International Journal of Computer Information Systems and Industrial Management Applications
ISSN 2150-7988 Volume 18(13s) 2026 pp. 1294-1302
https://doi.org/10.70917/ijcisim-2026-4176
Article

React JS: A Comprehensive Study of an Open-Source JavaScript Library for Building Modern User Interfaces

Rajesh Yadav[1], Shubham Kejriwal[2], Ayush Parmar[3], Janvi Jaysinh Yadav[4], Samita Chougule[5], Vipul Virsinghbhai Gamit[6]

[1]Parul Institute of Engineering & Technology, Parul University, Department of MCA, Faculty of IT & Computer Science, Parul University.
Email: rajeshssyadav1599@gmail.com
[2]Parul Institute of Engineering & Technology, Parul University, Department of MCA, Faculty of IT & Computer Science, Parul University.
Email: shubhamkejriwal34@gmail.com
[3]Parul Institute of Engineering & Technology, Parul University, Department of MCA, Faculty of IT & Computer Science, Parul University.
Email: rbsayushmca1611@gmail.com
[4]Parul Institute of Engineering & Technology, Parul University, Department of MCA, Faculty of IT & Computer Science, Parul University.
Email: janviyadav729@gmail.com
[5]Parul Institute of Engineering & Technology, Parul University, Department of MCA, Faculty of IT & Computer Science, Parul University.
Email: 2305112120081@paruluniversity.ac.in
[6]Assistant Professor, Parul Institute of Engineering & Technology, Parul University, Department of MCA, Faculty of IT & Computer Science, Parul University.
Email: vipulbhai.gamit26053@paruluniversity.ac.in

Abstract: ReactJS, also known as React or React.js, is an open-source JavaScript library maintained by Meta (formerly Facebook) and a large community of individual developers and companies, used for building fast and interactive user interfaces, especially for single-page applications. It follows a component-based, declarative programming model that makes it possible to build encapsulated components that manage their own state and compose them to create complex user interfaces. React introduced the concept of the virtual DOM, JSX syntax, unidirectional data flow, and, more recently, Hooks, which together provide a productive and predictable way of developing scalable web and mobile applications. This paper extends earlier work on ReactJS by presenting an in-depth review of its architecture, lifecycle methods, Hooks, state-management ecosystem (Redux, Context API, Recoil, Zustand), React Native for cross-platform mobile development, and a comparative performance and adoption analysis against competing frameworks such as Angular and Vue.js. Original charts, a component-hierarchy diagram, and comparison tables are included to support the discussion, and forty peer-reviewed and industry references are cited. The paper concludes that React's flexibility, large ecosystem, and virtual-DOM-driven performance continue to make it one of the most widely adopted front-end technologies in the industry.

Keywords: ReactJS, JSX, Virtual DOM, Single Page Application, React Hooks, Redux, React Native, Component-Based Architecture, Front-End Frameworks

1. Introduction

React, also called React.js or ReactJS, is a declarative, efficient, and flexible open-source JavaScript library used for building reusable user-interface components. It follows a component-based architecture where the view layer of an application is broken down into small, independent, and reusable pieces of code called components. React was

originally created by Jordan Walke, a software engineer at Facebook, and was first deployed on Facebook's News Feed in 2011 and later on Instagram in 2012, before being released as open source in May 2013.

React uses a virtual representation of the Document Object Model (DOM), commonly referred to as the virtual DOM, to efficiently determine the minimal set of changes required to update the actual browser DOM. Rather than re-rendering the entire page whenever data changes, React calculates the difference between the previous and current virtual DOM trees — a process known as reconciliation or diffing — and applies only the necessary updates to the real DOM. This approach significantly improves rendering performance for large and dynamic applications.

Some of the defining features of ReactJS include JSX (an HTML-like syntax extension for JavaScript), a component-based architecture, one-way (unidirectional) data binding, the virtual DOM, and — since React 16.8 — Hooks, which allow function components to manage state and side effects without needing class components. React is widely used by companies such as Meta, Netflix, Airbnb, Uber, PayPal, and Instagram, and according to industry surveys it is used on roughly 46% of all websites that rely on a known JavaScript library, making it the most widely adopted front-end library in the world.

