ChatGPT

Advanced React Patterns and Anti-Patterns (2025 Edition)

Introduction: This reference is a comprehensive guide to idiomatic React patterns and anti-patterns in modern frontend development. It focuses on functional components and Hooks, emphasizing composition, context, custom Hooks, and best practices. Each pattern is paired with concise examples and commentary, including how to integrate these patterns with Test-Driven Development (TDD) using Jest and React Testing Library. The guide concludes with tips on prompting Large Language Models (LLMs) to generate bug-free React code following these idioms and TDD principles.

Functional Components and Composition

React encourages **functional components** and composition over class-based components or inheritance. Composition means building UIs by combining small, focused components rather than one monolithic component. An application should be a *"composition of tens or hundreds of components… each component has its own reason to exist"* [1]. Relying on just a few giant components to do everything is an anti-pattern – *"I have seen projects with a few large page-wide components… Doing that is to abuse React"* [2]. Always prefer making new components and composing them via props and children.

Example – Composing Components: Suppose we have a `Page` component that needs a header, content, and footer. Instead of one large component handling all, we create distinct components and compose them:

```
function Header() {
  return <header><h1>My Site</h1></header>;
}
function Footer() {
  return <footer>© 2025 My Site</footer>;
}
function Page({ children }) {
  return (
    <div className="page">
      <Header />
      <main>{children}</main>
      <Footer />
    </div>
  );
}

// Usage:
<Page>
```

```
  <p>Welcome to my site!</p>
</Page>
```

Commentary: Here `Page` composes `Header` and `Footer` components and renders any passed `children` inside a main section. This follows React's core design pattern of component composition [3]. Each component has a single responsibility, making the code easier to maintain and test. Avoid designs that violate this principle by cramming too much logic or markup into one component.

Hooks and State Management Patterns

Hooks are central to modern React. Idiomatic usage involves keeping components **pure and declarative**, handling side effects in effects, and sharing reusable logic via custom Hooks. Key patterns include:

Using `useState` for State

Use the `useState` Hook (or `useReducer`) for component state instead of mutating variables. *Never declare stateful values as plain variables inside a component.* Doing so redeclares them on each render and breaks referential stability. For example, using a normal variable causes recomputation and re-renders on every parent update [4]. Instead, use `useState` so React can preserve state across renders [5].

```
function Counter() {
  //   Use useState for state
  const [count, setCount] = React.useState(0);

  //   Avoid: const count = 0; (would reset every render)
  return <button onClick={() => setCount(count + 1)}>Count: {count}</button>;
}
```

Commentary: In the above `Counter`, using `useState` ensures `count` persists and only changes via `setCount`. If we had used a local variable, it would reset to 0 on each render, and React wouldn't know when to re-render. Using Hooks like `useState` is essential for React to manage state lifecycles properly [4].

If you find yourself managing multiple related state variables, consider consolidating them with `useReducer`. The `useReducer` Hook is useful for complex state logic or when several state variables should update together. For example, form field state or any state with multiple sub-values can benefit from a reducer to avoid many independent `useState` calls (improving clarity and reducing bugs) [6].

Side Effects with `useEffect`

`useEffect` **Pattern:** Perform side effects (data fetching, subscriptions, timers, logging, etc.) inside `useEffect` Hooks, not during rendering. React components and Hooks must remain **pure** and free of side effects during render [7] [8]. By using `useEffect`, you instruct React to run the effect after rendering, thus keeping the render phase pure. Always specify a **dependency array** for your effects – listing every state or prop that the effect depends on. This makes the effect run when those dependencies

change and avoids stale data bugs [9]. If an effect doesn't need any dependencies (should run only on mount/unmount), use an empty array `[]` to run it once.

```
function DataFetcher({ url }) {
  const [data, setData] = React.useState(null);

  React.useEffect(() => {
    let isSubscribed = true;
    fetch(url)
      .then(res => res.json())
      .then(json => { if (isSubscribed) setData(json); });
    // Cleanup in case the component unmounts mid-request
    return () => { isSubscribed = false; };
  }, [url]); // effect runs whenever url changes

  if (!data) return <div>Loading...</div>;
  return <div>Loaded {data.items.length} items</div>;
}
```

Commentary: In `DataFetcher`, the `useEffect` handles fetching data whenever the `url` prop changes. The empty return array ensures the effect runs on mount and on `url` updates only [9]. We also demonstrate cleaning up (setting `isSubscribed=false` on unmount) to avoid setting state on an unmounted component – a best practice for avoiding memory leaks. **Anti-patterns to avoid:** calling data fetches (or other side effects) directly inside the component body (render) or forgetting to include `url` in the dependency list. The former breaks purity and can cause infinite loops, and the latter can cause missing updates or stale closures. An ESLint rule (`react-hooks/exhaustive-deps`) usually catches missing dependencies – always heed those warnings and include all dependencies in your effect [9].

