Identifying Trade-offs Associated with Cross-platform Mobile Development Tools

A thesis submitted to the
College of Graduate and Postdoctoral Studies
in partial fulfillment of the requirements
for the degree of Master of Science
in the Department of Computer Science
University of Saskatchewan
Saskatoon

By

Iman Jamali

©Iman Jamali, May 2022. All rights reserved.
Unless otherwise noted, copyright of the material in this thesis belongs to the author.

Permission to Use

In presenting this thesis in partial fulfillment of the requirements for a Postgraduate degree from the University of Saskatchewan, I agree that the Libraries of this University may make it freely available for inspection. I further agree that permission for copying of this thesis in any manner, in whole or in part, for scholarly purposes may be granted by the professor or professors who supervised my thesis work or, in their absence, by the Head of the Department or the Dean of the College in which my thesis work was done. It is understood that any copying or publication or use of this thesis or parts thereof for financial gain shall not be allowed without my written permission. It is also understood that due recognition shall be given to me and to the University of Saskatchewan in any scholarly use which may be made of any material in my thesis.

Disclaimer

Reference in this thesis to any specific commercial products, process, or service by trade name, trademark, manufacturer, or otherwise, does not constitute or imply its endorsement, recommendation, or favoring by the University of Saskatchewan. The views and opinions of the author expressed herein do not state or reflect those of the University of Saskatchewan, and shall not be used for advertising or product endorsement purposes.

Requests for permission to copy or to make other uses of materials in this thesis in whole or part should be addressed to:

Head of the Department of Computer Science
176 Thorvaldson Building
110 Science Place
University of Saskatchewan
Saskatoon, Saskatchewan
Canada
S7N 5C9

OR

Dean
College of Graduate and Postdoctoral Studies
University of Saskatchewan
116 Thorvaldson Building, 110 Science Place
Saskatoon, Saskatchewan S7N 5C9 Canada

Abstract

Native app development is still the primary development method on Android and iOS, the two major platforms for mobile devices. The development process for building an app for more than one platform with the native approach is slow and expensive. Although cross-platform mobile development tools (CPDTs for short) allow building apps for more than one platform using a single codebase, application of such tools has traditionally required considerable sacrifices in terms of performance, resource use and user experience. In recent years, a new generation of CPDTs has emerged that once again raise hopes of achieving the "write once, run everywhere" ideal. However, few studies have systematically assessed trade-offs associated with such recent CPDTs.

This thesis sought to identify some of the practical trade-offs between data-driven apps built by modern CPDTs when compared amongst themselves and against their native counterparts. The thesis established and applied a quantitative evaluation framework for investigation of the trade-offs between modern CPDTs with respect to responsiveness, convenience of use, resource use, performance, and developer experience. The study first identified three modern CPDTs: Flutter, React Native and PWA (Progressive Web Apps) based on perceived developer interest. The thesis then developed a three-fold evaluation system. For each evaluation component, an app was designed and developed for each baseline comparator (i.e., Android native and iOS native). This work then developed an app implementing the same design for each CPDT (i.e., Flutter, React Native and PWA). For that evaluation component, such evaluation-specific apps were then compared to one another. The first component of the evaluation investigated trade-offs associated with responsiveness and convenience of use, using measurements of three defined metrics: launch time, navigation time and size of the installer. The second evaluation component investigated trade-offs related to resource use and performance by implementing a set of resource-intensive tests to measure relevant metrics. The third and final component of the evaluation focused on trade-offs associated with the developer experience by measuring three metrics: lines of code, cyclomatic complexity, and build time across multi-page data-driven apps developed separately using each development method.

The responsiveness and convenience of use evaluation found some trade-offs imposed by CPDTs, with those being especially notable on the iOS platform. The resource use and performance evaluation results demonstrated that while native approaches retain an edge in resource use and performance, modern CPDTs are shrinking the gap — and, in some cases, outperforming native apps. The results of the developer experience evaluation showed that CPDTs could significantly improve the developer experience by requiring development and maintenance of just a single, shared body of source code that is less complex and requires fewer lines of code, in contrast to the per-platform sets of lower-level source code required in the native approach. This study conducted the first quantitative study investigating the trade-offs of modern CPDTs from the developer experience perspective, and provided empirical evidence regarding such trade-offs. Through the use of dedicated resource-intensive tests, this study also serves as one of the first quantitative studies

to provide insight on the resource use and performance overhead imposed by modern CPDTs. All of the artifacts developed in this work, including the multi-page data-driven apps and dedicated resource-intensive tests, are accessible in public open-source repositories.

Acknowledgements

First, I would like to thank Dr. Nathaniel D. Osgood, my supervisor, whose insight and knowledge steered me through this thesis. Dr. Osgood helped me find and follow my interests throughout my graduate program and has always guided me with his wealth of knowledge and experience. He gave me a chance to work on a variety of exciting projects. I am greatly honoured to work as a graduate student in Dr. Osgood's lab for the last few years and look forward to continuing to work with him in the future.

I thank my committee members, Drs. Carl Gutwin, Debajyoti Mondal, and Daniel Teng, for investing time and effort in evaluating my work.

I would also like to thank the Saskatchewan Prevention Institute, especially Noreen Agrey (Executive Director), for giving me an opportunity and sponsoring me to develop their Saskatchewan Pregnancy App as part of my Master's program.

I further thank my colleagues at Computational Epidemiology and Public Health Informatics Lab (CEPHIL), especially Wade McDonald, who kindly helped me with the statistical analyses of this work.

I would like to thank my wife, Hadis, for her unconditional support during the last year.

Finally, I thank my family and great friends Mehrdad and Kharazm for their help and encouragement.

Contents

Permission to Use . . . i

Abstract . . . ii

Acknowledgements . . . iv

Contents . . . v

List of Tables . . . viii

List of Figures . . . ix

List of Abbreviations . . . xi

1 Introduction . . . 1
1.1 Problem and Motivation . . . 1
1.2 Solution . . . 2
1.3 Steps to Solution . . . 2
1.4 Evaluation . . . 3
1.5 Contribution . . . 3
1.5.1 Scientific contributions . . . 3
1.6 Thesis Structure . . . 5

2 Background and Related Work . . . 6
2.1 Background . . . 6
2.1.1 Native Development . . . 6
2.1.2 Cross-platform . . . 6
2.1.3 React . . . 8
2.2 Related Work . . . 9
2.2.1 Criteria for CPDT selection . . . 9
2.2.2 Responsiveness, convenience of use, resource use and performance . . . 10
2.2.3 Developer experience . . . 12
2.3 Summary . . . 13

3 CPDT Selection . . . 15
3.1 Selection . . . 15
3.2 Flutter . . . 16
3.3 React Native . . . 18
3.4 Progressive Web Apps . . . 20
3.5 Summary . . . 21

4 Responsiveness and Convenience of Use . . . 22
4.1 Evaluation criteria and metrics . . . 22
4.1.1 Size of installer . . . 23
4.1.2 Navigation time . . . 24
4.2 Measurement tools and methods . . . 24
4.2.1 Launch time measurement . . . 25
4.2.2 Navigation time measurement . . . 27
4.2.3 Installer size measurement . . . 27
4.3 Device selection . . . 27

4.4 Design and implementation of apps . . . 27
4.5 Results . . . 29
4.5.1 Launch time . . . 30
4.5.2 Navigation time . . . 32
4.5.3 Size of installer . . . 35
4.6 Discussion . . . 35
4.6.1 Explanation and interpretation of results . . . 36
4.6.2 Overall results for each development method . . . 39
4.6.3 Overall results compared to previous work . . . 40
4.7 Summary . . . 40

5 Resource Use and Performance . . . 42
5.1 Evaluation criteria and metrics . . . 42
5.1.1 CPU usage . . . 43
5.1.2 Memory footprint . . . 43
5.1.3 Total data transferred . . . 44
5.1.4 Execution time . . . 44
5.2 Measurement tools and methods . . . 44
5.2.1 CPU usage measurement . . . 45
5.2.2 Memory footprint measurement . . . 45
5.2.3 Total data transferred . . . 46
5.2.4 Execution time . . . 46
5.3 Device selection . . . 46
5.4 Design and implementation of apps . . . 47
5.4.1 CPU-intensive Test . . . 48
5.4.2 Memory-intensive Test . . . 48
5.4.3 Network-intensive Test . . . 49
5.5 Results . . . 49
5.5.1 CPU-intensive test . . . 50
5.5.2 Memory-intensive test . . . 57
5.5.3 Network-intensive test . . . 63
5.6 Discussion . . . 71
5.6.1 Overall results for each development method . . . 76
5.6.2 Overall results compared to previous work . . . 78
5.7 Summary . . . 78

6 Developer Experience . . . 81
6.1 Evaluation criteria and metrics . . . 81
6.1.1 Lines of code (LOC) . . . 82
6.1.2 Cyclomatic complexity . . . 82
6.1.3 Build time . . . 83
6.2 Measurement tools and methods . . . 83
6.2.1 LOC . . . 84
6.2.2 Cyclomatic complexity . . . 84
6.2.3 Build time . . . 85
6.3 App design and implementation . . . 86
6.4 Results . . . 90
6.4.1 Lines of code . . . 91
6.4.2 Cyclomatic complexity . . . 91
6.4.3 Build time . . . 92
6.5 Discussion . . . 93
6.5.1 Explanation and interpretation of results . . . 93
6.5.2 Overall results for each development method . . . 97
6.5.3 Overall results compared to previous work . . . 98

6.6 Summary . . . 99

7 Conclusion . . . 101
7.1 Introduction . . . 101
7.2 Overall findings . . . 101
7.3 Contributions . . . 102
7.3.1 Scientific contributions . . . 102
7.4 Limitations . . . 103
7.5 Future work . . . 105

References . . . 107

Appendix A Statistical Analysis Code Listings of Responsiveness and Convenience of Use Experiments . . . 116

Appendix B Statistical Analysis Code Listings of Resource Use and Performance Experiments . . . 123

Appendix C Statistical Analysis Code Listing of Developer Experience Experiments . . . 149

Appendix D . . . 151

List of Tables

4.1 Selected devices for responsiveness and convenience of use evaluation (identical to Table 5.1) 28
4.2 Size of installer files (in MB) for different methods of development 35

5.1 Selected devices for resource use and performance evaluation (identical to Table 4.1) 47

6.1 Lines of code results of different methods of development 91
6.2 Cyclomatic complexity score for different methods of development 92

D.1 Lines of code and cyclomatic complexity results for the PWA app used in the experiments of Chapter 6 152
D.2 Lines of code and cyclomatic complexity results for the React Native app used in the experiments of Chapter 6 152
D.3 Lines of code and cyclomatic complexity results for the Flutter app used in the experiments of Chapter 6 152
D.4 Lines of code and cyclomatic complexity results for the native Android app used in the experiments of Chapter 6 152
D.5 Lines of code and cyclomatic complexity results for the native iOS app used in the experiments of Chapter 6 153
D.6 Web URL of the tools adopted to measure cyclomatic complexity of the apps in Chapter 6 . 153

List of Figures

3.1 Flutter architectural layers. Image from [67] 17
3.2 React Native architectural layers. Adapted from [79, 80] 19

4.1 Boxplot including median Android app launch time by development method by device. Dots indicate outliers. 30
4.2 Boxplot including median iOS app launch time by development method by device. Dots indicate outliers. 31
4.3 Boxplot including median Android navigation time by development method by device. Dots indicate outliers. 33
4.4 Boxplot including median iOS navigation time by development method by device class. Dots indicate outliers. 34

5.1 Boxplot including median Android CPU-intensive execution time by development method by device. Dots indicate outliers. 50
5.2 Boxplot including median Android CPU-intensive CPU usage by development method by device. Dots indicate outliers. 51
5.3 Boxplot including median Android CPU-intensive memory footprint by development method by device. Dots indicate outliers. 53
5.4 Boxplot including median iOS CPU-intensive execution time by development method by device. Dots indicate outliers. 54
5.5 Boxplot including median iOS CPU-intensive CPU usage by development method by device. Dots indicate outliers. 55
5.6 Boxplot including median iOS CPU-intensive memory footprint by development method by device. Dots indicate outliers. 56
5.7 Boxplot including median Android memory-intensive execution time by development method by device. Dots indicate outliers. 57
5.8 Boxplot including median Android memory-intensive CPU usage by development method by device. Dots indicate outliers. 58
5.9 Boxplot including median Android memory-intensive memory footprint by development method by device. Dots indicate outliers. 59
5.10 Boxplot including median iOS memory-intensive execution time by development method by device. Dots indicate outliers. 60
5.11 Boxplot including median iOS memory-intensive CPU usage by development method by device. Dots indicate outliers. 61
5.12 Boxplot including median iOS memory-intensive memory footprint by development method by device. Dots indicate outliers. 62
5.13 Boxplot including median Android network-intensive execution time by development method by device. Dots indicate outliers. 63
5.14 Boxplot including median Android network-intensive CPU usage by development method by device. Dots indicate outliers. 64
5.15 Boxplot including median Android network-intensive memory footprint by development method by device. Dots indicate outliers. 65
5.16 Boxplot including median Android network-intensive total data transferred by development method by device. Dots indicate outliers. 66
5.17 Boxplot including median iOS network-intensive execution time by development method by device. Dots indicate outliers. 68
5.18 Boxplot including median iOS network-intensive CPU usage by development method by device. Dots indicate outliers. 68
5.19 Boxplot including median iOS network-intensive memory footprint by development method by device. Dots indicate outliers. 69

5.20 Boxplot including median iOS network-intensive total data transferred by development method by device. Dots indicate outliers. 70

6.1 Three-tier architecture of the app 86
6.2 Pages of the app. From left to right: *Todo List* page, *Add Todo* page and *Edit Todo* page . . 87
6.3 Boxplots including median build time by development method by platform. Dots indicate outliers. 92

D.1 The style guidelines of the app designed in Chapter 6 151

List of Abbreviations

AI	Artificial Intelligence
AR	Augmented Reality
ARM	Advanced RISC Machines
ART	Android Runtime
CPU	Central Processing Unit
CSR	Client Side Rendering
CSS	Cascading Style Sheet
DEX	Dalvik Executable
DOM	Document Object Model
GB	Gigabyte
Gbit	Gigabit
GHz	Gigahertz
GPU	Graphics Processing Unit
GUI	Graphical User Interface
IDE	Integrated Development Environment
JSON	JavaScript Object Notation
JSX	JavaScript Syntax Extension
LLOC	Logical Lines Of Code
LOC	Lines Of Code
MB	Megabyte
MHz	Megahertz
ms	millisecond
OS	Operating System
RAM	Random-Access Memory
RTC	Real-time Communications
SDK	Software Development Kit
SLOC	Source Lines Of Code
SSD	Solid State Drive
SSR	Server-Side Rendering
UI	User Interface
URL	Uniform Resource Locator
UX	User Experience
VR	Virtual Reality

XML	Extensible Markup Language

1 Introduction

1.1 Problem and Motivation

Ever since the introduction of the iPhone in 2007, the use of smartphones has been growing rapidly, to the point where in 2021, there are an estimated 6 billion smartphone users worldwide [1].

One of the most important features offered by these hand-held devices is extended functionality through applications, the diversity of which is growing every year. The volume of mobile app downloads globally rose from 140 billion in 2015 to 218 billion in 2020 [2].

The mobile operating system market is dominated by two major platforms: Android by Google and iOS by Apple. As of June 2021, these two constituted more than 99% of worldwide mobile operating systems, with Android leading at 72.84%, followed by iOS with 26.34% [3].

Despite the fact that these mobile platforms have been around for more than a decade, native app development is still the primary way of app development on Android and iOS [4]. Native app development refers to building an app for a single platform leveraging the Software Development Kit (SDK), programming languages and user experience that are specific to that platform. As a result, the development process for building an app for more than one platform — with this approach becomes slower and more expensive since multiple developers with different skill sets are required to build that app for each platform. This problem persists despite the fact that what service providers, product owners and developers overwhelmingly want is to produce applications that run on both platforms.

Although cross-platform mobile development tools (CPDTs for short) allow the developers to build apps for both platforms using a single codebase, such methods have traditionally required notable sacrifices in terms of app performance, resource use and user experience [5, 6, 7, 8, 9].

The emergence of a new generation of CPDTs in recent years on one hand and improvement of smartphones' hardware and software on the other once again raised hopes that the ideal of "write once run everywhere" is achievable. Unfortunately, there have been few studies systematically assessing trade-offs associated with such recent CPDTs. While there is a considerable number of related studies in the earlier half of the 2010s across industry and academia that mostly focused on CPDT selection criteria and responsiveness and convenience of use assessment, these efforts did not keep up with the fast-paced mobile industry, leaving the development community to wonder about the trade-offs between the latest CPDTs. Besides, studies that investigate the CPDTs trade-offs in terms of resource use and performance with dedicated resource-intensive tests are hard to find. In addition, work that quantitatively reveals the CPDTs trade-offs from a developer

experience standpoint is not yet available.

This work seeks to fill this gap based on previous work by extending and introducing novel quantitative assessments that can serve developers, service providers and product owners as a reference regarding CPDT trade-offs.

1.2 Solution

The objective of this thesis is to identify some of the practical trade-offs of apps built by modern CPDTs when compared amongst themselves and against their native counterparts. This thesis investigated whether perceived inefficiencies associated with CPDTs identified by earlier research [5, 6, 7, 8, 9] still exist. This work established and applied a quantitative evaluation framework to investigate modern CPDTs with respect to metrics and other trade-offs involving responsiveness and convenience of use, resource use and performance, and developer experience.

The domain of our research considered here is the data-driven apps that run on mobile phones on either the Android or iOS platforms. This work did not consider apps that are primarily designed to run on tablets and graphic-intensive games, as they require other considerations.

The evaluation framework considered here was distinct but drew on elements of previous work in this area that can reveal trade-offs of using these modern CPDTs. The evaluation has three elements. The first component of the evaluation quantitatively investigates responsiveness and convenience of use related metrics. The second component evaluates resource use and performance trade-offs amongst CPDTs through several resource-intensive tests. Finally, in the last component, we turned our attention towards metrics that affect the development process and sought to identify trade-offs from the perspective of the developer experience.

1.3 Steps to Solution

Within this work, multiple steps were taken to investigate CPDT trade-offs.

The first step sought to find prominent CPDTs. We relied on both academic literature and industry surveys to identify candidates based on the perceived developer's interests and adoption rate in recent years.

The second step involved identifying key differences between native development and that using CPDTs. This work found that at the very high-level user experience and developer experience differ strongly across native development and CPDTs. This is backed by results of other literature [5, 6, 7, 8, 9, 10]. We further narrowed down these differences into three areas: responsiveness and convenience of use, resource use and performance, and the developer experience.

The third step involved developing a three-fold evaluation system based on these key differences to assess each of these areas quantitatively.

Finally, this work evaluated the results of our assessment and discussed the trade-offs discovered.

1.4 Evaluation

The study first identifies three modern CPDTs — Flutter, React Native and Progressive Web Apps (PWA) — based on developer interest and adoption rate in recent years, as suggested by academic and industry-driven surveys. It proceeds on to discuss the characteristics and architectures of those CPDTs.

The thesis then develops a three-fold evaluation system to identify trade-offs with respect to three main defined criteria between CPDTs and with reference to native development in data-driven mobile apps. This evaluation system drew on elements of other evaluation systems already developed in this domain, but also introduced measurements and metrics across a wider selection of CPDTs.

Each component of our evaluation system evaluates a corresponding element of our comparison criteria using a set of quantitative evaluation-specific metrics. For each evaluation, we first developed an evaluation-specific app for each of our baseline comparators, i.e., Android native and iOS native. This work then implemented an app with the same design for each successive CPDT. The evaluation-specific framework metrics were applied in turn against each of these apps.

The first component of the evaluation system focuses on trade-offs associated with responsiveness and convenience of use. Three metrics were defined and assessed to support that evaluation: launch time, navigation time, and size of the installer. These metrics were assessed in the apps by measuring the associated outcomes.

The second component of the evaluation supports investigation of trade-offs associated with resource use and performance. Evaluation of resource use and performance is achieved through the design and implementation of a set of resource-intensive tests and associated metrics, such as CPU usage, memory footprint and the associated execution times. Those tests were conducted and outcomes were measured in each of the development approaches explored here.

The third component of this work focuses on finding trade-offs related to developer experience. Developer experience is assessed by measuring three metrics: lines of code, cyclomatic complexity, and build time. Such metrics are then assessed across multi-page data-driven apps developed separately using each of the development tools. These apps were implemented based on a design specifying their functionalities and UI/UX.

1.5 Contribution

1.5.1 Scientific contributions

This study makes four primary scientific contributions:

- Conducting the first quantitative investigation studying the trade-offs of contemporary CPDTs from the developer experience standpoint, and providing empirical evidence of such trade-offs. This component of

the study introduced metrics and tools to measure developer experience-related parameters. It achieved greater cross-platform comparability through the use of a method in which a real-world data-driven app for evaluation was designed, and then systematically implemented and evaluated across multiple development tools using shared metrics.

- Serving as one of the first quantitative studies to provide insight on the resource use and performance overhead imposed by recent CPDTs. This section of the study demonstrated how to measure resource use and performance on the Android and iOS platforms between apps built using different development tools.

- Identification of CPDT trade-offs with respect to responsiveness and convenience of use. This element of the study also demonstrated methods by which to measure metrics such as start-time and navigation time for cross-platform apps.

- Description of architectural characteristics of prevalent contemporary CPDTs, particularly those germane to trade-offs involving responsiveness, convenience of use, resource use, performance, and developer experience.

Engineering contributions

There are three main engineering contributions presented in this work:

- Implementing and open-sourcing multi-page data-driven apps using five different development methods: native Android (Java), native iOS (Swift), Flutter (Dart), React Native (JavaScript) and PWA (JavaScript). The apps were built based on a set of design rules and complemented by a server-side application (Node.js), and were designed to support assessment of trade-offs between development methods. All the apps were open-sourced and published on a public repository.

- Implementation of apps featuring three resource-intensive tests — CPU-intensive, memory-intensive and network-intensive — for five different development tools: native Android (Java), native iOS (Swift), Flutter (Dart), React Native (JavaScript) and PWA (JavaScript). These apps were designed to push the mobile devices' targeted resources to their limit and to serve as the basis for evaluation of trade-offs between development methods. All the apps were open-sourced and published on a public repository.

- Implementation of a custom measuring tool that can count source lines of code specifically optimized for five prevalent development methods: native Android (Java), native iOS (Swift), Flutter (Dart), React Native (JavaScript) and PWA (JavaScript).

1.6 Thesis Structure

The structure of the thesis is as follows. Chapter 2 introduces different mobile app development methods and provides a literature review of related work concerning CPDT trade-offs. Chapter 3 identifies contemporary CPDTs based on recent academic and industry surveys and discusses their characteristics and architectures. Chapter 4 presents the first component of the evaluation, which investigates the CPDT trade-offs associated with responsiveness and convenience of use. Chapter 5 introduces the second section of the evaluation, an investigation of the CPDT trade-offs involving resource use and performance. Chapter 6 presents the last element of the evaluation that investigates CPDT trade-offs from the perspective of developer experience. Chapter 7 concludes this work by providing a summary of the overall findings, contributions, limitations, and potential directions for future work.

2 Background and Related Work

This chapter is divided into two sections. In the first section, we introduce native and cross-platform mobile development as well as the most common approaches in CPDTs (cross-platform development tools). The second section discusses contributions across industry and academia related to CPDT trade-offs and how this work can extend them.

2.1 Background

2.1.1 Native Development

While the iPhone was introduced in 2007, by 2010, the smartphone industry was a crowded market, with many operating systems trying to increase their market share. A few years into the competition it was clear that only two dominated the market. Google's operating system Android and iOS from Apple collectively rose their worldwide market share from 47.25% in 2010 to 82.53% in 2015 [3]. By June 2021, it had reached 99.18% [3].

The primary and longest-established way of developing a mobile app is called native development, and we term an app developed this way a native app. Such apps are developed by the platform's own software development kit (SDK) and programming language(s) and run exclusively on that platform. [11, 12].

Android offers Android Studio, an integrated development environment and the Java and Kotlin for native languages for app development. By contrast, iOS offers Xcode, an integrated development environment, and two distinct programming languages for development, Objective-C and Swift. Apps that are developed natively for one platform are not compatible with the other [13].

Naive development is difficult and expensive. It requires a high level of experience [14] and heavy additional cost and effort are required to write an app for multiple platforms [15]. This barrier of writing an app entirely for each platform has stimulated the quest for alternative solutions.

2.1.2 Cross-platform

An alternative to native development was inspired by the ideal of develop once and run anywhere [12]. This ideal triggered development of industry and academia development approaches grouped under the designation of cross-platform development [12, 16]. Major motivations for adoption of cross-platform development strategies include budget, development time and learning curve considerations [7, 17, 18].

In order to better characterize CPDT trade-offs, it is important to distinguish between different approaches to CPDT and their respective properties. Past research has established many classification schemes, such as those collected by El-Kassas et al. [12]. A common way for recent work to characterize CPDTs approaches classifies them into four categories: web apps, Hybrid, Interpreted, Cross-compiled [19, 20]. In recent years, these have been supplemented by an additional approach producing what is termed Progressive Web Apps [21]. Each of these types of apps development is characterized below.

Web apps

Web apps are built using web technologies including HTML, CSS and JavaScript and are accessed through a web browser with an active network connection [22]. Responsive web design can be used to make a web application — whether a conventional website or a single-page application — viewable on mobile screens, and that works well with touchscreens [23]. At the time of writing, prevalent web frameworks for building web apps include React [24], Angular [25] and Vue [26]. Ionic [27] also serves as a more specialized web framework designed for mobile web apps.

One shortcoming of web apps, when considered as a CPDT, is their limited access to device-specific features. A web app is limited to what is provided by its hosting browser's APIs rather than the resources and functionality exposed by the underlying mobile OS. Amongst other advantages, two approaches discussed below — hybrid and Progressive Web Apps — extend web apps with improved capacity to access to platform APIs.

Hybrid

At a very basic level, the hybrid approach involves a web app packed into a standalone app using a native wrapper [28]. Business logic and user interfaces are built using HTML, CSS and JavaScript leveraging a WebView component that is embedded in a native app [29]. WebView is an embedded browser that a native application can use to display web content. The native wrapper is a native app that can communicate with WebView through a generic JavaScript API that bridges service requests from the web-based code to the corresponding platform API [30].

The hybrid approach extends web apps by providing a wider range of platform APIs and allows distribution through an app store.

Apache Cordova (formerly PhoneGap) [31] is one of the most popular open-source examples of this approach.

Interpreted

The interpreted approach has a major difference from Hybrid as it no longer relies on a WebView to render views [12], instead of using the platform's native view components to render user interfaces [32]. Business

logic can be written in a language other than native and interpreted through an on-device interpreter [9]. JavaScript is a very common language used in this approach [33, 34, 35].

Some studies reported this approach improved the Hybrid in terms of performance and user experience by using native user interfaces instead of HTML and CSS-based views [6].

Two CPDTs that using this approach is Titanium [34] and React Native [33].

Cross-compiled

The cross-complied approach has one fundamental difference between Hybrid and Interpreter. It no longer needs a WebView or an interpreter to communicate with the native platform or to render user interfaces; instead, it compiles from a common language to the target native platform's byte code [20]. One advantage of this approach is the control over how the native user interfaces are rendered [36].

Two CPDTs taking advantage of this approach are Xamarin [37] and Flutter [38].

Progressive Web Apps

The last and most recent approach is Progressive Web Apps – PWAs for short. It was presented by Google in 2015 to enhance the traditional web apps [39]. Similar to a web app, it is delivered throw the web (hosted on a web server) and is built using web technologies: HTML, CSS, and JavaScript while looks and feels like a native app and runs across platforms both on mobiles and Desktops. The ability to be installed and be available on the device's home screen or app launcher and working offline are amongst the most important features of PWAs [40].

2.1.3 React

React is an open-source JavaScript front-end library for building user interfaces [24]. It has been created and maintained by Facebook since 2013. It was listed as the most popular web framework in a 2021 survey by Stackoverflow with over 67,000 participants [41].

The React API is declarative, making use of a programming paradigm that expresses the logic of a computation without describing its control flow [42]. Such an API makes it easier for developers to define user interfaces and states. A "Virtual DOM" and Reconciliation are two critical features in React architecture that have enabled React's declarative APIs. Virtual DOM offers a virtual representation of the program's user interfaces in memory. The process of syncing and effectively updating the Virtual DOM with the real DOM (of the browser) based on the most recent changes in the program is called reconciliation [43]. [44]

React is component-based and allows for complex user interfaces to be built by composing smaller encapsulated components [24]. Each component manages its state and has its business logic defined in JavaScript and its user interfaces defined in "JSX". JSX is a syntax extension to JavaScript, similar to a template language, and describes the UIs of a component. [45].

Amongst the significance of React library in this thesis was its contribution to the creation of PWA apps (Chapter 4, Chapter 5 and Chapter 6).

2.2 Related Work

This section reviews peer-reviewed related work concerning CPDT trade-offs in academia and industry and characterizes gaps addressed by this study. First, studies related to the CPDT selection criteria are discussed. Then we investigate academic and industry work related to CPDTs' responsiveness and convenience of use, and resource use and performance. Finally, studies that examined the CPDTs from a developer experience perspective are discussed.

2.2.1 Criteria for CPDT selection

When the first CPDT studies emerged in the early 2010s, many CPDTs existed, and choosing the most appropriate CPDT was a challenge. A study by Hartmann et al. [46] was the first work that investigated the CPDT selection challenge. Inspired by the boom in the mobile development landscape, they qualitatively analyzed several CPDTs in terms of capabilities, strengths and weaknesses. They investigated as many as 22 CPDTs across six mobile operating systems. Their selection criteria included an open-source CPDT that is mature, has an extensive community with decent documentation and can provide a complete toolbox for developing cross-platform mobile apps. They found that only a few frameworks — including Rhodes Mobile, PhoneGap and Titanium — supported such criteria.

In a highly cited paper in 2012 [47], Heitkotter et al. presented a set of criteria for evaluating cross-platform development approaches. In the first part of their work, they defined a list of criteria from the infrastructure's perspective. Their criteria included license and costs, supported platforms, access to platform-specific features, long-term feasibility, and the look and feel of the app built by a CPDT. They evaluated web apps, native, PhoneGap and Titanium and reported PhoneGap as their preferred CPDT.

In another highly cited study in 2012, Palmieri et al. [48] made a comparison amongst four popular cross-platform tools at the time, with an aim to support developers in making the right choice concerning their needs/constraints. Mobile OS support, licensing terms, offered programming language, availability of APIs, accessibility to native APIs, architecture and IDE for CPDTs were considered in their comparison criteria.

In 2013 Dalmasso et al. [6] put forward a number of decision criteria to aid the process of selecting a "suitable" CPDT. The general requirements reported in their work for CPDT selection included support for multiple mobile platforms, rich user interface, back-end communication, security, app extensions, optimized power consumption, accessing built-in features and availability as open-source.

Another set of criteria was considered in a study in 2014 by Angulo et al. [7] that investigated the impact of CPDTs from the perspective of user experience (UX). They studied the characteristics of eight CPDT candidates, and selected Titanium, PhoneGap and Intel XDK. Their selection criteria included support for

Android and iOS, a well-known programming language, and access to device APIs.

In 2015, Hudli et al. [10] proposed a multi-perspective evaluation framework to aid the CPDT selection process. This peer-reviewed study defined multiple criteria in three groups: platform and device support, development support and deployment factors. They included six CPDTs in their evaluation. Amongst those, Sencha Touch Mobile [49] and PhoneGap scored the highest. The researchers' platform and device support criteria included device resource support, touch support, in-app browser support and support for a local database.

Most of the studies regarding CPDT selection criteria were undertaken between 2011 and 2015 — a period offering many CPDT candidates from which to choose. More recent studies usually refer to this earlier body of work. A 2017 survey by Biørn-Hansen et al. [50] assessing the academic body of knowledge in the domain of CPDTs reported that new papers might too heavily draw from or rely upon considerably outdated research.

CPDT selection in Chapter 3 is based upon the peer-reviewed studies mentioned above while taking the recent changes in the mobile industry into account.

2.2.2 Responsiveness, convenience of use, resource use and performance

This section sheds light on previous work that contributed to identifying trade-offs associated with CPDTs in terms of responsiveness and convenience of use, resource use and performance. We then discuss the findings of such work, and explain how this thesis can extend that work.

Between 2011 and 2013, several studies from academia appeared that contributed to the foundation of CPDT evaluation criteria.

Hartmann et al. reported the strength and weaknesses of many CPDTs available at the time. Their work was followed by Mikkonen and Taivalsaari's paper that focused on the web as the solution for cross-platform development and presented an overview of Hybrid (called "Native Web" in their work) vs. web apps (called "open Web apps" in the original paper) [51].

Heitkötte et al. [47] evaluated several CPDTs based on a set of defined requirements and compiled their results in tables. Their proposed CPDT evaluation criteria are widely cited in later literature. However, their analysis required additional empirical evidence, as suggested in their recommendations for future work.

Palmieri et al. [48] compared four CPDTs in terms of features and support across mobile platforms. This study constituted one of the first comprehensive studies, focusing on (four) popular CPDTs at the time, and provided an overview of such CPDT's architecture, features and cross-platform support.

Amongst the first studies with a focus on CPDT performance trade-offs is a work by Corral et al. [5] in 2012. The researchers developed an app using PhoneGap and a baseline native Android app, and ran a series of benchmarks that evaluated hardware, network and data access time. Their work reported an overall performance penalty for the web-based approach (PhoneGap) in comparison to the native Android. Their analysis only measured execution time, and did not consider measures of usage of device resources such as CPU and memory during their benchmarks.

In the following year, Dalmasso et al., in another highly cited paper [6] extended the previous work by evaluating two CPDTs (PhoneGap and Titanium) in terms of memory, CPU usage and power consumption. The investigators reported that PhoneGap used less memory, CPU and power than Titanium. Their evaluation lacked a native reference point, and was limited to resource usage — with no performance-related metrics such as test completion time — in a benchmark that is best characterized as network-intensive.

In industry around 2012, an earlier mobile version of the Facebook app built as a web-based app reliant on HTML5 features soon became problematic, with many users reported issues regarding performance and quality. Facebook consequently rebuilt their app using a native features, abandoned their web-centric, HTML5 approach [52].

The mobile open-source community developed an open-source project called Property Cross [53] to help developers select a cross-platform mobile framework. Property Cross was a multi-page app that used many device capabilities — including navigation, geolocation, storage and web services. The app was implemented across several CPDTs [54]. The project started in 2012 and was maintained for a few years. During this time, it served the mobile app development and researcher community.

In two similar studies in 2015 [8] and 2016 [55], Willocx et al. used the Property Cross app in their quantitative CPDT evaluation. Their work investigated response time, performance and resource use across a range of CPDTs, including Xamarin and PhoneGap, and compared them with baseline native Android and native iOS; native Windows Phone was added as the third baseline in their second study. One novel aspect of their work was its assessment of the effect of device class — "high-end" and "low-end" — on CPDT app responsiveness and performance trade-offs. Their studies reported performance and resource use overhead imposed by CPDTs compared to their native counterparts. One of the shortcomings in the two studies from Willocx et al. was the absence of dedicated resource-intensive tests for measuring the resource outcomes such as CPU usage and memory footprint, which were measured during the app launch and navigation experiments.

With the emergence of Progressive Web Apps (PWAs), academia has started paying attention to this novel approach. Amongst the first PWA-related analysis work was a 2017 paper by Biørn-Hansen [21] that argued for progressive web apps as a technology that raised the possibility of unifying web and native apps. Their work consisted of an overview of PWA features and a quantitative performance evaluation that compared a PWA app against Hybrid (Ionic) and Interpreted (React Native) counterparts. The researchers conducted their evaluation only on the Android platform, and without a baseline native app comparator. The investigators reported that when compared against other tested CPDTs, PWAs offered excellent potential as a future unifying technology for web and native apps. As reported in their paper, their analysis aimed to gather and present preliminary results to spark interest in further work.

Kvist and Mathiasson, in a more recent (2019) study [56] assessed PWA in terms of performance and responsiveness. Their quantitative assessment was conducted by Google Lighthouse [57], an open-source tool that helps audit web apps. They evaluated a PWA, a regular web app and a hybrid Android app built by

Cordova. They reported that the PWA app was comparable — in terms of performance and responsiveness — to an app developed in Cordova, and faster than a web app. However, their work was limited by the absence of a baseline native app for comparison. Their performance outcome measures were limited to the launch time metrics available in the assessment tool (i.e., Google Lighthouse).

Amongst more recent surveys, Biørn-Hansen et al. [58] scrutinized the resource usage and performance overhead of multiple CPDTs, as compared against native development on Android. Their work evaluated access to native-side function interface calls in terms of performance and resource use, including those associated with the geolocation API, contacts API, file system, and accelerometer. Measurements in their work emphasized metrics such as test completion time and CPU and memory usage. They reported a decrease in performance and higher resource use in apps produced by CPDTs when compared to native counterparts. They also reported that some of the apps produced by CPDTs could perform equally or better than their native counterparts on certain metrics. An important study limitation is the absence of assessment results on the iOS platform and the lack of dedicated resource-intensive tests.

