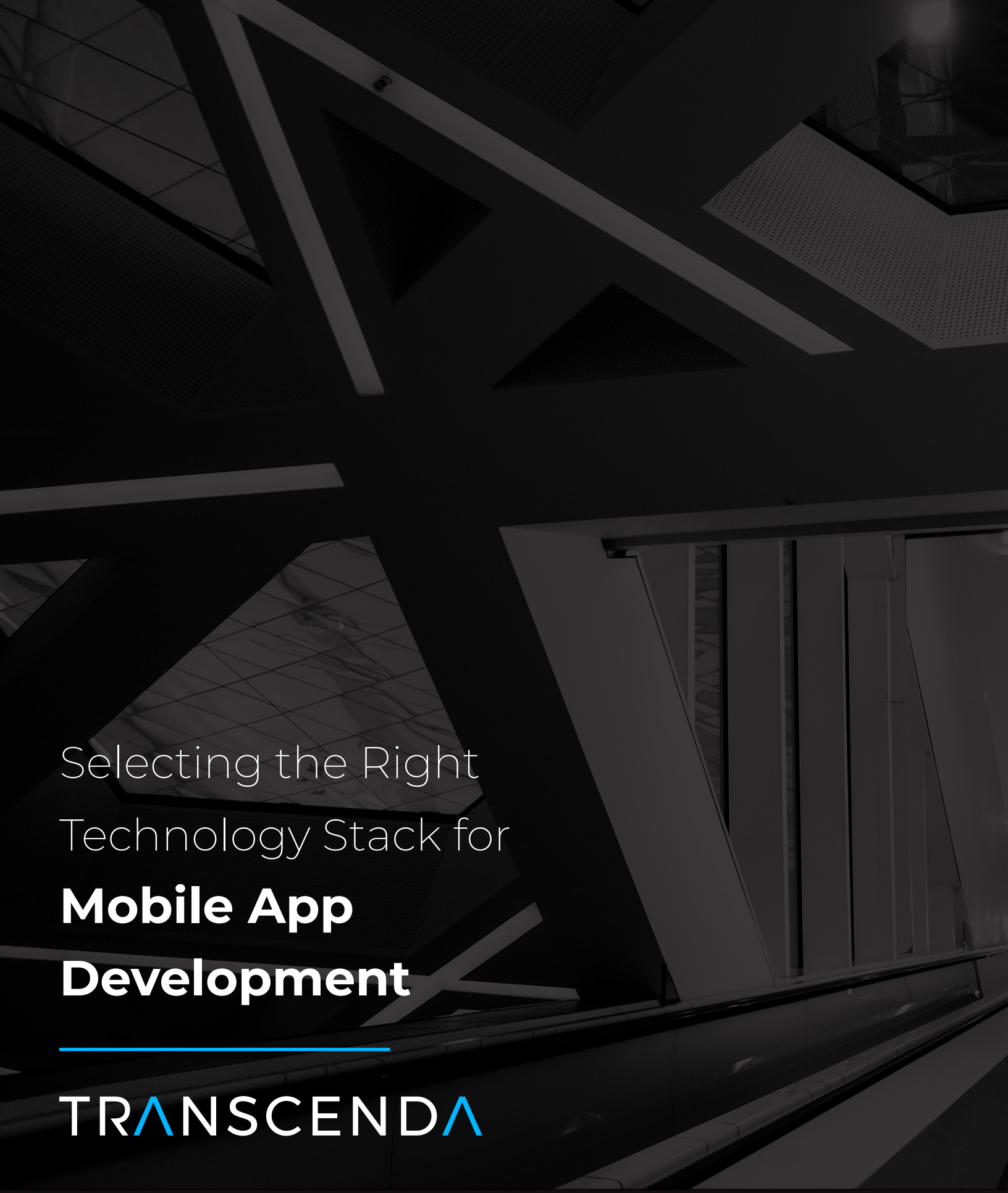


Selecting the Right Technology Stack for **Mobile App Development**

TRANSCENDA

Overview

Mobile applications have steadily become the main channels through which people interact with organizations, access information and perform numerous other tasks, ranging from reading product reviews to viewing real estate listings. In 2017, mobile website traffic surpassed desktop, and it has hovered around 50% ever since, as the high convenience and low barriers to entry of using mobile devices — plus the vast iOS and Android app ecosystems — have made them the primary platforms for millions.