Additionally, do not misuse `useEffect` for code that can be executed elsewhere. For example, **avoid using `useEffect` to run one-time initialization of third-party libraries** that don't interact with React state [10]. Such code can often run outside React or in a higher-level initialization. Each effect should ideally be triggered by some reactive value change or mount/unmount of the component.

Performance Optimizations with `useMemo` and `useCallback`

`useMemo` **Pattern:** Use `React.useMemo` to memoize expensive calculations so that they don't re-run on every render unnecessarily. `useMemo(fn, deps)` will recompute the value only when `deps` change, returning a cached result otherwise. This is useful when you have a computationally heavy operation or a derived value that should not thrash on each render. *However, do not abuse* `useMemo`. If the computed value is cheap or the component isn't re-rendered often, adding `useMemo` can complicate the code for little gain. **Never use `useMemo` with an empty dependency array for a value that doesn't need React's re-computation** – if a value truly doesn't depend on props or state, just calculate it outside the component or directly in the render without `useMemo` [11] [12]. Using `useMemo` with `[]` is an anti-pattern in most cases because it indicates a constant that shouldn't be inside the component in the first place [13].

Example – using `useMemo`:

```
function PrimeList({ count }) {
  // Compute first N primes (expensive) – memoize so it runs only when `count`
changes
  const primes = React.useMemo(() => {
    const result = [];
    let num = 2;
    while (result.length < count) {
      if (isPrime(num)) result.push(num);
      num++;
    }
    return result;
  }, [count]);

  return <div>{count} primes: {primes.join(', ')}</div>;
}
```

Here, `useMemo` ensures the prime calculation runs only when `count` changes, not on every parent re-render. If `count` is large, this optimization is valuable. If `count` stays same, the previous result is reused. If we had used `[]` as dependencies (attempting to run it only once), it would be incorrect because it wouldn't recompute when `count` updates – so always include necessary dependencies [9]. If the calculation was trivial or `count` is small, we might omit `useMemo` for simplicity.

`useCallback` **Pattern:** Use `React.useCallback` to memoize callback functions, particularly when passing callbacks to optimized child components (wrapped in `React.memo`) or as dependencies to other Hooks. Like `useMemo`, `useCallback(fn, deps)` returns the same function instance until one of `deps` changes. This can prevent child components from re-rendering due to changed prop references, and avoid triggering effects that depend on a function.

Example – using `useCallback`:

```
function Parent() {
  const [value, setValue] = React.useState(false);
  // Stable callback: changes only when `value` changes
  const handleToggle = React.useCallback(() => {
    setValue(v => !v);
  }, []);  // no dependencies here since we use setState's functional update

  return <Child onToggle={handleToggle} />;
}

const Child = React.memo(function Child({ onToggle }) {
  console.log("Child rendered");
```

```
  return <button onClick={onToggle}>Toggle</button>;
});
```

Commentary: In this example, `Parent` defines a memoized `handleToggle`. Because we passed `[]` (and use the functional form of setState which doesn't depend on external variables), the callback never changes identity. The `Child` component (wrapped in `React.memo`) will not re-render when `Parent` re-renders for other reasons, because `onToggle` prop remains the same reference. Without `useCallback`, the arrow function would be re-created on each render and cause `Child` to re-render each time. **Caveat:** You should **not wrap every single function in** `useCallback` **without reason** [14]. There is a slight cost to using it (React must track dependencies and preserve the function), and if the function doesn't cause expensive re-renders downstream, it might be unnecessary. Use it primarily when passing callbacks to optimized children or to avoid re-running an effect that depends on a stable callback.

Anti-Patterns: Avoid wrapping **external or standalone functions** in `useCallback` for no reason [15]. For instance, if you imported a helper function (which is already a stable reference) and call it inside an event handler, you don't need `useCallback` around the handler just for that – simply call the function. Likewise, do not use `useMemo` to do something that should be done in an effect or to lazily instantiate a value that could be created outside the component entirely [11]. Always ask if a memoization is needed; strive for clean, straightforward code unless performance measurements justify the optimization.

Custom Hooks for Reusable Logic

Pattern: Custom Hooks. When multiple components need the same behavior or a single component becomes too complex, **extract logic into a custom Hook**. A custom Hook is just a function that uses React Hooks (by convention, named `useSomething`) and encapsulates stateful logic, which can then be reused. This promotes the DRY principle and separates concerns, making components more focused on presentation. In fact, *if you aren't writing custom Hooks to reuse complex logic, "you are not writing proper React code"* [16]. Custom Hooks also improve testability and maintainability – logic in a Hook can be unit-tested in isolation, and the component using it becomes simpler [17] [18].

