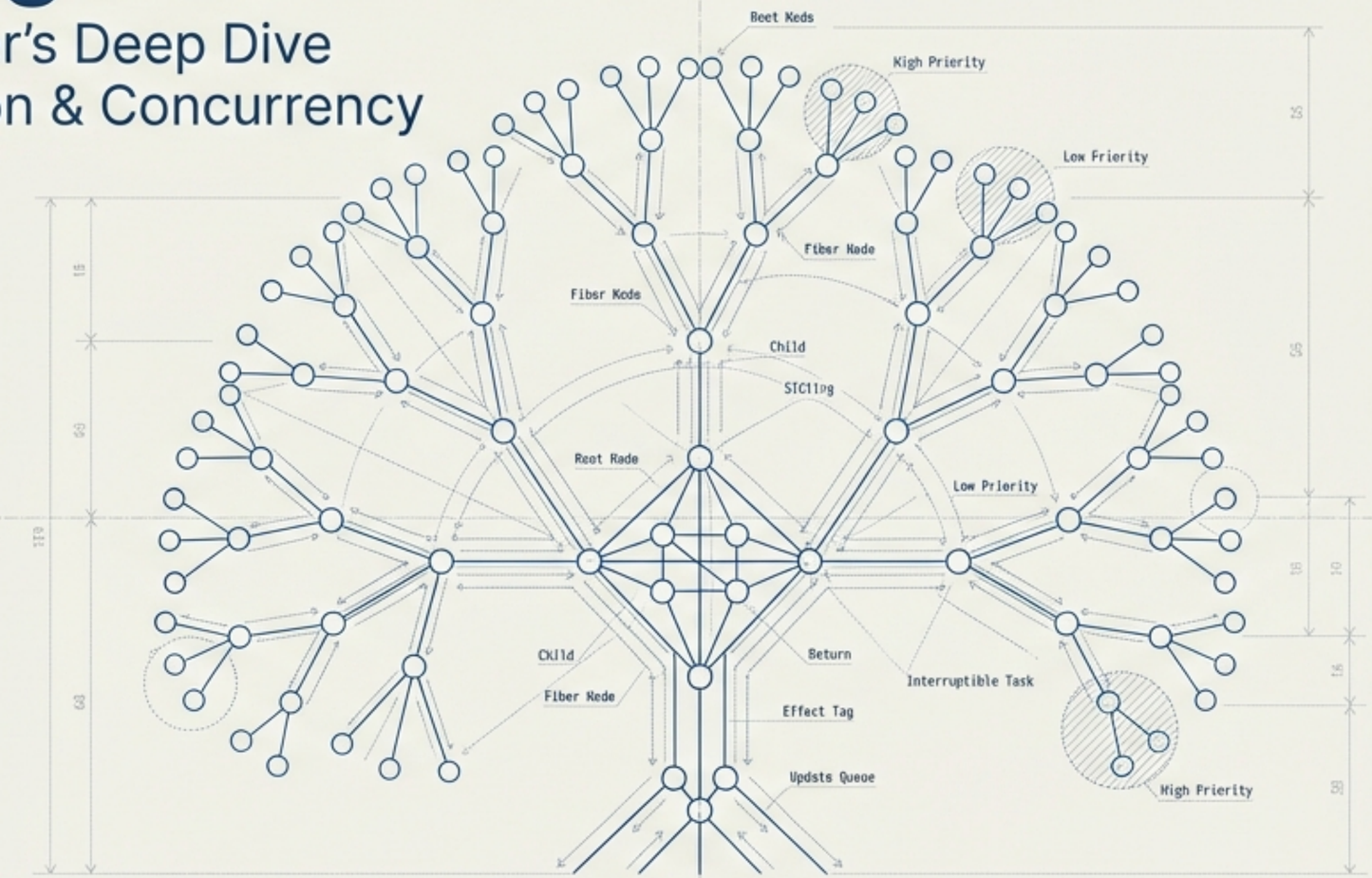
Mastering React Fiber

A Senior Engineer's Deep Dive into Reconciliation & Concurrency



Architecture

Reconciliation Algorithm
Concurrency Model
Data Structures (Fiber Tree)
Scheduler Integration

Internals

Fiber Node Anatomy
Work Loop & Time Slicing
Diffing Strategy
Priority Levels & Expiration

