Wrocław University of Science and Technology
Faculty of Information and Communication Technology

Field of study: **IST**
Speciality: **CE**

MASTER THESIS

Comparative analysis of selected cross-platform frameworks in relation to native development approaches

Maciej Sroczek

Supervisor
dr inż. Dariusz Konieczny

Keywords: mobile app performance, native, cross-platform, Kotlin, Swift, Flutter, React Native

WROCŁAW 2023

ABSTRACT

There are various aspects affecting the overall perception of quality of a mobile application, with performance being one of the most significant, especially from the perspective of the user. Having that in mind, it is crucial to understand the differences between the available mobile development approaches and in which use cases they are able to provide the highest value.

The purpose of this master's thesis was to perform a comparative analysis of the performance of mobile applications built using both native and cross-platform solutions. Exemplary applications were implemented with Kotlin, Swift, Flutter, and React Native, to be used as the environment for the experiments. The experiments provided results considering the selected performance metrics, e.g., CPU, memory, and power usage. The results were interpreted in order to find benefits and/or weaknesses for each studied solution, as well as to try to define optimal scenarios for their use.

STRESZCZENIE

Na ogólne postrzeganie jakości aplikacji mobilnej wpływają różne aspekty, przy czym wydajność jest jednym z najistotniejszych, zwłaszcza z perspektywy użytkownika. Mając to na uwadze, kluczowe jest zrozumienie różnic pomiędzy dostępnymi podejściami do wytwarzania aplikacji mobilnych i w jakich przypadkach użycia są one w stanie zapewnić najwyższą wartość.

Celem niniejszej pracy magisterskiej było przeprowadzenie analizy porównawczej wydajności aplikacji mobilnych zbudowanych z wykorzystaniem rozwiązań natywnych oraz cross-platformowych. Przykładowe aplikacje zostały zaimplementowane przy użyciu Kotlin, Swift, Flutter oraz React Native, aby posłużyć jako środowisko do przeprowadzenia eksperymentów. Eksperymenty dostarczyły wyniki uwzględniające wybrane metryki wydajności, np. zużycie procesora, pamięci i energii. Wyniki zostały zinterpretowane w celu znalezienia korzyści i/lub słabości dla każdego badanego rozwiązania, a także próby zdefiniowania optymalnych scenariuszy ich wykorzystania.

CONTENTS

1. Introduction . . . 3

1.1. The purpose of the thesis . . . 4
1.2. The scope of the thesis . . . 4
1.3. The structure of the thesis . . . 4

2. Related work . . . 5

3. Mobile development approaches . . . 8

3.1. Native mobile development . . . 8
3.1.1. Android . . . 10
3.1.2. iOS . . . 13
3.1.3. Web development in relation to native mobile development . . . 15
3.2. Cross-platform mobile development . . . 15
3.2.1. Cross-platform development approaches . . . 17
3.2.2. Flutter . . . 20
3.2.3. React Native . . . 23
3.2.4. Comparison . . . 27
3.2.5. Evaluation of cross-platform frameworks . . . 27

4. Mobile application performance measurement . . . 29

5. Research method . . . 32

5.1. Performance metrics . . . 32
5.2. Research scenarios . . . 32
5.3. Testing tool . . . 34
5.4. Testing devices . . . 34

6. Implementation of sample applications . . . 35

6.1. Research scenario 1: List scrolling and filtering . . . 35
6.2. Research scenario 2: Animations . . . 36
6.3. Research scenario 3: File I/O . . . 37
6.4. Research scenario 4: Common UI elements . . . 38
6.5. Key takeaways obtained during the implementation process . . . 40

7. Experiment results . . . 41

7.1. Research scenario 1: List scrolling and filtering . . . 42
7.2. Research scenario 2: Animations . . . 44
7.3. Research scenario 3: File I/O . . . 46

7.4. Research scenario 4: Common UI elements 47

8. Research discussion 49

8.1. Experiment results analysis 49

8.1.1. Generalized results analysis 49

8.1.2. Research scenario 1 results analysis 53

8.1.3. Research scenario 2 results analysis 54

8.1.4. Research scenario 3 results analysis 56

8.1.5. Research scenario 4 results analysis 57

8.2. Conclusions 58

9. Summary 60

9.1. Contributions 60

9.2. Limitations 61

9.3. Suggestions for future work 61

Bibliography 63

List of Figures 69

List of Tables 72

1. INTRODUCTION

Over the last few years, mobile devices, such as smartphones, tablets, or even smartwatches, have been acknowledged as a rather essential part of human lives. This is confirmed by the big and still increasing number of over 7 billion mobile users across the world [84]. Because nearly 90 percent of users spend their time using different apps, the number of mobile app downloads is very high, at over 200 billion in 2020, which has a direct impact on the expansion of the mobile app market [85]. According to the report from last year, the worldwide mobile application market was valued at over $206 billion. Considering its rapid growth, it is estimated to reach $565 billion in 2030 [38]. Such high demand impels mobile developers to constantly seek more innovative solutions as well as improve on existing ones. Nowadays, there are various aspects considered to be of high priority, the main ones being privacy and security as well as usability and user experience. Those factors, combined with the growth of the mentioned market, resulted in the evolution of different implementation methods for mobile development, with native and cross-platform being the most widely used.

Native mobile development implies creating software that can only be run on a specific platform (operating system), such as Android or iOS [21]. In order to do so, platform-specific tools must be utilized. In the case of Android, the programming language Kotlin may be used, and in the case of iOS, Swift. While it can be seen as a limitation, it provides some advantages, such as being able to use different elements of the system directly and, with that, maximize the achievable performance.

Cross-platform mobile development aims to eliminate the need to implement multiple versions of the same mobile app in order to make it available for users of different platforms. This method assumes the use of a single codebase that enables building the app for various operating systems. From the perspective of a user, each of them could perform and look as if they were implemented natively [49]. Such an approach quickly became popular among developers, including successful companies such as Meta and Google [47]. Some examples of cross-platform frameworks are Flutter, and React Native.

All of the differences between the above-mentioned implementation approaches can make them more or less applicable in various scenarios. The selection of either native or cross-platform development method as well as the specific technology is really important because it may directly affect aspects such as development time, cost, and overall end-product quality. However, most of the popular solutions are constantly being updated, which leads to the necessity of recurrent comparative analysis in order to obtain the most

up-to-date state of the art. Such knowledge will then be helpful to determine in which cases different development approaches and tools should be optimally used.

1.1. THE PURPOSE OF THE THESIS

The purpose of this master's thesis is to carry out research on the performance of mobile applications implemented with selected cross-platform frameworks in comparison to each other and to native development methods. A number of metrics will be selected for analysis based on a literature review and personal experience. Exemplary applications will be prepared as an environment for the experiments. The results will form the basis for defining the advantages and downsides of developing single codebase cross-platform applications. Furthermore, optimal development methods will be proposed for different types of mobile applications.

1.2. THE SCOPE OF THE THESIS

To begin with, a problem analysis will be performed, which will result in defining the specifications for the experiments to be carried out. Conducted experiments will provide data for further analysis, which will be organized into groups based on the experiment environments, studied platforms, and frameworks. The results will be interpreted in the context of quality and possible optimal use-cases for implementing mobile applications using the selected frameworks and native methods. All of the research must be documented.

1.3. THE STRUCTURE OF THE THESIS

The thesis has been divided into nine chapters. The first chapter aims to provide a brief introduction to the topic. The second chapter presents the selection of relevant related work. The third and fourth chapters purpose is to provide the knowledge necessary for the further work based on the literature. In the fifth chapter, the research method is defined, mostly based on the literature review. The sixth chapter concerns the implementation of testing environments and the realization of prepared experiments. In the seventh chapter, the results from performed experiments are visualized and described. The eighth chapter contains the discussion that emerged from the experiment results and the conclusions drawn. Finally, in the last chapter, the complete work is summarized, and key takeaways are featured. Additionally, limitations are explained, and suggestions for future work are proposed. The dissertation closes with a bibliography as well as lists of figures and tables.

2. RELATED WORK

Table 2.1 contains the selection of literature directly related to the topic of this thesis. Analysis of the related work should provide additional insights into the scope of carried out research, and potentially reveal obsolescence or shortfalls present in the previous research.

Table 2.1: Related work (Source: Own work)

Paper	**Key takeaways**
[61]	M. Ollson compares a mobile app developed using Kotlin, Swift and Flutter. The measures considered are: code size, development time and CPU usage. Additionally, a survey is performed among 39 programmers to assess the native look and feel of the Flutter application. (2020)
[40]	E. Hjort performs an evaluation of two cross-platform frameworks: Flutter and React Native aiming to select the ideal solution for a selected company. Based on that company's requirements, the technical and business criteria are selected with different weights mapped to them. (2020)
[24]	A. E. Fentaw compares a COVID-19 tracking application implemented with both Flutter and React Native. The criterion considered is the app performance in the form of CPU, GPU and memory consumption. (2020)
[22]	J. Crha performs a comparative analysis of different approaches to cross-platform development: cross-compiled frameworks and Responsive Web Applications. Flutter and React Native are evaluated based on app performance, primarily startup time, CPU and memory usage, frames per second (FPS) and app bundle size. (2021)
[23]	B. Denko, S. Pecnik and I. Fister Jr. perform a four-phase evaluation of different multi-platform solutions: Ionic, React Native, NativeScript and Flutter compared to native approach. A single target platform is considered (Android). The main criteria considered are app size and installation time, CPU and RAM usage and algorithm execution time. (2021)

Continued on next page

Table 2.1: Related work (Source: Own work) (Continued)

[78]	M. Singh and G. Shobha compare multiple solutions to cross-platform development: React Native, Ionic, Flutter and Xamarin. The criteria considered are exclusively coming from the development perspective, e.g. IDEs, platform support, code reusability and testing process. (2021)
[16]	D. Białkowski and J. Smołka perform a comparison of time performance of Android apps implemented natively (Java) and with a cross-platform framework (Flutter). There are three research scenarios proposed applied per each application. (2022)
[44]	M. Kocki, M. Urban and P. Kopniak compare the native and cross-platform approaches to iOS development in the form of Swift and Ionic. The research focuses on compilation time, database read and write time and data sorting execution time. (2022)
[58]	D. Mota and R. Martinho introduce a stable approach to performance assessment of mobile apps developed with cross-platform frameworks. They propose a set of metrics (CPU usage, RAM usage, execution time, FPS), as well as evaluation features and result normalization method. Thereafter, the proposed solution is applied to compare Flutter and React Native. (2021)
[90]	M. Willocx, J. Vossaert and V. Naessens propose an approach to mobile app performance assessment on the example of Xamarin and PhoneGap. Based on the results, some guidelines for framework selection are suggested. The performance metrics selected are: launch time, pause and resume time, time to open page, memory consumption, CPU usage and disk space. (2015)
[41]	M. Hort, M. Kechagia, F. Sarro and M. Harman describe a variety of mobile application performance optimization techniques inspired by the literature from 2008-2020. The metrics considered are responsiveness, launch time, memory usage and energy consumption. (2022)
[18]	A. Biørn-Hansen and T-M. Grønli and G. Ghinea provide a detailed overview of approaches to cross-platform development, as well as correct misconceptions found in other literature. For a group of concepts (User Experience, Software Platform Features, Performance and Hardware, Security) the state-of-research is described and suggestions for future work are proposed. (2018)

Continued on next page

Table 2.1: Related work (Source: Own work) (Continued)

[74]	C. Rieger and T. A. Majchrzak propose a detailed framework consisting of 33 criteria divided into 4 perspectives: Infrastructure, Development, App and Usage. The purpose of such framework is to be used for the complete assessment of a cross-platform solution. In this case, completeness means considering both technical and business aspects. (2019)
[9]	E. Angulo and X. Ferre perform a case study to determine the capability of cross-platform frameworks to provide a high-level user experience and usability compared to native solutions. (2014)

Numerous research papers exist on the topic of mobile application performance among cross-platform frameworks. However, their results tend to differ from each other, especially considering work conducted across a broad timeframe. The underlying issue is that existing cross-platform frameworks are constantly changing in order to improve their usability, as well as the quality of applications developed with them. Moreover, completely new solutions are being introduced. It seems the research in this scope should be carried out on a sporadic basis so that the up-to-date findings can be applied in decision-making.

3. MOBILE DEVELOPMENT APPROACHES

The definition of mobile development can be interpreted in a variety of ways. It can be seen as a broad process of implementing a mobile application, starting with planning and designing and finishing with testing, releasing and maintaining. A more software-oriented definition is that mobile development simply refers to implementing an application for mobile devices by coding it using a selected technology stack [57]. In this thesis, the latter definition is assumed.

Mobile development can become a complex task considering the variety of devices and platforms existing in the market. There are many different approaches available and in order to choose one over another the mobile application requirements should be taken into account as well as target platforms and devices, development and time costs [87].

In this chapter, there are presented selected popular approaches to mobile application development. Each of them is described mainly in the context of architecture, technology stack and tools, platforms supported and possible advantages or disadvantages.

3.1. NATIVE MOBILE DEVELOPMENT

Native mobile development encompasses building mobile applications that can only be implemented using a platform-specific programming language and deployed to a single operating system [88]. Such an approach brings with it the necessity for creating and maintaining multiple codebases and with that possibly multiple development teams [58]. The number of distinct codebases does not simply equal to the number of target platforms, as different versions of a single platform may require to be implemented independently [10]. Hence, development costs are high from the viewpoint of financing and time.

Native mobile applications are closely integrated with the operating system through using target platform's components [61, 78] and most recent features [40]. For that reason, at times they can be referred to as "embedded" [24] and in theory should provide the maximal performance. Furthermore, because native apps are developed according to the operating system's guidelines, such as Material design system for Android and Cupertino for iOS, they are naturally easy to use for users accustomed to that specific platform.

Since almost a decade, the mobile operating system market has been dominated by Android together with iOS, reaching 99,3% in March 2023, as shown in Figure 3.1. For this reason, in the context of this thesis only the above-mentioned operating systems are being taken into consideration.

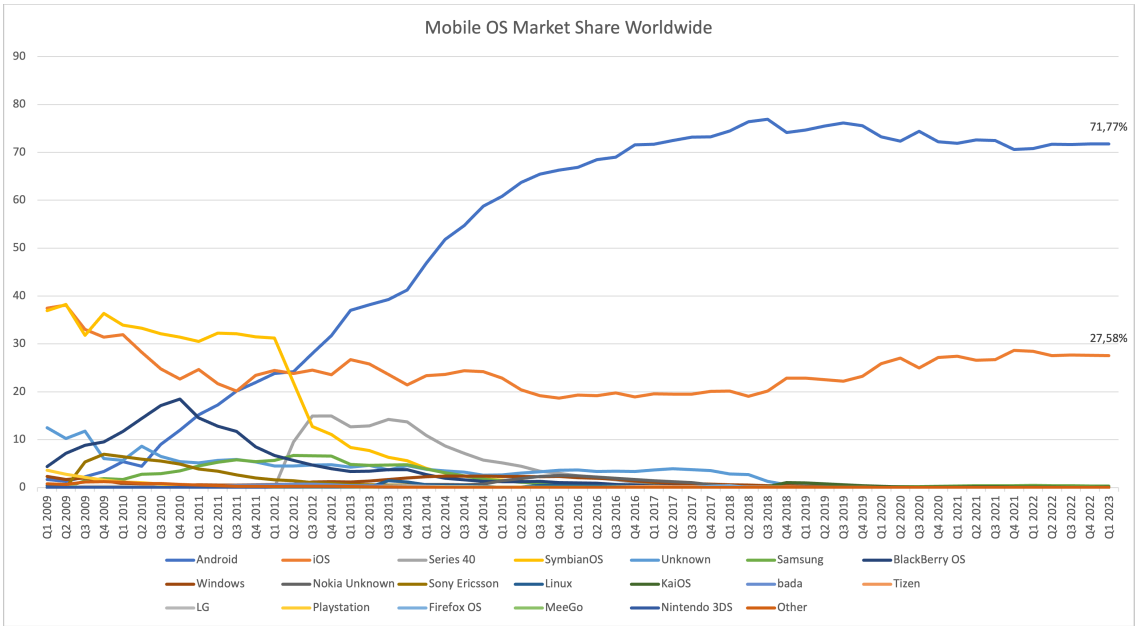

Fig. 3.1. Mobile OS market (Source: Own work based on [81])

As can be seen in the Figure 3.2, in case of iOS, almost 90% of devices are running either the most or second-most recent major version of the operating system. Therefore, when targeting the Apple's system, probably a single codebase would be enough to guarantee the appropriate coverage.

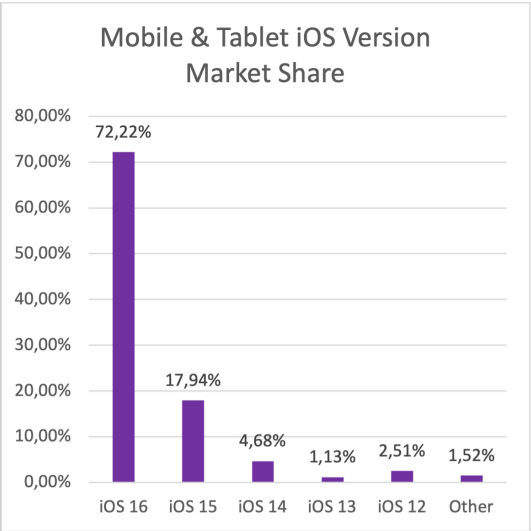

Fig. 3.2. iOS version market share (Source: Own work based on [83])

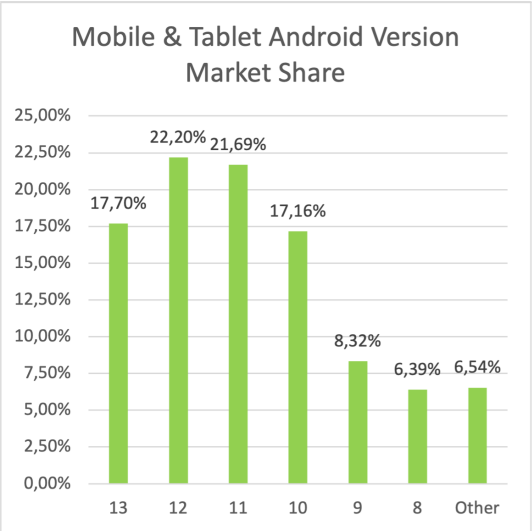

Fig. 3.3. Android version market share (Source: Own work based on [82])

However, in case of Android, there is a high level of market fragmentation, as nearly 20% of smartphones or tablets are running older versions released as far as in 2015 (Figure 3.3). Because there are limitations such as deprecation of code commands and API (Application Programming Interfaces) behavior changes between distant versions, multiple codebases

may be chosen to be maintained separately per a single mobile application. Another issue is the fact, that device manufacturers are able to apply various modifications to the operating system which can lead to errors occurring only on those devices [22], causing difficulties for developers. Because Apple is the exclusive manufacturer of devices running iOS, they do not suffer from such a problem.

3.1.1. Android

Android is an open-source operating system developed by the Open Handset Alliance and Google to run mainly on mobile devices such as smartphones and tablets, but also TVs and cars [2, 22]. It is based on the *Linux Kernel* and has a multiple-component structure [76], as can be seen in the Figure 3.4. Each part takes responsibility for different tasks, e.g., *Android Runtime* provides optimized garbage collection and the *View System* (included in *Java API Framework*) enables developers to implement the user interface layouts using various elements such as lists and buttons [3].

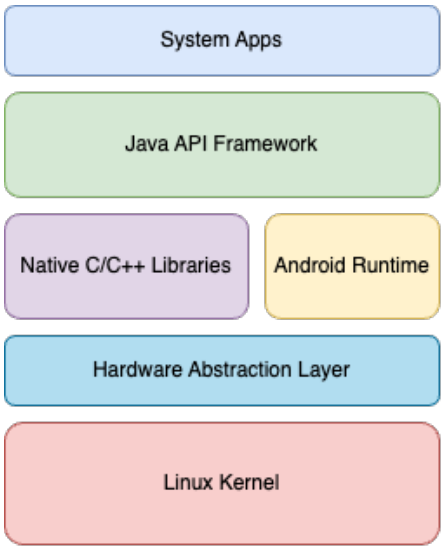

Fig. 3.4. Android architecture (Source: Own work based on [3])

Integrated Development Environment (IDE)

It is not necessary to use an IDE to build most software. Despite that, most programmers tend to reach out to them because of the guaranteed development comfort and productivity increase. Android Studio is the primary IDE for native mobile development of Android applications. It is created on top of IntelliJ's IDEA and provides numerous features such as Gradle, advanced debugging tools and profilers, etc. [4].