Example – refactoring logic into a custom Hook:

```
// Before: component with data-fetching logic embedded (harder to test or reuse)
function PostsComponent() {
  const [posts, setPosts] = useState(null);
  const [loading, setLoading] = useState(true);
  const [error, setError] = useState(null);

  useEffect(() => {
    const fetchPosts = async () => {
      try {
        const res = await fetch('/api/posts');
        const data = await res.json();
        setPosts(data);
      } catch (err) {
        setError(err);
```

```
      } finally {
        setLoading(false);
      }
    };
    fetchPosts();
  }, []);

  // ... UI rendering logic
}
```

We can move the fetching logic into a custom Hook, `usePosts`, to make the component cleaner:

```
// Custom Hook for fetching posts
function usePosts() {
  const [posts, setPosts] = useState(null);
  const [loading, setLoading] = useState(true);
  const [error, setError] = useState(null);
  useEffect(() => {
    let ignore = false;
    (async () => {
      try {
        const res = await fetch('/api/posts');
        const data = await res.json();
        if (!ignore) setPosts(data);
      } catch (err) {
        if (!ignore) setError(err);
      } finally {
        if (!ignore) setLoading(false);
      }
    })();
    return () => { ignore = true; };
  }, []);
  return { posts, loading, error };
}

function PostsComponent() {
  const { posts, loading, error } = usePosts();
  if (loading) return <div>Loading...</div>;
  if (error)   return <div>Error: {error.message}</div>;
  return (
    <ul>{posts.map(p => <li key={p.id}>{p.title}</li>)}</ul>
  );
}
```

Commentary: Now `PostsComponent` delegates data logic to `usePosts`, making it much easier to read and maintain. The hook can be reused for any component that needs to fetch posts. The hook's internals

can also be **unit-tested independently of any UI** (e.g., using a testing utility to call the hook and mocking `fetch`) – *"The hook's logic is easy to test, and the main component is much easier to read."* [18]. This exemplifies separation of concerns: the component focuses on presentation (how to render posts or loading/error states), while the hook manages data fetching (business logic). Custom Hooks often return state and functions, allowing components to use those as needed (e.g. a hook might return `[value, setValue]` just like `useState`, or an object with multiple values and callbacks).

Guidelines for Custom Hooks: Each custom Hook should have a clear purpose (e.g., useAuth, useForm, useDataFetch) and ideally adhere to the Single Responsibility Principle [19]. If a hook grows too complex or takes too many parameters, consider splitting it into smaller hooks or simplifying logic [20]. Also, **never call a custom Hook (or any Hook) inside a loop or conditional** – the same Rules of Hooks apply (only call Hooks at the top level of a React function) [21].

Context and the Provider Pattern

Pattern: Context for global or shared state. React's Context API enables sharing values across the component tree without prop drilling. Common use cases include theming, user authentication info, current language, or any "global" data that many components need. The **Provider pattern** involves wrapping a part of your component tree in a context provider so that any component inside can access the provided value via `useContext` [22]. This avoids having to pass props down through multiple levels. In essence, Context + Provider is React's built-in dependency injection or global state mechanism [23].

Example – using Context:

```
// 1. Create a Context for a theme (light/dark)
const ThemeContext = React.createContext();

// 2. Provider component that uses state and provides it
function ThemeProvider({ children }) {
  const [theme, setTheme] = React.useState('light');
  const toggleTheme = () => setTheme(t => t === 'light' ? 'dark' : 'light');
  return (
    <ThemeContext.Provider value={{ theme, toggleTheme }}>
      {children}
    </ThemeContext.Provider>
  );
}

// 3. Component consuming the context
function Toolbar() {
  const { theme, toggleTheme } = React.useContext(ThemeContext);
  return (
    <div className={`toolbar ${theme}`}>
      Current theme: {theme}
      <button onClick={toggleTheme}>Toggle Theme</button>
    </div>
```

```
  );
}

// Usage: wrap part of app with ThemeProvider
<ThemeProvider>
  <App />   {/* Inside App, any component can use ThemeContext via useContext
*/}
</ThemeProvider>
```

Commentary: The `ThemeProvider` encapsulates theme state and provides it to descendants. `Toolbar` can access `theme` and `toggleTheme` without any explicit prop passing. This is the Provider pattern in action [22], leveraging React Context. **Best practices:** use context at an appropriate granularity. Context updates will re-render all consuming components, so avoid putting extremely frequently changing or large data in a single context if performance is a concern. Sometimes splitting context (e.g., separate context for theme value vs. update function) can minimize re-renders. Also, only use context for truly shared data; not every prop drilling is bad – if only one or two components down need a value, passing as prop might be simpler. Overusing context for every piece of state can lead to complexity.