There are several ways that this thesis extends previous work with respect to responsiveness and convenience of use, and resource use and performance: The landscape of mobile development has exhibited rapid evolution — especially during the last decade. This fast pace of advancement has left a small number of CPDTs in the competition. The results of many studies presented in this section were based on the CPDTs that are no longer available, maintained or widely used. This study extends the previous work by including a wide selection of widely used contemporary CPDTs in our evaluation.

The absence of a baseline native app in an evaluation negatively affects the generalizability and comparability of the results. The majority of the studies discussed in this section were missing a baseline comparator in their assessment. Moreover, some only considered a single platform — often Android — and omitted the other platform out. This study seeks to address this issue by including baseline native Android and native iOS apps in all of the evaluation components in this thesis.

We did not find a resource use and performance assessment in the previous literature that based their evaluation on dedicated, resource-intensive tests. In other words, those previous studies that measured indicators of resource usage — such as CPU usage and memory footprint — did so using tests that were not originally designed for benchmarking that resource.

Another limitation observed in some peer-reviewed existing work is the absence of apps specifically designed and developed to support the evaluations conducted in that work. Such work instead often relied upon apps not originally designed with the research goals in mind. This shortcoming is addressed in this thesis by designing and implementing dedicated apps addressing the needs of each component of the evaluation.

2.2.3 Developer experience

This section discusses the previous literature that contributed to identifying trade-offs associated with CPDTs from the standpoint of developer experience. We characterize the findings from that work, and explain how

this thesis can extend their work.

Heitkotter et al. [47] presented the first peer reviewed work that defined a set of criteria for CPDT evaluation from a software development standpoint. Criteria involving areas such as development environment, GUI Design, ease of development and maintainability were defined and assessed qualitatively in tables. Maintainability was the only criterion defined with a quantitative metric.

Palmieri et al. in 2012[48] reported that CPDT adoption could improve maintainability due to maintaining just a single body of source code. Hudli et al. [10] proposed a list of criteria from a developer's perspective.

Amongst more recent work, Majchrzak et el. [9] studied design and implementation spheres such as navigation, data fetching and debugging across three CPDTs (React Native, Ionic and Fuse). In a high-level summary table, the authors qualitatively labelled them as "simple", "intermediate" or "complex" — in terms of the implementation effort involved.

The limited scope of the existing peer-reviewed literature suggests that developer experience is one of the most under-researched aspects of CPDT — especially in terms of providing empirical evidence and conducting quantitative assessments. To the best of our knowledge, no quantitative study has been undertaken so far in this area.

This work extends the limited previous work that focused on the developer experience domain of CPDTs by introducing the first quantitative evaluation that investigates the trade-offs of modern CPDTs from the developer experience perspective. This study presents metrics and tools to measure developer experience-related parameters. It achieved greater cross-platform comparability by the use of a method in which a real-world data-driven app for evaluation was designed and then systematically implemented across multiple development tools.

2.3 Summary

This chapter initially presented an overview of native and cross-platform development and how the latter aims to solve the challenges associated with the native development approach. The chapter then introduced different cross-platform development approaches, and noted their respective characteristics.

The balance of the chapter reviewed related work with respect to the CPDT trade-offs characterized by academic and industry observers.

First, related work with respect to CPDT selection criteria was visited. The majority of such work was conducted during the first half of the 2010s, and was heavily referred to by later studies. Our work in that area is based on those previous peer-reviewed studies while reflecting the most recent findings from industry-driven surveys on CPDT popularity and adoption.

The chapter then considered academic and industry-related work focused on CPDTs' responsiveness, convenience of use, resource use and performance. Limitations associated with such work were pointed out, and a number of ways to extend those lines of work in this thesis were identified, such as including a selection

of widely used contemporary CPDTs, extending the assessments to compare CPDTs to the baseline native Android and native iOS apps, and designing and implementing dedicated resource-intensive tests and apps to align with the goals of the resource use and performance evaluation.

Finally, the chapter continued on to characterize previous literature that examined developer experience trade-offs of CPDTs. Developer experience trade-offs were among the most under-researched aspects of CPDTs. To the best of our knowledge, no quantitative work has previously been conducted in this area. We then explained how this thesis extends research in this sphere by introducing the first quantitative work that assesses the trade-offs of modern CPDTs from a developer experience standpoint.

3 CPDT Selection

In this chapter, we first identify the most prominent CPDTs (cross-platform development tools) available today, explain the reasons behind their selection for our evaluation, then continue the chapter with an overview of each of those selected frameworks that describes their characteristics, architecture, strengths, and weaknesses.

3.1 Selection

Developer interest was one of the most crucial considerations motivating the selection of CPDTs for evaluation in this thesis. In order to assess the developer interest level in and degree of adoption of CPDTs, we relied on recent surveys from industry and academia.

The most recent study that surveyed the industry's perspective on CPDT popularity and adoption was a 2019 work by Biørn-Hansen et al. [59]. They conducted an online survey with 101 participants from developer-oriented online forums and websites. Their results exhibited that React Native and two Cordova-based CPDTs — Ionic and PhoneGap — were the most widespread CPDTs. React Native scored highest in participant interest.

As part of a 2021 industry-driven survey from Stackoverflow, 59,921 developers were asked about the frameworks and libraries in which they had undertaken extensive development work over the past year [60]. Amongst the CPDT related responses were React Native at 14.51%, Flutter at 13.55% and Cordova with 7.18%. As a side note, this survey's question was not exclusive to mobile frameworks and libraries.

Another survey from the industry by Statista targeted mobile app developers globally between 2019 and 2021 to find out which CPDTs they used in that year [61]. Based on responses from 31,743 mobile app developers and only focusing on the most recent year (2021), Flutter was the most popular CPDT — used by 42% of the survey responders, followed by React Native at 38%. Both Cordova and Ionic were used by 16% of the participants in that survey.

Given the perceived developer interest and adoption rate of the CPDTs in recent years reflected in responses to scientific and industry-driven surveys and the selection criteria discussed by previous work and characterized above and in Section 2.2.1, we decided to select Flutter and React Native for evaluation in this thesis. Cordova and the CPDTs based on that technology have also been popular and been included in many studies; however, considering Cordova's prevalence of use in the industry-driven survey from Statista [61] — dropping from 29% in 2019 to 16% in 2021, and the limitation of Cordova (and its similar hybrid apps)

requiring a native wrapper on each platform as mentioned in Section 2.1.2 — have convinced us instead to include another CPDT in this work's evaluations. Specifically, the other CPDT selected for evaluations in this work is a novel and recent web-based approach — Progressive Web Apps — with the potential to be a unifying technology for web and native apps [21] and can run across platforms without a need for a native wrapper. Each of these CPDTs is discussed below.

Before beginning the discussion of these CPDTs, it bears emphasis that CPDTs are not panaceas, and have their limitations. Most notably, CPDTs are poor matches to apps that, amongst other things, heavily rely on computation or advanced features such as Augmented Reality (AR), Virtual Reality (VR) and performance-intensive algorithms, such as those required in certain Artificial Intelligence (AI) tasks (e.g., those involved in training and testing deep learning models) as well as the apps that heavily rely on the sensors available in mobile devices such as accelerometers, proximity and heart rate sensors. For such use cases, native development remains advised. Some related considerations will be assessed in Chapter 4 and Chapter 5.

3.2 Flutter

Flutter is a cross-platform UI toolkit created by Google and initially released in 2017 [62]. It is an open-source framework that allows apps to be built from a single body of source code for operating systems such as iOS and Android.

Flutter interfaces directly with the native platform and has UI/UX similar to that offered by a native app on that platform. Flutter compiles directly to machine code when in production; during development, the Flutter runtime runs the app in a Virtual Machine that supports hot reload — the ability to apply changes to the code while the application is running without needing the app to be fully recompiled.[63]

Flutter apps are written in the "Dart" language [63], an object-oriented, class-based, garbage-collected language that can compile to either native code and JavaScript [64]. Dart, amongst other things, provides the language and runtime for Flutter apps. When deploying native apps including Android and iOS to production, the Dart compiler in Flutter enables ahead-of-time compilation to native ARM on iOS and both ARM and x86-64 (x86 64-bit) machine code on Android [65]. Thereafter, the (ahead-of-time) compiled code runs inside a Dart runtime [66]. For Android, a bundle containing both ARM and x86-64 compiled code is often uploaded to Google Play store. When a user than seeks to download the app, Google Play determines whether the user's device has an ARM or x86-64 processor and transfers the associated processor-specific compiled code to that requesting device.

Flutter has an extensible, layered system consisting of independent libraries that rely on the underlying layer. Architectural layers are shown in Figure 3.1.

The first layer that provides an entry point to the underlying operating systems is a platform-specific "embedder". It provides access to low-level services such as rendering surfaces, responding to input, leveraging the underlying operating system, and managing the message event loop. Embedder is written in Java and

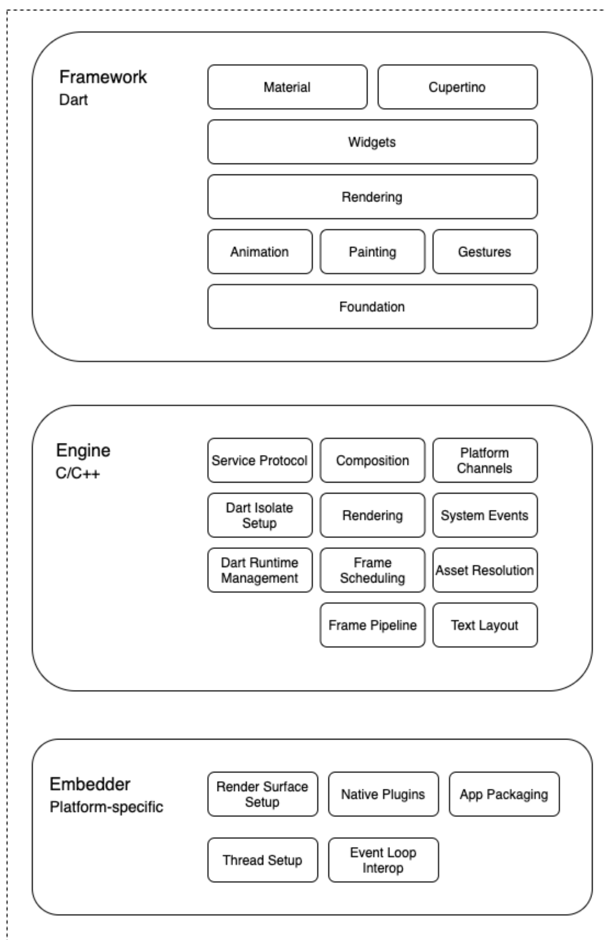


Figure 3.1: Flutter architectural layers. Image from [67]

C++ on Android and Objective-C/Objective-C++ on iOS. There are embedders for other operating systems, but we only focus on mobile platforms in this chapter. [67]

At a higher level from the embedder layer is the central layer, which is associated with the Flutter "engine", providing primitives to all the Flutter applications, including low-level implementation of Flutter's core API such as graphics, text layout, file and network I/O and Dart runtime. The Flutter framework access the engine through the *dart:ui*, which wraps the underlying C++ code in Dart classes [67].

In the Flutter architectural layers, the top layer is the Flutter framework. This is the layer with which developers usually work, and is written in Dart language. It includes a set of platforms, layouts and foundational libraries. The bottom layer in the Flutter framework consists of foundational classes and building block services. Next, we have a rendering layer, which is an abstraction for working with layout. The rendering layer can be leveraged to build a tree of renderable objects that can be manipulated dynamically, with a tree automatically updating the layout based on the changes. Above the rendering layer is the widget layer, a composition abstraction with a class for each render object. The top layer has two libraries — "Material" and "Cupertino" — that offer UI controls that serve to implement the Google Material (for Android) or iOS

design languages.

The availability of Flutter's own implementation of UI controls — in place of using the UIs provided on the native platform — makes it more extensible for the developer and can improve performance, since Flutter does not need to go back and forth with the native platform to composite the entire scene. The availability of the implementation of such controls within Flutter itself also decreases the dependency of the app behaviour from the underlying operating system.

Flutter uses the "Skia" library for rendering its visuals. Skia is an open-source C++ 2D graphics library created by Google that works across different platforms while providing common APIs [68]. Flutter is bundled with a copy of Skia, which allows developers to keep their apps up to date by having their apps updated to the latest version of Flutter — independent of the native platform's version.

Flutter provides a reactive, pseudo-declarative API for developers to work with, which is inspired by React, a framework created by Facebook [69]; the architecture of a React-based CPDT — React Native — will be discussed below.

Flutter comes with some weaknesses. One of the trade-offs associated with Flutter's implementation of user interfaces is that a Flutter app's UI/UX does not precisely match what is offered by the native platform. In this regard, a developer is limited to the UI widgets provided by the Flutter framework, though these widgets are extensible.

There are also concerns about the availability of Flutter community resources. The Flutter framework and Dart programming language are relatively new technologies; therefore, community support and resources are still growing.

The reader is advised that the characterization of Flutter-related in this section draws heavily on the content accessible via the following link: `https://docs.flutter.dev/resources/architectural-overview`; readers interested in learning more are recommended to consult that source.

3.3 React Native

React Native is an open-source framework created by Facebook and released in 2015. The framework is used for building cross-platform applications for the Android and iOS platforms using React and the platform's native capabilities. React Native allows developers to leverage JavaScript to access each platform's APIs, as well as to describe the appearance and behaviour of the app's UI using React components [70].

The working principles underlying React and React Native are very similar, except that React Native does not manipulate the Document Object Model (DOM) — which offers a data representation of the objects that comprise the structure and content of a document on the web [71] — and instead invokes the native rendering APIs in the target native platform using a bridge. As a result, React Native app uses native user interfaces that look and feel almost the same as a native app [72, 73].

While React directly accesses the browser's DOM elements, React Native is unable to access and manip-

ulate native elements directly. Instead, React Native's communications between the JavaScript side of the app and the native side is made possible through a "bridge". The bridge is a mechanism to pass messages between the two sides [74]. This communication is undertaken in an asynchronous fashion, with messages being batched and serialized into JSON (JavaScript Object Notation, a lightweight format for storing and transporting data [75]) and sent over the bridge. The native side of the React Native app uses the bridge to schedule JavaScript execution. [74].

A React component — and, by extension, a React Native app — can specify the layout of its children using the "Flexbox" algorithm. Flexbox is designed to provide a consistent layout on different screen sizes, and works the same way in React Native as it does in CSS on the web, with a few exceptions. While using React, a browser natively supports Flexbox, but native platforms like Android and iOS do not understand Flexbox and have their own layout systems instead [76]. Facebook addressed this issue by implementing its own layout system, called "Yoga", that can translate Flexbox to the platform-specific layout system [77].

All the business logic and user interface updates in the JavaScript, layout and native related operations happen in three different threads, called the "JavaScript", "Background" and "Main" (UI) thread, respectively [78, 79]. React Native's threads and architectural layers are shown in Figure 3.2. The role and interaction of those threads will now be discussed.

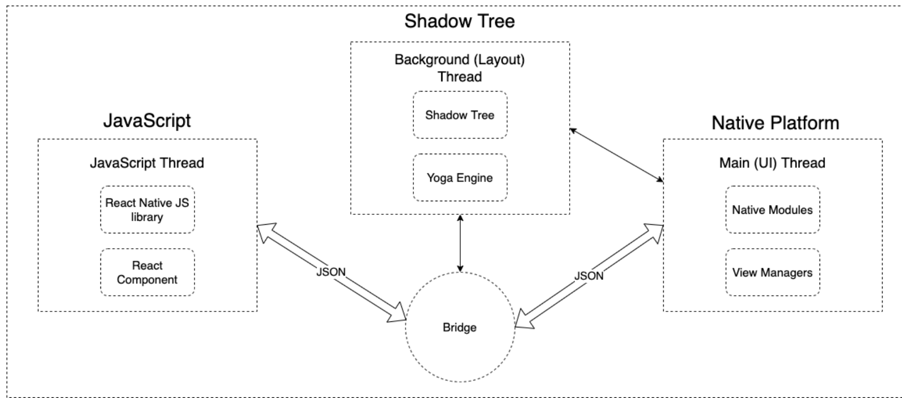


Figure 3.2: React Native architectural layers. Adapted from [79, 80]

The JavaScript thread is where the developer declaratively defines all of the business logic through React's declarative API; React then performs its "Reconciliation" on an ongoing basis to determine if something has changed. Reconciliation is the process through which React updates the tree of React elements. During this process, React figures out how to efficiently update the UI to match the most recent tree of React elements [43]. A tree comprised of React elements that implement a virtual DOM [81]. Subsequently, the Reconciliation results are sent over to the Background thread (also called Shadow tree).

The React Reconciler sends the tree of React elements through the bridge to the Background thread. In this thread, shadow nodes representing the layout tree are created and then passed to Yoga to determine the relative coordinates based on the Flexbox styles passed in from JavaScript. This layer of React Native is called the "Shadow Tree", since it creates a copy of the React Reconciliation tree. Once the layout calculation

has been performed on the Background thread, React Native sends that over to the Main (UI) thread to draw the UI. [79]

The "Main" (UI) thread is where native code is executed, and which handles UI operations such as creating native views and managing views' children. The Main (UI) thread is lightweight and can respond to a user's interactions quickly and communicate with the JavaScript thread whenever the app requires updating the UI. Two other types of components in the native side of React Native are "View managers" and "Native modules". The View Managers are used for drawing on the screen, while the Native modules are designed to invoke device capabilities. Both View Managers and Native modules are extensible.

One of the trade-offs associated with React Native architecture and reliance on the native UIs is the overhead of communication between the native and JavaScript sides of the app. Such communication can come with a potential performance penalty. This will factor into the investigations in Chapter 4 and Chapter 5.

3.4 Progressive Web Apps

Progressive Web Apps (PWA) are a class of apps built through a novel approach advocated by Google to enhance regular web apps in terms of capability, reliability and installability [82]. PWAs can be installed without a marketplace site such as the Apple Store or Google Play Store, work offline when there is no connection, and update when connectivity is available, and keep the user engaged by receiving push notifications and looking and feeling like other installed apps. Although the support for this novel approach is expanding across platforms, it is subject to some limitations, especially on iOS.

PWAs are similar to a regular website when first visited on a browser. However, unlike a regular website, they can be installed and added to the home screen after the first visit. Installation means that the user no longer needs to access the web app through the browser by typing the URL in the address bar and waiting for the website and its content to be loaded. Instead, the web app can also be launched from the home screen in a stand-alone browser instance that hides browser artifacts such as the address bar, menu, and navigation. PWAs can locally store static files, including HTML, CSS, JavaScript, assets, and dynamic content. Local storage of such files and content means they can work offline like other installed apps. PWAs can update their static files and dynamic content when there is connectivity. [83, 82]

Most of the core features associated with PWAs — such as offline support, background sync and push notifications — are enabled through "Service workers" [84]. A Service worker is a script registered on the first visit of a PWA. The browser runs such Service workers in the background, independent from the web page. A Service worker can intercept and manipulate network requests and cache static files and dynamic content [85].

With a Service worker, one can hijack connections and fabricate and filter the responses. While this is a compelling feature for the app developer, it can pose security risks in case of unwanted access. So as to mitigate such risks, Service workers can only be registered on pages served over HTTPS-enabled servers [86].

Another component of PWAs is the "manifest" — a JSON file that specifies how a PWA should be presented when installed on a device. Manifest file content includes the app's name, author, icon(s), version, and description. While such a manifest enjoys full support on Android [87, 88], iOS currently only provides partial support for the manifest.

PWAs are framework agnostic, and can be built using web technologies such as HTML, CSS and JavaScript. Two general architectures under which web pages can be developed are server-side rendering (SSR) and client-side rendering (CSR).

In SSR, an entire web page is fetched over HTTP(S) from the server into a pre-rendered HTML page. Loading the whole page in SSR means that pages are reloaded completely each time a user navigates to a different page. While this approach offers the benefits of a faster first render, subsequent reloading is costly and usually slower than for the CSR alternative [89]. In CSR, a single HTML page is fetched from the server along with the required JavaScript files. The fetched JavaScript runs in the browser, manipulates the DOM, and offloads the subsequent page updates to the client [89]. Amongst CSR benefits are faster navigation and interaction — yielding an experience better suited for a mobile app.

PWA apps that are built for the evaluations in this thesis take advantage of the CSR approach. Specifically, they are created in React, which is one of the most widely used JavaScript frameworks that support CSR.

PWA's status as a relatively new technology and its characteristic of running inside a browser imposes several limitations. Firstly, a PWA cannot be distributed in the app stores similarly to a native app. Second, although this is improving [90], there remain many device functionalities — such as access to contacts, Bluetooth and NFC — that are not yet accessible by PWAs, especially on the iOS platform. Finally, web apps' performance compared to native has traditionally been less favourable — especially for certain apps that require heavy computation. Chapter 4 and Chapter 5 investigate some implications of this limitation.

3.5 Summary

The goal of this chapter was to identify the most prominent recent CPDTs and shed light on their architecture and characteristics. In the first section of this chapter, we selected Flutter and React Native based on developer interests, as evidenced by recent surveys from industry and academia. PWA was also selected as a novel (recent) web-based CPDT with excellent growth potential. The rest of the chapter provided an overview of the architecture, characteristics, and some of the pros and cons of the selected CPDTs.

4 Responsiveness and Convenience of Use

This chapter introduces the first component of our evaluation system to support investigation of the CPDT (cross-platform development tool) trade-offs associated with responsiveness and convenience of use. In this chapter, we first start by defining criteria and parameters which can be measured to evaluate a perceived responsiveness and convenience of use. Next, measurement tools and the adopted methods to measure metrics associated with defined parameters across native and CPDTs are discussed. The chapter continues on to characterize devices on which experiments were conducted, the criteria by which they vary. We subsequently examine the design and implementation of the apps built for the assessments in this chapter. Thereafter, experiment results are presented, interpreted and findings are discussed.

4.1 Evaluation criteria and metrics

In this section, we discuss the importance of responsiveness and convenience as criteria for evaluating CPDTs. This section revisits some of the peer-reviewed related work with respect to responsiveness and convenience of use and tries to explain how this work sought to expand their work. The section continues on to define a range of metrics by which we can quantitatively measure perceived responsiveness and convenience of use, and then discusses the impact of and reasons for selecting each such metric.

Performance related evaluation has served as one the most important focus of studies that investigated mobile cross-platform development. In particular, amongst quantitative work, reporting of performance penalties was treated in a number of studies as one of the most important trade-offs amongst CPDTs. Corral et al. [5] had one of the earliest studies that reported CPDT performance penalties, in the context of their comparison of their web-based CPDT (PhoneGap) compared to native. As part of two of the more comprehensive CPDT evaluation studies by Willocx et al. [8, 55], the researchers assessed the response time related metrics across a range of CPDTs. They reported adverse performance impacts introduced by CPDTs when compared to their native apps.

Examining past work investigating the PWAs and Web-based solutions, we can see that responsiveness and convenience of use lie at the core of their evaluation criteria [56, 21]. In part of their work, Kvist and Mathiasson [56] reported improved launch time related metrics in PWAs when compared to web apps, but native applications were missing from their study, and their comparisons were only conducted on the Android Platform. Biørn-Hansen et al. [21] focused on comparing PWAs and two other CPDTs with respect to the convenience of use related metrics such as app size and installation time the researchers found a faster

installation time by PWAs. However, as with Kvist and Mathiasson, their assessment lacked a native baseline.

Although user experience evaluation is not the focus of our work, responsiveness and convenience of use play an important role in providing a better user experience. Hence, the findings of these studies are interesting for our work. What we can see from most of these studies is that users reported responsiveness and convenience of use related issues with non-native solutions. Results of a UX evaluation study by Angulo and Ferre [7] that evaluated native and CPDT apps found that the 37 study participants reported a higher level of UX achieved by the native apps. In another work, Andrade Cardieri et al. [91] explored UX aspects of PWAs, web and native applications by focusing on participants' interaction with the apps. Their findings reported some signs of interaction issues in non-native apps.

Building on insights and measures from the aforementioned studies and to better assess our criteria in this section, the following metrics are defined.

4.1.1 Size of installer

The first metric impacts convenience of use before users even start engaging with an app: The size of the installer. Mobile app distribution often takes place through a marketplace site such as the Apple App Store or Google Play Store for apps directly targeting mobile clients, or via a hosted website for web-based approaches. Regardless of the source, the mobile app needs to be loaded onto the user's device to be installed and run. The size of the installer greatly impacts the installation time on account of its joint impact on the two dominant components of that time: The time required to load the app from the marketplace or the hosted website, and the time required for the installation process transpiring locally on the device. Reflecting the central character of this impact, we decided to measure the size of installer to better assess the convenience of the first use of an app.

The size of the app is especially important for mobile users with lower-bandwidth connections. This size may also have cost implications associated with the user's internet bandwidth usage. In addition, given such lower-bandwidth connections, the actual or perceived waiting time for downloading an app might affect whether a user tries an app or not, given a large number of available apps on the marketplaces.

We define the installer size metric as the size in megabytes (MB) of the app executable file along with any necessary files included in the required app download after the app is built in release mode (or built for production in web-based apps). Please note that we refer to the "size of installer file" for PWA and web, in this study, as the collective size of that app's files (HTML, CSS and JavaScript files and other required resource files such as icons, images and fonts) required to be loaded when opened by a web browser.

As a side note, we did not include installation time as a measure in this work since the installation process of an app varies greatly by context, due to factors that have little to do with the development platform. For example, to be complete, an installation is often accompanied by one-time requests for permissions or credentials. Moreover, the download component of the installation time depends on connection speed on the associated WiFi or cellular data plans. Rather than directly measuring the installation time, we instead

chose to consider the size of the installer, which is a more fundamental measure that impacts installation time but is not affected by connection speed or installation-related prompts.

Launch time

One of the key metrics associated with the user's perception of an app's responsiveness and convenience of use is its launch time. The startup time of an app provides an important first impression for users expecting an app to be rapidly responsive for their first interaction in a given session. Unlike the size of installer that is involved in a one-time process of installation, process of starting an app is frequently executed. It is one of the most prevalent metrics assessed in other quantitative studies.

We define launch time as the amount of time required from the moment the user touches the app's icon to the time that the app is ready to respond to the user's interaction.

4.1.2 Navigation time

Our last metric that contributes to the evaluation of responsiveness during the app's runtime is navigation time. The small form factor of mobile devices imposes tight limitations on screen real estate, making navigation especially important. Shortcomings in these areas have resulted in apps with multi-page designs requiring frequent user navigation between pages. Poor navigation time adversely affects users' perceived responsiveness.

This work defines navigation time as the time elapsed between user initiation of a navigation and the point at which the destination page is loaded and ready for interaction.

4.2 Measurement tools and methods

In this section, we introduce the measuring tools and describe the methods which we adopted to measure launch time, navigation time and installer size.

There is a number of differences in the way debug and release version of apps work. While many of such differences serve to ease the development process, such differences can also affect our measurements and threaten the validity and generalizability of our results. To eliminate these differences and provide a close analogue to real-world conditions, we therefore based all our measurements on the release build of our apps.

The apps we are targeting to measure in this chapter are built using both Flutter and React Native – systems that generate native apps to run on Android and iOS – plus native Android and iOS apps for baseline comparison. In addition, we measure here characteristics of our web-based apps that run on these two platforms within the runtime of the browser. The way that these apps are built, and especially the character of their runtime environment (in the operating system or browser), poses challenges to measurement. In most cases, this forced the researchers to exclude certain platforms or approaches. We included all of our apps for assessment, and measured them across all the metrics we defined in this chapter by taking advantage

of contemporary measurement tools and selecting the most meaningful and portable measurement metrics available. The only measurement that could not be quantified across all the platforms was the case of PWA launch time on iOS, which will be discussed in greater detail in the next section.

4.2.1 Launch time measurement

Measuring launch time involved the most complicated measurement logistics in this chapter since it involved multiple steps from the time that the user taps the app's icon to the time that it is drawn on the screen and becomes ready for interaction. Because measurement of launch time differs by platform, the following subsections discuss how launch time was measured for each platform.

Launch time measurement on Android

The launch time measurement method discussed here is used for the Android version of our apps built for this chapter including React Native, Flutter and native Android.

There are three alternative processes that can be undertaken when an app is launched in Android: cold start, warm start and hot start [92]. A cold start is undertaken when an app is not already loaded in memory. Apps in this process start from scratch [93]. There are three cases when an app is launched in a cold start state: When an app starts for the first time after installation, after a reboot and when the app starts after being killed by the system. Warm and hot starts are processes in which an already running app is brought to the foreground. This reduces the app's start time relative to a cold start, since there are fewer steps involved [92].

There are two main steps in a cold start on Android. First, the system loads the app and displays a blank screen while creating the app's process. Second, after the process is created by the system, that process takes over and creates (instantiates) the app. App creation has its own steps, from creating an app object to launching the main thread and activity, inflating and laying out views followed by drawing the app for the first time. Once all of these steps have been completed, the system replaces the background window with a depiction of the main activity and lets the user start interacting with the app [93].

For all the apps running on Android 4.4 and higher, Android uses logging to report the time to initial display under the name ”Displayed”. This value is in milliseconds and can be found through *Logcat*, a command-line tool that logs various system messages [94]. Our cold start measurements in Android relied on this logged value.

To assess launch time measurements on Android, prior to testing, we first killed all the apps in the background and rebooted the test device once and then started the measurement by opening the app and logging the launch time with *Logcat*. We subsequently repeated the measurement (without restarting the device), with such repeat being preceded by killing the app in the system to ensure that each launch would involve a cold start.

Launch time measurement on iOS

The general character of launch time in iOS is similar to that in Android. The action when the user taps on the app's icon to open it or seeks to return to an app is called "activation" on iOS [95]. An activation can be launched when a process needs to start; it can also resume when an app is already associated with a live process. There are two types of app launch in iOS: cold launch and warm launch. As for the Android case, our focus for measurement under iOS was on the cold launch which happens when an app starts immediately after installation, after a reboot and when an app that already was open, was evicted from memory and the app's dependencies require re-launching. The launch process is initiated by iOS and is then handed over to the app process that executes the app's code and finally draws its UI [95].

As part of the Xcode toolset, Apple provides a testing tool termed "Instruments" to analyze performance and measure a wide range of parameters. It allows profiling of apps that are built for Apple devices [95]. We used the "App Launch" template of Instruments to profile and measure the launch time by inspecting the time profile that it creates during the launch process. We recorded the time duration from the moment that the system initialized the launching process until the app's initial frame was rendered [95]. Prior to measuring the launch time in this way, all the apps in the background were killed and the device was rebooted once. Subsequently, after each round of measurement, the app was killed to ensure that we had a cold launch every time (without restarting the device). The apps whose launch time was measured using this method were produced by iOS versions of React Native and Flutter as well as our baseline native iOS.

PWA launch time measurement on Android and iOS

When installing a PWA on Android, Chrome generates a WebAPK and installs it, thereby enabling the app to be displayed in the app launcher and settings [96]. This makes it possible to track the app's launch time in a fashion broadly similar to that of a native app, although with some points of difference. Most of the differences reflect the fact that a PWA requires a browser in which to run; as a result, the launch time should account for the whole process of launching a browser instance and the PWA. We measured the PWA launch time using the same methodology as a native Android app through *Logcat*, and added to those estimates the overhead of browser launch. In order to ensure comparable conditions, the device was rebooted once, ensuring that no app was running in the background prior to measurements. After each round of measurement, the PWA was killed (without restarting the device) to guarantee a cold launch every time. We did not observe the browser in the list of the running apps that can be killed by the user.

Although iOS allows a PWA to be installed and added to the home screen, there were certain limitations in assessing app performance compared to Android. Amongst the limitations is a lack of profiling support in Xcode Instruments. We were not able to measure PWA launch time on iOS as it requires system-level tooling and inspection that Apple did not support as of iOS 15.0.1 and Xcode 13.0 when we conducted our experiments.

4.2.2 Navigation time measurement

Measuring navigation time was easier than launch time since the process is controlled by the app and does not rely on system-level inspection. We measured the navigation time by logging timestamps right after the touch event that initiated page navigation and after the target page was created. Android native versions' timestamps were logged in Android *Logcat* and iOS native versions were logged in the Xcode Console. For PWA on Android, we used Chrome remote debugging, and for PWA on the iOS platform, we took advantage of the Apple Web-inspector, which made remote debugging possible through Safari.

Measurement for each CPDT started by rebooting the device once and killing all the apps running in the background. After each round of navigation time measurement, the app was killed and opened again (without restarting the device).

4.2.3 Installer size measurement

Installer size was calculated after building and bundling the apps for release and was measured in megabytes (MB). On Android, apps were signed and built for release as an APK. On iOS, apps were built in release mode. In order to have access to the actual files and be able to measure their sizes, we distributed the built app using *ad hoc* distribution and set it to be compatible with the latest iOS device at the time (iPhone 13). A PWA production build (equivalent to release in native versions) was created using the ***build*** command to bundle the code and its dependencies. For this component of our evaluation and to investigate the impact of adding a Service worker on the size of the app, a regular web app was also included and measured in size.

4.3 Device selection

Inspired by Willocx et al. [55] and to examine the effect of employing devices with different levels of capabilities in our assessment, the investigation considered both a newer, more capable device and an older, less capable device for Android and iOS. We will refer to these devices as "high-end" and 'low-end", respectively. The low-end and high-end devices on each platform were selected as widely used and capable devices — in terms of hardware and software — in 2017 (for low-end) and 2020 (for high-end). This work did not attempt to normalize for screen resolution or other factors. An overview of the characteristics and capabilities of these devices is given in Table 4.1. Please note that the selected devices in this chapter are identical to selected devices in Chapter 5 (as listed in Table 5.1).

4.4 Design and implementation of apps

The app designed for this chapter is a two-page application that allows users to navigate from the first screen called "Home" to another screen called "Page A". The home screen has a button centred on the screen that can initiate navigation to the second screen based on a touch event. That second screen (Page A) has a text

		Android	iOS
High-end	**Device**	**OnePlus 8**	**iPhone 12**
	OS	Android 11	iOS 15.0.1
	CPU	Snapdragon 865	Apple A14 Bionic
	RAM	8GB	4GB
	Released in	2020	2020
	Resolution	1080x2400 pixels	1170x2532 pixels
Low-end	**Device**	**Galaxy S8**	**iPhone 8**
	OS	Android 9	iOS 15.0.1
	CPU	Snapdragon 835	Apple A11 Bionic
	RAM	4GB	2GB
	Released in	2017	2017
	Resolution	1080x2220 pixels	750x1334 pixels

Table 4.1: Selected devices for responsiveness and convenience of use evaluation (identical to Table 5.1)

centred on the screen, which prints the name of the screen. Both screens have a header showing the name of the current screen and the second screen (Page A) has a back button to allow the user to navigate back to the first screen (Home).

All the apps were developed based on each platform's official starter project, most widespread programming language and using the recommended IDE. In addition, we followed the best practices and standard way of implementation that was available in the official guide and documentation of the tools we used in this chapter. In cases where functionality was needed that was not included in the starter project, a third-party library that best fits the requirements was added and mentioned.

The native Android version was implemented using Java and Android Studio IDE (version 2020.3.1 Patch 2), along with the latest SDK and developer tools. Screens of this version of the app were built by Activities [97] and an Intent initiated the navigation. The native iOS version was implemented using Swift programming language, UIKit (a framework from Apple that provides necessary classes for building iOS apps [98]) and Xcode IDE (version 13.0). Screens were built by UIViewControllers [99] and interfaces designed by Storyboard.

The React Native versions were built using JavaScript and Visual Studio Code (version 1.60.2) and the React Native (version 0.66.0). We used the newer Functional React components and leveraged React Hooks where needed. React Native framework, for some of its core functionality, relies on modules that are maintained by its community, such as navigation. We added the "React Navigation" library for navigation [100]. Flutter apps were built using Flutter (version 2.5) and Visual Studio Code (version 1.60.2) using the Dart programming language. There was no third-party library used in apps built by Flutter. The recently added

Flutter Navigation 2.0 was adopted for navigation.

The PWA instantiation of the apps was based on the PWA template of "Create-react-app" [101] written in JavaScript and the React library (version 17.0.2) in Visual Studio Code (version 1.60.2). The navigation library react-router-dom [102], was added to the project since React was not shipped with navigation. The "Workbox" library from Google was used to power the app with a service-workers [103]. Beyond the PWA app itself, we also created a regular web app for this work in order to see how PWA-specific code and libraries affect the size of the app. The regular web version is similar to the PWA version without service-worker related scripts and libraries. Both the PWA and Web versions were hosted on Google Firebase [104].

All of the artifacts developed for this chapter are accessible in public open-source repositories on the Github and can be found via the next link:

`https://github.com/Iman-Jamali/thesis-mobile-app-development-cross-platform-vs-native/tree/main/responsiveness-and-convenience`

4.5 Results

This section presents and discusses the results of the responsiveness and convenience experiments. We give a summary of how they were measured, present the results in tables and charts, characterize the respective statistical tests applied, report the results of such tests and, finally, interpret the results.

