Towards Evidence-Based Mobile Technology Migration: Evaluating Native and Cross-Platform Solutions in a Case Study

Mateus Henrique dos Anjos Oliveira
Department of Informatics, Federal University of Viçosa (UFV)
Viçosa, Brazil
mateus.h.oliveira@ufv.br

Lucas Vegi
Department of Informatics, Federal University of Viçosa (UFV)
Viçosa, Brazil
lucas.vegi@ufv.br

Abstract

Cross-platform frameworks such as React Native reduce development effort by enabling a single codebase for Android and iOS, but the runtime costs of this abstraction are rarely quantified in industrial applications. This paper presents an evidence-based technology migration study of a production home healthcare application originally implemented in React Native and fully reimplemented natively for Android (Kotlin) and iOS (Swift), while preserving all original functionality. Because all implementations follow the same functional-declarative programming model, the comparison primarily isolates differences in their execution models. An automated benchmarking campaign was conducted on two physical devices per platform, evaluating CPU and memory consumption, startup time, UI smoothness, build time, and application size. The native implementations consistently consumed fewer computational resources (up to half the memory usage and one-fifth of the CPU time on iOS), required up to 2.7× shorter build times, and produced application packages up to seven times smaller. In contrast, React Native consistently achieved faster application startup while maintaining UI smoothness comparable to the native implementations. These results provide quantitative support for evidence-based mobile technology migration decisions in the context of the analyzed industrial application.

Keywords

Mobile Development, Software Migration, Kotlin, Swift, React Native, Benchmarking

1 Introduction

The company MedYes develops and maintains CareYes, a mobile platform for home healthcare management that supports healthcare professionals during patient visits. The platform enables caregivers to access patient information, register clinical procedures, complete checklists, and synchronize collected data with the company's backend services. Its production mobile client is implemented using React Native [17].

Although cross-platform frameworks such as React Native significantly reduce development effort by enabling a single codebase for Android and iOS [1], they introduce additional abstraction layers that may affect application performance, resource consumption, startup time, and binary size [19]. As industrial mobile systems evolve and their user base grows, these trade-offs become increasingly relevant, particularly for applications that require high responsiveness and long-term maintainability. Consequently, organizations often face the difficult decision of whether the benefits of maintaining a single codebase outweigh the potential gains of migrating to fully native implementations [18, 20].

This need for quantitative evidence has motivated several empirical studies comparing native and cross-platform mobile technologies, reporting different trade-offs regarding performance, energy consumption, and developer productivity [2, 8, 13]. However, most existing evaluations rely on synthetic benchmarks or relatively small applications, providing limited evidence from production systems. To help bridge this gap, this work evaluates a real industrial mobile application.

To provide such evidence, we reimplemented the CareYes mobile client natively for both Android and iOS using Kotlin and Swift, respectively, while preserving its original functionality. All evaluated mobile implementations communicate with an equivalent Elixir backend developed in our previous work. We then conducted an automated benchmarking study comparing the original React Native implementation with the native versions under equivalent execution scenarios. The evaluation considers CPU and memory consumption, application startup time, UI smoothness, build time, and application size, using two different devices for each platform to reduce device-specific bias.

Despite relying on different implementation technologies, the evaluated applications follow the same functional-declarative programming model. React Native inherits the functional principles of React (JavaScript) [14], whereas Kotlin and Swift both encourage functional programming practices through immutable data structures, higher-order functions [6], and declarative UI frameworks such as Jetpack Compose [10] and SwiftUI [3]. This shared programming model allows the comparison to isolate the effects of the execution model rather than differences in programming style, contrasting ahead-of-time native compilation with JavaScript execution on Hermes [15], Meta's lightweight JavaScript engine optimized for React Native.

This paper makes three contributions: (i) an empirical comparison of equivalent React Native and native implementations in an industrial mobile application; (ii) quantitative evidence to support mobile technology migration decisions; and (iii) lessons learned from an industrial technology migration study.

The remainder of this paper is organized as follows. Section 2 presents the methodology adopted in this study. Section 3 reports and discusses the benchmarking results. Finally, Section 4 concludes the paper and outlines directions for future work.