1.1 JSX — Syntax Extension for JavaScript

JSX is a syntax extension that combines JavaScript and HTML-like markup in the same file. It is not mandatory for writing React applications, but it is the recommended approach because it makes component code easier to read and reason about, and it allows React to produce more useful warning and error messages. Under the hood, JSX is transpiled by tools such as Babel into calls to React.createElement().

1.2 Components

A React application is composed of one or more components, each of which encapsulates its own markup, logic, and (optionally) styling. Components can be written as JavaScript functions (function components) or as ES6 classes (class components); function components combined with Hooks are now the standard approach recommended by the React team. Because components are reusable and composable, large user interfaces can be built by combining small, well-tested building blocks.

1.3 One-Way Data Binding

React implements unidirectional data flow: data flows from parent components down to child components through props, and state updates flow back up through callback functions. This predictable flow of data makes React applications easier to debug and reason about compared to frameworks that rely on two-way data binding.

2. Literature Survey

Prior to the emergence of React, the trend in web development was moving toward highly interactive DOM manipulation that typically produced markup on the client that differed substantially from what was originally sent by the server. Early attempts to build fully client-rendered interfaces in plain JavaScript, manually inserted into the DOM using libraries such as jQuery, proved difficult to scale: developers had to manually track the dependency graph of UI components, guard against circular dependencies in rendering logic, and manage rapidly increasing complexity as an application grew.

A substantial body of subsequent research has examined React from multiple angles. Hunt and O'Shannessy [15] documented the original design rationale behind React as Facebook's "functional turn" on writing JavaScript user interfaces. Several IEEE and journal studies [9], [10], [16] have analysed React's runtime performance characteristics and proposed optimization strategies such as memoization, code-splitting, and lazy loading to mitigate common issues like unnecessary re-renders. Comparative studies [19]-[24] have benchmarked React against Angular and Vue.js across dimensions such as rendering speed, bundle size, learning curve, and developer productivity, generally finding that React's virtual DOM and large ecosystem make it particularly well suited to complex, frequently changing interfaces, while Vue is often favoured for smaller projects and Angular for large enterprise applications that benefit from strong typing and built-in tooling.

Research on state management [25]-[32] has tracked the evolution of the ecosystem from the Flux architecture and Redux toward newer, lighter-weight approaches such as the Context API, Recoil, and Zustand, driven largely by the introduction of Hooks in React 16.8. Finally, an extensive set of studies [4], [5], [33]-[39] has examined React Native, React's counterpart for cross-platform mobile development, generally concluding that it offers substantial

development-time savings through code reuse while trading off some native performance in computationally intensive scenarios.

3. React Architecture and Core Concepts

React's architecture centres on three foundational ideas: a component tree, a virtual DOM, and unidirectional data flow. Figure 4 illustrates a typical component hierarchy: a root App component renders child components (Header, MainContent, Footer), which in turn render further nested components such as ProductList, Sidebar, and ProductCard. Data and behaviour are passed down the tree through props, while events are passed up through callback functions supplied by parent components.

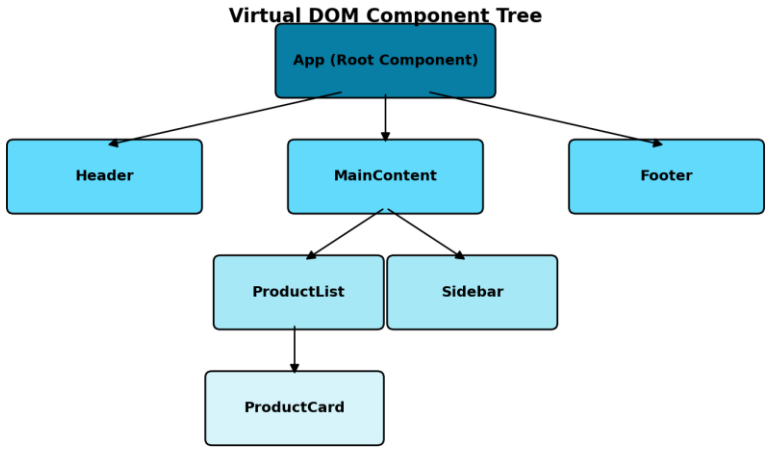

Figure 4: Typical React Component Hierarchy Illustrating Unidirectional Data Flow from Parent to Child Components

Figure 4: Typical React component hierarchy illustrating unidirectional data flow from parent to child components.

