



A Comparison of Native and Cross-Platform Frameworks for Mobile Applications

Piotr Nawrocki, Krzysztof Wrona, Mateusz Marczak, and Bartłomiej Sniezynski, AGH University of Science and Technology

The methods used for developing applications for mobile devices have become an important field of research. In this article, we analyze the two most popular approaches, native and cross platform, and compare identical applications developed using various tools.

The popularity of mobile devices such as cell phones, smartphones, and tablets has grown over the years. As their importance has grown, so too has the number of applications available for such devices. In the first quarter of 2019, Google Play offered more than 2 million applications, whereas the Apple App Store had approximately 1.8 million.¹ This shows that the mobile application market is very large and growing yearly. Therefore, the methods for developing such applications are becoming an important field of analysis. Presently, two approaches for creating

mobile applications exist: native and cross platform. At the same time, cross-platform environments are becoming increasingly strong competitors for native (Android and iOS) solutions. This is caused, among other reasons, by the ability to create software rapidly and develop applications for multiple mobile operating systems simultaneously. Most importantly, mobile app implementation costs can be reduced. Aside from the two main methods, there is also a third approach, which utilizes web programming, including HTML5 and supplementary mechanisms such as Web Socket and Application Cache. However, despite the development of solutions such as progressive web apps (PWAs), it is the least common method for developing mobile applications.

In this context, it is worth asking whether mobile applications created using cross-platform environments are comparable in their performance or in their ease of development with native solutions. We believe that the answer to this vital question may facilitate the prudent planning of mobile application development and allow the creators of such applications to choose the appropriate technological solutions.

An analysis of the current state of the art in this regard has revealed that there are no comprehensive studies comparing native and cross-platform solutions in the field of mobile applications. One of the few studies that attempts to approach the issue comprehensively is by Ohrt and Turau² (other studies that present research in this regard will be discussed further in this article). However, the authors of that study conducted their research in 2012, which, given the speed of the development of mobile devices and of the software installed on them, makes their results outdated. Most importantly, most of the cross-platform environments analyzed in their study are no longer commonly used (for example, RhoStudio and MoSync). Some of the solutions, such as Adobe PhoneGap, are still in use but are not as popular among developers anymore.

Among the mobile device operating systems analyzed in that study, including Symbian, Bada, and MeeGo, only two have a significant presence today:³ iOS and Android. The experiments conducted by the authors of that study, which compared various solutions, did not take into consideration the native iOS system; only the Android system and cross-platform solutions were compared. We believe that other important flaws of that study were the failure to compare the CPU usage levels

of various mobile devices and the fact that the applications tested were not classified in terms of their complexity. Experiment results for simpler mobile applications may differ from those for ones that have a complex GUI and perform tasks, which place a significant burden on mobile devices.

Therefore, to analyze native and cross-platform solutions, we decided to compare two application types—simple and complex applications—developed

there are other ones currently in use as well, such as Adobe PhoneGap, Ionic, and Mobile Angular UI. However, these are not as popular nor commonly used as the solutions we selected.

MOBILE APPLICATIONS DEVELOPMENT

The development of native applications (for Android and iOS) is widely discussed in scientific articles. Native solutions are also often described in

IN ADDITION TO COMPARING THE NATIVE AND CROSS-PLATFORM APPROACHES, WE ALSO COMPARED THE TWO NATIVE SOLUTIONS (ANDROID AND IOS).

using both approaches. During the experiments conducted, we compared CPU and random-access memory (RAM) usage as well as the application size and launching time for each approach. We also attempted to compare the ease of implementation of the application for each cross-platform environment.

In addition to comparing the native and cross-platform approaches, we also compared the two native solutions (Android and iOS). The selection of solutions to compare was an important issue. In the case of native solutions, in our study, we decided to use the iOS and Android systems, which account for more than 99.9% of the mobile device operating systems market.³ For cross-platform solutions, we selected the three most popular ones:⁴ Xamarin, Flutter, and React Native, developed by Microsoft, Google, and Facebook, respectively. Of course, aside from the environments we selected,

articles on cross-platform solutions for comparison purposes; however, there are many fewer descriptions of cross-platform solutions, most of which focus primarily on the platforms supported, the speed of development using individual cross-platform frameworks, the supported native application programming interfaces (APIs), and how native the resulting user interface (UI) looks. Therefore, at this point, we decided to focus on a description of cross-platform solutions and their comparison with native solutions, which we do not describe in greater detail.