The measured software performance metrics, in many cases fail to follow a normal distribution [105]. This is particularly true for the results of launch time and navigation time measurements in this chapter. Therefore, a non-parametric statistical test that did not assume normality was employed. The popular Mann-Whitney U test (also known as a Wilcoxon Rank-Sum test) was selected on account of both its non-parametric character and because it allowed for a wide range of sample sizes. Critically for the methodologies employed here, the one-way Mann-Whitney U test could support ranking of different development methods for a given type of measurement.

For each specific scenario consisting of a combination of development method, platform (Android and iOS) and device class (low-end and high-end), the measurement of each metric was repeated 20 times. To compare across scenarios, the ensembles of scenario-specific outcomes were compared by one-way Mann-Whitney U tests.

Specifically, for each combination of platform and device class, results were first ranked by their median values. One-way Mann-Whitney U Tests were then run on each successive pair of results to evaluate if there was a statistically significant difference in the ordering. In addition, for each combination of metric, development method and platform, the effect of device classes was evaluated by running a One-way Mann-Whitney U test. A sample size of 20 (i.e., 20 realizations, each yielding a separate measurement for each metric) and a significance level of 0.05 were applied for all the tests in this chapter. The Mann-Whitney U test can be applied to sample size as small as 5 [106]. The sample size of 20 was selected after considering the time constraints of conducting all tests and the effect of sample size on the statistical test's power. The R

statistical computing platform [107] was used to perform statistical tests. The source code of all the statistical analyses in this chapter is available in Appendix A.

4.5.1 Launch time

We defined the launch time as the elapsed time since the user opened an app until it is rendered and ready for the user's interaction. Measuring this metric required system-level tooling and profiling software that was provided by Android and iOS. We designed and implemented custom apps using multiple CPDTs and a baseline native application for each platform with an identical design and standard way of implementation offered by each development framework. Having the apps and assessment tools, launch time was measured across all the native and CPDTs, with the exception of PWA on iOS due to lack of support from Apple. Mann-Whitney U tests were run on each pair of successive results ranked by their median launch time in order to determine significant differences.

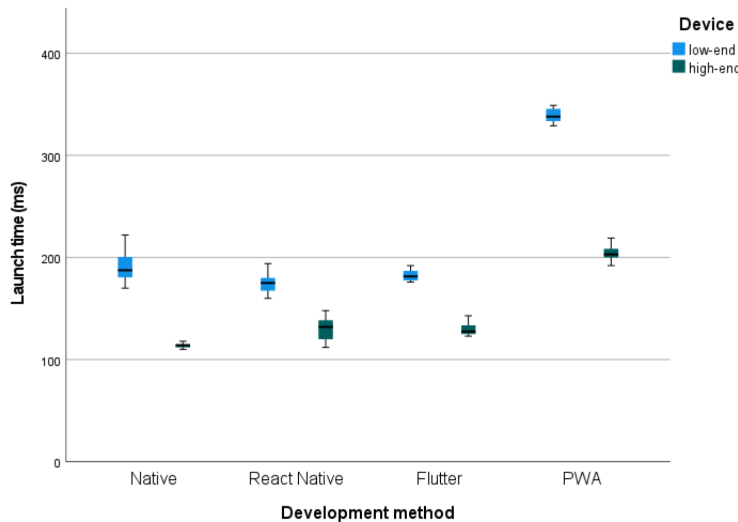


Figure 4.1: Boxplot including median Android app launch time by development method by device. Dots indicate outliers.

Figure 4.1 summarizes the results for median launch time on Android. As the chart shows for Android and on the low-end device, the median launch time for the React Native app was the fastest at 175 ms (s.d. of 9.28 ms), followed by the Flutter app at 182 ms (s.d. of 5.52 ms), and the native app at 188 ms (s.d. of 14.9 ms). The PWA app's median launch time was the longest, at 338 ms (s.d. of 7.11 ms). The Mann-Whitney U test showed a significant difference for launch time results on the low-end Android device between the React Native and the Flutter apps ($W = 89$, $p = 0.001$) and between the native and the PWA apps ($W = 0$, $p < 0.001$). By contrast, no significant difference was found between the Flutter and the native apps ($W = 131.5$, $p = 0.0327$).

Results on the Android high-end device were a little different. The native app was the fastest with a median time of 114 ms (s.d. of 2.23 ms). The Flutter and React Native apps ranked second and third with

nearly identical median times of 128 ms (s.d. of 5.92 ms) and 132 ms (s.d. of 11.3 ms), respectively. The PWA app was the slowest, with a median of 203 ms (s.d. of 6.56 ms). The Mann-Whitney U test revealed a statistically significant difference in launch time results on the high-end Android device between the native and the Flutter apps ($W = 0$, $p < 0.001$) and between the React Native and the PWA apps ($W = 0$, $p < 0.001$). No significant difference was found between Flutter and React Native ($W = 171.5$, $p = 0.223$).

For each development method, statistically significant differences were seen in the Android median launch time between the low-end and high-end devices. The PWA app median time decreased from 338 ms (s.d. of 7.11 ms) on the low-end device to 203 ms (s.d. of 6.56 ms) on the high-end device (a 39.9% decrease). A similar performance impact was observed for the native applications, where the median time declined from 188 ms (s.d. of 14.9 ms) on the low-end device to 114 ms (s.d. of 2.23 ms) on the high-end device (a 39.2% decrease). For the Flutter app, the device class exhibited a somewhat lower impact on launch time, with the median time decreasing from 182 ms (s.d. of 5.52 ms) on a low-end device to 128 ms (s.d. of 5.92 ms) on a high-end device (a decrease of 29.7%). Lastly, the React Native apps also exhibited a somewhat lower impact of a device class, with the median launch time decreasing from 175 ms (s.d. of 9.28 ms) on a low-end device to 132 ms (s.d. of 11.3 ms) on a high-end device (a 24.5% decrease). Despite the somewhat variable quantitative impact of device class on launch times across development methods, the Mann-Whitney U test revealed a statistically significant difference in median launch time between the two device classes for all of the apps, including for the native app ($W = 0$, $p < 0.001$), the React Native app ($W = 0$, $p < 0.001$), the Flutter app ($W = 0$, $p < 0.001$) and the PWA app ($W = 0$, $p < 0.001$).

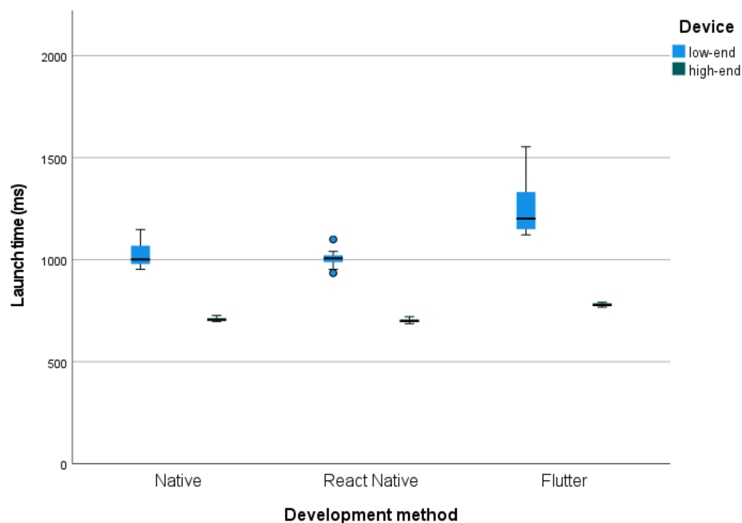


Figure 4.2: Boxplot including median iOS app launch time by development method by device. Dots indicate outliers.

The section above discussed launch times on Android devices. Launch time results for iOS are summarized in Figure 4.2. As illustrated, amongst the median launch times on the iOS low-end device, native and React Native implementations were fastest by 1002 ms (s.d. of 60.5 ms) and 1006 ms (s.d. of 45 ms), respectively.

Flutter was slower than the other two, finished loading an app in 1202 ms (s.d. of 149 ms). A statistically significant difference for launch time results on the low-end iOS device was reported by Mann-Whitney U test between React Native and Flutter ($W = 2$, $p < 0.001$). By contrast, no significant difference was found between native and React Native ($W = 221$, $p = 0.719$).

For the high-end iOS device, the median launch time for the development methods ranked as follows: React Native was the fastest at 700 ms (s.d. of 9.46 ms), followed closely by native at 706 ms (s.d. of 18.5 ms) and, lastly, Flutter at 778 ms (s.d. of 13.9 ms). With respect to the pairwise comparison of launch times for different development methods on the high-end iOS device, the Mann-Whitney U test revealed a statistically significant difference between React Native and native ($W = 110$, $p = 0.007$) and between native and Flutter ($W = 13$, $p < 0.001$).

On iOS as with Android, for the median launch time of each development method, significant differences were seen between the low-end and high-end devices. When switching from a low-end to a high-end device, the greatest decrease in median launch time was seen in Flutter, which decreased from 1202 ms (s.d. of 149 ms) on the low-end device to 778 ms (s.d. of 13.9 ms) on the high-end device (a 35.2% decrease). Next, React Native median time declined from 1006 ms (s.d. of 45 ms) on the low-end device to 700 ms (s.d. of 9.46 ms) on the high-end device (a 30.4% decrease). Using the native development method, the median launch time decreased from 1002 ms (s.d. of 60.5 ms) on the low-end device to 706 ms (s.d. of 18.5 ms) on the high-end device (a 29.5% decrease). With respect to the results obtained across device classes, the Mann-Whitney U test showed statistically significant differences in launch time for applications developed with all development platforms, including native development ($W = 0$, $p < 0.001$), React Native ($W = 0$, $p < 0.001$), and Flutter ($W = 0$, $p < 0.001$).

4.5.2 Navigation time

Navigation time is measured from the time that the navigation event is initiated to the time page is loaded and ready for interaction. We recorded the time by logging a timestamp at the start and end of the navigation in each of the apps of this experiment. For logging, native and generated native apps (i.e., Flutter and React Native apps) on Android and iOS, Logcat and Instruments tools were used, respectively. For the PWA app, Chrome and Safari remote debugging capabilities were used instead. Mann-Whitney U tests were run on each pair of successive results ranked by their median launch time, so as to determine significant differences.

Figure 4.3 summarizes the results for median navigation time in Android. As seen in the chart for Android and on the low-end device, the fastest median navigation time was for the Flutter app with 7.5 ms (s.d. of 3.57 ms), closely followed by the PWA app with 10 ms (s.d. of 1.41 ms). Next, we have the React Native app with 20 ms (s.d. of 2.8 ms). The native app was the slowest with 44 ms (s.d. of 2.51 ms). The Mann-Whitney U test showed a significant difference for navigation time results on the low-end Android device between the PWA and React Native apps ($W = 0$, $p < 0.001$) and between the React Native and native apps ($W = 0$, $p < 0.001$). By contrast, the Mann-Whitney U test identified no significant difference in navigation time

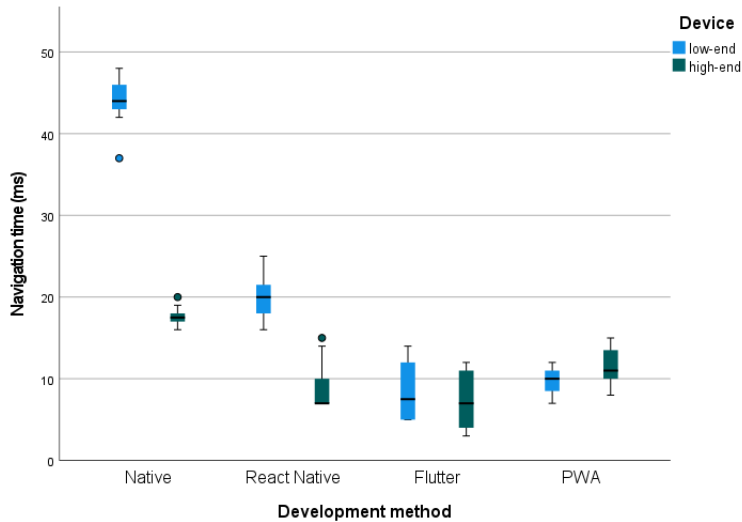


Figure 4.3: Boxplot including median Android navigation time by development method by device. Dots indicate outliers.

between the Flutter and PWA apps on the low-end Android device (W = 138.5, p = 0.047).

As shown in Figure 4.3 and looking at the median navigation time for Android and on the high-end device, the Flutter and React Native apps were the fastest, both had a median time of 7 ms (s.d. of 3.78 ms and 2.83 ms, respectively). The PWA app median time was 11 ms (s.d. of 2.04 ms), followed by the slowest app, native by 17.5 ms (s.d. of 0.93 ms). The Mann-Whitney U test showed a significant difference in navigation time results on the high-end Android device between the React Native and PWA apps ($W = 79.5$, $p < 0.001$) and between the PWA and native apps ($W = 0$, $p < 0.001$). That test identified no statistically significant difference between the Flutter and React Native apps on the high-end Android device (W = 151, p = 0.09).

For the median navigation time of each development method's app in Android, statistically significant differences were seen between the low-end and high-end devices. The median navigation time of the two apps was considerably faster on the high-end device than on the low-end; the median navigation time of the React Native app decreased from 20 ms (s.d. of 2.8 ms) on the low-end device to 7 ms (s.d. of 2.83 ms) on the high-end device (a 65% decrease) and for the native app from 44 ms (s.d. of 2.51 ms) on the low-end device to 17.5 ms (s.d. of 0.93 ms) on the high-end device (a 60.2% decrease). The Flutter app's median navigation time decreased from 7.5 ms (s.d. of 3.57 ms) on the low-end device to 7 ms (s.d. of 3.78 ms) on the high-end device (a 6.6% decrease). Finally, and counter-intuitively, the PWA app's median navigation time increased from 10 ms (s.d. of 1.41 ms) on the low-end device to 11 ms (s.d. of 2.04 ms) on the high-end device (a 10% increase). The Mann-Whitney U test showed a significant difference between the device class for the native app ($W = 0$, $p < 0.001$), the React Native app ($W = 0$, $p < 0.001$) and the PWA app (W = 294, p = 0.004). By contrast, the modest decrease in navigation time observed with the Flutter app (W = 145, p = 0.067) was not found to be statistically significant.

The navigation time for iOS is shown in Figure 4.4. Results on the low-end iOS device show that the

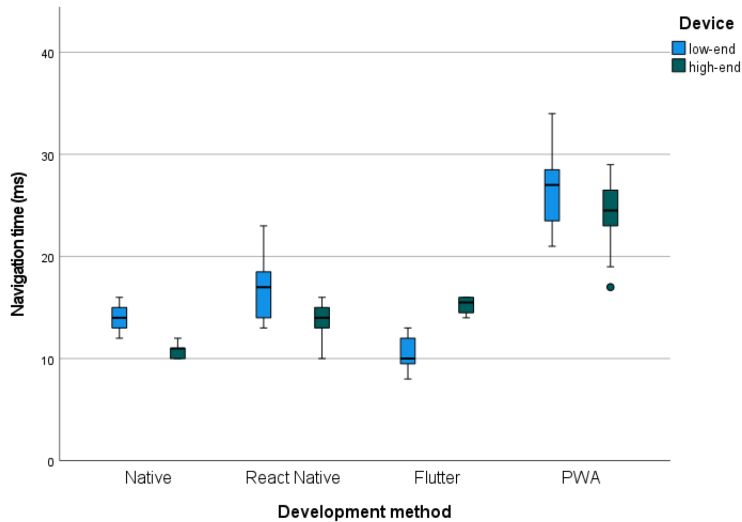


Figure 4.4: Boxplot including median iOS navigation time by development method by device class. Dots indicate outliers.

Flutter app's median navigation time was the fastest at 10 ms (s.d. of 1.50 ms). The runner-up was the native iOS app at 14 ms (s.d. of 1.44 ms), which was itself closely followed by the React Native app with 17 ms (s.d. of 2.68 ms). The slowest of all, the PWA app, had a median navigation time of 27 ms (s.d. of 3.28 ms). The Mann-Whitney U test demonstrated a significant difference for the navigation time results on the low-end iOS device between the Flutter and native apps ($W = 21$, $p < 0.001$), between the native and React Native apps ($W = 72$, $p < 0.001$), as well as between the React Native and PWA apps ($W = 4$, $p < 0.001$).

As shown in Figure 4.4, the results for median navigation time on the iOS high-end device demonstrated that the native app had the fastest navigation time, with a median of 11 ms (s.d. of 0.74 ms). The React Native and Flutter apps were next, with medians of 14 ms (s.d. of 1.92 ms) and 15.5 ms (s.d. of 0.85 ms), respectively. As with the low-end device, the PWA app exhibited the longest median navigation time, with 24.5 ms (s.d. of 3.08 ms). The Mann-Whitney U test demonstrated a significant difference in the navigation time results on the high-end iOS device between the native and React Native apps ($W = 50$, $p < 0.001$), between the React Native and Flutter apps ($W = 97.5$, $p = 0.002$) and between the Flutter and PWA apps ($W = 0$, $p < 0.001$).

Statistically significant differences were seen in median navigation time between the low-end and high-end devices for each development method's app in iOS. The greatest decrease in median navigation time from the low-end to the high-end iOS device was seen for the native app, with a 21% decrease, from 14 ms (s.d. of 1.44 ms) on the low-end device to 11 ms (s.d. of 0.74 ms) on the high-end device. The React Native app's median navigation time decreased from 17 ms (s.d. of 2.68 ms) on the low-end device to 14 ms (s.d. of 1.92 ms) on the high-end device (a 17.6% decrease). The PWA app exhibited a relatively moderate decrease from 27 ms (s.d. of 3.28 ms) on the low-end device to 24.50 ms (s.d. of 3.08 ms) on the high-end device (a decrease of 9.2%). The Flutter app was the only app that showed an increase in median time, from 10 ms

(s.d. of 1.5 ms) on the low-end to 15.5 ms (s.d. of 0.85 ms) on the high-end device (a 55% increase). The Mann-Whitney U test showed a significant difference between the device class for the native app ($W = 8$, $p < 0.001$), React Native app ($W = 78$, $p < 0.001$). The counter-intuitive increase in mean navigation time observed for the Flutter app ($W = 400$, $p < 0.001$) was also found to be statistically significant. By contrast, the decrease observed for the PWA app ($W = 136.5$, $p = 0.043$) was found to not be statistically significant.

4.5.3 Size of installer

Installer size is defined as the size of the files necessary for installing the app on a device, as generated by building an app for release. The native and the generated native Apps (i.e., React Native and Flutter) were built using Android studio and Xcode for Android and iOS, respectively. The PWA app was built by React for production (release). In addition, we also included a regular web app, which is the same app as for PWA, but without its PWA-related files and modules.

	Android	**iOS**
Native	4.8	1.2
React Native	32.5	5
Flutter	16.7	7.9
PWA	0.98	
Web	0.71	

Table 4.2: Size of installer files (in MB) for different methods of development

The size of installer file is listed in Table 4.2. Please note that the web-based approaches, PWA and web, had only one app that ran on both platforms, hence only one size is listed in the table. These web-based apps had the smallest installer size — 0.98 MB for the PWA app, and 0.71 MB for the regular web app. On Android, the native app required 4.8 MB. At a size of 16.7 MB, the Flutter app on Android was nearly four times larger than that native app. The Android React Native app was the largest, with a size of 32.5 MB — more than six times larger than the baseline native app. When compared to their Android counterparts, all the iOS apps were smaller, with the native iOS app measuring 1.2 MB, followed by the React Native app with 5 MB. The Flutter app on iOS had the largest installer size, at 7.9 MB.

4.6 Discussion

This section first reports the main results in responsiveness and convenience of use experiments, and then interprets these findings and discusses the contributing factors to such results. The section subsequently discusses the overall results for each development method and, finally, the overall results are compared to the findings of previous work.

The responsiveness and convenience of use experiments conducted in this chapter yielded seven main results:

- Each development method exhibited significantly faster launch times on the high-end device than for the low-end device on both Android and iOS.
- PWA launch time was significantly slower than for the other development methods.
- Navigation times of all the CPDTs on the Android platform were significantly faster than for native Android.
- The navigation time impact of device class on Flutter and PWA was quite different than for the other development methods on both platforms. Navigation time was lowered from low-end to high-end devices for most methods, but not reliably so for Flutter and PWA.
- PWA and native (Android and iOS) installer files were significantly smaller in size than those for Flutter and React Native. As a matter of side note, for those CPDTs producing platform-specific installers, the installer files for iOS were also significantly smaller than those for Android.

4.6.1 Explanation and interpretation of results

Why were the median launch times of the development methods for both platforms significantly faster on high-end devices in comparison to their low-end counterparts? Two key factors can help us better understand the effect of device class on launch time: the steps involved in the launch process, and how device specifications can affect the process.

The launch time results were based on cold start measurement, a situation when the app's process is not already alive (i.e., not in memory) and needs to be created by the system from scratch. In this initial step, frameworks, daemons and other app dependencies need to be loaded and paged in from the disk. As the process is created, it takes over and is responsible for the next steps. Subsequent steps on iOS include UIKit initialization [108], scene rendering and initial frame rendering. On Android, the ensuing steps are app object creation, launching the main thread and main activity, inflating views, laying out the screen, and, finally, performing the initial draw [109, 95].

Initializing the app's inner process components, creating the views, calculating view positions, and laying them out on the screen are amongst the processor- and memory-intensive tasks. In addition, the final steps, such as rendering the views and performing the initial draw, demand GPU power.

The devices considered in this chapter's experiments exhibited different specifications and levels of capabilities. The high-end devices were released more recently — in 2020 — in contrast with the earlier (2017) release year for the low-end devices. Those high-end devices were equipped with more capable CPUs, memory and SSD (a solid-state storage device that stores data persistently) units. These, amongst other things, contributed to the high-end devices achieving significantly faster launch times.

Why was PWA launch time was significantly slower than for other development methods on Android? PWA, unlike other development methods that launch directly by the operating system, relies upon the browser to get started. Dependency on another app (browser) introduced an overhead in the PWA launching process which made PWA's launch time 80.2% slower on the low-end devices and 53.7% slower on the high-end device compared to the next slowest method.

Why was the navigation time of native Android significantly slower than for all the other CPDTs on the Android platform? Navigation in the native Android app was based on recreating "Activity", a top-level application component in Android. In the native Android app, each page is hosted by an Activity component that is recreated each time the user navigates to a new page. This means that in navigating from one page to another, the Activity that hosts the initial page should be destroyed, and the Activity of the new page needs to be created. Recreating a top-level component during navigation was one of the most important factors contributing to the slower navigation time of the native Android app. Other CPDTs adopt a more optimized navigation method by reusing the top-level component.

The React Native app used React Navigation, a community-driven navigation library for React Native. On the Android platform, under the hood, React Navigation uses fragments [110] (a reusable component hosted by an Activity) to navigate from one page to another. By contrast, the Flutter app did not rely on the platform's native UIs, and had its own navigation and UIs. The Flutter app leveraged the "MaterialPageRoute" class [111] for navigation, which is a modal mechanism that replaces screens with platform-adaptive transitions. The PWA app was a Single Page Application that dynamically recreates a page according to the data passed to that page.

The Android native app's approach of creating a separate top-level component (Activity) was a more demanding technique and negatively impacted its navigation time compared to CPDTs using more optimized ways of handling navigation within a single top-level component.

Why did Flutter and PWA navigation time results exhibit distinct impacts of device-class compared to the other development methods? When considering navigation time on Android, changing from a low-end to a high-end device for the native and React Native apps lowered the navigation time by 60.2% and 65%, respectively; by contrast, this change on Flutter yielded only a 6.6% decrease, while for PWA the switch from a low-end to a high-end device yielded a 10% increase in navigation time. Corresponding results on iOS demonstrated that transitioning from a low-end to a high-end device achieved a 21.4% and 17.6% decrease in navigation time for native iOS and React Native, a decrease of just 9.2% for PWA, while for Flutter, the change to a high-end device resulted in a 5.5% increase in navigation time. Two factors, amongst other things, could contribute to the observed differences between the CPDTs.

Navigation in native and React Native apps was based on the native-provided APIs — Activity and Fragment (A reusable UI component) on Android and UINavigationController (a top-level component that provides navigation) on iOS. On the other hand, Flutter relied upon its own navigation and PWA's navigation

was provided by its hosted browser app. The greater capabilities of the high-end devices could have had a more significant effect on the apps with native-based navigation — such as native Android, native iOS and React Native – than the ones with alternative navigation methods such as those provided by Flutter and PWA.

Another contributing factor was the larger number of pixels on the high-end devices, especially on iOS. As shown in Table 4.1, there was a 196% increase in the number of pixels in the iOS high-end devices in comparison to their low-end counterparts; by contrast, for Android devices, there was only an 8.1% increase in pixels for high-end relative to low-end devices.

Why were installer file sizes for PWA and native apps smaller than those for React Native and Flutter? Native apps that are built by the software development kit (SDK), tools and programming languages that are provided by their platforms often use the frameworks, libraries and APIs that are already available on their respective platforms. As a result, the size of these apps tends to be relatively small compared to those produced by cross-platform tools like React Native and Flutter that use alternative solutions to generate apps for more than one platform. These CPDTs are bundled with libraries, frameworks and other components that enable them to provide cross-platform functionalities. Such dependencies are a major factor in increasing the size of installer files. In some cases, such CPDTs have to ship with dependencies required only for one platform. For instance, React Native app was bundled with JavaScriptCore, a JavaScript engine that executes JavaScript code, on the Android version while its iOS version exhibits no need to be shipped with JavaScriptCore since it is already part of the iOS platform [112]. This was one reason that the React Native installer file on Android exhibits is larger than its respective installer on iOS (32.5 MB on Android, 5 MB on iOS).

The installer size of the PWA app was even smaller than native apps — 79.58% smaller than the native Android app and 18.33% smaller than the native iOS app. This smaller size of the PWA app reflected a number of contributing factors. The PWA app is native to the browser and delegates some of the complexity of dealing with the underlying platform to its hosting browser. In addition, the browser ships with many features and APIs which the app uses by leveraging DOM (a programming interface for web documents in the browsers). In addition, HTML5 provides the app with access to many device-specific features. All in all, running in the browser's run-time environment and taking advantage of many features, APIs and device-specific features that are already shipped by the browser, and the higher-level specification involved, amongst other factors, contributed to the smaller installer size of the PWA app.

What is the perceived responsiveness associated with our launch time results? According to Google, five seconds or longer is considered an excessive cold startup time; corresponding values are two seconds or longer for warm startup and 1.5 seconds or longer for hot startup [113]. The launch time results from the experiments in this chapter (all based on cold launch) showed that even the slowest CPDT on each platform's low-end device (PWA on Android with 338 ms and Flutter on iOS with 1202 ms) were faster than

what is considered an excessive time for a hot start – the fastest launch type.

In addition, Jakob Nielsen, a human-computer interaction researcher, defined the limit for the operations in which a user's flow of thought stays uninterrupted as 1.0 seconds, although delay can be noticed [114]. With the exception of the low-end iOS device, all the recorded median launch time results for all methods of development were under the 1-second limit.

What is the perceived responsiveness in our navigation time results? Jakob Nielsen, in his book "Usability Engineering", defines the time limit for the system's response to be felt as instantaneous by the user as 0.1 seconds [114]. Results in this chapter indicated that the slowest navigation time on Android was 44 ms for native Android on the low-end device and 27 ms for PWA on the iOS low-end device. The median navigation times of all the development methods in this chapter's experiments are considered "instantaneous" according to the Nielsen definition, with no delay being felt by the user when transiting between pages.

4.6.2 Overall results for each development method

The results of the experiments in this chapter showed that the native iOS provided the best overall responsiveness and convenience of use on the iOS platform. The iOS native app had the fastest launch time on the low-end device and the second fastest on the high-end device. Its navigation time results showed that it was the fastest on the high-end device and the second fastest on the low-end device. Moreover, the size of its installer files was significantly smaller than for React Native and Flutter, and only slightly larger than for PWA.

Overall results for the Flutter and React Native apps were very similar. Together, these ranked as the most favourable development method in Android and runner-up in iOS. Launch time results of Flutter and React Native apps on Android were the fastest on the low-end device and second fastest on the high-end (launching slightly slower than native). The Flutter and React Native apps were the fastest in terms of navigation time on Android, with the exception of React Native on the low-end device, which was slower than PWA. On the iOS platform, their overall navigation time results were the fastest after the native iOS app. They both had the largest installer size, with Flutter being the largest on iOS, and React Native the largest on Android.

Overall results for the native Android app made it less favourable than React Native and Flutter but better than PWA on the Android platform. The native Android app launch time results were the fastest on the high-end device but slightly slower than React Native and Flutter on the low-end device. The native Android app was the slowest in navigation time on both the low-end and high-end devices. The size of its installer files was significantly smaller than for React Native and Flutter, but larger than for PWA.

Overall results in this chapter showed that PWA responsiveness and convenience of use are less favourable than for other development methods. PWA's launch time results on Android — the only platform on which it was measured — were the slowest. Navigation time results on Android were faster than for native apps but

slower than for others. Its navigation time results on iOS were the slowest of all the development methods. Despite being encumbered by slow launch and navigation times, PWA had the smallest installer size.

4.6.3 Overall results compared to previous work

Another important element to better understand CPDT responsiveness and convenience trade-offs is to compare this chapter's overall results to similar previous work.

Amongst the more recent work and the closest to our work in this chapter is a study by Willocx et al. [55]. Amongst the CPDTs measured in terms of launch time in their study, Titanium and Xamarin were the closer CPDTs in terms of architecture to React Native and Flutter, respectively, and Ionic was the closest to PWA, as two are web-based technologies. Only considering the results on the high-end devices, Titanium and Xamarin launch times were 179% and 203% above that of native on Android (respectively) and 73% and 81% above that of native on iOS, respectively. The similar results in our study for React Native and Flutter were 15% and 11% lower that of native Android, respectively, and 0.7% above and 10% lower that of native on iOS, respectively. In their work, Ionic was 318% lower that of native on Android, while for the PWA in our study, it was 78% lower that of native on Android. An earlier work by Willocx et al. [8] also had similar results.

Navigation time results in Willocx et al. [55] study comparing to the same results in this work exhibited similar differences between the development methods, yet overall navigation time results tend to be slower than in our work. For example, in their study, they measured native Android and iOS navigation time at 91ms and 28 ms, respectively. The similar results in our experiment on the high-end device were 17.5 ms for Android and 11 ms for iOS. Earlier work by Willocx et al. [8] yielded similar results.

In addition, Biørn-Hansen et al. [21] reported a smaller PWA app size compared to React Native and Ionic apps, similar to the findings in this chapter.

The aforementioned comparison of the launch time and navigation time results between this work and that of previous studies demonstrates that launch time and navigation time differences between native and CPDTs have become considerably smaller in the newer generation of CPDTs. This, amongst other things, reflects the advancement of hardware and software of newer generations of Android and iOS smartphones as well as the introduction of modern CPDTs (such as React Native, Flutter and PWA) that generate apps that are more responsive and convenient for use than the older generation of CPDTs.

4.7 Summary

This chapter aimed to investigate CPDT trade-offs from a responsiveness and convenience of use standpoint.

First, criteria and parameters that can support evaluation of perceived responsiveness and convenience of use were discussed, and then three quantitative metrics were introduced. The first was the size of the installer, defined as the size (in megabytes (MB)) of the app executable file and any necessary files included

in the required app download after the app is built in release mode (or built for production in web-based apps). The second metric was the launch time, defined as the amount of time required from the moment the user presses the app's icon to when the app is ready to respond to the user's interaction. The final metric considered in this area was navigation time, defined as the time elapsed between user initiation of navigation and the point at which the destination page is loaded and ready for interaction.

Next, measurement tools and methods for measuring the defined metrics across the apps built for the evaluation in this chapter were discussed. This chapter's launch time measurements were based on a cold start. A cold start is undertaken when an app is not already loaded in memory. Apps in this process start from scratch [93]. The launch time on the Android platform for all the apps was logged with *Logcat* command-line tool. The "App Launch" template of Xcode Instruments was used to profile and measure the launch time of all the iOS apps except PWA. Due to a lack of profiling support in Xcode Instruments for PWA on the iOS platform, we could not measure its launch time. This chapter measured navigation time by logging timestamps right after the touch event initiated page navigation and after the target page was created. On Android native apps, timestamps were logged in Android *Logcat*; on iOS native apps, timestamps were logged in the Xcode Console. For PWA's navigation time measurement on Android, we used Chrome remote debugging; on iOS, the Web-inspector in Safari was used via remote debugging. The installer sizes were calculated after building and bundling the apps for release, and were measured in megabytes. Tests on each platform (Android and iOS) were conducted on a newer, more capable device and an older, less capable device.

The app designed for this chapter is a two-page application that allows users to navigate from one screen to a second screen. All of the apps were developed based on each platform's official starter project, most widespread programming language, and using the recommended IDE. All of the artifacts developed for this chapter are accessible in public open-source repositories on Github.

Thereafter, results were presented, interpreted, and findings were discussed. Results showed that the native iOS app provided the best overall responsiveness and convenience of use on the iOS platform. Overall results for the Flutter and React Native apps were very similar to each other. Together, these ranked as the most favourable development method in Android, and the runner-up in iOS. The native Android app's overall results made it less favourable than React Native and Flutter but better than PWA on the Android platform. Overall results in this chapter suggest that PWA app responsiveness and convenience of use are less favourable than those of apps produced by other development methods.

The comparison of the launch time and navigation time results between this work and that of previous studies in Section 4.6.3 demonstrated that launch time and navigation time differences between native and CPDTs have narrowed considerably in the newer generation of CPDTs. This, amongst other things, reflects the advancement of hardware and software of newer generations of Android and iOS smartphones, as well as the introduction of modern CPDTs (such as React Native, Flutter and PWA) that generate apps that are more responsive and convenient for use than those implemented with the older generation of CPDTs.

5 Resource Use and Performance

This chapter introduces the second component of our evaluation system to support investigation of the CPDT (cross-platform development tool) trade-offs — a component associated with resource use and performance. In this chapter, we first start by defining criteria and parameters that can be measured to evaluate resource use and performance. The chapter then discusses measurement tools and the adopted methods to measure metrics associated with defined parameters across native and CPDTs. The chapter continues on to characterize devices on which experiments were conducted, and the criteria by which they vary. We subsequently examine the design and implementation of the apps built and the assessment tests in this chapter. Thereafter, experiment results are presented, interpreted and findings are discussed.

5.1 Evaluation criteria and metrics

In this section, we discuss the importance of resource use and performance as key aspects in evaluating CPDTs. This section revisits some of the peer-reviewed related work with respect to resource use and performance and tries to explain how the work presented here sought to expand such previous work. The section continues on to introduce and define a range of metrics by which we can quantitatively measure resource use and performance, and then discusses the impact of and reasons for selecting each such metric.

Amongst the first resource use and performance-related mobile CPDT work was the study of Corral et al. [5], which sought to compare a CPDT (PhoneGap) and Android native in terms of computational resource demand of a mobile device within different categories: hardware, network and data access. Their study reported an overall performance penalty for the CPDT compared to native, though they only measured execution time and not how the level of device resources used. In another study, Dalmasso et al. [6] evaluated the memory, CPU and power consumption of two CPDTs (PhoneGap and Titanium) on Android. They reported that PhoneGap used less memory, CPU and power than Titanium. Their study did not include a baseline native comparator, and only recorded resource use in a test that is best characterized as network-intensive; by contrast, no performance-related metrics were included.

In a component of their studies, Willocx et al. [8, 55] measured resource consumption across a range of CPDTs. They reported higher CPU usage and memory footprint by CPDTs, though their results were not based on resource-intensive specific tests, and instead measured system behaviour during the responsiveness-related activities such as app launch and navigation.

Amongst more recent studies, Biørn-Hansen et al. [58] scrutinized the resource usage and performance

overhead of multiple CPDTs compared to native development on Android. In their work, they designed apps that measured the performance and resource use of access to native-side function interface calls, including the geolocation API, contacts API, file system and accelerometer. Measurements in their work emphasized metrics such as test execution time and CPU and memory usage. They reported a decrease in performance and higher resource use of CPDTs in comparison to the native approach. They also reported that some of the CPDTs can perform equally or better than native on certain metrics.

Building on insights and measures from the aforementioned studies and to better assess our criteria in this section, we designed and implemented several resource-intensive tests that target several of the most important resources of a device, and measured both resource usage and performance-related metrics.

The following metrics are identified and defined for the assessments in this chapter.

5.1.1 CPU usage

The Central Processing Unit (CPU) is the core of most modern computing devices, and is responsible for processing and executing instructions. The processing power of a CPU is one the most important resources of mobile devices and impacts performance, responsiveness, battery life and health of a device, amongst other factors. A CPU-intensive program with a high level of CPU usage can cause issues, including longer load time, "freezing" of a device, loss of data, and higher device temperature.

CPU usage is a metric to quantify the processor load by running computer programs. It is defined as the percentage of total CPU capacity at a given moment. All of the resource-intensive tests examined here involved measuring CPU usage.

