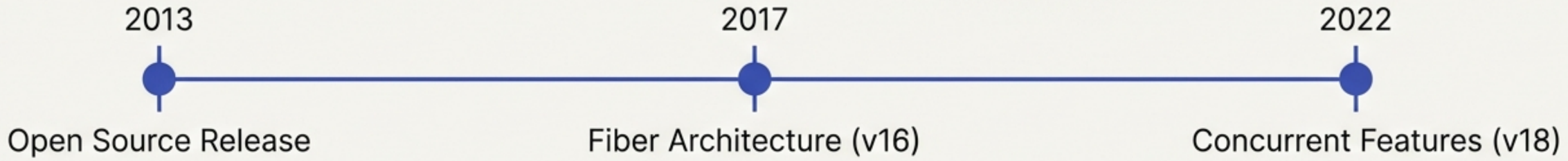
React Core Philosophy & Rendering Model

A Deep Dive for FAANG Interview Preparation & Architectural Mastery

React's philosophy is declarative UI programming where you describe the desired state, and React's rendering model—powered by Virtual DOM diffing and the Fiber reconciler—efficiently computes and applies the minimal DOM mutations needed.



Declarative vs. Imperative

Solving the State Synchronization Crisis

The Problem: Imperative (jQuery)

```
$('#btn').on('click', function() {
  const count = parseInt($('#count').text());
  $('#count').text(count + 1);
  if (count + 1 > 10) {
    $('#warning').show();
  }
});
```


Manual DOM selection. Race conditions. Hard to maintain.

The Solution: Declarative (React)

```
function Counter() {
  const [count, setCount] = useState(0);
  return (
    <>
      <button onClick={() => setCount(c => c + 1)} />
      <span>{count}</span>
      {count > 10 && <Warning />}
    </>
  );
}
```


UI = f(state). Describe the end state, not the steps.

React components don’t ‘update’ the DOM—they describe what the DOM should be. React handles the imperative ‘how’.”

The Mental Model: The Restaurant Kitchen

The Order

`Props & State`

The Customer's Order

Trigger

The Kitchen

`UI = f(state)`

`Component Function`

The Chef creates a new dish description (Virtual DOM).

