expo runtime) into a React Web environment—represents a classic architectural challenge in this domain: the reconciliation of the Native Bridge with the Document Object Model (DOM).

The source repository, characterized by its reliance on React Native and likely leveraging libraries such as react-native-paper, exists within a paradigm where User Interface (UI) primitives are strictly defined (e.g., `<View>`, `<Text>`, `<Image>`). The target environment, the modern web, operates on a fundamentally different set of primitives (semantic HTML tags like `<div>`, `<span>`, `<header>`) and layout engines (CSS Flexbox/Grid versus the Yoga layout engine used in React Native). This report provides an exhaustive analysis of three distinct architectural strategies to achieve this translation, ranging from runtime emulation to compile-time optimization. These strategies are evaluated not merely on their immediate feasibility, but on their long-term implications for performance, maintainability, and scalability within an enterprise context.

1. **Runtime Adaptation (React Native for Web):** A low-friction implementation leveraging aliasing and compatibility layers to render Native code in the browser via React DOM.
2. **Universal Design System (Tamagui / Unistyles / Gluestack):** A refactoring-heavy approach that adopts "Universal" libraries capable of optimizing output for specific targets (Native vs. Web) at build time.
3. **Framework-Agnostic Compilation (Mitosis):** A transformative approach utilizing static analysis to compile a single source of truth into distinct, framework-native codebases (React Native and React DOM) with zero runtime overhead.
4. **Manual Counterpart Creation (The "Twin" Strategy):** A pragmatic, labor-intensive approach involving the creation of distinct, parallel React components for the web that mirror the API and behavior of their React Native counterparts, allowing for maximum platform optimization without abstraction layers.

Furthermore, this analysis extends into the critical supporting infrastructure required to sustain such a migration, specifically the implementation of Monorepo architectures (Turborepo, Nx) and the integration of modern build tools (Vite) within the Storybook 8 ecosystem.

---

2. Strategy 1: Runtime Adaptation via React Native for Web

The most immediate and least invasive path for migrating a React Native Storybook to the web is the adoption of the **Runtime Adaptation** model, primarily facilitated by the react-native-web library. This strategy posits that the codebase need not change its source

primitives; rather, the runtime environment should adapt to interpret those primitives correctly for the browser.

2.1 The Mechanics of Runtime Translation

React Native for Web (RNW) operates as an interoperability layer. When a developer writes `<View>` in React Native, the bundler is instructed via aliasing to resolve this import not to the native iOS/Android implementation, but to a web-compatible implementation located in react-native-web.

2.1.1 The Rendering Pipeline and Atomic CSS

The core of RNW's web implementation is its styling engine. React Native relies on the Yoga layout engine to calculate flexbox layouts. RNW emulates this behavior in the browser by generating **Atomic CSS**. Unlike traditional CSS where a class might contain multiple declarations (e.g., `.card { padding: 10px; background: white; }`), Atomic CSS generates a unique class for every single style property-value pair. 1

For example, a `<View style={{ padding: 10, flexDirection: 'row' }}>` might render in the DOM as: `<div class="r-padding-10 r-flexDirection-row">` This approach ensures determinism and reduces the specificity wars common in CSS development. However, it introduces a runtime cost: the JavaScript thread must calculate these styles and inject them into the document head during hydration. For large Storybook libraries with complex, interactive components, this can lead to a phenomenon known as "div soup," where the DOM depth becomes excessive due to the need to emulate native view nesting. 2

Context Note for av-ui-library: The `AVCart` component, with its deep nesting of `View` elements (SeriesContainer -> SeriesHeader -> Ticket -> InfoContainer), is a primary candidate for this "div soup" phenomenon. This can lead to hydration performance issues on the web if not optimized.

2.1.2 Integration with React Native Paper

Given the reliance on react-native-paper, this strategy benefits from strong library support. React Native Paper (v 5 and above) explicitly supports React Native Web and implements Material Design 3 (Material You) standards. 4 However, implementation nuances exist. Paper's support for "Material You" involves dynamic color adaptations that may require specific web configurations to mirror the Android implementation correctly. 4 The library exposes a version prop in its theme configuration, allowing developers to toggle between Material Design 2 and 3 . 7 **Architectural Implication:** While react-native-paper claims web support, it treats the web as a secondary target. Research indicates that specific interactive elements, such as the Floating Action Button (FAB) and keyboard avoidance behaviors, frequently exhibit visual regressions on the web (e.g., z-index failures or layout overlaps) that do not manifest on native platforms. 8 This suggests that while the *code* runs, the *fidelity* requires manual quality assurance and CSS overrides.