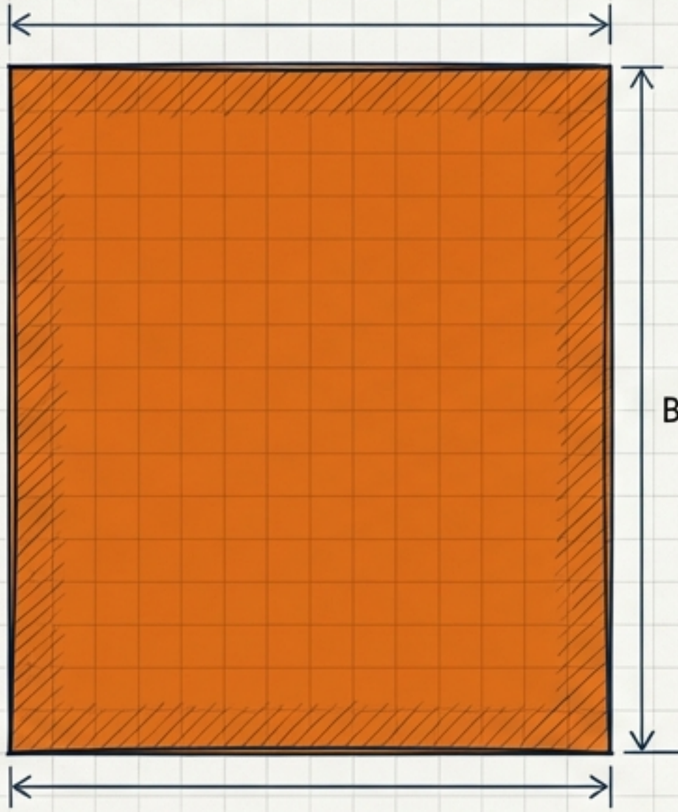
Performance

Optimizing Rendering
Avoiding Blockers
Effective Use of Suspense
Monitoring & Profiling Tools

REACT FIBER
v1.0 - TECHNICAL SPECIFICATION

The Evolution: From Stack to Fiber

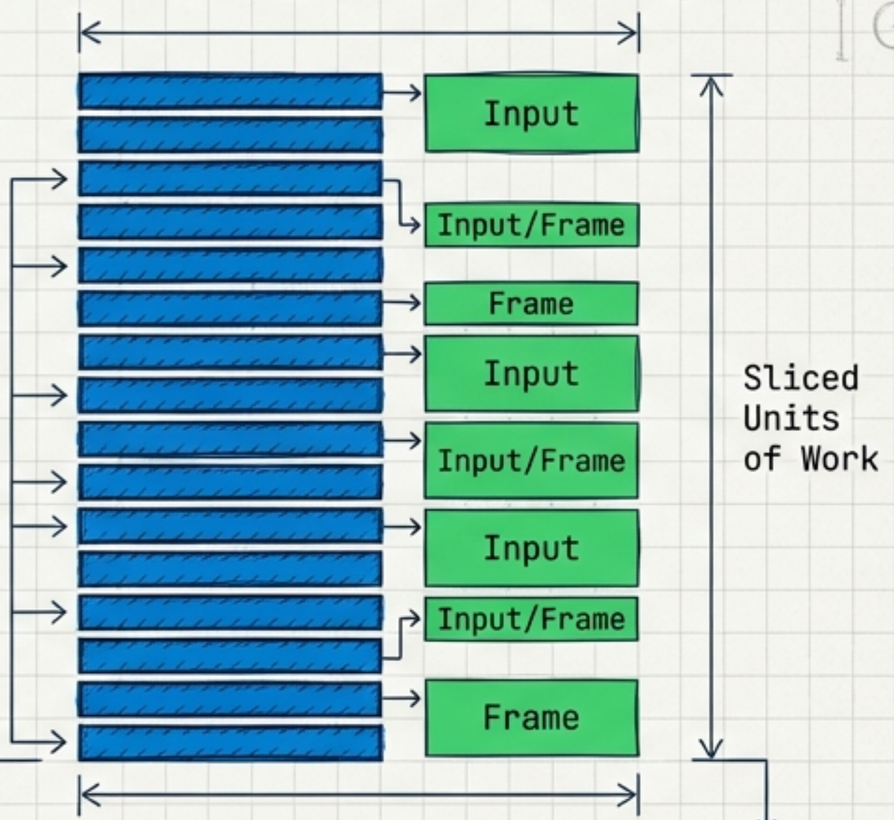
React 15 (Stack Reconciler)



Synchronous & Recursive

Once the process starts, the main thread is locked. User inputs are delayed until completion.