But connecting with and sustaining the interest of mobile users can be challenging, in large part because they have exceptional expectations for how the apps they use look, feel and perform. A Think With Google study found even a 1-second delay in mobile loading times reduced conversions by 20%, while negative mobile experiences decreased future purchase intent by 62%.

Delivering the experience mobile users expect is a balancing act, involving considerations of the relative performance characteristics of different technology stacks and approaches to development, along with their associated learning curves and costs. For example, an app harnessing the power of a device's biometrics sensors and voice assistant may be more useful than an equivalent web app, but costlier to build.

Fortunately, there are multiple viable options available, covering a wide range of budgets, development team capabilities and business requirements. Transcenda can help you determine if native, native cross-platform, hybrid, progressive web app (PWA) or responsive mobile website development is the best fit for your circumstances. Let's explore each of these routes in more detail.

"In 2017, mobile website traffic surpassed desktop, and it has hovered around 50% ever since."

Option No. 1: Native Applications

Native development entails building a bespoke application for each platform, using the official software development kits (SDKs) and tools for the OS in question. For iOS, this process often involves using Swift or the older Objective-C language within an integrated development environment (IDE) like XCode. On Android, Kotlin is Google's preferred language — though Java remains popular — and Android Studio is the official IDE.

Because native mobile applications use technology stacks built specifically for the platforms they run on, they deliver the fastest performance and tightest integration with platform APIs. A native app can fully utilize underlying hardware features such as voice-activated assistants, gyroscopes and biometrics, which may be off-limits to web applications. Moreover, native apps are more energy-efficient and can have smaller filesizes and memory footprints than cross-platform native apps. Plus, they always use native user interface (UI) controls.

These characteristics can make native apps ideal for consistently meeting mobile user expectations. After all, many app uninstalls stem from specific frustrations with apps that are too big, slow, and/or memory-hungry — shortcomings that native apps are better-positioned to avoid than non-native alternatives. However, native app development is relatively time-consuming and expensive. A dedicated repository must be also maintained for each OS, as code cannot be reused across platforms.

Ideal Use Cases for Native Development:

Complex and demanding applications that cannot compromise on performance and are designed to maximize the functionality of a particular platform.

TRANSCENDA

Pros

- Native apps offer the highest level of performance, minimizing the loading times and technical complications that might otherwise drive users to uninstall.
- There are opportunities for optimization in areas such as power consumption, memory utilization and filesize not available to other types of apps.
- Native UI controls are always available, for a seamless user experience closely aligned with that of the platform itself.
- All underlying hardware and APIs are accessible, for tightly integrating functionalities like voice controls and reading the states of other applications.

Cons

- Development time can be very protracted, due to the need to build and optimize separately for each platform and the inability to reuse code.
- Cost of development is also high, not only because of the added time but also as a result of more complex ongoing maintenance.
- Distribution is limited to official app stores, with no downloads or updates via web).

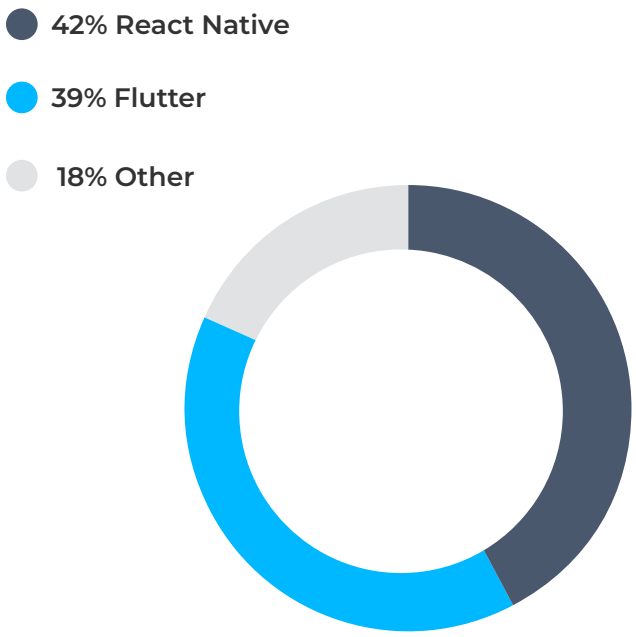
Option No. 2: Cross-Platform Native Applications

