Ecological Impact of Native versus Cross-Platform Mobile Apps: a Preliminary Study

Vincent Frattaroli, Olivier Le Goaer, Olivier Philippot

Ecological Impact of Native versus Cross-Platform Mobile Apps: a Preliminary Study

Vincent Frattaroli
Inside App
Paris, France
vincent.frattaroli@insideapp.fr

Olivier Le Goaer
E2S UPPA, LIUPPA
Universite de Pau et des Pays de l'Adour
Pau, France
olivier.legoaer@univ-pau.fr

Olivier Philippot
Greenspector
Nantes, France
ophilippot@greenspector.com

***Abstract*—What are the best mobile development approaches to cut the carbon footprint? To answer this question, this experience paper provides a life-size comparison of native versus cross-platform frameworks prevailing in the mobile software industry at the time of writing, namely Kotlin Multiplatform Mobile, React Native and Flutter. To do this, we collected metrics related to the package size, network traffic and battery drain issued by a boilerplate application developed following the different approaches. Our preliminary findings tend to show that the cross-platform solutions perform quite well.**

***Index Terms*—android, ios, kotlin, react, flutter, carbon**

I. Introduction

Within a decade, the mobile software sector has seen tremendous success. The landscape has also reorganised, leading to the overwhelming dominance of 2 mobile platforms that now share the market: almost 71% for Android (Google) and 27% for iOS (Apple). However, this market fragmentation is still a concern for mobile developers. Either they opt for native development, but have to write the app twice, or they opt for cross-platform development to write a single code base. The pros and cons of each development method are regularly debated, whether from a time-to-market or user experience perspective [8]. But as climate change rises up the global economic and political agenda, more and more (mobile) developers are also concerned about the sustainability of the software they create. It is therefore useful to compare development practices from an environmental perspective until the decarbonization of software becomes mainstream practice.

Unfortunately, the everyday mobile developers often finds himself alone when facing this challenge. In [11], a survey of experienced developers showed that they are genuinely interested in the energy consumption of software, despite the fact that little knowledge is available. Authors of [18] pinpointed the energy-related questions posed in Stack Overflow by mobile developers, anxious to learn about power-related problems that are encountered by others.

From the trenches, at the implementation stage, eco-friendly mobile developers may refer to catalogues of code smells inherited from embedded systems [12] or mobile-specific green patterns [2]. More recently, they may use a lint-like tool to automatically clean their codebase of energy code smells [4], [7]. Before that, the choice of programming language itself can have a small impact on energy consumption [5] in particular contexts. But in the case of Android for example, this choice is obviously limited, and it has been shown that migrating from Java to Kotlin has no significant impact on the energy efficiency of the app [1]. However, an even earlier choice that the development team has to make (and therefore the hardest to change later) is the choice between native and cross-platform development methods. Therefore, this paper investigates whether this key design decision will have an ecological impact once the mobile application is deployed on a potentially large number of devices.

To this end, we have formulated the following 3 research questions:

- **RQ1**: *Does the development method affect the size of the application archive file?*
- **RQ2**: *Does the development method affect the amount of data the application exchanges over the network?*
- **RQ3**: *Does the development method influence the energy consumption of the app?*

By answering RQ1, we are fighting the "fatware" syndrome, i.e. the inflation of software size over the last decades (e.g. TikTok on iOS is now 400Mb!), which marginalises owners of low-end devices. In fact, the number of bytes downloaded to install the application and its subsequent updates is constantly increasing. This is particularly salient on mobile platforms, where updates are frequent and automatic, regardless of new differential download techniques. Answering RQ2 leads to pinpoint how much the client side, network infrastructure and server side are stressed over the Internet. Indeed, it is not unreasonable to assume that the more data that is exchanged and processed, the more energy is likely to be consumed on a global scale. Last but not least, by answering RQ3, we are looking at the battery drain. The direct effect of this is an increased demand for electricity (and its source production[1]) to charge the handheld device. The indirect effect is to shorten the life of the device, since its lithium-ion battery has a limited number of charge/discharge cycles. It is worth recalling that the manufacture of new user device remains the main source of greenhouse gas emissions in the ICT sector [6].

