

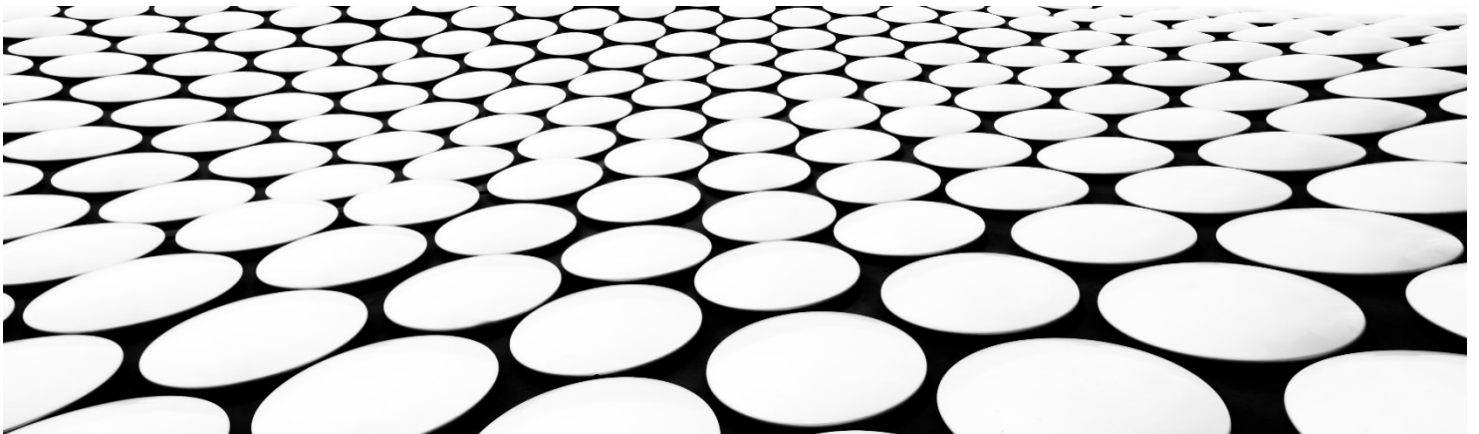
# React Native

# Hookbook

ANDRONOTE STUDIOS

1<sup>ST</sup> EDITION

2026



*If you'd like to support our work, we'd greatly appreciate it if you clicked [here](#) installed Andronote, and considered upgrading to the Pro version*



**2026**

# Copyright

© 2026 Andronote Studios

All rights reserved.

No part of this publication may be reproduced, distributed, or transmitted in any form or by any means, including photocopying, recording, or other electronic or mechanical methods, without prior written permission from the publisher, except in the case of brief quotations embodied in critical reviews or certain other noncommercial uses permitted by copyright law.

React and React Native are trademarks of Meta Platforms, Inc. This book is an independent educational publication and is not affiliated with, endorsed by, or sponsored by Meta Platforms, Inc.

# Disclaimer

This book is intended for educational and informational purposes only.

All code examples, explanations, and diagrams are created to help readers understand React Native concepts, rendering behavior, and hooks. While every effort has been made to ensure the accuracy of the content, Andronote Studio makes no guarantees regarding completeness, reliability, or suitability for any particular purpose.

The author and publisher shall not be held responsible for any errors, omissions, or damages arising from the use of the information contained in this book.

Some diagrams and visual materials in this book were created or refined with AI-assisted tools for educational illustration purposes.

# Preface

React Hooks are powerful, but they are also one of the most misunderstood parts of React and React Native development.

Many developers learn hooks by memorizing syntax without fully understanding how rendering, state, closures, and dependencies actually work. This often leads to bugs, unnecessary re-renders, stale closures, and performance issues.

This book was created to simplify these concepts through clear explanations, practical examples, and visual diagrams.

Rather than focusing only on API usage, this book focuses on mental models — how React thinks, how components re-render, and why hooks behave the way they do.

The goal is simple:

to help you reason about hooks, not just use them.

