React Component Library Audit & Enhancement

It looks like you've put together a solid foundation for a production-ready React component library. Managing **TypeScript**, **Storybook**, and a **Testing Suite** (likely Vitest or Jest) all at once is no small feat—it's essentially the "gold standard" for professional frontend infrastructure.

Since you've shared the archive, I can help you audit, expand, or refine this library. Here is a quick breakdown of how we can maximize the value of what you've built:

🛠️ Infrastructure Overview

Based on your description, your library likely follows this structural pattern:

- **TypeScript:** Provides the type safety necessary for consumers to use your components without guessing the props.
- **Storybook:** Serves as your "living documentation," allowing developers to interact with components in isolation.
- **Testing:** Ensures that logic-heavy components (like data tables or modals) don't break during future refactors.

🚀 How I Can Assist You Right Now

I can help you with the "last mile" tasks that usually take the most time:

1. Code Review & Optimization

I can analyze specific component files to:

- Identify unnecessary re-renders using memo or useMemo.
- Ensure **WAI-ARIA** compliance for accessibility (keyboard navigation, screen reader roles).
- Refine TypeScript interfaces for better IDE autocomplete.

2. Automated Documentation

If your Storybook stories are set up, I can help write **Markdown (MDX)** documentation or create complex "Knobs/Controls" so designers can test different UI states (Loading, Error, Disabled) without touching code.

3. CI/CD & Publishing

Are you planning to host this on **NPM** or a private registry? I can help you configure:

- **Rollup or TSDX:** To bundle your code into ESM and CJS formats.
- **Semantic Release:** To automate versioning (v1.0.1, v1.1.0) based on your commit messages.
- **GitHub Actions:** To run your test suite automatically on every Pull Request.

💡 Quick Tip: The "Barrel" Export

To make your library easy to use, ensure you have an index.ts at the root of your src folder that exports everything. This allows users to do this:

TypeScript

```
import { Button, Card, Modal } from '@your-library/core';
```

Would you like me to analyze a specific component from your ZIP file for performance and accessibility, or should we look at setting up a publishing pipeline first?

🖼️ Image-Related Component Enhancement

If your library includes any components that handle images—such as `Avatar`, `Thumbnail`, `Gallery`, or a simple `Image` component—we should focus on the following enhancements:

1. Performance Optimization

Strategy	Description	Key Benefit
Lazy Loading	Use the `loading="lazy"` attribute or an `IntersectionObserver` to defer loading images until they are near the viewport.	Faster initial page load time (First Contentful Paint).
Responsive Images	Utilize the `<picture>` element	Reduces data transfer and

Strategy	Description	Key Benefit
	and `srcset`/`sizes` attributes to serve appropriately sized images based on the user's viewport.	improves mobile performance.
Modern Formats	Recommend or enforce the use of modern image formats like **WebP** or **AVIF** when supported by the browser, with a fallback to JPEG/PNG.	Significantly smaller file sizes without noticeable quality loss.

2. Accessibility (A11y)

- **Alt Text Handling:** Ensure the `alt` prop is mandatory. Implement checks to prevent empty `alt=""` for meaningful images. If an image is purely decorative, enforce `role="presentation"` or `alt=""` explicitly.
- **Contextual Descriptions:** For complex images (e.g., charts or graphs), guide the user to provide a more detailed description via an optional `longdesc` attribute or a link to a separate description.

3. Error and State Management

- **Error Fallback:** Implement a state for when an image fails to load (`onError`). This usually involves displaying a default icon or a broken image placeholder instead of a blank space.
- **Loading State:** Show a simple skeleton screen or spinner (`onLoadStart`) while the image is being fetched to improve perceived performance.

Would you like to review an existing `Image` or `Avatar` component for these optimizations, or do you need help designing the component interface for these props?

The document outlines an audit and enhancement plan for a production-ready React component library that already utilizes TypeScript, Storybook, and a Testing Suite (like Vitest or Jest). The assistance offered focuses on three main areas: Code Review & Optimization (e.g., performance fixes, WAI-ARIA compliance, TypeScript refinement); **Automated Documentation** (writing MDX or creating complex Storybook controls); and **CI/CD & Publishing** (configuring Rollup/TSDX, Semantic Release, and GitHub Actions). Additionally, a specific section provides detailed recommendations for enhancing **Image-Related**

Components (Avatar, Gallery) with strategies for performance (Lazy Loading, Responsive Images, WebP/AVIF formats), accessibility (mandatory Alt Text, contextual descriptions), and state management (Error Fallbacks and Loading States).
The three main areas of assistance offered for the React component library are:

1. **Code Review & Optimization**
2. **Automated Documentation**
3. **CI/CD & Publishing**

The three main strategies recommended for Performance Optimization of image-related components, according to the document, are:

1. **Lazy Loading**: Use the `loading="lazy"` attribute or an `IntersectionObserver` to defer loading images until they are near the viewport.
2. **Responsive Images**: Utilize the `<picture>` element and `srcset`/`sizes` attributes to serve appropriately sized images based on the user's viewport.
3. **Modern Formats**: Recommend or enforce the use of modern image formats like **WebP** or **AVIF** when supported by the browser, with a fallback to JPEG/PNG.