Cross-platform applications

The first and oldest (2011) of the cross-platform environments we selected for our study was Xamarin. In 2016, Microsoft acquired Xamarin, and since then, the full version of the framework has been available for free. There are two ways

of developing applications using Xamarin: Xamarin.Forms and Xamarin Native. Mukesh et al.⁵ have presented a brief analysis of the Xamarin framework. Their work discusses the benefits of code sharing, which, in the case of Xamarin.Forms, can reach 100%. In their article, available APIs are com-

services like Bluetooth, cameras, and sensors are run using native code. Since the release of React Native, only a few research articles have been published regarding the framework's viability. A recent article⁷ compares React Native to Google's Flutter. The study focuses on app architecture

development of mobile applications for Android and iOS. This cross-platform solution was implemented in C/C++, Dart, and uses Google's Skia Graphics Engine. Unlike Xamarin and React Native, Flutter does not use native UI elements; instead, they are drawn using the Skia Graphics Engine. Flutter allows the use of "stateful hot reload"; changes are displayed in real time without rebuilding the entire application. Kho'i and Jahid¹⁰ compared the Flutter framework to React Native; their main topic was the comparison of application structure and state management. In conclusion, the results are rather subjective. There is a brief comparison of I/O performance and FPS between the frameworks, but it is not particularly thorough and multiple scenarios are not tested. It is much harder to find articles on the Flutter framework because it was released much later than the others.

UNLIKE XAMARIN AND REACT NATIVE, FLUTTER DOES NOT USE NATIVE UI ELEMENTS; INSTEAD, THEY ARE DRAWN USING THE SKIA GRAPHICS ENGINE.

pared to web-based frameworks, like Adobe PhoneGap and Apache Cordova. The Xamarin.Forms variant of the Xamarin framework makes use of the Model-View-ViewModel architecture, which makes the separation of concerns between views and business logic easier. This aspect of the framework is also discussed in that article. Another article that touches on the subject of Xamarin is by Sasi-daran,⁶ but it contains only a general comparison without going into much detail; the outline of the framework architecture is described, but no measurements of its performance are presented, and the ease of development with Xamarin is not analyzed.

The second framework (React Native) was developed by Facebook and released in 2015. It is based on the React framework used for web development, but it enables the use of native UI elements in mobile applications, unlike websites or frameworks like Apache Cordova. Much of the application was written using JavaScript, optionally, in TypeScript. Native UI elements and

but also touches on the performance of the frameworks. Frames per second (FPS) and disk input/output (I/O) operations were used as metrics to compare both approaches.

React Native was also touched upon in several other works, including by Bäcklund and Hedén,⁸ who described the product quality model in detail. In it, React Native is compared to PWAs, and the authors devote a major portion of their article to software quality evaluation. Major aspects of the comparison carried out include compatibility, usability, performance, reliability, and security. Another article worth noting is by Majchrzak et al.,⁹ who decided to put more emphasis on developer convenience. The authors identified the most significant characteristics: the ease of development, the available integrated drive electronics (IDEs), code reusability, and access to specific functionalities. In the aforementioned article, React Native is compared to both Ionic Framework and Fuse.

Flutter, the third framework, was released in 2019 and allows for the

Cross-platform and native framework comparison

A comparison between React Native and native Android was conducted by Danielsson,¹³ who focused primarily on framework performance. The author used metrics such as graphics processing unit frequency, CPU load, memory usage, and power consumption. The measurements were conducted with remarkable accuracy and illustrated using diagrams. Nearly all of the measurements indicated the advantages of using the native Android application over the React Native one, but the disparity between the two is not very significant.

Wu⁷ focuses on in-depth Flutter and React Native analysis. Disk I/O operations and FPS are measured and compared. The author's results demonstrate considerable differences between both cross-platform frameworks. On the one

hand, React Native handles I/O operations faster, but on the other hand, Flutter offers a better developer experience. The author pointed out that he did not compare cross-platform solutions to native technologies.

In the case of iOS, it is much more difficult to find articles in which the native and cross-platform approaches are compared. This is obviously because this system is less popular than Android, and hence, there are fewer tools available to develop applications and measure their performance.

One of the few works on native iOS applications is by Singh,¹⁴ who describes Apache Cordova mobile applications alongside traditional solutions. The article contains ample information about the advantages and disadvantages of both approaches, but there are no exact statistics confirming the correctness of the comparison.