# Who This Book Is For

This book is designed for:

- Beginner developers learning React Native
- Intermediate developers who want a deeper understanding of hooks
- React developers transitioning into React Native
- Developers struggling with re-renders, dependencies, and hook behavior
- Anyone who wants to build more predictable and efficient mobile applications

Basic JavaScript knowledge and familiarity with React fundamentals will help you get the most out of this book.

# How to Use This Book

This book is structured progressively.

The early chapters build the mental foundation of React rendering, snapshots, identity, and closures. These concepts are essential before understanding hooks deeply.

The middle chapters focus on individual hooks such as:

`useState`

`useEffect`

`useCallback`

`useMemo`

`useRef`

`useLayoutEffect`

`useReducer`

Each chapter includes visual diagrams, conceptual explanations, and practical examples.

The final chapters focus on real-world React Native patterns, debugging strategies, and production-ready mental checklists.

For the best learning experience, it is recommended to read the chapters in order.

# About Andronote Studios

Andronote Studios is an independent software and educational publishing brand focused on mobile application development, software engineering concepts, and practical programming resources.

Its goal is to simplify complex development topics through clear explanations, visual learning materials, and real-world examples.

Andronote Studios is also the creator of Andronote, an offline-first note-taking application built with React Native.

# Contents

<b>Chapter 1</b> .....	13
<b>How React Really Renders</b> .....	13
1.1 Rendering is not updating the screen.....	13
1.2 The three phases of a render cycle .....	13
1.3 A component is re-executed, not resumed .....	14
1.4 Renders are snapshots in time .....	15
1.5 Why this matters for hooks .....	15
<b>Chapter 2</b> .....	17
<b>The Mental Model: Snapshots, Identity, and Closures</b> .....	17
2.1 Snapshots: state is not mutable memory.....	17
2.2 Identity: why equality matters .....	17
2.3 Closures: functions remember their snapshot.....	18
2.4 Stale closures explained correctly .....	18
2.5 Why hooks have dependency arrays.....	19
2.6 The unified mental model .....	19
<b>Chapter 3</b> .....	20
<b>Why Hooks Exist: Design Constraints, Not Convenience</b> .....	20
3.1 The real problems with class components .....	20
3.2 Hooks are constrained by React’s execution model .....	21
3.3 Why hooks are not magic.....	21
3.4 Hooks are designed for concurrency .....	22
3.5 The tradeoff React chose intentionally .....	22
<b>Chapter 4</b> .....	23
<b>useState: State, Scheduling, and Batching</b> .....	23
4.1 What <code>useState</code> really does.....	23
4.2 State updates are scheduled, not applied .....	23
Figure 4.2 <code>useState</code> hook mechanism in React Native.....	24
4.3 Batching: why updates appear “delayed” .....	24
4.4 Functional updates: the correct mental model.....	25
4.5 Objects and arrays: immutability is not optional.....	25

4.6 Why state sometimes “resets” .....	25
4.7 When not to use useState.....	26
4.8 useState in the lifecycle .....	26
<b>Chapter 5.....</b>	<b>28</b>
<b>useEffect: Lifecycle Mapping, Timing, and Cleanup .....</b>	<b>28</b>
5.1 What useEffect is for .....	28
5.2 Why effects run after render .....	29
5.3 Effect timing and dependency arrays.....	29
5.4 Cleanup is part of the lifecycle .....	30
5.5 Effects do not “react” to state — they respond to snapshots .....	30
5.6 Async effects and race conditions .....	30
5.7 Infinite loops and false dependencies.....	31
5.8 Correct mental model for useEffect .....	31
<b>Chapter 6.....</b>	<b>32</b>
<b>useCallback: Identity, Closures, and Stability .....</b>	<b>32</b>
6.1 Why function identity matters.....	32
6.2 What useCallback actually does.....	33
6.3 Closures and correctness .....	33
6.4 useCallback and useEffect .....	33
6.5 When useCallback helps.....	34
6.6 When useCallback hurts.....	34
6.7 The correct mental model.....	34
<b>Chapter 7.....</b>	<b>35</b>
<b>useMemo: Derived State, Identity, and When Optimization Is Real.....</b>	<b>35</b>
7.1 What useMemo actually does .....	35
7.2 Derived state vs stored state .....	35
7.3 Identity is the real reason to use useMemo .....	36
7.4 When useMemo is justified .....	37
7.5 useMemo is not a guarantee of performance .....	37
7.6 useMemo vs useCallback.....	37
7.7 Mental model for useMemo.....	38
<b>Chapter 8.....</b>	<b>39</b>
<b>useRef: Mutable Values Without Re-renders .....</b>	<b>39</b>