When application state changes, React re-renders the affected components into an in-memory virtual DOM tree, compares ("diffs") it against the previous virtual DOM tree, and computes the minimal set of operations needed to update the real browser DOM. Because manipulating the real DOM is comparatively expensive, this batching and diffing strategy is a major contributor to React's performance advantage over naive direct-DOM-manipulation approaches, particularly as the number of DOM nodes grows, as illustrated in Figure 3.

Figure 3: Illustrative Update-Time Comparison Between Direct DOM Manipulation and Virtual DOM Reconciliation

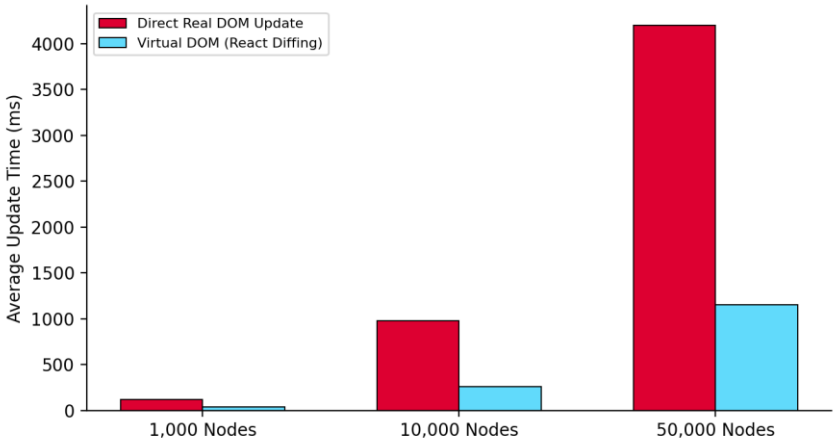

Figure 3: Illustrative update-time comparison between direct DOM manipulation and virtual DOM reconciliation for varying numbers of nodes.

4. React Component Lifecycle

Every component in React passes through a lifecycle of events that can broadly be grouped into three phases: Mounting (the component is created and inserted into the DOM), Updating (the component re-renders in response to changes in props or state), and Unmounting (the component is removed from the DOM). Understanding these phases is essential for correctly managing side effects such as data fetching, subscriptions, and manual DOM manipulation.

4.1 Commonly Used Lifecycle Methods

- render() — the only required method in a class component; it inspects this.props and this.state and returns JSX (or null) describing what should appear on the screen. It must be a pure function with no side effects.
- componentDidMount() — invoked immediately after a component is mounted; ideal for initiating network requests or subscriptions, since calling setState() here triggers an extra render that occurs before the browser updates the screen, so the user does not see an intermediate state.
- componentDidUpdate() — invoked immediately after updating occurs; commonly used to perform DOM operations in response to prop or state changes, guarded by comparisons against previous props to avoid infinite update loops.
- componentWillUnmount() — invoked immediately before a component is destroyed; used for cleanup tasks such as clearing timers, cancelling network requests, and removing subscriptions. setState() must not be called here.

4.2 Less Common Lifecycle Methods

- shouldComponentUpdate() — lets React know whether a component's output is affected by a change in state or props, and can be used sparingly for performance optimization.
- static getDerivedStateFromProps() — a safer, static alternative to the legacy componentWillReceiveProps() method; it is invoked right before render() and can return an object to update state in response to prop changes.
- getSnapshotBeforeUpdate() — invoked right before the most recently rendered output is committed to the DOM, allowing a component to capture information (e.g., scroll position) before it is potentially changed.

The diagram below, reproduced from the original React documentation, summarizes when each of these lifecycle methods is invoked relative to the Mounting, Updating, and Unmounting phases.