Anti-Patterns: Avoid updating context value without memoization if the value is derived from props or state – e.g., `<MyContext.Provider value={{ obj }}>` where a new object literal is created each render – that will trigger consumers every time. Instead, memoize or lift out stable references (e.g., use `useMemo` for the context value object or split context). Also, be careful not to abuse context as a substitute for proper component composition; global state is handy but too much can make data flow implicit and hard to trace.

Compound Components Pattern

The **Compound Components** pattern is an advanced composition technique often implemented with context. It allows a parent component to coordinate behavior of multiple related child components without requiring explicit prop ties for each child. A classic example is a `<Select>` component with `<Select.Option>` children, where the parent manages shared state (like which option is selected) and each child option knows whether it's active by accessing context. This pattern provides a **flexible, expressive API** for users of your component, and keeps the parent-child relationship explicit in JSX while hiding the implementation details.

Example – Compound Component (simplified):

```
const SelectContext = React.createContext();

function Select({ children }) {
  const [active, setActive] = React.useState(null);
  // Provide state and updater to children
  const contextValue = React.useMemo(() => ({active, setActive}), [active]);
  return (
    <SelectContext.Provider value={contextValue}>
      <div className="select">{children}</div>
```

```
    </SelectContext.Provider>
  );
}

function Option({ value, children }) {
  const { active, setActive } = React.useContext(SelectContext);
  const isActive = value === active;
  return (
    <div
      className={`option ${isActive ? 'active': ''}`}
      onClick={() => setActive(value)}>
      {children}
    </div>
  );
}

// Usage:
<Select>
  <Option value="apple">Apple</Option>
  <Option value="orange">Orange</Option>
</Select>
```

Here, `<Select>` is the compound parent and `<Option>` is a child that must be used within a `Select`. The context ties them together. When an `Option` is clicked, it calls `setActive(value)` from context, updating the parent's state and thereby all Options see the new `active`. This pattern lets the parent implicitly share state with children without requiring the user to manually wire props through each `Option`. It results in a cleaner API for complex components (similar to how a `<select><option>` works in HTML) [24] [25]. We also guard that the Option is used within a Select by requiring the context (if `useContext` returns undefined, you could throw an error, as seen in some implementations). **Note:** Under the hood, this is just an application of context and composition. As with any context usage, consider performance (many Options re-render when active changes, which is expected in this case).

Other Notable Patterns

- **Controlled vs Uncontrolled Components:** For form inputs, a controlled component means the React state is the single source of truth (you pass `value` and use `onChange` to update state). An uncontrolled component uses the DOM internal state (e.g., using refs or defaultValue). Idiomatic React leans toward controlled components for complex forms because they provide instant sync with state and easier validation, but uncontrolled can be simpler for basic use cases or performance (when you don't need to handle each keystroke).
- **Presentational vs Container Components (Separation of Concerns):** An older pattern (coined by Dan Abramov) is to divide components into "presentational" (UI only, no direct state or logic, just receive props) and "container" (stateful, logic, data fetching, and render presentational ones) [26] [27]. This can enforce a clean separation of concerns. In modern React, you can achieve a similar separation by using custom Hooks for logic and functional components for UI, rather than strictly splitting by file types. The concept is still useful: keep your data fetching/business logic decoupled from layout as much as possible.

- **Render Props and Higher-Order Components (HOCs):** These are **legacy patterns** for logic reuse. A render prop is a technique where a component takes a function as a prop to determine what to render (allowing injection of render logic), and an HOC is a function that takes a component and returns a new component with added props or behavior. For example, an HOC can inject a context or add state. Nowadays, custom Hooks can replace many use cases for HOCs/render props by providing reusable logic without creating additional component layers. They are still worth knowing (many libraries and older code use them), but prefer Hooks for new code due to their simplicity and type-friendliness. If you do use HOCs, **never call them inside render** (define them outside components), and be mindful of props collision and ref forwarding.

Common Anti-Patterns in React (and Better Alternatives)

This section highlights common React anti-patterns – practices that conflict with React's best practices or can lead to bugs – and how to avoid them. Each anti-pattern is described with what to do instead.

- **Mutating State or Props:** State and props in React are **immutable snapshots** in time [28]. Modifying them directly (e.g., `this.state.x = 5` or mutating an object in state) will not trigger re-renders and breaks purity. **Always use state setters** (`setState`, `useState` updater, or dispatch actions in `useReducer`) to update state, and treat props as read-only. If you have a complex object in state and need to update a nested field, create a new copy (e.g. use spread operator or immutability helpers) instead of mutating the original. This ensures React knows an update occurred. Likewise, do not reassign props or use them as mutable containers – if a prop needs to change, have the parent pass a new prop value.

