1. Problem Statement

Case Study: StreamVision Video Dashboard

StreamVision is a video analytics dashboard:

- It displays live video feeds, analytics charts, and user comments.
- Some components (like charts and video overlays) are expensive to render or compute.
- User interactions (like filtering comments or toggling overlays) can cause unnecessary re-renders, slowing down the UI.
- The team wants to optimize performance, especially as the dashboard grows more complex and data-intensive.

Case Study: StreamVision Video Dashboard
The challenge:
How do you prevent unnecessary recalculations and re-renders in React, ensuring the UI remains fast–even as state and props change frequently?

The challenge:
How do you prevent unnecessary recalculations and re-renders in React, ensuring the UI remains fast—even as state and props change frequently?

2. Learning Objectives

By the end of this tutorial, you will:

- Understand what memoization is and why it matters in React.
- Use `useMemo` to cache expensive computed values.
- Use `useCallback` to memoize event handlers and callbacks.
- Use `React.memo` to prevent unnecessary re-renders of functional components.

- Combine these techniques for optimal performance in real-world apps.
- Avoid common pitfalls (over-memoization, stale closures).

---

3. Concept Introduction with Analogy

Analogy: The StreamVision Control Room

- **useMemo** is like a results whiteboard: If you've already done a complex calculation (like analyzing a video feed), you write the result on the board and reuse it until the inputs change.
- **useCallback** is like a phone directory: You keep the same phone number (function reference) for callbacks, so you don't have to reprint the directory every time someone's name changes.
- **React.memo** is like a smart camera operator: They only update the camera angle (re-render) if the scene actually changes, not just because someone walked into the control room.

---

4. Technical Deep Dive

A. What Is Memoization in React?

- **Memoization** is the process of caching the result of a function so it doesn't need to be recomputed unless its inputs change.
- In React, memoization prevents unnecessary recalculations and re-renders, improving performance—especially in large or complex UIs.

B. useMemo: Memoizing Expensive Computations

When to Use

- When you have a computation that is expensive (heavy calculation, large filtering, etc.).
- When the computed value is used in rendering and only depends on specific props or state.

Syntax

```
import React, { useMemo } from 'react';

const ExpensiveChart = ({ data }) => {
  const processedData = useMemo(() => {
    // Heavy computation here
    return computeAnalytics(data);
  }, [data]); // Only recompute if data changes

  return <Chart data={processedData} />;
};
```

How It Works

- `useMemo` takes a function and a dependency array.
- It only recomputes the value if dependencies change.
- Otherwise, it returns the cached value from the last render.

Pitfalls

- Don't use `useMemo` for every value—only for expensive computations.
- If dependencies are unstable (e.g., new object/array each render), memoization is ineffective.

C. useCallback: Memoizing Functions and Event Handlers

When to Use

- When passing callbacks to child components that are memoized (e.g., with `React.memo`).
- When the callback is used in a dependency array (e.g., in `useEffect`).

Syntax

```
import React, { useCallback } from 'react';

const VideoControls = ({ onPlay, onPause }) => (
 <div>
   <button onClick={onPlay}>Play</button>
   <button onClick={onPause}>Pause</button>
 </div>
);

const Dashboard = () => {
 const [playing, setPlaying] = React.useState(false);

 const handlePlay = useCallback(() => setPlaying(true), []);
 const handlePause = useCallback(() => setPlaying(false), []);

 return <VideoControls onPlay={handlePlay} onPause={handlePause} />;
};
```

How It Works

- `useCallback` returns the same function reference unless dependencies change.
- Prevents child components from re-rendering due to new function props.

Pitfalls

- Overusing `useCallback` can add complexity with little benefit if the function is cheap or the child isn't memoized.
- Be careful with dependencies—stale closures can occur if dependencies are missing.

D. React.memo: Memoizing Functional Components

When to Use

- For pure functional components that render the same output given the same props.
- To prevent re-rendering unless props actually change.

Syntax

```
import React from 'react';

const CommentList = React.memo(({ comments }) => {
  console.log('Rendering CommentList');
  return (
    <ul>
```

```
      {comments.map((c) => (
        <li key={c.id}>{c.text}</li>
      ))}
    </ul>
  );
});
```

How It Works

- `React.memo` wraps a component and only re-renders it if its props change (shallow comparison).
- You can provide a custom comparison function for complex props.

Pitfalls

- If props are new objects/arrays each render, memoization won't help—use `useMemo` or `useCallback` to stabilize them.
- Not useful for components with side effects or non-deterministic rendering.

E. Combining All Three for Maximum Performance

- Use `useMemo` for expensive values.
- Use `useCallback` for event handlers passed to memoized children.
- Use `React.memo` for pure, presentational components.

---

5. Step-by-Step Data Modeling & Code Walkthrough

A. Memoizing Expensive Chart Data

```
import React, { useMemo } from 'react';

function computeAnalytics(data) {
  // Simulate heavy computation
  return data.reduce((acc, item) => acc + item.value, 0);
}

const AnalyticsChart = ({ data }) => {
  const analytics = useMemo(() => computeAnalytics(data), [data]);
  return <div>Analytics Value: {analytics}</div>;
};
```

B. Memoizing Event Handlers with useCallback

```
import React, { useCallback, useState } from 'react';

const FilterInput = React.memo(({ onFilter }) => {
  return <input onChange={e => onFilter(e.target.value)} placeholder="Filter comments..." />;
});

const CommentsPanel = ({ comments }) => {
  const [filter, setFilter] = useState('');
  const filtered = useMemo(
    () => comments.filter(c => c.text.includes(filter)),
    [comments, filter]
  );

  // Memoize setFilter to avoid unnecessary re-renders of FilterInput
  const handleFilter = useCallback(setFilter, []);
  return (
    <div>
      <FilterInput onFilter={handleFilter} />
      <ul>
        {filtered.map(c => <li key={c.id}>{c.text}</li>)}
      </ul>
    </div>
  );
};
```

C. Memoizing Components with React.memo

```
const VideoOverlay = React.memo(({ overlays }) => {
  return (
    <div>
      {overlays.map(o => (
        <span key={o.id}>{o.label}</span>
      ))}
    </div>
  );
});
```

6. Interactive Challenge / Mini-Project

Your Turn!

1. Create a `TagList` component that:
   - Receives a list of tags and a filter string.
   - Uses `useMemo` to compute the filtered list.
   - Is wrapped in `React.memo` to avoid unnecessary re-renders.
2. Create a `TagInput` component that:
   - Accepts a memoized `onAddTag` callback via `useCallback` .
   - Only re-renders when the callback or input value changes.
3. Show how changing unrelated state in the parent does **not** re-render the memoized `TagList` or `TagInput` .

7. Common Pitfalls & Best Practices

Common Pitfalls & Best Practices (React Memoization)

Pitfall	Best Practice
Overusing memoization	Only memoize expensive or frequently-changing values
Unstable dependencies	Use stable references for objects/arrays
Missing dependencies in hooks	Always include all dependencies
Stale closures in callbacks	Ensure dependencies are up-to-date
Memoizing impure or side-effectful functions	Only memoize pure computations

8. Optional: Programmer’s Workflow Checklist

- Use `useMemo` for expensive computations in render.
- Use `useCallback` for event handlers passed to memoized children.
- Use `React.memo` for pure, presentational components.
- Profile with React DevTools to find real performance bottlenecks.
- Avoid memoizing everything—measure before optimizing.