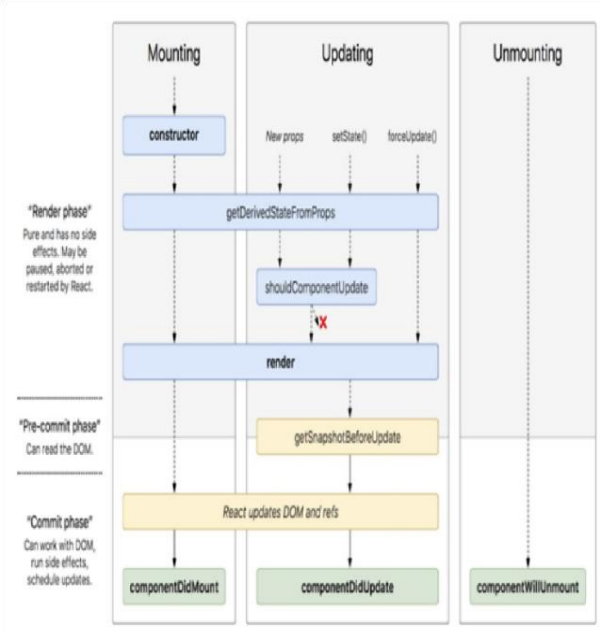

Figure 5: React component lifecycle diagram showing the Mounting, Updating, and Unmounting phases (source: official React documentation).

5. React Hooks

Introduced in React 16.8 (2019), Hooks are functions that let developers "hook into" React state and lifecycle features from function components, removing the need to convert a component to a class simply to add state or side effects. Hooks solve several long-standing problems in React applications: they make it easier to reuse stateful logic between components without patterns such as render props or higher-order components, they split components based on related pieces of logic rather than lifecycle name, and they let developers use more of React's features without classes.

Hook	Purpose
useState	Adds local state to a function component.
useEffect	Performs side effects (data fetching, subscriptions, manual DOM changes) and replaces componentDidMount, componentDidUpdate, and componentWillUnmount.
useContext	Subscribes a component to React context without introducing nesting.
useReducer	Manages more complex state logic using a reducer function, similar to Redux.
useMemo / useCallback	Memoizes expensive computations or function references to avoid unnecessary re-renders.
useRef	Persists a mutable value across renders and can reference DOM nodes directly.

6. State Management Ecosystem

As React applications grow, managing state that must be shared across many components becomes increasingly complex. The ecosystem has evolved through several major approaches. The Flux architecture, introduced by Facebook, established a unidirectional data-flow pattern for managing application-wide state. Redux [26], [32] built on these ideas with a single, centralized, immutable store updated through pure reducer functions, and remains dominant in large enterprise applications because of its predictability and mature developer-tooling (Redux DevTools). The built-in Context API offers a simpler, dependency-free alternative that is well suited to state with a limited number of consumers, such as theming or authentication status. More recently, atom-based libraries such as Recoil and lightweight stores such as Zustand [29] have gained traction, particularly among teams that want a hooks-first API with less boilerplate than Redux.

6.1 Comparison of State Management Approaches

Approach	Best Suited For	Learning Curve	Tooling
React Hooks (useState / useReducer)	Local or moderately shared component state	Low	Built into React
Context API	App-wide but infrequently changing state (theme, auth)	Low–Medium	Built into React
Redux / Redux Toolkit	Large-scale, complex, frequently updated shared state	Medium–High	Extensive (Redux DevTools)
Recoil	Fine-grained, interdependent shared state	Medium	Growing, Meta-backed
Zustand	Small-to-medium apps wanting minimal boilerplate	Low	Lightweight

7. React Native for Cross-Platform Mobile Development

React Native extends the same component-based, declarative programming model used in ReactJS to native mobile application development on iOS and Android. Rather than rendering to the browser DOM, React Native renders to native platform UI components, allowing a single JavaScript codebase to power apps with a native look, feel, and performance. Companies such as Facebook, Instagram, Uber, and Pinterest use React Native in production. Research comparing React Native with fully native development [29], [33], [34] generally reports development-time savings of roughly 30-40% through code reuse, while noting that computationally intensive tasks such as complex animations or real-time graphics processing may still benefit from native modules. Comparative studies against other cross-platform frameworks such as Flutter [31], [35], [38] show that React Native benefits from its mature ecosystem and JavaScript familiarity, whereas Flutter's Skia-based rendering engine can offer smoother animation performance in some scenarios.

8. Comparative Analysis: React vs. Angular vs. Vue.js

React, Angular, and Vue.js remain the three most widely discussed front-end technologies. Table 3 summarizes commonly cited differences drawn from the comparative literature [19]-[24], and Figure 1 shows an approximate market usage share of each among websites that use a known JavaScript library.