[1]Depending on the country, electricity production can be more or less decarbonised. See https://app.electricitymaps.com/

2151-0849/23/$31.00 ©2023 IEEE
DOI 10.1109/ASEW60602.2023.00005

The paper is structured as follows: Section II gives an overview of modern cross-platform frameworks. Section III describes the development principles of our validation application. The experimental results are presented in IV and discussed in section V. Related work is mentioned in section VI before we conclude in section VII.

II. Background

Targeting both the iOS and Android platforms is crucial to reach the largest customer market. Android development has historically been done natively in Java, and more recently in Kotlin, to which more and more developers are migrating [3]. Native development on iOS was done in Objective-C before it was gradually replaced by the Swift language. These native solutions are the ones officially recommended by Google and Apple for mobile app development and have become the *de facto* standard. However, it is an expensive and time-consuming task, as development has to be done twice. As a result, competitive cross-platform frameworks have emerged since 2015. Cross-platform solutions allow a single codebase to be used to build an app for both Android and iOS. These include React Native, Flutter and Kotlin Multiplatform Mobile, three very popular frameworks with three different approaches to implementing cross-platform and code-sharing. Other cross-platform solutions exist (Ionic, MAUI) but are less common.

A. React Native

React Native is an open source cross-platform framework initiated by Facebook and based on the ReactJS framework. It has been around since 2015 and is now the most popular cross-platform framework. Unlike native code, React Native runs on JavaScript code using a JavaScript engine (JavaScriptCore or Hermes), but uses the native platform components – through a binding mechanism – to render the UI, so that the final applications look like native applications.

B. Flutter

Flutter is an open source, cross-platform framework backed by Google. Its first stable release was released in 2018. It uses Dart (also supported by Google) as its programming language. Dart is a garbage collected language that can be compiled JIT (Just In Time, using the Dart VM) or AOT (Ahead Of Time, for better performance). Flutter uses its own rendering engine to manage the UI and does not rely on the native platform components at all (unlike React Native). However, it still provides a plugin system to access (non-UI) native platform features.

C. Kotlin Multiplatform Mobile

Kotlin Multiplatform Mobile (KMM) is more of a code-sharing framework than a complete cross-platform framework. In fact, its goal is to create portable libraries for iOS and Android native applications using a single code base written in the Kotlin language. Therefore, KMM always requires a native component to run. When running on Android, KMM runs on top of the Java VM, just like any native Android application. When running on iOS, KMM is compiled into native code (AOT compilation) using Kotlin Native, a native Kotlin compiler.

III. Methodology

The aim of this paper is to provide a fair comparison of the different development approaches available to developers. To do this, we followed the 5 development approaches (2 native + 3 cross-platform), resulting in 8 variants of a single app. This starting work represents several man-months, and for the sake of reproducible research, the entire codebase is open[2].

A. Baseline

From an end-user perspective, the 8 variants should look identical: same features, same user journey, same content. This also means identical technical decisions under the hood: same assets (format and image resolution) and same interactions with the server side (API calls and triggering events). However, it cannot be denied that each development approach has its own specificities. In order to remove these differences as much as possible, we have imposed the following design principles:

- No advanced architecture: the application is made up of basic screens, and each network call is made every time a screen is displayed (i.e. no prefetching or advanced state management),
- No fine-tuning: network and UI components of the platform are used as is: network cache, render cache, etc. are used with their default settings,
- No third-party libraries: the code must rely only on the platform's built-in components or, to a lesser extent, on external libraries officially supported by the editor.

It is worth noting that the development of the different versions of the app was done in parallel to avoid disruption to the practice at the time. In particular, the iOS native development was done using SwiftUI and the Android native development was done using Jetpack Compose.

B. Validation App

The mobile application created especially for this study is a (turnip) movie database application with master/detail navigation, whose data comes from The Movie Database API[3]. In our opinion, this application is fairly representative of a mobile application in app stores today (with the exception of video games). This high quality standard app uses modern-day UI elements and both light and dark modes have been implemented.