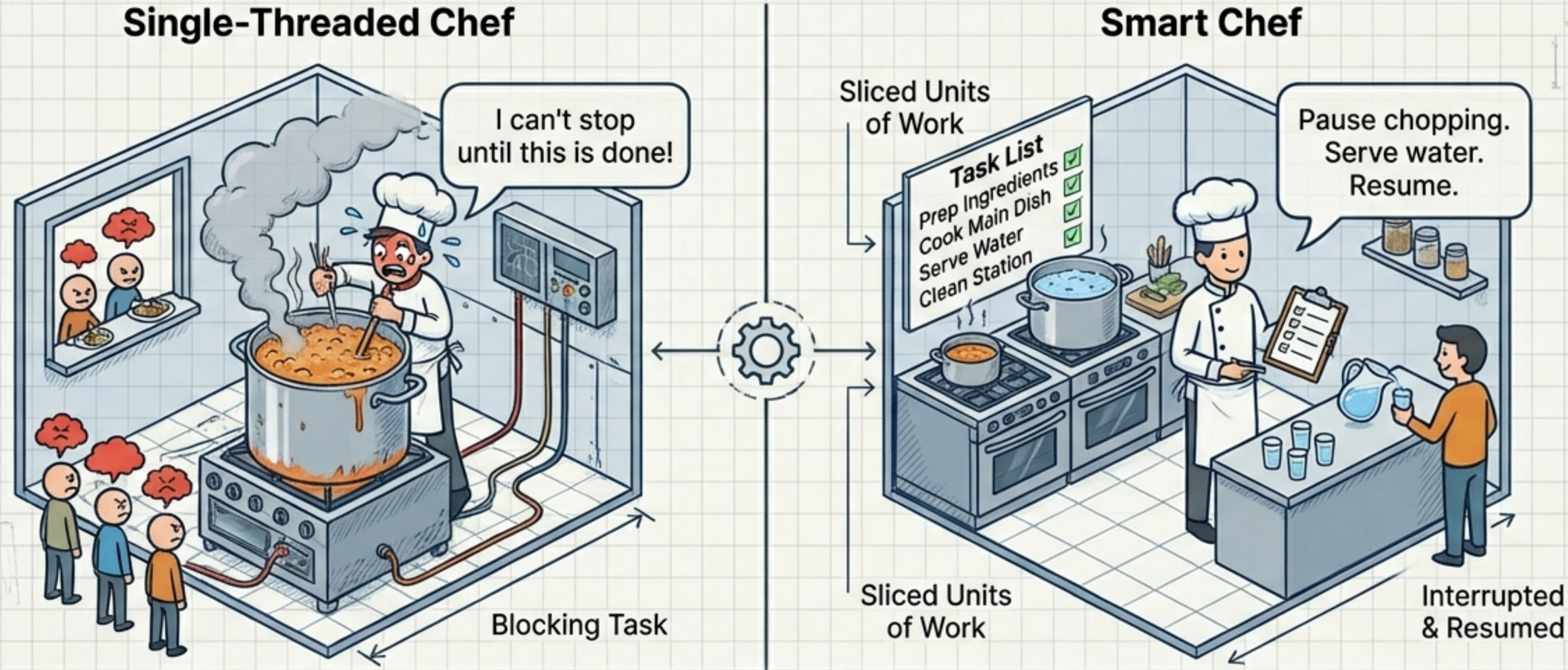
React 16+ (Fiber Reconciler)



Asynchronous & Interruptible

Work is broken into small units. The engine pauses to handle high-priority user inputs, then resumes.

Mental Model: The Intelligent Task Manager



Fiber shifts React from a library that ***runs*** code to an **operating system** that ***schedules*** UI work.

Anatomy of a Fiber Node

Fiber Object

Identity

Structure (The Tree)

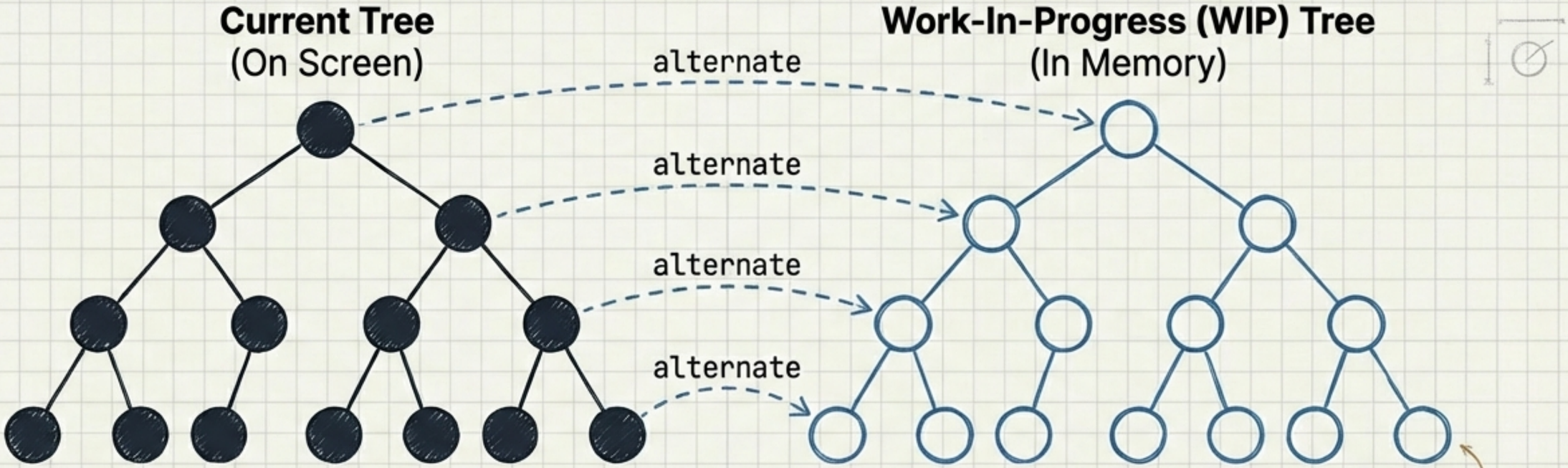
State

Mechanics

```
{
  tag: FunctionComponent | HostComponent,
  key: string,
  type: 'div' | ComponentFn
}
  return: Fiber (Parent) -> ↑,
  child: Fiber (First Child) -> ↓,
  sibling: Fiber (Next Sibling) -> →
}
  memoizedState: Linked List (Hooks),
  pendingProps: Object
{
  alternate: Fiber (Pointer to pair),
  lanes: Bitmask (Priority)
}
```

A **Fiber** is a plain JavaScript object representing a unit of work. It holds the state, props, and structural pointers for a specific component instance.