- **Deriving State from Props Incorrectly:** If you find yourself copying a prop into state (e.g., in `useEffect` or on render) just to use or modify it, think twice. This can lead to two sources of truth. It's often better to compute values on the fly or use props directly. Derive state *only* if you need to perform an expensive computation once or if the data truly needs to be decoupled and controlled internally. Even then, use caution and sync it properly in `useEffect` if the prop can change. An example anti-pattern is: `if (prop != stateProp) setState(prop)` inside an effect without proper dependency, which can trigger loops or missed updates. Prefer using props as is, or using memoization if computation is heavy.

- **Using Variables for State (instead of Hooks):** Declaring a variable inside a component to "hold" state is an anti-pattern [4]. For example: `let counter = 0;` and then updating it – this doesn't persist between renders. Each render will re-initialize it. The correct approach is to use `useState` (or `useRef` if the value's change shouldn't trigger re-renders). If a value is only needed for a calculation and not for UI, you might not need state at all (just calculate within render). If it is needed between renders but shouldn't cause re-render (like a timer ID or other ephemeral value), use a `useRef` to store it. **In summary:** use State or Ref hooks for any value that needs to last across renders – do not rely on function scope variables for that purpose [4].

- **Declaring Components Inside Other Components:** Defining a new component *within* the render of another component is almost always a mistake [29] [30]. Every time the parent renders, it re-declares a brand new component type, causing React to unmount and remount that child (losing any state in it) and preventing React from optimizing it. It also bloats the parent component and breaks the rule of one Hooks list per component. **Alternative:** declare the component outside (or use React's

children/render props patterns if you need to pass in dynamic content). If you need to conditionally include a piece of UI, you can conditionally render an existing component, but do not define a new component in place. *(Exception: in rare cases, for truly static helpers or list item markup, some define sub-components inside if they don't use state – but even then, it's better style to move them out.)*

- **Conditional Hooks (Violating Rules of Hooks): Never call Hooks inside loops, conditions, or after early returns.** The Rules of Hooks mandate Hooks run in the same order on every render [21]. An anti-pattern example:

```
if (someCondition) {
  const [x, setX] = useState(0); //   DON'T do this
}
```

or calling a Hook inside a for-loop. This will break the synchronization of Hook calls and lead to bugs (or runtime errors in strict mode). The correct pattern is to call all Hooks unconditionally at the top level of your component. If a Hook needs to be conditionally used, incorporate the condition *inside* the Hook logic or use multiple Hooks and choose which data to use based on condition. Similarly, **don't call Hooks after a `return` statement** (which is just a special case of the rule) [31] [32]. For instance:

```
if (!props.value) {
  return null;
}
const [val, setVal] = useState(props.value); // This hook is now
conditional
```

In this example, `useState` is not called when `!props.value` is true, so the order of Hook calls breaks. The fix is to rearrange logic so Hooks are called first (e.g., call `useState` before the `if` and handle the case after). In general, **always call Hooks at the top of your component, before any early returns** [32].

- **Ignoring `useEffect` Dependency Warnings:** A very common pitfall is intentionally omitting dependencies in a `useEffect` dependency array to "hack" some behavior (e.g., leaving out a prop to run effect only on mount). This leads to stale variables or missed updates. **Anti-pattern:**

```
// Want to run on mount only, but uses prop -> WRONG
useEffect(() => { console.log(props.value); }, []); // props.value not
listed
```

This will log the initial `props.value` forever, never again when `props.value` changes. The correct approach is either include all dependencies and handle the effect every time, or restructure your code (maybe the value should be in state or the effect logic should be elsewhere). If you truly need an effect only on mount, it must not depend on updated props or state. React's ESLint plugin will typically flag missing dependencies – heed those warnings and include them [9]. If including a

dependency causes unwanted re-execution, consider if you can move that value's calculation inside the effect or use a ref to hold stable value. **Bottom line:** never suppress the ESLint exhaustive-deps rule without very strong justification.

- **Excessive Prop Drilling:** Passing down props through many layers when intermediate levels don't need them is known as prop drilling. It's not a bug per se, but if you find props being threaded through multiple components only to reach one deep child, consider using Context or lifting that component higher. Deep drilling makes components less reusable and harder to manage. An anti-pattern is having a deeply nested tree where each level has to pass `{...rest}` or specific props down multiple levels. Instead, introduce a Context provider at an appropriate level or restructure your component hierarchy so the data is closer to where it's needed. This improves clarity and can reduce needless re-renders up the chain.

- **Overusing Context for Everything:** The flip side of prop drilling is using context for things that don't truly need global sharing. Overusing context can make it unclear where certain values come from and can introduce performance issues if not careful. A component receiving props explicitly is easier to test and understand. So use context where it provides clear benefit (e.g., theme, auth user, router, form data that many subcomponents need) but not as a blanket replacement for passing props. If only two levels deep, passing a prop is perfectly fine.

