The React Native Story

From a Facebook Hack Week Experiment to the World's Most Deployed Cross-Platform Mobile Framework

Abstract

React Native was born from a two-week internal hackathon at Facebook in 2013. It was not a product plan — it was a question: what if we could build truly native mobile apps using the same mental model as React on the web? A decade later, that question has produced the most widely adopted cross-platform mobile framework in the world, running in production at companies from Microsoft to Shopify to Discord. This white paper traces the full arc: the problem with mobile development before React Native, the origin and architecture, the painful early years, the major architectural reinvention underway today, and where the framework is heading. It is written for developers and engineering leaders who want the historical and technical context behind one of the most consequential decisions in mobile development.

Published by

H4 Tech SOLUTIONS

www.h4tech.com

2026 · Version 1.0

Contents

1. The Mobile Development Problem — Before React Native

To understand why React Native exists, you need to understand how painful mobile development was in the early 2010s. The smartphone era had exploded, and companies needed to ship on two completely separate platforms: iOS and Android. Those platforms did not merely use different programming languages — they used different paradigms, different UI component models, different build systems, different testing tools, and different deployment pipelines.

An iOS developer wrote Objective-C (and later Swift) in Xcode, building UIKit interfaces with Auto Layout constraints and Interface Builder. An Android developer wrote Java (and later Kotlin) in Android Studio, building layouts in XML with ConstraintLayout and fragment transactions. The two had almost nothing in common. A company that wanted to ship the same feature on both platforms needed two separate teams, two separate codebases, two separate review processes — and then, inevitably, two interfaces that slowly diverged from each other as each team made independent decisions.

The Hidden Cost of Dual-Platform Development

For Facebook in 2013, this was not an academic problem. The company had grown from a web-first product into a mobile-first company almost overnight — by 2013, more than half of Facebook's daily active users were accessing the service exclusively on mobile. Maintaining feature parity between a web codebase (written by hundreds of JavaScript engineers) and two separate native mobile codebases (written by a much smaller team of iOS and Android specialists) was a significant organisational drag. The people who knew how to build React components for the web were not, by default, able to build native mobile features.

The industry's previous answers to this problem had been unsatisfying. PhoneGap (later Apache Cordova) wrapped web content in a native browser shell — the apps looked like web pages because they were web pages, and users noticed. Xamarin allowed C# developers to share business logic across platforms, but UI code still had to be written separately. Titanium had similar limitations. None of these approached the performance, fluidity, and native feel that iOS and Android users had come to expect from first-party apps.

The fundamental trade-off seemed fixed: either you used web technologies and sacrificed the native feel, or you used native technologies and sacrificed the shared codebase. React Native was built on the premise that this trade-off was not inevitable.

2. The Origin Story — A Hackathon and a Question

The story of React Native begins not with React Native but with React. In 2011, Jordan Walke, a software engineer at Facebook, began experimenting with a new approach to building user interfaces — a declarative, component-based model where the UI was a pure function of application state. He built an early prototype he called FaxJS. The core idea was radical in its simplicity: describe what the UI should look like for a given state, and let the framework figure out the minimum set of DOM operations needed to get there.

Facebook started using this internally, first for the News Feed, then more broadly. In 2013, Tom Occhino and Jordan Walke presented React publicly at JSConf US. The reaction from the JavaScript community was, initially, scepticism — JSX, the syntax that embedded HTML-like markup inside JavaScript, struck many developers as a step backward. Within a year, that scepticism had largely reversed as developers discovered how much easier it was to reason about UI as a tree of components.

> *"We're not trying to write once and run everywhere. We're learning once and writing everywhere."*
> — Tom Occhino, Facebook — React.js Conf, 2015

The leap to mobile began at an internal Facebook Hack Week in 2013. Christopher Chedeau — known online as Vjeux — was a member of the React core team who had been thinking about the question of whether React's rendering model could be decoupled from the DOM entirely. If you separated the component and state model from the actual renderer, you could in principle render to anything: a native iOS view hierarchy, an Android layout, a PDF, a canvas. The renderer was an implementation detail.

Chedeau built a proof of concept during Hack Week. It was rough — text could scroll, views could be positioned, basic touch events worked. But the mental model was identical to React on the web. A Facebook engineer who knew how to build a React component for the desktop could, in principle, build a native mobile feature with the same knowledge. The implications were significant.