Architecture: Double Buffering



Step 1
Render happens in WIP

Step 2
Commit Phase

Step 3
Swap Pointers
(WIP becomes Current)

This technique enables smooth transitions. React builds the next state in the background and swaps it in only when complete, preventing inconsistent UIs.

The Engine: The Work Loop

The Old Way (Stack)

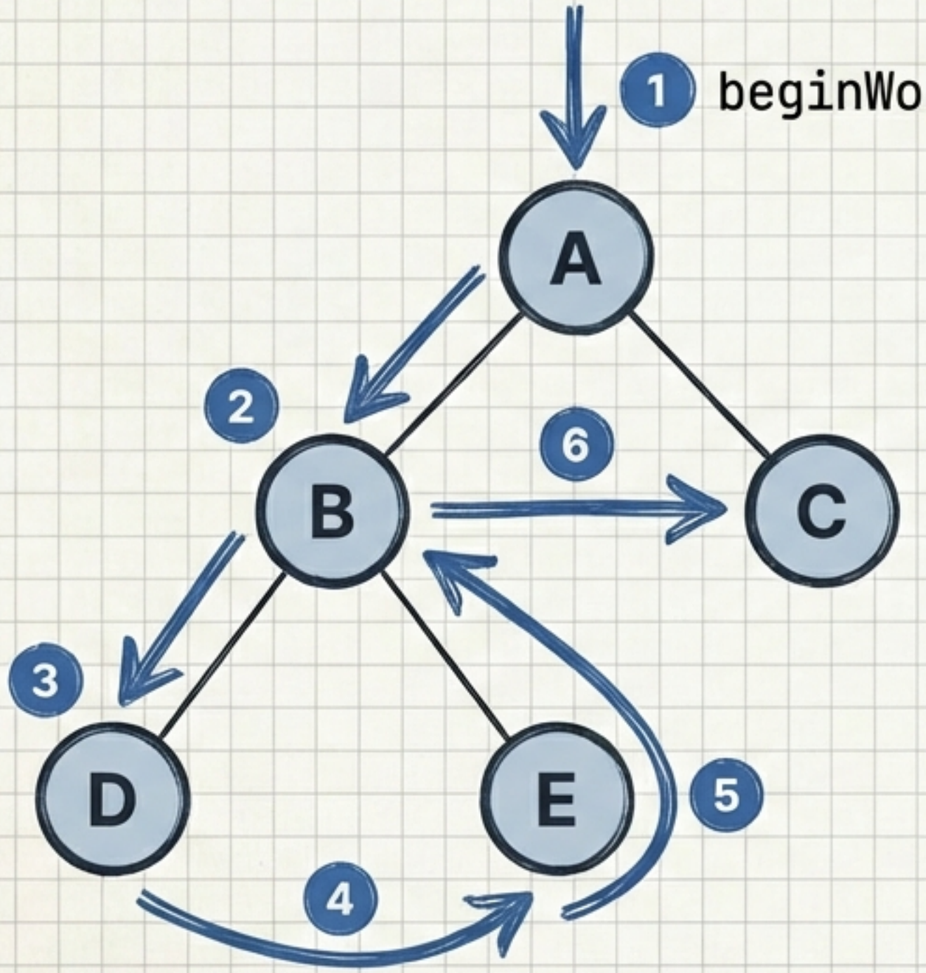
```
reconcile(child); // Recursive Call
// ...
reconcile(grandchild);
// ❌ Cannot stop. Context is in JS Stack.
```

The Fiber Way (Loop)

Checks if the browser needs the main thread (e.g., for a click or animation).

```
// Loop continues only if there is time
while (workInProgress && !shouldYield()) {
  workInProgress = performUnitOfWork(workInProgress);
}
// ✅ Explicit state. Can pause/resume.
```