To summarize, we note that many of the surveys comparing frameworks for mobile applications concern the Android system and older, less-popular cross-platform solutions. In many works,^{11,13} performance-comparison results between native Android applications and cross-platform solutions are described. RAM usage was also a frequently used metric by Ahti et al.¹² and Danielsson.¹³ A very important (sometimes the most important) feature is the access to native APIs. Full access to native libraries is possible when developing applications in languages such as Java or Kotlin. The situation is different for multiplatform solutions, where this access is limited. A description of access to native APIs can be found in the works of Sasidaran⁶ and in Kho'i and Jahid.¹⁰

As mentioned previously, there are very few studies comparing currently popular cross-platform solutions such as Xamarin, React Native, and Flutter.


Also, comparisons with the native iOS system and tests that take into account the CPU load are very rare, which is why we decided to carry out our research of the latest and most popular solutions.

METHODOLOGY


To compare the various solutions, we developed two applications using each technology. The first is a simple one designed to test the initial build size

in various forms in different technologies; for instance, the native iOS one does not have a dropdown menu as well as a checkbox. For this platform, the counterparts are picker and switch. The Multiple Form Controls page contains the same widgets as does the Form Control page, only more of them.

The Small List and Huge List pages appear to be the same but contain different numbers of list elements. Each



**ONE OF THE FEW WORKS ON NATIVE
IOS APPLICATIONS IS BY SINGH, WHO
DESCRIBES APACHE CORDOVA MOBILE
APPLICATIONS ALONGSIDE TRADITIONAL
SOLUTIONS.**



and application launch time as well as the RAM footprint. The second is a more complex application containing a few simple pages and controls for measuring the rendering speed.

Specification of applications

The simple application for each platform contains just the “Hello World” label displayed in the middle of the main application page. In case of this application, there were no deviations related to any technology because a label was available in every framework on all of the available operating systems.

The main page of the advanced application contains a list of buttons referencing other pages. This type of menu was chosen because it is available in all of the technologies. The first page (Form Control) consists of four types of widgets: label, edit text, dropdown menu, and checkbox. It should be mentioned that widgets can appear

of these pages is present in four versions: one with no background task and three with different background tasks running in a separate thread, that is, the CPU, I/O, and HTTP tasks. For the CPU task, sines and cosines were calculated so that the CPU load would be easily noticeable. The I/O task involved repeatedly writing and reading a short text to/from a file located in the document directory. This job simultaneously loaded the CPU and increased RAM usage. The HTTP task sent HTTP to get requests to the Google search engine. The result was ignored so as not to generate an unnecessary CPU load.

Performance

To measure the start-up time, both applications were launched manually, and the time it took to fully render the usable UI was measured. Then, the average of 10 measurements yielded the final result.

The application size was measured by checking the size of the default release build for each platform. In the case of Android, the size of the Android Package Kit (APK) file was measured, and for iOS, the size of .app files was measured.

For the Android applications, all of the performance metrics were measured using the Android Studio Profiler, and on iOS, all of the metrics were provided by XCode Instruments. To export data from Android Studio Profiler, a simple tool that reads data from graphs and saves them in a comma-separated values format was created. In XCode Instruments, it is not possible to generate a chart of CPU load and RAM usage for just one process (application), so the data were recorded manually from the detail pane of the XCode Instruments application.

On Android, the simple application was started, and the time series of RAM usage was captured with Android Studio Profiler. For the Advanced app, the page tested was opened, and then the usage time series was captured. On iOS, RAM usage is measured in the same way, except that XCode Instruments was used to capture the time series. From the time series, two main results were calculated: peak RAM usage and the value after the view was fully initialized and RAM usage was stabilized.

The CPU load test is analogous to the RAM usage test. The biggest difference is that CPU load tends not to fully stabilize after the view has been initialized, so instead, the average CPU load over 20 s was taken as the final result.

Ease of development

The comparison also included a few metrics related to the ease of development using each framework. By their nature, these metrics are much more

prone to subjectivity (for instance, it is difficult to measure which IDEs are better). When comparing the development experience, two sources of information were used: the experience of writing the applications for benchmark purposes and the Stack Overflow Developers Survey. Although the former source is, by its very nature, subjective, the survey does provide some hard data.

For analyzing the experience of writing the applications, several areas were considered, including the quality of available IDEs and tooling, feedback cycle length and compilation time, and quality of available libraries and their documentation as well as the availability of certain features in the base framework.

COMPARISON

The tests were carried out on real Android (LG G6) and iOS (iPhone 8) devices. Flagship devices were not used for testing so that the comparison could be more realistic, as just a small group of people can afford the latest devices.