Programming languages

For many years, Java has been the official language for Android development. However, since Google established Kotlin as the default choice in 2019 [6], over 60% of programmers

have switched to it [5]. Furthermore, data shows that almost 90% of the Google Play Store Top 500 USA mobile apps have been developed with it [48]. Kotlin's popularity is certainly going to grow in the upcoming years, considering the undeniable benefits it brings. Still, there are some scenarios in which Java could remain the first choice.

Table 3.1 shows that all the features necessary for Android application development, such as e.g., Android SDK or AndroidX support, are fully provided by both Java and Kotlin. Furthermore, the latter introduces additional advantages in the form of enabling the usage of Jetpack Compose toolkit and even the ability to create multi-platform projects (Kotlin Multiplatform is currently only accessible in Beta version [46]).

Table 3.1. Java and Kotlin comparison (Source: Own work based on [6])

Feature	**Java**	**Kotlin**
Platform SDK support	Yes	Yes
Android Studio support	Yes	Yes
Lint	Yes	Yes
Guided docs support	Yes	Yes
API docs support	Yes	Yes
AndroidX support	Yes	Yes
AndroidX Kotlin-specific APIs (KTX, croutines, ...)	N/A	Yes
Online training	Best effort	Yes
Samples	Best effort	Yes
Multi-platform projects	No	Yes
Jetpack Compose	No	Yes
Compiler plugin support	No	Yes (Kotlin Symbol Processing API)

Java is a high-level object-oriented programming language introduced as far back as 1995. It is one of the most popular languages in the world which is regularly updated, reaching major version 20 this year. However, in the context of Android development the supported versions are 8 and 11, with the latter requiring high Android API version in order to use all the offered elements (although upgraded API desugaring announced in February this year broadens the range of libraries available without increasing the app's minimum supported API level [7]). The advantages of Java mainly result from the fact it is present for a very long time. During the last 30 years it gathered a big community of developers with high-level experience. Therefore, it may be easier to form a competent team for the project. Moreover, there are numerous applications that had been created with Java which owners might not seek for a migration, which as a result maintains Java's importance in the market [48].

On the other hand, Kotlin offers many assets because of which it has replaced Java as an official first-choice language for Android development, as mentioned before. Kotlin is a

comparatively recent programming language introduced in 2016 by JetBrains. First and foremost, it is fully interoperable with Java and therefore, it is possible to call Kotlin code inside Java code and vice versa. Secondly, Kotlin's syntax is very concise and null-safe, thus increasing the speed of development and reducing the project's code lines and lowering the possibility of mistakes. For those reasons, implementation of new apps using Kotlin is fairly straight-forward and comfortable from the point of view of developers. Moreover, the migration of existing projects from Java to Kotlin is uncomplicated and can lead to size decrease and simplification of the codebase with much smaller Null Pointer Exception occurrence in runtime [6].

Since Kotlin has already been established as the preferred programming language for Android development, all considerations in the scope of this thesis will be limited to it rather than Java.

User interface development methods

One of the possibilities acquired when selecting Kotlin for Android development is the ability to use Jetpack Compose. It is a powerful toolkit used for building User Interface layouts, introduced in 2021 with a goal to improve that process compared to the previous XML approach [8]. The main difference between the above-mentioned methods is the fact that they represent declarative and imperative approach, respectively. The former greatly reduces the amount of boilerplate code improving readability as well as build time and therefore, increases development efficiency. Additionally, just as Kotlin offers interoperability with Java, Jetpack Compose provides the same in regard to XML. In short, XML approach implies the creation of layouts in XML markup files and later referencing them in the code while implementing the behavior. On the other hand, Jetpack Compose makes use of prebuilt components and intuitive state management [92].

User interface design system

In order to improve the user experience across different apps, Material design system was introduced by Google in 2014. First versions were strictly connected to Android only, however since then it has shifted towards being applicable for other platforms, including Web. Material provides detailed guidelines in the aspect of styling, accessibility, overall UX as well as prebuilt reusable UI components aiming to improve the efficiency of developing apps that are intuitive and responsive. The main principles are that every element of an app should be considered as a physical material, and they should be combined in layers while putting emphasis on natural animations and universal clarity. The big advantage of Material being applied in mobile apps is the fact that when a user understands how to use one, it is automatically transposed onto others. On the other hand, an issue may be raised that if all the mobile apps available conformed to one design system, they would become overly monothematic and characterless [50, 79].

3.1.2. iOS

iOS is the Apple's closed-source operating system based on *Darwin OS* that runs exclusively on Apple's smartphones (iPhones). It also lays beneath other mobile systems: iPadOS, tvOS and watchOS. It includes four layers that together enable the interaction between hardware and applications, as presented in Figure 3.5. *Core OS* layer provides necessary low-level services, e.g. Bluetooth, security and 64-bit support. *Core Services* layer incorporates multiple interfaces called *Frameworks* which are responsible for functionalities such as data management, cloud transfer and location. *Media Layer*'s task is to handle graphics and audio technology. Finally, *Cocoa Touch* enables user-application interaction, mainly through the implementation of touch gestures [59, 60].

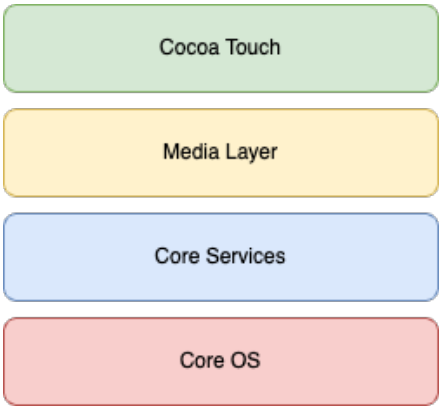

Fig. 3.5. iOS architecture (Source: Own work based on [59])

Integrated Development Environment (IDE)

Xcode is the primary IDE for building iOS, iPadOS, watchOS, tvOS and macOS applications. It offers tools necessary for the whole range of development: implementation, testing, optimization and deployment. It includes Developer Tools, e.g. Simulator for testing, Instruments for performance profiling and Reality Composer for 3D and AR features [11]. Native iOS development comes with additional costs compared to Android because Xcode requires a Mac device, e.g. the MacBook laptop, which itself can be expensive especially when considering a large development team [22, 63].

Programming languages

Currently, the officially recommended programming language for iOS development is Swift. It was introduced with the goal to replace previously used Objective-C. Essentially, it was supposed to increase the ease of development and maintenance of the codebase, make it more error-proof and performant. Objective-C remains supported by Apple for the indefinite future, however since 2011, it has not received any major updates staying at version 2.0 [12, 77, 86, 89].

Objective-C is an object-oriented, dynamically-typed programming language published in 1984. Similarly to Java for Android development, Objective-C is very stable because of its age and is still necessary for legacy applications support because Swift does not allow development of applications for lower than 7.0 version of iOS. Additionally, it allows Swift code usage by automatically regenerating files to enable the integration, which makes it possible to keep the existing code in old projects even when migrating to Swift [33, 34].

Swift is a compiled, statically-typed programming language introduced by Apple in 2014. It holds the features of Objective-C while providing various improvements such as automatic memory management (ARC). Its syntax is much more concise, resulting in readable and easily maintainable code, as well as quicker development time. It was emphasized on release that Swift increases safety and it does so by overflow checking, optionals, type safety, etc. Furthermore, Swift has been shown to run faster than Objective-C. It is also fully interoperable with Objective-C code [12, 33, 34]. However, one of the issues is differences between iOS versions that may force code rewriting. Finally, an important advantage of Swift is that it enables the usage of SwiftUI [22].

User interface development methods

In iOS development, there are multiple ways to implement user interface views. Apple offers two design solutions which can be used together if needed: UIKit and the more recent SwiftUI. The latter has been acknowledged by Apple to be the primary solution [22]. UIKit is an imperative framework that can be applied programmatically (imperatively) or by using Interface Builder to create XIBs (XML Interface Builder) or Storyboards, while SwiftUI is a declarative framework. The difference in programming paradigms results in less code lines necessary for the same task when programming with SwiftUI. Furthermore, SwiftUI apps can be built for all platforms of Apple devices, while UIKit is meant only for iOS, iPadOS and tvOS and requires AppKit and WatchKit to support the rest. On the other hand, SwiftUI supports iOS version 13.0 minimum so if further backwards compatibility is needed than UIKit remains the first choice [13, 20, 52].

User interface design system

In order to provide consistency among different applications and across different Apple platforms, Human Interface Guidelines (HIG) has been introduced. It is a document containing detailed general and platform-specific design principles and practices. Developing applications according to that information helps guarantee that the end product is in accordance with Apple's standard, both visually and behaviorally. Consequently, great user experience is reached as users can easily comprehend applications that are similar and consistent with the platform they are used to, e.g. the positioning of action buttons and touch gestures in iOS apps. Exemplary aspects underlined in HIG are layout creation, navigation, inputs, accessibility, technology support and more [14].

3.1.3. Web development in relation to native mobile development

While solely native mobile development has been considered in this chapter, it is important to reflect upon its connection to web development. Of course, it is a possibility that a service is designed to be available exclusively in the form of a mobile application and such attitude will remain unchanged in the future. However, there are some reasons for which a publisher may look to extend the supported platform list after time. For instance, a small service may start as an Android application and grow enough to need other platforms in order to reach a bigger volume of users. In such a case, iOS support may be added as well as a website may be developed. The second option is especially probable as it provides accessibility for all platforms. The fact that such a service launched in the shape of a single-platform application in the first place may be caused by many aspects, such as dictated functional requirements, detachment or overlook during the initial planning stages as well as market trends analysis [19]. Cross-platform frameworks could be the answer to the above-mentioned issues.

3.2. CROSS-PLATFORM MOBILE DEVELOPMENT

As the term suggests, cross-platform or multi-platform frameworks enable to create applications that can be installed on different platforms, and consequently reach a larger user base. There are many solutions available in the market, which all offer support for a narrow or wide subset of existing platforms. Therefore, applications may be published as, e.g., mobile apps (Android, iOS), web apps, desktop apps (Windows, macOS, Linux) or even embedded apps. The number of operating systems tends to increase under the principle of "write once, run everywhere" followed by frameworks' creators [22].

The main advantage of cross-platform development over native development is in line with its primary goal, which is the ability to create and maintain a single codebase, no matter the number of target platforms. Moreover, as described in the chapter 3.1, Android operating system suffers from high level of fragmentation which can be addressed with cross-platform framework as well. Being able to operate on a single codebase results in the development costs decrease. Implementation is more time-efficient without the need to build multiple projects. This remains true for post-release support in the form of updating versions and handling bugs or change requests. Cross-platform approach is also lighter on resources, requiring just a single development team, which additionally removes possible collaboration issues between teams that can occur in native approach. In the past, it would be assumed it is easier to gather an experienced team for native development, however currently the most popular multi-platform frameworks are mature enough to have created active communities with high level of know-how [58, 61, 78].

There are differences between available cross-platform frameworks, especially in the context of architecture or rendering and compilation method. They will be described in the

following chapters. Essentially, most of them require a middle layer that connects the app with the system and translates the commands to be natively called. This is considered to be a possible root of performance decrease [10]. And although a single workspace is used during development and testing, in order to publish the final application to different app stores (e.g. Google Play Store, App Store), there must be performed a build for each target platform to acquire separate app bundles that can then be uploaded [63].

As explained in the previous chapter, different operating systems usually have different guidelines provided when it comes to user interface (UI) design. This might become an issue depending on design assumptions. Considering the user's point of view, there are three approaches to UI implementation. Firstly, the app appearance may be identical regardless of the platform it runs on. In this case, the platform-specific conventions may not be fulfilled and only when implemented correctly it will not cause user experience decrease. Secondly, the app may have a completely system-compliant look and feel. In this case, the issue may be raised as to how efficient it is to implement disjointed layouts in the cross-platform solution compared to switching to the native approach. Finally, there is an intermediate approach which assumes that most of the app elements are shared between platforms, but there are some that are platform-specific, e.g., popups, action buttons [9, 17].

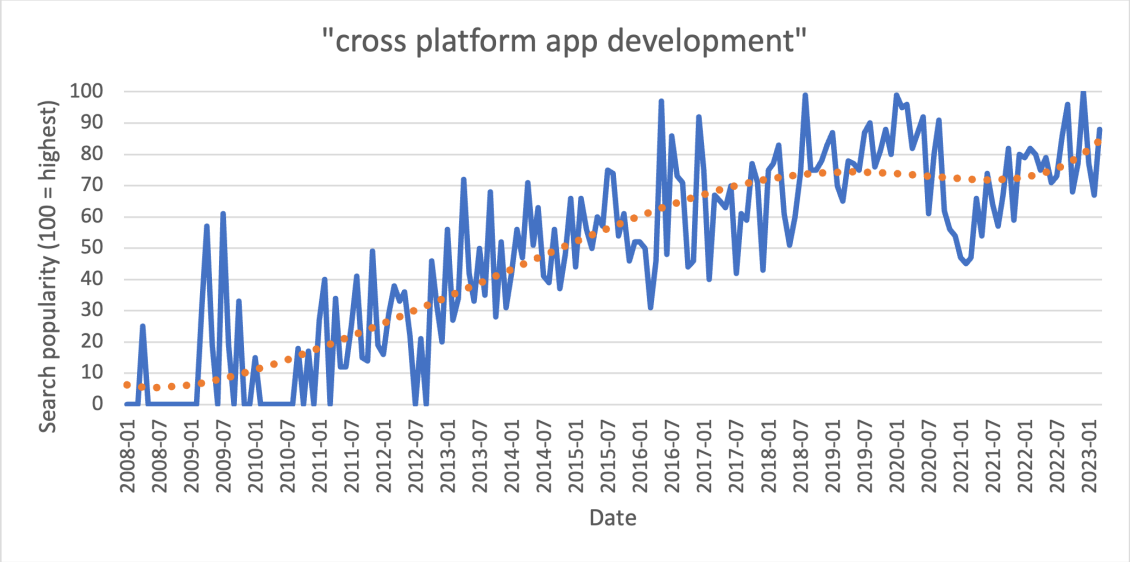

Fig. 3.6. Google Trends search popularity of cross-platform app development (Source: Own work based on [36])

Despite the existing assumption that cross-platform frameworks might produce applications suffering from performance decrease, their popularity significantly grew during the past couple of years. The Figure 3.6 shows the Google search popularity for the phrase "cross platform app development". The upward trend is clearly visible, which depicts the big interest in the topic amongst online users. Apache Cordova and Xamarin are examples of multi-platform frameworks which were the most popular in the beginning, however more

recent solutions have already surpassed them. In Figure 3.7 the rapid rise of Flutter and React Native can be observed. These three frameworks are different from each other but still share the fact that they gather big user groups. For that reason, they have been selected to undergo a review and performance analysis inline with the goal of this thesis.

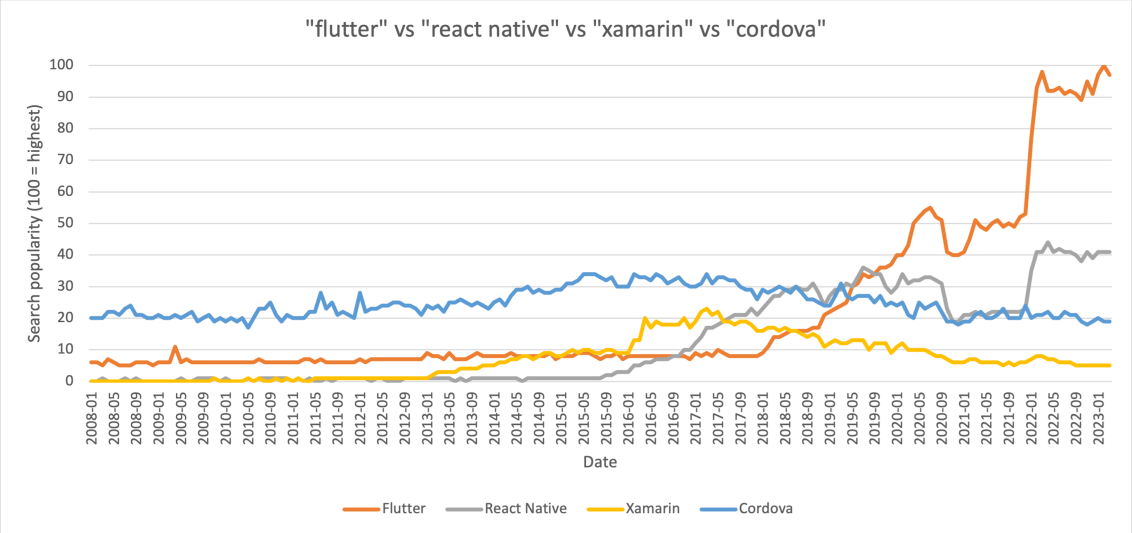

Fig. 3.7. Google Trends search popularity of selected cross-platform frameworks (Source: Own work based on [37])

3.2.1. Cross-platform development approaches

Multiple types of cross-platform solutions can be distinguished. The most commonly present in the scientific literature are the following: hybrid, interpreted, cross-compiled, and Progressive Web Apps (PWAs). Native mobile development may provide better connection with native components and services compared to some of the types mentioned above. This may become the cause of lower performance.

Hybrid approach

The term *cross-platform* tends to be used interchangeably with *hybrid* which technically is incorrect as these are not necessarily synonyms. The hybrid approach to cross-platform development assumes a merge of native and web applications. Typically, the web app is created with the use of web solutions, e.g. HyperText Markup Language (HTML), Cascading Style Sheets (CSS) and JavaScript (JS) which then runs inside a web view component provided by the native app, allowing to publish the end-product in regular app stores. Usually, there is a bridge enabling the usage of device native features which are out-of-scope for JS API. Figure 3.8 shows the overall architecture of a hybrid application. One of the drawbacks is that the access to native components is not possible while building the user interface, which makes it more difficult to mimic the native look and feel for the user. The hybrid approach may provide a lower performance for the bigger projects because

the application is executed by the browser engine and the bridge may cause overhead [18, 22, 24, 40, 78].

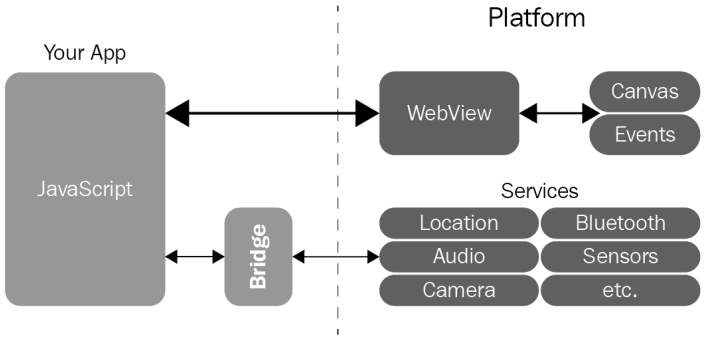

Fig. 3.8. Hybrid app architecture (Source: [42])

Interpreted approach

Similarly to the hybrid approach, web technologies are the primary component of the interpreted approach. The key difference between them is that interpreted apps do not use the web view. Instead, there is an interpreter involved which translates the web elements into native components that can be rendered directly in the operating system. Figure 3.9 shows the overall architecture of an interpreted application. In the context of accessing the native features, there is again a bridge applied. The main advantage over the hybrid approach is the guaranteed native appearance and behavior [18, 40].

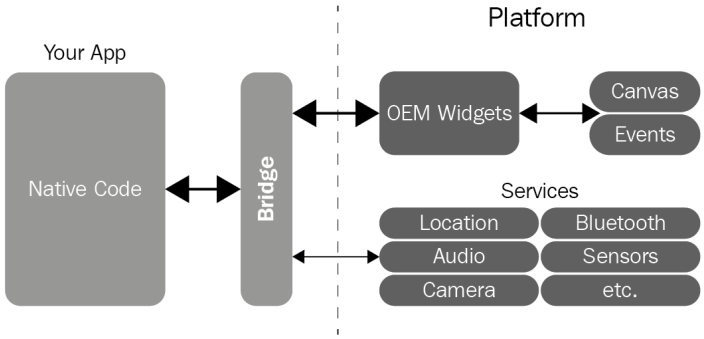

Fig. 3.9. Interpreted app architecture (Source: [42])

Cross-compiled approach

The cross-compiled approach introduces significant changes compared to the previously mentioned solutions. There are no additional layers between the app and the underlying operating system. The Software Development Kit enables the usage of native features and components. A cross-compiled framework uses a programming language of choice which is compiled to native byte code specific to the platform it runs on. Out of the three

described approaches, this one should provide the best user experience and performance, thus gaining a lot of popularity. [18, 22, 24, 40].