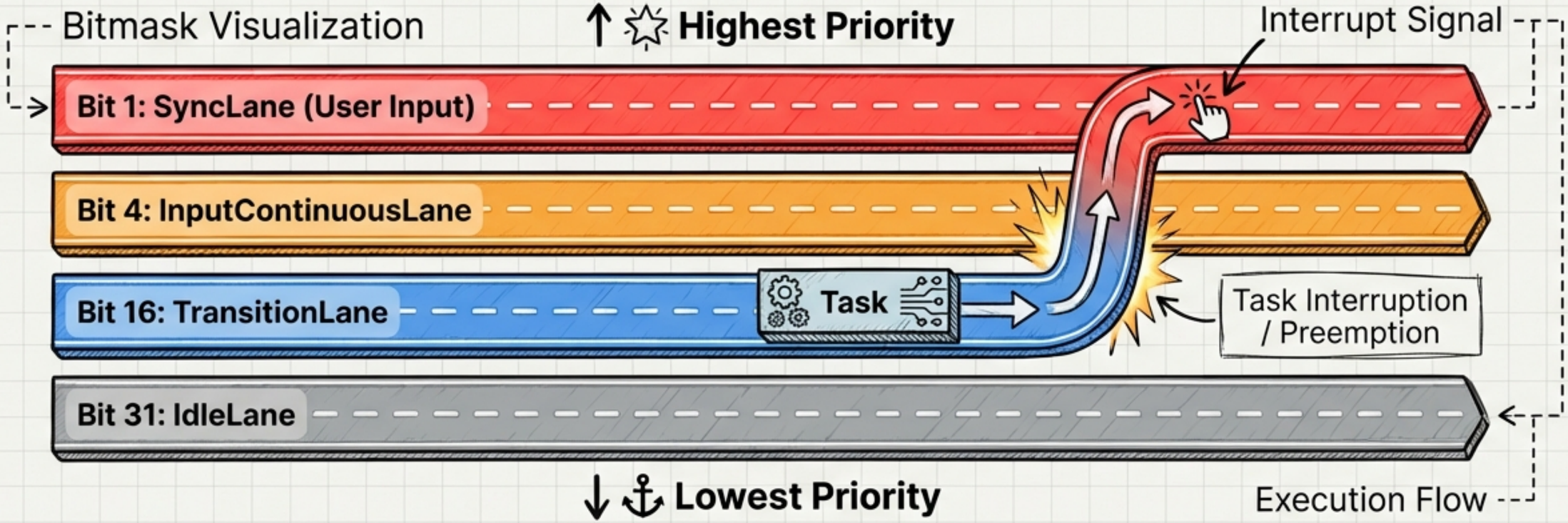
Traversal: The Specific Walk



Algorithm:

1. **Begin (Down):**
   Process Node -> Go to Child
2. **Complete (Up):**
   If no Child -> Complete Work
3. **Sibling (Right):**
   If Complete -> Go to Sibling
4. **Return (Up):**
   If no Sibling -> Return to Parent

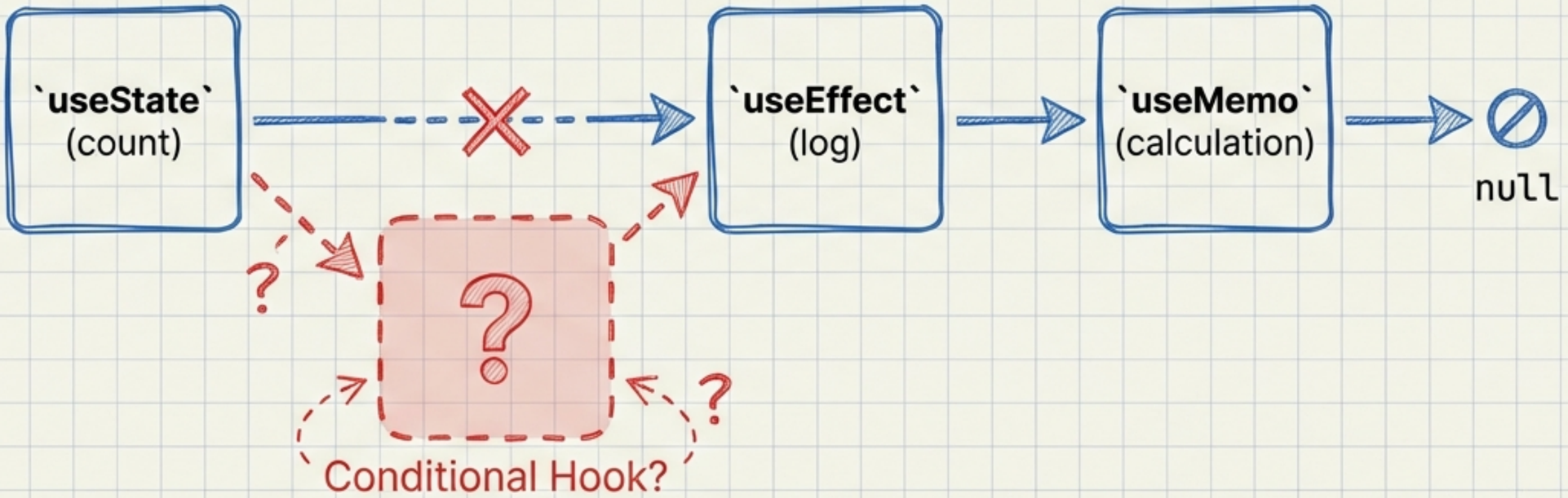
Scheduling & Priority: Lanes



Priority Preemption: If a SyncLane event (click) occurs while rendering a TransitionLane, React abandons the transition work and handles the sync update immediately.