The Hack Week That Became a Platform

The internal prototype from Hack Week 2013 became the seed of what Facebook shipped publicly in 2015. The journey from prototype to production framework took roughly 18 months — during which time Facebook used early React Native internally, first for the Ads Manager app for iOS, which was one of the first fully React Native apps shipped to the App Store. This internal dogfooding period was crucial: it forced the team to confront real-world performance, debugging, and platform integration challenges before the framework became public.

3. React: The Foundation That Made It Possible

To understand React Native's architecture, you first need to understand the three ideas from React that made it technically possible.

The Virtual DOM and Reconciliation

React does not let you manipulate the DOM directly. Instead, you describe what the UI should look like — declaratively, as a tree of component outputs — and React maintains a virtual representation of that tree in memory. When state changes, React re-renders the affected components into a new virtual tree, diffs the new tree against the old one (reconciliation), and applies only the minimum necessary mutations to the real DOM.

The key insight for React Native is that this reconciliation process is entirely independent of the DOM. The virtual tree is just a data structure. The reconciler does not care what you render into. If you give it a renderer that maps React elements to native iOS UIViews instead of DOM nodes, the same reconciler works without modification.

The Component Model

React components are functions (or classes) that take props — input data — and return a description of what should be rendered. They are composable: components are built from other components, forming a tree. They are reusable: the same component can appear in different parts of the tree with different props. This model maps naturally to mobile UI, where a screen is a hierarchy of native views — scroll views containing list items containing text labels and image views.

Unidirectional Data Flow

In React, data flows in one direction: from parent components to child components through props. State is held at the appropriate level in the tree and passed down. This discipline makes applications significantly easier to debug and reason about — when something looks wrong, you follow the data up the tree until you find where it diverges from expectation. This same discipline was essential for React Native, where debugging native UI issues across a JavaScript/native bridge would have been extremely difficult without a predictable data model.

4. React Native Arrives — Learn Once, Write Anywhere (2015)

Facebook open-sourced React Native in March 2015 at the React.js Conf. The response was immediate and enthusiastic. The promise was specific and carefully worded: not 'write once, run everywhere' — that had been the promise of every cross-platform mobile tool before it and had always meant 'write once, get a mediocre experience everywhere'. Instead, the promise was 'learn once, write anywhere': the same framework knowledge, the same component model, the same state management patterns — but the actual code for iOS and Android would be written separately where the platforms genuinely differed.

The crucial distinction was the output. React Native components did not render to a web view. They rendered to real, native platform components: on iOS, a React Native Text component became a UILabel. A React Native ScrollView became a UIScrollView. On Android, the same React Native components became their Android equivalents — TextView, ScrollView, and so on. The user could not tell the difference from a native app, because in terms of rendering, it was a native app.

```
// A simple React Native component — same mental model as React on the web
import React, { useState } from 'react';
import { View, Text, TextInput, TouchableOpacity, StyleSheet } from 'react-native';

export function ChatInput({ onSend }: { onSend: (msg: string) => void }) {
  const [text, setText] = useState('');

  const handleSend = () => {
    if (text.trim()) {
      onSend(text.trim());
      setText('');
    }
  };

  return (
    <View style={styles.container}>
      <TextInput
        style={styles.input}
        value={text}
        onChangeText={setText}
        placeholder="Type a message..."
        returnKeyType="send"
        onSubmitEditing={handleSend}
      />
```

```
      <TouchableOpacity style={styles.button} onPress={handleSend}>
        <Text style={styles.buttonText}>Send</Text>
      </TouchableOpacity>
    </View>
  );
}

// TextInput renders as UITextField on iOS, EditText on Android
// TouchableOpacity handles native touch feedback on both platforms
// — no web views, no HTML, pure native components
```

Android support followed iOS in September 2015, six months after the initial open-source release. Facebook had initially focused on iOS — which aligned with where the company's mobile investment was concentrated — but the architecture had always been designed with platform neutrality in mind. The same JavaScript component tree could, in principle, be mapped to any native view hierarchy, and Android proved that out.

5. The Bridge Architecture — How It Actually Works

The original React Native architecture introduced a concept called the Bridge — the communication channel between the JavaScript thread running your React components and the native thread running the platform's UI engine. Understanding the Bridge is essential to understanding both React Native's capabilities and its historical performance limitations.