Render

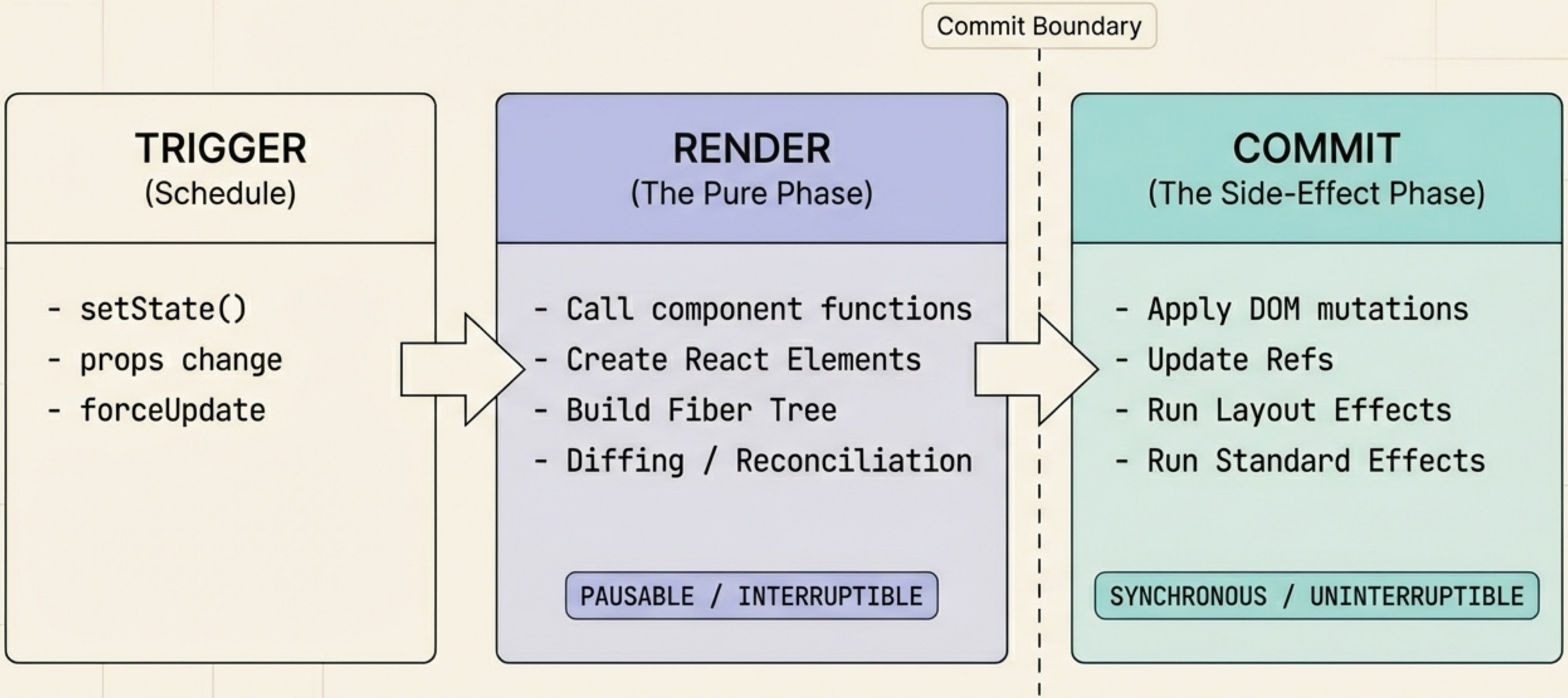
The Service

`Reconciler & Commit`

The Waiter (React) figures out the difference and swaps the plate.

The chef (Component) **doesn't walk to the table to modify the customer's plate directly.** They prepare a completely new dish description, and the waiter (React) figures out the difference to make minimal adjustments.

The Three Phases of the Rendering Cycle



Deconstructing the Virtual DOM

It is not magic. It is a plain JavaScript Object.

1. The Code (JSX)

```
<div className='container'>
  <h1>Hello</h1>
</div>
```

2. The Compilation

```
React.createElement(
  'div',
  { className: 'container' },
  React.createElement('h1', null, 'Hello')
)
```

3. The Virtual DOM (Object)

```
{
  $$typeof: Symbol(react.element),
  type: 'div',
  props: {
    className: 'container',
    children: {
      type: 'h1',
      props: { children: 'Hello' }
    }
  }
}
```

The Virtual DOM allows React to batch updates and abstract away platform specifics (web vs. native).

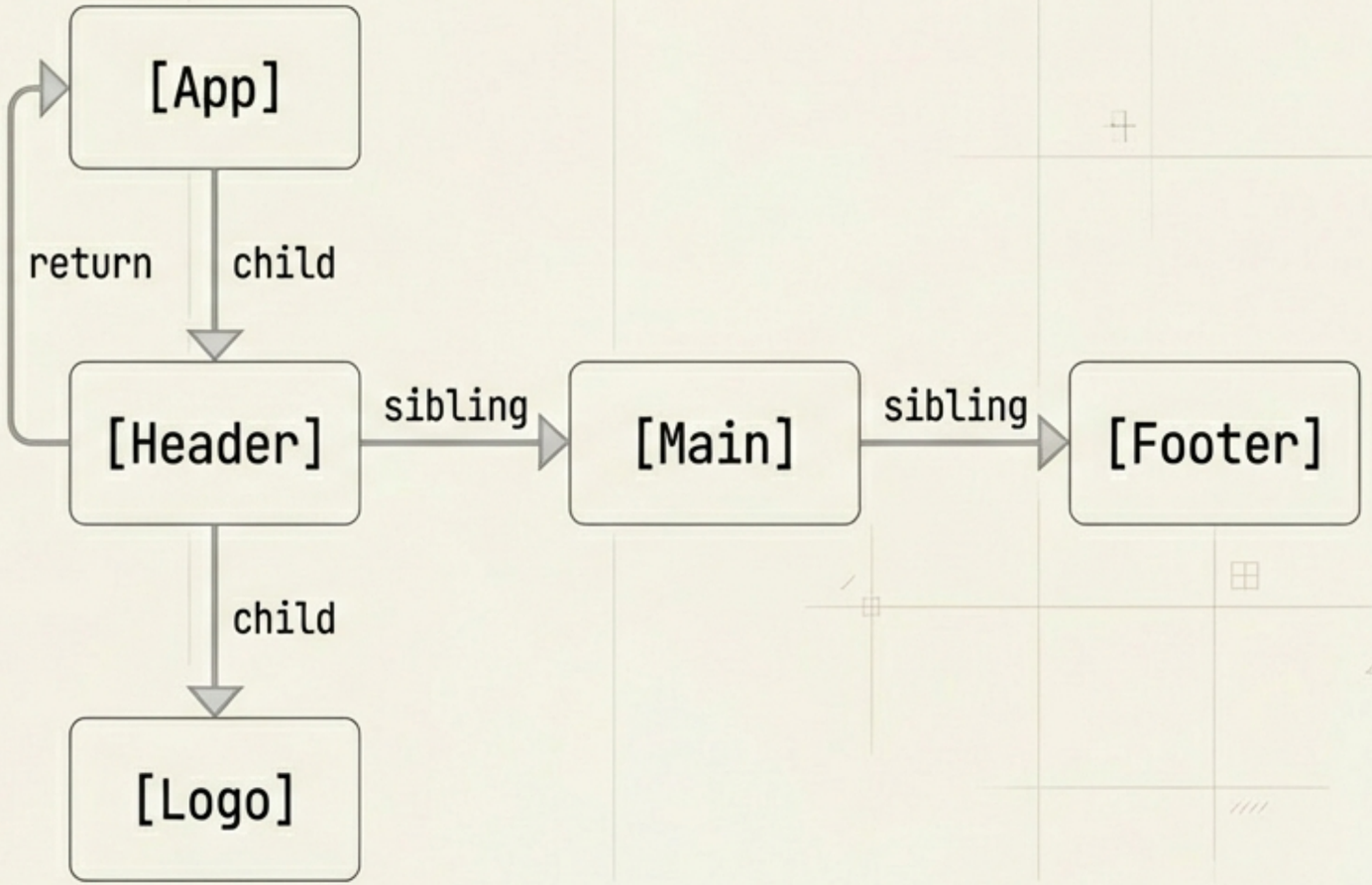
Fiber Architecture: The Engine of Concurrency