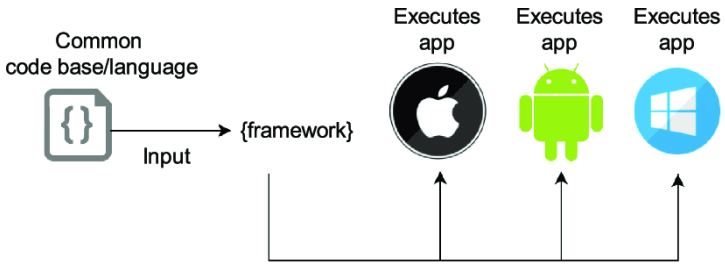

Fig. 3.10. Cross-compiled approach workflow (Source: [18])

Progressive Web Apps

Progressive Web App (PWA) is a web application that allows the user to install it on a mobile device as if it was a native app. The installation process differs because native apps are downloaded from the app store, while PWAs can be downloaded per request upon the user's visit using a web browser. In order for the web app to acquire the abilities of PWAs, they must fulfill some technical requirements, mainly the implementation of Service Workers and app manifest file. The Service Worker provides some important functions such as the control of data flow (determining whether a web server or locally cached content should be used to complete a request, see Figure 3.11). PWA runs in the browser with some features hidden, e.g. address bar, which results in the appearance indistinguishable from native apps. The disadvantage of PWAs is the fact that they cannot provide full support for native components because they are limited by browser APIs. Furthermore, there is a compatibility issue as iOS 11.3 is the minimal compatible version to enable PWA installation. [18, 22, 40].

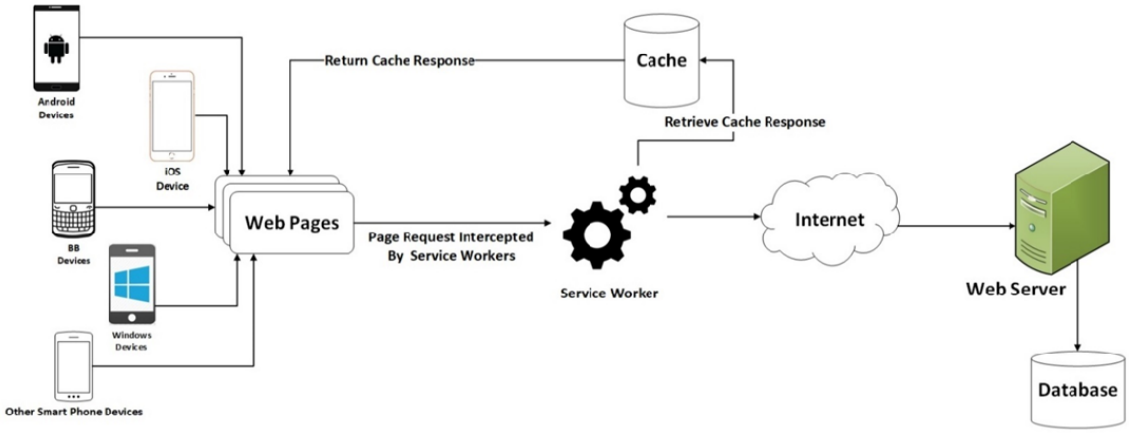

Fig. 3.11. Progressive Web App architecture (Source: [1])

3.2.2. Flutter

Flutter is a cross-platform framework belonging to the cross-compiled approach described above. It supports all the most common platforms: mobile (Android, iOS), web, desktop (Windows, macOS, Linux) and embedded. It utilizes a dedicated programming language, Dart. Being released by Google in 2017, it is a relatively recent solution. Even though, it has gathered a lot of traction in a short time, which resulted in its broad application across different segments of the market. Popular exemplary applications developed using Flutter are: Google Pay, eBay, Alibaba, BMW, Toyota and PUBG MOBILE [25, 26].

Architecture

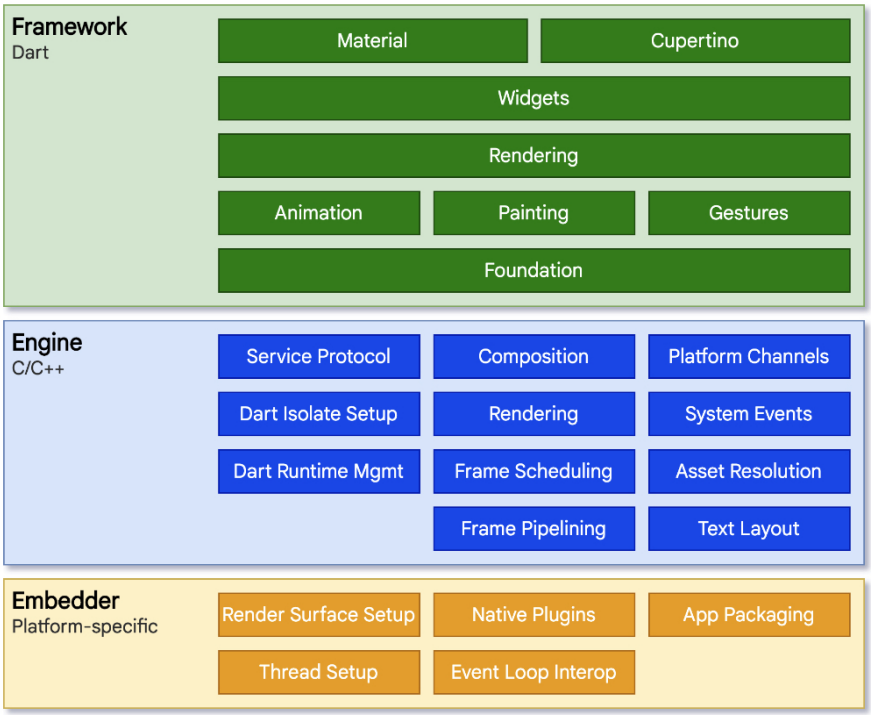

Fig. 3.12. Flutter architecture (Source: [26])

Flutter consists of three primary layers (libraries) that are completely autonomous, as shown in Figure 3.12. This makes it highly modular, with the possibility to substitute any layer by hand.

Embedder layer is placed at the very bottom to directly interact with the operating system in order to manage surface rendering, input methods, event loop and native plugins. It is platform-specific therefore is implemented with appropriate programming languages for each platform, e.g. Java, C++, Objective-C or Objective-C++ [26].

Engine is the most significant layer which provides the "low-level implementation of Flutter's core API", meaning runtime management, platform channels, text layout, scene composition, rendering, and more. Flutter engine compiles source code to platform-specific

byte code therefore there is no need for an external intermediate layer allowing communication with the underlying operating system to access any native features, such as JavaScript bridge in case of hybrid and interpreted frameworks. This provides a big advantage from the perspective of achievable performance. Moreover, from the point of view of developers, debugging is more complex with JS bridge because in order to fix any runtime errors they must be retraced through the bridge first [26, 91].

Engine is also where the graphics aspect is contained. Since the very first release, Flutter has been using Skia as its graphics engine. As stated above, Flutter translates the source code to byte code instead of using an intermediate layer which would use the source code to determine native components, in the contrast to interpreted cross-platform solutions. Therefore, the byte code itself is passed directly to Skia for rendering. Importantly, a Skia copy is included in the *Engine* layer in order to allow the latest version to be used even on devices with older operating system version. Moreover, this enables the app to look the same even on older devices, which would not be otherwise possible. Nonetheless, there have been some performance issues reported, especially by iOS-targeting developers. For that reason, Google has begun working on a new graphics engine for Flutter. Recently (May 2023), it was announced that the new engine, *Impeller*, became enabled by default for iOS applications and is available for testing for Android. It is supposed to improve performance and eliminate jank problems. [26, 27, 45].

The final layer, *Framework*, is the interface for developers. It contains all the components required for writing applications, such as widgets, animations, Material and Cupertino libraries, etc. [26].

Web support

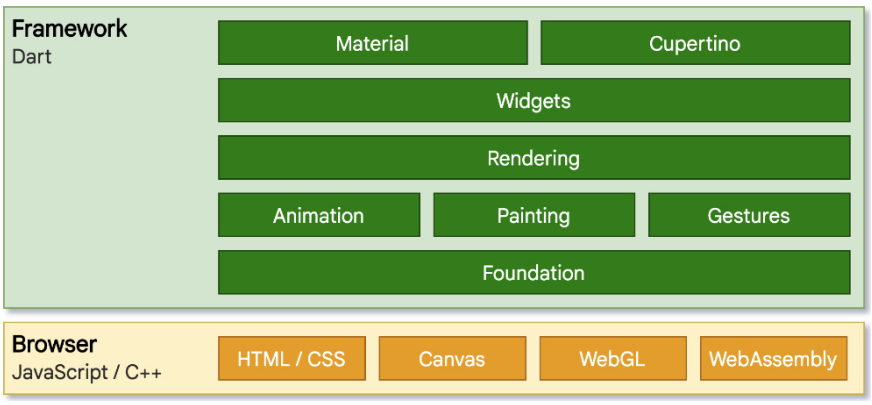

Fig. 3.13. Flutter web-adapted architecture (Source: [26])

In order to support web, for such builds the Flutter's engine is specifically adapted to include the standard browser APIs (see Figure 3.13). From the rendering perspective, there are two possible methods: HTML+CSS+Canvas+SVG or CanvasKit. The former minimizes the download size, while the latter usually offers better performance.

Rendering

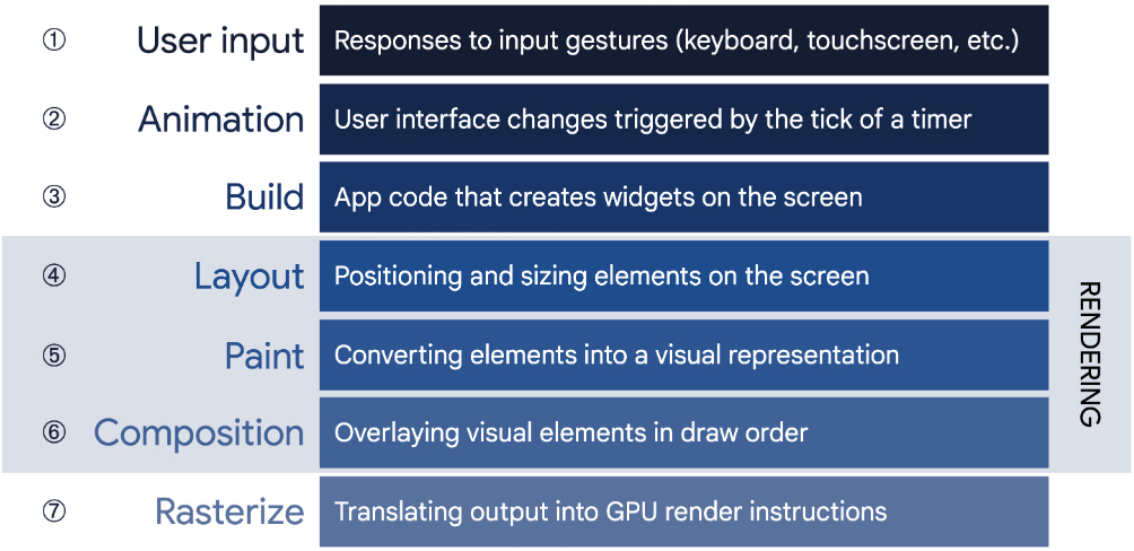

Fig. 3.14. Flutter rendering flow (Source: [26])

Figure 3.14 shows Flutter's rendering flow. Flutter is a "reactive, pseudo-declarative UI framework" which means the app's state and user interface layout are separate. User input affects the state, which then results in UI rebuild. Flutter solution to this methodology is the existence of three interconnected trees: *Widget Tree*, *Element Tree*, and *Render Tree* (see Figure 3.15). The primary components of the framework are widgets which are combined to create user interfaces. For each widget in the tree, the element object is created, and for each element- a render object. The element acts as a connector between widgets and their render objects. Whenever the app's state changes, runtime types of widgets and render objects are checked. Only if they are different, a new render object needs to be created. Otherwise, it is simply updated which is "cheap" performance-wise, just as rebuilding widgets thanks to the fact they are immutable [26].

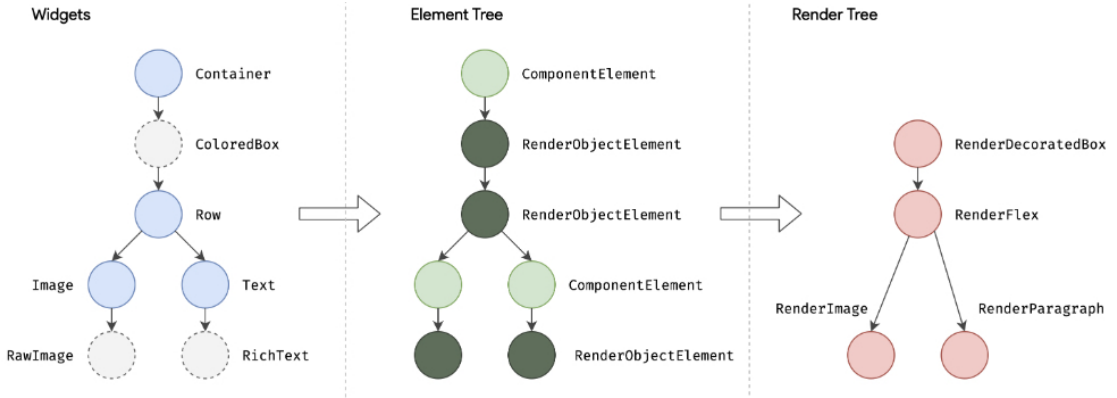

Fig. 3.15. Flutter Widget, Element, Render trees (Source: [26])

User interface design

By default, when creating a new Flutter project, the application depends on the Material library resulting with an appearance mostly found on Android devices, in compliance with the design system described in chapter 3.1.1. Even though it is possible to build such an application for any supported operating system, in most cases a different UI should be preferred for iOS devices to guarantee accordance with Human Interface Guidelines as well as platform styling conventions, as described in chapter 3.1.2. The components for iOS system are contained in the Cupertino library. Therefore, it is necessary to decide on a design approach early on. Should it be required for the application to have the exact same looks on all platforms (Material, Cupertino or custom), a single UI codebase is an obvious solution. However, if the appearance should be completely platform-specific, there would need to be a separation created and all the layouts would have to be developed in parallel. The third method assumes the usage of the same primary layout components with some minor OS-specific additions, e.g. popups, which does not require the full separation but rather sporadic one. Figures 3.16 and 3.17 show the difference between Material and Cupertino dialogs [26, 45].

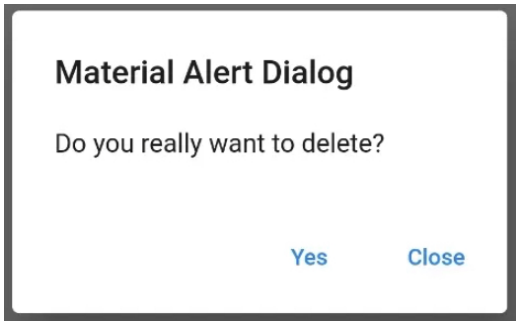

Fig. 3.16. Flutter Material dialog (Source: [28])

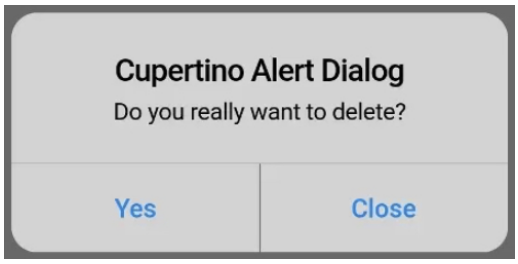

Fig. 3.17. Flutter Cupertino dialog (Source: [28])

3.2.3. React Native

React Native is a cross-platform framework released in 2015 belonging to the interpreted approach described above. Although, after recent architecture changes, it differs moderately from other solutions of the same type. It is a mobile-only cross-platform framework, meaning that it enables the usage of single codebase for different operating systems, however it is restricted to mobile platforms (Android and iOS). React Native utilizes JavaScript for both application development through React framework and native features access. It is one of the most popular mobile technologies used in the market by some of the biggest companies, e.g. Meta (Facebook), Microsoft (Office, Outlook, Teams), Discord, Tesla, Pinterest, and many more [64, 71].

Architecture

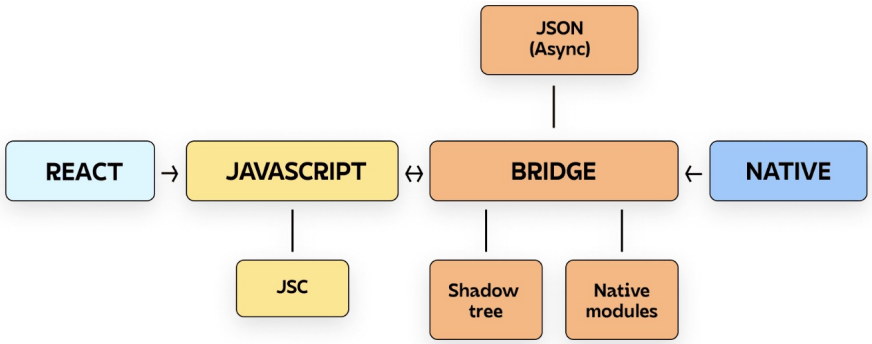

Fig. 3.18. React Native previous achitecture (pre-v0.68) (Source: [51])

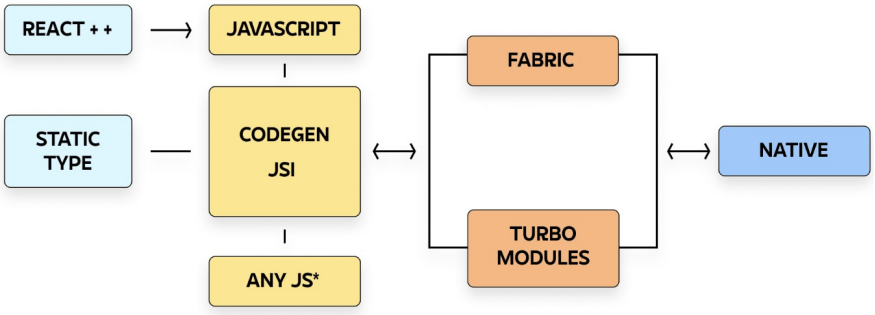

Fig. 3.19. React Native New Architecture (v0.68+) (Source: [51])

Up until React Native version 0.68 (30 Mar. 2022) the framework's architecture (shown in Figure 3.18) was analogous to the general interpreted cross-platform solution architecture presented in Chapter 3.2.1. Because of the drawbacks caused by the *Bridge*, the *New Architecture* (shown in Figure 3.19) has been developed and applied with the release of version 0.68. Table 3.2 explains the major improvements achieved.

The most important difference is the complete removal of the bridge and bringing the *JavaScript Interface (JSI)* in its place. This change aimed to eliminate the unnecessary delays the bridge caused and the uncertainty of data delivery. Moreover, *JSI* enables the cooperation between Native components and JavaScript via the ability to reference C++ Host objects and call their methods directly [51].

Fabric is the new React Native's renderer. The rendering process itself is described below. *Fabric* offers many advantages compared to the previous rendering solution. Its core code is shared by target platforms instead of being implemented separately for each of them, thus improving the cross-platform aspect. It decreases the launch time through lazy initialization of native components, as well as enables multithread operations. Overall, there are many benefits guaranteed by Fabric, which together account for the increase in performance and scalability [65].

Turbo Modules are the improved version of *Native Modules*, which were previously the solution to interconnectivity between JavaScript and the native code in order to allow the usage of platform-specific APIs not available directly via JS. *Turbo Modules* take advantage of lazy initialization, thus vastly shortening the loading times [66].

Codegen is an optional helper that assures the validity of the connection between JS and native threads. It automatically (upon the application build) generates the interface files for *Turbo Modules* and *Fabric Native Components*. For *Turbo Modules*, the interfaces provide the *JSI* initializer and functions directly executable from both JS and native sides. For *Fabric Native Components*, the interfaces enable the correct component load at runtime. *Codegen* can be additionally executed by hand when needed [67].

Table 3.2. React Native architecture improvements (Source: Own work based on [72])

Old architecture	New Architecture
— unnecessary asynchronous wait for processing caused delay — all the computation was forced to be performed on a single thread — data serialization and deserialization between layers caused additional delays	— synchronous function calls became possible — multithreaded execution is enabled — removed the need to serialize data — new core renderer became shared between platforms

Web support

As stated before, React Native only allows the development for mobile operating systems. Nevertheless, it is possible to add the support for web with little effort via plugins. The most common solution is *React Native for Web*. It functions as a "compatibility layer between React DOM and React Native" [73]. For example, the general `<View>` component used in React Native will be translated to `<div>` when building for web, similarly to the conversion performed by React Native itself for mobile platform [39].