Three Threads

The original React Native runtime had three main threads:

- The **JavaScript thread** — where your React components ran, reconciliation happened, and business logic executed. This was a single-threaded JavaScript engine (initially JavaScriptCore, the engine used by Safari)
- The **Native/UI thread** — the platform's main thread, where native view mutations, touch event handling, and all platform UI work happened
- The **Shadow thread** — a background thread where layout calculations were performed using Facebook's Yoga library, a cross-platform implementation of the CSS Flexbox algorithm

The Bridge

The Bridge was an asynchronous message-passing system between the JavaScript thread and the native thread. When your JavaScript code called a native API — say, asking to show an alert, accessing the camera, or triggering a haptic — the call was serialised to JSON, passed across the Bridge, deserialised on the native side, and executed. Results came back the same way, in the opposite direction.

This asynchronous model had an important implication: JavaScript and native could never call each other synchronously. Every cross-thread communication was a serialise-pass-deserialise cycle. For most operations this was fast enough to be invisible. But for operations requiring high-frequency communication — smooth gesture handling, complex animations, rapid touch response — the Bridge became a bottleneck. Dropping below 60 frames per second in a scroll or swipe gesture was a known class of problem for React Native apps throughout its first several years.

> ⚠️ **The Bridge's Real-World Impact**
>
> The Bridge architecture was the single most-cited performance concern in early React Native adoption. The 'jank' visible in fast lists and gesture-heavy interactions was a direct consequence of the serialisation overhead and the inability to synchronously call native code from JavaScript. It drove significant early investment in libraries like Reanimated (which moved animation logic to the native thread) and react-native-gesture-handler (which handled gesture recognition natively). The community found workarounds, but the architecture remained a fundamental constraint — which is precisely what the New Architecture (see Chapter 8) was designed to solve.

6. The Milestone Timeline — Version by Version

Version / Year	Milestone	What It Brought
2013	React created	Jordan Walke and the Facebook team ship React publicly at JSConf. Vjeux's Hack Week experiment on native rendering begins.
Mar 2015	v0.1 — iOS open-source	React Native for iOS released publicly. Components render to real UIKit views. Community adoption begins immediately.
Sep 2015	v0.11 — Android	Android support ships. React Native is now genuinely cross-platform. The ecosystem begins to bifurcate into iOS-only and cross-platform libraries.
2016	v0.2x series	ListView (the first major list component), improved Navigation, Animated API, and the beginning of third-party library ecosystem growth.
2017	v0.4x — Flat List	FlatList replaces ListView — virtualised, performant, much simpler API. One of the framework's most-used components to this day.
2018	Airbnb sunset	Airbnb publishes a widely read blog post sunsetting React Native, citing Bridge limitations and tooling immaturity. Sparks a significant debate about the framework's future.
2018	Facebook re-commitment	Facebook announces a complete architectural rewrite at Chain React 2018 — JSI, Fabric, TurboModules. The New Architecture project begins.
2018	v0.56 — Android improvements	Significant Android performance improvements. Threading model improvements begin.
2019	v0.60 — Auto-linking	Auto-linking eliminates manual native dependency linking — the most painful part of adding native libraries. CocoaPods integration improved.
2020	Lean Core initiative	Non-essential components extracted to community packages (WebView, AsyncStorage, Slider). Core becomes leaner and more maintainable.
2021	Hermes default on Android	Hermes — Facebook's custom-built, ahead-of-time compiled JS engine — becomes the default on Android. Startup time improves significantly.

Version / Year	Milestone	What It Brought
2022	v0.70 — Hermes default iOS	Hermes becomes the default JavaScript engine on iOS. JSI-based architecture begins shipping in production.
2023	v0.72 — New Arch stable	New Architecture (JSI, Fabric, TurboModules) reaches stable status. Expo SDK 50 adopts it. The industry begins migrating.
2024	v0.74 — Bridgeless default	Bridgeless mode (the fully JSI-based architecture with no legacy Bridge) becomes the default for new projects.
2025	New Arch mainstream	Majority of major libraries support New Architecture. React Native and React web begin converging on shared primitives.