8.1 What <code>useRef</code> provides .....	39
8.2 What <code>useRef</code> is for.....	40
8.3 Tracking previous values .....	40
8.4 Why refs do not cause re-renders.....	40
8.5 Refs vs state .....	41
8.6 Refs and effects .....	41
8.7 When NOT to use <code>useRef</code> .....	41
8.8 Mental model for <code>useRef</code> .....	41
<b>Chapter 9.....</b>	<b>42</b>
<b>useLayoutEffect: Layout Timing and When Blocking Is Justified.....</b>	<b>42</b>
9.1 How <code>useLayoutEffect</code> differs from <code>useEffect</code> .....	42
9.2 Why blocking paint is dangerous .....	42
9.3 Legitimate use cases.....	43
9.4 React Native considerations.....	43
9.5 <code>useLayoutEffect</code> is not for data logic .....	43
9.6 Mental model for <code>useLayoutEffect</code> .....	44
<b>Chapter 10.....</b>	<b>45</b>
<b>useReducer: Structured State Transitions and Predictability.....</b>	<b>45</b>
10.1 When <code>useState</code> stops scaling .....	45
10.2 What <code>useReducer</code> provides.....	45
Figure 10.1 React <code>useReducer</code> hook explained visually.....	46
10.3 Reducers describe transitions, not effects .....	46
10.4 Actions as intent .....	46
10.5 Why <code>useReducer</code> improves reasoning .....	47
10.6 <code>useReducer</code> and hooks composition.....	47
10.7 When NOT to use <code>useReducer</code> .....	47
10.8 Mental model for <code>useReducer</code> .....	48
<b>Chapter 11.....</b>	<b>49</b>
<b>Custom Hooks: Extracting Logic Without Losing Control .....</b>	<b>49</b>
11.1 What a custom hook really is.....	49
11.2 What should go into a custom hook .....	49
11.3 Custom hooks do not share state by default .....	50
11.4 Dependency hygiene inside custom hooks.....	50

11.5 Returning APIs, not implementation details.....	50
11.6 Avoiding abstraction leaks.....	51
11.7 Mental model for custom hooks.....	51
<b>Chapter 12.....</b>	<b>52</b>
<b>Real React Native Patterns: Performance, Effects, and Lifecycle.....</b>	<b>52</b>
12.1 FlatList performance patterns.....	52
12.2 Autosave with debounce.....	52
12.3 Navigation-driven lifecycles.....	53
12.4 Avoiding stale async updates.....	53
12.5 Separating data, logic, and UI.....	53
12.6 Performance myths.....	54
12.7 Mental checklist for production screens.....	54
<b>Chapter 13.....</b>	<b>55</b>
<b>When Not to Use Hooks: Avoiding Over-Abstraction.....</b>	<b>55</b>
13.1 Not everything needs state.....	55
13.2 Not every effect belongs in useEffect.....	55
13.3 Custom hooks can hide more than they reveal.....	55
13.4 Avoid hook-driven architecture for simple components.....	56
13.5 Mental model for restraint.....	56
<b>Chapter 14.....</b>	<b>57</b>
<b>Debugging Hooks: Reasoning From Snapshots, Not Guessing.....</b>	<b>57</b>
14.1 Start from the render snapshot.....	57
14.2 Debugging stale closures.....	57
14.3 Debugging infinite loops.....	58
14.4 Using logs effectively.....	58
14.5 Trust the model, not intuition.....	58
<b>Chapter 15.....</b>	<b>59</b>
<b>Production Checklist and Closing Thoughts.....</b>	<b>59</b>
15.1 Production readiness checklist.....	59
15.2 What hooks ultimately give you.....	59
15.3 The long-term benefit.....	59
15.4 Final thought.....	60
<b>Phase Mapping Cheat Sheet.....</b>	<b>62</b>

# Chapter 1

## How React Really Renders

To understand React Hooks, it is essential to first understand **how React renders**. Most confusion around hooks does not come from the hooks themselves, but from incorrect assumptions about rendering, state, and time.