2 Methodology

The main objective of this study is to evaluate whether migrating a production mobile application from React Native to fully native

implementations improves runtime performance and resource efficiency, thereby providing quantitative evidence to support mobile technology migration decisions in industrial settings. Accordingly, we formulated the following research questions:

- **RQ1:** How do equivalent React Native and native mobile implementations compare regarding CPU and memory consumption?
- **RQ2:** How do both approaches compare regarding application startup time, UI smoothness, compilation time, and application size?

To answer these research questions, the study was organized into three stages. First, we **analyzed** the production mobile application to understand its architecture, user workflows, and implementation technologies. Next, we **fully reimplemented** the application for Android and iOS using their respective native technologies while preserving its original behavior. Finally, we **evaluated** the original and reimplemented versions through an automated benchmarking campaign covering representative user scenarios. Figure 1 summarizes the methodology adopted in this study.

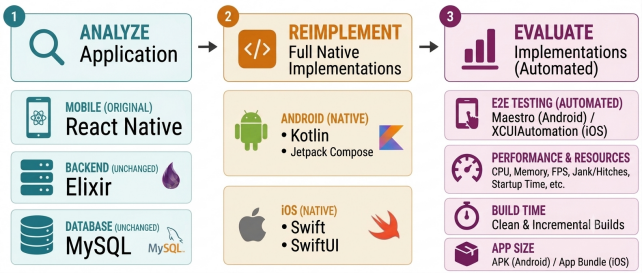

Figure 1: Overview of our methodology.

1) Application analysis: Before reimplementation, we manually analyzed the production version of CareYes to understand its architecture, business rules, navigation structure, UI components, and user workflows, ensuring that every feature would be preserved in the native implementations. The analyzed system is a home healthcare mobile application implemented with React Native, comprising approximately 250 source files, 72 screens, 45 thousand lines of code, and 44 functional modules. The application follows a Model-View-Presenter (MVP) architecture and uses an equivalent Elixir implementation of its production backend, developed in our previous work. The backend remained unchanged, isolating the mobile implementation technology as the only experimental variable.

2) Native reimplementation: The entire mobile application was independently reimplemented for Android using Kotlin with Jetpack Compose and for iOS using Swift with SwiftUI. Both native versions fully preserve the original React Native application's functionality, ensuring functional equivalence across all evaluated features. Whenever possible, the implementations reproduced the original application's behavior and user experience, adopting platform-specific development practices only when they did not affect functionality. All three mobile applications access the same Elixir backend and MySQL external database, ensuring identical business logic throughout the experiments.

3) Benchmarking: We benchmarked the React Native and native implementations using representative end-user scenarios. The evaluation covered the application's main functionalities, including *patient listing*, *monthly calendar*, *messaging*, *patient details*, and other representative workflows, for a total of 9 evaluation scenarios. Within these functionalities, we evaluated different execution behaviors, including *cold start*, intensive scrolling, screen transitions, and user interactions. The complete list of evaluated scenarios is available in the study's replication package.

Performance was assessed using complementary runtime metrics. Resource utilization included CPU usage and memory consumption. User experience metrics comprised application startup time, frame rate, frame rendering time, dropped frames, and *jank* (Android) or rendering hitches (iOS). Engineering-related metrics included compilation time (clean and incremental builds) and application size.

Following benchmarking best practices [9, 11, 12], each scenario was executed independently three times. Repeated executions reduce the influence of transient system effects and improve the statistical reliability of the reported results. For each metric, we report the median as the primary measure of central tendency because it is robust to outliers.

To ensure a fair comparison, all implementations were compiled in release mode using production optimizations (*e.g.,* Android R8 and Hermes bytecode for React Native). Experiments were conducted on physical devices using two devices per platform to reduce device-specific bias: a Samsung Galaxy Tab S7 FE (Android 14), a Google Pixel XL (Android 10), an iPhone 16 (iOS 26.5), and an iPhone 13 (iOS 17.4.1).

Data collection was fully automated through end-to-end test suites to ensure that every scenario was executed identically across repetitions, eliminating variability introduced by manual interaction. On Android, we used Maestro [16] together with `adb dumpsys` and `am start -W` to automate execution and collect runtime metrics. On iOS, automation was implemented using XCTest [4] and XCUIAutomation [5] together with Apple's `measure` blocks. All automation scripts, benchmarking configurations, and the complete list of evaluated scenarios are publicly available in the replication package, enabling full reproducibility of our methods.

3 Results

This section reports the benchmarking results according to the research questions defined in Section 2. We first present the results on resource utilization (RQ1), followed by runtime behavior and engineering-related metrics (RQ2). Unless otherwise stated, all reported values correspond to the median across repeated executions.