Performance

Application size was the first metric measured. For Android, the size of the APK file was checked as was the .app file for iOS. The build size was examined for both simple and advanced applications. This highlighted the initial size of the framework itself as well as the size increase in the complex application.

As presented in Figure 1(a), the variances in the sizes of applications developed using individual solutions differ considerably depending on the target operating system. The sizes of native applications are comparable for both systems. The applications written using Flutter and Xamarin are significantly larger on iOS. The result

for React Native is very surprising: a simple Android application is slightly larger than an iOS one, but in the case of the advanced application, the size difference grows significantly. The advanced React Native application is more than five-times larger on Android compared to iOS. An analysis of the APK file showed that libraries are responsible for a significant portion of the size increase. This article compares the sizes of default release builds, and build size optimization could result in a very significant decrease in size for this scenario.

Start-up time is an essential metric showing how long it takes for the framework to launch. Application start-up results differed slightly on Android and iOS, and only for Xamarin.Forms was the difference substantial. Figure 1(b) shows that iOS applications had significantly faster start-up times compared to their Android counterparts. iOS is better at application launching due to improvements in the latest system versions, while native applications were faster than cross-platform apps. For both operating systems, Xamarin had the worst start-up time, but only on Android did the difference in comparison to the other solutions become huge.

Figure 1(c) presents the RAM usage of Android and iOS applications after entering the Form Control page. It is immediately clear that iOS manages memory better than Android because iOS apps consume much less memory than do their Android equivalents. The native application on Android consumes 100.8 MB (1,210%) more RAM than the native iOS app. For Xamarin and React Native, the differences are also significant: 133.24 and 109.32 MB, respectively. The biggest disparity can be seen for Flutter, which consumes 143.27 MB more than iOS.

In most CPU usage experiments, after a page without a background task has been opened, the CPU load decreases significantly after approximately

1.5–2 s. Most frameworks stopped using the CPU completely after the initial rendering of the page; React Native was the only framework that

continued to use the CPU, although it was roughly just 1–2%, with occasional spikes reaching roughly 5% of the CPU's usage.