Web Asset Management (Fonts & Icons): A critical gap in the "Runtime Adaptation" strategy is asset loading. React Native links fonts (like `MaterialCommunityIcons` used by Paper) natively. On the web, these must be injected via CSS `@font-face`. * **Action:** The `av-ui-library` must implement a strategy to inject the MaterialCommunityIcons font file into the web head (e.g., inside `.storybook/preview.js` or a global `AppWrapper`). Without this, web components will render "tofu" (empty squares) instead of icons.

2.2 Bundler Configuration: The Vite Transition

Historically, React Native Web relied heavily on Webpack. However, the modern Storybook 8 ecosystem favors **Vite** for its superior development server performance. Integrating RNW with Vite requires specific plugin architectures to handle the flow of CommonJS and ESM modules which React Native libraries often mix indiscriminately.

2.2.1 Vite-Plugin-React-Native-Web

To operationalize this strategy in a modern stack, the vite-plugin-react-native-web is the standard middleware. It manages the aliasing of react-native to react-native-web and handles the transpilation of specific node_modules that ship uncompiled ES 6 code—a common practice in the React Native ecosystem but a breaking pattern for standard web bundlers. 9

Configuration Example for Storybook (.storybook/main.ts):

The configuration must explicitly invoke the React framework and apply the RNW aliases.

```
import type { StorybookConfig } from "@storybook/react-vite";

const config: StorybookConfig = {
  framework: {
    name: "@storybook/react-vite",
    options: {
      builder: {
        viteConfigPath: '.storybook/vite.config.ts',
      },
    },
  },
  addons: [
    "@storybook/addon-react-native-web", // Critical addon for alias handling
    "@storybook/addon-essentials"
  ],
  // Explicitly separate the native stories if necessary
  stories: ["../src/**/*.stories.@(js|jsx|ts|tsx)"],
};
export default config;
```

This configuration leverages @storybook/addon-react-native-web to inject the necessary Babel plugins and Webpack/Vite aliases automatically. 1 1

2.3 Pros and Cons: The Runtime Adaptation Model

Dimension	Analysis
Velocity	**High.** This is the path of least resistance. It honors the "Write Once, Run Anywhere" philosophy by requiring near-zero changes to the component source code. 1
Fidelity	**Variable.** While static components render well, complex interactions involving gestures or native APIs (Haptics, Camera) require distinct web polyfills or conditional logic, breaking the abstraction. 2
SEO & Semantics	**Low.** RNW renders `<View>` as `<div>` by default. Accessibility requires manual mapping (e.g., `accessibilityRole="heading"` -> `<h1>`). Without this, the web output is opaque to screen readers and search engines. 1

Dimension	Analysis
Performance	**Moderate.** The runtime overhead of Atomic CSS generation and the large bundle size (rendering engine + logic) makes this suboptimal for high-performance public-facing websites, though acceptable for internal dashboards or B2B apps. 2

---

3. Strategy 2: The Universal Design System (Tamagui / Unistyles / Gluestack)

The second architectural approach rejects the notion of "porting" or "adapting" code. Instead, it advocates for **Universal Design**, where components are constructed using primitives explicitly designed to compile into optimized Native views on mobile and optimized CSS/HTML on the web. This approach addresses the performance and semantic limitations of Strategy 1 by moving the abstraction cost from runtime to build-time.

3.1 The Rise of the Optimizing Compiler: Tamagui

Tamagui represents the vanguard of this strategy. Unlike RNW, which is a runtime library, Tamagui functions partially as a compiler. It utilizes static analysis to read the React component tree, flatten the styling logic, and extract it into pure CSS files for the web build. 1 4

3.1.1 Mechanism of Action

When a developer uses a Tamagui `<Stack>` or `<Button>`, the compiler attempts to resolve the styles at build time.

- **Native Output:** The compiler outputs standard React Native Views, preserving the exact behavior expected by the native bridge.
- **Web Output:** The compiler removes the JavaScript style objects entirely, replacing them with hashed CSS class names. This significantly reduces the main thread workload, addressing the "div soup" and hydration performance issues inherent in RNW. 1 5

3.1.2 Integration with Storybook

Migrating an existing react-native-paper library to Tamagui is a non-trivial refactor. It requires replacing the fundamental building blocks.