Why React moved from Recursion (Stack) to Linked Lists (Fiber).

The “Stack Reconciler” relied on recursion, which blocked the main thread. To enable interruptible rendering, React rewrote the engine to use a Linked List data structure. This allows React to pause work at any node, yield to the browser, and resume later.

Fiber Node Structure

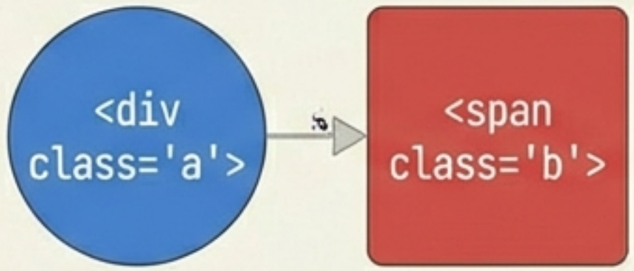
- return (parent)
- child (first child)
- sibling (next sibling)
- memoizedState
- alternate (double buffering)



The Reconciliation Algorithm (Diffing)

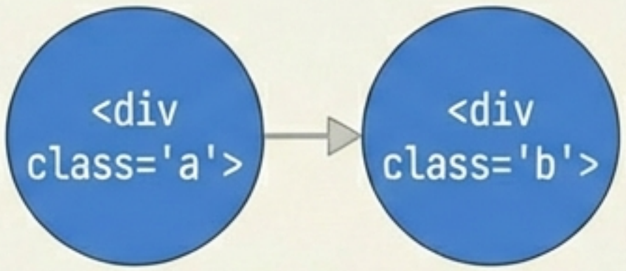
React uses an O(n) heuristic based on three primary rules.

Rule 1: Different Types



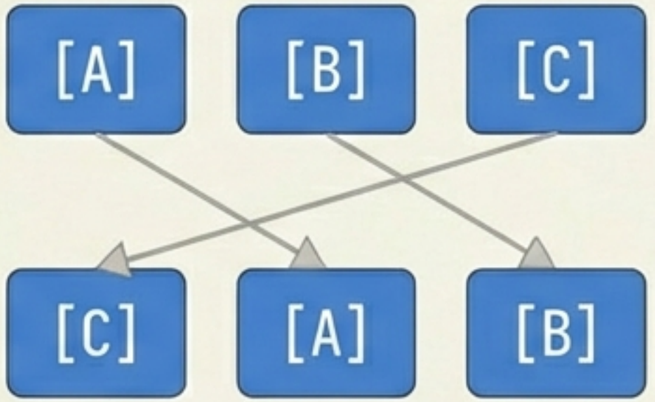
Full Teardown & Remount
(State Lost)

Rule 2: Same Type



Update Attributes Only
(State Preserved)