Rendering

React Native utilizes a three-phase render pipeline. The process flow is presented in Figure 3.20. The phases are: *Render*, *Commit* and *Mount*. The *Render* phase constitutes of creation of *React Element Trees* and *React Shadow Trees*. The *Commit* phase is responsible for determining which trees are ready to proceed to the final step, as well as calculating the layout constraints. In the *Mount* phase, the *Host View Tree* is build based on the finalized *React Shadow Tree* [68].

The tree creation process is similar to the one applied in the Flutter framework. In React Native, the building blocks of user interface layouts are *React Elements* written in JavaScript. Each of them is translated to a single *React Host Component* which is

able to become transformed into a platform-specific equivalent, e.g. `<Text>` can become `<TextView>` for Android. Those components connected together form a *React Element Tree*. Subsequently, each *React Host Component* is a base for a *React Shadow Node* which offers additional information: props and layout constraints (position and size). Finally, a *Host View Tree* is created, where each node (*Host View*) is rendered on the screen using the corresponding platform-specific component set up according to the properties and constraints obtained. Figure 3.21 illustrates the example of a tree creation process for a simple layout consisting of only text (built for Android). [68].

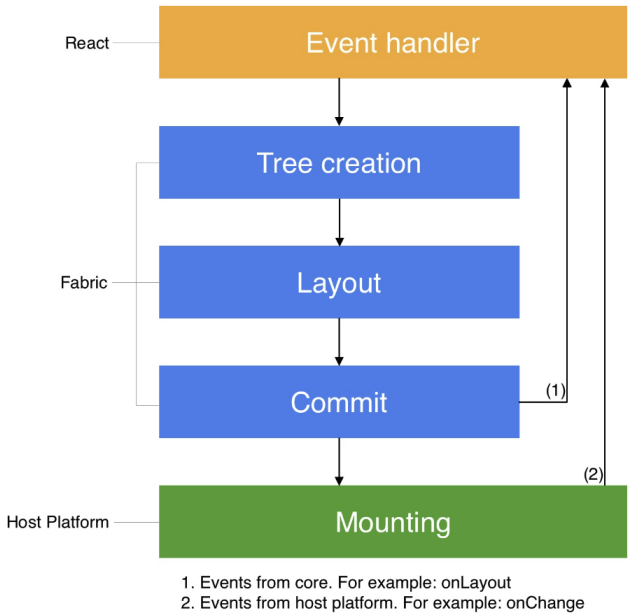

Fig. 3.20. React Native rendering flow (Source: [68])

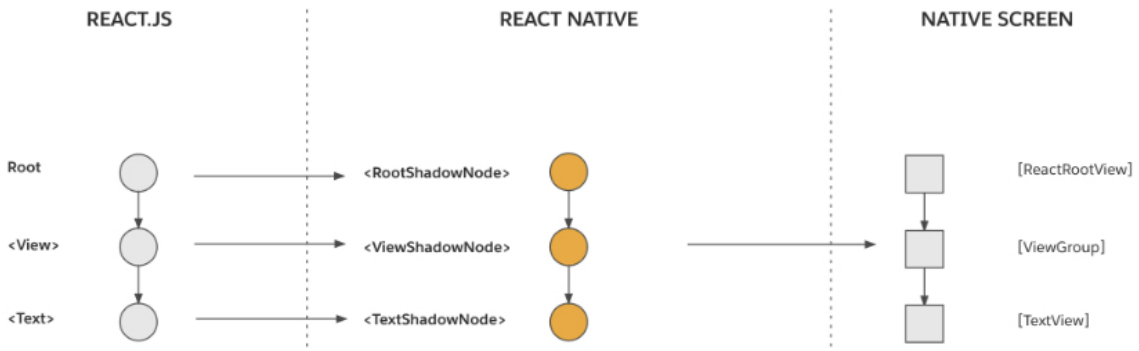

Fig. 3.21. React Native React Element, React Shadow, Host View trees (Source: [68])

The complete rendering process takes place across three threads: *UI*, *JS*, and *Background*. The first is responsible for host view management, the second facilitates the render phase and the third handles layouts. A *View Flattening* algorithm is leveraged with the goal of reducing the depth of layout trees whenever possible. [68, 69, 70].

User interface design

As explained in the Rendering section, React Native's user interface components end up being converted into platform native components. Therefore, in the context of UI design and user experience level, it is on par with the native development itself. This remains true even when extending the application to support web, although in this case some limitations may arise as not all the React Native components have their web component equivalent, e.g. when needing to access mobile platform native features [39].

3.2.4. Comparison

The following Table 3.3 includes the summarized comparison between Flutter and React Native.

Table 3.3. Cross-platform framework comparison (Source: Own work)

Characteristic	**Flutter**	**React Native**
Type	Cross-compiled	Interpreted
Initial release	2017	2015
Current version	3.10	0.71
Platforms	— Android — iOS — Web — Windows — macOS — Linux	— Android — iOS — Web (requires a plugin)
Technology	Dart	React
Rendering	Canvas drawing	Native components
Native APIs	Direct access	Direct access

3.2.5. Evaluation of cross-platform frameworks

Considering the ongoing changes to the mobile market as well as the saturation of different cross-platform solutions, performing the selection of a specific framework for application development becomes a more and more complicated task. Especially because for each use-case there might be a different optimal solution based on different aspects such as business costs, development limitations, etc. For that reason, a few evaluation methods have been proposed in the literature, some of them very simple and some of them more detailed. There is a lot of scientific work performing the assessment of a selection of available cross-platform frameworks according to those methods. The issue with such evaluation outcome lays in the fact it can become out-dated in a short time with the breaking updates to the considered technologies. For example, a majority of literature on the topic of

cross-platform development considers a tool Adobe PhoneGap which has been discontinued and thus is no longer relevant. Therefore, the assessment frameworks themselves are a strong base for performing further research without the risk of becoming obsolete, while the actual assessment ought to be repeated over time to ensure up-to-date results.

Table 3.4. Cross-platform framework evaluation criteria (Source: Own work based on [74])

Perspective	**Criteria**
Infrastructure	License, Supported target platforms, Supported development platforms, Distribution channels, Monetization, Internationalization, Long-term feasibility
Development	Development environment, Preparation time, Scalability, Development process fit, User interface design, Testing, Continuous delivery, Configuration management, Maintainability, Extensibility, Integrating custom code, Pace of development
App	Access to device-specific hardware, Access to platform-specific functionality, Support for connected devices, Input device heterogeneity, Output device heterogeneity, Application life cycle, System integration, Security, Robustness, Degree of mobility
Usage	Look and feel, Performance, Usage patterns, User authentication

Table 3.4 provides an overview of some of the evaluation criteria proposed in the literature, divided into four perspectives: Infrastructure, Development, App and Usage. The first perspective revolves around licensing, platform support, distribution and long-held operability. Development perspective considers all the aspects directly connected to software development, e.g. available IDEs, learning and implementation effort, scalability, user interface design abilities, maintainability and testing. App perspective includes mainly security, hardware access and external devices. Finally, Usage perspective concerns the aspects connected with end-user experience such as app appearance and behavior, authentication and performance [40, 74].

The process of the complete evaluation of a single cross-platform framework according to the above-mentioned criteria is a complex and labor-consuming task perfectly fitting into the methodology of Multi-Criteria Decision-Making (MCDM). This methodology can provide a standardized way of approaching the issue of framework selection. Some of the popular solutions for MCDM are Analytic Hierarchy Process (AHP), Technique for Order of Preference by Similarity to Ideal Solution (TOPSIS) or Multi-attribute Utility Theory (MAUT). In the literature there are also proposed approaches created from the merge of a subset of those solutions, e.g. Integrated Multi-Criteria Decision-Making [49].

4. MOBILE APPLICATION PERFORMANCE MEASUREMENT

Application performance can be analyzed differently depending on the defined context. It is usually performed during the implementation, testing and maintainance phases present in the general software development cycle. In this chapter, the importance of app performance measurement is ephasized and exemplary metrics are explained separately for mobile and web developement.

The possibility to install and uninstall mobile apps within seconds directly affects the commercial mobile app market. In order to compete, publishers need to make sure they provide their services at the highest quality acquirable so that users don't turn to the competitor's solution. Almost 30% of consumers instantly switch to other available products if their needs are not satisfied. App performance is considered to be one of the more important aspects in this context, as 70% of users perform an immediate switch solely based on loading time being too long. For that reason, no matter how big or successful, each mobile app publisher must not underestimate the importance of performance offered by their product [35].

Mobile app performance is a broad term and can be understood differently and described with higher or lower level of detail. It can be seen as only dependent on consumers' impressions. The main aspects of app performance as perceived by users are device resource usage, smoothness, loading times, app size etc. On the other hand, considering app performance from the publisher's perspective includes additional metrics, e.g., user return rate and crash occurrence frequency.

Table 4.1: Selected app performance metrics from the perspective of user experience (Source: Own work based on [10, 75, 90])

Metric	**Description**
CPU usage	Measures the percentage of CPU capacity utilized by the running application. Suboptimal source code and heavy computations may cause high CPU usage which negatively affects user experience.

Continued on next page

Table 4.1: Selected app performance metrics from the perspective of user experience (Source: Own work based on [10, 75, 90]) (Continued)

Memory consumption	Measures the amount of RAM consumed by the running application. It is especially significant when considering low-end devices with very little memory available. High memory usage results with fewer apps being able to run at the same time on a device. This metric is used to find and fix potential memory leaks.
Power consumption	Measures the amount of energy consumed by the running application. It is usually directly related to other metrics, especially CPU usage, because heavy load on CPU is one of the biggest causes for battery drain.
Smoothness	Measures the frame rate in FPS (Frames Per Second). The higher the frame rate, the higher the perceived smoothness of the app. 60 FPS is considered to be an indicator of a frame rate providing very good user experience. Furthermore, the frame rate should remain as stable as possible during the app use.
Loading times	There are numerous types of loading times measurement. First and foremost, the app's launch time can be considered but also the times of navigation, page load, computation, etc. These are the metrics highly impacting the user's perception of app performance.
App size	Consists of two measures: the storage space used by the installed application, as well as the size of the installer itself. It is especially important for low-end devices.

Table 4.2: Selected app performance metrics from the perspective of publisher (Source: Own work based on [15, 43, 80])

Metric	**Description**
Load handling	Load testing is an important part of the performance measurement. The goal is to simulate a very high user count at once. The outcome helps to find potential performance issues and bottlenecks in critical scenarios and thus, optimize the app scalability. Load testing is usually performed with specialized tools automating the process.

Continued on next page

Table 4.2: Selected app performance metrics from the perspective of publisher (Source: Own work based on [15, 43, 80]) (Continued)

Total and new users count	These metrics represent the popularity of the app. User count should be monitored in relation to issue occurrence in order to make sure the app is scalable.
Crash-free users percentage	Measures the frequency of crash occurrence among the user base. Should be considered of high priority and monitored regularly, especially after new app features introduction.
User return rate	Valuable metric used to detect existence of issues leading to user dissatisfaction resulting in abandoning the app. Calculated as the number of user sessions during a time period.
Revenue	Measures the sales connected to the app. It is important in relation to Return Of Investment. It is helpful with analyzing effectiveness of marketing strategies applied.
User satisfaction	Metric based on ratings and surveys. Should be monitored in order to discover existence of functional and performance issues.

Table 4.1 contains an overview of the major application performance metrics understood to be related to user experience. Per contra, Table 4.2 provides descriptions for some of the metrics considered important from the perspective of the app publisher. As can be observed, app performance can be viewed in different ways depending on the context, e.g. business or technical.

5. RESEARCH METHOD

Literature review concluded in Chapter 3.2.5 portrays the process of cross-platform framework evaluation as a multifaceted problem. The reason being, there are numerous aspects (criteria) that must be considered. Therefore, different major research approaches may be distinguished. A potential approach could revolve around choosing a single cross-platform framework that would get extensively analyzed in accordance with those criteria. Another method could be to select a few frameworks for a comparison while restricting the criteria to a concise subset that will decrease the complexity and labor consumption but still represent a relevant research perspective. The latter approach is the one applied for the purpose of this thesis. While various criteria from all the perspectives listed in Table 3.4 are accounted for within the framework descriptions in Chapter 3.2, this thesis's research focuses on the *Performance* criterion. Some other criteria answered indirectly are: supported platforms, user interface design, access to platform-specific functionality, as well as look and feel.

5.1. PERFORMANCE METRICS

In this chapter, specific performance metrics are chosen according to the literature review conducted to provide a knowledge base for Chapter 4.

The following metrics are proposed for the research on mobile application performance:

— CPU usage
— Memory consumption
— Power consumption
— Frame rate (FPS)

The variety of those metrics should allow for diversified experiments with valuable outcomes regarding multiple mobile application performance aspects.

5.2. RESEARCH SCENARIOS

Naturally, the best solution to acquire real-life performance data per each tested technology would be to use multiple complete real-life applications. However, implementation of even one of such application would require a lot of work and arguably a whole team

of specialists responsible for different aspects of the development. In order to somehow abstract the testing environment, the most common and significant functionalities can be extracted from available mobile applications to provide a list of elements that should be tested.

The following research scenarios are proposed in relation to the selected functionalities:

— **Research scenario 1: List scrolling and filtering**
One of the most common elements of mobile applications is a scrollable list view. It can be found in social media apps, news apps, games and e-commerce apps. Additionally, quite often a mechanism of filtering those lists by the user is applied. For this research scenario, a view should be constructed with a scrollable and filterable list of randomly generated text data.

— **Research scenario 2: Animations**
Currently, a big emphasis is put on the look and feel of mobile applications in order to guarantee the highest level of user experience. One of the crucial aspects are the animations, which both improve the visuals but also provide the user with the feedback resulting from the actions performed. For this research scenario, a view should be constructed with exemplary animations utilized, e.g., element size change, rotations.

— **Research scenario 3: File I/O (Input & Output)**
Another important functionality that can be found in many popular applications is the ability to upload files to the app from the device storage, as well as save files with some app data on the device. For this research scenario, a view should be constructed with the buttons enabling uploading and presenting an image from the device and saving a file with some text data.

— **Research scenario 4: Common UI elements**
User interface layouts are built with many different components, such as toggle buttons, drop-down buttons, and text fields. One of the most common usage of text fields are fillable forms that can be found in registration process, for example. Moreover, almost always there are multiple pages that can be navigated between. For this research scenario, a view should be constructed with a bottom navigation bar enabling switching between two views. The first view should consist of various controls, e.g., toggle buttons, and the second view should include a text field form. Additionally, there should be a button enabling navigating to a separate page.

5.3. TESTING TOOL

IDEs such as Android Studio and Xcode offer built-in profilers that can be utilized for performance testing during mobile application development. However, in case of analyzing more than one platform, multiple profilers would have to be used as those IDEs support only a single operation system each. Therefore, in order to produce the most comparable results, *GameBench Studio PRO*, a standalone testing tool, has been selected for the purpose of this thesis.

GameBench Studio PRO is a performance monitoring tool which can be used for both Android and iOS applications. It is mostly dedicated to mobile game profiling but nevertheless it can be successfully used with virtually any mobile app. Some of the key features offered by the tool are: a Web dashboard for session management and a wide range of supported metrics (CPU, GPU, RAM, Battery and Network usage, Frame rendering, Data download and upload).

5.4. TESTING DEVICES

Table 5.1 contains technical specification of the devices used as the environment for experiments.

Table 5.1: Testing devices (Source: Own work based on [53, 54, 55, 56])

Device	Sony Xperia Z1	Xiaomi Mi 9T Pro	iPhone 7	iPhone 13 mini
Released	Q3 2013	Q3 2019	Q3 2016	Q3 2021
OS	Android 5.1.1	Android 11	iOS 15.7.3	iOS 16.5
CPU	Qualcomm Snapdragon 800 (2.20GHz, 4-core)	Qualcomm Snapdragon 855 (2.84GHz, 8-core)	Apple A10 Fusion (2.37GHz, 4-core)	Apple A15 Bionic (3.22GHz, 6-core)
RAM	2GB	6GB	2GB	4GB
Display	TFT 1080x1920px (5.00") 441ppi	AMOLED 1080x2340px (6.39") 403ppi	IPS TFT 750x1334px (4.70") 326ppi	OLED 1080x2340px (5.40") 477ppi

The devices were selected in a way to cover different price points and ages so that the research can lead to the most general results possible, having regard to the restrictions caused by the limited access to devices.

6. IMPLEMENTATION OF SAMPLE APPLICATIONS

In this chapter, the development of the sample applications is presented. The implementation is based on the research scenarios defined in chapter 5.2. Native solutions for Android and iOS were created using Jetpack Compose and SwiftUI, respectively.

6.1. RESEARCH SCENARIO 1: LIST SCROLLING AND FILTERING

The application consists of two main elements: a *Switch* and a *List*. Toggling the *Switch* applies filtering (only every 6th element remains). The *List* is generated automatically with 10000 elements and is scrollable. The optimal components (e.g. Flutter's *ListView.builder*) have been chosen per each technology, therefore only visible items are rendered.

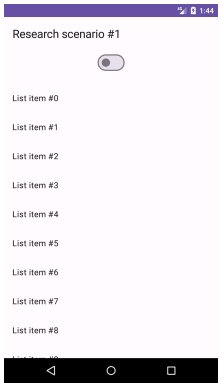

Fig. 6.1. App 1: Kotlin (Source: Own work)

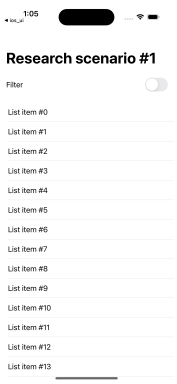

Fig. 6.2. App 1: Swift (Source: Own work)

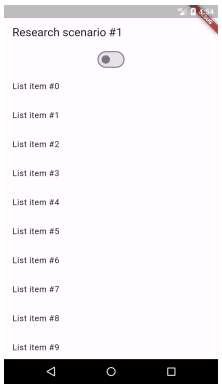

Fig. 6.3. App 1: Flutter Android (Source: Own work)

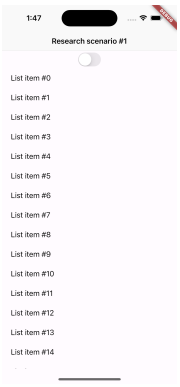

Fig. 6.4. App 1: Flutter iOS (Source: Own work)

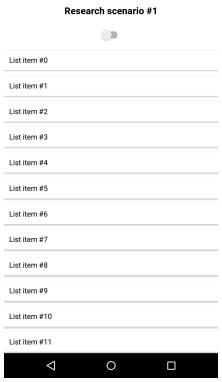

Fig. 6.5. App 1: React Native Android (Source: Own work)

Fig. 6.6. App 1: React Native iOS (Source: Own work)

6.2. RESEARCH SCENARIO 2: ANIMATIONS

The application consists of rows of *RotatingIcons* and *GrowingIcons*. The former animates its rotation and the latter animates its size, with different durations and directions.

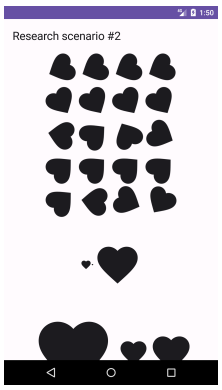

Fig. 6.7. App 2: Kotlin (Source: Own work)

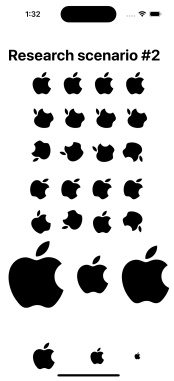

Fig. 6.8. App 2: Swift (Source: Own work)

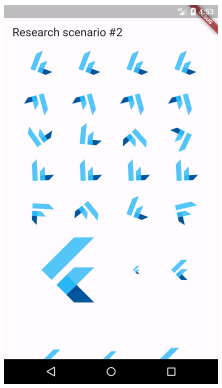

Fig. 6.9. App 2: Flutter Android (Source: Own work)

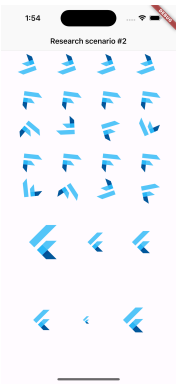

Fig. 6.10. App 2: Flutter iOS (Source: Own work)

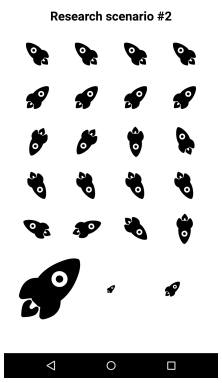

Fig. 6.11. App 1: React Native Android (Source: Own work)

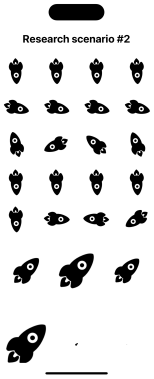

Fig. 6.12. App 1: React Native iOS (Source: Own work)

6.3. RESEARCH SCENARIO 3: FILE I/O

The application consists of two *Buttons* and a *Grid View*. The first *Button* saves a text file on the device and the second allows selecting images from the Gallery. Selected images are then presented in a 2-column scrollable *Grid view*.