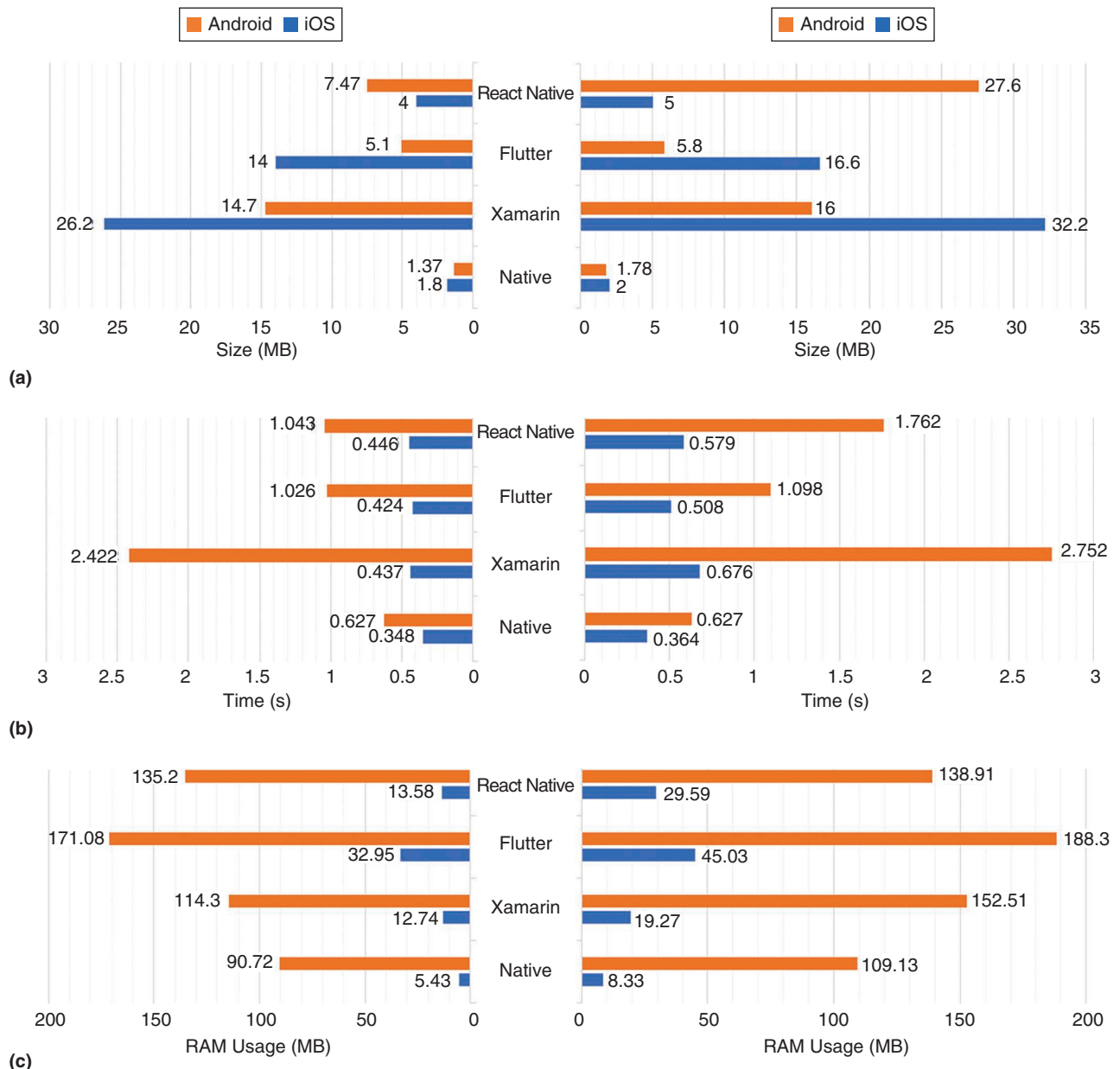


FIGURE 1. The experiments that take into account (a) the application size, (b) the start-up time, and (c) the RAM usage for the simple (left) and advanced (right) applications on the Android and iOS systems, respectively.

CPU usage on the controls page with a CPU task in the background is presented in Figure 2(a). Surprisingly, on Android (left), Flutter managed to complete the task faster than the native implementation. Another surprising result was the slowness of both Xamarin and React Native. Although Flutter and the native (Android) implementation

