Analyzing the Performance of Apps Developed by using Cross-Platform and Native Technologies

Lucas Pugliese Barros
Informatics Coordination
Federal Institute of Alagoas
Maceió, Alagoas, Brasil
lucas.pugliese.barros@gmail.com

Flávio Medeiros
Informatics Coordination
Federal Institute of Alagoas
Maceió, Alagoas, Brasil
flavio.medeiros@ifal.edu.br

Eduardo Moraes
Informatics Coordination
Federal Institute of Alagoas
Maceió, Alagoas, Brasil
ecmoraes@gmail.com

Anderson Feitosa Júnior
Informatics Coordination
Federal Institute of Alagoas
Maceió, Alagoas, Brasil
andersonmfjr@gmail.com

***Abstract*—The number of mobile devices are increasing worldwide and there are a number of new mobile apps being delivered daily. When developing mobile applications, developers need to support a number of platforms, such as *iOS* and *Android*. Many cross-platform technologies appeared, including *Flutter* and *React Native*, to avoid the need of developing different applications for every platform. In this context, companies and developers have to analyze different issues when selecting a cross-platform or native technology. Both strategies have positive and negative points, and one important aspect when choosing a technology is performance. In this study, we perform a comparative study to analyze the performance of mobile applications developed by using *Flutter*, *React Native*, and *iOS* and *Android* native technologies. The results show that native technologies are still a little faster for most functionalities, but there are also a number of cases in which *Flutter* and *React Native* perform statically equivalent when compared to native technologies. Our study complements previous work by including these two modern cross-platform technologies that have not be considered in comparative studies previously.**

***Index Terms*—Mobile Development, Performance Analysis**

I. Introduction

Mobile devices are currently a crucial part of our daily life. We find mobile applications for a wide range of domains, such as health, tourism, planning, and sports. There are many a number of platforms that uses different Software Development Kits (SDK) to build mobile applications, including *iOS*, and *Android*. In this context, developers should provide their solutions across those platforms. A number of cross-platform technologies appeared, allowing developers to support different platforms by using a single source code, such as *React Native*, and *Flutter* [1–4].

In particular, when using native development, companies need to develop different mobile applications for every supported platform. Thus, the cost of native development gets higher. For this reason, cross-platform development has emerged as a potential alternative. On the other hand, cross-platform technologies also have negative points [3, 5, 6]:

- Performance: it is one of the most important requirement of an application, and it might be slower in applications developed by using cross-platform technologies;
- Harder code design: developers have to adapt their design and functionalities to handle the specific peculiarities of every platform;
- Longer time for new features: every new feature for *Android* or *iOS* takes some time to be available in cross-platform frameworks.

To help developers and companies to select a technology for their mobile applications, we performed a comparative study to analyze the performance of cross-platform applications developed by using *Flutter* and *React Native* when compared to native applications written in *Java* and *Swift*, for *Android* and *iOS* devices respectively. This study complements previous work by including two modern cross-platform technologies not considered before [6–8]. The first on is *Flutter*, which has appeared as a promising cross-platform technology that aims to bundle mobile applications with performance improvements. This technology has been used by many companies, including *Alibaba*, and *Google Ads*. The second is *React Native*, which is the most popular technology currently and it is also used by many enterprises, such as *Facebook*, *Airbnb*, *Tesla*, *Walmart*, and *Uber*.

To perform this comparative study, we define a set of functionalities that are present in most mobile applications. Then, we implemented these functionalities by using different languages and technologies: (1) in *Flutter* by using the *Dart* language; (2) in *JavaScript* with *React Native*; (3) in *Java* for *Android* devices; and (4) in *Swift* for *iOS* devices. In all implementations we use the same functionalities and layout, and we strictly follow the documentation of each technology to get the best benefits of each one. The functionalities implemented include storing and retrieving information of local storage, making *HTTP* calls, and rendering data as lists. To answer our research questions, we use User Interface (UI) Test [9, 10] to automatically run every implementation and execute each functionality 100 times to compute the time of execution in the different technologies.