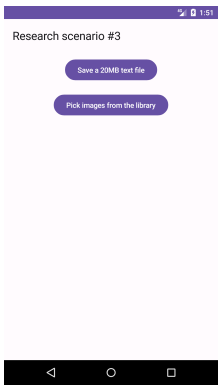

Fig. 6.13. App 3: Kotlin (Source: Own work)

Fig. 6.14. App 3: Swift (Source: Own work)

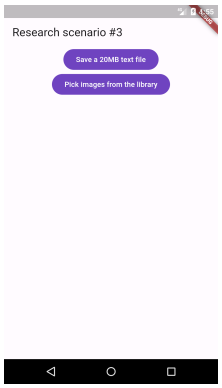

Fig. 6.15. App 3: Flutter Android (Source: Own work)

Fig. 6.16. App 3: Flutter iOS (Source: Own work)

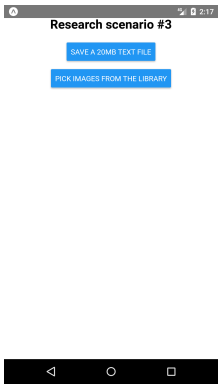

Fig. 6.17. App 3: React Native Android (Source: Own work)

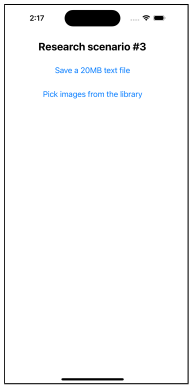

Fig. 6.18. App 3: React Native iOS (Source: Own work)

6.4. RESEARCH SCENARIO 4: COMMON UI ELEMENTS

The application consists of three views: *Controls Page*, *Form Page* and *External Page*. Various common UI components are utilized as well as the bottom tab bar navigation.

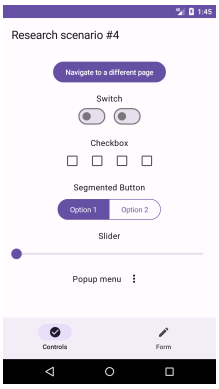

Fig. 6.19. App 4 (1/3): Kotlin (Source: Own work)

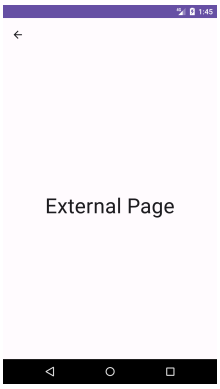

Fig. 6.20. App 4 (2/3): Kotlin (Source: Own work)

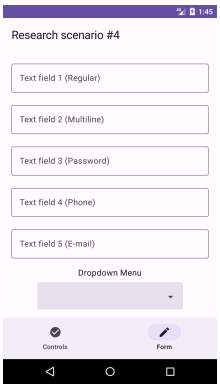

Fig. 6.21. App 4 (3/3): Kotlin (Source: Own work)

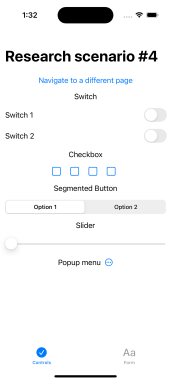

Fig. 6.22. App 4 (1/3): Swift (Source: Own work)

Fig. 6.23. App 4 (2/3): Swift (Source: Own work)

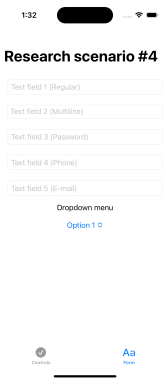

Fig. 6.24. App 4 (3/3): Swift (Source: Own work)

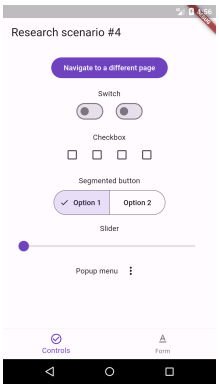

Fig. 6.25. App 4 (1/3): Flutter Android (Source: Own work)

Fig. 6.26. App 4 (2/3): Flutter Android (Source: Own work)

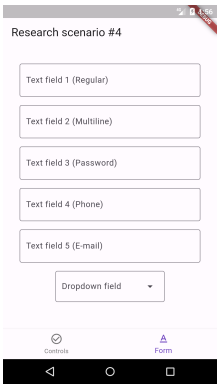

Fig. 6.27. App 4 (3/3): Flutter Android (Source: Own work)

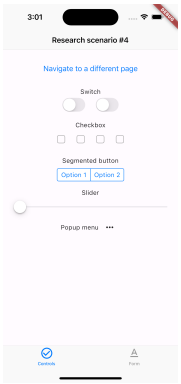

Fig. 6.28. App 4 (1/3): Flutter iOS (Source: Own work)

Fig. 6.29. App 4 (2/3): Flutter iOS (Source: Own work)

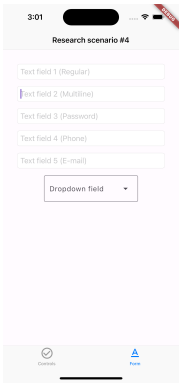

Fig. 6.30. App 4 (3/3): Flutter iOS (Source: Own work)

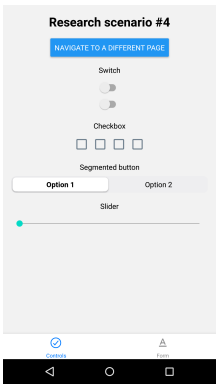

Fig. 6.31. App 4 (1/3): React Native Android (Source: Own work)

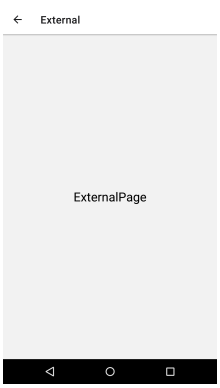

Fig. 6.32. App 4 (2/3): React Native Android (Source: Own work)

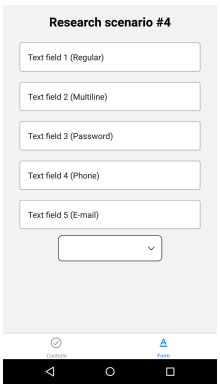

Fig. 6.33. App 4 (3/3): React Native Android (Source: Own work)

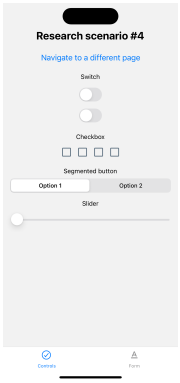

Fig. 6.34. App 4 (1/3): React Native iOS (Source: Own work)

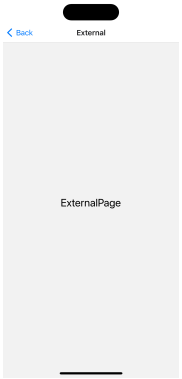

Fig. 6.35. App 4 (2/3): React Native iOS (Source: Own work)

Fig. 6.36. App 4 (3/3): React Native iOS (Source: Own work)

6.5. KEY TAKEAWAYS OBTAINED DURING THE IMPLEMENTATION PROCESS

As considered in Chapter 3.2, one of the significant aspects of cross-platform framework development is the ability to acquire the native look and feel of mobile applications. Of course, this does not concern native approaches. As can be observed in the screenshots above, mobile apps implemented with *Flutter* tend to be almost completely identical to their native equivalents. In the case of *React Native*, iOS apps seem to be quite similar, although not as much as in the case of *Flutter*. However, a big difference can be noticed when comparing Android apps. The reason for that is that *Material Design 3* is currently the primary design system for user interfaces, and *React Native* has not yet adjusted to it. Moreover, the process of theming is easier in *Flutter* because it is set up for both platforms by default, which is not true for *React Native*. On the other hand, *Flutter* requires more code in order to accomplish this native look and feel. For example, `TextField` and `CupertinoTextField` widgets should be used for Android and iOS, respectively, for this purpose. This is usually achieved by creating custom widgets, e.g., `MyCustomTextField` which renders the correct widget according to the running operating system and removes the necessity to check the platform each time a text field is used across the whole application.

During the implementation, the official documentation as well as the community posts could be used as the main sources of knowledge. The development process carried out for the purpose of this thesis provided information for the comparison regarding this context. Both *SwiftUI* and *Flutter* offer the most extensible documentation, which enables quick understanding of new concepts. Next in line is *React Native* which provides a less detailed description of some aspects but is still not lacking. Finally, *Jetpack Compose* documentation comes across as the most general, which can result in the necessity of further research in order to gather enough knowledge for the implementation of a specific component.

7. EXPERIMENT RESULTS

This chapter contains a complete summary of results obtained through the testing process carried out with *GameBench Studio PRO*. Each research scenario yielded four mobile applications, one for each of the considered native and cross-platform technologies. In order to acquire the most reliable results, each app has been tested five times consecutively for a period of one minute. Between each application run, app data was cleared from the device. Afterward, averages and maxima were calculated, and the obtained data is presented in the tables below.

The general choice of performance metrics is described in Chapter 5.1. The specific measurements collected for further analysis, corresponding to the table rows, are:

— **CPU (Average)**
This metric measures the percentage of CPU capacity used by the app. It is calculated differently for Android and iOS. It directly affects power usage. In the case of Android, the normalized value in the range 0-100% is obtained based on CPU load, the number of available cores, and the frequency. For quad-core CPUs, the consistent value of 25% is considered to be an issue, as this means one of the cores is being totally occupied by the app. In the case of iOS, CPU load is measured per core; therefore, the maximum value can be 400% for a quad-core CPU, etc. Consistently high values should be analyzed [29]. In each table, the average value of five recorded sessions is reported.

— **CPU (Max)**
This is the same metric as above. Average CPU usage is not enough to discover potential spikes. In each table, the maximum value of five recorded sessions is reported.

— **RAM (Average)**
This metric measures the amount of memory consumed by the app. This is especially important for low-end devices that offer a smaller amount of memory. Also, unreasonably high values could indicate the existence of memory leaks and cause crashes [30]. In each table, the average value of five recorded sessions is reported.

— **RAM (Max)**
This is the same metric as above. Average memory consumption is not enough to discover potential spikes. In each table, the maximum value of five recorded sessions is reported.

— **Power**
This metric measures the amount of power consumed by the app. It is acquired directly in the case of Qualcomm Snapdragon and Samsung Exynos CPUs and calculated in the case of iOS devices. Thanks to that, the measurement data is more precise than the value reported by the operating system [31]. In each table, the average value of five recorded sessions is reported.

— **FPS (Average)**
This metric measures the frame rate (the number of frames rendered each second), which is directly related to the observed smoothness. The goal should be to achieve 60 FPS, which is the frame rate of the operating system [32]. In each table, the average value of five recorded sessions is reported.

— **FPS Stability**
This metric measures the percentage of the app run with the frame rate being at most 20% higher or lower than the average. The result of 80%+ is considered to be "good" and stable [32]. In each table, the average of five recorded sessions is reported.

7.1. RESEARCH SCENARIO 1: LIST SCROLLING AND FILTERING

Tables 7.1–7.4 contain the values obtained from the sessions performed in the first sample app corresponding to the Research scenario 1. The test was performed manually. The list was scrolled down and up while getting filtered multiple times along the way.

Table 7.1: Research scenario 1 results: Kotlin (Source: Own work)

Device	Sony Xperia Z1	Xiaomi Mi 9T Pro
CPU (Average)	14,32%	3,57%
CPU (Max)	32,09%	6,25%
RAM (Average)	74 MB	75 MB
RAM (Max)	82 MB	96 MB
Power	4,25 mAh	2,69 mAh
FPS (Average)	54	55
FPS (Min)	10	38
FPS Stability	93%	98%

Table 7.2: Research scenario 1 results: Swift (Source: Own work)

Device	iPhone 7	iPhone 13 mini
CPU (Average)	32,66%	31,79%
CPU (Max)	89,60%	56,39%
RAM (Average)	13 MB	19 MB
RAM (Max)	18 MB	23 MB
Power	42 mAh	24 mAh
FPS (Average)	55	55
FPS (Min)	15	42
FPS Stability	90%	97%

Table 7.3: Research scenario 1 results: Flutter (Source: Own work)

Device	Sony Xperia Z1	Xiaomi Mi 9T Pro	iPhone 7	iPhone 13 mini
CPU (Average)	6,40%	2,94%	41,92%	41,68%
CPU (Max)	11,72%	6,52%	51,20%	46,52%
RAM (Average)	69 MB	88 MB	61 MB	95 MB
RAM (Max)	74 MB	147 MB	76 MB	111 MB
Power	3,76 mAh	2,93 mAh	40,80 mAh	38 mAh
FPS (Average)	58	56	59	58
FPS (Min)	35	17	5	46
FPS Stability	98%	98%	90%	98%

Table 7.4: Research scenario 1 results: React Native (Source: Own work)

Device	Sony Xperia Z1	Xiaomi Mi 9T Pro	iPhone 7	iPhone 13 mini
CPU (Average)	26,06%	12,04%	91,18%	48,64%
CPU (Max)	81,69%	26,49%	150,56%	141,89%
RAM (Average)	140 MB	153 MB	107 MB	58 MB
RAM (Max)	151 MB	181 MB	125 MB	68 MB

Continued on next page

Table 7.4: Research scenario 1 results: React Native (Source: Own work) (Continued)

Power	7,26 mAh	3,64 mAh	47,20 mAh	37 mAh
FPS (Average)	55	56	54	52
FPS (Min)	15	35	7	33
FPS Stability	78%	95%	78%	96%

7.2. RESEARCH SCENARIO 2: ANIMATIONS

Tables 7.5–7.8 contain the values obtained from the sessions performed in the second sample app corresponding to the Research scenario 2. The test was performed automatically. The app was opened and left running as the animations were repeated in the loop.

Table 7.5: Research scenario 2 results: Kotlin (Source: Own work)

Device	Sony Xperia Z1	Xiaomi Mi 9T Pro
CPU (Average)	13,56%	11,96%
CPU (Max)	36,79%	14,18%
RAM (Average)	85 MB	140 MB
RAM (Max)	92 MB	183 MB
Power	5,88 mAh	3,17 mAh
FPS (Average)	11	58
FPS (Min)	10	50
FPS Stability	97%	100%

Table 7.6: Research scenario 2 results: Swift (Source: Own work)

Device	iPhone 7	iPhone 13 mini
CPU (Average)	26,11%	33,62%
CPU (Max)	36,62%	40,96%
RAM (Average)	6 MB	9 MB
RAM (Max)	7 MB	10 MB
Power	41 mAh	15,6 mAh

Continued on next page

Table 7.6: Research scenario 2 results: Swift (Source: Own work) (Continued)

FPS (Average)	59	59
FPS (Min)	50	59
FPS Stability	98%	100%

Table 7.7: Research scenario 2 results: Flutter (Source: Own work)

Device	Sony Xperia Z1	Xiaomi Mi 9T Pro	iPhone 7	iPhone 13 mini
CPU (Average)	13,29%	4,10%	49,78%	41,65%
CPU (Max)	19,67%	5,17%	56,48%	46,99%
RAM (Average)	71 MB	84 MB	58 MB	90 MB
RAM (Max)	86 MB	86 MB	68 MB	106 MB
Power	4,08 mAh	2,97 mAh	43 mAh	35,6 mAh
FPS (Average)	55	60	59	60
FPS (Min)	36	60	49	59
FPS Stability	97%	100%	98%	100%

Table 7.8: Research scenario 2 results: React Native (Source: Own work)

Device	Sony Xperia Z1	Xiaomi Mi 9T Pro	iPhone 7	iPhone 13 mini
CPU (Average)	20,04%	4,38%	42,42%	23,30%
CPU (Max)	34,32%	6,10%	90,36%	27,75%
RAM (Average)	105 MB	193 MB	90 MB	34 MB
RAM (Max)	116 MB	221 MB	98 MB	37 MB
Power	7,5 mAh	2,45 mAh	41 mAh	29,6 mAh
FPS (Average)	59	60	59	60
FPS (Min)	28	60	58	58
FPS Stability	99%	100%	93%	100%

7.3. RESEARCH SCENARIO 3: FILE I/O

Tables 7.9–7.12 contain the values obtained from the sessions performed in the third sample app corresponding to the Research scenario 3. The test was performed manually. Firstly, the text file was saved on the device three times by clicking on the relevant button. Secondly, twelve images were loaded from the device through an image picker and displayed. The frame rate metrics were excluded in this scenario because FPS were registered as being equal to 0 while waiting for the file to be saved or images to be uploaded without any screen movement as no new frames would be rendered. Therefore, the results did not provide much value.

Table 7.9: Research scenario 3 results: Kotlin (Source: Own work)

Device	Sony Xperia Z1	Xiaomi Mi 9T Pro
CPU (Average)	2,26%	1,55%
CPU (Max)	16,47%	7,11%
RAM (Average)	68 MB	86 MB
RAM (Max)	89 MB	123 MB
Power	1,01 mAh	0,89 mAh

Table 7.10: Research scenario 3 results: Swift (Source: Own work)

Device	iPhone 7	iPhone 13 mini
CPU (Average)	7,29%	6,97%
CPU (Max)	22,27%	38,84%
RAM (Average)	12 MB	95 MB
RAM (Max)	35 MB	264 MB
Power	14,2 mAh	11,8 mAh

Table 7.11: Research scenario 3 results: Flutter (Source: Own work)

Device	Sony Xperia Z1	Xiaomi Mi 9T Pro	iPhone 7	iPhone 13 mini
CPU (Average)	5,71%	2,97%	25,20%	45,54%

Continued on next page

Table 7.11: Research scenario 3 results: Flutter (Source: Own work) (Continued)

CPU (Max)	27,71%	13,59%	59,82%	194,04%
RAM (Average)	135 MB	287 MB	109 MB	188 MB
RAM (Max)	238 MB	525 MB	156 MB	486 MB
Power	1,8 mAh	1,51 mAh	19 mAh	13 mAh

Table 7.12: Research scenario 3 results: React Native (Source: Own work)

Device	Sony Xperia Z1	Xiaomi Mi 9T Pro	iPhone 7	iPhone 13 mini
CPU (Average)	11,97%	3,64%	16,78%	18,32%
CPU (Max)	30,04%	12,57%	98,25%	92,74%
RAM (Average)	177 MB	295 MB	157 MB	162 MB
RAM (Max)	331 MB	440 MB	396 MB	364 MB
Power	3,06 mAh	1,01 mAh	13,8 mAh	11,8 mAh

7.4. RESEARCH SCENARIO 4: COMMON UI ELEMENTS

Tables 7.13–7.16 contain the values obtained from the sessions performed in the fourth sample app corresponding to the Research scenario 4. The test was performed manually. Firstly, navigation to the External Page was invoked three times consecutively via the relevant button. After that, all the controls were used multiple times, e.g., the switches were toggled and the slider was adjusted. Finally, navigation to the Form Page was invoked via the bottom tab bar, and different types of text fields were filled out sequentially. The frame rate metrics were excluded in this scenario because FPS was registered as being equal to 0 in each second without activity. Therefore, the results did not provide much value.