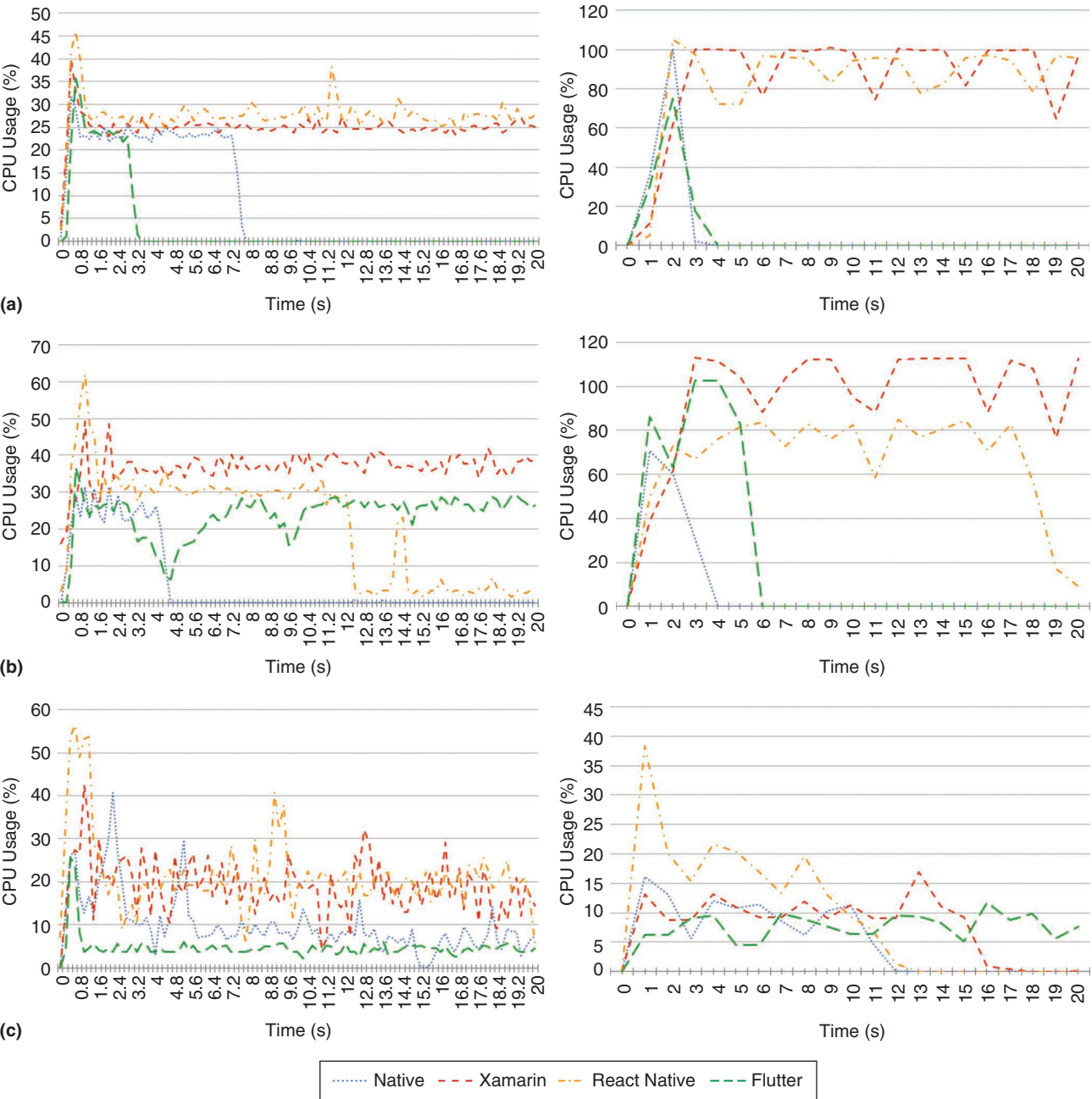


FIGURE 2. The CPU load experiments for the Android (left) and iOS (right) systems with background tasks running in separate threads: the (a) CPU, (b) I/O, and (c) HTTP tasks.

both finished the task in fewer than 4 and 8 s, respectively, both Xamarin and React Native did not finish the task during the 20-s time frame captured. For iOS (right), it can be seen that both the native (iOS) and Flutter applications finished their tasks in approximately 4 s, while for Xamarin and React Native, the tasks were not completed in with the 20-s time frame. For each technology, there was a sharp increase in the CPU load after entering the page. It is worth mentioning that the maximum value for Flutter (75%) was lower than for the native application (100.4%). For the React Native app, the CPU load oscillated between 72.3 and 97.4%. The chart for Xamarin.Forms looked similar: the values above 100% mean that more than one core was utilized. However, Apple XCode Instruments does not allow for checking how heavily loaded each core was.

In Figure 2(b), the CPU usage on the controls page with an I/O task in the background was presented. As expected, on Android (left), the native implementation was the fastest one, finishing after fewer than 5 s. React Native was the only other framework to finish the task during the 20-s time frame captured—in fact, in fewer than 15 s. Another interesting observation concerns the CPU usage of Xamarin, which is significantly higher than that of the other frameworks, staying at roughly 40% through the entire background task. For iOS (right), the native application performed best, completing the I/O task in 4 s. Flutter turned out to be slightly worse and finished in fewer than 6 s. Xamarin was not able to complete the task in fewer than 20 s. The CPU load for React Native did not fall to 0%, but 9.1% was recorded at the end. It should be noted that even though Xamarin did not finish the

task, it generated the highest CPU load nearly all of the time.

The chart with CPU usage for the network background task (HTTP) on the controls page is presented in Figure 2(c). On Android (left), Flutter maintained the lowest CPU usage, using approximately 5% of the CPU. The native Android implementation achieved a similar result, with roughly 8% of the CPU used. React Native and Xamarin had significantly higher CPU usage, averaging 20% with occasional spikes to 30% in the case of Xamarin, and 40% in the case of React Native. At first glance, on iOS (right), a huge increase in CPU load for the React Native application can be seen. Despite the large initial resource consumption (38.3%), React Native completed the task after 12 s, similar to the native application. Flutter failed to complete the task even though its maximum CPU load was equal to 11.7%. Xamarin finished the background task in roughly 16 s. The results for the other pages with all of the background tasks for both operating systems were very similar.

Ease of development

During the development of benchmark applications, multiple issues were encountered. Because ease of development is a highly subjective matter, these issues might not affect all developers to the same extent. Some issues could have been avoided thanks to the authors' prior experience, and some might have been encountered due to their inexperience. Nonetheless, the issues encountered during development are described in this section.

The framework posing the most problems was React Native, caused by the ecosystem created around its framework. For many libraries, even handling very basic features are of low

quality because they are maintained and documented by community developers. A seemingly simple task, i.e., creating a new thread, was not successfully implemented even after trying two libraries: react-native workers and react-native threads. Both of the libraries are maintained by the community and did not work correctly in iOS release builds.