Our study reveals that the applications developed by using native technologies, i.e., *Swift* and *Java*, are still a little faster for most functionalities. However, there are also cases in which *Flutter* and *React Native* are statistically equivalent in terms of performance, such as storing and retrieving data by using local storage in *Android* and *iOS* devices.

DOI reference number: 10.18293/SEKE2020-122

The remainder of this paper is organized as follows. In Section II, we present the settings of our comparative study to analyze the performance of mobile applications. Section III shows the results of our study with regards to *Flutter*, *React Native*, and *iOS* and *Android* development. In Section IV, we depict the threats to validity of our study, and Section V discusses some implications to practice. In Section VI, we present the related work, and we discuss the concluding remarks in Section VII.

We provide all information about this comparative study, including the source codes, and the result data in a complementary website.[1]

II. Study Settings

In this section, we present the settings of our comparative study to analyze the performance of mobile applications developed by using cross-platform and native technologies. To better structure our study, we use the Goal, Question, and Metrics (GQM) approach [11].

A. Definition

The goal of this comparative study is to analyze implementations of a mobile application for the purpose of evaluation with respect to verifying the performance of cross-platform and native mobile technologies in the context of the languages *Dart* (*Flutter*), *JavaScript* (*React Native*), *Java*, and *Swift*. In particular, this study addresses the following research questions:

- **RQ1.** What is the difference in terms of performance for *Android* applications developed by using *Java*, *Flutter* (*Dart*), and *React Native* (*JavaScript*)?
- **RQ2.** What is the difference in terms of performance for *iOS* applications developed by using *Swift*, *Flutter* (*Dart*), and *React Native* (*JavaScript*)?

To answer these two research questions included in our study, we define four metrics:

- **REMOTE:** this metric computes the processing time to make a request to an external Application Programming Interface (API). It computes the time of the *HTTP* request, that is, the time to make the request and receive all data by using the JavaScript Object Notation (JSON);
- **RENDER:** it computes the time to render the first five items of a list by showing them on the screen for the users;
- **STORE:** this metric computes the time to save one item of a list in the local database;
- **RETRIEVE:** it computes the time necessary to retrieve the information of five items from the local database.

B. Planning and Operation

In this section, we describe the subjects, instrumentation, and operation of our study.

Our study uses *Flutter* (Dart) and *React Native* (JavaScript) as the cross-platform technologies, and *Java* and *Swift* as the natives ones. The reason to select *Flutter* is because it has appeared as a promising cross-platform technology that aims to bundle mobile applications with performance improvements. As performance is a negative point of cross-platform applications, it makes sense to include *Flutter* in our study. We choose *React Native* because it is the most common framework in practice. In addition, both frameworks are used by known companies, such as *Alibaba*, *Facebook*, *Airbnb*, *Walmart*, and *Uber*. We select *Android* and *iOS* because they are the two most common platforms for mobile devices.

To perform this comparative study, we developed four implementations of an application by using different technologies: (1) *Flutter*; (2) *React Native*; (3) *Java*; and (4) *Swift*. The implementations contain the same functionalities and layout, and we developed the applications by strictly following the documentation of each technology.

To compute metric *REMOTE*, we select the *Google APIs Explorer* service because there is no authentication restriction to access it, and because it is possible to return a considerable amount of information (258 items) through a single request. As the local database, we use the *SQLite* database because it is compact as well as available in the *Android*, *IOS*, *Flutter*, and *React Native* SDKs. In addition, the documentations of the technologies recommend to use this database in practice.

In Figure 1, we can see one implementation of the application written in *Swift* for *iOS*. On the left-hand side, we see the list of items returned by the API. On the right-hand side, we present the items retrieved from the *SQLite* database. In Figure 2, we show the screens but considering the implementation for *Android*, written in *Java*.

Fig. 1. Interface of one implementation of the application in *iOS*.