Rule 3: Keys



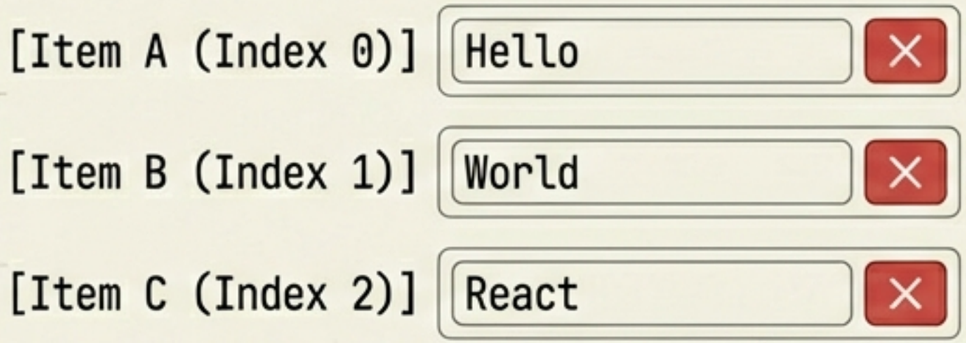
Reorder existing nodes
(No destruction)

The “Key” to Identity

Why using Index as a Key breaks your application.

(The Setup)

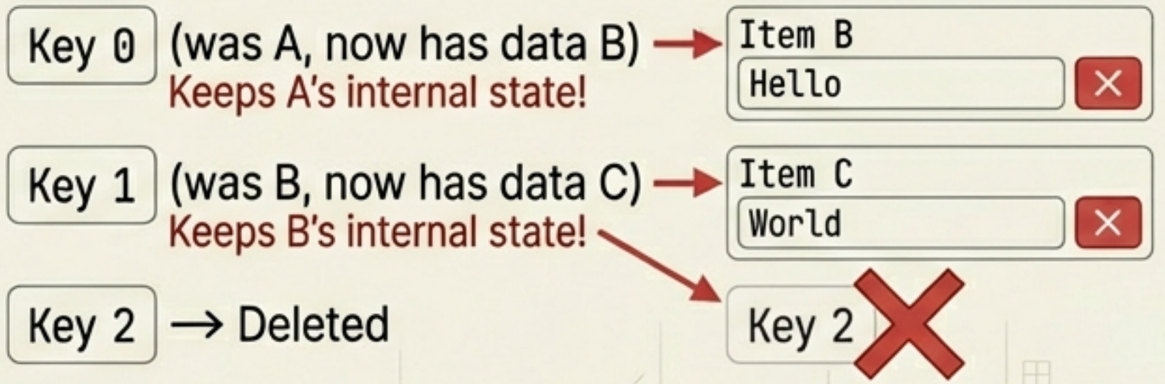
Scenario: A list of inputs with local state (user text).



User deletes Item A

(The Bug)

React sees that Index 0 and Index 1 still exist. It reuses the component instances associated with those keys.



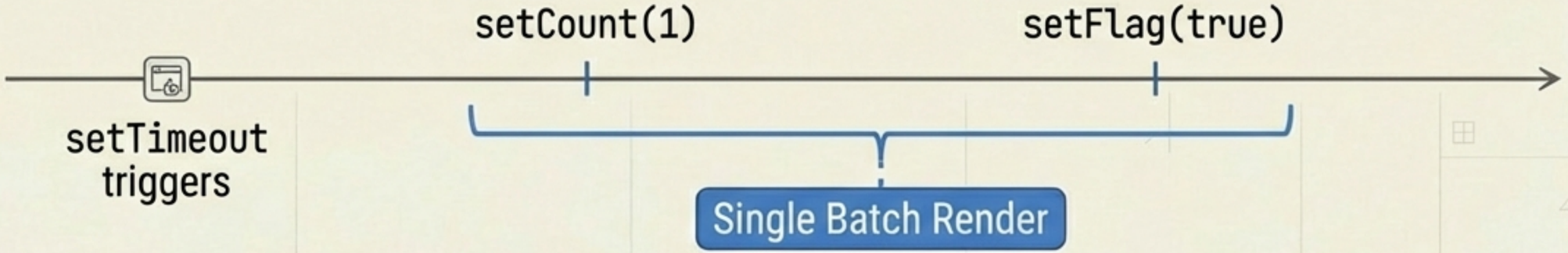
The data shifted, but the local state (text input) stayed in place. User sees wrong data.