1. **Provider Setup:** The Storybook preview must be wrapped in a TamaguiProvider, which injects the design tokens (colors, spacing, themes) into the context. 1 5
2. **Babel Configuration:** The @tamagui/babel-plugin is mandatory. It intercepts the component code during the bundle process to perform the static extraction. 1 7

3.2 The C++ Powered Styling Engine: Unistyles 3.0

For teams that prefer to author their own components rather than adopting a UI kit like Tamagui, **Unistyles** offers a compelling "middle way." It is a styling library that supersedes the deprecated react-native-extended-stylesheet and standard StyleSheet APIs. 1 9

3.2.1 The JSI Advantage

Unistyles leverages the JavaScript Interface (JSI) to bind directly to the native C++ layer.

- **Synchronous Access:** Unlike the async bridge of old React Native, Unistyles allows synchronous access to device capabilities and theme preferences.
- **Web Parity via CSS:** In version 3.0, Unistyles introduced experimental support for mapping JSI logic to **CSS Media Queries** on the web.[21] This is a critical architectural distinction: instead of using JavaScript event listeners to handle window resizing (which causes layout thrashing), Unistyles generates native CSS media queries, ensuring the web component is responsive even before React hydrates.[23]

3.2.2 Migration from Legacy Stylesheets

The research identifies react-native-extended-stylesheet as a deprecated library that may exist in older repos.[24] Unistyles provides a direct migration path because it mimics the StyleSheet.create syntax.

- **Refactoring Pattern:** The StyleSheet.create calls remain largely the same, but dynamic values are handled via functions.

```
// Legacy
const styles = EStyleSheet.create({ width: '100%' });

// Unistyles
const styles = StyleSheet.create(theme => ({
  width: '100%',
  backgroundColor: theme.colors.background
}));
```

This API parity makes Unistyles the most efficient refactoring target for maintaining the existing code structure while gaining modern web capabilities.[19]

Context Note for av-ui-library: The repository heavily uses `react-native-extended-stylesheet` (e.g., `EStyleSheet.build` in `App.js`, `EStyleSheet.create` in `AVRadioCards`). This library is deprecated. **Unistyles** provides a direct migration path as shown above. Refactoring `AVRadioCards` to use Unistyles would immediately improve web performance by generating native CSS media queries instead of relying on JS-driven layout changes.

3.3 Gluestack UI (The NativeBase Evolution)

Gluestack UI (formerly NativeBase) offers another Universal option. It focuses on accessibility and provides unstyled "headless" components that can be styled using Tailwind-like utility props.[27] While robust, community benchmarks and sentiment analysis suggest that Gluestack v2 suffered from performance issues, leading to a rewrite in v3 to address the "bridge traffic" caused by excessive styling props.[29] It is a viable alternative if the team prefers a utility-first styling approach (similar to Tailwind) over the object-oriented approach of Unistyles.

3.4 Pros and Cons: The Universal Design Model

Dimension	Analysis
Performance	**Superior.** Tamagui and Unistyles effectively eliminate the runtime styling cost on the web via compilation and CSS extraction. 1 4
Responsiveness	**Native-Tier.** The ability to generate CSS Media Queries (Unistyles) or utilize atomic CSS (Tamagui) ensures resize behavior is instant and jank-free, unlike JS-driven responsiveness. 2 3
Refactoring Cost	**High.** This requires replacing react-native-paper components with new primitives. It is a rewrite, not a port. Every story in Storybook will likely need adjustment. 3 1
Config Complexity	**High.** Setting up the compilers, Babel plugins, and Webpack/Vite loaders for these advanced libraries is significantly more complex than the standard RNW setup. 3 0

---

4. Strategy 3: Framework-Agnostic Compilation (Mitosis)

The third strategy represents a radical departure from the "Runtime" (Strategy 1) vs. "Universal" (Strategy 2) dichotomy. **Mitosis** introduces a "Compile-Time" architecture where the source of truth is neither React Native nor React Web, but a meta-syntax that compiles to both.

4.1 The "Write Once, Compile to Anything" Architecture

Mitosis parses a specific subset of JSX (often called "Lite JSX") into a JSON-based Abstract Syntax Tree (AST). This intermediate representation (the Mitosis Component JSON) is then fed into serializers that output code for different frameworks. 3 2

- **Input:** A `.lite.tsx` file using a restricted set of hooks (`useStore`, `useMetadata`).
- **Output 1 (Web):** A React component using `<div>`, `className`, and CSS modules.
- **Output 2 (Mobile):** A React Native component using `<View>`, `style`, and `StyleSheet`.

4.2 Architectural Implications for Storybook

Adopting Mitosis implies treating the component library as a build artifact rather than source code.