Table 7.13: Research scenario 4 results: Kotlin (Source: Own work)

Device	Sony Xperia Z1	Xiaomi Mi 9T Pro
CPU (Average)	8,22%	2,23%
CPU (Max)	25,78%	4,55%
RAM (Average)	78 MB	127 MB
RAM (Max)	86 MB	150 MB
Power	2,44 mAh	1,32 mAh

Table 7.14: Research scenario 4 results: Swift (Source: Own work)

Device	iPhone 7	iPhone 13 mini
CPU (Average)	13,38%	12,81%
CPU (Max)	29,14%	29,80%
RAM (Average)	11 MB	12 MB
RAM (Max)	14 MB	22 MB
Power	14,6 mAh	16,8 mAh

Table 7.15: Research scenario 4 results: Flutter (Source: Own work)

Device	Sony Xperia Z1	Xiaomi Mi 9T Pro	iPhone 7	iPhone 13 mini
CPU (Average)	4,42%	1,95%	45,42%	34,88%
CPU (Max)	19,68%	4,22%	69,37%	50,00%
RAM (Average)	76 MB	168 MB	80 MB	116 MB
RAM (Max)	88 MB	202 MB	105 MB	144 MB
Power	1,93 mAh	1,33 mAh	24,6 mAh	20,4 mAh

Table 7.16: Research scenario 4 results: React Native (Source: Own work)

Device	Sony Xperia Z1	Xiaomi Mi 9T Pro	iPhone 7	iPhone 13 mini
CPU (Average)	8,65%	2,75%	28,84%	19,53%
CPU (Max)	39,40%	9,77%	93,35%	39,57%
RAM (Average)	131 MB	151 MB	111 MB	36 MB
RAM (Max)	145 MB	176 MB	125 MB	42 MB
Power	3,2 mAh	1,61 mAh	16,8 mAh	19,8 mAh

This chapter should be viewed as a container for the data acquired from the experiments. Actual analysis and comparisons are realized in the following chapter.

8. RESEARCH DISCUSSION

In this chapter, the results from the performed experiments are aggregated, presented in a visual form, and analyzed in accordance with the purpose and scope of this thesis. The analysis outcome became a base for forming final conclusions.

8.1. EXPERIMENT RESULTS ANALYSIS

In the following sections, firstly the average metric values are calculated to perform the overall comparison between the technologies. Afterward, each research scenario is considered separately.

8.1.1. Generalized results analysis

CPU usage is a significantly valuable metric, especially for low-end devices, because it directly affects power consumption, and batteries in low-end devices are usually of smaller capacity.

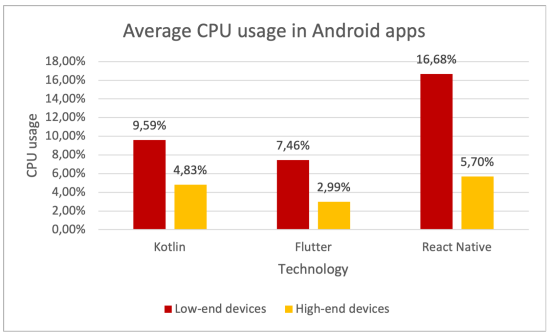

Fig. 8.1. Average CPU usage in Android apps (Source: Own work)

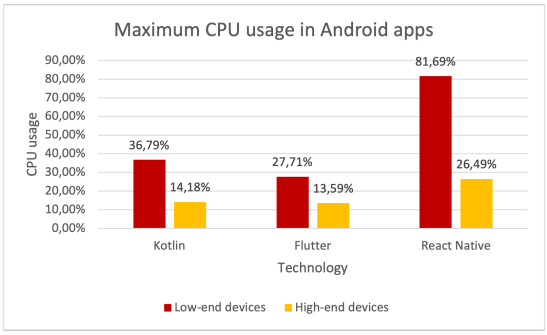

Fig. 8.2. Maximum CPU usage in Android apps (Source: Own work)

Figures 8.1 and 8.2 show the comparison of CPU usage among Android apps developed with Kotlin, Flutter, and React Native. Overall, Flutter apps require the least CPU capacity across both low-end and high-end devices, thus implying the most efficient utilization of system resources. Kotlin apps perform slightly worse than Flutter apps, but they still do not exceed even 10% of CPU usage, which is a great result. React Native apps show the highest CPU usage, most notably on low-end devices. Furthermore, they experience the highest spikes, reaching over 80%. Such spikes may be concerning if they keep happening regularly. Kotlin and Flutter apps exhibit much lower maximum CPU usage.

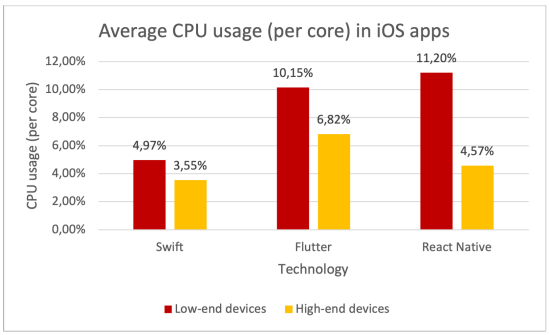

Fig. 8.3. Average CPU usage in iOS apps (Source: Own work)

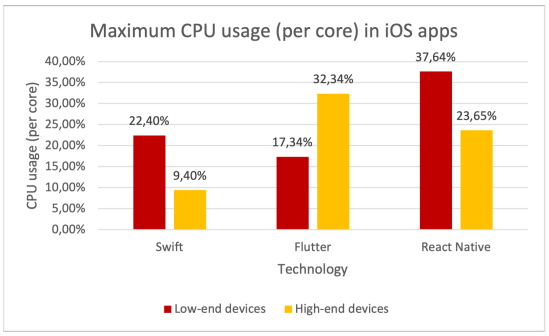

Fig. 8.4. Maximum CPU usage in iOS apps (Source: Own work)

Figures 8.3 and 8.4 show the comparison of CPU usage among iOS apps developed with Swift, Flutter, and React Native. Swift apps perform the best on both low-end and high-end devices, with the average CPU usage remaining just under 5% per core. Flutter apps and React Native apps exhibit similar results, although the former perform better on low-end devices and the latter perform better on high-end devices. However, React Native apps demonstrate the highest CPU usage spikes, similar to their Android equivalents.

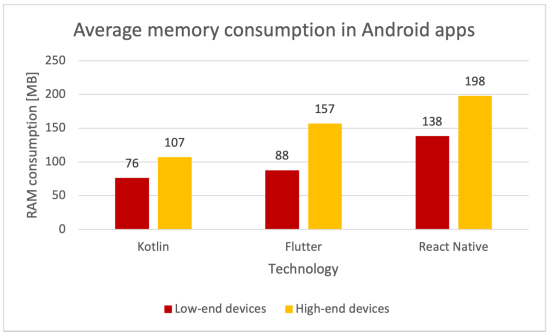

Fig. 8.5. Average memory consumption in Android apps (Source: Own work)

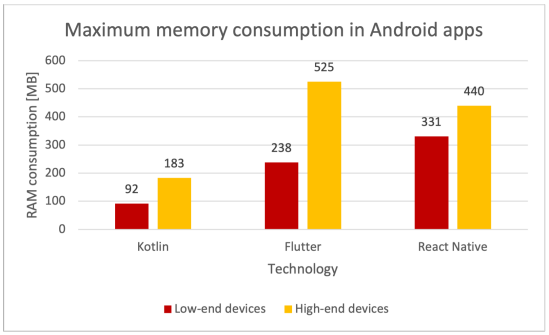

Fig. 8.6. Maximum memory consumption in Android apps (Source: Own work)

Figures 8.5 and 8.6 show the comparison of memory consumption among Android apps developed with Kotlin, Flutter, and React Native. It can be observed that each technology has a similar ratio of memory used by low-end devices to memory used by high-end devices. Overall, Kotlin apps utilize the least memory resources, followed by Flutter apps, and then React Native apps. However, Flutter apps experience the highest spikes in RAM consumption, with the maximum reaching 525 MB on high-end devices as compared to Kotlin apps 183 MB. Nevertheless, high-end devices currently offer a solid amount of memory; therefore, such values (if sporadic) do not have to mean performance issues. Maximum memory usage should be tracked primarily on low-end devices.

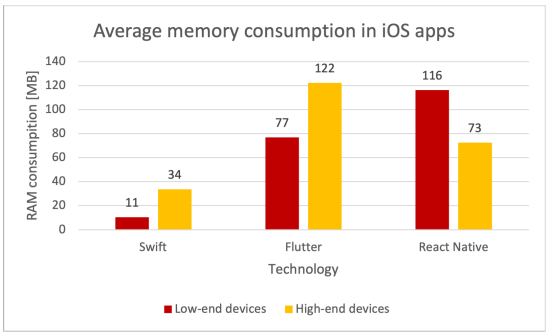

Fig. 8.7. Average memory consumption in iOS apps (Source: Own work)

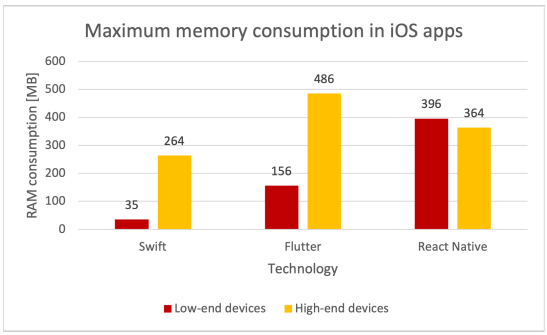

Fig. 8.8. Maximum memory consumption in iOS apps (Source: Own work)

Figures 8.7 and 8.8 show the comparison of memory consumption among iOS apps developed with Swift, Flutter, and React Native. Swift apps exhibit considerably lower memory consumption compared to the other technologies. For low-end devices, Flutter performs moderately better than React Native. On the other hand, React Native apps utilize less memory on high-end devices than Flutter apps. Again, Flutter is responsible for the highest spikes on high-end devices.

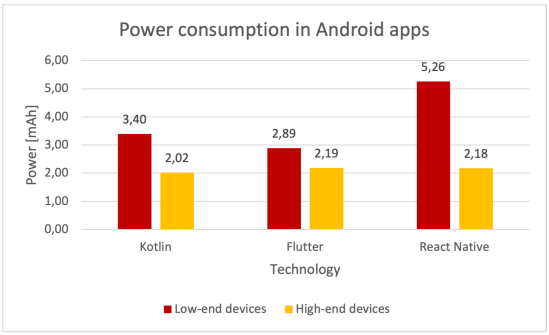

Fig. 8.9. Power consumption in Android apps (Source: Own work)

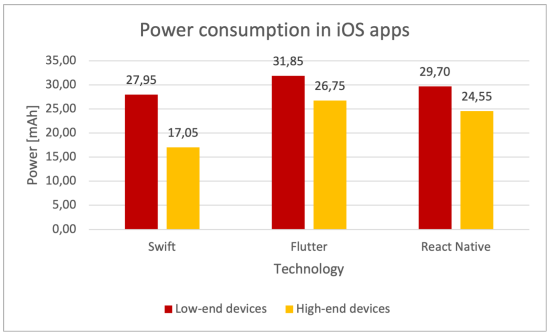

Fig. 8.10. Power consumption in iOS apps (Source: Own work)

Figures 8.9 and 8.10 show the comparison of power consumption among Android and iOS apps developed with Kotlin, Flutter, Swift, and React Native. Overall, Android apps perform similarly on high-end devices. Considering power consumption on low-end devices, Flutter handles it the best. Although the values themselves are really low, the experiment was only carried out for a duration of one minute; therefore, with real-life usage, those values would be further apart. In the case of iOS apps, the results are quite close, apart from Swift apps that perform noticeably better on high-end devices.

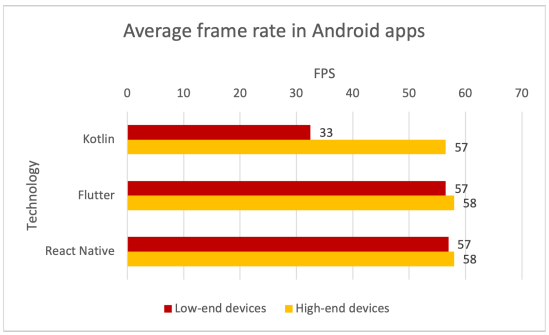

Fig. 8.11. Average frame rate in Android apps (Source: Own work)

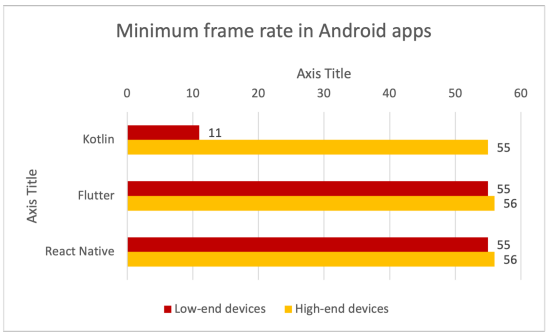

Fig. 8.12. Minimum frame rate in Android apps (Source: Own work)

Figures 8.11 and 8.12 show the comparison of frame rate among Android apps developed with Kotlin, Flutter, and React Native. For high-end devices, there are no significant differences in the smoothness of apps developed with either technology. The range of 55–58 FPS on average is considered to be an excellent result. However, Kotlin apps struggle on low-end devices, experiencing 10 FPS drops and an average frame rate of 33 FPS. Flutter and React Native apps achieve similarly good results on low-end and high-end devices.

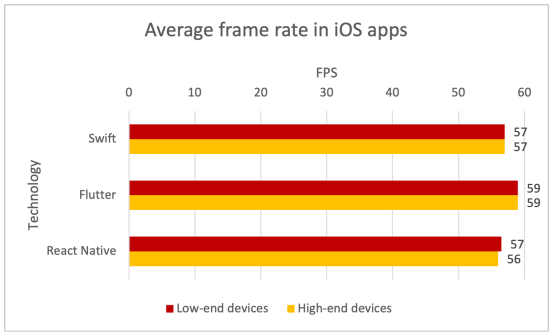

Fig. 8.13. Average frame rate in iOS apps (Source: Own work)

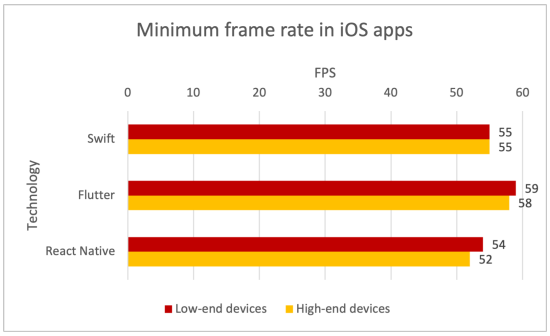

Fig. 8.14. Minimum frame rate in iOS apps (Source: Own work)

Figures 8.13 and 8.14 show the comparison of frame rate among iOS apps developed with Swift, Flutter, and React Native. Overall, Flutter apps exhibit the highest frame rate of 59 FPS on low-end and high-end devices. Swift and React Native apps demonstrate 56–57 FPS frame rate on average. No significant FPS drops can be observed, with React Native apps experiencing a minimum of 52 FPS on high-end devices.

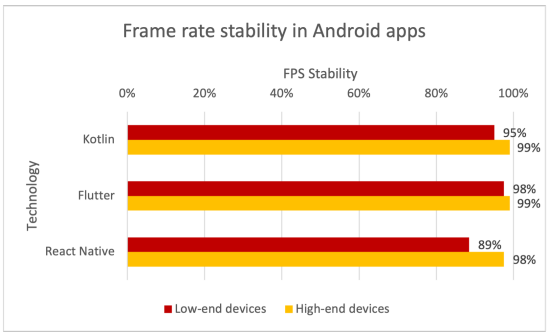

Fig. 8.15. Frame rate stability in Android apps (Source: Own work)

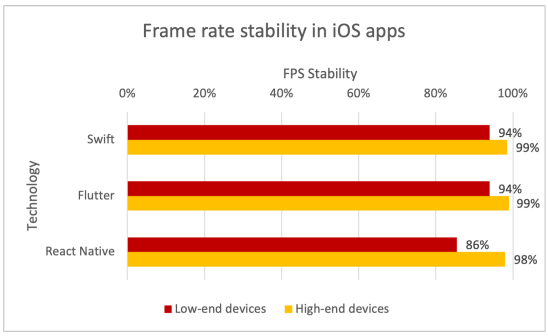

Fig. 8.16. Frame rate stability in iOS apps (Source: Own work)

Figures 8.15 and 8.16 show the comparison of frame rate stability among Android and iOS apps developed with Kotlin, Swift, Flutter, and React Native. The results are closely comparable, considering either platform. On both low-end and high-end devices, each technology offers high frame rate stability. Only React Native apps installed on low-end devices experience stability below 90%.

8.1.2. Research scenario 1 results analysis

The following figures illustrate the aggregated results from the experiments conducted within Research scenario 1 described in Chapter 5.2.

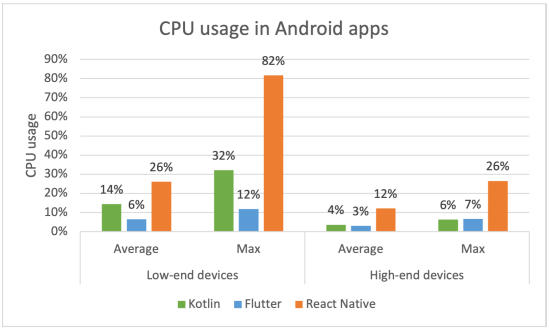

Fig. 8.17. Research scenario 1: CPU usage in Android apps (Source: Own work)

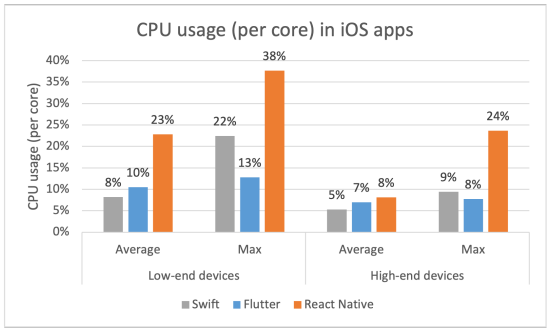

Fig. 8.18. Research scenario 1: CPU usage in iOS apps (Source: Own work)

Figures 8.17 and 8.18 show the comparison of CPU usage among Android and iOS apps developed with Kotlin, Swift, Flutter, and React Native. For Android apps, Flutter offers the best performance, especially on low-end devices. Kotlin apps utilize 14% of CPU capacity on average, which is almost two times less than React Native apps (26%). React Native is again subject to the highest spikes, reaching a maximum of 82% on low-end devices. On high-end devices, Flutter and Kotlin are on par, while React Native requires moderately more system resources. In the case of iOS apps, native apps developed with Swift demonstrate comparable CPU usage to Flutter apps on both low-end and high-end

devices. Again, Flutter experiences the smallest spikes, while React Native exhibits the highest average CPU load combined with the most significant spikes.

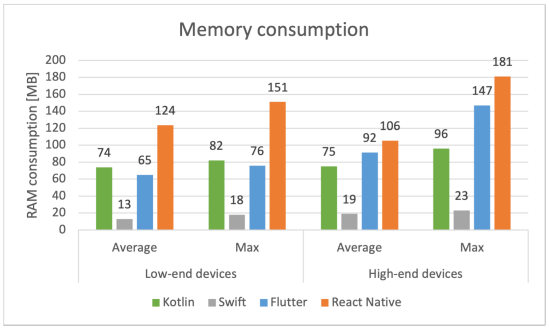

Fig. 8.19. Research scenario 1: Memory consumption (Source: Own work)

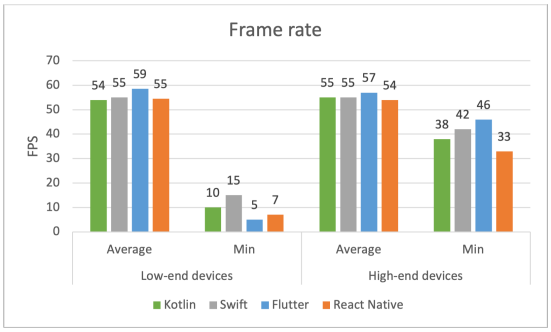

Fig. 8.20. Research scenario 1: Frame rate (Source: Own work)

Figure 8.19 shows the comparison of memory consumption among Android and iOS apps developed with Kotlin, Swift, Flutter, and React Native. It can be observed that Swift apps demonstrate the lowest memory consumption on all types of devices compared to other technologies. Flutter apps exhibit similar memory usage, with the latter experiencing higher spikes on high-end devices. React Native apps, on the other hand, perform moderately worse on low-end devices, slightly worse on high-end devices and suffer from bigger spikes.

Figure 8.20 show the comparison of frame rate among Android and iOS apps developed with Kotlin, Swift, Flutter, and React Native. In general, none of the technologies suffer from low frame rate on average. However, all of them experience sporadic FPS drops to as little as 5 FPS on low-end devices. Flutter seems to offer the highest overall smoothness with the best result on average and the highest minimum of 46 FPS on high-end devices.

8.1.3. Research scenario 2 results analysis

The following figures illustrate the aggregated results from the experiments conducted within Research scenario 2 described in Chapter 5.2.