Cross-platform native applications are rendered with native code on iOS and Android, but built with different programming languages and tools than those used in traditional mobile native app development. Instead of a Swift or Kotlin stack, developers can use one centered on JavaScript, Dart, or another general-purpose language to build a codebase that can be readily reused across multiple platforms, while still being able to access the native APIs on each one.

This approach has rapidly become one of the most popular methodologies for mobile application development. It saves significant time and money compared to building separately for iOS and Android (development time can be halved compared to native development), offers close-to-native performance and delivers a richer experience than responsive web applications or PWA.

Cross-platform native apps are distributed via app stores and look more or less like traditional native apps to end-users, with some UI variations possible. Like native apps, they face the possible drawback of delays during app review.

Currently, two cross-platform native frameworks dominate the landscape: React Native, which is supported by Facebook, and the more recently released Flutter, a Google-led project.



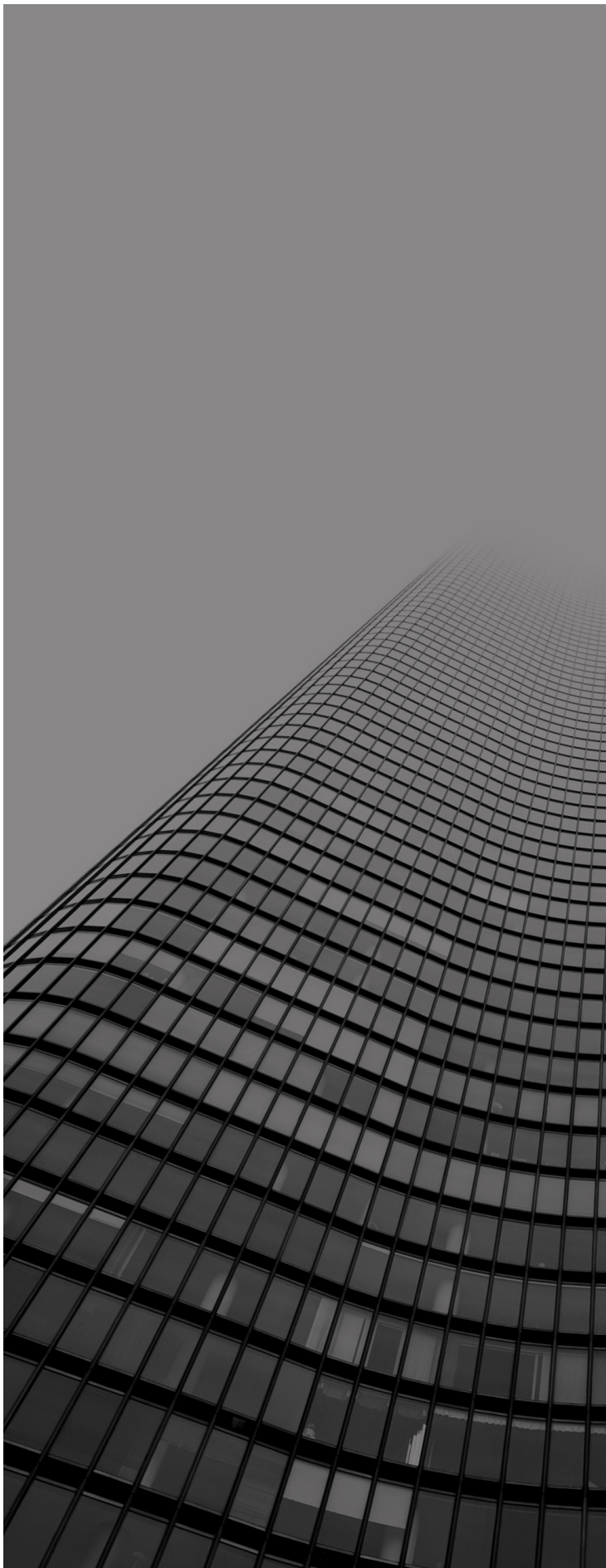
As of 2020, **42%** of mobile developers used React Native and **39%** used Flutter — each more than 20 percentage points ahead of any other solution. Both have risen in popularity as alternative frameworks have declined.