Table 1: Summary of benchmarking results.

	Android		iOS	
Dimension	**Native**	**RN**	**Native**	**RN**
Memory	1×	1.6–2.0×	1×	1.4–1.9×
CPU	1×	1.3–1.7×	1×	1.9–2.2×
Startup Time	1.1–1.2×	1×	1.2×	1×
UI Smoothness	Tie		Tie	

3.1 Resource Utilization (RQ1)

Table 1 shows that the native implementations consistently consumed less CPU and memory than React Native (RN) across all evaluated devices. The differences were more pronounced on iOS, where React Native required up to twice as much memory and more than five times the CPU time of the Swift implementation. On Android, the Kotlin implementation reduced memory consumption by 39–50% and CPU utilization by 22–43% relative to React Native.

Table 2 illustrates these differences under an intensive scrolling workload. The Android results report the median average PSS memory measured during execution, whereas the iOS results correspond to peak memory usage collected through XCTest. Overall, these findings are consistent with the additional runtime overhead introduced by JavaScript execution on Hermes and the React Native execution architecture.

Table 2: Intensive scrolling scenario. Values correspond to the median across three independent executions.

Device	Metric	Native	RN	RN / Native
Tab S7 FE	Avg. PSS Memory	97.47 MB	176.69 MB	1.81×
(Android)	Avg. CPU Usage	30.03%	40.37%	1.34×
iPhone 16	Peak Memory	32.25 MB	64.23 MB	1.99×
(iOS)	CPU Time	7.51 s	39.56 s	5.27×

Although direct battery measurements were not feasible in our experimental environment, CPU utilization is widely adopted as an indirect proxy for energy consumption [7]. Therefore, the substantially lower CPU utilization observed for the native implementations suggests improved battery efficiency during prolonged use.

3.2 Runtime and Engineering Metrics (RQ2)

Regarding user experience, both approaches achieved comparable UI smoothness. Frame rates remained close to 60 FPS on iOS without severe rendering hitches, while Android *jank* metrics alternated between implementations depending on the evaluated device, indicating no clear advantage for either approach. In contrast, React Native consistently achieved faster cold startup times, reducing application launch time by approximately 8—35% across the evaluated devices.

Engineering-related metrics, however, favored the native implementations (Table 3). The largest difference was observed on iOS, where clean builds required 59 s for Swift and 160 s for React Native. Native applications were also substantially smaller. On Android, the device-specific APK occupied only 4.30 MB compared with 30.30 MB for React Native, while the universal APK reached 70.70 MB largely due to the inclusion of the Hermes runtime and the React Native runtime libraries. Similar results were observed on iOS, where the native application occupied 20.60 MB compared with 36.40 MB for React Native.

Table 3: Engineering-related metrics.

Platform	Metric	Native	RN	RN / Native
Android	Clean Build	52 s	81 s	1.56×
	Universal APK	4.30 MB	70.70 MB	16.44×
	Device APK	4.30 MB	30.30 MB	7.05×
iOS	Clean Build	59 s	160 s	2.71×
	App Bundle	20.60 MB	36.40 MB	1.77×

3.3 Discussion

Overall, the results reveal a clear trade-off between the evaluated technologies. Native implementations consistently improved resource efficiency, build time, and application size while maintaining UI smoothness comparable to React Native. Conversely, React Native consistently provided faster application startup while preserving the benefits of a shared cross-platform codebase. These findings provide quantitative evidence to support mobile technology migration decisions. While the results are specific to the analyzed industrial application, they suggest that native development may be advantageous when resource efficiency and engineering costs outweigh the benefits of maintaining a single cross-platform codebase.

4 Conclusion and Future Work

This paper presented an evidence-based technology migration assessment conducted on an industrial home healthcare application, fully reimplemented natively for Android and iOS from its original React Native version. In the analyzed case study, the native implementations consistently achieved lower CPU and memory consumption while maintaining UI smoothness comparable to React Native. Conversely, React Native consistently provided faster application startup, whereas the native implementations required shorter build times and produced substantially smaller application packages. Because all implementations follow a common functional-declarative programming model, these results suggest that the observed differences are primarily associated with the underlying execution model rather than programming style. Overall, the findings provide quantitative evidence to support mobile technology migration decisions in industrial settings, while remaining specific to the characteristics of the analyzed application.