To answer our research questions, we created user interface tests to execute the applications 100 times and to compute the time to perform each functionality in the different technologies. The user interface tests repeat a sequence of steps 100 times on real devices, simulating the behavior of users. Thus, instead of run the experiment manually, it was possible to automate the complete process.

[1] http://cpsoftware.com.br/performance-study

To compute the processing time for each metric, we create a class in *Java*, *Swift*, *JavaScript*, and *Dart*, respectively for *Android*, *iOS*, *React Native*, and *Flutter*. So that, the logic was the same for all four implementations with the purpose of avoiding influences on the processing time of the metrics. These classes are responsible for managing all metrics, calculating the processing time in milliseconds, and formatting and printing the processing time of the metrics. With the data collected, the next step was to format the data in the Comma Separated Values (CSV) format to run the statistics by using the *R Project for Statistical Computing*.

We execute the study on a MacBook Pro 2.4GHz dual-core Intel Core i5 8GB, running Mac OS X 10.8 Mountain Lion. Furthermore, we run the applications by using an *iPhone* 7 32GB 3GB RAM and a *Samsung Galaxy* S8 64GB 4GB RAM. That is, we run the *Android* native application and the *Flutter* and *React Native* versions for *Android* by using the *Samsung Galaxy*, and the *iOS* native application and the *Flutter* and *React Native* versions for *iOS* by using the *iPhone*. We perform all the analyses by using the same network connection.

To run the experiment, we need several tools. We use *Android Studio* 3.6, *Android* 10, *Java* 1.8, *Flutter* version 1.12.13, *Xcode* 11.3.1, *Visual Studio Code* 1.42, *React Native* 0.61, *Volley* 1.1.1, *SQLite* 3.31.1, *Espresso* 3.1.1, *XCTest* 5.2, and *Detox* 15.4.2.

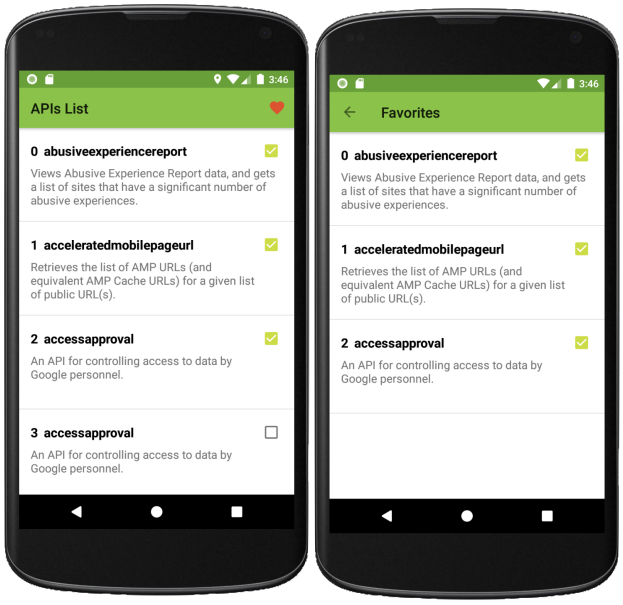

Fig. 2. Interface of one implementation of the application in *Android*.

Next, we interpret and discuss the results of this study to analyze the performance of mobile applications developed by using cross-platform and native technologies.

III. Results

In the next subsections, we present the results of our comparative study. Section III-A presents the results related to the *Android* platform, and in Section III-B, we show the results of the *iOS* platform.

A. Android

In this section, we answer **RQ1** by presenting the results for each metric by considering the applications running on the *Android* device.

REMOTE: In Figure 3a, we show the mean and the confidence interval (95%) for each technology regarding metric *REMOTE*. The means are 1772, 1818, and 2295 milliseconds for *React Native*, *Android* native, and *Flutter* respectively.