### 1.1 Rendering is not updating the screen

In React (including React Native), **rendering does not mean drawing pixels**.

Rendering means:

Running your component function to calculate what the UI *should look like*.

When React renders a component:

- It **calls the function**
- Executes all hooks **in order**
- Produces a **virtual description** of the UI

At this stage:

- No native views are touched
- No side effects run
- No layout happens

Rendering is **pure computation**.

---

### 1.2 The three phases of a render cycle

Every update in React goes through three conceptual phases:

#### 1) Render phase

- Component functions run
- Hooks are evaluated
- JSX is returned

This phase must be:

- Pure
- Deterministic
- Free of side effects

## 2) Commit phase

- React applies changes to the native UI
- Views are created, updated, or removed
- Layout happens

This phase is synchronous with the UI.

## 3) Effect phase

- `useEffect` callbacks run
- Subscriptions start
- Async work begins

This phase happens **after** the UI is visible.

Hooks are tightly mapped to these phases:

- `useState`, `useMemo`, `useCallback` → render phase
  - `useEffect` → effect phase
  - `useLayoutEffect` → between commit and paint
- 

## 1.3 A component is re-executed, not resumed

A critical concept:

React does **not** pause and resume your component — it **re-runs it from the top**.

Every render:

- Recreates local variables
- Recreates functions
- Re-evaluates expressions

Example:

```
function Screen() {  
  const count = 0;  
  return null;  
}
```

On every render, `count` is **redeclared**.

So how does state persist?

👉 **Hooks store state outside the function**, indexed by call order.

This is why:

- Hooks must be called unconditionally
  - Hook order must never change
- 

## 1.4 Renders are snapshots in time

Each render is a **snapshot** of:

- Props
- State
- Variables
- Closures

Once a render finishes:

- Its values never change
- They become “frozen in time”

If a state update happens:

- React schedules a **new render**
- A **new snapshot** is created

This explains why:

```
setCount(1);  
console.log(count); // old value
```

The count you log belongs to the **previous snapshot**.

---

## 1.5 Why this matters for hooks

Hooks are built on top of this model:

- `useState` selects which snapshot to render next
- `useEffect` runs *after* a snapshot is committed
- Closures capture values from the snapshot they were created in

Every hook rule exists to preserve this consistency.

If you understand rendering as **pure re-execution + snapshots**, most hook behavior becomes predictable.