As shown in Figure 1, the app consists of three screen types. The first type of screen is the home/category, which consists of 2 sections:

- A top section displaying a "trending carousel" that makes an API call at each screen appearance to retrieve its content,
- A bottom section displaying a list of 4 category carousels. Carousel content is retrieved with one API call per

[2]https://github.com/orgs/TurnipOffApp/repositories
[3]https://developers.themoviedb.org/

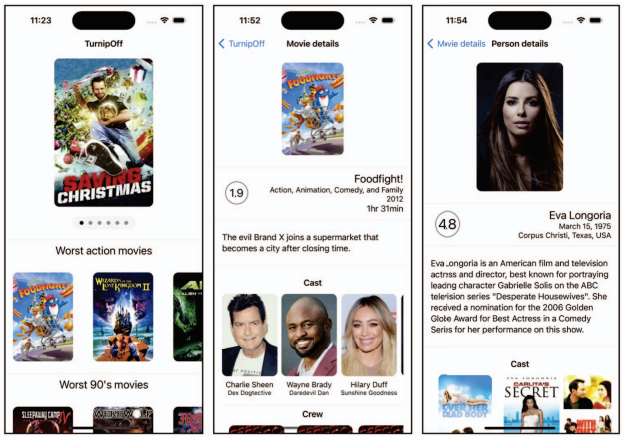

Fig. 1. The 3 app screens (iOS, light mode)

carousel (made on each screen appearance). Category carousels use infinite scrolling with a page size of 20 items, making a new API call to load the next page.

The second type of screen is the film details, which consists of 2 sections:

- A header containing the poster image, film title and other film related information. It is loaded with an API call triggered on screen appearance,
- A credits section that displays 2 carousels (cast and crew) without infinite scrolling / lazy loading. Content for these 2 carousels is retrieved in a single API call (performed on each screen appearance).

The third type of screen is the actor details, which is made up of 2 sections:

- A header containing the actor's image, name and other actor-related information. It is loaded with an API call triggered on screen appearance,
- A credits section that displays 2 carousels (cast and crew) without infinite scrolling / lazy loading. Content for these 2 carousels is retrieved in a single API call (performed on each screen appearance).

IV. Evaluation

Apart from the weight of the application, which is easy to obtain, measuring the energy and data consumed at runtime is cumbersome, especially on iOS. That's why we used a premium solution called "App Scan"[4], developed by the company Greenspector, which remains a reference in the field.

A. Experimental Setup

To conduct our experiment, GreenSpector has provided us with its bench of physical handled devices (see Figure 2), consisting of Samsung S7 (Android 8), Samsung S9 (Android 9 and 10), Pixel C tablet (Android 8) and Apple iPhone 8 (iOS 13).

[4]https://greenspector.com/en/evaluate-app-scan/

The probes are based on software-based sensors embedded in Android phones. Calibration work makes it possible to qualify the quality of the measurement. In fact, some information provided by the APIs or the manufacturer cannot be used. A quality score is given between 0 and 10. A score close to 10 indicates a high measurement frequency and precision. A score below 5 indicates that the measurement is not accurate enough to be used. So only phones with a score above 8 were included in the test. For iOS, since energy data is not provided by the platform, the solution is hardware-based, with a wattmeter placed between the battery and the iPhone.