Hooks Under the Hood

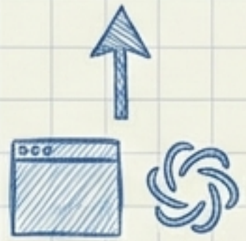
Focus: Fiber Node > **memoizedState**



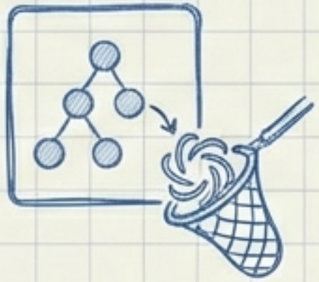
Hooks are stored as a **linked list** within the **`memoizedState`** property. React relies on the **index order** to find the correct state for each hook during re-renders. This is why hooks cannot be placed inside **`if`** statements or **loops**.

Enabling Concurrency: Suspense

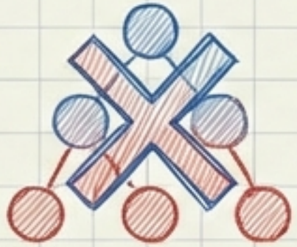
Component Throws Promise



Fiber Catches at Boundary

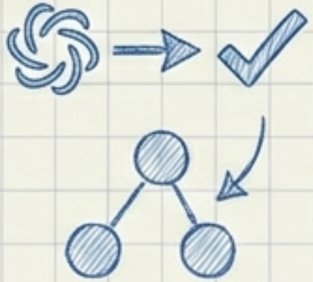


Subtree Marked **"Suspended"**



Render **Fallback UI** (Spinner)

Promise Resolves -> Resume Render



Key Insight: Suspense is only possible because Fiber can save the state of a half-finished tree in memory, show a fallback, and resume later.

Controlling the Scheduler: Transitions

Code Block	UI Effect (Simulation)

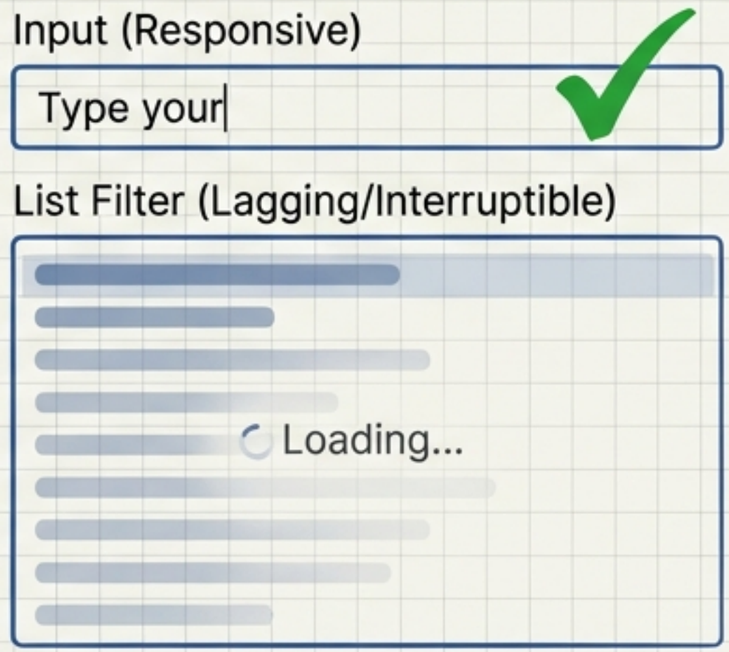
```
const [isPending, startTransition] = useTransition();

// 1. High Priority (SyncLane)
setInputValue(e.target.value);

// 2. Low Priority (TransitionLane)
startTransition(() => {
  setFilter(e.target.value);
});
```

Urgent

Non-Urgent / Interruptible



We explicitly tell Fiber to split the update. The typing remains buttery smooth (Urgent), while the heavy filtering happens in the background (Non-Urgent).

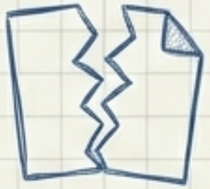
Common Pitfalls & Anti-Patterns

Blocking

Heavy calculation in render body.

```
function Component() {
  const data = calculateHeavyStuff();
  return ...
}
```

Fiber yields *between* components, not *inside* them. Heavy math here still blocks the thread.



Tearing

Reading mutable external state.

```
const value = window.externalValue;
// Mutable!
```

In concurrent rendering, `window.value` might change between the start and end of a render.

✔ **Fix:** Use `useSyncExternalStore`.