React Native

Reactive Native provides a JavaScript library for building applications that render as native code. As its name suggests, it uses the actual native UI controls on both iOS and Android, rather than imitations of it or standard web views.

A React Native application features an on-device main (native) UI thread, built with the iOS or Android SDK, and a JavaScript thread running in a separate virtual machine. The two are connected via an asynchronous, serialized and batched bridge, which sends the views and business logic from the JavaScript side to the native side for execution at runtime.

Essentially, React Native apps are built with components that wrap native code and then interact with native APIs using JavaScript and the declarative React UI paradigm. These components, such as “View” and “Text,” are platform-agnostic and get “mapped” later on to each mobile platform’s native UI. Accordingly, developers can work quickly and productively, while targeting both iOS and Android thanks to the shared JavaScript code.

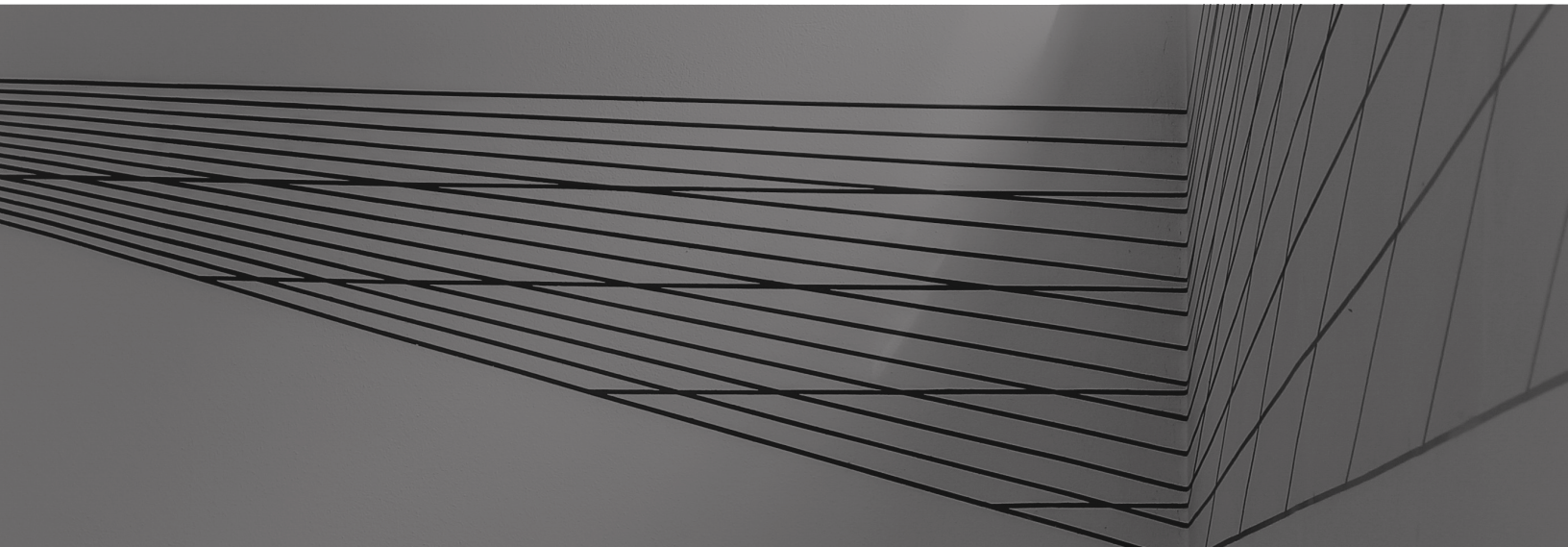


Pros

- JavaScript is one of the most used programming languages, with an extensive package ecosystem. Plus, the large React Native community is always adding new functionality.
- React Native apps have reasonable performance along with true native UI controls from official SDKs, for a native-like experience for end users.
- CodePush support is available for the JavaScript side, so that “live” app updates can be sent directly to a user’s device.
- Overall, React Native enables cost-effective and streamlined cross-platform development, with a convenient mapping system for components and shared code for mobile and web.