7. Expo — Democratising React Native Development

No account of React Native's history is complete without Expo. Founded in 2016 by Charlie Cheever and James Ide, Expo started as a set of tools to make React Native development dramatically less painful — and gradually became the recommended way to start new React Native projects.

The early React Native developer experience had significant friction. Setting up a project required Xcode, Android Studio, Java, the Android SDK, CocoaPods, and a complex series of linking steps for any native library. Developers with web backgrounds — exactly the developers React Native was trying to attract — found this environment foreign and error-prone. The first day of a React Native project was often a day of fighting tooling, not writing components.

Expo introduced a managed workflow: install one CLI, run one command, scan a QR code with the Expo Go app on your phone, and see your React Native app running on a real device — with no Xcode, no Android Studio, no native build environment required at all. For learning, prototyping, and teams that did not need custom native code, this was transformative.

Expo Go — React Native's Best On-Ramp

Expo Go is a pre-built React Native host app, available on the App Store and Google Play. When you start a project with Expo, you run a development server, and any device with Expo Go installed can connect to it and run your code over the local network. No App Store deployment, no provisioning profiles, no certificates. This single capability has probably introduced more developers to React Native than any other feature of the ecosystem.

Expo has evolved significantly since 2016. The Expo Router (released in 2023) brought file-system-based routing — similar to Next.js — to React Native, creating a unified navigation model for web and native from a single file structure. Expo SDK tracks React Native releases closely, wrapping native APIs (camera, location, notifications, contacts, filesystem) in well-maintained JavaScript interfaces. As of 2024, Expo is the officially recommended starting point in the React Native documentation.

> *"If React Native is the engine, Expo is the car. The engine is impressive. The car is what most people actually drive."*
>
> — Developer community observation, 2023

8. The New Architecture — JSI, Fabric, and TurboModules

The Bridge architecture served React Native well for its first several years, but its fundamental limitations — the serialisation overhead, the inability to synchronously call native code from JavaScript, the lack of direct memory sharing between threads — set a ceiling on what was possible. In 2018, Facebook announced a complete architectural rewrite. The New Architecture has three major components.

JSI — The JavaScript Interface

JSI (JavaScript Interface) is the foundational change. It replaces the Bridge's serialised message-passing with a direct binding layer between the JavaScript engine and C++ host objects. JavaScript code can hold a reference to a C++ object and call methods on it synchronously, without serialisation, without the overhead of the Bridge queue.

The implications are significant. Native modules no longer need to be asynchronous by default. Synchronous operations that previously required workarounds are now first-class. And the runtime is no longer coupled to any specific JavaScript engine — JSI is engine-agnostic, which is how Hermes, V8, and JavaScriptCore can all be used as drop-in engines in React Native.

```
// Before JSI — asynchronous bridge call (simplified)
NativeModules.AsyncStorage.getItem('user_token', (error, result) => {
  // Callback runs after serialise → Bridge → deserialise → native → Bridge →
deserialise
  if (!error) setToken(result);
});

// After JSI — synchronous host object access (simplified)
// Native code registered a C++ host object on the JS runtime
// JavaScript can call it synchronously, no Bridge, no serialisation
const token = nativeStorage.getItemSync('user_token');
setToken(token);

// The mental model shifts: native is no longer 'over there somewhere'
// — it is directly accessible from JavaScript as a first-class object
```

Fabric — The New Renderer

Fabric is the rewrite of React Native's rendering system. It uses JSI to give the JavaScript thread direct, synchronous access to the C++ shadow tree that drives layout and the native view hierarchy. This enables several capabilities that were impossible with the Bridge architecture.

- Synchronous layout measurement — JavaScript can query the size and position of a view synchronously, enabling interaction patterns that require layout information
- Concurrent rendering support — Fabric is built to work with React's concurrent features (Suspense, transitions), where rendering can be interrupted, prioritised, and resumed
- Reduced thread-crossing — layout and rendering decisions can be made with fewer round-trips between threads, improving performance for complex UI

TurboModules — Lazy Native Module Loading

TurboModules replace the original Native Modules system. In the original architecture, every native module was loaded and initialised at startup — even if the module was never used. In a large app with hundreds of native modules, this contributed to slow startup times.

TurboModules are loaded lazily, on first use. They use CodeGen — a code generation tool that takes TypeScript interface definitions and generates the native C++ bindings automatically — eliminating an entire class of manual, error-prone binding code that library authors previously had to write and maintain.