Another important characteristic of the framework is how easy it is to use the tools available. In this aspect, of all the cross-platform frameworks, Flutter seemed to have the best quality tools. For the initial environment setup, compilation, deployment, and profiling, Flutter provides easy-to-use tools that work well. The same cannot be said of the other frameworks. The React Native compilation for release builds can take a very long time, and the time of completion is quite indeterministic. Sometimes the release build was completed in 2 min, and in other cases, 10 min were required. Such indeterminism was not noticed for any other framework.

The length of the feedback cycle is a very important aspect of mobile application development. Especially when designing the user interface, long feedback loops can significantly increase development time and decrease developer satisfaction. Of all the frameworks compared, Xamarin has by far the longest feedback loop (because there is not a hot reload tool). Some third-party solutions exist; however, they are not free. Both React Native and Flutter provide hot reload tools for free. During the development of benchmark applications, Flutter's hot reload solutions proved to be slightly faster and much more stable, as React Native sometimes did not refresh changes or refreshed the changed code twice.

TABLE 1. A comparison of all the tested solutions.

Platform	Xamarin. Forms	React Native	Flutter	Native Android	Native iOS
Start-up time	Slow	Medium	Medium	Fast	Fast
App size	Big	Medium	Medium	Small	Small
Memory usage	Medium	Medium	High	Small	Small
CPU usage	Medium to high	Medium to high	Medium	Medium	Medium
Development experience	Medium	Medium	Very good	Good	Good

Although it is hard to measure the quality of the development experience, surveys can provide some hard data. According to data from the 2019 Stack Overflow Developers Survey,¹⁵ Flutter is by far the most liked cross-platform framework, with 75.4% of developers who are working with Flutter having expressed interest in continuing to use it. For React Native, 62.5% of respondents gave the same answer, and in the case of Xamarin, the figure was just 48.3%. The survey also contained a question about having expressed interest in developing for respondents who were not working with a specific technology. For that question, React Native was at the top, with 13.1% of the developers wanting to learn it, Flutter was second with 6.7%, and Xamarin was third with just 4.9%.


f all the cross-platform solutions, Flutter seems to be the best one overall (Table 1). Although it does have the biggest memory footprint, its start-up time and overall performance are either the best of all the solutions or are on par with the other frameworks. It was also the only

framework to beat the native implementation in a benchmark (the CPU background task on Android). More importantly, Flutter provides great development experience. On top of that, today, with mobile devices becoming faster all the time, ease of development becomes increasingly important compared to the performance of the application developed.

Applications written with React Native perform well in some cases, feature the lowest average memory footprint of all the cross-platform frameworks, and have very good start-up times. However, in the authors' experience, developing applications using React Native is not as easy as with Flutter or even Xamarin in some respects. The greatest strength (and, at the same time, drawback) of the React Native ecosystem is its dependence on community-maintained libraries, which allows for rapid development, but also negatively affects the quality of documentation and the compatibility of libraries.

The applications written with Xamarin.Forms seem to be the worst based on the benchmarks. The framework does have the lowest memory consumption on iOS; however, iOS

applications generally have very small memory footprints, and it is rarely an issue. Xamarin has problems in very important areas; most noticeably, the start-up time on Android is a big detriment to user experience. Even for a "Hello World" application, the start-up time exceeded 2 s, and a more advanced application took nearly 3 s to open. Another big issue is application size. On top of that, Xamarin does not have any hot reload tools built in, significantly slowing down its development.

The landscape of cross-platform frameworks is very dynamic, so new frameworks should appear in the future. Further research should involve testing those solutions. It could also include comparisons with the increasingly popular PWAs applications (Angular, React.js, and Vue.js) and the fast-growing Kotlin Native solution. The comparison would also benefit from more research of user experience because this article has focused more on testing performance and the ease of development of applications. 

ACKNOWLEDGMENT

The research presented in this article was supported with funds from the Polish Ministry of Science and Higher Education assigned to AGH University of Science and Technology.