1. **Source Logic:** The team writes code in components/src/MyComponent.lite.tsx.
2. **Build Pipeline:** A watcher process compiles this file into components/dist/web/MyComponent.js and components/dist/native/MyComponent.js.
3. **Storybook Consumption:** The Storybook instance must be configured to import from the dist folders depending on the active environment.

4.2.1 Metadata and Platform Specifics

Mitosis handles platform divergence via the useMetadata hook. This allows developers to annotate the component with instructions for specific compilers. For example, one might tag a component to use a specific specialized import when compiling to React Native, while using a standard HTML tag for the web. 3 2

4.3 Pros and Cons: The Compiler Model

Dimension	Analysis
Purity	**Maximum.** The web output is pure, semantic HTML. The mobile output is pure Native components. There is no runtime bridge or aliasing. This results in the absolute smallest bundle sizes and best possible performance on both platforms. 3 4
Flexibility	**Extreme.** The same source code could theoretically compile to Vue, Svelte, or Angular in the future, providing immense long-term value for large enterprises with heterogeneous tech stacks. 3 3
Developer Experience	**Mixed.** The developer is restricted to a subset of React. You cannot simply use any NPM package or arbitrary JavaScript logic; it must be compatible with the Mitosis parser. Debugging generated code can be opaque. 3 2
Migration Effort	**Extreme.** This requires converting the entire react-native-paper based library into Mitosis syntax. It effectively means rewriting the library from scratch, using Mitosis as the authoring language.

---

5. Strategy 4: Manual Counterpart Creation (The "Twin" Strategy)

While the previous strategies focus on code sharing or compilation, the **Manual Counterpart Creation** strategy accepts the fundamental differences between platforms and addresses them through duplication rather than abstraction. This approach involves creating a dedicated `.web.tsx` file for every component, implementing the UI using standard React DOM primitives (`<div>`, `<span>`, CSS Modules/Tailwind) while maintaining an identical prop interface to the React Native version.

5.1 The Pattern: Platform-Specific Extensions

React Native's bundler (Metro) and modern web bundlers (Webpack/Vite) support platform-specific file extensions. This allows for a clean separation of concerns where the consumer imports `Button`, and the bundler resolves the correct file based on the target platform.

- `Button.tsx` (or `Button.native.tsx`): Contains the React Native implementation using `<View>` and `StyleSheet`.
- `Button.web.tsx`: Contains the React DOM implementation using `<div>` and CSS.

5.1.1 Implementation Workflow

1. **Interface Definition:** Define a shared TypeScript interface for the component props in a separate file (e.g., `Button.types.ts`). This ensures that both the web and native

versions adhere to the same contract.

2. **Native Implementation:** Build the mobile component using standard React Native patterns.
3. **Web Implementation:** Build the web component using pure HTML/CSS or a web-first library (like Material UI or Headless UI).
4. **Barrel Export:** In `index.ts`, simply export the component. The bundler handles the rest.

```
// Button.types.ts
export interface ButtonProps {
  label: string;
  onPress: () => void;
  variant?: 'primary' | 'secondary';
}

// Button.native.tsx
import { TouchableOpacity, Text } from 'react-native';
import { ButtonProps } from './Button.types';

export const Button = ({ label, onPress }: ButtonProps) => (
  <TouchableOpacity onPress={onPress}>
    <Text>{label}</Text>
  </TouchableOpacity>
);

// Button.web.tsx
import { ButtonProps } from './Button.types';
import './Button.css'; // Standard CSS

export const Button = ({ label, onPress }: ButtonProps) => (
  <button className="btn" onClick={onPress}>
    {label}
  </button>
);
```

5.2 Pros and Cons: The Twin Strategy

Dimension	Analysis
Optimization	**Maximum.** Since the web code is written specifically for the DOM, it can leverage all web-specific features (CSS Grid, hover states, accessibility attributes) without any bridge overhead or emulation quirks.
Maintainability	**Low.** This violates the DRY (Don't Repeat Yourself) principle. Every feature request or bug fix potentially requires changes in two separate files. Divergence between platforms is a constant risk.
Dependencies	**Decoupled.** The web version doesn't need `react-native-web` or heavy compatibility libraries. It can use standard web packages, keeping the bundle size minimal.
Migration Effort	**High.** It requires writing every component twice. However, for complex components where `react-native-web` fails (like maps or complex gestures), this might be the *only* viable option.

6. Infrastructure and Tooling: Supporting the Migration

Regardless of the chosen strategy, the operational success of this migration depends on the supporting infrastructure. The research strongly supports the implementation of a **Monorepo** and a unified **Navigation** strategy.