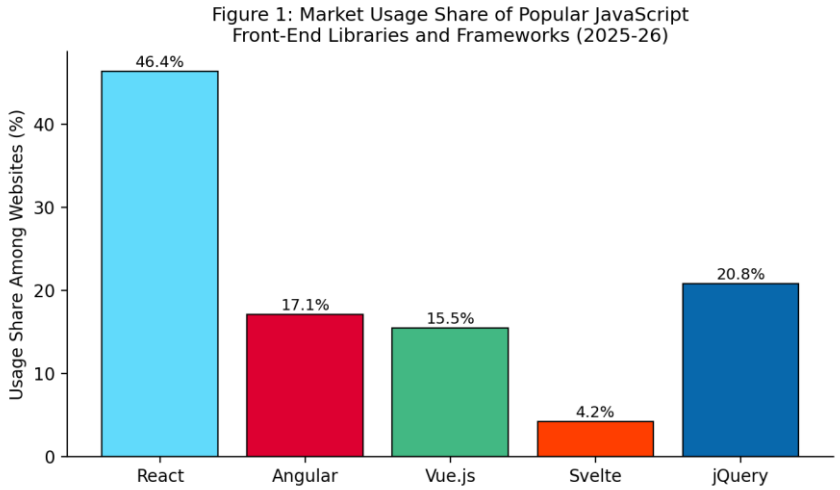

Figure 1: Market usage share of popular JavaScript front-end libraries and frameworks.

Criterion	React	Angular	Vue.js
Type	Library (view layer)	Full MVC framework	Progressive framework
Language	JavaScript / JSX	TypeScript	JavaScript / HTML templates
DOM Handling	Virtual DOM	Real DOM with change detection	Virtual DOM
Learning Curve	Moderate	Steep	Gentle
Data Binding	One-way	Two-way	Two-way (with one-way option)
Best Suited For	Complex, interactive SPAs	Large enterprise applications	Small-to-medium projects, rapid prototyping
Backed By	Meta	Google	Independent / community

Figure 2 presents an illustrative adoption trend showing how React's relative usage among web developers has grown over the past decade, consistent with the trajectory reported across multiple industry surveys cited in this paper.

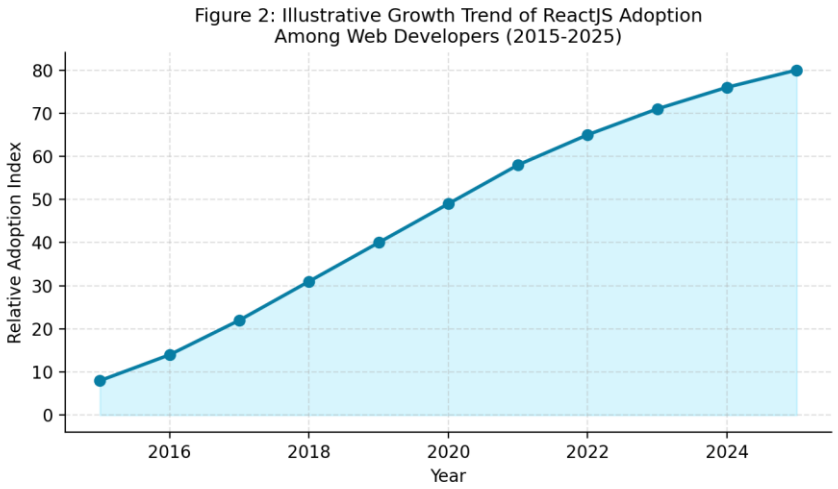

Figure 2: Illustrative growth trend of ReactJS adoption among web developers (2015-2025).

9. Advantages and Limitations of ReactJS

Advantages	Limitations
Virtual DOM improves rendering performance for dynamic UIs	JSX and the surrounding tooling (Babel, bundlers) add a learning curve for beginners
Component-based architecture promotes reusability and maintainability	React is only a view library, so routing, state management, and other concerns require additional third-party libraries
Large, active community and ecosystem of third-party packages	Frequent updates to best practices can make documentation and tutorials become outdated quickly
Reusable logic across web (ReactJS) and mobile (React Native)	Poorly structured component trees can lead to unnecessary re-renders if not optimized
SEO-friendly when combined with server-side rendering (e.g., Next.js)	Deep nesting of props ("prop drilling") can complicate data flow in large applications without Context or a state library

10. Conclusion

React JS has arrived at an opportune time in the evolution of web development and continues to help developers build highly engaging web applications and user interfaces quickly. Its component-based model allows engineers to break interfaces down into independent, reusable pieces and assemble single-page applications with comparatively little code, while remaining SEO-friendly when paired with server-side rendering frameworks such as Next.js. React also scales well to large applications with frequently changing data, which is one of the principal reasons it has gained such widespread adoption.