5.1.2 Memory footprint

Computer Memory (also called main memory) is a device or system that is used to store information for immediate use in a computer [115]. It holds data and instructions on which the computer is currently working, and has a finite data storage capacity.

The CPU relies upon memory to read and provide data for processing, making memory a key factor in device performance. Each program will require allocating a certain amount of memory to run properly. A smaller amount of available memory will cause programs to run slower and, in some cases, stop them from running. Shortage of main memory, amongst other things, would force an operating system to start transferring pages of data from the main memory to non-volatile storage such as SSD, which is slower than main memory. Memory is a particularly valuable resource on a mobile device with a relatively small memory capacity. For such reasons, an app's memory footprint is a major factor in resource use and performance.

The memory footprint of an app is defined as the amount of main memory that a program uses or references whilst running [116]. This metric was measured in all of the resource-intensive tests.

5.1.3 Total data transferred

Most of today's mobile devices are constantly connected to networks, via which they send and receive data. This is especially important to phones with data-driven apps that are often connected to a remote server. Because of the limited bandwidth associated with network connections, transferring successively larger amounts of data over the network requires successively longer amounts of time. In addition, higher network usage can impose added costs on the users.

We defined a metric termed "total data transferred" as the total data received and sent over a network within a specific period of time, as measured in megabytes (MB). This metric is measured and reported here only for network-intensive tests.

5.1.4 Execution time

In order to evaluate the performance of each of the development methods with respect to one of the resource-intensive tests in this chapter, we measured the time each development method required to complete that test.

Execution time is the time elapsed between the start of a resource-intensive test and its completion. This metric was measured in all of the resource-intensive tests.

5.2 Measurement tools and methods

In this section, we describe the methods which we adopted to measure CPU usage, memory footprint, total data transferred and execution time in the resource-intensive tests, and introduce the measurement tools employed to perform such measurements. Before doing so, a few points of background understanding will be useful.

Debug and release versions of apps work differently in a number of ways. While many of such differences serve to ease the development process, such differences can also affect our measurements and threaten the validity and generalizability of our results. To provide a close analogue to real-world conditions, we therefore based all our measurements on the release build of our apps.

To provide better comparability in results, all measurements in this chapter are performed after a cold launch, with all the apps in the background killed prior to the test. Due to time constraints, we did not restart the devices before or after measurements in this chapter.

The apps we are seeking to measure in this chapter are built using both Flutter and React Native — systems that generate native apps to run on Android and iOS — plus native Android and iOS apps for baseline comparison. In addition, this chapter measures the characteristics of our progressive web-based app (PWA) that run on these two platforms within the runtime of the browser.

5.2.1 CPU usage measurement

The CPU usage measurement method on Android was used across the board for the Android versions of our apps built for this chapter, including React Native, Flutter and native Android as well as PWA running on Android. In order to measure CPU usage on Android, the Linux "top" was used. Top (table of processes) is a task manager program that runs on many Unix-like operating systems[117]. It updates periodically at a certain interval, lists currently running processes, and displays information such as CPU and memory usage. Android Debug Bridge (ADB) was used to access the device's shell and execute the Top Command from a connected computer. The Linux/Unix utility "top" was used to report the percentage of CPU usage for the app's process. Please note that CPU usage in "top", for multi-core systems, could have percentages that are greater than 100%. For instance, if 2 cores are at 80% use, "top" will show a CPU usage of 160%.

Measuring CPU usage on iOS for the native and iOS versions of React Native and Flutter apps was possible by the use of "Instrument", an Xcode tool for analyzing performance and measuring a range of parameters. "Time Profiler" in Instrument tracked the CPU usage of the app's process.

At the point of writing this thesis, PWA, running on iOS, cannot be tracked in Instrument in a similar fashion to a native app. Therefore, we relied on Safari's remote debugging through the Apple Web-inspector. Web-inspector in Safari features "Timelines", a tool that tracks activities of an open Safari page and provides insights such as CPU usage and memory allocations, amongst other things.

For each resource-intensive test, the CPU usage expressed as a percent was sampled every 500 ms, the shortest common time between the measuring tools, and the mean of all the samples in a single test was calculated as the mean CPU usage.

In addition, it should be acknowledged that CPU usage of the systems with heterogeneous CPU topology [118], such as those in devices listed in Table 5.1, are difficult to measure and compare between such devices, due to the variety of different specialized processors that are available on such systems on a chip. Measurement tools adopted in this work such as "top" only provide a broad measure.

5.2.2 Memory footprint measurement

Measurement of memory footprint was undertaken by similar measurement tools as CPU usage. On Android, for all the apps — including native, React native, Flutter and PWA running on Android — the ***top*** command, lists memory-related metrics. Amongst these, "RES", reporting the non-swapped physical memory[117] was measured, being viewed as the closest to the actual memory footprint of the app's process.

On iOS, for native, React Native and Flutter apps, "Allocations" in Instrument tracked the total memory allocated by the app. For PWA on iOS using the Safari Web-inspector, this work relied upon the total memory usage reported under the "Memory" tab (page [119].

For each resource-intensive test, memory usage in megabytes (MB) was sampled every 500 ms, the shortest common time between the measuring tools. The mean of all the samples in a single test was calculated as

the mean memory footprint.

5.2.3 Total data transferred

On Android, for native, React Native and Flutter apps, Android Studio's Profiler was used to assess the total data sent and received over the network during the test. For PWA apps running on Android and relying on Chrome remote debugging, we measured the total data transferred from the "Network" tab of the Chrome Developer Tools.

On iOS, for native, React Native and Flutter apps, "Network Connections" in Instrument reported the total transmitted and received data. To measure the total data transferred for PWA running on iOS, the "Network" tab of the Safari Web-inspector was used.

Due to its relevance, total data transferred was only measured in the network-intensive tasks. It was measured with units of megabytes (MB).

5.2.4 Execution time

In contrast to other measurements in this chapter, the execution time was controlled by the app and did not rely on system-level inspection. For each resource-intensive test, the execution time was measured by logging elapsed time in milliseconds, between the moment the button that starts a test is pressed and the moment that test completes. Android native versions' execution times were logged in Android *logcat*; iOS native versions were logged in the Xcode Console. For PWA on Android, we used Chrome remote debugging to monitor the console. On iOS, we took advantage of the Apple Web-inspector, which made remote debugging possible through Safari.

5.3 Device selection

To examine the effect of employing devices with different levels of capabilities in our assessment, this study conducted tests on two devices: a newer, more capable device and an older, less capable device for Android and iOS. We will refer to these devices as "high-end" and "low-end", respectively. The low-end and high-end devices on each platform were selected as widely used and capable devices — in terms of hardware and software — in 2017 (for low-end) and 2020 (for high-end). As was noted earlier, no attempt was made to secure comparable display resolutions between devices. An overview of the characteristics and capabilities of these devices is listed in Table 5.1. Please note that the selected devices in this chapter are identical to selected devices in Chapter 4 (listed in Table 4.1).

		Android	iOS
High-end	**Device**	**OnePlus 8**	**iPhone 12**
	OS	Android 11	iOS 15.0.1
	CPU	Snapdragon 865	Apple A14 Bionic
	RAM	8GB	4GB
	Released in	2020	2020
	Resolution	1080x2400 pixels	1170x2532 pixels
Low-end	**Device**	**Galaxy S8**	**iPhone 8**
	OS	Android 9	iOS 15.0.1
	CPU	Snapdragon 835	Apple A11 Bionic
	RAM	4GB	2GB
	Released in	2017	2017
	Resolution	1080x2220 pixels	750x1334 pixels

Table 5.1: Selected devices for resource use and performance evaluation (identical to Table 4.1)

5.4 Design and implementation of apps

The app designed for this chapter is a single-page application with three buttons. Each button starts a resource-intensive test and logs the execution time of the test while its resource usage is being monitored by the appropriate measuring tool.

All the apps were developed with their default configuration, most conventional programming language and using the recommended IDE. In addition, we followed the best practices and standard implementation method that was available in the official guide and documentation of the tools we used in this chapter. In cases where functionality was required that was not included in the starter project, this work sought to include the most viable third-party library.

The native Android version was implemented using Java and the Android Studio IDE (version 2020.3.1 Patch 2), along with the latest SDK and developer tools. The native iOS version was implemented using the Swift programming language, UIKit (a framework from Apple that provides necessary classes for building iOS apps [98]) and Xcode IDE (version 13.0). Xcode Interface Builder and Storyboard were used for designing the UI of the native iOS app [120].

React Native apps were built using JavaScript and Visual Studio Code (version 1.60.2) and React Native (version 0.66.0). We used the newer Functional React components. Flutter apps were built using Flutter (version 2.5) and Visual Studio Code (version 1.60.2) using the Dart programming language.

The PWA implementation of the app was based on the PWA template of the Create-react-app [101], and was written in JavaScript and React (version 17.0.2) in Visual Studio Code (version 1.60.2). The Workbox library from Google was used to power the app with a service-worker [103]. PWA was hosted on Google

Firebase [104] for the CPU and memory intensive tests, while it ran locally for the network-intensive test, due to special requirements of this test discussed below.

A RESTful API [121] was built specifically for the network-intensive test. This was built using NestJS (version 8.0.0) [122], a Node.js server-side framework, in Visual Studio Code (version 1.60.2).

All of the artifacts developed for this chapter are accessible in public open-source repositories on the Github and can be found via the following link:

https://github.com/Iman-Jamali/thesis-mobile-app-development-cross-platform-vs-native/tree/main/resource-use

5.4.1 CPU-intensive Test

A CPU-intensive test was designed for this chapter to evaluate resource use and performance, with a special focus on the app's CPU usage. This test was designed to find Prime numbers. Prime numbers are natural numbers (positive integers) that are not the product of two smaller natural numbers [123].

The test implementation receives an integer number "N" as an argument, and identifies all the Prime numbers between 1 and "N" (inclusive). We adjust "N" in a way that the test pushes the CPU to its limit for a sustained period while the test retains a reasonable execution time (less than a few minutes) in different development methods. The implementation of tests between development methods is kept as similar as possible by using a simple algorithm that only relies on *if* statements and iteration via *for* loops, and by employing the closest-matched data structures between the programming languages involved to minimize the effect of implementation bias on the results.

5.4.2 Memory-intensive Test

A memory-intensive test was designed to evaluate resource use and performance, with a special focus on the app's memory footprint. The memory-intensive test is based on computing the transitive closure of a graph, which is introduced as a memory-intensive algorithm[124]. Transitive closure is reachability matrix of a given directed graph computed based on whether there is a path between vertex i and vertex j for all vertex pairs of the given graph.

An implementation of Floyd-Warshall [125] algorithm was used to derive the reachability matrix. The test was customized to push the memory consumption to its limit and at the same time, be able to complete. Only one development method — PWA on the iOS low-end device — was not able to complete the test.

The core data structure used in this test was a multidimensional array implemented by the closest standard data type offered by each programming language of the development methods. No third-party library was used in this test. In addition, the implementation adhered to the official guide and documentation of each programming language to minimize implementation bias.

5.4.3 Network-intensive Test

A network-intensive test was designed to evaluate resource use and performance of an app during intensive network operations and the transfer of data over the network. The test is based on communication and transfer of data between the apps and a back-end through a RESTful API [121].

Construction of this test involved taking into account a number of considerations to ensure that the results are fair whilst device resource usage was pushed to the limit. Firstly, the back-end API ran on a local network to minimize network latency. Secondly, network communications were handled in a concurrent fashion on all the apps. In addition, request data was serialized prior to sending and de-serialized when received.

A major component of this test was the HTTP client that was used for each development method. Most of the development methods relied upon third-party libraries. These libraries were recommended by the official guide of development methods and were used in substantial numbers by other projects. On PWA and React Native, "Axios", a promise-based library was used [126]. The Android native implementation employed "Retrofit", a type-safe HTTP client for Android and Java [127]. The native iOS app — built by the Swift programming language — was the only one that relied on an internal solution. The Flutter implementation took advantage of "http", A composable, Future-based library for making HTTP requests [128]. Despite this setup and the planning, the Flutter app on iOS failed to complete this test and was inconsistent on Android and had to be run multiple times to complete. This issue will be discussed in more detail in Section 5.6.

In an exception limited to the network-intensive tests, PWA ran in the regular browser, as opposed to running in a stand-alone browser instance that hides the browser's artifacts. This exception was made to enable connection to a local back-end API running on the same network.

5.5 Results

This section characterizes the results of resource use and performance experiments.

As noted above, in the resource-intensive test measurements, CPU usage and memory footprint were sampled every 500 ms — the shortest common time between the measuring tools. The mean of those resource-specific samples (i.e., CPU usage and memory footprint) were calculated.

Software performance data, in most cases, do not follow a normal distribution [105] as was the case for the results of resource-intensive tests' measurements in this chapter. Therefore, a non-parametric statistical approach that did not rely upon the normality of measurements was employed. Beyond carrying the advantages of a non-parameteric test, the Mann-Whitney U test (also known as Wilcoxon Rank-Sum test) was selected since it offered support for a wide range of sample sizes, and could support ranking development methods, each associated with an ensemble of measurements for a particular metric.

For each specific scenario consisting of a combination of metric, development method, platform (Android and iOS) and device class (low-end and high-end), the measurement of each resource-intensive test was repeated 20 times. To compare across scenarios, the ensembles of scenario-specific outcomes were compared

by non-parametric statistical tests.

For each combination of metric, platform and device class, results of each resource-intensive test were first ranked by their median values. One-way Mann-Whitney U Tests (also known as Wilcoxon Rank-Sum tests) were then run on each successive pair of results to evaluate if there was a statistically significant difference in the ordering. In addition, for each combination of metric, development method and platform — in each resource-intensive test — the effect of device classes was evaluated by running a One-way Mann-Whitney U test. A sample size of 20 and a significance level of 0.05 were applied for all of the statistical tests in this chapter. Mann-Whitney U test can be applied to sample size as small as 5 [106]. The sample size of 20 was selected after considering the time constraints of conducting all tests and the effect of sample size on the statistical test's power. The R statistical computing platform [107] was used to perform statistical tests. The source code of all the statistical analyses in this chapter is available in Appendix B.

5.5.1 CPU-intensive test

Results of the CPU-intensive test consist of measurement of three metrics: CPU usage (in percent), memory footprint (in MB) and execution time (in ms). This section reports the median result of each metric and presents it in charts.

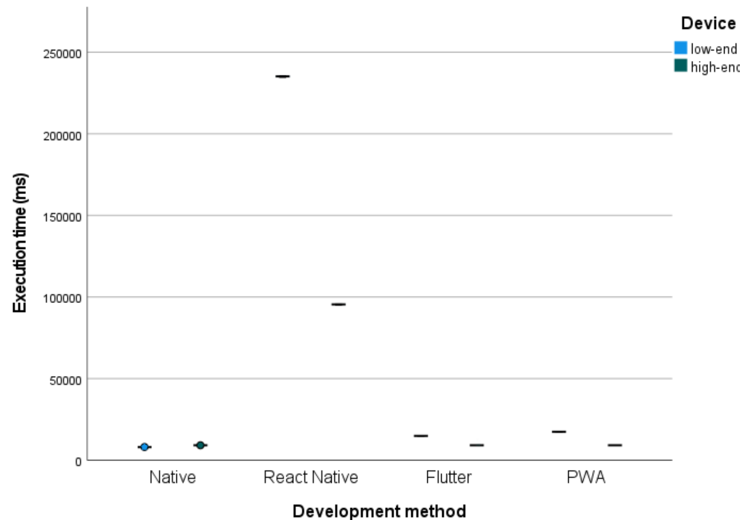


Figure 5.1: Boxplot including median Android CPU-intensive execution time by development method by device. Dots indicate outliers.

Figure 5.1 summarizes the results for median execution time of the CPU-intensive test on the two Android devices. As the chart shows, for the Android low-end device, median execution time for the native app was the fastest, at 8100.5 ms (s.d. of 15.88 ms), followed by Flutter at 14901.5 ms (s.d of 16.3 ms), and the PWA implementation at 17494.5 ms (s.d. of 38.97 ms). The React Native implementation was by far the slowest development method in this test, with a median execution time of 235223 ms (s.d. of 265.01 ms). The Mann-Whitney U test showed a statistically significant difference in median CPU-intensive execution

time on the Android low-end device between native and Flutter (W = 0, $p < 0.001$), between Flutter and PWA (W = 0, $p < 0.001$), and between PWA and React Native implementations of the apps (W = 0, $p < 0.001$).

Results on the Android high-end device were a little different. On this device, PWA was the fastest, with a median execution time of 9206 ms (s.d. of 4.65 ms), closely followed by Flutter at 9219 ms (s.d. of 489.93 ms), and native at 9230 ms (s.d. of 11.96 ms). Reflecting the pattern for the low-end Android device, React Native had the largest median execution time at 95471.5 (s.d. of 137.37 ms). The Mann-Whitney U test demonstrated statistically significant differences in median CPU-intensive execution time on the Android high-end device between PWA and Flutter (W = 32.5, $p < 0.001$), between Flutter and native (W = 108, p = 0.006), and between native and React Native implementations of the apps (W = 0, $p < 0.001$).

Statistically significant differences were seen between the low-end and high-end Android devices for median execution time of the CPU-intensive test for each development method. Native median execution time increased from 8100.5 ms (s.d. of 15.88 ms) on the low-end device to 9230 ms (s.d. of 11.96 ms) on the high-end device (a 13.9% increase). React Native median execution time decreased from 235223 (s.d. of 137.37 ms) on the low-end device to 95471.5 ms (s.d. of 137.37 ms) on the high-end device (a 59.4% decrease). Flutter median execution time decreased from 14901.5 ms (s.d. of 16.3 ms) on the low-end device to 9219 ms (s.d. of 489.93) on the high-end device (a decrease of 38.1%). PWA median execution time decreased from 17494.5 ms (s.d. of 38.97 ms) on the low-end device to 9206 ms (s.d. of 4.65 ms) on the high-end device (a decrease of 47.3%). The Mann-Whitney U test demonstrated a statistically significant difference in the execution time of the CPU-intensive task between the Android device classes for all of the development methods, including native (W = 400, $p < 0.001$), React Native (W = 0, $p < 0.001$), Flutter (W = 0, $p < 0.001$) and PWA (W = 0, $p < 0.001$).

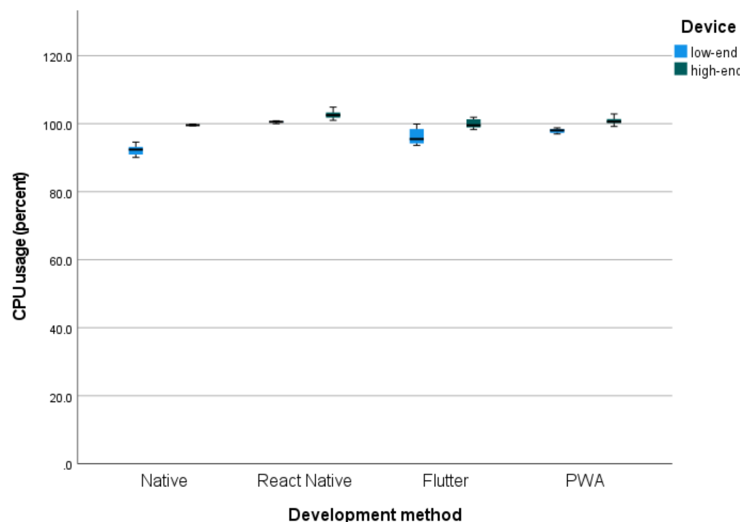


Figure 5.2: Boxplot including median Android CPU-intensive CPU usage by development method by device. Dots indicate outliers.

Figure 5.2 summarizes the results for median CPU usage of the CPU-intensive test on Android devices. As the chart shows, on the low-end Android device, median CPU usage for native was the smallest at 92.4% (s.d. of 1.30%), followed by Flutter at 95.5% (s.d. of 2.26%) and PWA at 98.0% (s.d. of 0.62%). React Native had the largest CPU usage, with a median of 100.5% (s.d. of 0.25%). The Mann-Whitney U test exhibited a statistically significant difference in median CPU-intensive CPU usage on the Android low-end device between native and Flutter ($W = 13$, $p < 0.001$), between Flutter and PWA ($W = 125$, $p = 0.021$), and between PWA and React Native ($W = 0$, $p < 0.001$).

On the high-end Android device, Flutter had the smallest median CPU usage at 99.5% (s.d. of 1.28%), followed by native at 99.6% (s.d. of 0.14%), and PWA at 100.6% (s.d. of 1.01%). React Native had the largest median CPU usage at 102.5% (s.d. of 1.18%). The Mann-Whitney U test demonstrated a statistically significant difference in median CPU-intensive CPU usage on the Android high-end device between native and PWA ($W = 42.5$, $p < 0.001$) and between PWA and React Native ($W = 46$, $p < 0.001$); by contrast, no significant difference was found between Flutter and native ($W = 194$, $p = 0.445$).

Statistically significant differences in median CPU usage of the CPU-intensive test of each development method were seen on the Android platform between the low-end and high-end devices. Native median CPU usage increased from 92.4% (s.d. of 1.30%) on the low-end device to 99.6% (s.d. of 0.14%) on the high-end device (a 7.7% increase). React Native median CPU usage increased from 100.5% (s.d. of 0.25%) on the low-end device to 102.5% (s.d. of 1.18%) on the high-end device (a 1.9% increase). Flutter median CPU usage increased from 95.5% (s.d. of 2.26%) on the low-end device to 99.5% (s.d. of 1.28%) on the high-end device (an increase of 4.1%). PWA median CPU usage increased from 98.0% (s.d. of 0.62%) on the low-end device to 100.6% (s.d. of 1.01%) on the high-end device (an increase of 2.6%). The Mann-Whitney U test exhibited significant difference between the Android device classes for all platforms, including native ($W = 400$, $p < 0.001$), React Native ($W = 400$, $p < 0.001$), Flutter ($W = 358$, $p < 0.001$) and PWA ($W = 400$, $p < 0.001$).

Figure 5.3 summarizes the results for median memory footprint of the CPU-intensive test on Android devices. As the chart shows for the Android low-end device, median memory footprint for PWA was the lowest at 111.6 MB (s.d. of 1.74 MB), followed by native at 115.7 MB (s.d. of 1.38 MB), and Flutter at 133.5 MB (s.d. of 3.10 MB). React Native consumed the most memory, with a median of 145.9 MB (s.d. of 3.95 MB). The Mann-Whitney U test demonstrated a statistically significant difference in median memory footprint in the CPU-intensive test on the Android low-end device between PWA and native ($W = 29$, $p < 0.001$), between native and Flutter ($W = 29$, $p < 0.001$), and between Flutter and React Native implementations ($W = 0$, $p < 0.001$).

On the Android high-end device, native had the smallest median memory footprint at 121.5 MB (s.d. of 1.08 MB), followed by PWA at 140.7 MB (s.d. of 1.25 MB), and Flutter at 141.8 MB (s.d. of 3.73 MB). As with the Android low-end device, React Native used the most amount of memory at 145.9 MB (s.d. of 1.54 MB). The Mann-Whitney U test demonstrated a statistically significant difference in median CPU-intensive

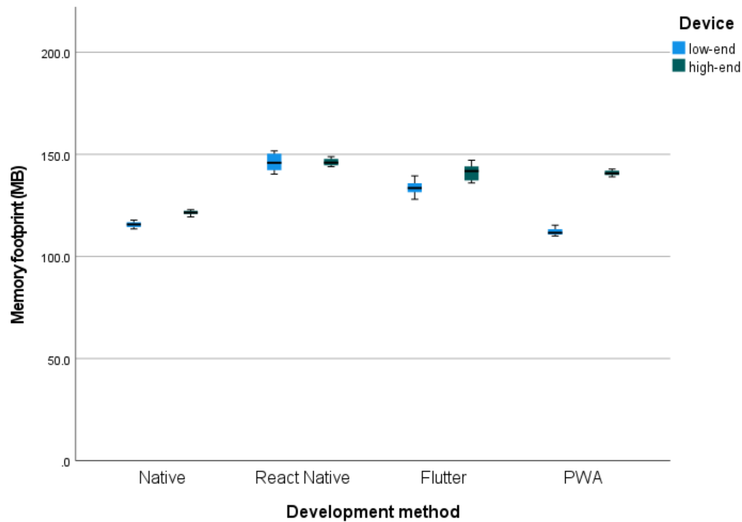


Figure 5.3: Boxplot including median Android CPU-intensive memory footprint by development method by device. Dots indicate outliers.

memory footprint on the Android high-end device between native and PWA (W = 0, $p < 0.001$) and between Flutter and React Native (W = 42.5, $p < 0.001$) implementations; by contrast, no significant difference was found in this metric between PWA and Flutter implementations (W = 190, p = 0.403).

Statistically significant differences were seen between the Android low-end and high-end devices for median memory footprint of the CPU-intensive test of each development method. Native median memory footprint increased from 115.7 MB (s.d. of 1.38 MB) on the low-end device to 121.5 MB (s.d. of 1.08 MB) on the high-end device (a 5% increase). React Native median memory footprint did not change, remaining at 145.9 MB on both the low-end and high-end devices. Flutter median memory footprint increased from 133.5 MB (s.d. of 3.10 MB) on the low-end device to 141.8 MB (s.d. of 3.73 MB) on the high-end device (an increase of 6.2%). PWA median memory footprint increased from 111.6 MB (s.d. of 1.74 MB) on the low-end device to 140.7 MB (s.d. of 1.25 MB) on the high-end device (an increase of 26%). The Mann-Whitney U test demonstrated significant difference between the Android device classes for native (W = 400, $p < 0.001$), Flutter (W = 374, $p < 0.001$) and PWA (W = 400, $p < 0.001$) implementations, with no significant difference being found for React Native (W = 191, p = 0.601).

Figure 5.4 summarizes the results for median execution time of the CPU-intensive test on iOS devices. As the chart shows, for the iOS low-end device, median execution time for Flutter was the lowest at 7874 ms (s.d. of 13.05 ms), followed closed by PWA at 8136 ms (s.d. of 5.88 ms) and native at 8140.5 ms (s.d. of 9.09 ms). React Native was by far the slowest to finish the test, with a median execution time of 144525.5 ms (s.d. of 18133.76 ms). The Mann-Whitney U test demonstrated a statistically significant difference in median execution time in the CPU-intensive test on the iOS low-end device between Flutter and PWA (W = 0, $p < 0.001$) and between native and React Native (W = 0, $p < 0.001$) implementations. By contrast, no significant difference was found between PWA and native implementations (W = 151.5, p = 0.096).

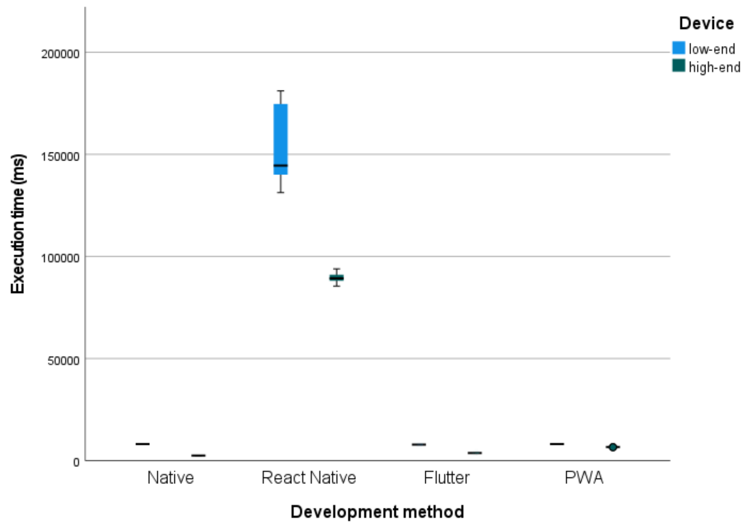


Figure 5.4: Boxplot including median iOS CPU-intensive execution time by development method by device. Dots indicate outliers.

On the iOS high-end device, native had the smallest median execution time at 2537 ms (s.d. of 1.7 ms), followed by Flutter at 3768.5 ms (s.d. of 38.22 ms), and PWA at 6718.5 ms (s.d. of 45.51 ms). React Native was the slowest development method, with a median execution time of 89284 ms (s.d. of 2141.37 ms). The Mann-Whitney U test demonstrated a statistically significant difference in median CPU-intensive execution time on the iOS high-end device between native and Flutter ($W = 0$, $p < 0.001$), between Flutter and PWA ($W = 0$, $p < 0.001$), and between PWA and React Native ($W = 0$, $p < 0.001$) implementations.

Statistically significant differences were seen between the iOS low-end and high-end devices for median execution time of the CPU-intensive test of each development method. Native median execution time decreased from 8140.5 ms (s.d. of 9.09 ms) on the iOS low-end device to 2537 ms (s.d. of 1.7 ms) on the high-end device (a 68.8% decrease). React Native median execution time decreased from 144525.5 ms (s.d. of 18133.76 ms) on the low-end device to 89284 ms (s.d. of 2141.37 ms) on the high-end device (a decrease of 38.2%). Flutter median execution time decreased from 7874 ms (s.d. of 13.05 ms) on the low-end device to 3768.5 ms (s.d. of 38.22 ms) on the high-end device (a decrease of 52.1%). PWA median execution time decreased from 8136 ms (s.d. of 5.88 ms) on the low-end device to 6718.5 ms (s.d. of 45.51 ms) on the high-end device (a decrease of 17.4%). The Mann-Whitney U test demonstrated a significant difference between the device classes for native ($W = 0$, $p < 0.001$), React Native ($W = 0$, $p < 0.001$), Flutter ($W = 0$, $p < 0.001$), and PWA ($W = 0$, $p < 0.001$).

Figure 5.5 summarizes the results for median CPU usage of the CPU-intensive test on iOS devices. As the chart shows, for the iOS low-end device, median CPU usage for PWA was the lowest at 82.7% (s.d of 1.13%), followed by native at 99.6% (s.d of 0.21%) and Flutter at 99.6% (s.d of 0.08%) and React Native at 99.6% (s.d of 0.15%). The Mann-Whitney U test demonstrated a statistically significant difference in median CPU usage in the CPU-intensive test on the iOS low-end device between PWA and native ($W = 0$, $p <$

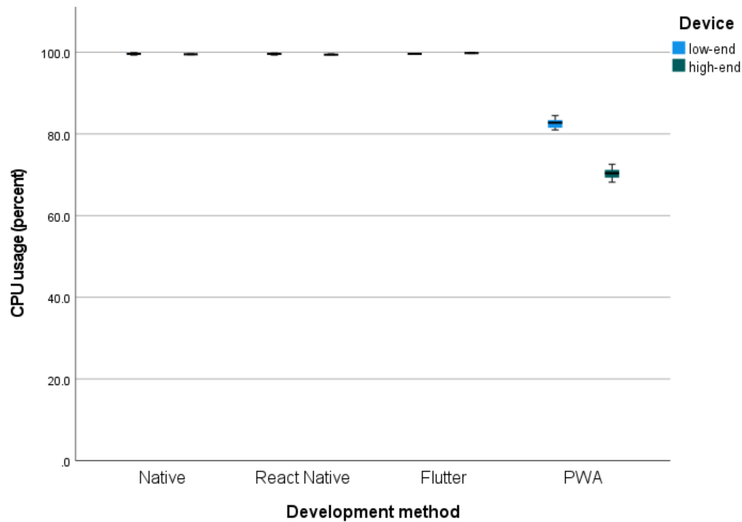


Figure 5.5: Boxplot including median iOS CPU-intensive CPU usage by development method by device. Dots indicate outliers.

0.001). No significant difference was found between native and Flutter (W = 224, p = 0.750), and between Flutter and React Native (W = 214, p = 0.658) on that device.

On the iOS high-end device, PWA had the smallest median CPU usage at 70.4% (s.d. of 1.33%), followed by React Native at 99.4% (s.d. of 0.08%), native at 99.5% (s.d. of 0.07%), and Flutter at 99.8% (s.d. of 0.06%). The Mann-Whitney U test demonstrated a statistically significant difference in median CPU-intensive CPU usage on the iOS high-end device between PWA and React Native (W = 0, p < 0.001), between native and Flutter (W = 0, p < 0.001). By contrast, no significant difference was found between React Native and native (W = 129, p = 0.022).

Statistically significant differences were seen between the iOS low-end and high-end devices for median CPU usage of the CPU-intensive test for two of the development methods. Flutter median CPU usage increased from 99.6% on the low-end device to 99.8% on the high-end device. PWA median CPU usage decreased from 82.7% (s.d of 1.13%) on the low-end device to 70.4% (s.d. of 1.33%) on the high-end device (a decrease of 14.8%). Native and React Native median CPU usage decreased from 99.6% (s.d of 0.21%) and 99.6% (s.d of 0.15%) on the low-end device to 99.5% (s.d. of 0.07%) and 99.4% (s.d. of 0.08%), respectively, on the high-end device. The Mann-Whitney U test demonstrated a significant difference between the device classes for Flutter (W = 382, p < 0.001), PWA (W = 0, p < 0.001), and React Native (W = 130, p = 0.025), with no significant difference being found for native (W = 145, p = 0.065).

Figure 5.6 summarizes the results for median memory footprint of the CPU-intensive test on iOS devices. As the chart shows, for the iOS low-end device, median memory footprint for native was the lowest at 8.6 MB (s.d. of 0.19 MB), followed by React Native at 19.3 MB (s.d. of 0.85 MB). PWA and Flutter consumed more memory, with a median memory footprint of 40.9 MB (s.d. of 2.19 MB) and 45.6 MB (s.d. of 0.86 MB), respectively. The Mann-Whitney U test demonstrated a significant difference in median memory footprint

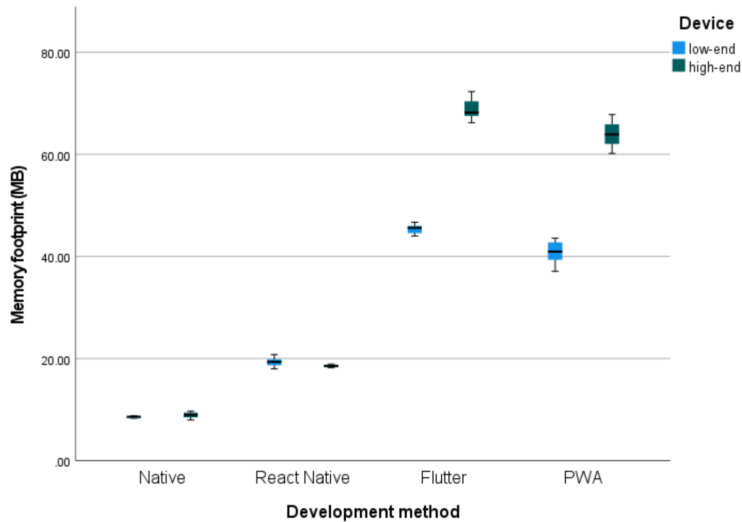


Figure 5.6: Boxplot including median iOS CPU-intensive memory footprint by development method by device. Dots indicate outliers.

in the CPU-intensive test on the iOS low-end device between native and React Native (W = 0, $p < 0.001$), between React Native and PWA (W = 0, $p < 0.001$), and between PWA and Flutter (W = 0, $p < 0.001$).

On the iOS high-end device, native had the smallest median memory footprint at 9.0 MB (s.d. of 0.53 MB), followed by React Native at 18.5 MB (s.d. of 0.25 MB), PWA at 63.9 MB (s.d. of 2.24 MB), and Flutter at 68.2 MB (s.d. of 1.89 MB). The Mann-Whitney U test demonstrated a statistically significant difference in median CPU-intensive memory footprint on the iOS high-end device between native and React Native (W = 0, $p < 0.001$), between React Native and PWA (W = 0, $p < 0.001$), and between PWA and Flutter (W = 14, $p < 0.001$).

Statistically significant differences were seen in the median memory footprint of the CPU-intensive test of each development method between the iOS low-end and high-end devices. Native median memory footprint increased from 8.6 MB (s.d. of 0.19 MB) on the low-end device to 9.0 MB (s.d. of 0.53 MB) on the high-end device (an increase of 4.6%). React Native median memory footprint decreased from 19.3 MB (s.d. of 0.85 MB) on the low-end device to 18.5 MB (s.d. of 0.25 MB) on the high-end device (a decrease of 4.1%). Flutter median memory footprint increased from 45.6 MB (s.d. of 0.86 MB) on the low-end device to 68.2 MB (s.d. of 1.89 MB) on the high-end device (an increase of 49.5%). PWA median memory footprint increased from 40.9 MB (s.d. of 2.19 MB) on the low-end device to 63.9 MB (s.d. of 2.24 MB) on the high-end device (an increase of 56.2%). The Mann-Whitney U test demonstrated a significant difference between the device classes for native (W = 282, $p = 0.013$), React Native (W = 82, $p < 0.001$), Flutter (W = 400, $p < 0.001$), and PWA (W = 400, $p < 0.001$) implementations.

5.5.2 Memory-intensive test

Results of the memory-intensive test consist of measurement of three metrics, CPU usage (in percent), memory footprint (in MB) and execution time (in ms). The median result of each metric are presented in this section and reported in charts.