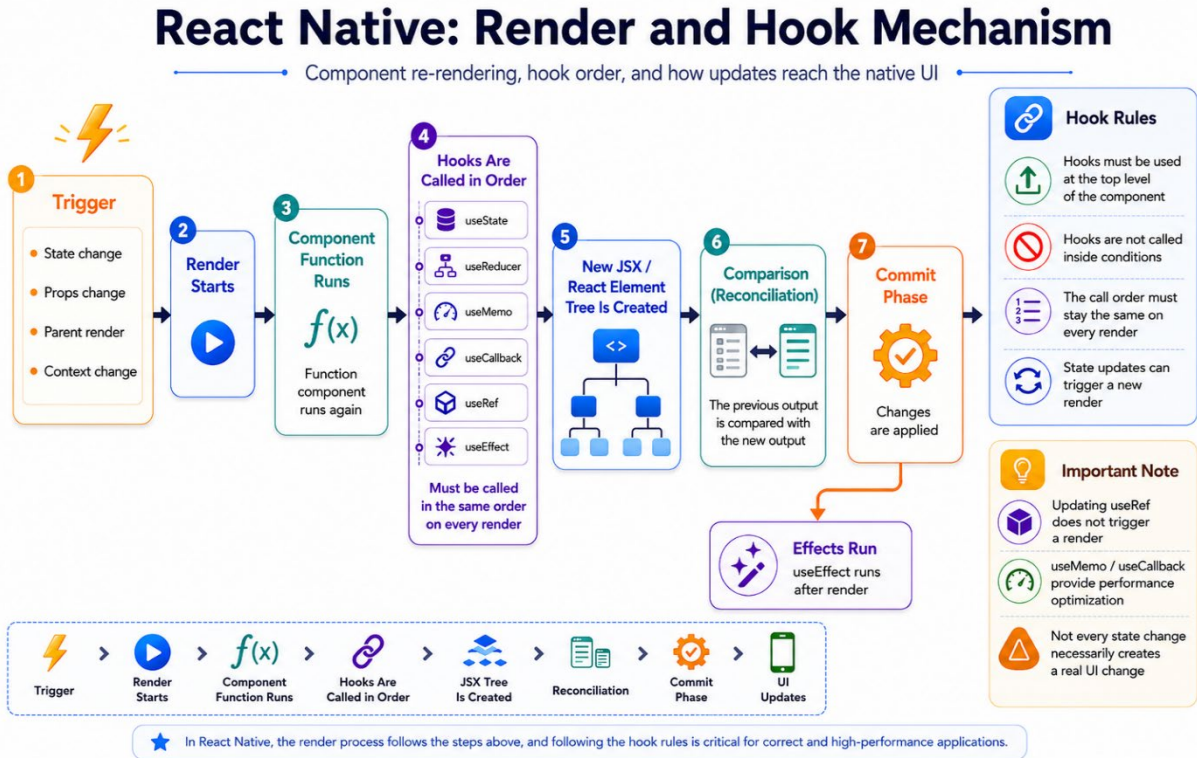
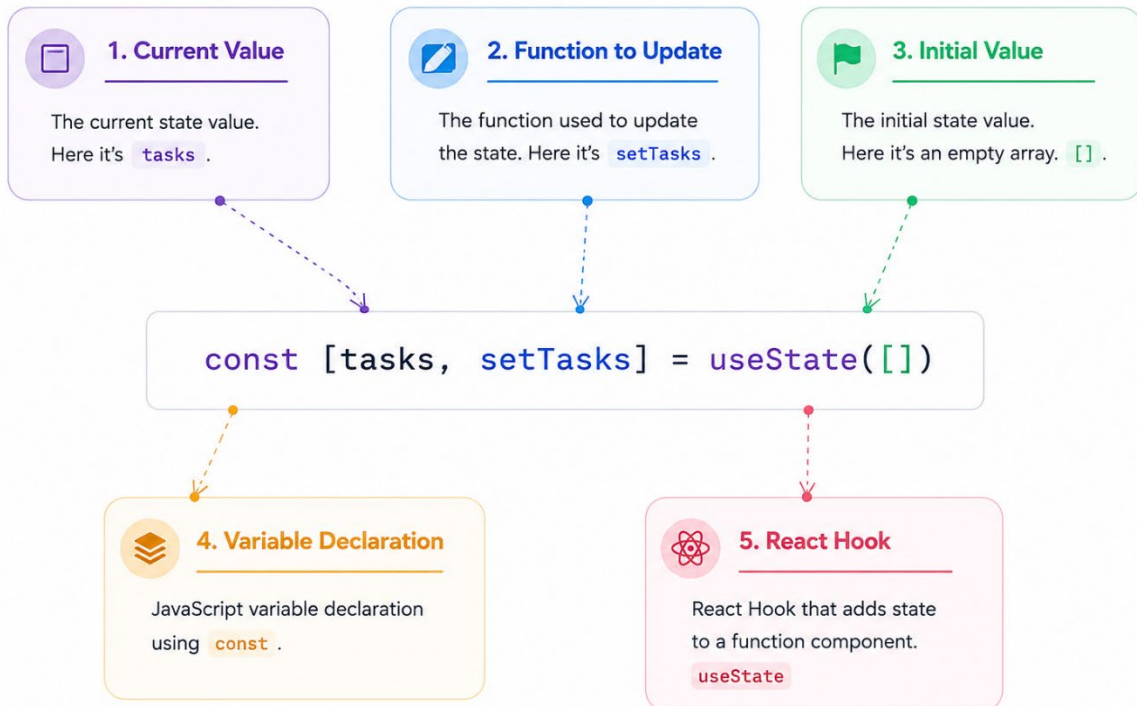


Figure 1.1 Render and hooks mechanism in React Native.