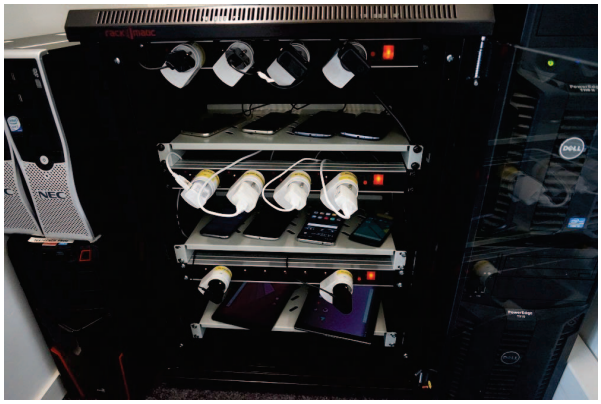

Fig. 2. Picture of the test bench with real devices

B. Experimental Scenario

The end user scenario for such a Master-Detail styled application is straightforward: *Browse the home page* > *Click on a movie* > *Browse the movie detail* > *Click on an actor* > *Browse the actor detail*. Although the user journey is very commonplace, it must be possible to play it without a human in the loop in order to obtain the least biased comparison possible. This is allowed by the Greenspector Domain-Specific Language (GDSL) in which test scenarios can be written once, and run several times in both mobile platforms. Contrary to frameworks like Appium [15] which are based on frequent network exchanges to execute the scripts, the GDSL framework is designed to have a minimal energy and data overhead.

The GDSL is a series of actions that are performed in sequence on the device. It includes basic actions such as *wait*, *click* or *pause*, as well as more complex actions such as launching an application or managing the GPS. The GDSL language allows measurement points to be placed in the scenario, which allows a fine granularity of the metrics. An excerpt of the GDSL script written for the experiment is shown below:

```
pressHome
measureStart
applicationStart;fr.insideapp.turnipoff
waitUntilText;Worst action movies
pause;1000
```

```
measureStop;LOADING_launch
measureStart
pause;30000
measureStop;PAUSE_home
measureStart
swipe;355;500;20;500
...
```

During the automated execution of the tests, the devices are placed in a situation that allows for the most stable measurements. For example, applications are uninstalled at each iteration. The user scenario was played 10 times – an acceptable accuracy/duration trade-off – for each version of the app. We then took the average of the results. Only the light mode of the app was tested.

C. Experimental Results

Table I shows the metrics extracted from Greenspector's App Scan dashboard, namely app size (in kilobytes), network data exchange (in kilobytes) and energy consumption (in milli-ampere-hours), if applicable.

1) App Size: Each mobile operating system uses its own file format to distribute and install application software: IPA (iOS App Store Package) and APK (Android Application Package). Such a file contains all the bytecode, of course, but also all the static resources needed to run the application (icons, String literals, misc. properties). The weight of this file characterises the volume of data exchanged by the stores (installation, then updates), as well as the space occupied on the user's device. Techniques exist to reduce the size of the application for both operating systems [9], [10].

2) Network Traffic: The amount of data initiated by user activity at run-time has a strong impact on the energy consumption of the radio and the global network equipment chain. In other words, this is about the greenhouse gas emissions to power the Internet. Opportunities for energy-savings of HTTP requests have been investigated in [14]. A technical limitation of App Scan has forced us to ignore the computed values for iOS, as they are system-wide and not application-specific.

3) Energy Consumption: Essentially, an application's energy consumption is a complex combination of data processing, network interactions and UI rendering, all with power-hungry hardware components behind the scenes. For interested readers, a per-component stress test has been performed by Malavolta et al. in [13] on Android devices. The more these components are stimulated, the higher the power consumption and the faster the battery drains. Unfortunately, an unknown technical error prevented the value for KMM on iOS from being known.

V. Discussions

In this section, an attempt is made to provide reasons for the observed differences in the results with a view to answering the 3 research questions. Some limitations are also noted.

A. Analysis

1) App Size: Native development in Swift or Kotlin undeniably produces smaller applications than crossplatform solutions. These are the languages officially recommended by the mobile platforms and therefore take full advantage of them. Notice the lightweight of the Swift version compared to its Kotlin counterpart, which is much heavier because it has to cope with a variety of hardware configurations and manufacturers.

On their side, cross-platform solutions generate larger packages. As KMM only deals with the business logic and relies on native UI rendering, it offers a significant gain in size for both platforms. Flutter and React are far behind but with a big difference: Flutter generates comparable occupancy on both OS, while for React it is double on Android compared to iOS. We believe that this situation would be even worse if the validation application had embedded third party libraries. Indeed, the React Native or Flutter SDKs are usually built upon a native SDK with a Flutter/React Native wrapper to interface with it. Adding external SDKs will then widen the gap between native and cross-platform applications.