- **Not Cleaning up Side Effects:** When using `useEffect`, always clean up side effects when needed. Anti-pattern examples include setting up an event listener on mount and not removing it on unmount, or starting a timer/interval and not clearing it. This can cause memory leaks or even stray behaviors when components unmount. Always return a cleanup function from `useEffect` if you subscribed to something:

```
useEffect(() => {
  const id = setInterval(doStuff, 1000);
  return () => clearInterval(id);  // cleanup
}, []);
```

Also, if an effect makes asynchronous calls (like fetching data), consider cancelling or ignoring stale responses as shown in previous examples. In React strict mode (which mounts, unmounts, and re-mounts components to help find issues), failing to cleanup can cause effects to run twice or continue running even after unmount, leading to confusing bugs.

- **Using Array Index as Key:** Using array indices as React list keys is a known anti-pattern **when list items can be reordered, inserted, or removed**. It can lead to items **appearing to switch identity** (state or DOM attributes carrying over incorrectly) because the index-based key does not uniquely identify the same item across updates [33]. Always use a stable unique id for keys (like item ID). Index keys are only acceptable for a static list that never changes order or content. Better to avoid it to be safe. Example anti-pattern:

```
{items.map((item, idx) => <Item key={idx} data={item} />)}  //
```

If `items` changes in non-append/prepend ways, React might reuse DOM nodes incorrectly. The solution is:

```
{items.map(item => <Item key={item.id} data={item} />)}  //   use a real id
```

If no stable id is available, consider restructuring data or as a last resort, use the index but be aware of the caveats.

- **Premature Optimization with Hooks:** While Hooks like `useMemo` and `useCallback` are for optimization, using them everywhere without cause is an anti-pattern. Wrapping everything in `useMemo` / `useCallback` can make code harder to read and even **worse for performance** due to the overhead of maintaining memoization [14]. The React core team often suggests: do optimizations last. First make it work in a clear way, then measure and optimize where needed. A common anti-pattern is wrapping every component in `React.memo` and every function in `useCallback` "just in case" – this can backfire if not done carefully. Use these tools when a performance problem is proven (via profiling or obvious heavy work). Keep components simple and follow the **KISS principle** (Keep It Simple, Stupid) whenever possible [34].

- **Not Using React Developer Tools / Warnings:** Failing to run your app in development mode with **Strict Mode** enabled is an anti-pattern from a process perspective. React's Strict Mode (usually `<React.StrictMode>`) will help catch issues like side effects running during render or certain legacy API usage. The official advice: *use Strict Mode and the React ESLint plugin to help follow these rules and catch bugs early* [35]. Warnings in the console (for keys, deprecated methods, etc.) are there to help you – don't ignore them. They often point out exactly the anti-patterns we've listed. A maintainable React codebase embraces the linter and warnings as guidance towards idiomatic code.

TDD with React: Patterns and Testing Practices

Test-Driven Development (TDD) can be seamlessly integrated with React's Hooks and component patterns. In TDD, you write tests for the desired behaviors first, watch them fail (red), implement the minimal code to pass (green), and then refactor while keeping tests green. This process influences how you structure your components and hooks for testability, often for the better (more modular and pure).

Design for Testability: Many of the patterns above inherently improve testability. For example, separating logic into a custom Hook not only promotes reuse but makes it *much easier to test that logic in isolation* [36]. With React Testing Library and Jest, you can test Hooks directly via `@testing-library/react`'s `renderHook` utility or by testing components that use those hooks. Similarly, using context in a controlled way allows you to wrap components in test providers to verify behaviors under different context values.

React Testing Library (RTL) Best Practices: RTL encourages testing the component from the user's perspective – **test behavior and output, not internal implementation details** [37]. This means in tests you typically render your component (or hook) and then simulate user interactions or state changes, then assert on the output DOM or return values. Avoid reaching into component internals or state; instead, write

tests like a user: find elements by text or role, click buttons, enter input, and assert that the expected results appear in the DOM.

- *Example:* If you have a `<Counter />` component that increments a number on button click, a test might render it and then:

```
import { render, screen, fireEvent } from "@testing-library/react";
test("Counter increments value on click", () => {
  render(<Counter initial={0} />);
  const button = screen.getByRole('button', { name: /increment/i });
  fireEvent.click(button);
  expect(screen.getByText(/Count: 1/)).toBeInTheDocument();
});
```

This test doesn't know about the component's state variable or how it increments – it only checks that when the user clicks, the UI shows "Count: 1". This aligns with RTL's guiding principle: assert on the visible behavior (the count text) rather than implementation (like checking a state value). The test is resilient to refactors as long as the outward behavior remains the same.