# Chapter 4

## useState: State, Scheduling, and Batching

`useState` is the foundation of all hook-based logic. It is also the most commonly misunderstood hook.



**Figure 4.1** `useState` variables declaration.

### 4.1 What `useState` really does

```
const [value, setValue] = useState(initialValue);
```

This does **not** create a variable.

It creates:

- A state slot stored outside the component
- A setter that schedules a new render

The `initialValue`:

- Is used **only on the first render**
- Is ignored on subsequent renders

### 4.2 State updates are scheduled, not applied

Calling `setState` does not change the current value.  
Instead:

1. React queues the update
2. React schedules a re-render
3. The next render receives the new value

This explains why:

```
setCount(1);  
console.log(count); // old value
```

The log belongs to the **current snapshot**, not the next one.

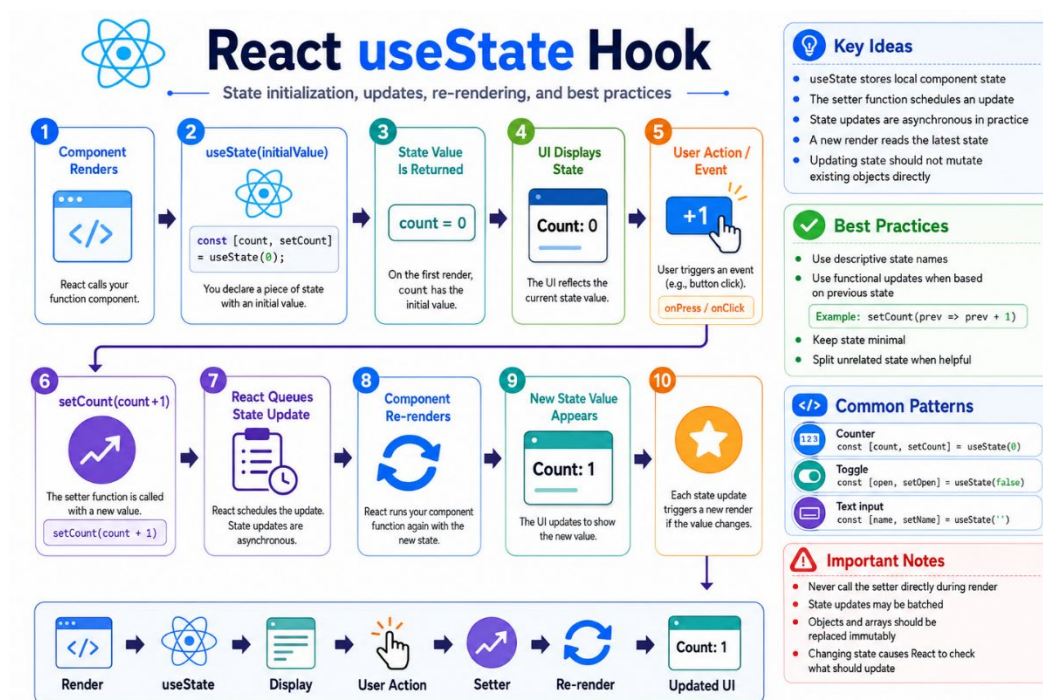


Figure 4.2 useState hook mechanism in React Native.

### 4.3 Batching: why updates appear “delayed”

React batches multiple updates together for performance.

```
setCount(1);  
setCount(2);
```

In most cases:

- Only one render happens
- The last update wins

Batching:

- Reduces unnecessary renders
- Keeps UI responsive
- Enables concurrency

You should never rely on intermediate state during the same render cycle.

#### 4.4 Functional updates: the correct mental model

When new state depends on old state, this is mandatory:

```
setCount(prev => prev + 1);
```

Why this works:

- React passes the **latest committed state**
- Independent of render timing
- Safe under batching and concurrency

This is not an optimization — it is **correctness**.