To check if the data is normal distributed, we use the *Shapiro-Wilk* normality test (95%). The null-hypothesis of this test is that the population is normally distributed. Thus, if the *p-value* is less than the chosen alpha level (0.05), then the null hypothesis is rejected and there is evidence that the data tested are not normally distributed [12, 13].

- Null Hypothesis (H_0): it tests if the population is normally distributed;
- Alternative Hypothesis (H_1): there is evidence that the population is not normally distributed.

By running the *Shapiro-Wilk* normality test (95%), we find that the data is not normal. Thus, we compare the groups of data by using the *Wilcoxon* test [12, 13], which considers:

- Null Hypothesis (H_0): there is not any difference in the data sets, that is, the medians are equal;
- Alternative Hypothesis (H_1): there is evidence that the data sets are different in terms of median.

The results show that the performances of *Android* native and *React Native* are statistically faster than the performance of the application developed by using *Flutter*. That is, the median values are different according to the statistical test. In Table I, we present the statistical tests we run and their respective results.

Test	p-value	(H_0)
Shapiro-Wilk for *Android*	$9.9e^{-13}$	rejected
Shapiro-Wilk for *Flutter*	$2.9e^{-08}$	rejected
Shapiro-Wilk for *React Native*	$5.7e^{-11}$	rejected
Wilcoxon for *Android* and *Flutter*	$2.3e^{-13}$	rejected
Wilcoxon for *Android* and *React*	$6.9e^{-04}$	rejected
Wilcoxon for *Flutter* and *React*	$2.2e^{-16}$	rejected

TABLE I
STATISTICAL TESTS FOR METRIC REMOTE IN ANDROID APPLICATIONS.

RENDER: In Figure 3b, we present the mean and the confidence interval (95%) for each technology regarding metric *RENDER*. The means are 173, 247, and 277 for *Android* native, *Flutter*, and *React Native* respectively.

The data is not normal here also according to the *Shapiro-Wilk* test, as we can see in Table II. *Android* native is the technology with better performance with statistical significance. It has the lowest median, which is different from the other data sets according to the *Wilcoxon* test. On the other hand, there is no statistical difference between the performance of the technologies *React Native* and *Flutter*.

STORE: In Figure 3c, we show the results for metric *Store*. We present the mean and the confidence interval (95%) for

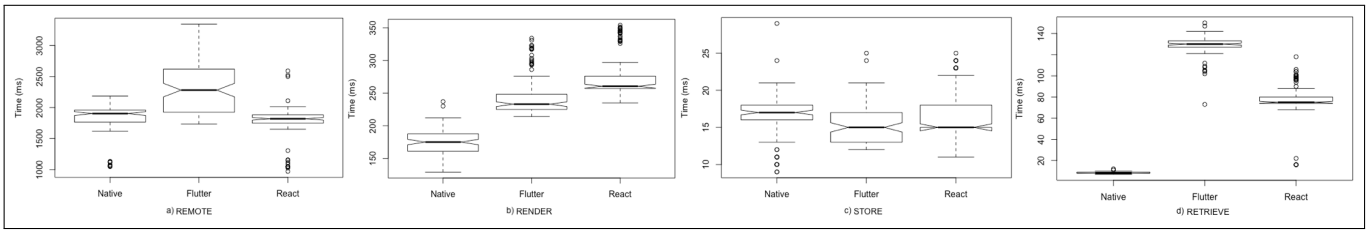

Fig. 3. Results for metrics in *Android*.

Test	p-value	(H_0)
Shapiro-Wilk for *Android*	0.3	accepted
Shapiro-Wilk for *Flutter*	$9.2e^{-12}$	rejected
Shapiro-Wilk for *React Native*	$1.1e^{-12}$	rejected
Wilcoxon for *Android* and *Flutter*	$2.2e^{-16}$	rejected
Wilcoxon for *Android* and *React*	$2.2e^{-16}$	rejected
Wilcoxon for *Flutter* and *React*	$1.3e^{-14}$	rejected

TABLE II
STATISTICAL TESTS FOR METRIC RENDER IN ANDROID APPLICATIONS.

each technology. The means are 15.59, 16.43, and 16.96 for *Flutter*, *Android* native, and *React Native* respectively.