Rule: Keys must be unique among siblings and stable across renders. Never use `index` or `Math.random()`.

Modern React: Concurrency & Automatic Batching

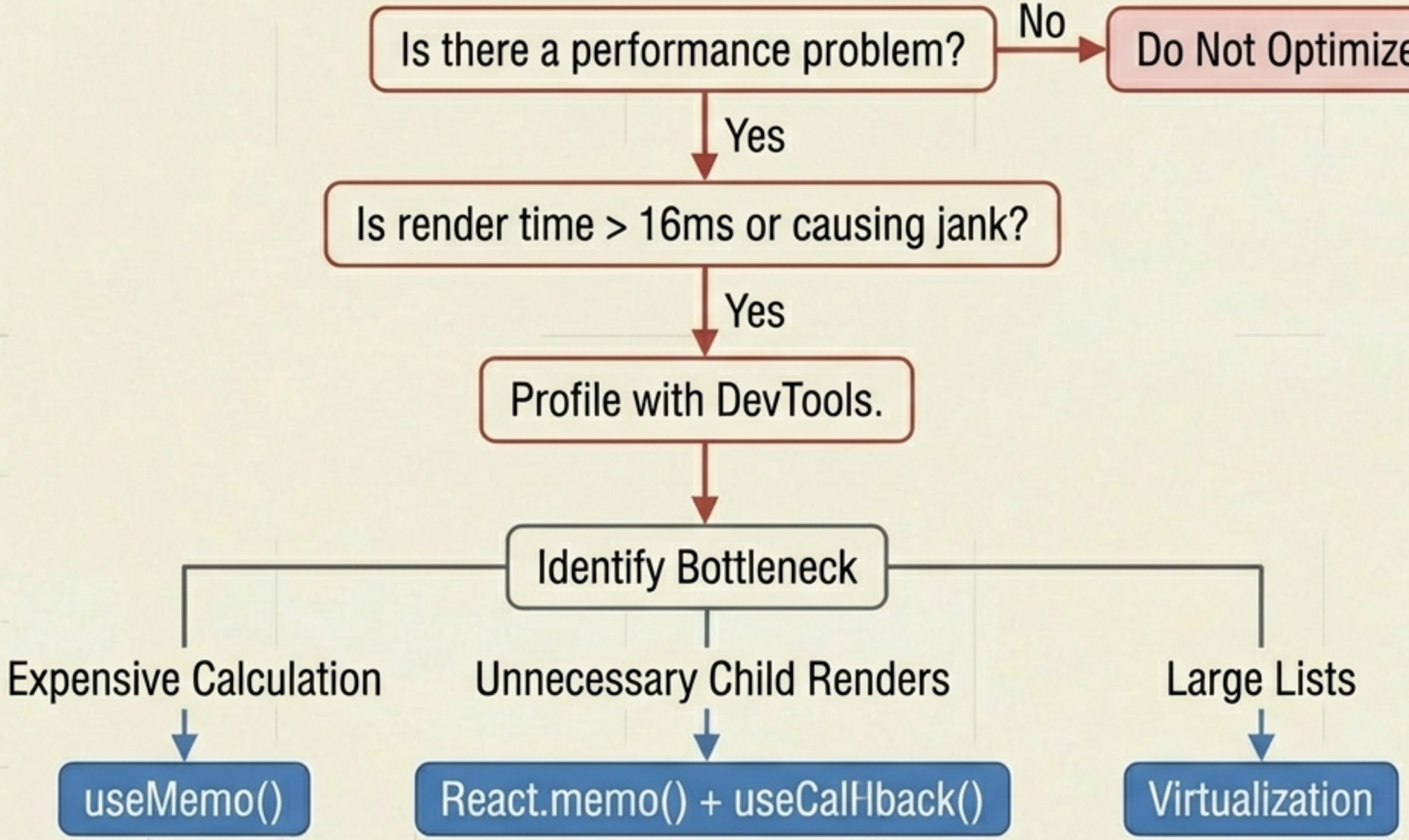
React 17 (Legacy)

React 18 (Automatic Batching)



React 18 batches updates inside promises, timeouts, and event handlers automatically via ‘createRoot’.

Optimization Strategy: When to Memoize



Note:

`React.memo` skips re-render if props are shallow-equal. `useCallback` ensures function references remain stable.