#### 4.5 Objects and arrays: immutability is not optional

State identity matters.

 Incorrect:

```
state.value = 1;  
setState(state);
```

Why this fails:

- Identity does not change
- React sees “same object”
- No meaningful update detected

 Correct:

```
setState(prev => ({  
  ...prev,  
  value: 1,  
}));
```

Every update must:

- Create a new object or array
- Preserve unchanged fields
- Change identity intentionally

#### 4.6 Why state sometimes “resets”

State resets when:

- A component unmounts
- A key changes
- A different component replaces it

React ties state to:

Component identity in the tree, not variable names.

Changing structure changes identity.

This is expected behavior — not a bug.

#### **4.7 When not to use `useState`**

Do not use `useState` for:

- Derived values → use computation
- Cached values → `useMemo`
- Mutable references → `useRef`
- Global state → context or external store

Overusing state increases re-render complexity.

#### **4.8 `useState` in the lifecycle**

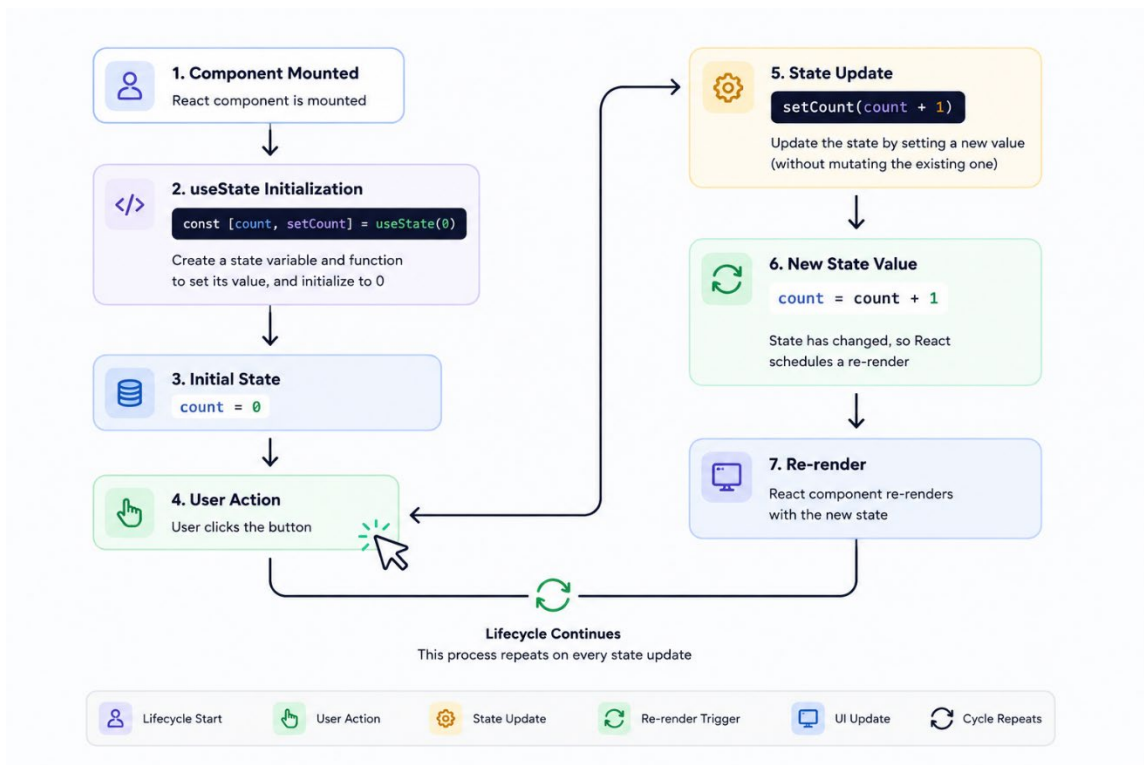
In a typical screen:

- Initial render → default state
- Effect runs → async update
- State update → re-render
- UI reflects new snapshot

State drives rendering.

Effects respond to state changes.

Never invert this relationship.



**Figure 4.3** How `useState` triggers re-renders.