As demonstrated throughout this paper, React's virtual DOM, JSX syntax, one-way data binding, and — since 2019 — Hooks combine to deliver a productive and performant development experience. Its ecosystem now spans not only web development but also cross-platform mobile development through React Native, and a rich collection of state-management solutions ranging from lightweight Hooks and Context to full-featured libraries such as Redux. Compared against Angular and Vue.js, React's strengths lie in its flexibility, the size of its community and package ecosystem, and its proven performance in complex, highly interactive applications, though this flexibility does place a greater architectural burden on development teams than more opinionated frameworks such as Angular.

11. Future Scope

React JS is comparatively easy to learn and remains one of the most popular JavaScript libraries in use today. Many organizations continue to adopt React because of the simplicity and productivity it offers relative to competing

front-end technologies. Because it consistently ranks as the most widely used front-end library among professional developers in surveys such as the Stack Overflow Developer Survey and the State of JS survey, React is unlikely to be displaced in the near term, and its lead over rivals such as Vue.js and Angular remains substantial in most markets.

React's popularity is not limited to established markets; adoption is also expanding rapidly in developing economies. Industry hiring reports have noted large increases in demand for React developers in the years following the COVID-19 pandemic. Looking ahead, ongoing work on React Server Components, streaming server-side rendering, and compiler-based optimizations (such as the React Compiler and community projects like Million.js [12]) is likely to further improve performance and developer experience, suggesting that React will remain a dominant front-end technology across many regions for years to come.