Pattern: Controlled Rendering & Virtualization

```
function VirtualizedList({ items }) {
  const [filter, setFilter] = useState('');
  const deferredFilter = useDeferredValue(filter);

  const visibleItems = useMemo(() => {
    return items
      .filter(i => i.includes(deferredFilter))
      .slice(0, 20); // Only render window
  }, [items, deferredFilter]);

  return (
    <div>
      {visibleItems.map(item => (
        <Row key={item.id} data={item} />
      ))}
    </div>
  );
}
```

Defers expensive filtering to keep input responsive.

Caches the visible slice. Only recalculates when dependencies change.

Virtualization concept: Render only what is visible.

Stable ID maintains identity.

Anti-Patterns: How to Break Reconciliation

❌ The Inline Trap

```
<List
  style={{ margin: 10 }}
  onClick={() => handle(id)}
/>
```

Creating objects or functions inline creates a **NEW** reference every render. This defeats `React.memo` in children.

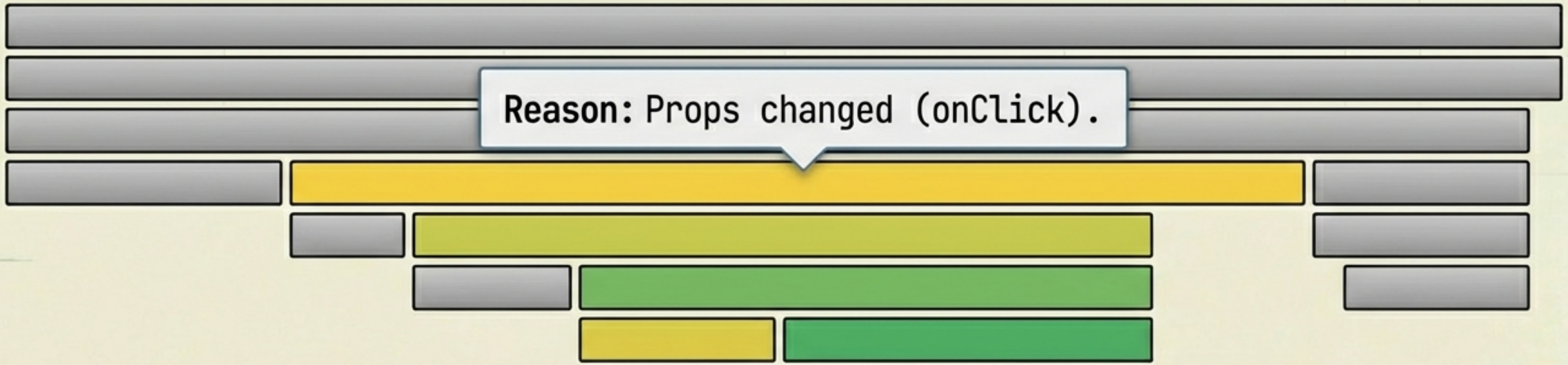
❌ The Mutation Trap

```
const addItem = () => {
  items.push(newId);
  setItems(items);
};
```

Mutating the array directly keeps the same memory reference. React compares (`old === new`), sees true, and **BAILS OUT** of the update.

🔧 **Fix:** Use stable references (`useCallback`/`useMemo`) and immutable state updates (`[...items, newId]`).

Debugging & Performance Profiling



- Width of bar = Render duration.
- Color = Render frequency.
- **Common Error:** “Too many re-renders” (Infinite loop in render phase).
- **Performance Metric:** Component render ~0.1ms vs DOM mutation ~1.0ms.

The Interview Defense: Core Concepts

What is VDOM?

A blueprint. React diffs this object tree to batch updates and avoid slow DOM reads.

Why Fiber?

To make rendering interruptible. It uses linked lists to pause work and prioritize user input.

Render vs Commit

Render is pure calculation (pausable). Commit is DOM application (synchronous/uninterruptible).

Why Keys?

To track identity across reorders. Index keys fail when lists change order or items are deleted.

State Batching

React 18 groups multiple state updates (even in promises) into a single re-render.

The Golden Rule

UI = f(state). Components describe the destination, not the journey.

The Path to Mastery

Self-Assessment

- ☐ Do you understand the 3 phases (`Trigger`, `Render`, `Commit`)?
- ☐ Can you explain why mutation fails (`reference equality`)?
- ☐ Do you know when NOT to use `useEffect` (`derived state`)?
- ☐ Can you implement a `virtualized list`?

“Don't optimize prematurely. Derive state whenever possible. Think in references, not just values.”

Mastering the rendering model turns 'React Magic' into predictable engineering.