In Table III. we present the statistical tests and their results. There is no statistical significance in the data sets. That is, the performance of all technologies is statistically equivalent.

Test	p-value	(H_0)
Shapiro-Wilk for *Android*	$2.97e^{-10}$	rejected
Shapiro-Wilk for *Flutter*	$2.486e^{-5}$	rejected
Shapiro-Wilk for *React Native*	$4.038e^{-10}$	rejected
Wilcoxon for *Android* and *Flutter*	0.003335	rejected
Wilcoxon for *Android* and *React*	0.03211	rejected
Wilcoxon for *Flutter* and *React*	0.0096	rejected

TABLE III
STATISTICAL TESTS FOR METRIC LIKE IN ANDROID APPLICATIONS.

RETRIEVE: In Figure 3d, we show the results for metric *Retrieve Data*. We present the mean and the confidence interval (95%) for each technology. The means are 8.35, 77, and 128.6 for *Android* native, *React Native*, and *Flutter* respectively.

In Table IV, we present the results of metric *RETRIEVE*. The *Android* native technology is the fastest one with statistical significance. On the other hand, the performance of the technologies *React Native* and *Flutter* is statically equivalent.

Test	p-value	(H_0)
Shapiro-Wilk for *Android*	$6.24e^{-12}$	rejected
Shapiro-Wilk for *Flutter*	$1.074e^{-11}$	rejected
Shapiro-Wilk for *React Native*	$7.78e^{-13}$	rejected
Wilcoxon for *Android* and *Flutter*	$2.2e^{-16}$	rejected
Wilcoxon for *Android* and *React*	$2.2e^{-16}$	rejected
Wilcoxon for *Flutter* and *React*	$2.2e^{-16}$	rejected

TABLE IV
STATISTICAL TESTS FOR METRIC LIKE IN ANDROID APPLICATIONS.

SUMMARY

Regarding data rendering and retrieving data from local storage, Android native is faster than Flutter and React Native. However, performance is statistically equivalent when considering accessing remote data and storing data in local storage.

B. iOS

In this section, we answer **RQ2** by showing the results of our experiment in *iOS*. Next, we present the results for each metric.

REMOTE: In Figure 4a, we show the mean and the confidence interval (95%) for each technology regarding metric *REMOTE*. The means are 45.35, 1,802.46, and 2,474.89 milliseconds for *iOS* native, *React Native*, and *Flutter* respectively.

In Table V, we present the statistical tests and their results. In this metric, *iOS* native is statically faster than *Flutter* and *React*. Further, *React* is statistically faster than *Flutter*.

Test	p-value	(H_0)
Shapiro-Wilk for *Android*	0.002446	rejected
Shapiro-Wilk for *Flutter*	$7.975e^{-11}$	rejected
Shapiro-Wilk for *React Native*	$6.888e^{-15}$	rejected
Wilcoxon for *Android* and *Flutter*	$2.2e^{-16}$	rejected
Wilcoxon for *Android* and *React*	$2.2e^{-16}$	rejected
Wilcoxon for *Flutter* and *React*	$5.805e^{-12}$	rejected

TABLE V
STATISTICAL TESTS FOR METRIC REMOTE IN *i*OS APPLICATIONS.

RENDER: In Figure 4b, we present the mean and the confidence interval (95%) for each technology regarding metric *RENDER*. The means are 15.79, 56.71, and 72.91 for *iOS* native, *React Native*, and *Flutter* respectively.

In Table VI, we present the statistical tests and their results. In this metric, *iOS* native is also statically faster than *Flutter* and *React*. Further, *React* is statistically faster than *Flutter*.