Cons

- You may still need to perform native development to build custom React Native components from scratch, negating the advantages of a cross-platform approach.
- The dual runtime environments (native and JavaScript) make debugging more difficult and can also lessen performance compared to native or Flutter apps.
- There is a significant learning curve, not only for learning React Native but for navigating some of the particular technical issues React Native apps often encounter.



Flutter

Flutter is a complete open source UI toolkit for building cross-platform, natively compiled applications with one codebase. Flutter apps are built with Dart, Google's own client-optimized language. The Flutter framework includes layered libraries for animations, painting, gestures, rendering and widgets, along with Material (Android) and Cupertino (iOS) controls for implementing the appropriate design language on each platform. Note that these design libraries are not actually native Android or iOS UI, though, despite the similarities.

Unlike React Native, Flutter does not need a bridge, nor does it rely on just-in-time (JIT) compilation on mobile devices. It instead uses ahead-of-time (AOT) compilation of Dart code into native ARM code, boosting performance compared to JIT execution. The Flutter framework sits atop the Flutter engine, which is written primarily in C++ and contains the Dart runtime and Skia graphics library, and a platform-specific embedder.

During development, Flutter allows for relatively fast, productive work, due to its Stateful Hot Reload feature for seeing changes without discarding the current app state. The same codebase can be used for both iOS and Android. Plus, Flutter supports web and desktop development, too.

Pros

- Flutter apps perform as close as possible to native apps, through AOT compilation of their Dart code.
- Flutter's layered architecture allows for extensive customization of virtually every pixel in an application.
- Development and debugging in general are straightforward compared to other cross-platform frameworks.
- Code can be easily shared across multiple mobile and desktop web platforms.

Cons

- Dart is not nearly as familiar or commonly used as JavaScript, CodePush isn't supported and the overall community and ecosystem are not yet as large as React Native's.
- The Material and Cupertino libraries do not implement true native UI controls.

Other Frameworks

Aside from React Native and Flutter, there are comparable frameworks including Xamarin (a Microsoft subsidiary), NativeScript and Appcelerator Titanium. These solutions may offer unique benefits such as the ability to use C# and .NET, or TypeScript, which transpiles to JavaScript.

But these frameworks have steadily lost ground, in terms of developer interest, to the big two over time. The specific reasons for their decline may include their overly complex tools, limited community support and worse performance than competitors.

Ideal Use Cases For Cross-Platform Native Applications:

Cross-platform applications that need close-to-native performance, but without the cost, timeframes and complexity of native development.

Option No. 3: Hybrid Applications

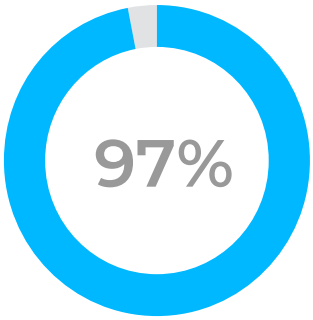
Hybrid applications combine native and web technologies, albeit in a very different way than JavaScript-based solutions such as React Native, and with much less performwance optimization. Typically, this approach involves encapsulating a mobile website or application core written in HTML, CSS and JavaScript inside of a native shell. This encapsulation serves three important functions:

1. The application can be accessed outside of a standalone web browser, via an application distributed through an official app store. This app includes its own embedded browser to render its web views.

2. Using plugins made available by popular wrapper frameworks, the resulting app can also access underlying hardware features, such as biometrics sensors, GPS location data and the device filesystem.

3. The same site or core can be wrapped and distributed through the official iOS and Android storefronts, enabling convenient and low-cost cross-platform development without having to write new native code.