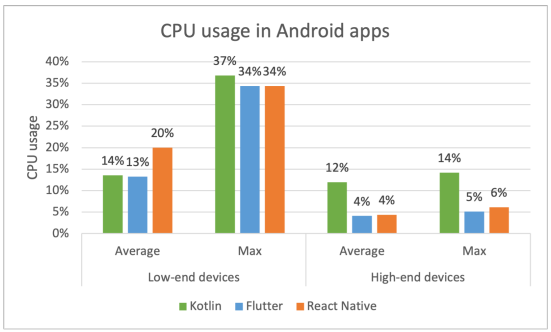

Fig. 8.21. Research scenario 2: CPU usage in Android apps (Source: Own work)

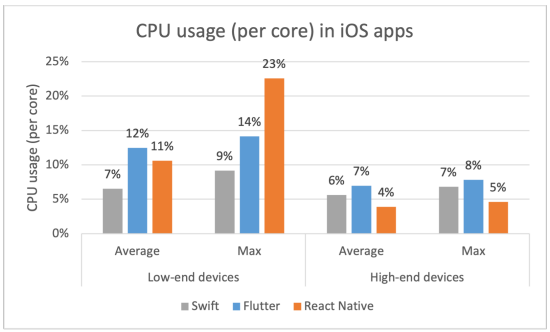

Fig. 8.22. Research scenario 2: CPU usage in iOS apps (Source: Own work)

Figures 8.21 and 8.22 show the comparison of CPU usage among Android and iOS apps developed with Kotlin, Swift, Flutter, and React Native. In the case of Android apps, on lower-end devices, Kotlin and Flutter apps perform similarly, while React Native apps require slightly more CPU resources. However, on high-end devices, Flutter and React Native apps exhibit very low CPU usage and visibly outperform Kotlin apps. Swift apps running on low-end devices require the least CPU capacity, 7% on average. Flutter and React Native apps exhibit similar CPU loads; however, the latter experience higher spikes. On high-end devices, all three technologies offer excellent performance while keeping CPU load within the threshold of 4–8%.

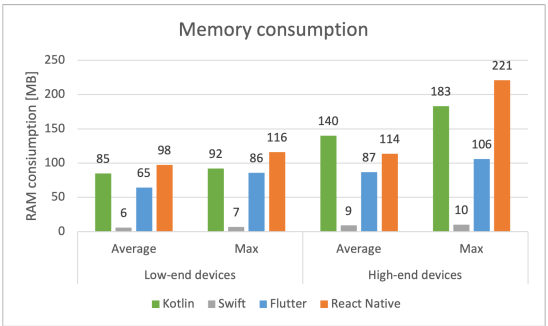

Fig. 8.23. Research scenario 2: Memory consumption (Source: Own work)

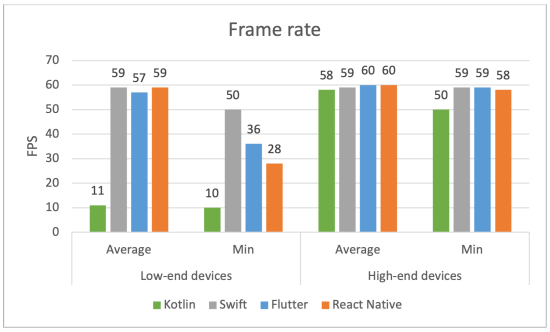

Fig. 8.24. Research scenario 2: Frame rate (Source: Own work)

Figure 8.23 shows the comparison of memory consumption among Android and iOS apps developed with Kotlin, Swift, Flutter, and React Native. Swift apps again demonstrate extremely low memory usage compared to other technologies and do not experience any spikes. Flutter apps slightly outperform Kotlin and React Native apps. Kotlin and React Native apps results are rather comparable. The former utilize less memory on low-end devices, and the latter require less memory on high-end devices, although they still experience higher spikes.

Figure 8.24 shows the comparison of frame rate among Android and iOS apps developed with Kotlin, Swift, Flutter, and React Native. On high-end devices, each technology exhibits similar performance, maintaining 58–FPS on average. Kotlin apps experience the biggest FPS drops; however, a minimum of 50 FPS is still not a concerning result. On low-end devices, Swift, Flutter, and React Native offer the same great performance, while Kotlin apps seem to suffer from a very low average frame rate of 11%. Such a result corresponds to users experiencing visible stutters and lags.

8.1.4. Research scenario 3 results analysis

The following figures illustrate the aggregated results from the experiments conducted within Research scenario 3 described in Chapter 5.2.

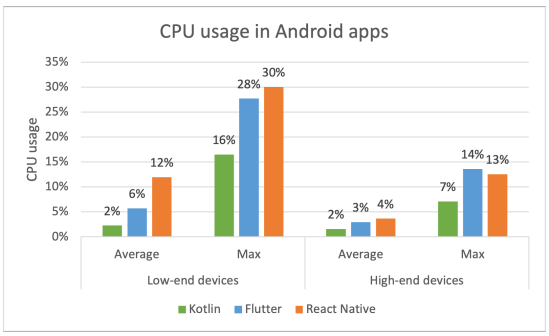

Fig. 8.25. Research scenario 3: CPU usage in Android apps (Source: Own work)

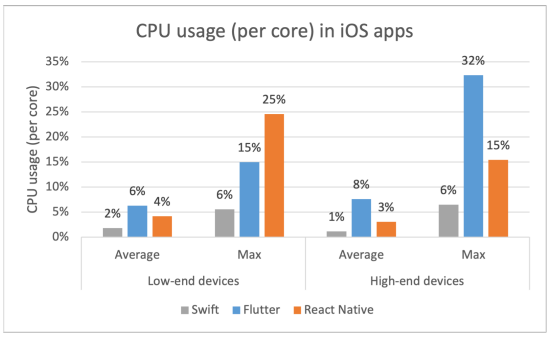

Fig. 8.26. Research scenario 3: CPU usage in iOS apps (Source: Own work)

Figures 8.25 and 8.26 show the comparison of CPU usage among Android and iOS apps developed with Kotlin, Swift, Flutter, and React Native. On low-end and high-end Android devices, Kotlin apps provide the best performance at 2% CPU load, followed by Flutter apps at 3–6% and finally React Native apps at 4-12%. When it comes to maximums reached, Flutter and React Native are again outperformed by Kotlin, but the values themselves are still relatively low. In the case of iOS, Swift apps show the lowest CPU load as well as no spikes on both low-end and high-end devices. React Native apps demonstrate slightly lower CPU usage than Flutter apps, although they experience higher spikes on low-end devices.

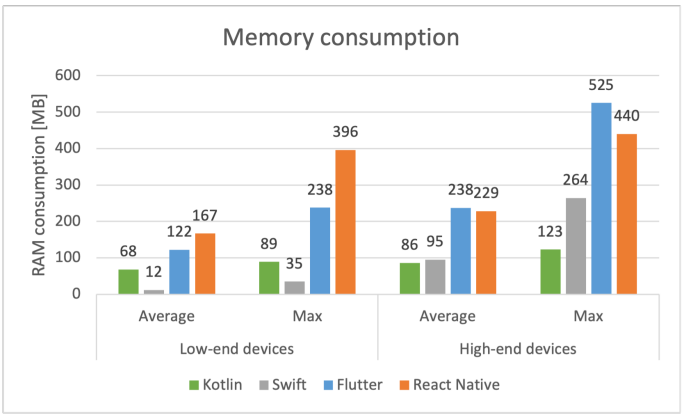

Fig. 8.27. Research scenario 3: Memory consumption (Source: Own work)

Figure 8.27 shows the comparison of memory consumption among Android and iOS apps developed with Kotlin, Swift, Flutter, and React Native. On low-end devices, Swift apps heavily outperform other technologies; however, on high-end devices, Kotlin apps exhibit the lowest memory usage. Flutter and React Native apps demonstrate similar results.

The former utilizes over 30% less memory on low-end devices but about 4% more on high-end devices. Analogously, Flutter apps experience smaller spikes on low-end devices and bigger spikes on high-end devices, reaching a maximum of over 500 MB.

8.1.5. Research scenario 4 results analysis

The following figures illustrate the aggregated results from the experiments conducted within Research scenario 4 described in Chapter 5.2.

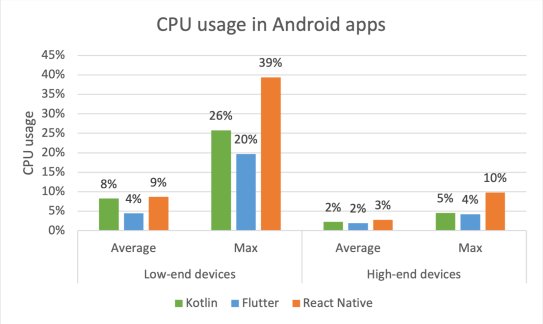

Fig. 8.28. Research scenario 4: CPU usage in Android apps (Source: Own work)

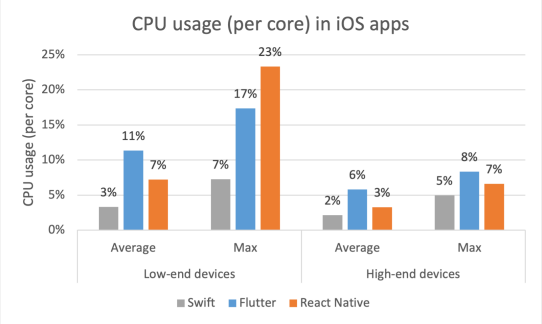

Fig. 8.29. Research scenario 4: CPU usage in iOS apps (Source: Own work)

Figures 8.28 and 8.29 show the comparison of CPU usage among Android and iOS apps developed with Kotlin, Swift, Flutter, and React Native. On both platforms, the three technologies considered demonstrate similar CPU usage within the threshold of 2-10% on high-end devices. On low-end Android devices, Flutter requires the least CPU capacity, while Kotlin and React Native achieve similar results, although React Native experiences higher spikes. Apps written in Swift exhibit the lowest CPU load on lower-end iOS devices. On average, React Native apps utilize less CPU resources than Flutter; however, they suffer from lower stability with spikes of 23%.

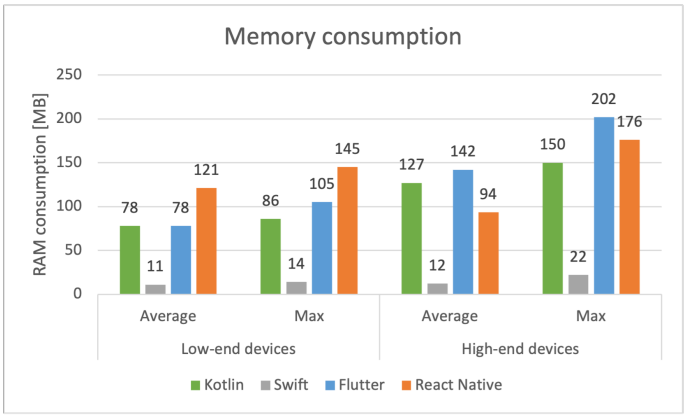

Fig. 8.30. Research scenario 4: Memory consumption (Source: Own work)

Figure 8.30 shows the comparison of memory consumption among Android and iOS apps developed with Kotlin, Swift, Flutter, and React Native. It can be observed that Swift apps utilize significantly less memory than other technologies, maintaining consumption within the threshold of 11–22 MB. Kotlin and Flutter apps achieve similar performance on both low-end and high-end devices. On the other hand, React Native apps exhibit higher RAM usage on low-end devices but lower usage on high-end devices.

8.2. CONCLUSIONS

According to the literature review combined with the conducted experiments, cross--platform development is a strong alternative to the native approach. The ability to utilize a single code base and deploy the application on more than one platform seems very efficient from the perspective of resource management, in both financial and timescale aspects. The concerns connected to cross-platform technologies come from their assumed inferiority to their native equivalents in the context of app performance and, thus, user experience, which is currently a top priority in mobile development. Nevertheless, the results from the experiments carried out in this thesis seem to at least partly invalidate those concerns. For example, even though native apps usually demonstrate lower memory usage on average, in many cases, cross-platform apps (especially those implemented with Flutter) achieve better results in the context of CPU usage and power consumption. When it comes to the frame rate, directly corresponding to the perceived smoothness, both native and cross-platform solutions performed similarly, except for Kotlin apps suffering from FPS drops on low-end devices.

Table 8.1 contains an overview of the best-performing development technologies grouped by research scenarios, performance metrics, target platforms, and device types. Each technology is represented by its abbreviation: Kotlin – K, Swift – S, Flutter – F, and React Native – RN. According to the information this table provides, the process of selecting a development method should become easier. One should consider the functionalities that must be implemented and compare them with the research scenarios considered in this thesis. Research scenario 1 related to one of the most common mobile application elements, which is the scrollable and filterable list. Research scenario 2 related to animations, which are responsible for a big part of user perception of a mobile application. Research scenario 3 related to file input and output, which is a very common feature found in many mobile applications. Research scenario 4 related to the functionality of user-controlled elements, e.g., switches and text fields, as well as the navigation between mobile application pages.

Table 8.1: Best performing development solutions (Source: Own work)

Variable			**Scenario 1**	**Scenario 2**	**Scenario 3**	**Scenario 4**
CPU	Android	Low-end	F	K, F	K	F
		High-end	F, K	F, RN	K	K, F
	iOS	Low-end	S, F	S	S	S
		High-end	S, F	RN	S	S
RAM	Android	Low-end	F	F	K	K, F
		High-end	K	F	K	RN
	iOS	Low-end	S	S	S	S
		High-end	S	S	S	S
FPS	Android	Low-end	F	F, RN	–	–
		High-end	F	F, RN	–	–
	iOS	Low-end	F	S	–	–
		High-end	F	S, F, RN	–	–

The research carried out for the purpose of this master's thesis leads to the following conclusions:

— On average, Flutter provides the best performance out of all the technologies considered.
— Kotlin and Flutter apps require similar CPU usage, mutually outperforming each other in different scenarios.
— In most cases, Kotlin utilizes the least device memory in Android apps.
— Flutter seems to offer the best frame rate results with a high degree of stability, which may be caused by the usage of a new rendering engine *Impeller*.
— Flutter applications exhibit the highest memory consumption, which may be an issue when considering low-end devices.
— In the considered scenarios, React Native offers the lowest overall performance.
— Swift offers great performance in every aspect; therefore, if a mobile application does not have to be multi-platform, it is a great choice, especially considering the excellent memory consumption levels.

9. SUMMARY

In multiple literature positions, statements can be found about mobile app performance being significantly higher when developed using a native approach compared to a cross-platform approach. However, even in many rather recent papers, this statement is assumed based on the work of P. Que, X. Guo and M. Zhu ([62]) which was published in 2016. Over the past 7 years, there have been many breaking changes to both native and multi-platform solutions. It is reasonable to assume these changes may have affected those findings. Therefore, it seemed worthwhile to review the current state of the art and draw present-day conclusions.

The purpose of this master's thesis was to carry out research on the performance of mobile applications implemented with cross-platform solutions in the form of Flutter and React Native as compared to native approaches in the form of Jetpack Compose and SwiftUI. These goals have been successfully fulfilled by conducting multiple experiments and drawing relevant conclusions based on their outcomes. The experiments have been prepared according to an exhaustive literature review and covered a diverse set of devices in order to yield valuable results. Based on the obtained knowledge, the technologies have been compared in the context of their optimal scenarios of use and overall advantages and disadvantages. The described findings can be applied in real life in the process of choosing the development approach for mobile application building.

9.1. CONTRIBUTIONS

The successful completion of this master's thesis brought a number of contributions in the context of mobile application development.

Various aspects of the leading mobile technologies were described in order to update the state of the art. For example, the new rendering engine *Impeller* had just been released during the period of research for this thesis, and therefore the performance described in the previous literature potentially became out-dated.

The experiment results provided support for the decision-making problem of choosing an optimal development method for a specific mobile application. Considered technologies were compared in the context of app performance; however, some information from the perspective of a developer was acquired and described during the sample apps' implementation process, which can be used as a secondary input when selecting the development solution.

According to the findings acquired within this thesis, the assumption that applications implemented with cross-platform solutions experience performance issues compared to their native equivalents is no longer correct.

9.2. LIMITATIONS

There are two main limitations concerning this thesis that may affect the acquired results to some extent: the iOS testing method and the device limit.

The former is caused by the lack of access to *Apple Developer Program* which is a paid membership ($99) enabling complete iOS app testing and deployment. It is not possible to create an App Store bundle (*.ipa* file) without being subscribed to the membership. Therefore, for the purpose of this thesis, sample apps have been installed on the device directly from Xcode in profile mode. Arguably, this may have some influence on the results of testing; however, it cannot be confirmed without access to the program.

The other limitation is the device limit. The experiments have been performed using four smartphones, two per operating system. Although they have been selected in a way to provide diversified results by choosing old and new devices with different operating system versions, in order to achieve the most reliable results, even more devices could be used. This would not only cover a wider range of operating system versions and hardware specifications, but could also cover the missing mobile device type: tablets.

9.3. SUGGESTIONS FOR FUTURE WORK

The research conducted for the purpose of this master's thesis could definitely be extended. There are different improvement directions that could be taken.

First and foremost, the limitations described in the previous chapter could be eliminated. The experiments concerning iOS applications could be repeated with access to *Apple Developer Program*. The number of devices could be increased to cover more operating systems and device types.

Furthermore, some other cross-platform frameworks could be included in the analysis, e.g., *Ionic* or *Kotlin Multiplatform*. Especially the second one should yield worthwhile results considering it is still a novelty accessible only in Beta version.

Another aspect of the research that could be enhanced is the choice of sample applications. For the most valuable outcome, "complete (production-level)" apps would have to be implemented, thus providing a true real-life comparison.

Moreover, the scope of this thesis is limited to the mobile platform. However, many cross-platform frameworks cover more platforms. The experiment environment could be extended by considering the web and/or desktop apps (Windows, macOS, or Linux). Of

course, such a task carries a large volume of additional implementation work because each platform requires the development of a native equivalent to the cross-platform solution.

Finally, even without performing any modifications to the scope of the research proposed in this thesis, the experiments could be conducted again after further breaking changes to the included technologies, which will surely occur considering the continuous growth of cross-platform development.

BIBLIOGRAPHY

[1] Adetunji, O., Ajaegbu, C., Omotosho, O.J., *Dawning of Progressive Web Applications (PWA): Edging Out the Pitfalls of Traditional Mobile Development*, American Scientific Research Journal for Engineering, Technology, and Sciences. 2020, Volume 68, pages 85–99.

[2] Android, `https://www.android.com/what-is-android/`. Accessed 24 Apr. 2023.

[3] Android Developers, `https://developer.android.com/guide/platform`. Accessed 24 Apr. 2023.

[4] Android Developers, `https://developer.android.com/studio/intro`. Accessed 25 Apr. 2023.

[5] Android Developers, `https://developer.android.com/kotlin`. Accessed 25 Apr. 2023.

[6] Android Developers, *Android's Kotlin-first approach*, `https://developer.android.com/kotlin/first`. Accessed 25 Apr. 2023.

[7] Android Developers, *Api desugaring supporting Android 13 and java.nio*, `https://android-developers.googleblog.com/2023/02/api-desugaring-supporting-android-13-and-java-nio.html`. Accessed 26 Apr. 2023.

[8] Android Developers Blog, *Jetpack compose is now 1.0: announcing android's modern toolkit for building native ui*, `https://android-developers.googleblog.com/2021/07/jetpack-compose-announcement.html`. Accessed 27 Apr. 2023.

[9] Angulo, E., Ferre, X., *A case study on cross-platform development frameworks for mobile applications and UX*, Interacción '14: Proceedings of the XV International Conference on Human Computer Interaction. 2014, pages 1–8.

[10] AppDynamics, *Mobile app performance explained*, `https://www.appdynamics.com/media/uploaded-files/mobileapp.pdf`. Accessed 23 Apr. 2023.

[11] Apple Developer, `https://developer.apple.com/documentation/xcode/`. Accessed 6 May 2023.

[12] Apple Developer, `https://developer.apple.com/swift/`. Accessed 6 May 2023.

[13] Apple Developer, `https://developer.apple.com/documentation/swiftui/`. Accessed 6 May 2023.

[14] Apple Developer, *Human Interface Guidelines*, `https://developer.apple.com/design/human-interface-guidelines`. Accessed 7 May 2023.

[15] AppSamurai, *Mobile app performance metrics for crash-free apps*, `https://appsamurai.com/blog/mobile-app-performance-metrics-for-crash-free-apps/`. Accessed 28 May 2023.

[16] Białkowski, D., Smołka, J., *Evaluation of flutter framework time efficiency in context of user interface tasks*, Journal of Computer Sciences Institute. 2022, Volume 25, pages 309–314.