Test	p-value	(H_0)
Shapiro-Wilk for *Android*	$7.206e^{-15}$	rejected
Shapiro-Wilk for *Flutter*	$2.2e^{-16}$	rejected
Shapiro-Wilk for *React Native*	$2.042e^{-15}$	rejected
Wilcoxon for *Android* and *Flutter*	$2.2e^{-16}$	rejected
Wilcoxon for *Android* and *React*	$2.2e^{-16}$	rejected
Wilcoxon for *Flutter* and *React*	$2.2e^{-16}$	rejected

TABLE VI
STATISTICAL TESTS FOR METRIC RENDER IN *i*OS APPLICATIONS.

STORE: In Figure 4c, we show the results for metric *STORE*. We present the mean and the confidence interval (95%) for each technology. The means are 2.89, 5.73, and 6.23 for *React Native*, *iOS* native, and *Flutter* respectively.

In Table VII, we present the statistical tests and their results. In this metric, *React Native* is statically faster than *iOS* Native and *Flutter*.

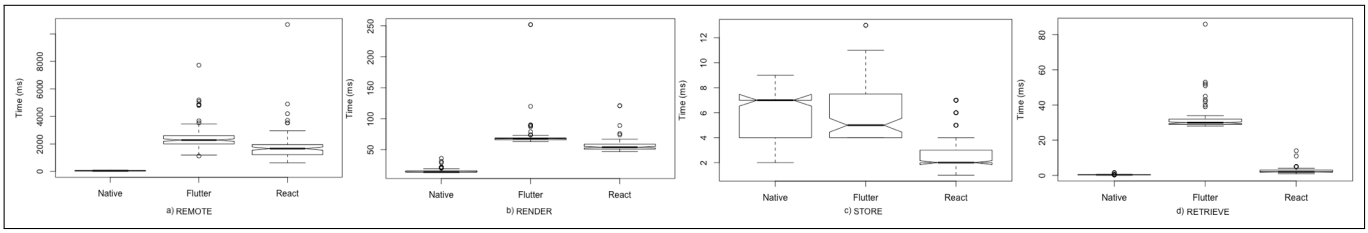

Fig. 4. Results for metrics in *iOS*.

Test	p-value	(H_0)
Shapiro-Wilk for *Android*	$1.346e^{-11}$	rejected
Shapiro-Wilk for *Flutter*	$4.202e^{-08}$	rejected
Shapiro-Wilk for *React Native*	$1.94e^{-13}$	rejected
Wilcoxon for *Android* and *Flutter*	0.5488	rejected
Wilcoxon for *Android* and *React*	$2.2e^{-16}$	rejected
Wilcoxon for *Flutter* and *React*	$2.2e^{-16}$	rejected

TABLE VII
STATISTICAL TESTS FOR METRIC RENDER IN *i*OS APPLICATIONS.

RETRIEVE: In Figure 4d, we show the results for metric *RETRIEVE*. We present the mean and the confidence interval (95%) for each technology. The means are 0.38, 2.82, and 32.18 for *Android* native, *React Native*, and *Flutter* respectively.

Here, the hypotheses of all statistical tests were rejected with $p - value = 2.2e^{-16}$. It is the lowest number that *R* can represent, that is, the null hypotheses (H_0) are all rejected with a small *p-value*, as shown in Table VIII.

Test	p-value	(H_0)
Shapiro-Wilk for *Android*	$2.2e^{-16}$	rejected
Shapiro-Wilk for *Flutter*	$2.2e^{-16}$	rejected
Shapiro-Wilk for *React Native*	$2.2e^{-16}$	rejected
Wilcoxon for *Android* and *Flutter*	$2.2e^{-16}$	rejected
Wilcoxon for *Android* and *React*	$2.2e^{-16}$	rejected
Wilcoxon for *Flutter* and *React*	$2.2e^{-16}$	rejected

TABLE VIII
STATISTICAL TESTS FOR METRIC RENDER IN *i*OS APPLICATIONS.

SUMMARY

Regarding data rendering, accessing remote data and retrieving data from local storage, iOS native is faster than Flutter and React Native. However, performance is statistically equivalent or faster when considering storing data into local storage.