As the mobile app economy grew throughout the 2010s, many developers abandoned native-only development to focus on hybrid-only or a blend of native and hybrid.



Hybrid development using frameworks like Ionic, Apache Cordova and Capacitor peaked around 2017, when **97%** of respondents to an Ionic survey reported plans to pursue it over the next few years.

Since then, though, the growing uptake of React Native and Flutter has cut into the popularity of these solutions. Both React Native and Flutter provide the same cross-platform benefit of hybrid applications, while also delivering superior performance.

That said, hybrid development using HTML, CSS and JavaScript remains a worthwhile option for some use cases. Wrapping a site is a straightforward, low-cost endeavor that maximizes the value of something already built. The learning curve is slight compared to both native and cross-platform native development, plus there is an extensive system of development and debugging tools, as well as applicable open source technologies.

Pros

- JavaScript, HTML and CSS were the most used languages in the 2020 Developer Survey from Stack Overflow, indicating the high familiarity and low learning curve of this stack.
- Hybrid development with web technologies is the lowest-cost option for building a cross-platform application that can be distributed through official mobile app stores.
- The development timeline is also short, since a hybrid app can be assembled from an existing mobile website.

Cons

- Hybrid app performance is noticeably worse than that of either native or cross-platform native apps.
- Because they don't use native UI controls and instead wrap a site, hybrid apps can look and feel more like mobile web apps than "real" mobile apps.

Option No. 4: Responsive Mobile Web Applications

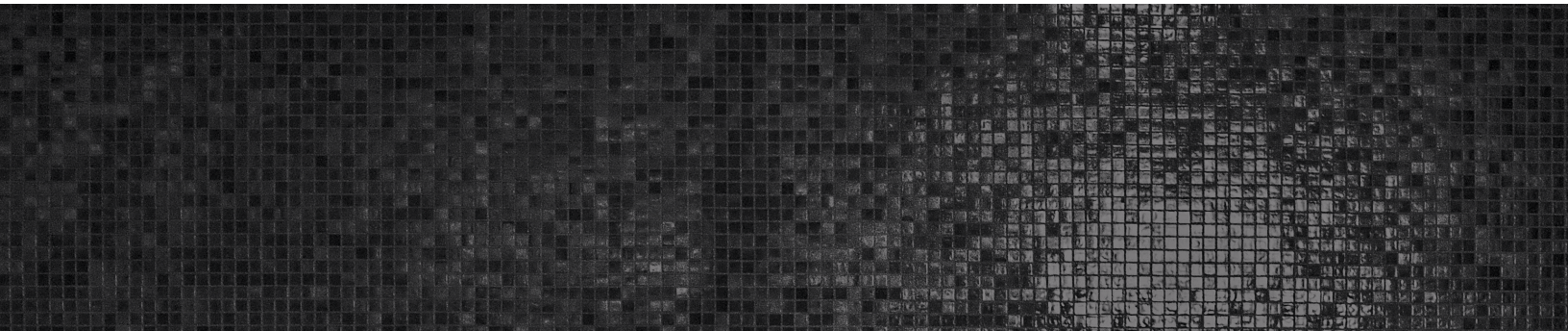
A responsive mobile web application is a website that optimizes its layout and design for each device accessing it, in keeping with principles of responsive web design.

Although the same HTML gets sent to each device, CSS makes the responsive page render and behave differently depending on screen and browser window size. Responsive sites allow for a single URL to be used by all devices, while also being easier for search engines to crawl and index, helping with discoverability.

Responsive mobile web apps are accessed like any other website, via a web browser that renders HTML5. As such, they are relatively simple and economical to develop, distribute and maintain.

Over time, HTML5 has added APIs enabling web apps to access deeper features of the underlying hardware and OS, including geolocation data, Bluetooth, magnetometers, battery status, vibration, WebRTC and HDCP. These APIs are usually easy to call using JavaScript, meaning it's possible to build a richly featured experience with existing web technologies instead of native code.