[17] Biørn-Hansen, A., Grønli, T.M., *Baseline requirements for comparative research on cross-platform mobile development*, Norwegian Informatics Conference. 2017.

[18] Biørn-Hansen, A., Grønli, T.M., Ghinea, G., *A survey and taxonomy of core concepts and research challenges in cross-platform mobile development*, ACM Computing Surveys. 2018, Volume 51, pages 1–34.

[19] BrainHub, *Web app vs mobile app - which to develop first?*, `https://brainhub.eu/library/app-vs-website-which-to-develop-first`. Accessed 7 May 2023.

[20] Carney, T., *What's a XIB and why would i ever use one?*, `https://medium.com/@tjcarney89/whats-a-xib-and-why-would-i-ever-use-one-58d608cd5e9b`. Accessed 6 May 2023.

[21] Competition Markets Authority, *Mobile ecosystems. Market study final report.* Updated 4 Aug. 2022.

[22] Crha, J., *Comparison of Technologies for Multiplatform Mobile Applications Development*, Master's thesis, Masaryk University. 2021.

[23] Denko, B., Pecnik, S., Fister jr, I., *A comprehensive comparison of hybrid mobile application development frameworks*, International Journal of Security and Privacy in Pervasive Computing. 2021, Volume 13, pages 78–90.

[24] Fentaw, A.E., *Cross platform mobile application development: a comparison study of React Native Vs Flutter*, Master's thesis, University of Jyväskylä. 2020.

[25] Flutter, `https://flutter.dev/showcase`. Accessed 15 May 2023.

[26] Flutter, *Flutter architectural overview*, `https://docs.flutter.dev/resources/architectural-overview`. Accessed 15 May 2023.

[27] Flutter, *Impeller rendering engine*, `https://docs.flutter.dev/perf/impeller`. Accessed 15 May 2023.

[28] Flutter Campus, `https://www.fluttercampus.com/guide/110/how-to-show-material-alert-and-cupertino-dialog-in-flutter/`. Accessed 16 May 2023.

[29] GameBench, `https://docs.gamebench.net/web-dashboard/the-cpu-pane/`. Accessed 21 June 2023.

[30] GameBench, `https://docs.gamebench.net/web-dashboard/the-memory-pane/`. Accessed 21 June 2023.

[31] GameBench, `https://docs.gamebench.net/web-dashboard/the-battery-pane/`. Accessed 21 June 2023.

[32] GameBench, `https://docs.gamebench.net/web-dashboard/the-performance-pane/`. Accessed 21 June 2023.

[33] García, C.G., Espada, J.P., García-Bustelo, B.P., Lovelle, J.M.C., *Swift vs. objective-c: A new programming language*, International Journal of Artificial Intelligence and Interactive Multimedia. 2015, Volume 3, pages 74–81.

[34] GeeksForGeeks, `https://www.geeksforgeeks.org/difference-between-swift-vs-objective-c/`. Accessed 6 May 2023.

[35] Google, *Micro-moments: Your guide to winning the shift to mobile*, `https://`

www.thinkwithgoogle.com/marketing-strategies/micro-moments/micromoments-guide/. Accessed 26 Mar. 2023.

[36] Google Trends, https://trends.google.pl/trends/explore?date=2008-01-01%202023-03-26&q=cross%20platform%20app%20development&hl=pl. Accessed 26 Mar. 2023.

[37] Google Trends, https://trends.google.pl/trends/explore?date=2008-01-01%202023-03-26&q=ionic,flutter,react%20native,xamarin,cordova&hl=pl. Accessed 26 Mar. 2023.

[38] Grand View Research, *Global Mobile Application Market Size, Share, & Trends Analysis Report by Store Type (Google Store, Apple Store, Others), by Application, by Region, and Segment Forecasts, 2022-2030*. 2022.

[39] Harsh, K., *The complete guide to React Native for Web*, https://blog.logrocket.com/complete-guide-react-native-web/. Accessed 18 May 2023.

[40] Hjort, E., *Evaluation of React Native and Flutter for cross-platform mobile application development*, Master's thesis, Åbo Akademi University. 2020.

[41] Hort, M., Kechagia, M., Sarro, F., Harman, M., *A survey of performance optimization for mobile applications*, IEEE Transactions on Software Engineering. 2022, Volume 48, 8, pages 2879–2904.

[42] Karim, W., *How is Flutter different from Native, Web-view, and other cross-platform frameworks*, https://wajahatkarim.com/2019/11/how-is-flutter-different-from-native-web-view-and-other-cross-platform-frameworks/. Accessed 13 May 2023.

[43] Khandelwal, A., *12 Best Load Testing tools for mobile Applications | What is Load testing*, https://testinggenez.com/load-testing-tools-for-mobile-applications/. Accessed 23 May 2023.

[44] Kocki, M., Urban, M., Kopniak, P., *Comparison of hybrid and native iOS mobile application development technologies*, Journal of Computer Sciences Institute. 2022, Volume 25, pages 280–287.

[45] Łukasz Kosiński, *Flutter vs React Native vs Qt in 2022*, https://scythe-studio.com/en/blog/flutter-vs-react-native-vs-qt-in-2022. Accessed 15 May 2023.

[46] Kotlin, https://kotlinlang.org/docs/multiplatform.html. Accessed 26 Apr. 2023.

[47] Kotlin, *The six most popular cross-platform app development frameworks*, https://kotlinlang.org/docs/cross-platform-frameworks.html. Accessed 18 Apr. 2023.

[48] Krusche Company, *Kotlin vs Java: strengths, weaknesses and when to use which*, https://kruschecompany.com/kotlin-vs-java/. Accessed 26 Apr. 2023.

[49] Lachgar, M., Hanine, M., Benouda, H., Ommane, Y., *Decision Framework for Cross-Platform Mobile Development Frameworks Using an Integrated Multi-Criteria Decision-Making Methodology*, International Journal of Mobile Computing and Multimedia Communications. 2022, Volume 13, 1, pages 1–22.

[50] Material Design 3, https://m3.material.io/get-started. Accessed 28 Apr. 2023.

[51] Matijević, M., *React Native's upcoming re-architecture*, https://collectivemind.dev/blog/react-native-re-architecture. Accessed 18 May 2023.

[52] Mejia, R., *Declarative and imperative programming using SwiftUI and UIKit*, https://medium.com/@rmeji1/declarative-and-imperative-programming-using-swiftui-and-uikit-c91f1f104252. Accessed 6 May 2023.

[53] MGSM, https://www.mgsm.pl/pl/katalog/apple/iphone7/. Accessed 1 June 2023.

[54] MGSM, https://www.mgsm.pl/pl/katalog/apple/iphone13mini/. Accessed 1 June 2023.

[55] MGSM, https://www.mgsm.pl/pl/katalog/sony/xperiaz1c6903/. Accessed 1 June 2023.

[56] MGSM, https://www.mgsm.pl/pl/katalog/xiaomi/mi9tpro/. Accessed 31 May 2023.

[57] Microsoft, *What is mobile application development?*, https://azure.microsoft.com/en-us/resources/cloud-computing-dictionary/what-is-mobile-app-development/#definition. Accessed 22 Apr. 2023.

[58] Mota, D., Martinho, R., *An approach to assess the performance of mobile applications: A case study of multiplatform development frameworks*, Proceedings of the 16th International Conference on Evaluation of Novel Approaches to Software Engineering - Volume 1: ENASE,. 2021.

[59] Naveen, https://intellipaat.com/blog/tutorial/ios-tutorial/ios-architecture/?US. Accessed 5 May 2023.

[60] Okediran, O.O., Arulogun, O.T., Ganiyu, R.A., Oyeleye, C.A., *Mobile operating systems and application development platforms: A survey*, International Journal of Advanced Networking and Applications. 2014, Volume 6, pages 2195–2201.

[61] Olsson, M., *A Comparison of Performance and Looks Between Flutter and Native Applications*, Bachelor's thesis, Blekinge Institute of Technology. 2020.

[62] Que, P., Guo, X., Zhu, M., *A comprehensive comparison between hybrid and native app paradigms*, 2016 8th International Conference on Computational Intelligence and Communication Networks (CICN). 2016.

[63] Rahman, A., *A comparative study of hybrid mobile application development*, EasyChair Preprint no. 3905. EasyChair, 2020.

[64] React Native, https://reactnative.dev/showcase. Accessed 18 May 2023.

[65] React Native, https://reactnative.dev/architecture/fabric-renderer. Accessed 18 May 2023.

[66] React Native, https://reactnative.dev/docs/the-new-architecture/pillars-turbomodules. Accessed 18 May 2023.

[67] React Native, https://reactnative.dev/docs/next/the-new-architecture/pillars-codegen. Accessed 18 May 2023.

[68] React Native, https://reactnative.dev/architecture/render-pipeline. Accessed 18 May 2023.

[69] React Native, https://reactnative.dev/architecture/view-flattening. Accessed 18 May 2023.

[70] React Native, https://reactnative.dev/architecture/threading-model. Accessed 18 May 2023.

[71] React Native, *Core Components and Native Components*, https://reactnative.dev/docs/intro-react-native-components. Accessed 18 May 2023.

[72] React Native, *Why a New Architecture*, https://reactnative.dev/docs/the-new-architecture/why. Accessed 18 May 2023.

[73] React Native for Web, https://necolas.github.io/react-native-web/docs/. Accessed 18 May 2023.

[74] Rieger, C., Majchrzak, T.A., *Towards the definitive evaluation framework for cross-platform app development approaches*, Journal of Systems and Software. 2019, Volume 153, C, pages 175–199.

[75] Saborido Infantes, R., Khomh, F., Hindle, A., Alba, E., *An App Performance Optimization Advisor for Mobile Device App Marketplaces*, Sustainable Computing: Informatics and Systems. 2018, Volume 19.

[76] Shaheen, J.A., Asghar, M.A., Hussain, A., *Android os with its architecture and android application with dalvik virtual machine review*, International Journal of Multimedia and Ubiquitous Engineering. 2017, Volume 12, 7.

[77] Singh, H., *SPEED PERFORMANCE BETWEEN SWIFT AND OBJECTIVE-C*, International Journal of Engineering Applied Sciences and Technology. 2016, Volume 1(10), pages 185–189.

[78] Singh, M., Shobha, G., *Comparative analysis of hybrid mobile app development frameworks*, International Journal of Soft Computing and Engineering (IJSCE). 2021, Volume 10, 6.

[79] Sinicki, A., *Implementing Material Design components and guidelines - Googlify your app!*, https://www.androidauthority.com/material-design-components-1177515/. Accessed 28 Apr. 2023.

[80] Smartlook, *The top 16 most important mobile app KPIs to measure performance [updated]*, https://www.smartlook.com/blog/top-15-most-important-mobile-app-kpis-to-measure-the-performance/. Accessed 29 May 2023.

[81] Statcounter, *Mobile operating system market share worldwide*, https://gs.statcounter.com/os-market-share/mobile/worldwide/#monthly-202303-202303-bar. Accessed 23 Apr. 2023.

[82] Statcounter, *Mobile tablet android version market share worldwide*, https://gs.statcounter.com/android-version-market-share/mobile-tablet/worldwide/#monthly-202303-202303-bar. Accessed 23 Apr. 2023.

[83] Statcounter, *Mobile tablet ios version market share worldwide*, https://gs.statcounter.com/ios-version-market-share/mobile-tablet/worldwide/#monthly-202303-202303-bar. Accessed 23 Apr. 2023.

[84] Statista, *Forecast number of mobile users worldwide 2020-2025*, https://www.statista.com/statistics/218984/number-of-global-mobile-users-since-2010/. Accessed 18 Apr. 2023.

[85] TechJury, *55+ jaw dropping app usage statistics in 2023*, https://techjury.net/blog/app-usage-statistics/#gref. Accessed 18 Apr. 2023.

[86] Upwork, https://www.upwork.com/resources/swift-vs-objective-c-a-look-at-ios-programming-languages. Accessed 6 May 2023.

[87] Velvetech, *5 key mobile development approaches*, https://www.velvetech.com/blog/5-key-mobile-development-approaches/. Accessed 22 Apr. 2023.

[88] Vilček, T., Jakopec, T., *Comparative analysis of tools for development of native and hybrid mobile applications*, 2017 40th International Convention on Information and Communication Technology, Electronics and Microelectronics (MIPRO). 2017.

[89] Wikipedia, https://en.wikipedia.org/wiki/Objective-C. Accessed 6 May 2023.

[90] Willocx, M., Vossaert, J., Naessens, V., *A quantitative assessment of performance in mobile app development tools*, 2015 IEEE International Conference on Mobile Services. 2015.

[91] Windmill, E., *Flutter in Action* (Manning, 2020).

[92] Yiğit, M., *Say hello to Jetpack Compose and compare with XML*, https://blog.kotlin-academy.com/say-hello-to-jetpack-compose-and-compare-with-xml-6bc6053aec13. Accessed 27 Apr. 2023.

LIST OF FIGURES

3.1. Mobile OS market (Source: Own work based on [81]) 9
3.2. iOS version market share (Source: Own work based on [83]) 9
3.3. Android version market share (Source: Own work based on [82]) 9
3.4. Android architecture (Source: Own work based on [3]) 10
3.5. iOS architecture (Source: Own work based on [59]) 13
3.6. Google Trends search popularity of cross-platform app development (Source: Own work based on [36]) 16
3.7. Google Trends search popularity of selected cross-platform frameworks (Source: Own work based on [37]) 17
3.8. Hybrid app architecture (Source: [42]) 18
3.9. Interpreted app architecture (Source: [42]) 18
3.10. Cross-compiled approach workflow (Source: [18]) 19
3.11. Progressive Web App architecture (Source: [1]) 19
3.12. Flutter architecture (Source: [26]) 20
3.13. Flutter web-adapted architecture (Source: [26]) 21
3.14. Flutter rendering flow (Source: [26]) 22
3.15. Flutter Widget, Element, Render trees (Source: [26]) 22
3.16. Flutter Material dialog (Source: [28]) 23
3.17. Flutter Cupertino dialog (Source: [28]) 23
3.18. React Native previous achitecture (pre-v0.68) (Source: [51]) 24
3.19. React Native New Architecture (v0.68+) (Source: [51]) 24
3.20. React Native rendering flow (Source: [68]) 26
3.21. React Native React Element, React Shadow, Host View trees (Source: [68]) 26

6.1. App 1: Kotlin (Source: Own work) 35
6.2. App 1: Swift (Source: Own work) 35
6.3. App 1: Flutter Android (Source: Own work) 35
6.4. App 1: Flutter iOS (Source: Own work) 35
6.5. App 1: React Native Android (Source: Own work) 36
6.6. App 1: React Native iOS (Source: Own work) 36
6.7. App 2: Kotlin (Source: Own work) 36
6.8. App 2: Swift (Source: Own work) 36
6.9. App 2: Flutter Android (Source: Own work) 36
6.10. App 2: Flutter iOS (Source: Own work) 36
6.11. App 1: React Native Android (Source: Own work) 37

6.12. App 1: React Native iOS (Source: Own work) 37
6.13. App 3: Kotlin (Source: Own work) 37
6.14. App 3: Swift (Source: Own work) 37
6.15. App 3: Flutter Android (Source: Own work) 37
6.16. App 3: Flutter iOS (Source: Own work) 37
6.17. App 3: React Native Android (Source: Own work) 38
6.18. App 3: React Native iOS (Source: Own work) 38
6.19. App 4 (1/3): Kotlin (Source: Own work) 38
6.20. App 4 (2/3): Kotlin (Source: Own work) 38
6.21. App 4 (3/3): Kotlin (Source: Own work) 38
6.22. App 4 (1/3): Swift (Source: Own work) 38
6.23. App 4 (2/3): Swift (Source: Own work) 38
6.24. App 4 (3/3): Swift (Source: Own work) 38
6.25. App 4 (1/3): Flutter Android (Source: Own work) 39
6.26. App 4 (2/3): Flutter Android (Source: Own work) 39
6.27. App 4 (3/3): Flutter Android (Source: Own work) 39
6.28. App 4 (1/3): Flutter iOS (Source: Own work) 39
6.29. App 4 (2/3): Flutter iOS (Source: Own work) 39
6.30. App 4 (3/3): Flutter iOS (Source: Own work) 39
6.31. App 4 (1/3): React Native Android (Source: Own work) 39
6.32. App 4 (2/3): React Native Android (Source: Own work) 39
6.33. App 4 (3/3): React Native Android (Source: Own work) 39
6.34. App 4 (1/3): React Native iOS (Source: Own work) 40
6.35. App 4 (2/3): React Native iOS (Source: Own work) 40
6.36. App 4 (3/3): React Native iOS (Source: Own work) 40

8.1. Average CPU usage in Android apps (Source: Own work) 49
8.2. Maximum CPU usage in Android apps (Source: Own work) 49
8.3. Average CPU usage in iOS apps (Source: Own work) 50
8.4. Maximum CPU usage in iOS apps (Source: Own work) 50
8.5. Average memory consumption in Android apps (Source: Own work) 50
8.6. Maximum memory consumption in Android apps (Source: Own work) 50
8.7. Average memory consumption in iOS apps (Source: Own work) 51
8.8. Maximum memory consumption in iOS apps (Source: Own work) 51
8.9. Power consumption in Android apps (Source: Own work) 51
8.10. Power consumption in iOS apps (Source: Own work) 51
8.11. Average frame rate in Android apps (Source: Own work) 52
8.12. Minimum frame rate in Android apps (Source: Own work) 52
8.13. Average frame rate in iOS apps (Source: Own work) 52
8.14. Minimum frame rate in iOS apps (Source: Own work) 52
8.15. Frame rate stability in Android apps (Source: Own work) 53
8.16. Frame rate stability in iOS apps (Source: Own work) 53

8.17. Research scenario 1: CPU usage in Android apps (Source: Own work) 53
8.18. Research scenario 1: CPU usage in iOS apps (Source: Own work) 53
8.19. Research scenario 1: Memory consumption (Source: Own work) 54
8.20. Research scenario 1: Frame rate (Source: Own work) 54
8.21. Research scenario 2: CPU usage in Android apps (Source: Own work) 54
8.22. Research scenario 2: CPU usage in iOS apps (Source: Own work) 54
8.23. Research scenario 2: Memory consumption (Source: Own work) 55
8.24. Research scenario 2: Frame rate (Source: Own work) 55
8.25. Research scenario 3: CPU usage in Android apps (Source: Own work) 56
8.26. Research scenario 3: CPU usage in iOS apps (Source: Own work) 56
8.27. Research scenario 3: Memory consumption (Source: Own work) 56
8.28. Research scenario 4: CPU usage in Android apps (Source: Own work) 57
8.29. Research scenario 4: CPU usage in iOS apps (Source: Own work) 57
8.30. Research scenario 4: Memory consumption (Source: Own work) 57

LIST OF TABLES

2.1 Related work (Source: Own work) 5

3.1. Java and Kotlin comparison (Source: Own work based on [6]) 11
3.2. React Native architecture improvements (Source: Own work based on [72]) 25
3.3. Cross-platform framework comparison (Source: Own work) 27
3.4. Cross-platform framework evaluation criteria (Source: Own work based on [74]) . . 28

4.1 Selected app performance metrics from the perspective of user experience (Source: Own work based on [10, 75, 90]) 29
4.2 Selected app performance metrics from the perspective of publisher (Source: Own work based on [15, 43, 80]) 30

5.1 Testing devices (Source: Own work based on [53, 54, 55, 56]) 34

7.1 Research scenario 1 results: Kotlin (Source: Own work) 42
7.2 Research scenario 1 results: Swift (Source: Own work) 43
7.3 Research scenario 1 results: Flutter (Source: Own work) 43
7.4 Research scenario 1 results: React Native (Source: Own work) 43
7.5 Research scenario 2 results: Kotlin (Source: Own work) 44
7.6 Research scenario 2 results: Swift (Source: Own work) 44
7.7 Research scenario 2 results: Flutter (Source: Own work) 45
7.8 Research scenario 2 results: React Native (Source: Own work) 45
7.9 Research scenario 3 results: Kotlin (Source: Own work) 46
7.10 Research scenario 3 results: Swift (Source: Own work) 46
7.11 Research scenario 3 results: Flutter (Source: Own work) 46
7.12 Research scenario 3 results: React Native (Source: Own work) 47
7.13 Research scenario 4 results: Kotlin (Source: Own work) 47
7.14 Research scenario 4 results: Swift (Source: Own work) 48
7.15 Research scenario 4 results: Flutter (Source: Own work) 48
7.16 Research scenario 4 results: React Native (Source: Own work) 48

8.1 Best performing development solutions (Source: Own work) 59