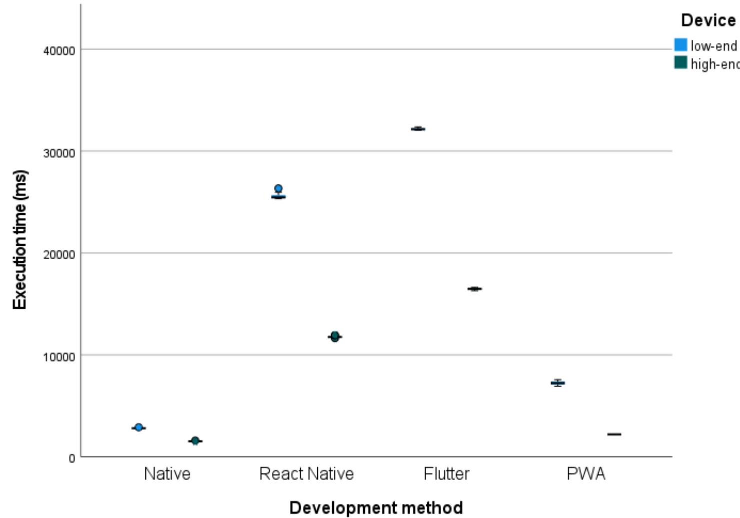


Figure 5.7: Boxplot including median Android memory-intensive execution time by development method by device. Dots indicate outliers.

Figure 5.7 summarizes the results for median execution time of the memory-intensive test on Android devices. As the chart shows for the Android low-end device, median execution time for native was the lowest at 2804.5 ms (s.d. of 36.77 ms), followed by PWA at 7233 ms (s.d. of 181.32 ms) and React Native at 25465 ms (s.d. of 260.16 ms). Flutter was the slowest on the low-end Android having median execution time of 32155 ms (s.d. of 90.85 ms). The Mann-Whitney U test demonstrated a statistically significant difference in median execution time in the memory-intensive test on the Android low-end device between native and PWA ($W = 0$, $p < 0.001$), and between PWA and React Native ($W = 0$, $p < 0.001$), and between React Native and Flutter ($W = 0$, $p < 0.001$).

On the Android high-end device native was the fastest, with a median execution time of 1530 ms (s.d. of 48.39 ms), followed by PWA at 2203 ms (s.d. of 134.38 ms) and React Native at 11763 ms (s.d. of 127.31 ms). Flutter was the slowest on the iOS high-end at execution time of 16487 ms (s.d. of 96.56 ms). The Mann-Whitney U test demonstrated a statistically significant difference in the median memory-intensive execution time on the Android high-end device between native and PWA ($W = 0$, $p < 0.001$), and between PWA and React Native ($W = 0$, $p < 0.001$), and between React Native and Flutter ($W = 0$, $p < 0.001$).

Statistically significant differences were seen between the Android low-end and high-end devices for median execution time of the memory-intensive test of each development method. Native median execution time decreased from 2804.5 ms (s.d. of 36.77 ms) on the low-end device to 1530 ms (s.d. of 48.39 ms) on the

high-end device (a decrease of 45.4%). React Native median execution time decreased from 25465 ms (s.d. of 260.16 ms) on the low-end device to 11763 ms (s.d. of 127.31 ms) on the high-end device (a decrease of 53.8%). Flutter median execution time decreased from 32155 ms (s.d. of 90.85 ms) on the low-end device to 16487 ms (s.d. of 96.56 ms) on the high-end device (a decrease of 48.7%). PWA median execution time decreased from 7233 ms (s.d. of 181.32 ms) on the low-end device to 2203 ms (s.d. of 134.38 ms) on the high-end device (a decrease of 69.5%). The Mann-Whitney U test demonstrated a statistically significant difference between the device classes for native ($W = 0$, $p < 0.001$), React Native ($W = 0$, $p < 0.001$), Flutter ($W = 0$, $p < 0.001$) and PWA ($W = 0$, $p < 0.001$).

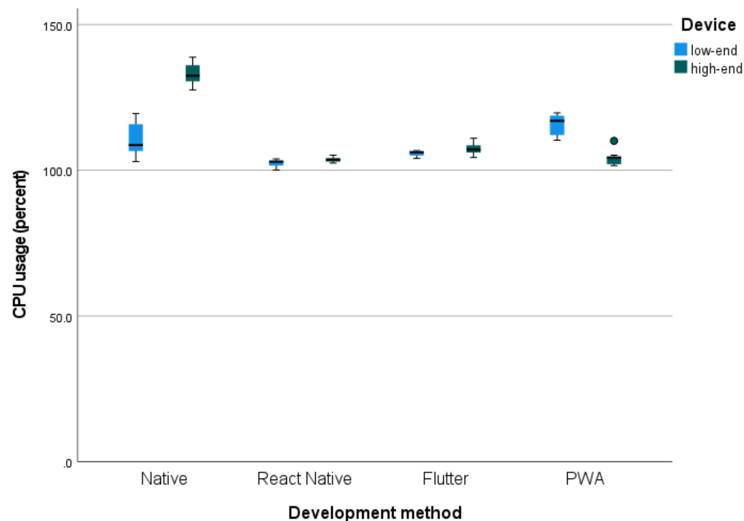


Figure 5.8: Boxplot including median Android memory-intensive CPU usage by development method by device. Dots indicate outliers.

Figure 5.8 summarizes the results for median CPU usage of the memory-intensive test on Android devices. As the chart shows for the Android low-end device, median CPU usage for React Native was lowest at 102.9% (s.d. of 1.10%), followed by Flutter 106.1% (s.d. of 0.93%) and native 108.6% (s.d. of 5.37%). PWA had the highest median CPU usage on the low-end android at 116.9% (s.d. of 3.51%). The Mann-Whitney U test demonstrated a statistically significant difference in median CPU usage in the memory-intensive test on the Android low-end device between React Native and Flutter ($W = 0$, $p < 0.001$), between Flutter and native ($W = 78$, $p < 0.001$), and between native and PWA ($W = 93$, $p = 0.001$).

On the Android high-end device React Native had the lowest median CPU usage of 103.6% (s.d. of 0.81%), followed by PWA 104.3% (s.d. of 1.93%) and Flutter 107.1% (s.d. of 1.85%). Native had the highest median CPU usage on the high-end Android at 132.4% (s.d. of 3.43%). The Mann-Whitney U test demonstrated a statistically significant difference in the median memory-intensive CPU usage on the Android high-end device between PWA and Flutter ($W = 30$, $p < 0.001$), and between Flutter and native ($W = 0$, $p < 0.001$), with no significant difference was found between React Native and PWA ($W = 173$, $p = 0.240$).

Statistically significant differences were seen between the Android low-end and high-end devices for median

CPU usage of the memory-intensive test of each development method. Native median CPU usage increased from 108.6% (s.d. of 5.37%) on the low-end device to 132.4% (s.d. of 3.43%) on the high-end device (an increase of 21.9%). React Native median CPU usage increased from 102.9% (s.d. of 1.10%) on the low-end device to 103.6% (s.d. of 0.81%) on the high-end device (an increase of 0.6%). Flutter median CPU usage increased from 106.1% (s.d. of 0.93%) on the low-end device to 107.1% (s.d. of 1.85%) on the high-end device (an increase of 0.9%). PWA median CPU usage decreased from 116.9% (s.d. of 3.51%) on the low-end device to 104.3% (s.d. of 1.93%) on the high-end device (a decrease of 10.7%). The Mann-Whitney U test demonstrated a statistically significant difference between the device classes for native ($W = 0$, $p < 0.001$), React Native ($W = 313$, $p = 0.001$), Flutter ($W = 291$, $p = 0.007$) and PWA ($W = 0$, $p < 0.001$).

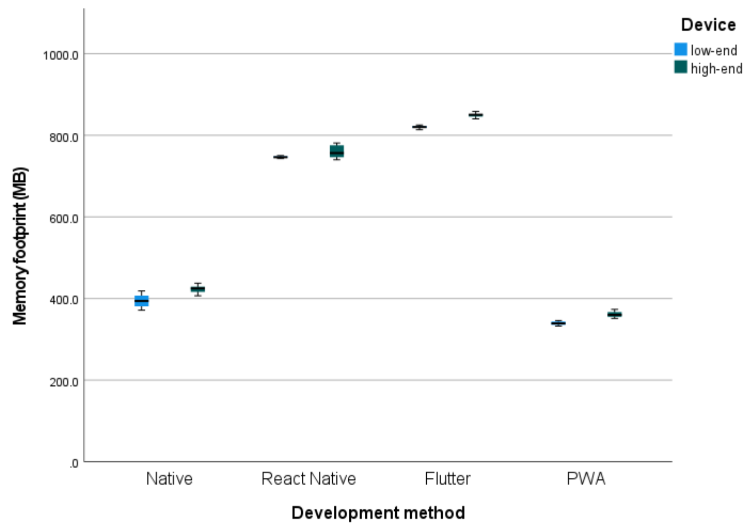


Figure 5.9: Boxplot including median Android memory-intensive memory footprint by development method by device. Dots indicate outliers.

Figure 5.9 summarizes the results for median memory footprint of the memory-intensive test on Android devices. As the chart shows for the Android low-end device, median memory footprint for PWA was lowest at 339.2 MB (s.d. of 4.62 MB), followed by native 394.1 MB (s.d. of 15.98 MB). React Native and Flutter had a relatively higher median memory footprint at 746.5 MB (s.d. of 2.33 MB) and 820.4 MB (s.d. of 2.94 MB), respectively. The Mann-Whitney U test demonstrated a statistically significant difference in median memory footprint in the memory-intensive test on the Android low-end device between PWA and native ($W = 0$, $p < 0.001$), between native and React Native ($W = 0$, $p < 0.001$), and between React Native and Flutter ($W = 0$, $p < 0.001$).

On the Android high-end device PWA had the lowest median memory footprint of 360.2 MB (s.d. of 7.21 MB), followed by native 424.9 MB (s.d. of 8.61 MB). React Native and Flutter had a relatively higher median memory footprint at 756.7 MB (s.d. of 14.43 MB) and 850.3 MB (s.d. of 4.70 MB), respectively. The Mann-Whitney U test demonstrated a statistically significant difference in median memory footprint in the memory-intensive test on the Android high-end device between PWA and native ($W = 0$, $p < 0.001$),

between native and React Native (W = 0, $p < 0.001$), and between React Native and Flutter (W = 0, $p < 0.001$).

Statistically significant differences were seen between the Android low-end and high-end devices for median memory footprint of the memory-intensive test of each development method. Native median memory footprint increased from 394.1 MB (s.d. of 15.98 MB) on the low-end device to 424.9 MB (s.d. of 8.61 MB) on the high-end device (an increase of 7.8%). React Native median memory footprint increased from 746.5 MB (s.d. of 2.33 MB) on the low-end device to 756.7 MB (s.d. of 14.43 MB) on the high-end device (an increase of 1.3%). Flutter median memory footprint increased from 820.4 MB (s.d. of 2.94 MB) on the low-end device to 850.3 MB (s.d. of 4.70 MB) on the high-end device (an increase of 3.6%). PWA median memory footprint increased from 339.2 MB (s.d. of 4.62 MB) on the low-end device to 360.2 MB (s.d. of 7.21 MB) on the high-end device (an increase of 6.2%). The Mann-Whitney U test demonstrated a statistically significant difference between the device classes for native (W = 378, $p < 0.001$), React Native (W = 300, p = 0.003), Flutter (W = 400, $p < 0.001$) and PWA (W = 400, $p < 0.001$).

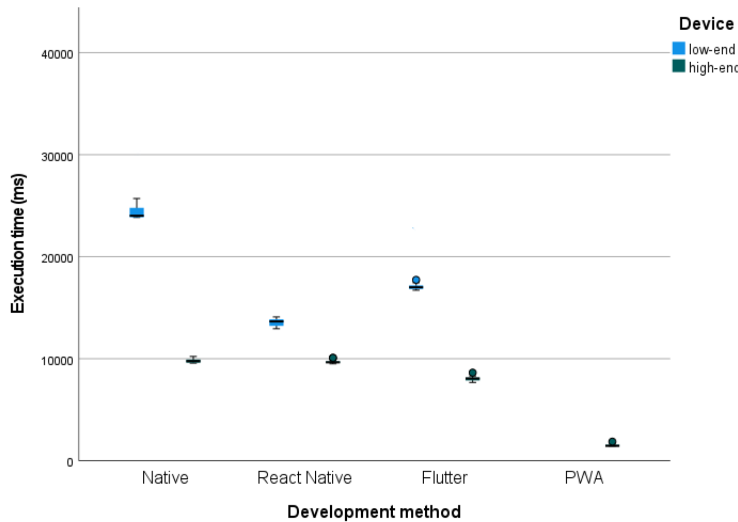


Figure 5.10: Boxplot including median iOS memory-intensive execution time by development method by device. Dots indicate outliers.

Figure 5.10 summarizes the results for median execution time of the memory-intensive test on iOS devices. As the chart shows, for the iOS low-end device, median execution time for React Native was the fastest at 13659 ms (s.d. of 379.44 ms), followed by Flutter 17012 ms (s.d. of 1298.37) and native 24026.5 ms (s.d. of 1462.96 ms). The Mann-Whitney U test demonstrated a statistically significant difference in median execution time in the memory-intensive test on the iOS low-end device between React Native and Flutter (W = 0, $p < 0.001$), and between Flutter and native (W = 0, $p < 0.001$).

On the iOS high-end device, PWA was by far the fastest at median execution time of 1460 ms (s.d. of 107.44 ms), followed by Flutter 8072.5 ms (s.d. of 222.42 ms), React Native 9651 ms (s.d. of 158.52 ms) and native 9775 ms (s.d. of 200.41 ms). The Mann-Whitney U test demonstrated a statistically significant

difference in median execution time in the memory-intensive test on the iOS high-end device between PWA and Flutter (W = 0, p < 0.001), between Flutter and React Native (W = 0, p < 0.001), and between React Native and native (W = 126, p = 0.023).

Statistically significant differences were seen between the iOS low-end and high-end devices for median execution time of the memory-intensive test of each development method. Native median execution time decreased from 24026.5 ms (s.d. of 1462.96 ms) on the low-end device to 9775 ms (s.d. of 200.41 ms) on the high-end device (a decrease of 59.3%). React Native median execution time decreased from 13659 ms (s.d. of 379.44 ms) on the low-end device to 9651 ms (s.d. of 158.52 ms) on the high-end device (a decrease of 29.3%). Flutter median execution time decreased from 17012 ms (s.d. of 1298.37) on the low-end device to 8072.5 ms (s.d. of 222.42 ms) on the high-end device (a decrease of 52.5%). The Mann-Whitney U test demonstrated a statistically significant difference between the device classes for native (W = 0, p < 0.001), React Native (W = 0, p < 0.001), Flutter (W = 0, p < 0.001).

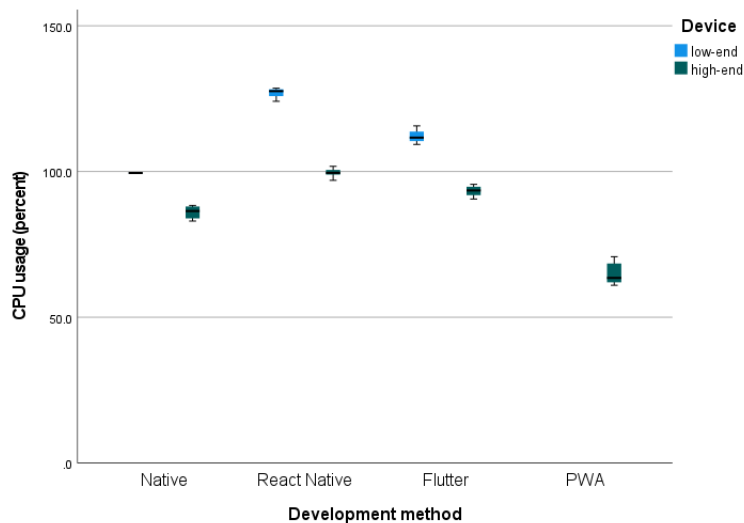


Figure 5.11: Boxplot including median iOS memory-intensive CPU usage by development method by device. Dots indicate outliers.

Figure 5.11 summarizes the results for median CPU usage of the memory-intensive test on iOS devices. As the chart shows, for the iOS low-end device, median CPU usage for native was the lowest at 99.5% (s.d. of 0.07%), followed by Flutter 111.6% (s.d. of 2.06%) and React Native 127.6% (s.d. of 1.43%). The Mann-Whitney U test demonstrated a statistically significant difference in median CPU usage in the memory-intensive test on the iOS low-end device between native and Flutter (W = 0, p < 0.001), and between Flutter and React Native (W = 0, p < 0.001).

On the iOS high-end device, PWA had the lowest median CPU usage at 63.5% (s.d. of 3.48%) and was followed by native 86.4% (s.d. of 1.95%). Flutter and React Native median CPU usage was 93.5% (s.d. of 1.69%) and 99.5% (s.d. of 1.32%), respectively. The Mann-Whitney U test demonstrated a statistically significant difference in median CPU usage in the memory-intensive test on the iOS high-end device between

PWA and native (W = 0, p < 0.001), and between native and Flutter (W = 0, p < 0.001), and between Flutter and React Native (W = 0, p < 0.001).

Statistically significant differences were seen between the iOS low-end and high-end devices for median CPU usage of the memory-intensive test of each development method. Native median CPU usage decreased from 99.5% (s.d. of 0.07%) on the low-end device to 86.4% (s.d. of 1.95%) on the high-end device (a decrease of 13.1%). React Native median CPU usage decreased from 127.6% (s.d. of 1.43%) on the low-end device to 99.5% (s.d. of 1.32%) on the high-end device (a decrease of 22%). Flutter median CPU usage decreased from 111.6% (s.d. of 2.06%) on the low-end device to 93.5% (s.d. of 1.69%) on the high-end device (a decrease of 16.2%). The Mann-Whitney U test demonstrated a statistically significant difference between the device classes for native (W = 0, p < 0.001), React Native (W = 0, p < 0.001), Flutter (W = 0, p < 0.001).

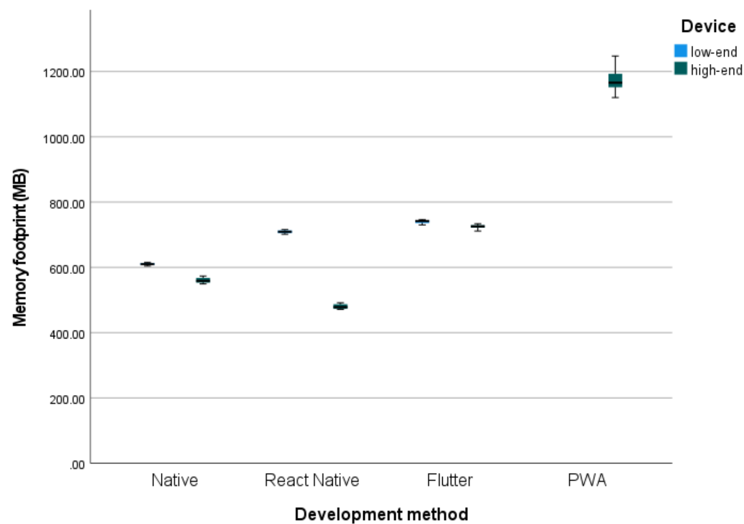


Figure 5.12: Boxplot including median iOS memory-intensive memory footprint by development method by device. Dots indicate outliers.

Figure 5.12 summarizes the results for median memory footprint of the memory-intensive test on iOS devices. As the chart shows, for the iOS low-end device, median memory footprint for native was the lowest at 610.0 MB (s.d. of 3.89 MB). React Native and Flutter median memory footprint on the low-end iOS were 709.2 MB (s.d. of 4.58 MB) and 742.1 MB (s.d. of 5.13 MB), respectively. The Mann-Whitney U test demonstrated a statistically significant difference in median memory footprint in the memory-intensive test on the iOS low-end device between native and React Native (W = 0, p < 0.001), and between React Native and Flutter (W = 0, p < 0.001).

On the iOS high-end device, native had the lowest median memory footprint at 559 MB (s.d. of 8.30 MB), followed by React Native 478.2 MB (s.d. of 7.30 MB) and Flutter 725.5 MB (s.d. of 6.47 MB). PWA had the highest median memory footprint at 1165.7 MB (s.d. of 32.44 MB). The Mann-Whitney U test demonstrated a statistically significant difference in median memory footprint in the memory-intensive test on the iOS high-end device between native and React Native (W = 400, p < 0.001), between React Native

and Flutter (W = 0, $p < 0.001$), and between Flutter and PWA (W = 0, $p < 0.001$).

Statistically significant differences were seen between the iOS low-end and high-end devices for median memory footprint of the memory-intensive test of each development method. Native median memory footprint decreased from 610.0 MB (s.d. of 3.89 MB) on the low-end device to 559 MB (s.d. of 8.30 MB) on the high-end device (a decrease of 8.3%). React Native median memory footprint decreased from 709.2 MB (s.d. of 4.58 MB) on the low-end device to 478.2 MB (s.d. of 7.30 MB) on the high-end device (a decrease of 32.5%). Flutter median memory footprint decreased from 742.1 MB (s.d. of 5.13 MB) on the low-end device to 725.5 MB (s.d. of 6.47 MB) on the high-end device (a decrease of 2.2%). The Mann-Whitney U test demonstrated a statistically significant difference between the device classes for native (W = 0, $p < 0.001$), React Native (W = 0, $p < 0.001$), Flutter (W = 11, $p < 0.001$).

5.5.3 Network-intensive test

Results of the network-intensive test consist of measurement of four metrics, CPU usage (in percent), memory footprint (in MB), total data transferred (in MB) and execution time (in ms). The median result of each metric are presented in this section and reported in charts.

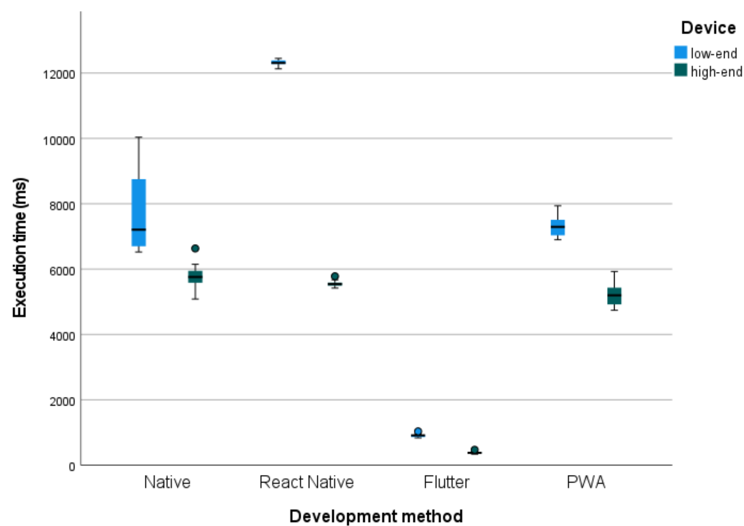


Figure 5.13: Boxplot including median Android network-intensive execution time by development method by device. Dots indicate outliers.

Figure 5.13 summarizes the results for median execution time of the network-intensive on Android devices. As the chart shows for the Android low-end device, median execution time for Flutter was the lowest at 914 ms (s.d. of 46.33 ms). Next, Native and PWA median execution times for the low-end Android were 7207.5 ms (s.d. of 1142.82 ms) and 7296 ms (s.d. of 307.70 ms), respectively. React Native was the slowest at median execution time of 12312.5 ms (s.d. of 85.82 ms) on the low-end Android. The Mann-Whitney U test demonstrated a statistically significant difference in median execution time in the network-intensive on the Android low-end device between Flutter and native (W = 0, $p < 0.001$), and between PWA and React

Native ($W = 0$, $p < 0.001$), with no significant difference was found between native and PWA ($W = 207$, $p = 0.585$).

On the Android high-end device, Flutter had the lowest median execution time at 380 ms (s.d. of 31.53 ms), followed by PWA 5200 ms (s.d. of 328.09 ms), React Native 5544.5 ms (s.d. of 85.04 ms) and native 5761 ms (s.d. of 345.45 ms). The Mann-Whitney U test demonstrated a statistically significant difference in median execution time in the network-intensive on the Android high-end device between Flutter and PWA ($W = 0$, $p < 0.001$), between PWA and React Native ($W = 76$, $p < 0.001$), and between React Native and native ($W = 82$, $p < 0.001$).

Statistically significant differences were seen between the Android low-end and high-end devices for median execution time of the network-intensive of each development method. Native median execution time decreased from 7207.5 ms (s.d. of 1142.82 ms) on the low-end device to 5761 ms (s.d. of 345.45 ms) on the high-end device (a decrease of 20%). React Native median execution time decreased from 12312.5 ms (s.d. of 85.82 ms) on the low-end device to 5544.5 ms (s.d. of 85.04 ms) on the high-end device (a decrease of 54.9%). Flutter median execution time decreased from 914 ms (s.d. of 46.33 ms) on the low-end device to 380 ms (s.d. of 31.53 ms) on the high-end device (a decrease of 58.4%). PWA median execution time decreased from 7296 ms (s.d. of 307.70 ms) on the low-end device to 5200 ms (s.d. of 328.09 ms) on the high-end device (a decrease of 28.7%). The Mann-Whitney U test demonstrated a statistically significant difference between the device classes for native ($W = 2$, $p < 0.001$), React Native ($W = 0$, $p < 0.001$), Flutter ($W = 0$, $p < 0.001$) and PWA ($W = 0$, $p < 0.001$).

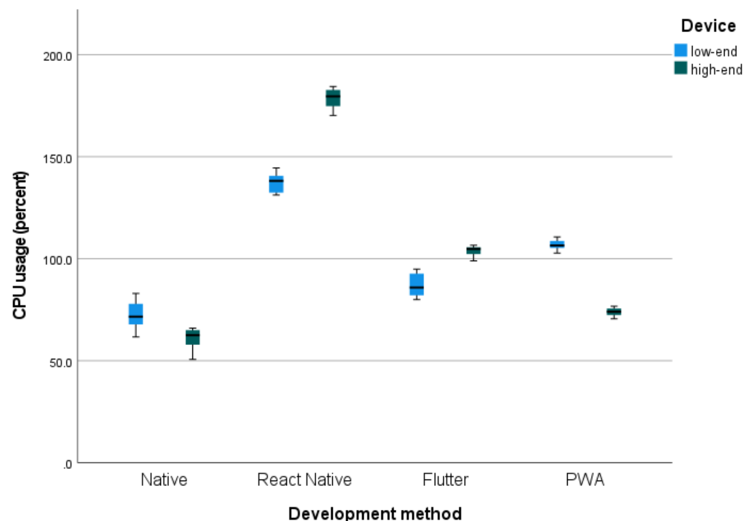


Figure 5.14: Boxplot including median Android network-intensive CPU usage by development method by device. Dots indicate outliers.

Figure 5.14 summarizes the results for median CPU usage of the network-intensive on Android devices. As the chart shows for the Android low-end device, median CPU usage for native was the lowest at 71.6% (s.d. of 6.23%), followed by Flutter 85.9% (s.d. of 5.36%), PWA 106.5% (s.d. of 2.43%) and React Native

138.2% (s.d. of 4.54%). The Mann-Whitney U test demonstrated a statistically significant difference in median CPU usage in the network-intensive on the Android low-end device between native and Flutter (W = 11, $p < 0.001$), and between Flutter and PWA (W = 0, $p < 0.001$), and between PWA and React Native (W = 0, $p < 0.001$).

On the Android high-end device, native had the lowest median CPU usage at 62.5% (s.d. of 4.40%), followed by PWA 74% (s.d. of 1.95%), Flutter 104.8% (s.d. of 2.32%) and React Native 179.6% (s.d. of 4.47%). The Mann-Whitney U test demonstrated a statistically significant difference in median CPU usage in the network-intensive on the Android high-end device between native and PWA (W = 0, $p < 0.001$) and between PWA and Flutter (W = 0, $p < 0.001$) and between Flutter and React Native (W = 0, $p < 0.001$).

Statistically significant differences were seen between the Android low-end and high-end devices for median CPU usage of the network-intensive of each development method. Native median CPU usage decreased from 71.6% (s.d. of 6.23%) on the low-end device to 62.5% (s.d. of 4.40%) on the high-end device (a decrease of 14.5%). React Native median CPU usage increased from 138.2% (s.d. of 4.54%) on the low-end device to 179.6% (s.d. of 4.47%) on the high-end device (an increase of 29.9%). Flutter median CPU usage increased from 85.9% (s.d. of 5.36%) on the low-end device to 104.8% (s.d. of 2.32%) on the high-end device (an increase of 22%). PWA median CPU usage decreased from 106.5% (s.d. of 2.43%) on the low-end device to 74% (s.d. of 1.95%) on the high-end device (a decrease of 30%). The Mann-Whitney U test demonstrated a statistically significant difference between the device classes for native (W = 18, $p < 0.001$), React Native (W = 400, $p < 0.001$), Flutter (W = 400, $p < 0.001$) and PWA (W = 0, $p < 0.001$).

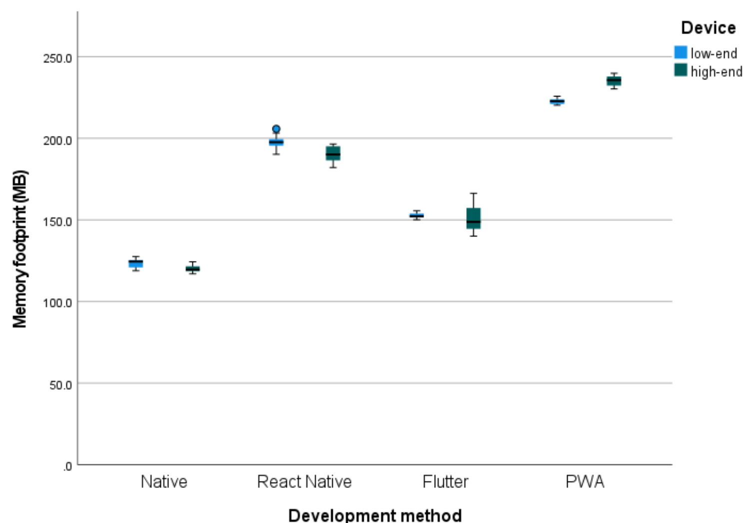


Figure 5.15: Boxplot including median Android network-intensive memory footprint by development method by device. Dots indicate outliers.

Figure 5.15 summarizes the results for median memory footprint of the network-intensive on Android devices. As the chart shows for the Android low-end device, median memory footprint for native was the lowest at 124.5 MB (s.d. of 2.70 MB), followed by Flutter 152.3 MB (s.d. of 1.47 MB), React Native 197.6

MB (s.d. of 3.95 MB) and PWA 222.7 MB (s.d. of 1.64 MB). The Mann-Whitney U test demonstrated a statistically significant difference in median memory footprint in the network-intensive on the Android low-end device between native and Flutter ($W = 0$, $p < 0.001$), between Flutter and React Native ($W = 0$, $p < 0.001$), and between React Native and PWA ($W = 0$, $p < 0.001$).

On the Android high-end device, native had the lowest median memory footprint at 119.6 MB (s.d. of 2.05 MB), followed by Flutter 148.8 MB (s.d. of 7.85 MB), React Native 190.1 MB (s.d. of 4.71 MB) and PWA 235.7 MB (s.d. of 3.05 MB). The Mann-Whitney U test demonstrated a statistically significant difference in median memory footprint in the network-intensive on the Android high-end device between native and Flutter ($W = 0$, $p < 0.001$), between Flutter and React Native ($W = 400$, $p < 0.001$), and between React Native and PWA ($W = 0$, $p < 0.001$).

Statistically significant differences were seen between the Android low-end and high-end devices for median memory footprint of the network-intensive of each development method. Native median memory footprint decreased from 124.5 MB (s.d. of 2.70 MB) on the low-end device to 119.6 MB (s.d. of 2.05 MB) on the high-end device (a decrease of 3.9%). React Native median memory footprint decreased from 197.6 MB (s.d. of 3.95 MB) on the low-end device to 190.1 MB (s.d. of 4.71 MB) on the high-end device (a decrease of 3.8%). Flutter median memory footprint decreased from 152.3 MB (s.d. of 1.47 MB) on the low-end device to 148.8 MB (s.d. of 7.85 MB) on the high-end device (a decrease of 2.3%). PWA median memory footprint increased from 222.7 MB (s.d. of 1.64 MB) on the low-end device to 235.7 MB (s.d. of 3.05 MB) on the high-end device (an increase of 5.8%). The Mann-Whitney U test demonstrated a statistically significant difference between the device classes for native ($W = 78$, $p < 0.001$), React Native ($W = 42.5$, $p < 0.001$) and PWA ($W = 400$, $p < 0.001$), with no significant difference found for Flutter ($W = 177$, $p = 0.271$).

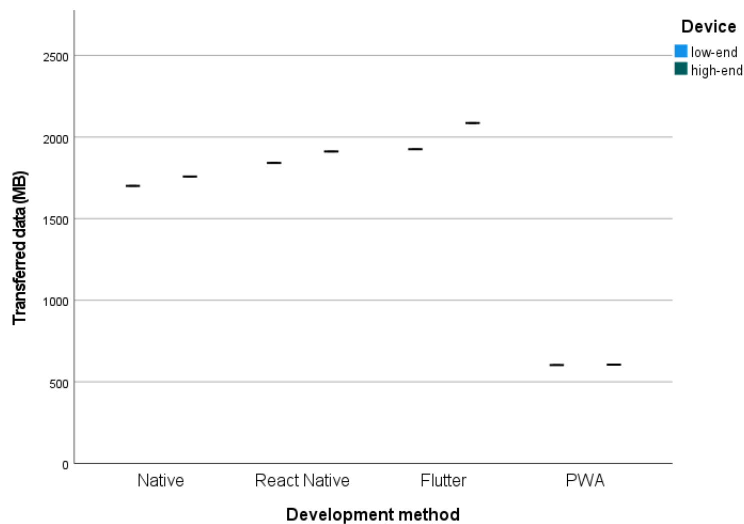


Figure 5.16: Boxplot including median Android network-intensive total data transferred by development method by device. Dots indicate outliers.

Figure 5.16 summarizes the results for median total data transferred of the network-intensive on Android

devices. As the chart shows for the Android low-end device, median total data transferred for PWA was by far the lowest at 603 MB (s.d. of 0.82 MB), followed by native 1701 MB (s.d. of 0.98 MB), React Native 1842 MB (s.d. of 1.41 MB) and Flutter 1926 MB (s.d. of 0.81 MB). The Mann-Whitney U test demonstrated a statistically significant difference in median total data transferred in the network-intensive on the Android low-end device between PWA and native ($W = 0$, $p < 0.001$), between native and React Native ($W = 0$, $p < 0.001$), and between React Native and Flutter ($W = 0$, $p < 0.001$).

On the Android high-end device, PWA, by far, had the lowest median total data transferred at 605.5 MB (s.d. of 1.05 MB), followed by native 1758 MB (s.d. of 0.93 MB), React Native 1912 MB (s.d. of 1.27 MB) and Flutter 2086 MB (s.d. of 1.70 MB). The Mann-Whitney U test demonstrated a statistically significant difference in median total data transferred in the network-intensive on the Android high-end device between PWA and native ($W = 0$, $p < 0.001$), between native and React Native ($W = 0$, $p < 0.001$), and between React Native and Flutter ($W = 0$, $p < 0.001$).

Statistically significant differences were seen between the Android low-end and high-end devices for median total data transferred of the network-intensive of each development method. Native median total data transferred increased from 1701 MB (s.d. of 0.98 MB) on the low-end device to 1758 MB (s.d. of 0.93 MB) on the high-end device (an increase of 3.3%). React Native median total data transferred increased from 1842 MB (s.d. of 1.41 MB) on the low-end device to 1912 MB (s.d. of 1.27 MB) on the high-end device (an increase of 3.8%). Flutter median total data transferred increased from 1926 MB (s.d. of 0.81 MB) on the low-end device to 2086 MB (s.d. of 1.70 MB) on the high-end device (an increase of 8.3%). PWA median total data transferred increased from 603 MB (s.d. of 0.82 MB) on the low-end device to 605.5 MB (s.d. of 1.05 MB) on the high-end device (an increase of 0.4%). The Mann-Whitney U test demonstrated a statistically significant difference between the device classes for native ($W = 400$, $p < 0.001$), React Native ($W = 400$, $p < 0.001$), Flutter ($W = 400$, $p < 0.001$) and PWA ($W = 368$, $p < 0.001$).

Figure 5.17 summarizes the results for median execution time of the network-intensive on iOS devices. As the chart shows, for the iOS low-end device, median execution time for PWA was the lowest at 6049 ms (s.d. of 1055.14 ms), followed by native 7273 ms (s.d. of 764.48 ms) and React Native 9442.5 ms (s.d. of 140.40 ms). The Mann-Whitney U test demonstrated a statistically significant difference in median execution time in the network-intensive on the iOS low-end device between PWA and native ($W = 104$, $p = 0.004$), and between native and React Native ($W = 380$, $p < 0.001$).