- **Testing Custom Hooks:** For Hooks, you can use `renderHook` from React Testing Library. *"To test custom hooks in React, we can use the `renderHook()` function... This allows us to render a hook and access its return values."* [38] For example, to test our `usePosts` hook without a component, we could:

```
import { renderHook } from "@testing-library/react";
import { usePosts } from "./usePosts";

test("usePosts fetches and returns data", async () => {
  // Assume we have a mock server or we will simulate fetch
  const { result, waitFor } = renderHook(() => usePosts());
  // Initially, loading should be true
  expect(result.current.loading).toBe(true);
  // Wait for the hook state to update (for the fetch to complete)
  await waitFor(() => !result.current.loading);
  // Now check that posts are loaded
  expect(result.current.posts).toBeDefined();
  expect(result.current.error).toBeNull();
});
```

In this pseudo-test, we render the hook, then wait for `loading` to become false. We then assert that `posts` got set and no error occurred. Under the hood, `renderHook` creates a temporary component to call our hook and gives us the `result.current`. Note that when we call functions that update state inside hooks, we should wrap them in React's `act()` to flush effects. For example, if testing a counter hook's increment function:

```
const { result } = renderHook(() => useCounter(0));
act(() => {
  result.current.increment();
});
expect(result.current.count).toBe(1);
```

Wrapping `increment()` in `act()` ensures state updates are processed before the assertion [39]. RTL usually handles `act()` internally for events, but for direct hook calls you manage it.

- **Testing Components with Context or Providers:** If a component relies on context, in tests you should wrap it with the matching provider. RTL's `render` allows a `wrapper` option for this. For example, to test `Toolbar` from the context example, you might do:

```
render(<Toolbar />, { wrapper: ThemeProvider });
```

This ensures that inside the test, `Toolbar` finds a `ThemeContext`. Alternatively, you can manually wrap:

```
render(<ThemeProvider><Toolbar /></ThemeProvider>);
```

Then proceed to simulate a click on the toggle button and assert the theme text changes. Encapsulating such provider logic in a custom render function for tests can reduce repetition (a common pattern is to create a `renderWithProvider` utility).

- **TDD Workflow Tips:** Start by writing a test for a small piece of functionality (e.g., "should show error message when API call fails"). Run the test, see it fail, then implement just enough in the component/hook to make it pass. This often means writing minimal conditional logic or state, guided by what the test expects. For instance, you might write a test that ensures a loading spinner shows initially. You'd run it, see it fail (because spinner not implemented), then add the `loading` state and conditional render of a spinner, then the test passes. This cycle ensures you only write code that has a purpose defined by a test.

- **Testing Library vs Enzyme/Shallow Rendering:** Modern best practice (and the React team's recommendation) is to use React Testing Library which mounts components fully on a virtual DOM (JSDOM). Shallow rendering (testing components in isolation by not rendering children) is less common now. Embrace integration-style tests where components are tested with their children and context, as it provides more confidence. However, for very specific logic (like a custom hook or pure function), unit tests are still great.

- **Mocks and Stubs:** For TDD, when writing tests, you may need to simulate certain conditions like API failures. Jest allows mocking modules (e.g., stub out `fetch` or a module that returns a promise). RTL's philosophy is to avoid mocking internals of React and rather test the real component behavior, but mocking network or timeouts or other environment aspects is fine. Consider using tools like

MSW (Mock Service Worker) to simulate server responses in tests; this can make your data-fetching component tests more realistic by actually performing a "fake" fetch.

- **Continuous Integration:** Integrate your test suite to run on each commit or push. TDD is most powerful when tests run frequently and automatically, catching regressions. Also, aim for a mix of test types: some very granular (testing a util or hook function) and some higher-level (render a page and simulate a full user flow). This layered approach gives confidence that your React patterns are working as intended.

Prompting LLMs for Idiomatic, Bug-Free React Code

Large Language Models like ChatGPT can assist in generating React code, but the quality depends heavily on how you prompt them. To get **bug-free, idiomatic React code that follows best practices and TDD principles**, keep these tips in mind when prompting:

- **Be explicit about patterns to use:** Clearly state that you want functional components with Hooks and which Hooks or patterns to apply. For example: *"Write a React function component that uses* `useState` *and* `useEffect` *(with proper cleanup) to fetch user data."* If you expect a custom Hook or context, mention it: *"...use a custom Hook for the data logic and context for global state."* This guides the LLM to follow idiomatic structure.

- **Mention anti-patterns to avoid:** You can instruct the LLM to avoid certain pitfalls. E.g., *"Avoid any anti-patterns like mutating state directly or calling Hooks conditionally."* This reminds the model of the constraints. If the task involves list rendering, you might say *"ensure to use stable keys, not array indices, for list items."* If performance is a concern, *"include `useMemo` or `useCallback` where appropriate to optimize re-renders, but do not overuse them."* Being specific helps the AI produce better adherence to best practices.