6.1 Monorepo Architectures: Turborepo vs. Nx

Moving a standalone library to a monorepo structure enables the coexistence of the Native App, the Web App, and the Shared UI Library (Storybook).

6.1.1 Turborepo Implementation

Turborepo is favored for its simplicity and pipeline caching capabilities. A typical structure for this migration would be:

- apps/native: The Expo application.
- apps/web: The Next.js or Vite React web application.
- packages/ui: The shared UI library containing the components and the Storybook configuration. [3] [5]

Configuration: Turborepo requires careful handling of "hoisted" dependencies. To make React Native work within a monorepo, the metro.config.js in the native app must be patched to resolve node_modules from the workspace root, as Metro does not support symlinks by default. [3] [7]

6.1.2 Nx Implementation

Nx offers a more opinionated, "batteries-included" approach. It excels at dependency graph analysis and can automatically detect that changes in the ui library should trigger a rebuild of the web app.

- **Project Crystal:** Nx's new inference engine simplifies the configuration, potentially automating the Metro/Webpack wiring that requires manual scripting in Turborepo. [3] [8]
- **Shared Libraries:** Nx allows for the creation of "local libraries" that can be imported via simple paths (e.g., @my-org/ui), handling the TypeScript path mapping automatically. [3] [9]

6.2 The Animation Conundrum

Animation represents the deepest fissure between the platforms.

- **React Native:** Relies on Animated or react-native-reanimated (running on the UI thread via JSI).
- **Web:** Relies on CSS Transitions or Main Thread JS (framer-motion).

Reanimated 4.0 has introduced significant strides in bridging this gap by implementing a CSS Transition API that mirrors web standards, allowing for high-performance animations on web that don't block the JS thread. [4] [3] However, standard Animated API usage often results in poor web performance.

- **Recommendation:** If using Strategy 2 (Universal), leverage the library's animation driver (e.g., @tamagui/animations-react-native) which abstracts this complexity. If using Strategy

1, prefer simple layout animations or accept the performance penalty of JS-driven animations on the web. 4 4

Context Note for av-ui-library: The `AVSwitch` component currently uses the native `Animated` API (`Animated.Value`, `Animated.spring`). For a high-fidelity web port, `AVSwitch` should be refactored to use **CSS Transitions** (via Unistyles or Tamagui) or `react-native-reanimated` (which has better web support) instead of the legacy `Animated` API.

6.3 Navigation Decoupling

Components like `AVCart` often imply navigation flows (e.g., "Proceed to Checkout"). Hardcoding `react-navigation` (Native) or `react-router` (Web) creates a platform lock-in.

- **Link Injection Pattern:** The library should expose a `LinkComponent` prop in its provider or individual components.
  - *Native App* passes: `<Link to="/checkout" component={TouchableOpacity} />`
  - *Web App* passes: `<Link href="/checkout" component={a} />`
  - The library simply uses `props.LinkComponent` without knowing which router is active.

---

6. Migration Roadmaps and Recommendations

The choice of strategy dictates the roadmap. Below are the prescribed paths based on the analysis.

6.1 The "Strangler Fig" Migration (Recommended)

This roadmap combines Strategy 1 and Strategy 2 to balance immediate delivery with long-term quality. It avoids the risk of a total rewrite while steadily improving the codebase.

Phase 1: The Monorepo Lift-and-Shift

1. Initialize a **Turborepo** workspace using a starter template. 3 5
2. Move the existing Expo code into apps/native and extract the components into packages/ui.
3. Ensure metro.config.js correctly resolves the workspace packages. 3 7

Phase 2: The Web shim (Strategy 1)

1. Install react-native-web and @storybook/react-vite in the packages/ui workspace.
2. Configure .storybook/main.ts with vite-plugin-react-native-web to alias the react-native-paper components. 1 0
3. **Outcome:** You now have a functioning Web Storybook. It may have "div soup" and some visual bugs, but it is visible and testable.

Phase 3: Universal Refactoring (Strategy 2)

1. Install **Unistyles** alongside the existing styling solution.
2. Select a low-complexity component (e.g., a Button or Badge).
3. Refactor strictly that component to use Unistyles instead of StyleSheet.
4. Verify that Unistyles is generating CSS media queries on the web output. 2 1
5. Repeat this process iteratively. This gradually replaces the runtime-heavy RNW styling with optimized Universal styling without breaking the app.

Context Note for av-ui-library: For Phase 3, we should prioritize refactoring `AVCart` to break down its monolithic structure into smaller, semantic web components. Also, target the removal of `react-native-extended-stylesheet` from `App.js` and `AVRadioCards` by migrating to Unistyles.