On the iOS high-end device, PWA, by far, had the lowest median execution time at 4917 ms (s.d. of 606.26 ms), followed by React Native 5438.5 ms (s.d. of 280.24 ms) and native 6468.5 ms (s.d. of 404.59 ms). The Mann-Whitney U test demonstrated a statistically significant difference in median execution time in the network-intensive on the iOS high-end device between PWA and React Native ($W = 63$, $p < 0.001$), and between React Native and native ($W = 15$, $p < 0.001$).

Statistically significant differences were seen between the iOS low-end and high-end devices for median execution time of the network-intensive of each development method. Native median execution time decreased

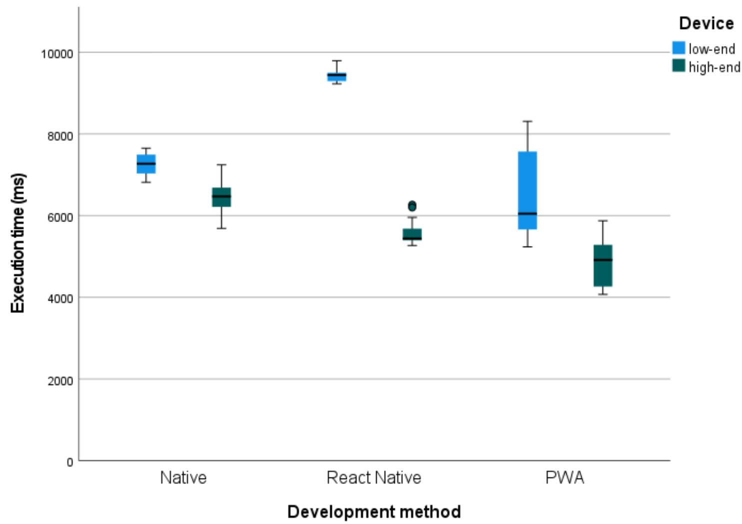


Figure 5.17: Boxplot including median iOS network-intensive execution time by development method by device. Dots indicate outliers.

from 7273 ms (s.d. of 764.48 ms) on the low-end device to 6468.5 ms (s.d. of 404.59 ms) on the high-end device (a decrease of 11%). React Native median execution time decreased from 9442.5 ms (s.d. of 140.40 ms) on the low-end device to 5438.5 ms (s.d. of 280.24 ms) on the high-end device (a decrease of 42.4%). PWA median execution time decreased from 6049 ms (s.d. of 1055.14 ms) on the low-end device to 4917 ms (s.d. of 606.26 ms) on the high-end device (a decrease of 18.7%). The Mann-Whitney U test demonstrated a statistically significant difference between the device classes for native ($W = 21$, $p < 0.001$), React Native ($W = 0$, $p < 0.001$), and PWA ($W = 25$, $p < 0.001$).

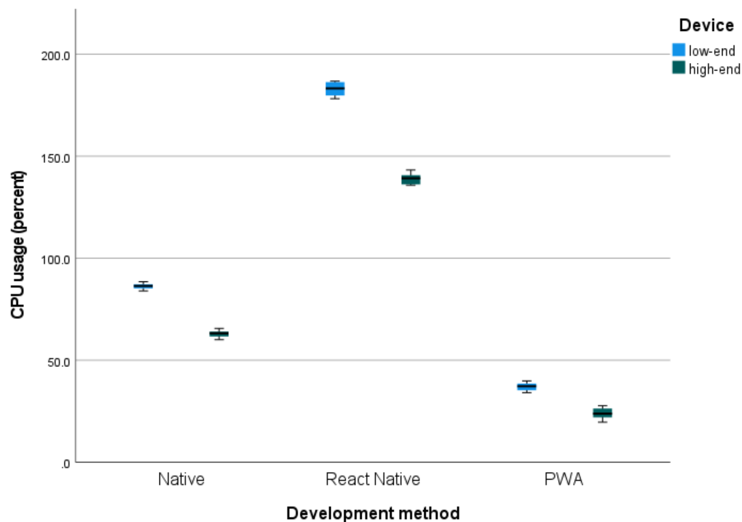


Figure 5.18: Boxplot including median iOS network-intensive CPU usage by development method by device. Dots indicate outliers.

Figure 5.18 summarizes the results for median CPU usage of the network-intensive on iOS devices. As the

chart shows, for the iOS low-end device, median CPU usage for PWA was the lowest at 37.2% (s.d. of 1.89%), followed by native 86.4% (s.d. of 1.33%) and React Native 183.2% (s.d. of 3.28%). The Mann-Whitney U test demonstrated a statistically significant difference in median CPU usage in the network-intensive on the iOS low-end device between PWA and native ($W = 0$, $p < 0.001$), and between native and React Native ($W = 0$, $p < 0.001$).

On the iOS high-end device, PWA, by far, had the lowest median CPU usage at 23.8% (s.d. of 2.28%), followed by native 63.2% (s.d. of 1.59%) and React Native 139.2% (s.d. of 2.49%). The Mann-Whitney U test demonstrated a statistically significant difference in median CPU usage in the network-intensive on the iOS high-end device between PWA and native ($W = 0$, $p < 0.001$), and between native and React Native ($W = 0$, $p < 0.001$).

Statistically significant differences were seen between the iOS low-end and high-end devices for median CPU usage of the network-intensive of each development method. Native median CPU usage decreased from 86.4% (s.d. of 1.33%) on the low-end device to 63.2% (s.d. of 1.59%) on the high-end device (a decrease of 26.8%). React Native median CPU usage decreased from 183.2% (s.d. of 3.28%) on the low-end device to 139.2% (s.d. of 2.49%) on the high-end device (a decrease of 24%). PWA median CPU usage decreased from 37.2% (s.d. of 1.89%) on the low-end device to 23.8% (s.d. of 2.28%) on the high-end device (a decrease of 36%). The Mann-Whitney U test demonstrated a statistically significant difference between the device classes for native ($W = 0$, $p < 0.001$), React Native ($W = 0$, $p < 0.001$) and PWA ($W = 0$, $p < 0.001$).

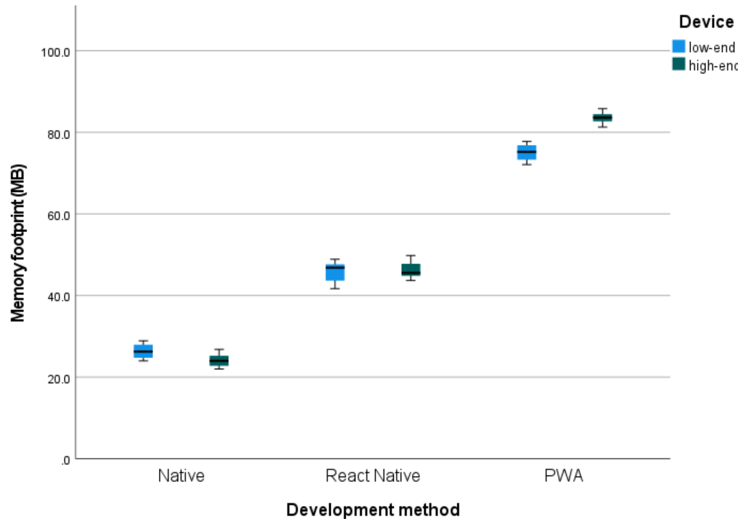


Figure 5.19: Boxplot including median iOS network-intensive memory footprint by development method by device. Dots indicate outliers.

Figure 5.19 summarizes the results for median memory footprint of the network-intensive on iOS devices. As the chart shows, for the iOS low-end device, median memory footprint for native was the lowest at 26.2 MB (s.d. of 1.64 MB), followed by React Native 46.8 MB (s.d. of 2.42 MB) and PWA 75.2 MB (s.d. of 1.93 MB). The Mann-Whitney U test demonstrated a statistically significant difference in median memory

footprint in the network-intensive on the iOS low-end device between native and React Native (W = 0, p < 0.001), and between React Native and PWA (W = 0, p < 0.001).

On the iOS high-end device, native had the lowest median memory footprint at 24 MB (s.d. of 1.56 MB), followed by React Native 45.5 MB (s.d. of 1.95 MB) and PWA 83.6 MB (s.d. of 1.32 MB). The Mann-Whitney U test demonstrated a statistically significant difference in median memory footprint in the network-intensive on the iOS high-end device between native and React Native (W = 0, p < 0.001), and between React Native and PWA (W = 0, p < 0.001).

Statistically significant differences were seen between the iOS low-end and high-end devices for median memory footprint of the network-intensive of each development method. Native median memory footprint decreased from 26.2 MB (s.d. of 1.64 MB) on the low-end device to 24 MB (s.d. of 1.56 MB) on the high-end device (a decrease of 8.4%). React Native median memory footprint decreased from 46.8 MB (s.d. of 2.42 MB) on the low-end device to 45.5 MB (s.d. of 1.95 MB) on the high-end device (a decrease of 2.7%). PWA median memory footprint increased from 75.2 MB (s.d. of 1.93 MB) on the low-end device to 83.6 MB (s.d. of 1.32 MB) on the high-end device (an increase of 11.1%). The Mann-Whitney U test showed a significant difference between the device classes for native (W = 65.5, p < 0.001) and PWA (W = 400, p < 0.001). By contrast, no significant difference was found for React Native (W = 218, p = 0.691).

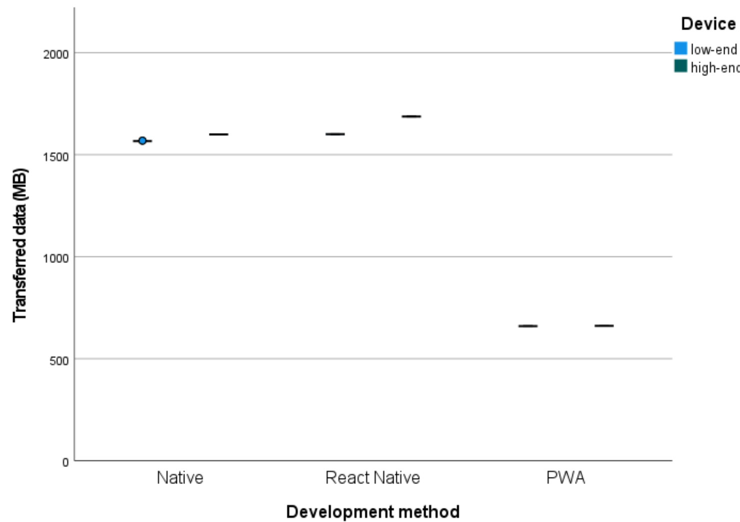


Figure 5.20: Boxplot including median iOS network-intensive total data transferred by development method by device. Dots indicate outliers.

Figure 5.20 summarizes the results for median total data transferred of the network-intensive on iOS devices. As the chart shows, for the iOS low-end device, median total data transferred for PWA was by far the lowest at 660 MB (s.d. of 0.95 MB). Native and React Native had median total data transferred of 1567 MB (s.d. of 1.22 MB) and 1601 MB (s.d. of 1.72 MB), respectively, on the iOS low-end device. Median total data transferred in the network-intensive on the iOS low-end device between PWA and native (W = 0, p < 0.001), and between native and React Native (W = 0, p < 0.001).

On the iOS high-end device, PWA, by far, had the lowest median total data transferred at 661.5 MB (s.d. of 0.74 MB). Native and React Native had the median total data transferred of 1599.5 MB (s.d. of 0.93 MB) and 1687.5 MB (s.d. of 1.35 MB), respectively, on the iOS high-end device. The Mann-Whitney U test demonstrated a statistically significant difference in median total data transferred in the network-intensive on the iOS high-end device between PWA and native ($W = 0$, $p < 0.001$), and between native and React Native ($W = 0$, $p < 0.001$).

Statistically significant differences were seen between low-end and high-end devices for median total data transferred of the network-intensive of each development method in iOS. Native median total data transferred increased from 1567 MB (s.d. of 1.22 MB) on the low-end device to 1599.5 MB (s.d. of 0.93 MB) on the high-end device (an increase of 32.5%). React Native median total data transferred increased from 1601 MB (s.d. of 1.72 MB) on the low-end device to 1687.5 MB (s.d. of 1.35 MB) on the high-end device (an increase of 5.4%). PWA median total data transferred increased from 660 MB (s.d. of 0.95 MB) on the low-end device to 661.5 MB (s.d. of 0.74 MB) on the high-end device (an increase of 0.2%). The Mann-Whitney U test demonstrated a statistically significant difference between the device classes for native ($W = 400$, $p < 0.001$), React Native ($W = 400$, $p < 0.001$), and PWA ($W = 325$, $p < 0.001$).

5.6 Discussion

This section first reports the main results in resource use and performance experiments and then interprets the findings and discusses major factors contributing to such results. The section subsequently discusses the overall results for each development method. Finally, the overall results are compared to the findings of previous studies.

The resource use and performance of experiments conducted in this chapter yielded 11 main results:

- React Native exhibited a significantly lower test execution time than for the other development methods in the CPU-intensive tests on both platforms.
- There was a significant effect of device class on the test execution time of all the development methods in the CPU-intensive tests on both platforms, with the exception of the Android native app.
- For the CPU-intensive tests on iOS, PWA exhibited a significantly lower CPU usage than other development methods.
- In the memory-intensive tests on Android, the native and PWA apps were significantly faster in terms of test execution time and used less memory than their React Native and Flutter counterparts.
- PWA failed to complete the memory-intensive tests on the low-end iOS device.
- When compared to its iOS counterparts, the native app exhibited the highest execution time in the memory-intensive test.

- High levels of CPU usage were observed in the memory-intensive test results.
- There was a significant effect of device class on the execution time of the memory-intensive tests.
- On Android, Flutter had the fastest execution time in the network-intensive test.
- During the network-intensive test, Flutter was unstable on the Android platform and failed to complete the test on the iOS platform.
- PWA had the lowest total data transferred in the network-intensive tests on both platforms.
- There was a significant effect of device class on the execution time of the network-intensive tests.

Why did React Native suffer a significantly higher execution time than the other development methods in the CPU-intensive tests on both platforms? There was a significant disparity between the execution time results of React Native and the other development methods in the CPU-intensive tests. Results on both platforms revealed that its execution times were 1244% and 934% of those for the next slowest method on the low-end and high-end Android devices, respectively. On the iOS, they were 1675% and 1228% of those for the next slowest method on the low-end and high-end iOS devices, respectively.

React Native's architecture was one of the factors that contributed to its poor execution time results in the CPU-intensive tests. As discussed in 3.3, within React Native, business logic, user interface updates, layout and native operations make use of three different threads, namely: The JavaScript, Background and Main (UI) threads. The JavaScript thread is where all of the business logic is defined. Results from the JavaScript thread will be subsequently sent over to the Background thread, and from there to the Main (UI) thread. Communication between JavaScript and native is possible through a bridge (in an asynchronous fashion) where messages are batched and serialized into JSON. By contrast, as mentioned in 3.2, Flutter did not rely on a bridge and decreased the dependency of app behaviour on the underlying operating system by providing its own implementations of UI controls, amongst other components. This architectural difference may be one contributor to the empirically observed performance gap between Flutter and React Native.

Another possible contributing factor could be the programming language and runtime characteristics of each development method. Dart in Flutter and Swift in iOS both compile to native code that runs fast during runtime. Java code on native Android first compiles into bytecode, an intermediate form of representation that allows it to be translated at runtime into machine-level code [129]. It is then run by ART, a managed runtime used by applications and some system services on Android [130]. The runtime speed of JavaScript code depends on the JavaScript engine on which it runs. The JavaScript engine is a virtual machine that compiles JavaScript code. Although the earliest JavaScript engines were interpreters — yielding a slower runtime — modern engines are using JIT (just-in-time) compilation for improved performance [131]. React Native uses JavaScriptCore engine, the same JavaScript engine that powers Safari. But unlike Safari, React Native's JavaScriptCore on iOS does not use JIT due to the absence of writable executable memory in

iOS apps [132]. We did not find evidence that on Android, React Native's JavaScriptCode uses JIT. This dependence of React Native on interpreted — rather than compiled — JavaScript also contributed to slower execution of the ReactNative app in CPU-intensive tests.

React Native and PWA had a similar implementation and codebase structure in JavaScript as both were React-based, yet PWA managed to outperform React Native and had a execution time close to that of other methods in the CPU-intensive tests. One factor contributing to PWA's superior performance in the CPU-intensive tests compared to React Native is the architectural differences between the two. PWA is designed to run natively in the browser. In other words, its HTML, CSS and JavaScript run directly in the browser and communicate directly to DOM (a programming interface for web documents in the browser) as opposed to the way React Native communicates with the native platform through a bridge.

Why was there a significant effect of device class on execution time results of all the development methods in the CPU-intensive tests on both platforms (except for the native app on Android)? The first contributing factor for this difference across device classes lay in the devices' hardware capabilities. In comparison to its low-end device counterpart, the high-end Android device had a more capable CPU with a higher clock speed (2840 MHz vs. 2450 MHz) and higher memory bandwidth (34.1 GB/s vs. 29.8 GB/s) [133]. Similarly, in comparison to its low-end counterpart, the iOS high-end device was recommended by a more capable CPU, with a higher CPU clock speed (3100 MHz vs. 2380 MHz) and higher memory bandwidth (42.7 Gbit/s vs. 14.9 Gbit/s) [134].

The second contributing factor was specific to Android, and lay in the age of the software involved. The low-end Android device supported the older version of Android OS version 9, whereas the high-end device supported Android 11. By contrast, both iOS devices supported the latest iOS version — version 15. The newer OS version and its related software optimization, along with the higher level of hardware capabilities of the high-end devices, were factors contributing to improved results in the CPU-intensive tests.

Why did PWA on iOS and in the CPU-intensive tests have a significantly lower CPU usage than did the other development methods? In the CPU-intensive tests, the PWA app on iOS had a median CPU usage of 82.7% and 70% on the low-end and high-end iOS devices, respectively, while the results for the other development methods were more than 99%.

Unlike the other development methods' apps that are run directly by the operating system, the PWA app on iOS, ran in the browser, and that browser (i.e., Safari) managed the app's resources use, including the CPU usage. One possible reason for the low CPU usage of PWA compared to the other apps would be the higher resource use restrictions imposed by Apple on browser apps. Such restrictions stand in contrast to the looser restrictions in place for native apps that have undergone the review process of the Apple App Store prior to distribution.

In the memory-intensive tests on Android, why did the native Android and PWA apps complete significantly faster and use less memory than did their React Native and Flutter counterparts? Execution times of the native Android app were 808% and 1046% of those for React Native and Flutter on the low-end device and 668% and 977% of those (for React Native and Flutter) on the high-end device, respectively. The same results for the PWA app were 252% and 344% of those for React Native and Flutter on the low-end device and 433% and 648% of those (for React Native and Flutter) on the high-end device, respectively. Their memory footprint during the test was nearly half of what React Native and Flutter apps used.

One factor that contributed to this disparity was that native Android and PWA apps were designed to run natively on the respective platforms — the Android operating system for native Android and the browser for PWA — as opposed to the React Native and Flutter apps that have to take differences between running on the Android and iOS platforms into account. The overhead of supporting more than one platform in React Native and Flutter has affected their efficiency of resource use and performance in the memory-intensive tests on Android.

Why did PWA fail to complete the memory-intensive tests on the low-end iOS device? The PWA app on the low-end iOS device failed to complete the memory-intensive test because of its large memory footprint. It managed to complete the test on the high-end iOS device since it was equipped with a larger amount of available memory — 4 GB in comparison to 2 GB in the low-end device.

Why was there a high level of CPU usage in the memory-intensive test results? A memory unit consists of smaller components called pages. The memory-management system in an OS allocates memory for a process by selecting free pages or other processes' pages that are unlikely to be needed. This process is called "page scan", and it represents a resource-intensive task. [135]

Memory management systems are often based on the least recently used algorithm, and copy certain pages of memory to disk to free up that memory for allocation by other processes. These activities, amongst other things, require processing power, which resulted in a higher CPU use in the memory-intensive tasks [135].

Why was there a significant effect of device class on the execution time of the memory-intensive tests? The memory-intensive tests were both CPU and memory-intensive. As discussed earlier, the high-end devices on both Android and iOS had more capable CPUs. In addition, they were equipped with faster and larger memory units. In terms of memory size, the high-end devices' memory was twice as large as the low-end. These along with more up-to-date software on the high-end devices — especially on Android — contributed to lower execution time on the memory-intensive tests.

Why did Flutter have the fastest execution time in the network-intensive test on Android, and why was it unstable on Android and failed to complete the network-intensive test on iOS? The Flutter app was by far the fastest of the apps produced by the development methods in terms of finishing the network-intensive test on Android, despite failing in more than half of the test rounds. We had to repeat the test 43 times to have it completed 20 times (in accordance with the protocol applied in this chapter). On the iOS platform, the Flutter app failed to finish any network-intensive test rounds on both the low-end and high-end devices.

The primary reason that the Flutter app failed partially in Android and completely in iOS in the network-intensive test was rooted in the number of network connections that remained open while it was making API calls to the back-end API. Failures were due to having too many open network connections.

Making the HTTP calls over many connections was also the reason behind the Flutter app's faster execution time in the network-intensive tests. Our investigation showed that throughout the network-intensive tests, on average, the Flutter app used 400 HTTP connections. Other development methods in this test only used 5.

We tried to reduce the number of HTTP connections for the Flutter app by applying different implementations based on the official document [136] and the community suggestions to fix the issue with many open network connections, but all attempts led to the same number of open HTTP connections and consequently same test outcomes.

Why did the PWA app result in the lowest total data transferred in the network-intensive tests on both platforms? The PWA app's total data transferred on Android was 603 MB on the low-end device and 605 MB on the high-end device. The app with the next lowest total data transferred was the native app, with 1701 MB transferred on the low-end device and 1758 MB on the high-end device.

One contributing factor impacting the size of the data transferred over the network was the algorithm used in the HTTP compression. HTTP compression refers to the built-in capability of servers and clients to improve transfers speed and bandwidth utilization [137]. Here there are two common optimized algorithms for text: "gzip" and "br", short for "Brotli". The former is widely used and supported by many clients and servers, while the latter is a modern compression algorithm, developed by Google to deliver a better compression ratio [138]. Client-side support of Brotli at this point is limited to modern browsers and a handful of other clients [138].

For a certain compression algorithm to work, both the client and server should support it. The back-end API that was used in the network-intensive tests supported both "gzip" and "br" compression algorithms. However, further investigation revealed that amongst the apps created for this chapter, only PWA employed this modern compression method.

A higher rate of compression of data transferred, along with running on browsers that are amongst the most network-optimized software platforms, contributed to a lower total data transfer by PWA compared to

other development methods that run by the OS.

Why was there a significant effect of device class on the execution time of the network-intensive tests? Resource usage results of the network-intensive tests showed a high level of CPU usage. Multiple concurrent calls to the back-end API and serialization/de-serialization of data were amongst the most process-intensive tasks in the tests.

As mentioned earlier, the high-end devices on both platforms had more capable hardware, including faster and more efficient CPUs [133, 134]. In addition, for the Android devices, a newer OS version was running on the high-end device. These were amongst attributes that made the high-end devices on iOS and Android faster in the network-intensive tests.

5.6.1 Overall results for each development method

The results of all the experiments in this chapter showed that the native Android and native iOS apps were the overall best in terms of resource use and performance on the Android and iOS platforms, respectively. Flutter and PWA ranked second and third, with a slight difference. The React Native app had the worst overall results.

Overall results showed that amongst the apps targeting the Android platform, the native app had the best results in the resource-intensive tests in this chapter. The native Android app was by far the fastest and most resource-efficient app in the CPU-intensive test. In the memory-intensive test, it was the fastest app to finish the test while it imposed the least memory footprint. However, its CPU usage was slightly more than for the other apps. In the network-intensive test, the Android native app lagged behind the PWA and Flutter apps in terms of execution time; however, the native app was the best in terms of CPU and memory usage. All in all, the native Android app delivered high performance and low resource use, which contributed to having the overall best results on the Android platform.

Amongst the apps targeting the iOS platform, the native app provided the best results in the resource-intensive tests in this chapter. In the CPU-intensive test, the iOS native app was the fastest and used the least amount of memory, while having a CPU usage very similar to React Native and Flutter but greater than that of PWA. In the memory-intensive test, the iOS native app was the fastest in terms of execution time and had the smallest memory footprint; however, React Native and Flutter performed more favourably in terms of CPU usage. In the network-intensive test, native was the best after the PWA app. On that test, the iOS native app ranked second in execution time, CPU usage and total data transferred after PWA. In terms of memory use, however, the iOS native app was the best, achieving the lowest memory consumption. Altogether, the native iOS app achieved the best overall results in this chapter on the iOS platform.

Flutter app's overall results in this chapter ranked it as the second most favourable app after native apps on both platforms. In the CPU-intensive test on Android, the Flutter app had the second-fastest execution time after native, and exhibited CPU and memory usage results that were less favourable than for native,

while still very close to the PWA app. The ranking with respect to CPU-intensive results on iOS was very similar to that on Android: The Flutter app had the overall best results after native iOS, and was closely followed by the PWA app. Flutter had the worst memory-intensive results on the Android platform, having the largest execution time and memory usage. Its memory-intensive results on iOS, however, were much better, where had the best results, by a small margin, after the native app. In the network-intensive test on Android, when it completed the test, the Flutter app was the best, with the fastest execution time and relatively low CPU and memory usage. However, as emphasized earlier, the Flutter app used a large number of open HTTP connections during the test, which improved its performance but at the same time made it unstable in the tests. On iOS, the Flutter app was not able to finish the network-intensive tests. All test results considered, the Flutter app was the second-best in the resource-intensive tests on both platforms in this chapter.

Overall results of the PWA app ranked it in third place after the native and Flutter apps. Although the PWA app ranked third in this chapter, its overall results were very close to those of the Flutter app. In the CPU-intensive test on Android, the PWA app ranked third in terms of execution time, and had the second-lowest CPU and memory usage in that test on Android. The CPU-intensive test results on iOS showed results similar to those on Android, except in CPU usage. The PWA app had the lowest CPU usage in the CPU-intensive test on iOS. PWA results in the memory-intensive tests were quite different between the two platforms (i.e., Android and iOS). On Android, the PWA app had the best overall results after the native app, with very good results in terms of execution time and CPU and memory usage. On iOS, however, the PWA app was unstable and failed to complete the memory-intensive tests on the low-end device. Its results on the high-end iOS device were fastest in terms of execution time, and the PWA app on iOS had by far the highest memory usage of any app. The PWA app had its best showing in the network-intensive tests. On Android, it was the best after the Flutter app, being the second-fastest app to complete the test, and having the least total data transferred of any of the apps. PWA's results on iOS in the network-intensive test were the best overall. It had the fastest execution time and the lowest CPU, and memory usage, as well as the smallest total data transferred during the test. Altogether, the PWA app had the third-best overall results on both platforms in this chapter.

The React Native app had the worst overall resource usage and performance results in this chapter. In all the tests in this chapter, the React Native app had its worst comparative results in the CPU-intensive tests. On both Android and iOS platforms, React Native was by far the slowest app in terms of execution time and had the highest CPU and memory usage. While the React Native app did better in the memory-intensive tests, it ranked as the third overall best app — better than only Flutter on Android and only PWA on iOS. React Native app's poor overall results in the network-intensive tests made it the worst app in the test on both Android and iOS platforms. It was the slowest in terms of execution time and had the highest CPU usage in the network-intensive test on both platforms. All in all, React Native app was the least favourable app according to the overall results of the resource-intensive tests on both platforms in this chapter.

5.6.2 Overall results compared to previous work

An important element to better understanding CPDTs' trade-offs in terms of resource use and performance is to compare this chapter's overall results to similar previous work.

One of the first studies that investigated performance and resource use was by Corral et al. [5], which reported an overall performance penalty when a CPDT was adopted. The study compared a CPDT (PhoneGap) to native Android in several resource-intensive tests, including hardware, network and data access, with a focus on the execution time metric. Another study, by Dalmasso et al. [6], assessed the memory, CPU and power consumption of two CPDTs on Android. Their results showed that PhoneGap, a web-based CPDT, had better overall results in their test, which is best characterized as network-intensive.

In part of two similar studies by Willocx et al. [8, 55], several CPDTs' resource usage — were measured during responsiveness-related tests such as app launch and navigation. The two studies reported higher CPU usage and a larger memory footprint of CPDTs compared to native apps.

In a more recent study and one similar to our work, Biørn-Hansen et al. [58] evaluated the resource usage and performance overhead of a number of CPDTs compared to the native development on Android. The study focused on measuring the performance and resource use of access to native APIs such as the geolocation API, contacts API, file system and accelerometer. The authors reported a decrease in performance and higher resource use of CPDTs in comparison to the native approach. Such contributions were some of the first studies that reported that some of the CPDTs can perform equally or better than native apps on certain metrics.

The overall results of the aforementioned previous work that assessed CPDTs trade-offs in terms of resource use and performance indicate that in many cases, there was resource use and performance overhead of adopting CPDTs compared to the native approach. Nevertheless, the newer studies are showing that modern CPDTs are catching up to, and in some cases performing better, than native apps.

The overall results of our work in this chapter were in line with the related literature showing that native approaches still have the upper hand in terms of resource use and performance, but that some modern CPDTs are coming close and, in some cases, even outperforming native apps.

5.7 Summary

This chapter aimed to support investigation of CPDT resource use and performance trade-offs.

First, criteria and parameters that can contribute to evaluating perceived resource use and performance were discussed, and then four quantitative metrics were introduced. The first was CPU usage, a metric to quantify the processor load imposed by running computer programs. It was defined as the percentage of total CPU capacity being occupied at a given moment. The second metric was memory footprint, defined as the amount of main memory that a program uses or references whilst running [116]. Total data transferred served as the third metric. This was defined as the total data received and sent over a network within a

specific period of time, as measured in megabytes (MB). This metric was measured and reported only for network-intensive tests. The fourth and final metric considered was execution time, defined as the time elapsed between the start of a resource-intensive test and its completion. This metric was applied to evaluate the performance of each of the development methods for one of the resource-intensive tests in this chapter, with the metric serving to measure the time each development method required to complete that test.

The chapter next discussed the specific methods and tools adopted to measure CPU usage, memory footprint, total data transferred and execution time. CPU usage and memory footprint of all of the apps on the Android platform were measured by the Linux "top" command-line tool. On the iOS platform, all of the apps' CPU usage and memory footprint, except for the PWA app, were measured by Xcode Instrument. The PWA app's CPU usage and memory footprint on the iOS platform were measured instead using Web-inspector through Safari's remote debugging mechanism. On the Android platform, total data transferred was measured using Android Studio's Profiler for all the apps except the PWA app. On the iOS platform, for all the apps except for the PWA app, total data transferred was measured in Xcode Instrument. PWA's total data transferred on the Android platform was measured using the Chrome Developer Tools; on the iOS platform, it was measured by the Safari Web-inspector. Execution time (For each resource-intensive test) was measured by logging elapsed time between the moment the button that starts a test is pressed and the point at which that test completes. Tests on each platform (Android and iOS) were conducted on a newer, more capable device and an older, less capable device.

The app designed for this chapter contains three resource-intensive tests, each of which can be started with a button. The CPU-intensive test was designed to assess resource use and performance, focusing on the app's CPU usage. This test used the identification of prime numbers as the CPU-intensive task. The memory-intensive test was designed to evaluate resource use and performance, with a special focus on the app's memory footprint. The memory-intensive test was based on computing the transitive closure of a graph, which is introduced as a memory-intensive algorithm[124]. The network-intensive test was designed to assess the resource use and performance of an app during intensive network operations and the transfer of data over the network. That test was based on communication and transfer of data between the apps and a back-end through a RESTful API [121]. All of the artifacts developed for this chapter are accessible in public open-source repositories on Github.

Thereafter, results were presented, interpreted, and findings were discussed. Overall results showed that amongst the apps targeting the Android platform, the native app achieved the best results in the resource-intensive tests by delivering high performance and low resource use. Amongst the apps targeting the iOS platform, the native app also achieved the best overall results in resource-intensive tests. The Flutter app's overall results supported a second-place ranking in the resource-intensive tests after native on both platforms. The PWA app had the third-best overall results on both platforms after the native and Flutter apps. Although the PWA app ranked third in this chapter, its overall results were very close to those of the Flutter app. Finally, the React Native app exhibited the worst overall resource usage and performance results in this

chapter.

Previous work that assessed CPDTs trade-offs in terms of resource use and performance indicated that in many cases, there was resource use and performance overhead of adopting CPDTs in comparison to the native approach. Nevertheless, newer studies show that modern CPDTs are catching up to, and in some cases, performing better than native apps. The overall results of this investigation were in line with the related literature showing that while native approaches retain an upper hand in terms of resource use and performance, modern CPDTs are coming close and, in some cases, even outperforming native apps.

6 Developer Experience

This chapter introduces the third component of our evaluation system to support investigation of the CPDT (cross-platform development tool) trade-offs associated with developer experience. In this chapter, we first start by defining criteria and parameters that can be measured to evaluate a perceived developer experience. Next, measurement tools and the adopted methods to measure metrics associated with defined parameters across native and CPDTs are discussed. We subsequently examine the design and implementation of the apps built for this chapter. Thereafter, assessments are introduced, experiment results are presented, interpreted, and findings are discussed.

6.1 Evaluation criteria and metrics

In this section, we discuss the importance of developer experience as a key aspect of evaluating CPDTs. This section revisits some of the related peer-reviewed work with respect to developer experience and aims to explain how this work sought to expand such work. The section continues on to introduce and define a range of metrics by which we can quantitatively measure developer experience, and then discusses the impact of and reasons for selecting each such metric.

Heitkotter et al. [47] introduced the first study that defined a set of criteria for CPDT evaluation from a software development perspective. Criteria such as development environment, GUI Design, ease of development and maintainability were defined and assessed qualitatively in tables. Maintainability was the sole criterion defined with a quantitative metric.

Palmieri et al. in 2012[48] reported that the adoption of CPDT can improve maintainability due to using a set of single source code. Hudli et al. [10] proposed a list of criteria from the developer's point of view.

Amongst more recent work, Majchrzak et el. [9] studied design and implementation paradigms such as navigation, data fetching and debugging across three CPDTs, and qualitatively labelled them as "simple", "intermediate", or "complex" — in terms of the implementation effort involved — in a high-level summary table.

Developer experience is one of the most under-researched aspects of CPDT, especially in terms of providing empirical evidence and conducting quantitative measurements. To the best of our knowledge, no quantitative study has been conducted so far in this area. In addition, modern CPDTs (e.g., React Native, PWA and Flutter) were either not available or not included in the aforementioned studies. Building on the insights from the past studies and to shed light on the trade-offs associated with the developer experience for the

modern CPDTs, this work introduced and assessed three quantitative software metrics across apps that were built for this thesis.

6.1.1 Lines of code (LOC)

Lines of code (LOC) is one of the most widely used software metrics that aims to estimate the effort of writing a program as well as maintainability once a program is written [139]. It is a metric that at the basic level, counts the number of lines of a written program.

There are multiple ways to measure LOC; each has a different level of effectiveness and accuracy and can complicate the task of building a suitable measuring tool. Source lines of code (SLOC) and logical lines of code (LLOC) are amongst the most common types of LOC [139].

SLOC commonly refers to the count of source code lines excluding comments and blank lines. This is the most common method of measurement, reflecting its relative ease of tooling and explanation, and is widely used in software metric tools. One of its downsides is the inability to exclude formatting and aspects of coding style that have little or no effect on the logic of the code. Use of the logical lines of code (LLOC) metric remedies this by measuring the number of executable statements instead of physical lines of code, excluding formatting and irrelevant code style. A major drawback associated with LLOC measurement is the difficulty in recognizing statements in programming languages. This requires a measurement tool that can identify different keywords and language-specific syntactic constructs.

Unfortunately, for our experiment, we did not find an LLOC tool that could match our requirement of assessing different programming languages that exhibit such disparate syntax and programming paradigms as those examined here. The lack of LLOC tooling on the one hand and less accurate measurement of SLOC on the other led us to build a custom LOC measurement that is more accurate than SLOC and feasible for us to implement and can take the differences between our multi-language comparison into effect. This will be discussed in additional detail in Section 6.2.1.

Please note that for notational simplicity, the quantity measured by our custom LOC measurement tool is simply called LOC throughout this thesis.

6.1.2 Cyclomatic complexity

Cyclomatic complexity is a quantitative software metric that measures the complexity of code. It is a measure of the number of linearly independent paths in source code, with the name being coined by Thomas J. McCabe in 1976 [140, 141].

Mathematically, the cyclomatic complexity of a program is computed based on its control-flow graph. The control-flow graph is a directed graph whose vertices correspond to blocks of the code, and where a given pair of vertices (X,Y) are connected with a directed edge from vertex X to Y if control passes from the block associated with X to that associated with Y. The complexity M is then defined as $M = E - N + 2P$, where

E is the number of edges of the graph, N is the number of nodes of the graph, and P equals the count of connected components [141].

There are a number of applications associated with cyclomatic complexity metric. One of the McCabe's original applications sought to limit the complexity of code during the development [142]. Based on this idea, it was recommended that during the software development, complex modules should be split into smaller ones, specifically when their cyclomatic complexity number exceeds 10. Another important application is to estimate the number of required test cases. Having a criterion to base the test count on the count of linearly independent paths could contribute to achieving higher test coverage. In addition, the correlation between McCabe's cyclomatic complexity with the number of defects in a program has been studied, with some work finding a positive correlation [143, 142].