The New Architecture Migration Reality

The New Architecture was announced in 2018 and reached stable status in 2023 — a five-year implementation journey that reflects both the complexity of the rewrite and the difficulty of maintaining backwards compatibility for thousands of existing libraries and applications. As of 2024, the majority of widely-used community libraries have migrated. New projects created with Expo SDK 51 or React Native 0.74 use Bridgeless mode (the New Architecture without any Bridge fallback) by default.

9. React Native in Production — Who Uses It and Why

React Native's production adoption tells a nuanced story. The framework is used at significant scale by companies across very different contexts, and their reasons for choosing it vary considerably.

Meta (Facebook, Instagram)

Meta remains both the creator and one of the heaviest users. Significant portions of the Facebook and Instagram apps are built with React Native, giving the company direct incentive to invest in the framework's performance. Meta's internal usage drives much of the core development and is a meaningful reason the New Architecture exists — the limitations of the Bridge architecture were a real cost for Meta's mobile teams before the rewrite.

Microsoft

Microsoft has been one of the most significant enterprise contributors to React Native since 2019, when they began working on React Native for Windows and macOS. Office apps including Teams and Outlook have components built with React Native. Microsoft's investment in the framework reflects a broader strategy of sharing UI code across platforms, including desktop — a dimension Facebook's original vision had not prioritised.

Shopify

Shopify migrated its mobile apps to React Native in 2020 and has become one of the framework's most prominent advocates. The company has open-sourced significant tooling including Flash List (a high-performance replacement for FlatList), Restyle (a type-safe styling system), and Skia bindings for React Native. Shopify's adoption decision — made at a moment when the Airbnb sunset post had cast doubt on the framework's future — was a significant signal that React Native remained viable for serious production use.

Discord

Discord uses React Native for its mobile app and has been public about both the challenges and the benefits. The company has contributed to core React Native performance work, particularly around startup time and list performance. Discord's mobile app is a useful data point because it is a technically demanding product — real-time events, complex state, high-frequency UI updates — that works well on React Native when properly optimised.

React Native in 2025 — The Numbers

React Native is used by approximately 42% of mobile developers who use cross-platform frameworks (Statista, 2024). The React Native npm package receives over 3 million weekly downloads. Expo CLI receives over 2 million weekly downloads. The framework's GitHub repository has over 118,000 stars. There are over 3,000 React Native packages on npm. Flutter, Google's competing framework, has grown significantly since 2021 and now has comparable adoption numbers — the two frameworks compete closely for the cross-platform mobile development market.

10. The Ecosystem — Navigation, State, and the Community

Navigation

Navigation has historically been one of React Native's most complex challenges. The platform-native navigation patterns on iOS (stack-based UINavigationController with push/pop transitions) and Android (back stack with fragment transactions) behave differently and are difficult to abstract consistently. The community has produced several solutions, with React Navigation — originally created by Spencer Ahrens and later maintained by Satyajit Sahoo and a large community — becoming the dominant choice. React Navigation v6 (2021) and Expo Router (which builds on React Navigation) have significantly improved the developer experience.

State Management

React Native benefits from the full JavaScript ecosystem for state management — Redux, Zustand, Jotai, MobX, TanStack Query, and React's own Context API and hooks are all available and widely used. The challenge is understanding which tool is appropriate for which concern: server cache state (TanStack Query), local UI state (useState), and global application state (Zustand or Redux Toolkit) are different problems that benefit from different solutions.

Styling

React Native uses a StyleSheet API that maps JavaScript objects to native layout and style properties. It is not CSS — properties use camelCase, many CSS properties are unsupported, and the layout engine is Flexbox-only. Libraries like NativeWind (which brings Tailwind-style utility classes to React Native), Shopify's Restyle, and Styled Components for React Native offer alternative approaches. The lack of true CSS remains a point of friction for developers coming from web development.

11. Where React Native Is Going

React Native Web — The Shared Component Future

React Native Web (originally created by Nicolas Gallagher at Twitter) allows React Native components to render in a web browser. Combined with Expo's web support, this means a single component can render natively on iOS and Android and as a DOM element on the web. This is the practical realisation of 'learn once, write anywhere' taken to its logical extreme — one component library for all platforms.