It should be noted that a difference of 15 to 20 MB between the average native package and those of Flutter or React Native will generate 7 to 10 MB more network traffic per installation. Indeed, app downloads are optimised/compressed by the store, so the actual amount of data transmitted over the network is half the total app package size. But for a mass-market app that generates around 10 million installs per year, this translates into 70 to 100 TB of additional network traffic and carbon emissions thereof.

> **RQ1 –** *Native development is by far the best solution for both iOS and Android in terms of app size package criteria. The gap between native and cross-platform development will widen as app complexity increases.*

2) Network Traffic: The results show quite significant differences between the development approaches, with React Native being by far the best performing. As stated in our baseline, we didn't try to optimise the number of requests and the amount of data, leaving that to the frameworks. The truth is that React Native naturally optimises traffic, perhaps gaining its strength from the huge and active community around ReactJS/React Native.

On the Android side, there is a wide gap of 69% of data exchanged between the least data-intensive solution (React native) and the most data-intensive (Flutter), with KMM and Kotlin equally far from these extremes. At this point, it is difficult to explain why Flutter underperforms so much compared to the other 3 solutions. But it seems wise to say that applications based on remote content (e.g. social networking) should flee Flutter. If a few hundred kilobytes can make a difference in a very simple master-detail scenario, imagine the volume at stake for behemoths apps like TikTok or Twitter. Here we touch on the underlying reasons for

Development method	App size (kB)		Net traffic (kB)		Energy (mAh)	
	Android	iOS	Android	iOS	Android	iOS
Swift	N/A	216	N/A	—	N/A	8,59
Kotlin	1200	N/A	944	N/A	21,60	N/A
KMM	3600	1600	932	—	21,66	—
Flutter	17500	18000	1190	—	18,27	9,19
React Native	27300	13100	706	—	19,45	11,62

TABLE I
COMPARISON OF THE ECOLOGICAL IMPACT OF THE 5 DEVELOPMENT METHODS (PARTIAL)

the continued growth of the global network infrastructure to meet the demands of software applications. Without a deep understanding of cross-platform solutions, developers must remain vigilant on this aspect.

RQ2 – *React Native is the most data-saving solution over the network on Android. This development approach really widens the gap with its competitors, especially Flutter.*

3) Energy Consumption: In terms of power consumption, the app acts like a black box, draining the battery over time. That's pretty much all the end user cares about, and they're sure to complain if an app drains their battery abnormally fast. For our part, we have no way of knowing whether this or that element is involved. Broadly, it is difficult for developers to determine the root causes of energy hotspots. Native developers may rely on energy profilers within their IDE (namely, Android Studio and Xcode), but nothing is available for cross-platform developers.

Here the results between iOS and Android are different and quite difficult to analyse. Flutter has incredibly good results on both platforms, helped by its modern rendering engine, even though it generates more data exchanges than the other solutions (see previous section). This tends to prove that there is not such a direct relationship between these two metrics. Kotlin and KMM are close together, probably because KMM uses the native UI developed for Kotlin. React Native performs well, although we would expect it to be the most energy consuming solution. However, React Native generates 25% less traffic than KMM and native development, which probably compensates for its inherently power-hungry UI rendering. The results are quite different on iOS, where native development in Swift remains slightly more efficient than Flutter, and React Native comes in last. The difference between the best and worst performers is less than 10%, much less than on Android.

RQ3 – *In terms of energy consumption, Flutter and React Native perform best than native development on Android, while native prevails on iOS, heeled by Flutter.*

B. Limitations

A single tested application is clearly not enough to provide statistical evidence of the environmental impact of development methods. It is worth noting that this would require a colossal amount of upstream programming. Nevertheless, the results already give an idea of the orders of magnitude involved for a contemporary mobile application. As a side note, this highlights the lack of a collection of apps coded using the different existing methods to foster research on this topic, as did AndroZoo [22] for closed-source Android app analysis.