Software metric tools often measure the McCabe cyclomatic complexity by adding points when reaching flow-related elements (e.g., returns, selections, loops and operations) in a program [144]. In this chapter, some of these McCabe-compliant tools were employed to calculate the cyclomatic complexity score of the apps that were built for this chapter. Details related to these tools will be discussed in Section 6.2.2.

6.1.3 Build time

"Build" in software development loosely refers to as the process of converting source code into a form that can be run on a computer [145]. This is a multi-step process that involves — but is not limited to — compilation, trans-compilation (the process of converting source code from one language to another language or to a newer or older version of the same language), and bundling the compiled source code and resource files into an executable form. This process is often managed by build tool(s) and targeted for a specific platform(s).

For the sake of this thesis, the "build time" will refer to the time that it takes for a program to initially build the debug version of the app (or development version for the PWA app) for a specific platform. This time will be expressed in milliseconds.

Build time is an important metric impacting developer experience in various forms, in such spheres as responding to changes, lowering the risk of merge conflicts, continuous code improvement, testing and deployment [146]. While our work only measures the build time for local development, in larger projects, build time can also impact continuous integration and deployment (CI/CD), in which the build process is a core step of the pipeline.

6.2 Measurement tools and methods

In this section, we introduce the measurement tools and methods adopted by which LOC, cyclomatic complexity score and build time were measured.

The apps we measured in this chapter are built using Flutter and React Native — systems that generate native apps to run on Android and iOS — plus native Android and native iOS apps for baseline comparison.

In addition, we measured the characteristics of our web-based app — progressive web apps that run on each of the two platforms — within the runtime of the browser.

6.2.1 LOC

In order to measure LOC across the apps that are built for this chapter and to account for the differences between their programming languages, syntax and programming paradigms, a custom tool to count the source lines of code was written based on a set of predefined rules to provide comparable LOC results.

First, we specified what file types should be included and counted. Included files consist of source code files that contain business logic, as well as files that define UI and style, such as XML files in native Android and iOS. This is necessary since some of the frameworks like Flutter and React Native have only one file type for business logic, UI or style, while Android Native and iOS native have separate file types for business logic, UI and style. PWA, on the other hand, has business logic and UI defined in the same type of file while having a separate type of file (CSS) for styles. In addition, file types that were part of the app's configuration or related to the build system, as well as auto-generated files that shipped with the starter project were excluded.

Having specified the files on which metrics should be applied, the custom LOC tool counted the total source lines of code in the files that were specified in the last step. The custom LOC tool measurements were based on a set of rules that indicate which lines should not be counted. In general, lines of code that carry no or little logic were excluded. The exclusion rules are listed as follows:

- Blank lines
- Single and multi-line comments
- Any combination of closing or opening curly braces or parenthesis without any logic code in that line
- Single return keyword
- Any line that has import and export module/class
- Lines with only auto-generated method annotations
- Closing tags in UI related code

The script is based on the ***grep*** command and is available in Appendix D. ***Grep*** is a Linux and Unix command-line tool that can search a given text file and return lines matching the search criteria [147].

6.2.2 Cyclomatic complexity

Cyclomatic complexity scores were measured using several existing code analysis tools.

Android native (Java-based), PWA (JavaScript-based) and React Native (JavaScript-based) were measured by SonarQube, an open-source platform that, amongst other things, performs static analysis of

code [148]. SonarQube did not support Swift and Dart projects at the time of writing this thesis; therefore, other tools that had support for these relatively new programming languages were used. The Dart-based Flutter app was measured using Dart Code Metrics, a software analysis tool created specifically for Dart-based projects [149]. The Swift-based Native iOS app was measured using Lizard, a code analyzer tool that supports multiple programming languages, including Swift [150].

Web URLs of the web page indicating how each tool measured the cyclomatic complexity score are available in Appendix D.

6.2.3 Build time

Build time measurements were undertaken on three different runtime environments: Android, iOS and browser. All the measurements were in milliseconds, and the apps were built in debug mode for the Android and iOS apps and development mode for the browser.

For the Android version of the apps — including React Native, Flutter and native Android — build time measurements were conducted in Android Studio (version 2020.3.1 Patch 2). Apps were built using ***Make Build*** command in Android Studio; once a build process was finished, the build time duration in milliseconds was logged in the "Event Log". The Event Log is a tool window in Android Studio that logs various types of information, including build-related events [151].

Xcode (version 13.0) was used to measure the build time of the iOS version of the apps, including React Native, Flutter and native iOS. By default, Xcode does not show the build time duration. This feature can be enabled by setting a flag in the Mac terminal. Thereafter, Xcode will display the build time duration in milliseconds as soon as the build process finishes. The command to enable displaying of the build time duration is shown in the next line.

```
defaults write com.apple.dt.Xcode ShowBuildOperationDuration YES
```

The PWA app, being a React-based web app, was built for the browser using **Node.js** (version 14.15.4) and **npm** (version 7.20.6). Node.js is a JavaScript runtime environment, and npm is its default package manager [152, 153]. The app was built for the development environment by running ***npm start*** command, with the build time duration being tracked by **gnomon** (version 1.5.0), a console logging Node.js utility that — amongst other things — logs statements with timestamps[154].

All the build time experiments were run on an Apple MacBook Pro 15-inch laptop, the late 2019 model, with a 6-Core Intel Core i7 processor (2.6 GHz) and 16 GB of memory, running macOS Big Sur (version 11.6).

Prior to the first round of build time measurement for each development method, all the non-system apps in the background were killed. We did not restart the measuring device (MacBook) before or after measurements.

6.3 App design and implementation

The app that is designed for this chapter is a data-driven app that allows a user to create, update and delete to-do items, and view all the to-do items in a list. Each to-do item has an id, title, description and date as a timestamp of the last time that it was changed.

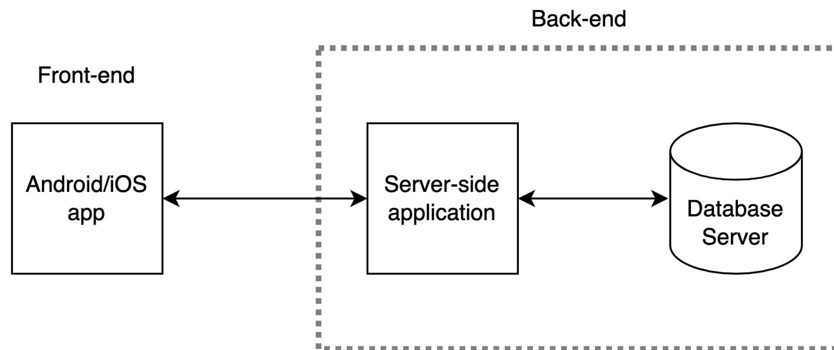


Figure 6.1: Three-tier architecture of the app

The app connects to a server-side application that provides a RESTful API to persist the to-do items in a database. The architecture of this system is based on a three-tier application design. Three-tier design is an established software architecture pattern that decouples the system into three logical and physical computing tiers: the presentation or user interface, the application — site of the logic to process data — and the data tier — where data live [155, 156]. As shown in Figure 6.1, the presentation or user interface tier for our system is provided by the mobile apps (native Android, native iOS, PWA, Flutter, and React Native) that we refer to as the “front-end” of the system. The server-side application represents the application tier and the database server is the data tier. We loosely refer to the server-side application and database server as the “back-end”. Communication between the front-end app and the server-side applications is possible by RESTful APIs.

Please note that throughout this chapter, we refer to the mobile apps (the front-end of the system) as “the apps”. They are also called “the app” when discussing common characteristics between them.

The app has three pages — “Todo List”, “Add Todo” and “Edit Todo” page, as shown in Figure 6.2. The Todo List page is the first page of the app that is shown when a user opens the app. This page consists of a page title, a button to add a to-do item and a list of existing to-do items. The list is populated by to-do items that have already been created by the user. Each item in the list has a title, a date indicating the last time that it was changed, and a description. The “Add” button allows the user to add a new to-do item by opening the Add Todo page.

The Add Todo page consists of a page title, a button to navigate back, and a form. The “Back” button

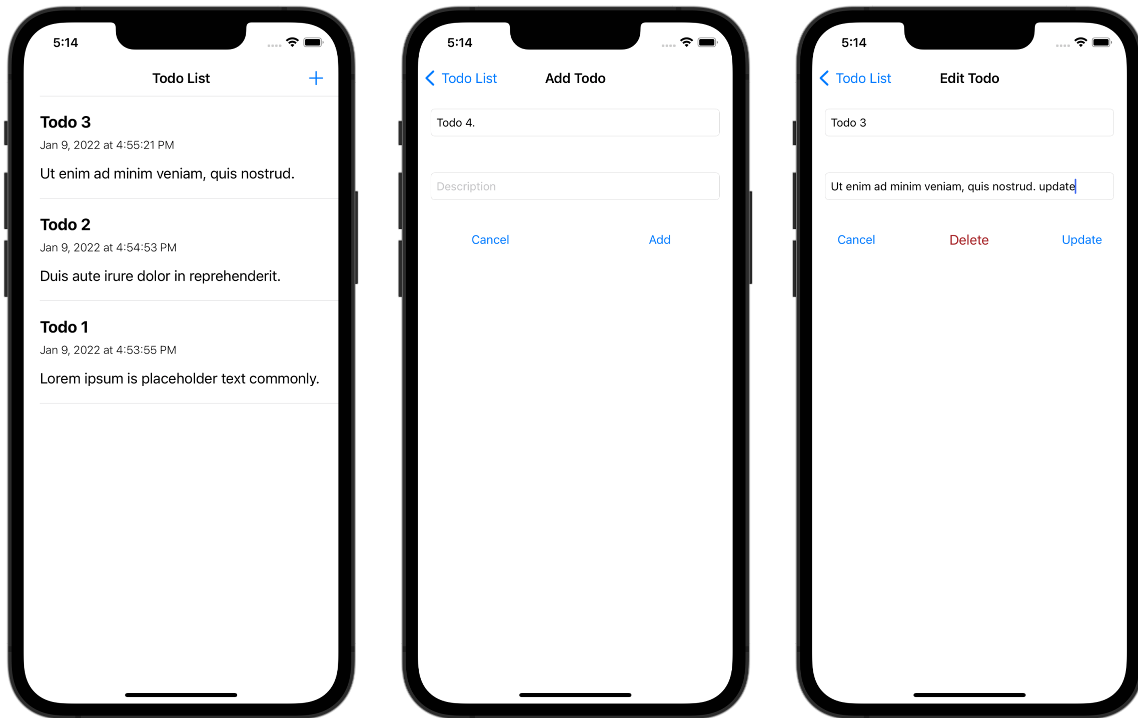


Figure 6.2: Pages of the app. From left to right: *Todo List* page, *Add Todo* page and *Edit Todo* page

takes the user to the previous page (Todo List page). The form has two text fields that allow a user to input a to-do item's title and description, along with two action buttons — *Cancel* and *Add*. The *Cancel* button discards the changes on this page and takes the user back to the Todo List page. The *Add* button saves the to-do as a new to-do item. Add functionality is triggered only if the form is valid. Form validation fails if either a to-do item's title or description text field is empty. An error message is displayed under each text field (title or description) that is empty. After a to-do item is successfully added, the user will be taken to the Todo List page, where the newly added to-do item is appended to the list.

The Edit Todo page is opened when a user on the Todo List page clicks on one of the to-do items in the list. The Edit Todo page consists of a page title, a back button and a form. The "Back" button takes the user to the previous page (Todo List page). The form has two text fields (title and description) that are populated by the title and description of the to-do item on which the user initially clicked on the Todo List page. There are three form action buttons. The *Cancel* button discards the changes and takes the user back to the Todo List page. The *Delete* button deletes the item and takes the user back to the Todo List page (but with the item deleted from the list). The *Update* button updates the to-do item with the new title and description in the form. Update functionality triggers only if the form is valid. The form is valid when neither of title nor description text field is empty. An error message is displayed under each text field (title or description) that is empty. After the item is updated successfully, the user will be taken to the Todo List

page and the updated to-do item will be shown in the list.

The apps are connected to a RESTful API that persists data (to-do items) in a database. The RESTful API allows the mobile apps to create a new to-do item, fetch a single or all of the to-do items, update an existing item, or delete one from the database through API calls.

Multiple versions of the app were developed for this chapter using each of the development methods: native Android, native iOS, Flutter, React Native and PWA. Considering the differences between these development methods in terms of programming languages and design patterns, and the direct impact of implementation on the measurements such as LOC and cyclomatic complexity and — to some extent — on build time, a set of implementation rules were established prior to the development of the apps. This step was undertaken to reduce the potential implementation bias and to enhance the comparability of the results.

The following rules were rendered as a blueprint during the development:

- A to-do item data model is required and should have an id, title, description and a timestamp representing the last time there was a change to this to-do item.
- An API class/module is required to handle API calls, including calls to get a list of to-do items, a single to-do item, add, update and delete a to-do item.
- API calls should be asynchronous and potential errors should be displayed in the console.
- Data passed through API calls should be serialized to JSON format before sending, and deserialized from JSON when receiving from the back-end (API).
- A navigation mechanism is required to go back and forth between pages.
- Technically, there should be only two pages in the app — the Todo List page and a page that can reuse UI and logic to act as an Add Todo or Edit Todo page based on the argument that it receives. The argument is the id of a to-do item. If an id is passed to this page, this page should fetch (get) the to-do item based on that id from the back-end (API) and populate the page's form with the fetched to-do content, and allow the user to edit. Otherwise, the page should show an empty form and allow the user to add a new to-do item.
- A Todo List page UI should consist of a page title, a list and an add button.
  - Each list item should have a title, a date in local time representing the last time the to-do item has changed, and a description.
  - Each list item should be clickable. If a user clicks on an item, it should take the user to the Edit Todo page and pass the id of the clicked to-do item.
  - The Add button should take a user to the Add Todo page. This is possible by not passing an id to that page, so the page knows that a new to-do item is to be added.

- The Edit/Add page should have a page title, a back button and a form.
  - A state should be defined for the page to show whether this page is Add Todo or Edit Todo. This page state should be set according to the receiving to-do id argument of the page.
  - If the page is Edit Todo, a to-do item based on the passed id should be fetched from the back-end (API).
  - The page title should be set according to whether this page is Add Todo or Edit Todo.
  - The back button should discard the changes that were made on this page and take the user back to the Todo List page.
  - The form should consist of a title and description text input fields and action buttons.
    - If the page is Edit Todo, title and description text fields should be populated with the content of the fetched to-do.
    - If the page is Add Todo, the title and description text fields should be empty.
    - If the page is Edit Todo, the form should have three action buttons: cancel, delete and update.
      - The cancel button should discard the changes and take the user back to the Todo List page
      - The delete button should make an API call to the back-end (API) to delete the item. Once the item is deleted from the database, the user should be taken to the Todo List page. The Todo List page should then be reloaded, with its list reflecting the changes.
      - If the form is valid, the update button should make an API call to the back-end API to update the to-do item data with the new title and description. Once the to-do item data is updated in the database, it should take the user to the Todo List page. Todo List page should then reload the page and reflect the changes.
    - If the page is Add Todo, the form should have two action buttons: cancel and add.
      - The cancel button should discard the changes and take the user back to the Todo List page.
      - If the form is valid, the add button should make an API call to the back-end (API) to add a new to-do item in the database, with the to-do's title and description entered in the form the text fields. Once the new to-do is added to the database, it should take the user to the Todo List page. The Todo List page should then be reloaded, such that the changes in the listed items are reflected.
    - The form validation rules should check that the title and description text fields are not empty (i.e., include at least one character). If either of the title or description text fields is empty after the user touched their text fields or after the user clicked add or update button, an error message should be displayed under the empty text fields.

Since style related code is the focus of part of the LOC and cyclomatic complexity measurements, we also defined the app's style prior to implementation. It can be found in Appendix D.

The native Android version was implemented using Java and Android Studio IDE (version 2020.3.1 Patch 2), along with the latest SDK and developer tools. The Retrofit (version 2.9.0) [127] library was used as an HTTP client to perform the HTTP API calls. The GSON [157] library (version 2.6.2), a Java library for serialization/deserialization of Java objects to and from JSON, was used as a converter for the Retrofit library.

The native iOS version was implemented using the Swift programming language, UIKit (a framework from Apple that provides necessary classes for building iOS apps [98]) and the Xcode IDE (version 13.0). Xcode Interface Builder and Storyboard were used for designing the UI of the app [120]. No third-party library was used in the native iOS app.

The React Native versions were built using JavaScript and Visual Studio Code (version 1.60.2) and the React Native (version 0.66.0). The React Native framework, for some of its core functionality, relies on modules that are maintained by its community, such as Navigation. We added the "React Navigation" library (version 6.0.4) for navigation [100]. "Axios" (version 0.24.0), a promise-based HTTP library [126], was used to perform HTTP API calls.

The Flutter versions were built using Flutter (version 2.5) and Visual Studio Code (version 1.60.2) using the Dart programming language. The "http" library (version 0.13.4) was used for making HTTP API calls. The "intl" library (version 0.17.0) was used to format and display dates.

The PWA version was based on the PWA template of "Create-react-app" [101] written in JavaScript and React (version 17.0.2) in Visual Studio Code (version 1.60.2). The "Workbox" library (version 5.1.4) from Google was used to power the app with a Service worker [103]. The "MaterialUI" library for React (version 5.2.5) was employed to provide the PWA version with comparable UI components [158]. As for the React Native version of the app "Axios" (version 0.24.0) was used for making HTTP API calls.

The server-side application was built using "NestJS" (version 8.0.0) [122], a Node.js server-side framework, in Visual Studio Code (version 1.60.2). Postgres (version 14.1) was used as the database server.

All of the artifacts developed for this chapter are accessible in public open-source repositories on Github, and can be found via the next link:

`https://github.com/Iman-Jamali/thesis-mobile-app-development-cross-platform-vs-native/tree/main/developer-experience`

6.4 Results

The results of developer experience measurements are presented and discussed in this section.

Software performance data, in many cases, fail to follow a normal distribution [105]. The results of build time measurements in this chapter were notably non-normal. Therefore, a non-parametric statistical approach was selected. As noted earlier, the Mann-Whitney U test (also known as a Wilcoxon Rank-Sum tests) was

also selected based on its support for a wide range of sample sizes, and the capacity to rank development methods based on ensembles of development-method-specific measurements of a particular metric.

For each specific scenario consisting of a combination of development method and platform (Android, iOS and Browser), measurement of build time experiments were repeated 20 times, thus forming ensembles of sample size 20. To compare across scenarios, the ensembles of scenario-specific outcomes were compared by non-parametric statistical tests.

For each platform, results were first ranked by their median values. One-way Mann-Whitney U Tests (also known as Wilcoxon Rank-Sum tests) were then run on each successive pair of results to evaluate if there was a statistically significant difference in the ordering. A sample size of 20 and a significance level of 0.05 were set for all the tests in this chapter. While a Mann-Whitney U test can be applied to sample sizes as small as 5 [106], a sample size of 20 was selected after considering the time constraints of conducting all tests and the effect of sample size on the statistical test's power. The R statistical computing platform [107] was used to perform statistical tests. The source code of all the statistical analyses in this chapter is available in Appendix C.

6.4.1 Lines of code

The results of LOC measurements are reported in this section.

Development Method	**LOC**
Native Android	371
Native iOS	352
React Native	174
Flutter	224
PWA	185

Table 6.1: Lines of code results of different methods of development

The LOC results for the apps built by each method of development are listed in Table 6.1. The React Native app had the smallest LOC at 174. PWA had the second smallest, with a LOC of 185. Next, Flutter had LOC of 224. The LOC number was significantly larger for native apps. Specifically, native iOS had 352 LOC, and native Android app had the largest LOC of 371.

6.4.2 Cyclomatic complexity

The results of cyclomatic complexity score measurements are reported in this section.

Table 6.2 shows the cyclomatic complexity score of the apps built by different development methods. As shown by the table, PWA had the lowest cyclomatic complexity score at 39. This was closely followed by React Native, with a cyclomatic complexity score of 40. The Flutter cyclomatic complexity score of 42 was

Development Method	Cyclomatic Complexity
Native Android	63
Native iOS	53
React Native	40
Flutter	42
PWA	39

Table 6.2: Cyclomatic complexity score for different methods of development

slightly greater than that of PWA and React Native. Native iOS and native Android cyclomatic complexity scores were significantly higher, at 53 and 63, respectively.

6.4.3 Build time

This section characterized the median build time measurements for each app in milliseconds (ms).

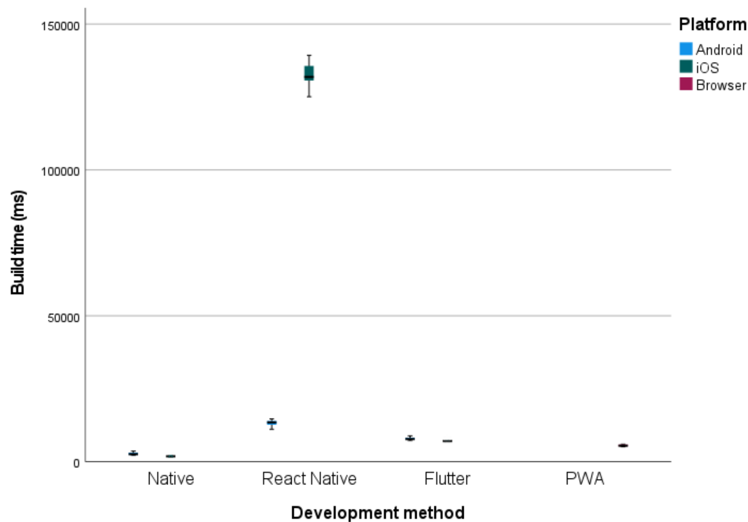


Figure 6.3: Boxplots including median build time by development method by platform. Dots indicate outliers.

Figure 6.3 summarizes the median build time of different development methods on each platform.

As the chart shows for the Android platform, the app built by native Android had the fastest median build time at 2604 ms (s.d. of 450 ms). Nearly three times slower, the Flutter app for Android had a median build time of 7789 ms (s.d. of 490 ms). Reflecting a slower build time than those for other methods on Android, the React Native app for Android had a median build time of 13490 ms (s.d. of 1078 ms).

Results for the iOS platform showed that the native iOS app exhibited by far the fastest build time, with a median of 1848 ms (s.d. of 168 ms). The Flutter app on iOS exhibited a slower build time than that of the native map, with a median build time of 7091 ms (s.d. of 196 ms). The build time of the React Native app

on iOS was the slowest, with a median of 131897 ms (s.d. of 3454 ms).

PWA's median build time was 5494 ms (s.d. of 307 ms). As mentioned in Section 6.2.3, PWA builds only for one platform — the browser, and can run on any platform with a browser.

Build time statistical analysis was based on the mobile platform on which the apps run (i.e., Android and iOS). Because the PWA app can run on both of these mobile platforms, it was included in the analysis of both the Android and iOS platforms. A Mann-Whitney U test demonstrated statistically significant difference in median build time between the native Android and PWA apps ($W = 0$, $p < 0.001$), between the PWA and Flutter Android apps ($W = 0$, $p < 0.001$), and between the Flutter Android and React Native Android apps ($W = 0$, $p < 0.001$). Similarly, on iOS, the Mann-Whitney U test showed a statistically significant difference in median build time between the native iOS and PWA apps ($W = 0$, $p < 0.001$), between the PWA and Flutter iOS apps ($W = 0$, $p < 0.001$), and between the Flutter iOS app and React Native iOS apps ($W = 0$, $p < 0.001$).

6.5 Discussion

This section first reports the main results in developer experience experiments and then interprets these findings and discusses important factors contributing to such results. The section subsequently discusses the overall results for each development method. Finally, the overall results are compared to the findings of previous work.

The developer experience experiments conducted in this chapter yielded six primary findings:

- React Native, PWA and Flutter had significantly lower LOC than did native Android and native iOS apps.
- PWA and React Native apps exhibited the lowest LOC and cyclomatic complexity scores.
- The cyclomatic complexity scores from the React Native, PWA and Flutter codebases were significantly lower than those of native iOS and native Android codebases.
- Native iOS and native Android exhibited the fastest build time.
- The build times of React Native and Flutter apps were slower than those for the others (native Android, native iOS and PWA) on the Android and iOS platforms.
- PWA's build time was faster than for Flutter and React Native, but slower than that of native Android and native iOS.

6.5.1 Explanation and interpretation of results

Why did React Native, PWA and Flutter codebases exhibit significantly lower LOC than did codebases for native Android and native iOS apps? One of the most important factors driving the

disparity in LOC results between the PWA, React Native and Flutter apps and their native counterparts (native Android and native iOS) was the difference in the UI programming paradigms employed in these development methods. React Native, React (the framework in which PWA was built) and Flutter employ declarative means of specifying UI behaviour, while frameworks by which native Android (Android SDK) and native iOS (UIKit) were built, use imperative methods for specifying UI behaviour.

Declarative programming is a programming paradigm that expresses the logic of a computation without describing its control flow [42]. In contrast, imperative programming relies on statements to successively evolve the state of the program [159]. Imperative programming focuses on describing how a program should work while declarative programming focuses on what the result should look like without specifying all the details of how to achieve it [160, 161].

The declarative approach to UI programming — adopted in React, React Native and PWA — allows a user to characterize what the UI should look like for any given state. Such a declarative UI framework builds/rebuilds its user interface on any state change to reflect the current state of the app. By contrast, imperative approaches to programming UI dynamics requires the programmer to explicitly update the configuration of the UI on an ongoing basis.

Delegation of the details of how an app's UI changes to a declarative UI framework, amongst other things, led to lower LOC for PWA, React Native and Flutter app codebases. The highest impact can be seen in the source code related to the Edit/Add Todo page, where the page had many states. Looking at details of LOC measurements in Appendix D, the LOC for source code of the Edit/Add Todo page for the React Native implementation was the lowest at 82, while native Android's LOC was the highest at 165 LOC, more than twice what it was for React Native.

Another differentiating factor in LOC results between the PWA, React Native and Flutter codebases and for their native counterparts (Android native and iOS native) lay in how the user interface was defined in these frameworks.

Native Android and native iOS decoupled the definition of the UI from the logic. In the Android Studio used for building native Android applications, the User Interface can be designed in the "Layout Editor", with the result stored in XML files; alternatively, it can also be defined directly in XML files. Similarly, for native iOS and in Xcode, the user interface is designed independently from the app's business logic. The "Interface Builder" editor in Xcode is used to design the user interface, with its definition then being stored in XML format.

By contrast, in React-based frameworks and Flutter, UI and related logic are coupled and defined in a single unit called "Component" in React and React Native and "Widget" in Flutter. This modern approach to describing the UI had a significant impact on reducing the LOC. Apps built with these frameworks had considerably smaller UI-related code than that of the native Android and native iOS approaches to decoupling UI and Logic. Looking at details of LOC measurements in Appendix D, UI definition code in native iOS and native Android apps accounted for 145 LOC and 132 LOC, respectively, while the LOC measurements for

the entire app for React Native, PWA and Flutter were 174, 185 and 224, respectively.

Why did PWA and React Native apps exhibit the lowest LOC and cyclomatic complexity scores? One contributing factor to lower LOC and cyclomatic complexity scores of PWA and React Native apps was how these apps dealt with API calls and data serialization/deserialization.

JavaScript Object Notation (JSON) is a standard text-based encoding format widely used to transmit data over the network. All the apps in this chapter were designed to support serialization/deserialization of data (to-do items) to and from JSON. Although all the programming languages used in this chapter offered built-in support for JSON, JavaScript was the only one that managed this without the need of any helper method or constructor, thanks to its native support of JSON (with support for JSON being part of the language definition) [162].

In addition, being the language of the web, JavaScript has a relatively simple syntax when dealing with API calls to the back-end on the server. The contrast is especially large when JavaScript is compared to the Java-based native Android app, which had the most verbose and complex code. This Java-based app required an interface and a helper class to define the cross-tier API calls, and was encumbered by a verbose syntax when instantiating the API calls and the interfaces and handling the fetch events.

Why were the cyclomatic complexity scores of React Native, PWA and Flutter significantly lower than that of native iOS and native Android? One of the most important factors contributing to this disparity in cyclomatic complexity score lies in the contrasting mechanics of apps created by "reactive" declarative UI frameworks such as React, React Native and Flutter (on the one hand) and those built by Android SDK (native Android app) and UIKit (native iOS) (on the other).

In the reactive declarative UI frameworks — including React, React Native and Flutter — business logic and the mapping of the app's state to the UI's state is defined by user code, and the framework is responsible for updating the UI as the app's state changes at runtime.

By contrast, Native Android and native iOS apps were built with a UI framework that follows a more traditional approach. In these frameworks, initially, the state of the UI is defined, often in a separate file in XML format. Later, at runtime, it is the responsibility of user code to imperatively update the UI. The apps built in this traditional way become more complex when the app's state grows in complexity as the developer needs to reflect the app's state changes throughout the entire UI. By contrast, in the reactive frameworks, the user only creates UI descriptions, and the framework takes care of both creating and updating the UI. In React (as defined in Section 2.1.3), the process of figuring out how to efficiently update the UI based on the program's state changes is called reconciliation [43].

The process of syncing and effectively updating the Virtual DOM with the real DOM (of the browser) based on the most recent changes in the program is called reconciliation.

Why did the native iOS and native Android apps exhibit the fastest build time? To better understand the disparity in the build time of the development methods on the Android and iOS platforms, we first characterize the build process across these methods, and then discuss why the native apps (native Android and native iOS) were faster.

The build process for iOS in Xcode involves multiple steps. The first steps involve the compilation of the source code file (whether in Swift or Objective C), the Storyboard and the app's assets. Then, the ***info.plist*** file that contains essential configuration information for the bundled executable is read by the build system to finally link the project by producing an executable or library output and creating the app bundle [163].

The Android build process uses the Gradle build system to compile source code and resources of the app as well as dependencies, including dependency modules and libraries. Source code is converted into DEX (Dalvik Executable) files and everything else into complied resources. Next, a packager combines DEX files and complied resources into a bundled executable to be installed onto an Android device. The process continues on by having the bundled executable signed and optimized for installation onto an Android device [164].

Several factors contribute to the faster measured build times of native Android and native iOS apps. First, the native apps had little or no external dependencies, unlike other development methods that relied on external packages for some of their functionality. Such minimal dependencies considerably reduced the build time. Second, native apps (native Android and native iOS) were built to only one platform and did not have the overhead of considering the code and resources required to support multiple platforms, such as layout engines, graphics engines and user interface components. Third, native apps can rely heavily upon platform-specific code that has already shipped with the platform and therefore do not have to include required elements in the project source code, resources and libraries. As a result, they can have a relatively smaller project size which contributed to their faster build time.

Why was the build time of React Native and Flutter apps slower than for the others (native Android, native iOS and PWA) on the Android and iOS platforms? The React Native app in Xcode and Android Studio had a long list of dependencies, including "Yoga" (a layout engine), "Flipper" (a mobile app debugger), and many dependencies related to providing real-time communication (RTC) between JavaScript and native parts of the app. In addition, some of the dependencies that required native features that were not available on the JavaScript side of the React Native app had a matching native library (on the native side of the React Native app) to provide that native feature (as discussed in Section 3.3), with that support often being provided by communicating with the original JavaScript library through the bridge. One of the implications of React Native's reliance on such communication between the two sides (JavaScript and Native) is the need for a large number of dependencies in JavaScript and the native side of the React Native app, which has led to longer compile — and, by extension, build — times.

The Flutter app build time was slower than for native apps but faster than for React Native. As discussed

in Section 3.2, Flutter architecture sought to limit the dependencies of the app behaviour on the underlying operating system. Flutter has its own implementation of UI controllers and has less run-time reliance on calls to the native platform than does React Native. This resulted in a smaller number of native platform-specific dependencies, which contributed to its faster build time in comparison to React Native. At the same time, Flutter's bundling of its own set of dependencies (e.g., UI library and graphics engine) had a negative impact on its build time and made it slower than for native Android and native iOS; as described in Section 4.5.3, it also contributed to large executable sizes.

Why was the PWA build time faster than for Flutter and React Native but slower than for native Android and native iOS? The PWA app was React-based, and employed React-specific syntax (JSX), features and functionality that cannot be run on the browser out of the box. Therefore a number of tools were used to build and compile the app for the browser. This process relied on "Webpack" and "Babel", amongst other tools, to build and bundle the app for the browser. Babel is a trans-compiler that is mainly used to convert modern JavaScript to the older version of JavaScript that is compatible with most browsers [165]. Webpack is a static module bundler for JavaScript apps that can use Babel and other tools to compile JavaScript, assets and the required dependencies into one or more bundles that can be statically served to browsers [166]. The build time was prolonged by a relatively complex build system and a large number of modules and packages that were shipped with the starter project, contributing to the PWA app's unfavourable build time when compared to native Android and native iOS apps. On the other hand, when PWA is compared to Flutter and React Native, the PWA app only builds to one platform (the browser), removing some of the overhead of supporting multiple platforms, and contributing to a faster build time in comparison to the development methods that build for more than one platform (i.e., Flutter and React Native).

When comparing build times of different platforms, it bears emphasis that a single PWA build time accounts for all of the platforms on which it can run. By contrast, all the other development methods in our comparison either needed to be built separately for each platform (React Native and Flutter) or are exclusive to a single platform on which they run (i.e., Android native and iOS native). One of the most important benefits of having a one-time build is that a developer only needs to deal with the potential build-time errors once, which improves the developer experience.

6.5.2 Overall results for each development method

The results of the experiments in this chapter indicated that the PWA app offered the best overall developer experience. With a modern reactive framework and a flexible programming language that offers excellent support and concise syntax for network operations, it managed to achieve one of the smallest LOC and the lowest cyclomatic complexity score. Finally, PWA build time was the fastest amongst CPDTs and offered the unique characteristic of building only once, regardless of the platform (e.g., Android and iOS) on which

it is to run.

Flutter and React Native apps provided a very good developer experience, with Flutter conferring slight advantages compared to React Native. Results showed that both such platforms achieved low LOC and cyclomatic complexity scores, with those for React Native being slightly favourable to their Flutter comparators. Amongst the most important factors in having such low LOC and cyclomatic scores was both CPDT's modern reactive frameworks and modern programming languages. Flutter build time results were relatively slower than for native apps and PWA, while React Native build time was by far the slowest of any development method compared here. Although both Flutter and React Native build time were affected, amongst other things, by the overhead of building their apps for the two platforms (i.e., Android and iOS), the resulting overhead was more severe for React Native, since its architecture is heavily reliant on native-to-JavaScript communication.

Overall results suggested that developer experience of the native apps (native Android and native iOS) were less favourable than for CPDTs, with native Android having the worst results. LOC and cyclomatic complexity score of native iOS app was ranked second to last; native Android suffered the worst results of all of the development methods. Both of these development methods adopted a more traditional development paradigm compared to the CPDTs. This, amongst other factors, adversely impacted LOC and cyclomatic complexity, and thereby developer experience. Nevertheless, the native iOS app, with its relatively more recent Swift programming language, demonstrated better overall results than for native Android. Such native development platforms did offer one sphere of notable advantage: Build time results were the fastest on each native platform, mostly due to the absence of the overhead required to support more than one platform. Please note that the LOC and cyclomatic complexity result of native apps only accounted for one platform, in contrast to the CPDTs, which offered a single body of source code for both the Android and iOS platforms.

6.5.3 Overall results compared to previous work

Although there was a limited number of previous studies investigating CPDTs from the developer experience perspective, overall results in this chapter were in line with their findings. A study by Palmieri et al. [48] reported that the adoption of CPDTs in building apps improved maintainability. In a more recent study, Majchrzak et al. [9], in part of their work, qualitatively compared three CPDTs (React Native, Ionic and Fuse) in terms of developer experience. In a high-level summary table, they reported the complexity of navigation implementation, sidebar implementation, remote data fetching, debugging and framework setup. In four out of five categories, the complexity of implementing these features by CPDTs was labelled as "simple". Results of the quantitative work in this chapter also provided evidence that the efforts and complexity associated with building mobile apps reduce when CPDTs are adopted.

6.6 Summary

This chapter sought to investigate CPDT trade-offs from a developer experience perspective.

First, criteria and parameters that can be measured to evaluate a developer experience were discussed. Three quantitative software metrics were then introduced. The first was the widely used software metric termed lines of code. This metric serves to estimate the effort and maintenance associated with writing and maintaining a program. The second metric employed was cyclomatic complexity, a quantitative software metric measuring the complexity of code. Finally, this approach used build time as a third metric: a metric in software development loosely referring to the time required for the automated conversion of source code into a form that can be run on a computer.

Next, the chapter discussed the measurement tools and methods applied to measure the selected metrics across the apps built for the evaluation in this chapter. To deliver on this work, a custom tool was built and adopted to count the source lines of code in a more judicious manner. Cyclomatic complexity scores were measured using several existing code analysis tools. Build time measurements were undertaken on three different runtime environments: Android, iOS and browser.