However, responsive web apps are at the mercy of the browser they're running on, and not all browsers and mobile operating systems offer the same support for advanced HTML5 APIs. Chromium-based browsers and Mozilla Firefox offer more support for newer web APIs than Safari or any iOS versions of major browsers. This disparity means that the responsive web app experience, despite being universal in theory, can differ substantially by platform.



Pros

- The same URL is accessible from any standards-compliant browser on any OS.
- Web apps are simple to access and use, as there are no downloads, installations or upgrades to worry about, and the latest version is always available.
- Development is fast, straightforward and cost-effective, as is distribution since official app stores are not involved.

Cons

- Web API support varies significantly by browser and OS.
- Web apps are not accessible offline.
- Web UI is relatively slow and not specifically optimized for each platform.

Ideal Use Cases for Responsive Mobile Web Applications:

General web presence, prototypes and apps that need to be brought to market as soon as possible..

Option No. 5: Progressive Web Applications

PWAs go a step further than responsive web applications, by integrating capabilities like service workers and web app manifests to deliver a more native-like user experience that works offline. These apps were pioneered by Google on Chrome and Android, and they have since gotten official supports on iOS as well as desktop.

Some of the key differences between PWAs and responsive web apps include the ability of PWAs to do some or all of the following, depending on their design and platform:

- Run offline.
- Be installed on the home screen via a shortcut.
- Launch quickly, thanks to local caching.
- Run in their own windows without normal browser UI.
- Utilize keyboard shortcuts.
- Accept content from other apps or be set as a default app.
- Send push notifications.

Ideal Use Cases For Progressive Web Applications:

General web presence, prototypes, upgrades from responsive mobile web apps and somewhat complicated apps that cannot go through official app stores.

Overall, PWAs sit somewhere between responsive web apps and hybrid apps. They are a logical upgrade route for organizations that already have a responsive website and want to deliver a more app-like experience to users, especially on Android or desktop.

Pros

- PWAs are fast and inexpensive to build and maintain, just like responsive mobile web apps.
- At the same time, they offer a more immersive experience that better harnesses the power of the underlying hardware and OS.
- Despite looking and feeling like apps, they don't have to go through official app stores for approval or updates. Some mobile gaming platforms have already pivoted to PWAs for this reason.

Cons

- Support is not uniform across platforms, with Android offering a better PWA experience than iOS.
- PWAs are still slower than native and cross-platform native apps.

Conclusion

There is no universally “correct” option among the five above. The right choice(s) will always depend on your particular business objectives, the intended use cases and performance characteristics of your application(s), your desired go-to-market strategy and timeline, the skills of your teams and their levels of experience in building and maintaining mobile applications. For example, it often makes sense to maintain both a responsive web app or PWA and a cross-platform native app to serve the widest possible range of users.

All of these major technology stacks will remain viable for the foreseeable future, although there may be some decline in the popularity of native apps vis-a-vis cross-platform native apps, and of responsive web applications vis-a-vis PWAs.

TRANSCENDA

The following table offers a high-level overview of how the main options stack up across key criteria including performance and overall cost:

	Native Apps	Cross-Platform Native Apps	Hybrid Apps	Responsive Web Apps	PWAs
Relative Performance	Highest	High	Medium to Low	Lowest	Medium to Low
Main Programming Languages	Swift (iOS), Kotlin (Android)	JavaScript (React Native), Dart (Flutter)	HTML, CSS and JavaScript	HTML, CSS and JavaScript	HTML, CSS and JavaScript
Cross-Platform?	No	Yes	Yes	Yes, but more limited on iOS	Yes, but more limited on iOS
Hardware & OS Integration	Highest	High	High	Lowest	Medium to Low
Distribution Channel	Official app stores	Official app stores	Official app stores	Web	Web
Development & Maintenance Costs	High	Medium	Low	Low	Low
Development Speed	Slow	Medium to Slow	Fast	Fastest	Fast
Run Offline?	Yes	Yes	Yes	No	Yes
Live updates & CodePush?	No	Depends	Yes	Yes	Yes

The Transcenda team will work closely with your internal teams to determine the best way forward, based on your current goals and infrastructure for mobile development. Connect with us directly or view our case studies to learn more.


TRANSCENDA
www.transcenda.com