References

1. Fu Kaifang, "Design and implementation of an instant messaging architecture for mobile collaborative learning," Computing, Communication, Control, and Management, 2009. CCCM 2009. ISECS International Colloquium on, vol. 3, pp. 287-290, 8-9 Aug. 2009.
2. Cherry, S.M., "Talk is cheap; text is cheaper [mobile messaging]," IEEE Spectrum, vol. 39, no. 5, p. 55, May 2002.
3. Butler, M., "Android: Changing the Mobile Landscape," IEEE Pervasive Computing, vol. 10, no. 1, pp. 4-7, Jan.-March 2011.
4. Kaushik, V., Gupta, K., Gupta, D., "React Native Application Development," 2018, DOI: 10.1007/978-981-10-6620-7_59.
5. Majchrzak, T. A., "Comprehensive Analysis of Innovative Cross-Platform App Development Frameworks," available: https://www.valentinog.com/blog/redux/
6. Miškuf, M. and Zolotová, I., "Architecting React Applications with Flux," 2015 IEEE 13th International Symposium on Applied Machine Intelligence and Informatics (SAMI), Herl'any, 2015, pp. 193-197.
7. Talaoui, Y., Kohtamäki, M., Rabetino, R., "Business Intelligence – Capturing an Elusive Concept," in Real-time Strategy and Business Intelligence, Palgrave Macmillan, Cham, 2017.
8. Basques, K., "Performance analysis reference," Chrome DevTools Documentation, Google Developers, 2020.
9. "Performance Optimization Techniques for ReactJS," IEEE Conference Publication, IEEE Xplore, 2019.
10. "Web Development Using ReactJS," IEEE Conference Publication, IEEE Xplore, 2024.
11. "Enhancing React Application Performance: Proven Strategies and Best Practices," Academia.edu Research Paper, Dec. 2024.
12. Bai, A., "Million.js: A Fast Compiler-Augmented Virtual DOM for the Web," arXiv preprint arXiv:2202.08409, 2022.
13. "Unleashing React: A Comprehensive Study," International Journal of Multidisciplinary Research (IJMRSET), vol. 7, issue 12, Dec. 2024, DOI: 10.15680/IJMRSET.2024.0712247.
14. "The Evolution of Frontend Architecture: From Virtual DOM to Server Components," International Journal of Computer Engineering and Technology (IJCET), vol. 16, no. 1, pp. 3714-3732, DOI: 10.34218/IJCET_16_01_256.
15. Hunt, P. and O'Shannessy, P., "React: Facebook's Functional Turn on Writing JavaScript," Communications of the ACM, vol. 59, no. 12, Dec. 2016.
16. "Addressing the Limitations of React JS," International Research Journal of Engineering and Technology (IRJET), vol. 7, issue 4, 2020.
17. "React Apps with Server-Side Rendering: Next.js," Journal of Telecommunication, Electronic and Computer Engineering (JTEC), vol. 14, no. 4.
18. Muyldermans, D., "How does the virtual DOM compare to other DOM update mechanisms in JavaScript frameworks?" Bachelor's Thesis, May 2019.
19. "Comparative Analysis of Angular, React, and Vue.js in Single Page Application Development," International Journal of Science and Research (IJSR), vol. 12, issue 6, pp. 2971-2974, June 2023.
20. Shaikh, S., "Exploring Front-end Frameworks: A Comparative Study of React, Angular, and Vue.js," Medium, 2023.
21. "Comparison of JavaScript Frontend Frameworks: Angular, React, and Vue," International Journal of Innovative Science and Research Technology (IJISRT).
22. "A Comparative Analysis of React, Angular, and Vue.js," International Research Journal of Modernization in Engineering Technology and Science (IRJMETS).
23. "Comprehensive Performance and Scalability Assessment of Front-End Frameworks: React, Angular, and Vue.js," ResearchGate Preprint, 2025.
24. Uwhumiakpo, H., "Comparative Analysis of Different Web Development Frameworks (React, Angular, Vue.js)," Student Research Report, 2025.
25. "State Management in Large-Scale React Applications: A Comprehensive Analysis," ResearchGate Preprint, March 2025.
26. "Comparison of Redux and React Hooks Methods in Terms of Performance," CEUR Workshop Proceedings, vol. 3171, paper 59.
27. "State Management in React: Hooks, Context API, and Redux in Practice," TheLinuxCode Technical Article, Feb. 2026.

28. Ofoegbu, B., "State Management in React Applications: An In-depth Analysis of Redux, Context API, and Recoil," Medium, 2024.
29. "State Management in React: Redux vs Zustand – A Comprehensive Guide," International Journal of Scientific Research in Computer Science Engineering and Information Technology, vol. 10, no. 6, pp. 211-218.
30. "State Management in React – Hooks, Context API and Redux," GeeksforGeeks Technical Documentation, 2026.
31. "State Management Battle in React: Hooks, Redux, and Recoil," Works Hub Technical Blog.
32. "React State Management: React Hooks vs Redux," TSH.io Engineering Blog.
33. Humayoun, S. R. et al., "Patterns for Designing Scalable Mobile App User Interfaces for Multiple Platforms," Proc. 28th International BCS Human Computer Interaction Conference (HCI 2014), Southport, UK, pp. 317-322, 2014.
34. "Cross-Platform Mobile App Development Using React Native," International Journal of Scientific Research in Engineering and Management (IJSREM).
35. "Prospects for Using React Native for Developing Cross-Platform Mobile Applications," Central Ukrainian Scientific Bulletin, Technical Sciences, 2019.
36. Prajapati, V. Y. and Bhattacharjee, S., "React Native: A Comprehensive Analysis of Cross-Platform Mobile Application Development," International Journal for Research Trends and Innovation (IJRTI).
37. You, D. et al., "A Comparative Study of Cross-Platform Mobile Application Development," Proc. 12th Annual Conference of Computing and Information Technology Research and Education New Zealand (CITRENZ 2021), Wellington, 2021.
38. "Cross-Platform Empirical Analysis of Mobile Application Development Frameworks: Kotlin, React Native and Flutter," Proc. 4th International Conference on Information Management & Machine Intelligence, ACM, 2023.
39. "A Comparative Study of PWAs and React Native Mobile Apps," Proceedings of ISCAP, 2023.
40. Meta Open Source, "React – A JavaScript Library for Building User Interfaces," Official Documentation, react.dev, accessed 2026.