REFERENCES

1. "Number of apps available in leading app stores as of 1st quarter 2019," Statista, Inc., Hamburg, Germany, Apr. 2019. [Online]. Available: <https://www.statista.com/statistics/276623/number-of-apps-available-in-leading-app-stores/>
2. J. Ohrt and V. Turau, "Cross-platform development tools for smartphone applications," *Computer*, vol. 45,

- no. 9, pp. 72–79, Sept. 2012. doi: 10.1109/MC.2012.121.
3. “Global mobile OS market share in sales to end users from 1st quarter 2009 to 2nd quarter 2018,” Statista, Inc., Hamburg, Germany, Aug. 2018. [Online]. Available: <https://www.statista.com/statistics/266136/global-market-share-held-by-smartphone-operating-systems>
 4. “Cross-platform app development: Trends, tactics & tools,” RipenApps, Oct. 26, 2018. [Online]. Available: <https://medium.com/@ripenapps/cross-platform-app-development-trends-tactics-tools-d05f78bc657c>
 5. P. Mukesh, P. Dhananjay, and P. Archit, “Study on Xamarin cross-platform framework,” *Int. J. Tech. Res. Appl.*, vol. 4, no. 4, pp. 13–18, July–Aug. 2016.
 6. S. Sasidaran, “Survey on native and hybrid mobile application,” *Int. J. Adv. Res. Comput. Eng. Technol. (IJARCET)*, vol. 6, no. 9, pp. 1389–1393, Sept. 2017.
 7. W. Wu, “React Native vs Flutter, cross-platform mobile application frameworks,” B.Eng. thesis, Helsinki Metropolia Univ. Appl. Sci., Helsinki, Vantaa, and Espoo, Finland, Mar. 2018.
 8. L. Bäcklund and O. Hedén, “Evaluating React Native and progressive web app development using ISO 25010,” B.A. thesis, Dept. Comput. Inf. Sci., Linköping Univ., 2018.
 9. T. A. Majchrzak, A. Bjørn-Hansen and T. M. Grønli, “Comprehensive analysis of innovative cross-platform app development frameworks,” in *Proc. 50th Hawaii Int. Conf. Syst. Sci.*, 2017, pp. 6162–6171. doi: 10.24251/HICSS.2017.745.
 10. F. M. Kho'i and J. Jahid, “Comparing native and hybrid applications with focus on features,” B.A. thesis, Faculty Comput., Blekinge Inst. Tech., Karlskrona, Sweden, 2016.

ABOUT THE AUTHORS

PIOTR NAWROCKI is an associate professor at the Institute of Computer Science at AGH University of Science and Technology, Krakow, Poland. His research interests include distributed systems, computer networks, mobile systems, cloud computing, the Internet of Things, and service-oriented architectures. Contact him at piotr.nawrocki@agh.edu.pl.

KRZYSZTOF WRONA is a software developer at EZY, focusing on .NET solutions, including Xamarin.Forms mobile applications. His research interests include software development and mobile applications. Wrona received an M.A. in computer science from AGH University of Science and Technology, Krakow, Poland. Contact him at krzychuwr1@gmail.com.

MATEUSZ MARCZAK is a software developer. Marczak received an M.A. in computer science from AGH University of Science and Technology, Krakow, Poland. Contact him at mpmarczak@gmail.com.

BARTŁOMIEJ SNIEZYNSKI is an associate professor at the Institute of Computer Science at AGH University of Science and Technology, Krakow, Poland. His research interests include machine learning, multiagent systems, and knowledge engineering. Contact him at bartlomiej.sniezynski@agh.edu.pl.

11. F. Otávio, R. Silveira, F. da Silva, P. de Andrade, and A. Albuquerque, “Cross platform app a comparative study,” *Int. J. Comput. Sci. Inform. Technol.* vol. 7, no. 1, pp. 33–40, 2015.
12. V. Ahti, S. Hyrynsalmi, and O. Nevalainen, “An evaluation framework for cross-platform mobile app development tools: A case analysis of Adobe PhoneGap framework,” in *Proc. Int. Conf. Comput. Syst. Technol. – CompSysTech*, Palermo, Italy, June 2016, pp. 41–48. [Online]. Available: <https://doi.org/10.1145/2983468.2983484>. doi: 10.1145/2983468.2983484.
13. W. Danielsson, “React Native application development—A comparison between native Android and React Native,” M.A. thesis, Dept. Comput. Inf. Sci, Human-Centered Syst., Linköping Univ., Linköping, Norrköping, and Lidingö, Sweden, 2016.
14. N. Singh, “An comparative analysis of Cordova Mobile Applications V/S Native Mobile Application,” *Int. J. Recent Innovat. Trends Comput. Commun.*, vol. 3, no. 6, pp. 3777–3782, 2015. doi: 10.17762/ijritcc.v3i6.4536.
15. “Developer survey results 2019,” Stack Overflow. Accessed Mar. 25, 2020. [Online]. Available: https://insights.stackoverflow.com/survey/2019#technology-_most-loved-dreaded-and-wanted-other-frameworks-libraries-and-tools