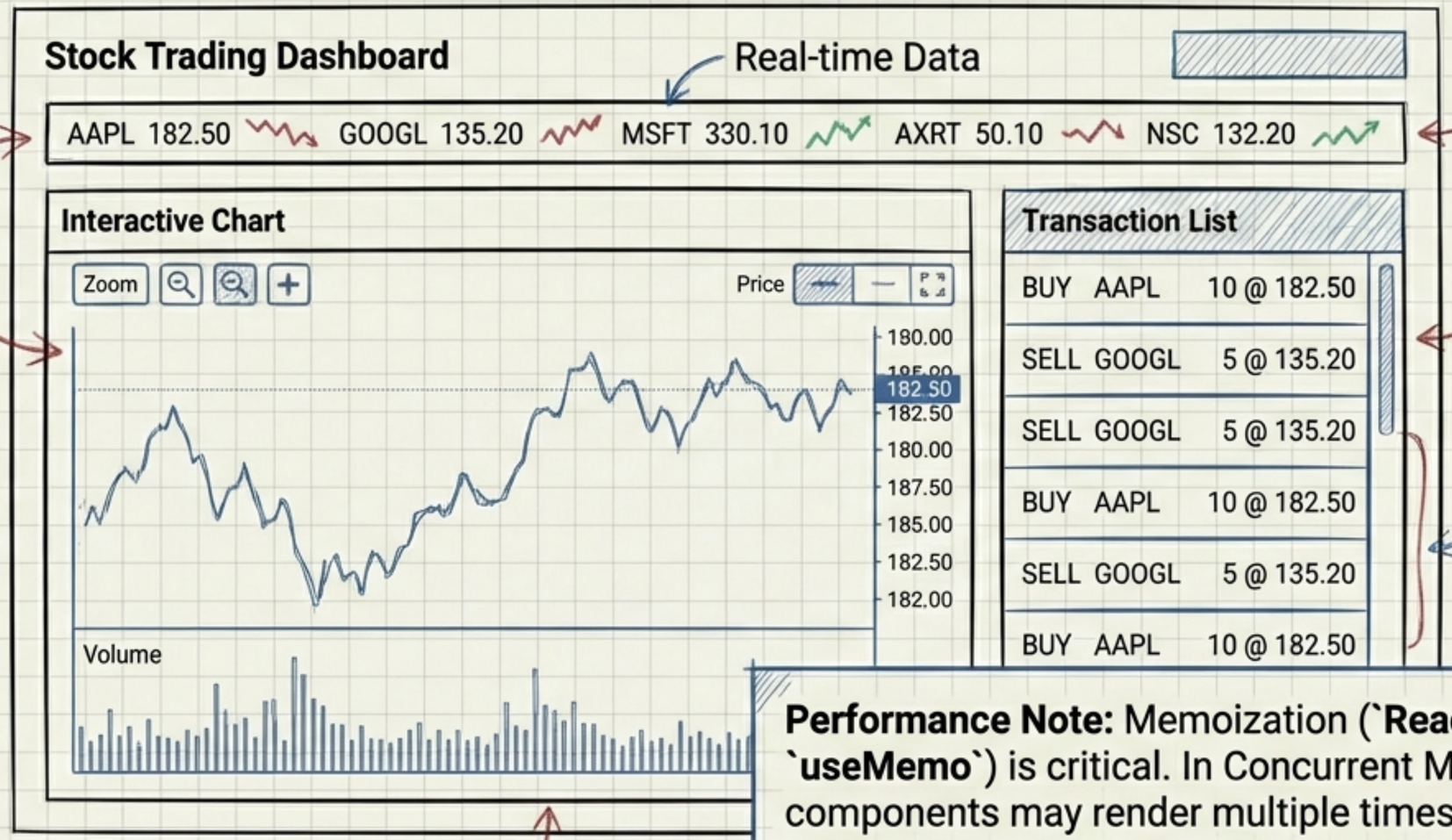
Overuse

Wrapping everything in Transitions.

```
startTransition(() => {
  // Click handler, etc.
});
```

Only use transitions for CPU-heavy, non-urgent updates. Don't make clicks feels sluggish.

System Design & Performance

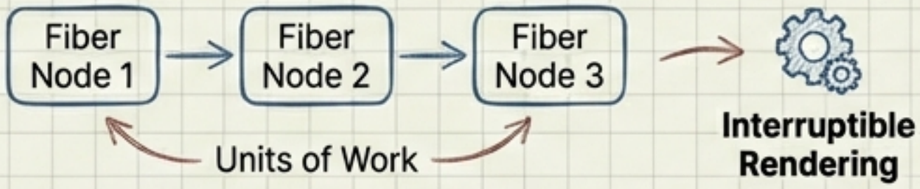


Performance Note: Memoization (`**React.memo**`, `**useMemo**`) is critical. In Concurrent Mode, components may render multiple times before committing. Without memoization, you waste CPU cycles on discarded work.

The Cheat Sheet

Fiber (Definition)

Fiber is a linked-list data structure representing units of work. It enables incremental, interruptible rendering.