Measurement is a difficult art, and this paper is no exception. The physical test bench cannot reflect the diversity of devices and OS versions into the wild, and our well-controlled test environment (stable WiFi, battery charge level between 20% and 80%, etc.) is not the real life. Also, our neutral baseline (see III-A), while useful to ensure a fair comparison, can be criticised when it comes to actual development aimed at the mass market. Finally, mobile development frameworks are evolving at a rapid pace, bringing with them a host of optimisations, so the metrics calculated are no longer valid at the time you read this document.

VI. RELATED WORKS

There is virtually no work in the literature that looks at the ecological impact of mobile development methods beyond the energy aspect. However, we can cite [21] where the authors state that React Native consumes between 6% and 8% more power than its native Android counterpart, which contradicts our own findings. Generally speaking, if we assume that performance is directly related to power, then some of the existing research can be taken into account. To a lesser extent, what happens with mobile-friendly web apps or hybrid apps can also be taken into account.

We can cite the work of Biørn-Hansen et al. [19] on the performance overhead of cross-platform versus native for Android only. The authors found that cross-platform frameworks for mobile application development can lead to lower performance compared to the native development approach. However, the results also show that certain cross-platform frameworks can perform as well as or better than native on some of the five metrics they observed (CPU usage, PreRAM, RAM, ComputedRAM and Time-to-Completion), but no framework scored best across all of the features in their study. At this

point, we argue that the link between run-time efficiency and energy/carbon efficiency is an open research question.

On Android, some work has looked directly at the energy implications of the choices made by developers. The authors of [17] analysed the impact of different languages (C/C++/Java) and compiler optimisations on energy consumption, as well as the impact of different Android system runtimes (ART vs. Dalvik) on energy efficiency. The main finding was that ART, the successor to Dalvik, has greatly improved the energy efficiency of Java code, making it comparable to NDK programming. In [16], the authors focus on the different development approaches to deliver an app for the Android platform. They compare regular Java development with C/C++ development throughout Native Development Kit (NDK) and hybrid app (web-based with JavaScript) from the energy consumption perspective. Although their results were not uniform, they found that JavaScript was more energy efficient in 75% of all benchmarks, and that Java was the big loser. However, this result is contradicted by a very recent study [20] in which native Android apps consume significantly less energy than their Web counterparts, with large effect size.

VII. Conclusion

Despite the partial results obtained, this first-of-its-kind study can provide useful insights for software researchers and developers interested in the low-carbon transition issue. Understanding the impact of a primer development decision on application size, network traffic and power consumption is a key step. And this is available for both mobile platforms, although iOS is largely underrepresented in the scientific literature. Of course, this preliminary work is intended to be completed as iOS-related technical hurdles are overcome and development frameworks evolve.

Already as it is, certain preconceptions are being challenged as cross-platform solutions have improved significantly over the years. Native development is probably still the best in terms of performance (i.e. run-time efficiency), but not necessarily in terms of carbon efficiency. As usual, it is about making trade-offs between a number of quality attributes. But it's a safe bet that a low ecological impact will be promoted as a general best practice in the years to come.

References

[1] M. Peters, G. Scoccia and I. Malavolta, (2021) "How does Migrating to Kotlin Impact the Run-time Efficiency of Android Apps?," in 2021 IEEE 21st International Working Conference on Source Code Analysis and Manipulation (SCAM), Luxembourg, pp. 36-46.

[2] Cruz, L., Abreu, R. (2019) "Catalog of energy patterns for mobile applications". Empirical Software Engineering 24, 2209–2235.

[3] M. Martinez and B. Gois Mateus, (2021) "Why did developers migrate Android Applications from Java to Kotlin?," in IEEE Transactions on Software Engineering.

[4] O. Le Goaer and J. Hertout, (2022) "ecoCode: a SonarQube Plugin to Remove Energy Smells from Android Projects", The 37th IEEE/ACM International Conference on Automated Software Engineering (ASE 2022).

[5] R. Pereira et al., (2021) "Ranking programming languages by energy efficiency," Science of Computer Programming, Volume 205.

[6] C. Freitag et al., (2021) "The real climate and transformative impact of ICT: A critique of estimates, trends, and regulations," Patterns, Volume 2, Issue 9.