As future work, we plan to quantify the engineering cost of the migration itself by comparing the development and maintenance effort required for two independent native codebases against a single cross-platform implementation. We also intend to replicate the study with additional industrial mobile applications to investigate the extent to which the observed trade-offs generalize to other application domains.

Artifact Availability

The complete benchmarking results and automation scripts are available at https://doi.org/10.5281/zenodo.21280765. The source code of the React Native, Kotlin, and Swift implementations is proprietary and cannot be made publicly available.

Acknowledgements

This research was supported by CNPq (PIBIC grant, 2025–2026) and MedYes: https://medyes.com.br/.

References

[1] Arshad Ahmad, Kan Li, Chong Feng, Syed Mohammad Asim, Abdallah Yousif, and Shi Ge. 2018. An Empirical Study of Investigating Mobile Applications Development Challenges. *IEEE Access* 6 (2018), 17711–17728.

[2] Ville Ahti, Sami Hyrynsalmi, and Olli Nevalainen. 2016. An Evaluation Framework for Cross-Platform Mobile App Development Tools: A case analysis of Adobe PhoneGap framework. In *17th International Conference on Computer Systems and Technologies (CompSysTech)*. 41–48.

[3] Apple Inc. 2026. *SwiftUI Documentation*. https://developer.apple.com/swiftui/

[4] Apple Inc. 2026. *XCTest Documentation*. https://developer.apple.com/documentation/xctest

[5] Apple Inc. 2026. *XCUIAutomation Documentation*. https://developer.apple.com/documentation/XCUIAutomation

[6] Uberto Barbini. 2023. *From Objects to Functions: Build Your Software Faster and Safer with Functional Programming and Kotlin* (1 ed.). The Pragmatic Bookshelf.

[7] Yonghun Choi, Seonghoon Park, Seunghyeok Jeon, Rhan Ha, and Hojung Cha. 2022. Optimizing Energy Consumption of Mobile Games. *IEEE Transactions on Mobile Computing* 21, 10 (2022), 3744–3756.

[8] Leonardo Corbalan, Juan Fernandez, Alfonso Cuitiño, Lisandro Delia, Germán Cáseres, Pablo Thomas, and Patricia Pesado. 2018. Development frameworks for mobile devices: a comparative study about energy consumption. In *5th International Conference on Mobile Software Engineering and Systems (MOBILESoft)*. 191–201.

[9] Andy Georges, Dries Buytaert, and Lieven Eeckhout. 2007. Statistically rigorous java performance evaluation. In *22nd Annual ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages and Applications (OOPSLA)*. 57–76.

[10] Google. 2026. *Jetpack Compose Documentation*. https://developer.android.com/compose

[11] Brendan Gregg. 2020. *Systems Performance: Enterprise and the Cloud* (2 ed.). Addison-Wesley Professional.

[12] Raj Jain. 1991. *The Art of Computer Systems Performance Analysis*. John Wiley & Sons.

[13] Xiaoping Jia, Aline Ebone, and Yongshan Tan. 2018. A performance evaluation of cross-platform mobile application development approaches. In *5th International Conference on Mobile Software Engineering and Systems (MOBILESoft)*. 92–93.

[14] Meta. 2026. *React Documentation*. https://react.dev/

[15] Meta. 2026. *Using Hermes*. https://reactnative.dev/docs/hermes

[16] Mobile.dev. 2026. *Maestro: End-to-End UI Testing for Mobile and Web*. https://maestro.dev/

[17] Tom Occhino. 2015. *React Native: Bringing Modern Web Techniques to Mobile*. https://engineering.fb.com/2015/03/26/android/react-native-bringing-modern-web-techniques-to-mobile/

[18] Manuel Palmieri, Inderjeet Singh, and Antonio Cicchetti. 2012. Comparison of cross-platform mobile development tools. In *16th International Conference on Intelligence in Next Generation Networks (ICIN)*. 179–186.

[19] Christoph Rieger and Tim A. Majchrzak. 2019. Towards the definitive evaluation framework for cross-platform app development approaches. *Journal of Systems and Software* 153 (2019), 175–199.

[20] Andreas Sommer and Stephan Krusche. 2013. Evaluation of cross-platform frameworks for mobile applications. *Lecture Notes in Informatics (LNI), Proceedings - Series of the Gesellschaft fur Informatik (GI)*.