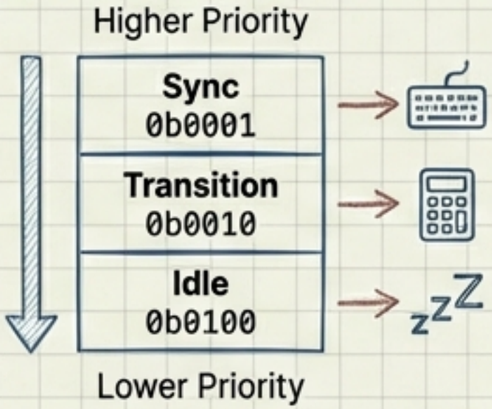
Double Buffering (Structure)

Current Tree (Screen) vs. WIP Tree (Memory). Pointers swap on commit.

Lanes (Scheduling)

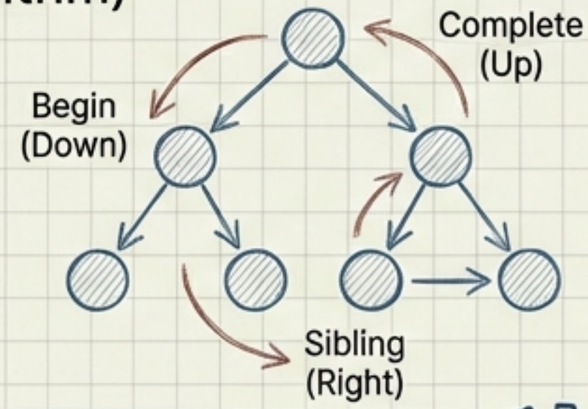
Bitmask priority system.

Sync (User Input) > **Transition** (Calculations) > **Idle**.

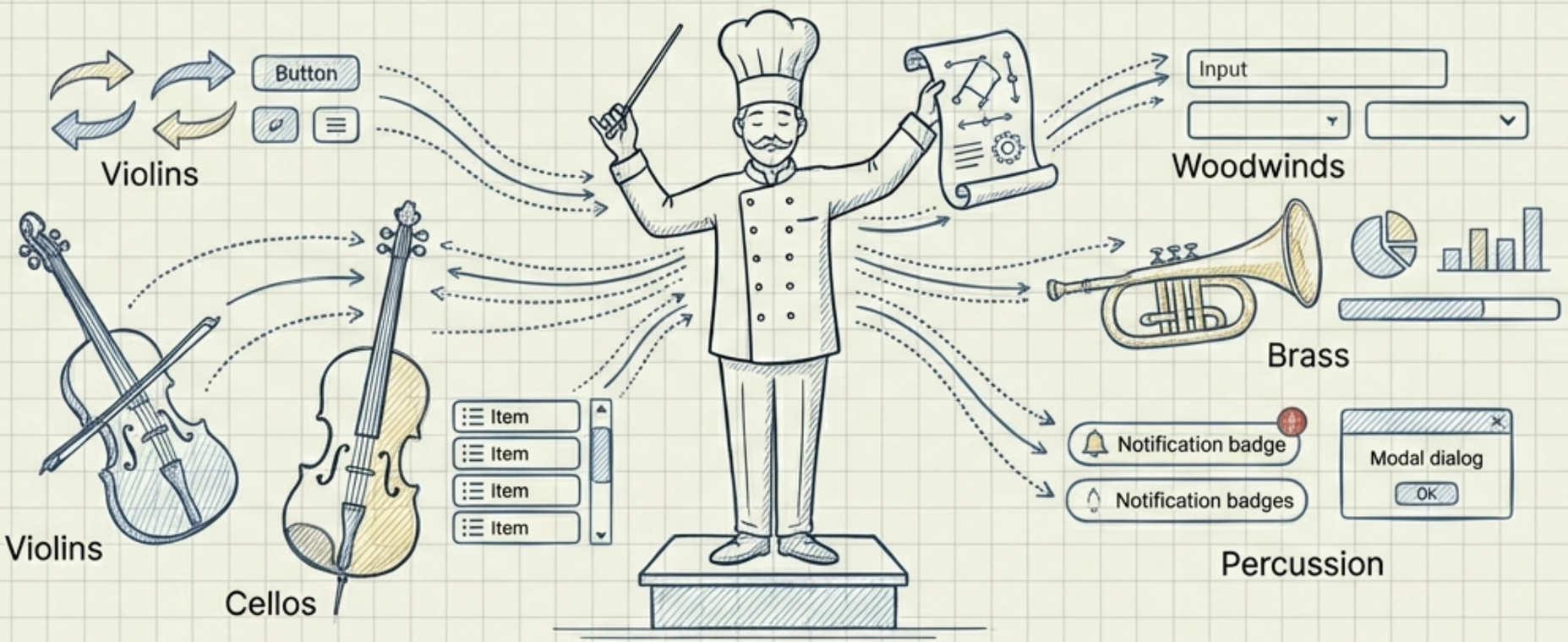


Traversal (Algorithm)

Depth-first walk.
Begin (Down) -> Complete (Up) -> Sibling (Right).



The Paradigm Shift



React has evolved from a library that executes code to an Operating System that schedules UI work.

Mastering the internals makes you a better architect of the surface.