6.2 The Total Rewrite (For Long-Term Products)

If the current codebase is legacy or technically indebted, a clean break may be preferable.

1. Initialize a **Tamagui** Monorepo. 4 6
2. Port components one by one from the old repo to the new packages/ui folder, rewriting them to use Tamagui primitives (`<Stack>`, `<Text>`) instead of react-native-paper.
3. This approach yields the highest possible quality but halts feature development during the migration. 1 4

---

7. Comparison Summary

Feature	Strategy 1: RNW Shim	Strategy 2: Universal (Tamagui/Unistyles)	Strategy 3: Mitosis Compiler	Strategy 4: Twin Components
Integration Difficulty	**Low** (Config only)	**High** (Code rewrite required)	**Very High** (New syntax adoption)	**Moderate** (Parallel dev)
React Native Paper	Supported (Native)	Not Supported (Must replace)	Not Supported (Must rewrite)	**N/A** (Web uses different lib)
Web Performance	Moderate (Runtime CSS)	Excellent (Static CSS)	Best (Native DOM)	**Best** (Native DOM)
SEO / Accessibility	Poor (Divs)	Good (Semantic Tags)	Best (Semantic Tags)	**Best** (Semantic Tags)
Build Complexity	Low (Vite/Webpack)	High (Custom Compilers)	High (AST Serializers)	**Low** (Standard Bundlers)

Final Recommendation

For an existing React Native/Expo Storybook utilizing react-native-paper, **Strategy 1 (Runtime Adaptation via React Native for Web)** is the only viable option for rapid translation. It preserves the react-native-paper investment.

However, the team must acknowledge the performance ceiling of this approach. It is strongly recommended to pair this with **Unistyles** (Strategy 2) for *new* components and iterative refactoring. This hybrid model leverages the ease of the shim for legacy code while building a high-performance, CSS-native future for the application via the Unistyles JSI engine.

Additionally, for critical components where performance or SEO is paramount and `react-native-web` falls short, **Strategy 4 (Twin Components)** should be employed as a surgical intervention. This allows for native-quality web experiences without rewriting the entire library.

The implementation should be housed within a **Turborepo** to enforce strict boundary separation between the Native, Web, and UI concerns.

---

8. Strategic Recommendations for av-ui-library Evolution

Beyond the web migration, the following recommendations address the architecture, organization, and scalability of the av-ui-library project itself.

8.1 Folder Structure & Organization

The current structure (`stories/Component/Component.jsx`) mixes source code with documentation. A more scalable "Package-Based" structure is recommended to separate concerns and facilitate future extraction.

Recommended Structure:

```
src/
  components/
    Button/
      index.tsx        (Export barrel)
      Button.tsx       (Source code)
      Button.styles.ts (Styles - ideally Unistyles)
      Button.test.tsx  (Unit tests)
    Cart/
      ...
stories/
  Button.stories.tsx   (Storybook documentation)
  Cart.stories.tsx
```

- **Benefit:** This separates the *library code* (src) from the *documentation/dev environment* (stories), making it easier to publish `src` as an NPM package later.

8.2 Tech Stack Modernization

1. **Strict TypeScript Adoption:** While `tsconfig.json` exists, many files are `.jsx`. Migrating to `.tsx` and enforcing strict typing for props (removing `prop-types`) will significantly reduce runtime errors and improve developer experience (IntelliSense).
2. **Testing Framework:** The current repo lacks a visible testing setup. Integrating **Jest** and **React Native Testing Library (RNTL)** is critical for preventing regressions during the web migration.
    - *Action:* Add `jest.config.js` and write a basic snapshot test for `AVButton`.
3. **Visual Regression Testing (Chromatic):** The repository includes a `chromatic` script, which is excellent. To fully leverage this for migration, configure Chromatic to take snapshots in both **mobile** (simulating Native) and **desktop** (Web) viewports. This is the only automated way to catch if a CSS change for Web accidentally breaks the Native layout.
4. **Legacy Pattern Cleanup (Hoverable.js):** The repository contains `Hoverable.js` and `HoverState.js`, which manually attach event listeners to the DOM. This is a legacy pattern that causes performance issues ("zombie listeners").

- *Action:* Deprecate `Hoverable.js`. Modern React Native Web supports hover natively via the `Pressable` component's `hoverStyle` prop, or via CSS if switching to Unistyles.

8.3 Best Practices & Quality

1. **Accessibility (a11y) Enforcement:** While `stories/assets` mentions accessibility, it should be enforced via linting.
   - *Action:* Install `eslint-plugin-jsx-a11y` and `eslint-plugin-react-native-a11y` to catch missing accessibility labels during development.