IV. THREATS TO VALIDITY

Construct Validity. It refers to whether the functionalities that we choose are indeed relevant for mobile applications. We minimize this threat by analyzing existing mobile applications. The functionalities that we select, i.e., call remote API, render items as lists, and store and retrieve data from database, are present in almost all applications that we analyzed in our study.

Internal Validity. In our study, we implemented a specific class in different languages, i.e., *Java*, *Swift*, *JavaScript*, and *Dart*, to measure the time of performing the functionalities in the exact way for each technology. Furthermore, we select a stable internet connection to avoid differences when running the applications in the different technologies. However, notice that we may still face network differences. To minimize this threat, we run the experiment for all technologies in an interval of three hours.

External Validity. We analyze four functionalities that we commonly find in existing applications. However, we have not considered functionalities that use resources, such as GPS and camera. Thus, we cannot generalize the results of this study to this context.

V. IMPLICATIONS TO PRACTICE

The difference in terms of performance among *React Native* and *Flutter* when compared to native technologies can be explained based on how these cross-platform technologies work. In *React Native*, the source code is not compiled to *C/C++* or other native languages. Instead, the user interface components are compiled into native equivalents and *JavaScript* is executed on a separate thread (called bridge). In *Flutter*, the source code is compiled to native language directly, but it also uses an engine that goes together with the source code of the mobile applications, and a platform channel is necessary to access native resources, such as camera and GPS. That is, they work as intermediates that may decrease performance also. However, these technologies are evolving, they are used in practice by many companies, and they can perform as fast as native technologies for specific functionalities, such as storing and retrieving data by using local storage, as we shown in our comparative study.

Another important point to take into account is the costs for development. When using native technologies, such as *Java* and *Swift*, developers need to develop the same application for every supported platform, making the development more expensive and time-consuming. Thus, when performance requirements are not extremely important, it makes sense to use cross-platform technologies, as their performance is closer to natives ones to most functionalities, as we have shown in our current study.

VI. RELATED WORK

In this section, we compare our study with previous work. Sommer and Krusche [8] performed a comparative study with a number of technologies, including *Titanium*, *Rhodes*, and *PhoneGap*. The authors recommend to use cross-platform technologies in general, but they alert for high requirements with regards to performance issues, usability or native user experience. In [7], the authors compared the same three cross-platform technologies and measured performance in terms of

memory, CPU usage and power consumption. The former provided a similar result when comparing to our study in terms of performance. In the latter study, the authors used a different strategy to measure performance, as we computed the time of execution in our study.

Heitkötter et al. [14] performed a study by comparing the *PhoneGap* and *Titanium Mobile* technologies. The authors compared the technologies by considering the native look and feel, supported platforms, license and costs, as well as the application speed at start-up and run-time. Different from *React Native* and *Flutter*, the technologies used do not generate native code, they use a *Web app* approach knows as *WebView* [15], which let the application slower, as discussed by the authors in their results. Furthermore, technologies that use *WebViews* may cause security issues [16].

In the study of Xiaoping et al. [6], the authors compared the *Apache Cordova*, *Microsoft Xamarin*, and *Appcelerator Titanium* against the native technologies, that is, *Android* and *iOS*. The study is similar to ours, it presents trade-offs of different technologies and offer guidance in selecting an appropriate technology based on performance requirements. Further, the results are similar with our results, but the results of *Flutter* and *React Native* seem to be closer to the performance of native technologies.

In [17], the authors discussed the different strategies used by cross-platform technologies, and mention the advantages in productivity when using a model-driven approach. Our study complements all these related work by considering two modern cross-platform technologies for mobile development.