[7] A. Ribeiro, J. F. Ferreira and A. Mendes, (2021). "EcoAndroid: An Android Studio Plugin for Developing Energy-Efficient Java Mobile Applications," 2021 IEEE 21st International Conference on Software Quality, Reliability and Security (QRS), pp. 62-69.

[8] Andreas Biørn-Hansen, Tor-Morten Grønli, Gheorghita Ghinea, Sahel Alouneh, (2019) "An Empirical Study of Cross-Platform Mobile Development in Industry", Wireless Communications and Mobile Computing, vol. 2019, Article ID 5743892, 12 pages.

[9] R. Kumar, (2021) "Android App Size Reduction: Analysis and different methodology," 2021 Fourth International Conference on Electrical, Computer and Communication Technologies (ICECCT), pp. 1-5.

[10] M. Chabbi, J. Lin and R. Barik, (2021) "An Experience with Code-Size Optimization for Production iOS Mobile Applications," 2021 IEEE/ACM International Symposium on Code Generation and Optimization (CGO), pp. 363-377.

[11] Ournani, Zakaria et al. (2020) "On Reducing the Energy Consumption of Software: From Hurdles to Requirements." Proceedings of the 14th ACM / IEEE International Symposium on Empirical Software Engineering and Measurement (ESEM).

[12] Antonio Vetro et al., (2013). "Definition, implementation and validation of energy code smells: an exploratory study on an embedded system." 34-39.

[13] I. Malavolta et al., (2020) "A Framework for the Automatic Execution of Measurement-based Experiments on Android Devices," 35th IEEE/ACM International Conference on Automated Software Engineering Workshops (ASEW), Melbourne, VIC, Australia, 2020, pp. 61-66.

[14] Ding Li and William G. J. Halfond. (2014). "An investigation into energy-saving programming practices for Android smartphone app development". In Proceedings of the 3rd International Workshop on Green and Sustainable Software (GREENS 2014). Association for Computing Machinery, New York, NY, USA, 46–53.

[15] Singh, S., Gadgil, R., Chudgor, A. (2014). "Automated testing of mobile applications using scripting technique: A study on appium". International Journal of Current Engineering and Technology (IJCET), 4(5), 3627-3630.

[16] Wellington Oliveira, Renato Oliveira, and Fernando Castor (2017). "A study on the energy consumption of Android app development approaches." In Proceedings of the 14th International Conference on Mining Software Repositories (MSR '17)". IEEE Press, 42–52.

[17] Chen, X., Zong, Z. (2016). "Android App Energy Efficiency: The Impact of Language, Runtime, Compiler, and Implementation". (2016) IEEE International Conferences on Big Data and Cloud Computing (BDCloud), Social Computing and Networking (SocialCom), Sustainable Computing and Communications (SustainCom), 485-492.

[18] H. Malik, P. Zhao and M. Godfrey, (2015) "Going Green: An Exploratory Analysis of Energy-Related Questions," IEEE/ACM 12th Working Conference on Mining Software Repositories, Florence, Italy, 2015, pp. 418-421.

[19] Biørn-Hansen, A., Rieger, C., Grønli, TM. et al. (2020) "An empirical investigation of performance overhead in cross-platform mobile development frameworks." Empirical Software Engineering, Volume 25, pp. 2997–3040.

[20] R. Horn et al., "Native vs Web Apps: Comparing the Energy Consumption and Performance of Android Apps and their Web Counterparts", 2023 IEEE/ACM 10th International Conference on Mobile Software Engineering and Systems (MOBILESoft), Melbourne, Australia, 2023, pp. 44-54.

[21] Thomas Dorfer, Lukas Demetz, Stefan Huber, (2020) "Impact of mobile cross-platform development on CPU, memory and battery of mobile devices when using common mobile app features", Procedia Computer Science, Volume 175, pp. 189-196.

[22] K. Allix, T. F. Bissyandé, J. Klein and Y. L. Traon, "AndroZoo: Collecting Millions of Android Apps for the Research Community," 2016 IEEE/ACM 13th Working Conference on Mining Software Repositories (MSR), Austin, TX, USA, 2016, pp. 468-471.