2. **Atomic Design:** The `AVCart` is a "Organism" composed of "Molecules" (Ticket, Header). Explicitly categorizing components (Atoms, Molecules, Organisms) in the folder structure can help developers understand the complexity and dependency chain of each component.

8.4 Scalability

1. **Build System & Package Distribution:** Currently, the library seems to be consumed as source. To scale as a true library, it should be bundled.
   - *Recommendation:* Use **Rollup** or **Vite Library Mode** to bundle the `src` folder into CommonJS (CJS) and ES Modules (ESM) formats.
   - *Conditional Exports:* Update `package.json` to implement **Conditional Exports**. This ensures consumers automatically get the correct file for their platform:

```
"exports": {
  ".": {
    "react-native": "./dist/index.native.js",
    "import": "./dist/index.web.js",
    "require": "./dist/index.web.cjs"
  }
}
```

1. **Monorepo Adoption:** As mentioned in the research, moving this repo into a **Turborepo** is the single best move for scalability. It allows you to have:
   - `packages/ui` (This library)
   - `apps/docs` (Storybook static build)
   - `apps/example` (A test app consuming the built package)

Works cited

1. React Native and React Native Web : A New Choice for Cross-Platform Development | by Tianya School | Nov, 2025 | Medium, accessed November 18, 2025, https://medium.com/@tianyaschool/react-native-and-react-native-web-a-new-choice-for-cross-platform-development-bf96f1e12077
2. React Native for Web: Advantages, Disadvantages, and Use Cases - Apiko, accessed November 18, 2025, https://apiko.com/blog/react-native-for-web/
3. Migrating React and Native Apps To React Native - Callstack, accessed November 18, 2025, https://www.callstack.com/blog/migration-to-react-native
4. React Native Paper, accessed November 18, 2025, https://reactnativepaper.com/
5. callstack/react-native-paper: Material Design for React Native (Android & iOS) - GitHub, accessed November 18, 2025, https://github.com/callstack/react-native-paper

6. Using on the Web | React Native Paper, accessed November 18, 2025, https://callstack.github.io/react-native-paper/docs/guides/react-native-web/
7. Introducing v5 with Material You | React Native Paper, accessed November 18, 2025, https://callstack.github.io/react-native-paper/docs/guides/migration-guide-to-5.0/
8. can I use React Native Paper for the normal reactjs web application? - Stack Overflow, accessed November 18, 2025, https://stackoverflow.com/questions/71380201/can-i-use-react-native-paper-for-the-normal-reactjs-web-application
9. Run Storybook with NX Expo and React Native Paper - DEV Community, accessed November 18, 2025, https://dev.to/typescriptteatime/run-storybook-with-nx-expo-and-react-native-paper-4l8l
10. React Native Web with Vite - DEV Community, accessed November 18, 2025, https://dev.to/dannyhw/react-native-web-with-vite-1jg5
11. React Native Web addon for Storybook, accessed November 18, 2025, https://storybook.js.org/addons/@storybook/addon-react-native-web
12. storybookjs/addon-react-native-web: Build react-native-web projects in Storybook for React - GitHub, accessed November 18, 2025, https://github.com/storybookjs/addon-react-native-web
13. Part 1 — Converting react native app to react-native-web (react PWA) in monorepo architecture | by Zia ul Rehman | Medium, accessed November 18, 2025, https://medium.com/@ziaulrehman/part-1-converting-react-native-app-to-react-native-web-react-pwa-in-monorepo-architecture-34b43cad74b8
14. React Native Component Libraries: Accelerate Your Development | by Facundo Acosta | The Tech Collective | Medium, accessed November 18, 2025, https://medium.com/the-tech-collective/react-native-component-libraries-accelerate-your-development-8ace8477a61f0
15. Installation - Tamagui, accessed November 18, 2025, https://tamagui.dev/docs/intro/installation
16. style library and UI kit for React Native and React Web - Tamagui, accessed November 18, 2025, https://tamagui.dev/ui/intro
17. Getting Started with Tamagui: A Complete Guide to Modern React Native Styling, accessed November 18, 2025, https://dev.to/cathylai/getting-started-with-tamagui-a-complete-guide-to-modern-react-native-styling-3fff
18. Integrate Tamagui with existing Expo React Native app - Stack Overflow, accessed November 18, 2025, https://stackoverflow.com/questions/75154160/integrate-tamagui-with-existing-expo-react-native-app
19. Migration guide | react-native-unistyles, accessed November 18, 2025, https://www.unistyl.es/v3/start/migration-guide
20. Introduction | react-native-unistyles, accessed November 18, 2025, https://www.unistyl.es/v3/start/introduction/
21. Web support - Unistyles 3.0, accessed November 18, 2025, https://v2.unistyl.es/reference/web-support/
22. Media Queries - Unistyles 3.0, accessed November 18, 2025, https://v2.unistyl.es/reference/media-queries/
23. Unistyles 3.0: Beyond React Native StyleSheet - Expo, accessed November 18, 2025, https://expo.dev/blog/unistyles-3-0-beyond-react-native-stylesheet
24. vitalets/react-native-extended-stylesheet - GitHub, accessed November 18, 2025, https://github.com/vitalets/react-native-extended-stylesheet
25. @tokenstreet/react-native-extended-stylesheet - NPM, accessed November 18, 2025, https://www.npmjs.com/package/@tokenstreet/react-native-extended-stylesheet
26. Style Your React Native App with Unistyles - Agence Premier Octet, accessed November 18, 2025, https://www.premieroctet.com/blog/en/style-your-react-native-app-with-unistyles
27. Choosing the Right UI Library for My React Native App (Need Advice) : r/reactnative - Reddit, accessed November 18, 2025, https://www.reddit.com/r/reactnative/comments/1o8x7j3/choosing_the_right_ui_library_for_my_react_native/