VII. Conclusions

In this study, we present a comparative study to analyze the performance of cross-platform and native technologies for mobile development. To run this study, we developed the same application by using different technologies: *Android (Java)*, *iOS (Swift)*, *Flutter (Dart)*, and *React Native (JavaScript)*. Our results reveal that native technologies are faster, as we expected, but the performance of the current cross-platform technologies are pretty closer. In a number of cases, we could not show statistical different among the performance of the applications written in different technologies. For instance, when storing and retrieving data by using local storage in *Android* and *iOS* devices. As future work, we plan to implement more functionalities in the applications to run the study again considering different aspects, such including functionalities that use geolocation, camera, and accelerometer.

References

1. Latif, M., Lakhrissi, Y., Nfaoui, E. H. & Es-Sbai, N. *Cross platform approach for mobile application development: A survey* in *International Conference on Information Technology for Organizations Development* (IEEE, 2016), 1–5.
2. Palmieri, M., Singh, I. & Cicchetti, A. *Comparison of cross-platform mobile development tools* in *International Conference on Intelligence in Next Generation Networks* (IEEE, 2012), 179–186.
3. Javeed, A. *Performance Optimization Techniques for ReactJS* in *Int. Conf. on Electrical, Computer and Communication Technologies* (IEEE, 2019), 1–5.
4. Serrano, N., Hernantes, J. & Gallardo, G. Mobile Web Apps. *IEEE Software* **30,** 22–27 (2013).
5. Lin, H. & Lee, G. *Building a Secure Cross Platform Enterprise Service Mobile Apps Using HTML5* in *International Conference on Network-Based Information Systems* (IEEE, 2015), 162–166.
6. Jia, X., Ebone, A. & Tan, Y. *A Performance Evaluation of Cross-Platform Mobile Application Development Approaches* in *Int. Conf. on Mobile Software Engineering and Systems* (Association for Computing Machinery, 2018), 92–93.
7. Dalmasso, I., Datta, S. K., Bonnet, C. & Nikaein, N. *Survey, comparison and evaluation of cross platform mobile application development tools* in *Int. Wireless Communications and Mobile Computing Conference* (IEEE, 2013), 323–328.
8. Sommer, A. & Krusche, S. *Evaluation of cross-platform frameworks for mobile applications* in *Software Engineering - Workshopband* (eds Wagner, S. & Lichter, H.) (Gesellschaft für Informatik e.V., 2013), 363–376.
9. Daniel, F. *et al.* Understanding UI Integration: A Survey of Problems, Technologies, and Opportunities. *IEEE Internet Computing* **11,** 59–66 (2007).
10. Tirodkar, A. A. & Khandpur, S. S. *EarlGrey: iOS UI Automation Testing Framework* in *International Conference on Mobile Software Engineering and Systems* (IEEE/ACM, 2019), 12–15.
11. Basili, V., Caldiera, G. & Rombach, D. H. in *Encyclopedia of Software Engineering* (Wiley, 1994).
12. Kanji, G. K. *100 statistical tests* 3rd ed. (Sage Publications, 2006).
13. Boslaugh, S. & Watters, P. A. *Statistics in a nutshell - a desktop quick reference.* (O'Reilly, 2008).
14. Heitkötter, H., Hanschke, S. & Majchrzak, T. A. *Comparing Cross-platform Development Approaches for Mobile Applications* in *Int. Conf. on Web Information Systems and Technologies* (2012).
15. Shin, D., Yao, H. & Rosi, U. *Supporting Visual Security Cues for WebView-Based Android Apps* in *Proceedings of the Annual Symposium on Applied Computing* (ACM, 2013), 1867–1876.
16. Bao, W., Yao, W., Zong, M. & Wang, D. *Cross-Site Scripting Attacks on Android Hybrid Applications* in *Proceedings of the International Conference on Cryptography, Security and Privacy* (ACM, 2017), 56–61.
17. Gaouar, L., Benamar, A. & Bendimerad, F. T. *Model Driven Approaches to Cross Platform Mobile Development* in *Proceedings of the International Conference on Intelligent Information Processing, Security and Advanced Communication* (Association for Computing Machinery, 2015).