- **Request code** and **tests (TDD style):** To leverage TDD with an LLM, you can actually ask the model to generate tests first or alongside the implementation. For instance: *"First, write a Jest + React Testing Library test suite for a new component* `<LoginForm>` *that has an email and password field and submit button. Then provide the React component implementation that makes those tests pass. Use TDD best practices."* This way, the model will output tests and code. By reviewing the tests, you can see if the component's intended behavior is fully covered, and you get a sense of the design. This approach mirrors red-green-refactor, and the AI can assist in each step. Always double-check the tests for correctness and completeness – the AI might miss edge cases that you can then prompt it to add.

- **Ask for explanations or reasoning:** If you want to ensure the LLM is "thinking" in terms of best practices, you can prompt it to explain its choices. For example: *"Provide the React component and tests, and include comments explaining how each part follows React best practices and why certain anti-patterns are avoided."* This not only gives you code but also the rationale, reinforcing the idioms.

- **Example Prompt (putting it together):**

```
Prompt: "Create a React 18 function component called `<TodoList>` that displays
a list of todo items and allows adding a new item.
- Use `useState` to manage the list state and a controlled input form to add
items.
- Use `useEffect` to log a message to the console whenever the list changes (but
avoid any other side effects).
- The component should be wrapped in a context provider that supplies a theme
(light/dark) via React Context, and style the component accordingly (just add a
CSS class based on theme).
- Include a code snippet of a Jest/React Testing Library test suite for this
component's functionality (adding items, theme application).
- Ensure the code follows best practices (no state mutation, no conditional
hooks, proper keys for list items, etc.) and explain briefly how these best
practices are implemented."
```

This prompt sets clear requirements (functional component, which Hooks to use, context usage), includes a testing aspect, and explicitly mentions best practices. An LLM responding to this will likely produce a component that adheres to the patterns we've discussed and even generate tests. The explanation request forces it to verify its approach (which can help catch any mistakes in following the idioms).

- **Iterate with the LLM:** If the first output has issues (e.g., it introduced an anti-pattern or the tests are not robust), you can iteratively refine the prompt. Point out the issue: *"The code you provided uses an anti-pattern (it mutates state directly). Please fix that by using the state setter properly."* or *"Add a cleanup function to the* `useEffect` *in the code, as it currently might cause a memory leak."* LLMs respond well to incremental improvements.

- **Leverage LLM for refactoring:** You can also have the LLM refactor existing code to conform to these idioms. For instance: *"Here is some React code [paste code]. Refactor it to use a custom Hook for the data fetching logic and add a test for the hook. Ensure no anti-patterns are present."* This can be a powerful way to improve legacy code.

In summary, **be precise and instructive in your prompts**. State the desired patterns (Hooks, context, etc.), request TDD elements (tests), and mention specific anti-patterns to avoid. The LLM will use this guidance to shape its output. Finally, always review and run the generated code and tests. Even with a good prompt, you are the final QA – run the tests, ensure everything passes, and that the code is clean. By combining a strong knowledge of React best practices (like those in this guide) with clear prompt engineering, you can effectively collaborate with AI to produce high-quality, bug-free React code.

Sources: The patterns and anti-patterns above are drawn from React's official documentation and expert insights [8] [21], as well as community best practices [36] [40]. Following these guidelines will help keep your React codebase **maintainable, testable, and resilient** well into 2025 and beyond. Remember, writing idiomatic React means writing components that are predictable and easy to debug, which is exactly what these patterns promote [41]. Happy coding!

---

1 2 3 16 17 18 19 23 34 36 Persson Dennis - 21 Fantastic React Design Patterns and When to Use Them | Web Development Blog
https://www.perssondennis.com/articles/21-fantastic-react-design-patterns-and-when-to-use-them

4 5 6 9 10 11 12 13 14 15 29 30 31 32 40 Persson Dennis - React Anti-Patterns and Best Practices - Do's and Don'ts | Web Development Blog
https://www.perssondennis.com/articles/react-anti-patterns-and-best-practices-dos-and-donts

7 8 21 28 35 41 Rules of React – React
https://react.dev/reference/rules

20 Popular patterns and anti-patterns with React Hooks - DEV Community
https://dev.to/justboris/popular-patterns-and-anti-patterns-with-react-hooks-4da2

22 24 25 26 27 React Design Patterns: Essential Knowledge for 2024
https://www.bigscal.com/blogs/frontend/react-design-patterns/

33 React: Anti patterns to avoid. - Medium
https://medium.com/@shriharim006/react-anti-patterns-to-void-5dbc930a714a

37 How to Integrate Mocha, Chai, React Testing Library, and TypeScript in Next.js | by Aysha Cham | Medium
https://medium.com/@blacjay33/how-to-integrate-mocha-chai-react-testing-library-and-typescript-in-next-js-3cd62e13ad7c

38 39 How to Test Custom React Hooks with React Testing Library
https://www.builder.io/blog/test-custom-hooks-react-testing-library