28. Manual Migration to gluestack-ui v3, accessed November 18, 2025, https://gluestack.io/ui/docs/guides/more/upgrade-to-v3
29. What are your thoughts on gluestack-ui? : r/reactnative - Reddit, accessed November 18, 2025, https://www.reddit.com/r/reactnative/comments/1k52ec/what_are_your_thoughts_on_gluestackui/
30. Starting a new project, come from NativeBase and Gluestack benchmarks look bad, any advice? : r/reactnative - Reddit, accessed November 18, 2025, https://www.reddit.com/r/reactnative/comments/16d1wn9/starting_a_new_project_come_from_nativebase_and/
31. Optimal steps to switch from nativebase to gluestack v2 : r/reactnative - Reddit, accessed November 18, 2025, https://www.reddit.com/r/reactnative/comments/1hvogf2/optimal_steps_to_switch_from_nativebase_to/
32. Customization - Mitosis - Builder.io, accessed November 18, 2025, https://mitosis.builder.io/docs/customizability/
33. Getting Started with Mitosis: Creating a Cross-Framework Design System - Builder.io, accessed November 18, 2025, https://www.builder.io/blog/mitosis-get-started
34. Create a reusable component library for Angular, React and Vue using Mitosis and Builder.io | by Abhishek Jha | Medium, accessed November 18, 2025, https://medium.com/@abhishekjha1993/create-a-reusable-component-library-for-angular-react-and-vue-using-mitosis-and-builder-io-d9f5858 0cb56
35. Turborepo & React Native Starter - Vercel, accessed November 18, 2025, https://vercel.com/templates/next.js/turborepo-react-native
36. Start with an example - Turborepo, accessed November 18, 2025, https://turborepo.com/docs/getting-started/examples
37. How to use shared react-native code in turborepo - Stack Overflow, accessed November 18, 2025, https://stackoverflow.com/questions/78442517/how-to-use-shared-react-native-code-in-turborepo
38. Building and Testing React Apps in Nx, accessed November 18, 2025, https://nx.dev/docs/getting-started/tutorials/react-monorepo-tutorial
39. Step by Step Guide on Creating a Monorepo for React Native Apps using Nx | Nx Blog, accessed November 18, 2025, https://nx.dev/blog/step-by-step-guide-on-creating-a-monorepo-for-react-native-apps-using-nx
40. Starter Project - Solito, accessed November 18, 2025, https://solito.dev/starter
41. Build a React Native app with Solito - LogRocket Blog, accessed November 18, 2025, https://blog.logrocket.com/build-react-native-app-solito/
42. Installation - Solito, accessed November 18, 2025, https://solito.dev/install
43. Reanimated 4 \= CSS for React Native? - YouTube, accessed November 18, 2025, https://www.youtube.com/watch?v=eshIq0EBLkI
44. The Web Animation Performance Tier List - Motion Blog, accessed November 18, 2025, https://motion.dev/blog/web-animation-performance-tier-list
45. Animations - React Native, accessed November 18, 2025, https://reactnative.dev/docs/animations
46. tamagui/starter-free: Tamagui React Native Starter Repo - Next, Expo, Solito - GitHub, accessed November 18, 2025, https://github.com/tamagui/starter-free