Server Components on Native

React Server Components — which allow components to render on the server, sending UI descriptions rather than data to the client — are being explored for React Native. Expo Router has begun experimenting with server rendering and API routes. If Server Components come to mobile, the implications for data-fetching patterns and app architecture would be significant.

Static Hermes and the Performance Ceiling

Facebook is developing Static Hermes, a version of the Hermes engine with a type-directed ahead-of-time compiler that can generate significantly more optimised native code for TypeScript-typed JavaScript. Early benchmarks suggest performance improvements of 10-40x for compute-intensive operations. If this lands in production, it would effectively close the remaining performance gap between React Native and fully native code for most workloads.

React Native Skia and the Graphics Layer

React Native Skia, developed by Software Mansion and Shopify, brings Google's Skia graphics engine (the same engine used by Chrome and Flutter) to React Native as a fully composable React renderer. It enables high-performance custom graphics, complex animations, and image manipulation at 60fps or higher, running entirely on the native thread. This extends React Native into graphics-intensive use cases that were previously the exclusive domain of fully native code.

12. Conclusion

React Native's trajectory is one of the more instructive stories in modern software development. It began as a hack week experiment, shipped with significant architectural limitations, survived a high-profile public sunset from a major adopter, and responded with a complete architectural rewrite that is now reaching maturity. Throughout that arc, it remained the most widely adopted cross-platform mobile framework in the world.

The framework's durability stems from the same source as its initial appeal: the mental model. React's declarative, component-based approach to UI turns out to be genuinely good for building mobile interfaces, not merely for building web interfaces. Developers who learn React once can build for iOS, Android, and web with the same fundamental understanding. That is not a marketing claim — it is an architectural property that has been validated at production scale across thousands of applications.

The New Architecture — JSI, Fabric, and TurboModules — eliminates the Bridge limitations that drove the Airbnb sunset decision and positions React Native to compete directly with fully native development on performance. Combined with Expo's continued investment in developer experience and the growing ecosystem of high-quality community libraries, React Native in 2025 is meaningfully more capable than the framework that prompted those early criticisms.

> *"The best prediction I can make about React Native's future is that it will keep surprising people who write it off."*
> — Evan Bacon, Expo — 2024

The question for engineers and organisations evaluating React Native today is not whether it is ready — it is. The question is whether the trade-offs of shared JavaScript logic and a unified component model across platforms outweigh the additional complexity that any cross-platform abstraction introduces. For most teams shipping to both iOS and Android, the evidence of a decade of production use suggests the answer is yes.

Sources & Further Reading

Primary Sources

- Tom Occhino & Christopher Chedeau, 'Introducing React Native' — React.js Conf 2015 (YouTube)

- Sebastian Markbåge, 'React Native New Architecture' — Chain React 2018
- React Native Blog — reactnative.dev/blog (official release notes and architecture posts)
- Parashuram N, 'React Native New Architecture Overview' — React Native EU 2019

Architecture & Technical Deep Dives

- Nicola Corti & Riccardo Cipolleschi, 'The New Architecture Is Here' — React Native blog, 2022
- 'Under the Hood of React Native' — callstack.io/blog (ongoing series)
- 'Fabric — The New Rendering System' — reactnative.dev/architecture
- React Native Skia docs — shopify.github.io/react-native-skia

The Airbnb Posts (Essential Reading for Context)

- Gabriel Peal, 'React Native at Airbnb' — medium.com/airbnb-engineering , 2018 (five-part series)
- Adam Ernst (Facebook), 'Facebook's React Native Commitment' — response to Airbnb, 2018

Statistics & Adoption

- Stack Overflow Developer Survey 2024 — survey.stackoverflow.co/2024
- Statista: Cross-platform mobile frameworks 2024 — statista.com
- State of React Native 2024 — results.stateofreactnative.com

Further Reading

- Expo documentation — docs.expo.dev
- Jamon Holmgren & Gant Laborde, 'Infinite Red blog' — infinite.red/blog (deep technical content)
- Software Mansion blog — blog.swmansion.com (Reanimated, Gesture Handler, Skia)

About H4 Tech Solutions

At H4 Tech Solutions, our four Hs aren't just values - they're our foundation. Happiness fuels us, Humanity centres us, Honesty anchors us, and Humility teaches us. Together, they shape not just what we build, but who we are and how we walk beside you.