Thereafter, the design and implementation of the apps built for the assessment in this chapter were examined. The app designed for this chapter is a data-driven app that allows users to create, update, and delete to-do items and view all the to-do items in a list. The app connects to a server-side application that provides a RESTful API to persist the to-do items in a database. The work in this chapter developed a version of this app using each successive development method: native Android, native iOS, Flutter, React Native and PWA. Considering the differences between these development methods in terms of programming languages and design patterns, and the direct impact of implementation on the measurements such as lines of code and Cyclomatic Complexity and — to some extent — on build time, a set of implementation rules and a style guide were established before the development of the apps. All of the artifacts developed for this chapter are accessible in public open-source repositories on Github.

Finally, results were presented, interpreted, and findings were discussed. The results of lines of code measurements are listed in Table 6.1. The cyclomatic complexity scores are listed in Table 6.2; the median build time of different development methods on each platform is shown in Figure 6.3.

The results of the experiments in this chapter indicated that the PWA app offered the best overall developer experience due to a modern reactive framework, a flexible programming language, and build time that was the fastest amongst CPDTs and offered the unique characteristic of building only once, regardless of the platform (e.g., Android and iOS) on which it is to run.

Flutter and React Native apps provided a very good developer experience, with Flutter conferring slight advantages compared to React Native. Amongst the most critical factors in lowering lines of code and cyclomatic complexity scores on the Flutter and React Native apps were both CPDT's modern reactive frameworks and modern programming languages. Although both Flutter and React Native build time were

affected, amongst other things, by the overhead of building their apps for the two platforms (i.e., Android and iOS), the resulting build time overhead was more severe for React Native, since its architecture is heavily reliant on native-to-JavaScript communication.

Overall results suggest that developer experience for the native apps (native Android and native iOS) was less favourable than for CPDTs, with native Android exhibiting the worst results. These development methods adopted a more traditional development paradigm than the CPDTs. This, amongst other factors, adversely impacted lines of code and cyclomatic complexity, and thereby developer experience. However, the build time results of such native development frameworks were the fastest on each native platform, primarily due to the absence of the overhead required to support more than one platform.

Although there was a limited number of previous studies investigating CPDTs from the developer experience perspective, overall results in this chapter were in line with their findings. The quantitative work in this chapter also provided evidence that the effort and complexity associated with building mobile apps reduce when CPDTs are adopted.

7 Conclusion

7.1 Introduction

This chapter summarizes the overall thesis findings with respect to the trade-offs associated with adopting the cross-platform development methods instead of the native approach. Contributions of the study will be characterized and the limitations that were faced in this work will be discussed. Finally, the chapter proposes opportunities and recommendations for future work.

7.2 Overall findings

This study aimed to identify trade-offs associated with adopting modern CPDTs (cross-platform development tools) in the building of data-driven apps in three areas: responsiveness and convenience of use, resource use and performance, and developer experience.

The study first identified three modern CPDTs — Flutter, React Native and PWA — selected based on the perceived developer interest and adoption rate in recent years from scientific and industry-driven surveys. Chapter 3 discussed their characteristics and architectures.

The study then conducted a three-fold quantitative evaluation of the trade-offs imposed by the selected CPDTs from the perspective of responsiveness and convenience of use, resource use and performance, and developer experience. For each evaluation component and each platform (Android and iOS), apps were built using the selected modern CPDTs, with a native app being built to serve as the baseline comparison.

The first component of the evaluation in Chapter 4 investigated responsiveness and convenience of use by measuring launch time, navigation time and size of the installer. Results showed that there were some trade-offs imposed by CPDTs — especially on the iOS platform. The native app on the iOS platform had the best overall results while on the Android, React Native and Flutter apps outperformed their native counterpart. PWA's responsiveness and convenience of use results were less favourable than other development methods on both platforms. Comparing the responsiveness and convenience of use results to that of previous work demonstrated that the newer generation of CPDTs such as Flutter, React Naive and PWA offer much greater competitiveness with native apps and in some cases exceeding the responsiveness and convenience of use of native. For example, the median navigation time of all the CPDTs (on both device classes) was faster than for the native Android app. Also, the median navigation time of the Flutter app on iOS and on the high-end device was faster than for iOS native. In addition, the size of the installer of the PWA app was significantly

smaller than for native on both platforms.

The second component of the evaluation in Chapter 5 scrutinized resource use and performance by measuring relevant metrics, such as CPU usage, memory footprint and execution time across a number of resource-intensive tests. Results indicated that there was resource use and performance overhead associated with the apps built by CPDTs. Results showed that native apps were the overall best on both platforms, with Flutter and PWA being ranked second and third. Results also showed that React Native app added significant overhead in terms of resource use and considerably reduced the performance — especially in CPU-intensive tests. The overall resource use and performance evaluation findings of this study were in line with the previous literature showing that while native approaches retain an edge in resource use and performance, modern CPDTs are shrinking the gap and in some cases outperforming native apps.

The third and final component of the evaluation in Chapter 6 investigated the developer experience by measuring three metrics: lines of code, cyclomatic complexity and build time across multi-page data-driven apps that were developed separately using each of the development tools. The app was designed based on defined rules that specified its functionalities and UI/UX. The app allowed a user to create, update and delete to-do items and persisted the items in a remote database through a RESTful API back-end. Results showed that CPDTs can significantly improve the developer experience by having a single, shared source code that is less complex and requires fewer lines of code, in contrast to the per-platform sets of source code as are required in the native approach. The results also demonstrated that the PWA app provided the best overall developer experience, followed by Flutter and React Native. Results for the native apps — native Android and native iOS — were not as good as CPDTs, with native Android having the worst results. There was a very small number of peer-reviewed lines of previous work in the developer experience area, and their — mostly qualitative — reported results were in line with our findings.

7.3 Contributions

7.3.1 Scientific contributions

This study was one of the first to identify trade-offs of adopting contemporary CPDTs in the building of data-driven apps in comparison to the baseline native approach on two mobile platforms (Android and iOS) in three distinct areas: responsiveness and convenience of use, resource use and performance, and developer experience. This thesis makes four central scientific contributions:

- Conducting the first quantitative study investigating the trade-offs of modern CPDTs from the developer experience perspective, and providing empirical evidence of such trade-offs. This component of the study introduced metrics and tools to measure developer experience-related parameters. It achieved greater cross-platform comparability through the use of a method in which a real-world data-driven app for evaluation was designed, and then systematically implemented across multiple development tools.

- Serving as one of the first quantitative studies that provide insight on the resource use and performance overhead imposed by modern CPDTs. This section of the study demonstrated how to measure resource use and performance on the Android and iOS platforms between apps built using different development methods.

- Identification of CPDT trade-offs with respect to responsiveness and convenience of use. This component of the study also demonstrated methods by which to measure metrics such as start-time and navigation time for cross-platform apps.

- Description of architectural characteristics of prevalent contemporary CPDTs, particularly those germane to trade-offs involving responsiveness, convenience of use, resource use, performance, and developer experience.

Engineering contributions

There are three central engineering contributions presented in this thesis:

- Implementing and open-sourcing multi-page data-driven apps using five different development tools: native Android (Java), native iOS (Swift), Flutter (Dart), React Native (JavaScript) and PWA (JavaScript). The apps were built based on a set of design rules and complemented by a server-side application implemented in Node.js. All the apps were open-sourced and published on a public repository.

- Implementation of apps featuring three resource-intensive tests — CPU-intensive, memory-intensive and network-intensive — for five different development tools: native Android (Java), native iOS (Swift), Flutter (Dart), React Native (JavaScript) and PWA (JavaScript). These apps were designed and implemented to push the low-end and high-end mobile devices' targeted resources to their limit. All the apps were open-sourced and published on a public repository.

- Implementation of a custom measuring tool that can count source lines of code specifically optimized for the aforementioned five development methods.

7.4 Limitations

Limitations in evaluating responsiveness and convenience of use

- The PWA app's launch time on the iOS platform (specifically) was missing from the comparison since there was no tooling available (at the time of writing this thesis) to measure launch time for a PWA app on iOS.

- The native Android app navigation relied upon recreating a top-level application component — Activity — when navigating to a new page. It should be acknowledged that there were other ways to implement

navigation on native Android that were avoided due to time constraints. This may have led to different results — and possibly faster navigation time — for the native Android app.

Limitations in evaluating resource use and performance

- CPU and memory usage of the PWA app was measured by different tools than those used for the other apps on the iOS platform. While CPU and memory for the native, React Native and Flutter apps were measured on iOS by the Xcode Instrument tool, reflecting PWA's central use of the browser, the PWA app measurements were conducted remotely in the Safari browser's web-inspector.
- The PWA app's measurements of total data transferred in the network-intensive test were conducted differently than for the other apps on the iOS and Android platforms. The Android Studio Profiler and Instrument tool in Xcode were used for the total data transferred measurements of the native, React Native and Flutter apps on the Android and iOS, respectively, whilst the corresponding PWA app measurements were conducted by browser remote debugging capabilities (via Chrome on Android and Safari on iOS).
- There were several cases in which resource-intensive tests failed to finish. It was one of the trade-offs associated with adjusting the test parameters to push the devices' targeted resource usage to their limit. Failures were mostly due to the limited resources in the low-end devices; however, in one instance in a network-intensive test, the HTTP client library that was used in the Flutter app was the source of failure due to excessive resource use. Due to time constraints, we did not try the Flutter app's network-intensive test with another HTTP client library.

Limitations in evaluating developer experience

- The researcher had no prior experience in developing iOS native and Flutter apps. A set of rules for the development of the data-driven apps were laid out to minimize the implementation disparity between the apps.
- The apps that were built for the developer experience evaluation covered some of the basic functionalities of a data-driven app. The generalizability of the evaluation results could have improved if the apps had more features, for example, components associated with authentication and data ownership.
- The cyclomatic complexity metric was measured by three different code analysis tools. The Android native, PWA and React Native apps were measured by SonarQube, while the Flutter app was measured using Dart Code Metrics and the native iOS app was measured using Lizard.
- Given the differences between the development methods' programming languages, syntax and programming paradigms, a custom measurement tool was built to measure the source lines of code of the developed apps by taking those differences into effect. However, a more accurate measurement based on the logical lines of code could have further improved the generalizability and accuracy of the results.

Limitations in statistical tests In order to rank the development methods and further evaluate the effect of device class, this work made heavy use of the non-parametric one-way Mann-Whitney U test. Use of this test freed comparisons from assuming that the underlying measurements followed normal distributions when ranking within a category, but the analysis did not support examination of interaction effects between multiple factors. Examination of such interaction effects would support insight into the combined effects of factors (e.g, development method, platform and device class) on the dependent variable (e.g, launch time, navigation time, execution time and etc), and may benefit from use of additional statistical tools.

7.5 Future work

There are several potential research directions that can help expand this work, enhance its contributions and overcome some of the mentioned limitations. Hence, we propose the following directions and improvements for future work.

The primary potential direction to extend this work could be to enhance the developer experience evaluation as follows:

- Build time measurements can be extended beyond debug build time to include incremental build time for development as well as build time for production.
- Studying the trade-offs associated with the distribution method of the apps — especially those that are distributed via app stores, as compared to web-based apps that are distributed by sharing a URL and can be run in the browsers.
- Extending the software metrics to measure programming complexity, development effort and proxies for software quality, amongst other things. This could involve building custom measuring tools.
- Enhancing the data-driven apps in the developer experience chapter to have more features, such as authentication and user ownership of data, amongst other things, to improve finding generalizability.

Other future work could include:

- The launch time and resource usage metrics were measured by different tools between the PWA app and the other apps that run natively on the iOS platform. Unifying the measurement methods by measuring them all in the same tool can improve the generalizability of the results. This can be addressed in future studies — especially if Apple were to extend its iOS support for the measurement of PWA apps.
- The tasks considered in resource use and performance evaluation can be extended to include resource-intensive tests that target, amongst other things, file systems and databases.

- A branch of this work could investigate the trade-offs associated with PWA apps that are built by JavaScript frameworks such as React, Angular and Vue with respect to the baseline version that is built by the core technologies: HTML, CSS and JavaScript but outside of a JavaScript framework.
- Further work is needed to better understand the implications of the responsiveness and convenience of use as well as performance results by investigating these areas from the standpoint of user experience and Human-Computer Interaction. Such a study can use the data-driven apps that were built for the developer experience chapter (Chapter 6) as a starting point and aim to identify the trade-offs between these apps through the users' interaction and perception of the metrics.

References

[1] Statista. Number of smartphone users from 2016 to 2021. https://www.statista.com/statistics/330695/number-of-smartphone-users-worldwide/. Last accessed on 2021-09-08.

[2] Statista. Number of mobile app downloads worldwide from 2016 to 2020. https://www.statista.com/statistics/271644/worldwide-free-and-paid-mobile-app-store-downloads/. Last accessed on 2021-09-08.

[3] Statista. Mobile operating systems' market share worldwide from january 2012 to june 2021. https://www.statista.com/statistics/272698/global-market-share-held-by-mobile-operating-systems-since-2009. Last accessed on 2021-09-08.

[4] Appfigures. Native vs. non-native app development. https://appfigures.com/resources/insights/ios-developers-ship-less-apps-for-first-time. Last accessed on 2022-03-12.

[5] Luis Corral, Alberto Sillitti, and Giancarlo Succi. Mobile multiplatform development: An experiment for performance analysis. *Procedia Computer Science*, 10:736–743, 2012.

[6] Isabelle Dalmasso, Soumya Kanti Datta, Christian Bonnet, and Navid Nikaein. Survey, comparison and evaluation of cross platform mobile application development tools. In *2013 9th International Wireless Communications and Mobile Computing Conference (IWCMC)*, pages 323–328. IEEE, 2013.

[7] Esteban Angulo and Xavier Ferre. A case study on cross-platform development frameworks for mobile applications and ux. In *Proceedings of the XV International Conference on Human Computer Interaction*, pages 1–8, 2014.

[8] Michiel Willocx, Jan Vossaert, and Vincent Naessens. A quantitative assessment of performance in mobile app development tools. In *2015 IEEE International Conference on Mobile Services*, pages 454–461. IEEE, 2015.

[9] Tim Majchrzak and Tor-Morten Grønli. Comprehensive analysis of innovative cross-platform app development frameworks. In *Proceedings of the 50th Hawaii International Conference on System Sciences*, 2017.

[10] Arvind Hudli, Shrinidhi Hudli, and Raghu Hudli. An evaluation framework for selection of mobile app development platform. In *Proceedings of the 3rd International Workshop on Mobile Development Lifecycle*, pages 13–16, 2015.

[11] Pavel Smutnỳ. Mobile development tools and cross-platform solutions. In *Proceedings of the 13th International Carpathian Control Conference (ICCC)*, pages 653–656. IEEE, 2012.

[12] Wafaa S El-Kassas, Bassem A Abdullah, Ahmed H Yousef, and Ayman M Wahba. Taxonomy of cross-platform mobile applications development approaches. *Ain Shams Engineering Journal*, 8(2):163–190, 2017.

[13] Ville Ahti, Sami Hyrynsalmi, and Olli Nevalainen. An evaluation framework for cross-platform mobile app development tools: A case analysis of adobe phonegap framework. In *Proceedings of the 17th International Conference on Computer Systems and Technologies 2016*, pages 41–48, 2016.

[14] Spyros Xanthopoulos and Stelios Xinogalos. A comparative analysis of cross-platform development approaches for mobile applications. In *Proceedings of the 6th Balkan Conference in Informatics*, pages 213–220, 2013.

[15] Divya Sambasivan, Nikita John, Shruthi Udayakumar, and Rajat Gupta. Generic framework for mobile application development. In *2011 Second Asian Himalayas International Conference on Internet (AH-ICI)*, pages 1–5. IEEE, 2011.

[16] Christoph Rieger and Tim A Majchrzak. A taxonomy for app-enabled devices: mastering the mobile device jungle. In *International Conference on Web Information Systems and Technologies*, pages 202–220. Springer, 2017.

[17] Andre Charland and Brian LeRoux. Mobile application development: Web vs. native: Web apps are cheaper to develop and deploy than native apps, but can they match the native user experience? *Queue*, 9(4):20–28, 2011.

[18] Mona Erfani Joorabchi, Ali Mesbah, and Philippe Kruchten. Real challenges in mobile app development. In *2013 ACM / IEEE International Symposium on Empirical Software Engineering and Measurement*, pages 15–24, 2013.

[19] CP Rahul Raj and Seshu Babu Tolety. A study on approaches to build cross-platform mobile applications and criteria to select appropriate approach. In *2012 Annual IEEE India Conference (INDICON)*, pages 625–629. IEEE, 2012.

[20] Matteo Ciman and Ombretta Gaggi. An empirical analysis of energy consumption of cross-platform frameworks for mobile development. *Pervasive and Mobile Computing*, 39:214–230, 2017.

[21] Andreas Biørn-Hansen, Tim A Majchrzak, and Tor-Morten Grønli. Progressive web apps: The possible web-native unifier for mobile development. In *International Conference on Web Information Systems and Technologies*, volume 2, pages 344–351. SciTePress, 2017.

[22] Wikipedia. Web application. `https://en.wikipedia.org/wiki/Web_application`. Last accessed on 2021-09-11.

[23] Wikipedia. Responsive web design. `https://en.wikipedia.org/wiki/Responsive_web_design`. Last accessed on 2021-09-11.

[24] Facebook. React.js. `https://reactjs.org/`. Last accessed on 2021-09-16.

[25] Angular. The modern web developer's platform. `https://angular.io/`. Last accessed on 2021-09-16.

[26] Vue. The progressive javascript framework. `https://vuejs.org/`. Last accessed on 2021-09-16.

[27] Ionic. Ionic framework. `https://ionicframework.com/`. Last accessed on 2021-09-16.

[28] Michiel Willocx, Jan Vossaert, and Vincent Naessens. Comparing performance parameters of mobile app development strategies. In *Proceedings of the International Conference on Mobile Software Engineering and Systems*, pages 38–47, 2016.

[29] Mounaim Latif, Younes Lakhrissi, Najia Es-Sbai, et al. Cross platform approach for mobile application development: A survey. In *2016 International Conference on Information Technology for Organizations Development (IT4OD)*, pages 1–5. IEEE, 2016.

[30] Ivano Malavolta, Stefano Ruberto, Tommaso Soru, and Valerio Terragni. End users' perception of hybrid mobile apps in the google play store. In *2015 IEEE International Conference on Mobile Services*, pages 25–32. IEEE, 2015.

[31] Apache Cordova. `https://cordova.apache.org/`. Last accessed on 2021-09-11.

[32] Sunny Dhillon and Qusay H Mahmoud. An evaluation framework for cross-platform mobile application development tools. *Software: Practice and Experience*, 45(10):1331–1357, 2015.

[33] Facebook. React native. `https://reactnative.dev/`. Last accessed on 2021-09-12.

[34] Appcelerator. Titanium mobile. `https://titaniumsdk.com/`. Last accessed on 2021-09-12.

[35] NativeScript. https://nativescript.org/. Last accessed on 2021-09-12.

[36] Michiel Willocx, Jan Vossaert, and Vincent Naessens. A quantitative assessment of performance in mobile app development tools. In *2015 IEEE International Conference on Mobile Services*, pages 454–461. IEEE, 2015.

[37] Microsoft. Xamarin. https://dotnet.microsoft.com/apps/xamarin. Last accessed on 2021-09-12.

[38] Google. Flutter. https://flutter.dev/. Last accessed on 2021-09-12.

[39] Google. Pwa history. https://infrequently.org/2015/06/progressive-apps-escaping-tabs-without-losing-our-soul/. Last accessed on 2021-09-12.

[40] Mozilla. Pwa introduction. https://developer.mozilla.org/en-US/docs/Web/Progressive_web_apps/Introduction. Last accessed on 2021-09-12.

[41] Stackoverflow. Developer survey 2021. https://insights.stackoverflow.com/survey/2021#web-frameworks. Last accessed on 2021-09-23.

[42] Wikipedia. Declarative programming. https://en.wikipedia.org/wiki/Declarative_programming. Last accessed on 2021-12-29.

[43] Facebook. React reconciliation. https://reactjs.org/docs/reconciliation.html. Last accessed on 2021-09-22.

[44] Facebook. React virtual dom. https://reactjs.org/docs/faq-internals.html. Last accessed on 2021-09-23.

[45] Facebook. Jsx introduction. https://reactjs.org/docs/introducing-jsx.html. Last accessed on 2021-09-25.

[46] Gustavo Hartmann, Geoff Stead, and Asi DeGani. Cross-platform mobile development. *Mobile Learning Environment, Cambridge*, 16(9):158–171, 2011.

[47] Henning Heitkötter, Sebastian Hanschke, and Tim A Majchrzak. Evaluating cross-platform development approaches for mobile applications. In *International Conference on Web Information Systems and Technologies*, pages 120–138. Springer, 2012.

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

[49] Sencha. Cross-platform mobile development tool. https://www.sencha.com/products/touch. Last accessed on 2021-09-12.

[50] Andreas Biørn-Hansen, Tor-Morten Grønli, and Gheorghita Ghinea. A survey and taxonomy of core concepts and research challenges in cross-platform mobile development. *ACM Computing Surveys (CSUR)*, 51(5):1–34, 2018.

[51] Tommi Mikkonen and Antero Taivalsaari. Apps vs. open web: The battle of the decade. In *Proceedings of the 2nd Workshop on Software Engineering for Mobile Application Development*, pages 22–26. MSE Santa Monica, CA, 2011.

[52] Techcrunch. What happened to the facebook mobile web app. https://techcrunch.com/2012/09/11/mark-zuckerberg-our-biggest-mistake-with-mobile-was-betting-too-much-on-html5/. Last accessed on 2021-09-15.

[53] Github. Property cross repo. https://github.com/tastejs/PropertyCross. Last accessed on 2021-09-15.

[54] Property Cross. An opensource project that helps developers select a cross platform mobile framework. http://propertycross.com/. Last accessed on 2021-09-15.

[55] Michiel Willocx, Jan Vossaert, and Vincent Naessens. Comparing performance parameters of mobile app development strategies. In *Proceedings of the International Conference on Mobile Software Engineering and Systems*, pages 38–47, 2016.

[56] Jonathan Kvist and Pontus Mathiasson. Progressive web apps and other mobile developing techniques: a comparison, 2019.

[57] Google. Lighthouse. https://developers.google.com/web/tools/lighthouse. Last accessed on 2021-09-15.

[58] Andreas Biørn-Hansen, Christoph Rieger, Tor-Morten Grønli, Tim A Majchrzak, and Gheorghita Ghinea. An empirical investigation of performance overhead in cross-platform mobile development frameworks. *Empirical Software Engineering*, 25(4):2997–3040, 2020.

[59] Andreas Biørn-Hansen, Tor-Morten Grønli, Gheorghita Ghinea, and Sahel Alouneh. An empirical study of cross-platform mobile development in industry. *Wireless Communications and Mobile Computing*, 2019, 2019.

[60] Stackoverflow. Other frameworks and libraries used (2021 survey). https://insights.stackoverflow.com/survey/2021#most-popular-technologies-misc-tech. Last accessed on 2021-09-20.

[61] Statista. Cross-platform mobile frameworks used by software developers worldwide from 2019 to 2021. https://www.statista.com/statistics/869224/worldwide-software-developer-working-hours/. Last accessed on 2021-09-20.

[62] Google. Flutter's first release. https://github.com/flutter/flutter/releases/tag/v0.0.6. Last accessed on 2021-09-24.

[63] Google. Flutter architectural overview. https://flutter.dev/docs/resources/architectural-overview. Last accessed on 2021-09-23.

[64] Google. Dart language concepts. https://dart.dev/guides/language/language-tour#important-concepts. Last accessed on 2021-09-23.

[65] Google. Build and release flutter apps on android. https://docs.flutter.dev/deployment/android. Last accessed on 2022-05-25.

[66] Google. Dart overview. https://dart.dev/overview. Last accessed on 2022-05-15.

[67] Google. Flutter architectural layers. https://flutter.dev/docs/resources/architectural-overview#architectural-layers. Last accessed on 2021-09-24.

[68] Google. Skia: The 2d graphics library. https://skia.org/. Last accessed on 2021-09-20.

[69] Google. Flutter reactive ui. https://flutter.dev/docs/resources/architectural-overview#reactive-user-interfaces. Last accessed on 2021-09-24.

[70] Facebook. React native: intro. https://reactnative.dev/docs/intro-react-native-components. Last accessed on 2021-09-22.

[71] Mozilla. Dom intro. https://developer.mozilla.org/en-US/docs/Web/API/Document_Object_Model/Introduction. Last accessed on 2021-09-22.

[72] OReilly. Learning react native. https://www.oreilly.com/library/view/learning-react-native/9781491929049/ch01.html. Last accessed on 2021-09-22.

[73] Uptech. How we build react native apps. https://www.uptech.team/blog/save-development-time-build-react-native-app. Last accessed on 2021-09-22.

[74] Parashuram. React natives new architecture glossary. http://blog.nparashuram.com/2019/01/react-natives-new-architecture-glossary.html. Last accessed on 2021-09-25.

[75] W3Schools. What is json. https://www.w3schools.com/whatis/whatis_json.asp. Last accessed on 2021-09-22.

[76] Facebook. Flexbox. https://reactnative.dev/docs/flexbox. Last accessed on 2021-09-22.

[77] Facebook. Yoga layout engine. https://github.com/facebook/yoga. Last accessed on 2021-09-25.

[78] Tadeu Zagallo. An in-depth look into react native's core. https://tadeuzagallo.com/blog/react-native-bridge/. Last accessed on 2021-09-25.

[79] Facebook. React native's new architecture at react conf 2018. https://www.youtube.com/watch?v=UcqRXTriUVI&t=1s&ab_channel=ReactConf. Last accessed on 2021-09-22.

[80] Jun Kaneko. Understanding react native architecture. https://dev.to/goodpic/understanding-react-native-architecture-22hh. Last accessed on 2021-09-25.

[81] Datacdamia. What is react tree. https://datacadamia.com/web/javascript/react/tree. Last accessed on 2022-04-02.

[82] Google. What are pwas. https://web.dev/what-are-pwas/. Last accessed on 2021-09-22.

[83] Tim A Majchrzak, Andreas Biørn-Hansen, and Tor-Morten Grønli. Progressive web apps: the definite approach to cross-platform development? 2018.

[84] Google. Service worker fundamentals. https://developers.google.com/web/fundamentals/primers/service-workers. Last accessed on 2021-09-23.

[85] Addy Osmani and Matt Gaunt. Instant loading web apps with an application shell architecture. *Google Developers Documentation.[Online]. Available: developers. google. com/web/updates/2015/11/appshell*, 2017.

[86] Google. Service worker requirement. https://developers.google.com/web/fundamentals/primers/service-workers#you_need_https. Last accessed on 2021-09-23.

[87] Google. App manifest. https://web.dev/add-manifest/. Last accessed on 2021-09-23.

[88] Mozilla. App manifest. https://developer.mozilla.org/en-US/docs/Web/Manifest. Last accessed on 2021-09-23.

[89] Google. Pwa architecture. https://developers.google.com/web/ilt/pwa/introduction-to-progressive-web-app-architectures#architectural_styles_and_patterns. Last accessed on 2021-09-23.

[90] What web can do today website. https://whatwebcando.today/. Last accessed on 2022-03-21.

[91] Giulia de Andrade Cardieri and Luciana Martinez Zaina. Analyzing user experience in mobile web, native and progressive web applications: A user and hci specialist perspectives. In *Proceedings of the 17th Brazilian Symposium on Human Factors in Computing Systems*, pages 1–11, 2018.

[92] Google. Android startup time - understanding internals. https://developer.android.com/topic/performance/vitals/launch-time#internals. Last accessed on 2021-09-29.

[93] Google. Android startup time - cold start. https://developer.android.com/topic/performance/vitals/launch-time#cold. Last accessed on 2021-09-29.

[94] Google. Android logcat. https://developer.android.com/studio/command-line/logcat. Last accessed on 2021-09-30.

[95] Apple. Reducing your app's launch time. https://developer.apple.com/documentation/xcode/reducing-your-app-s-launch-time. Last accessed on 2021-09-30.

[96] Google. Webapks. https://developers.google.com/web/fundamentals/integration/webapks. Last accessed on 2021-10-07.

[97] Google. Android activity intro. https://developer.android.com/guide/components/activities/intro-activities. Last accessed on 2021-10-07.

[98] Apple. Uikit. https://developer.apple.com/documentation/uikit. Last accessed on 2022-01-17.

[99] Apple. Ios uiviewcontroller. https://developer.apple.com/documentation/uikit/uiviewcontroller. Last accessed on 2021-10-07.

[100] React Navigation. Routing and navigation for expo and react native apps. https://reactnavigation.org/. Last accessed on 2021-10-07.

[101] Facebook. Create react app. https://reactjs.org/docs/create-a-new-react-app.html. Last accessed on 2021-10-09.

[102] React Router. A routing library for the react javascript library. https://reactrouter.com/web/guides/quick-start. Last accessed on 2021-10-07.

[103] Google. Work box. https://developers.google.com/web/tools/workbox. Last accessed on 2021-10-07.

[104] Google. Firebase. https://firebase.google.com/?gclid=CjwKCAjwtfqKBhBoEiwAZuesiNkxLS5Rg0dDCPQ6zboRRK-Xg 9rUj0kVRoC2eMQAvD_BwE&gclsrc=aw.ds. Last accessed on 2021-10-07.

[105] Andrey Akinshin. Statistical approaches for performance analysis. https://aakinshin.net/posts/statistics-for-performance/. Last accessed on 2022-05-14.

[106] Data science blog. Mann-whitney u test (wilcoxon rank sum test) in python. https://www.reneshbedre.com/blog/mann-whitney-u-test.html. Last accessed on 2022-05-14.

[107] R Project. R software environment. https://www.r-project.org/. Last accessed on 2022-01-16.

[108] Apple. Uikit. https://developer.apple.com/documentation/uikit/. Last accessed on 2021-10-20.

[109] Google. Android startup time. https://developer.android.com/topic/performance/vitals/launch-time. Last accessed on 2021-09-29.

[110] React Navigation. React navigation native stack. https://reactnavigation.org/docs/native-stack-navigator. Last accessed on 2021-10-20.

[111] Google. Materialpageroute. https://master-api.flutter.dev/flutter/material/MaterialPageRoute-class.html. Last accessed on 2021-10-21.

[112] Apple. Javascriptcore. https://developer.apple.com/documentation/javascriptcore. Last accessed on 2021-10-21.

[113] Google. Android vitals - excessive startup times. https://developer.android.com/topic/performance/vitals/launch-time#av. Last accessed on 2021-10-21.

[114] Jakob Nielsen. *Usability engineering*. Morgan Kaufmann, 1994.

[115] Wikipedia. Computer memory. https://en.wikipedia.org/wiki/Computer_memory#cite_note-:1-1. Last accessed on 2021-11-15.

[116] Wikipedia. Memory footprint. https://en.wikipedia.org/wiki/Memory_footprint. Last accessed on 2021-11-15.

[117] Ubuntu. Top command. http://manpages.ubuntu.com/manpages/precise/en/man1/top.1.html. Last accessed on 2021-11-20.

[118] Wikipedia. Heterogeneous computing. https://en.wikipedia.org/wiki/Heterogeneous_computing. Last accessed on 2022-05-13.

[119] Webkit. Safari memory debugging. https://webkit.org/blog/6425/memory-debugging-with-web-inspector/. Last accessed on 2021-11-15.

[120] Apple. Interface builder. https://developer.apple.com/xcode/interface-builder/. Last accessed on 2022-01-15.

[121] Oracle. Restful api. https://docs.oracle.com/javaee/6/tutorial/doc/gijqy.html. Last accessed on 2021-11-19.

[122] NestJS. A progressive node.js framework for building efficient, reliable and scalable server-side applications. https://nestjs.com/. Last accessed on 2021-11-20.

[123] Wkipedia. Prime numbers. https://en.wikipedia.org/wiki/Prime_number. Last accessed on 2021-11-15.

[124] Brian R Gaeke, Parry Husbands, Xiaoye S Li, Leonid Oliker, Katherine A Yelick, and Rupak Biswas. Memory-intensive benchmarks: Iram vs. cache-based machines. In *Proceedings 16th International Parallel and Distributed Processing Symposium*, pages 7–pp. IEEE, 2002.

[125] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. *Introduction to Algorithms, Third Edition*. The MIT Press, 3rd edition, 2009.

[126] Axios. An http clinet library. https://github.com/axios/axios. Last accessed on 2021-11-19.

[127] Square. Retrofit http clinet. https://square.github.io/retrofit/. Last accessed on 2021-11-19.

[128] Dart Package Manager. http client for dart. https://pub.dev/packages/http. Last accessed on 2021-11-19.

[129] Baeldung. Java bytecode. https://www.baeldung.com/java-class-view-bytecode. Last accessed on 2022-02-06.

[130] Android. Art. https://source.android.com/devices/tech/dalvik. Last accessed on 2021-11-21.

[131] Telerik. Javascript engines. https://web.archive.org/web/20181208123231/http://developer.telerik.com/featured/a-guide-to-javascript-engines-for-idiots/. Last accessed on 2021-11-21.

[132] Facebook. React native js environment. https://reactnative.dev/docs/javascript-environment. Last accessed on 2021-11-21.

[133] Nano Review. Snapdragon 865 vs 835. https://nanoreview.net/en/soc-compare/qualcomm-snapdragon-865-vs-qualcomm-snapdragon-835. Last accessed on 2021-11-23.

[134] Nano Review. Apple a14 vs a11. https://nanoreview.net/en/soc-compare/apple-a14-bionic-vs-apple-a11-bionic. Last accessed on 2021-11-23.

[135] IBM. Memory utilization. https://www.ibm.com/docs/en/informix-servers/14.10?topic=performance-memory-utilization. Last accessed on 2021-11-23.

[136] Flutter. Fetch data documentation. https://docs.flutter.dev/cookbook/networking/fetch-data. Last accessed on 2021-11-24.

[137] Wikipedia. HTTP compression. https://en.wikipedia.org/wiki/HTTP_compression. Last accessed on 2021-11-23.

[138] Wikipedia. Brotli. https://en.wikipedia.org/wiki/Brotli#cite_note-ietf-2. Last accessed on 2021-11-23.

[139] Wikipedia. Source lines of code. https://en.wikipedia.org/wiki/Source_lines_of_code. Last accessed on 2022-01-1.

[140] Arthur Henry Watson, Dolores R Wallace, and Thomas J McCabe. *Structured testing: A testing methodology using the cyclomatic complexity metric*, volume 500. US Department of Commerce, Technology Administration, National Institute of ..., 1996.

[141] Wikipedia. Cyclomatic complexity. https://en.wikipedia.org/wiki/Cyclomatic_complexity. Last accessed on 2022-01-5.

[142] T.J. McCabe. A complexity measure. *IEEE Transactions on Software Engineering*, SE-2(4):308–320, 1976.

[143] N.E. Fenton and M. Neil. A critique of software defect prediction models. *IEEE Transactions on Software Engineering*, 25(5):675–689, 1999.

[144] Dart Code Metric. Cyclomatic complexity. https://dartcodemetrics.dev/docs/metrics/cyclomatic-complexity. Last accessed on 2022-01-5.

[145] Techopedia. Build definition. https://www.techopedia.com/definition/3759/build. Last accessed on 2022-01-5.

[146] industriallogic. Build time impact. https://www.industriallogic.com/blog/how-long-is-too-long-for-a-build/. Last accessed on 2022-01-5.

[147] Wikipedia. Grep command. https://en.wikipedia.org/wiki/Grep. Last accessed on 2022-01-7.

[148] SonarQube. Static code analysis. https://www.sonarqube.org/downloads/?gads_campaign=North-America-SonarQube&gads_ad_group=SonarQube&gads_keyword=sonarqube&gclid=Cj0KCQiAieWOBhCYARIsANcOw0xfUp2U3DBIlUveiqr9e1_2LK7WT5a7DB9nwN3HT9LsCkJMMdjEg9IaAsvhEALw_wcB. Last accessed on 2022-01-8.

[149] Dart Code Metrics. Code metrics. https://dartcodemetrics.dev/. Last accessed on 2022-01-8.

[150] Lizard. Code analysis. https://github.com/terryyin/lizard. Last accessed on 2022-01-8.

[151] Jetbrains. Event log. https://www.jetbrains.com/help/idea/2020.3/event-log-tool-window.html. Last accessed on 2022-01-9.

[152] Nodejs. A javascript runtime. https://nodejs.org/en/. Last accessed on 2022-01-9.

[153] NPM. Node package manager. https://www.npmjs.com/. Last accessed on 2022-01-9.

[154] Paypal. Gnomon. https://github.com/paypal/gnomon. Last accessed on 2022-01-9.

[155] Techopedia. Three-tier architecture thechopedia. https://www.techopedia.com/definition/24649/three-tier-architecture. Last accessed on 2022-01-16.

[156] IBM. Three-tier architecture. https://www.ibm.com/cloud/learn/three-tier-architecture. Last accessed on 2022-01-16.

[157] Google. Gson. https://github.com/google/gson. Last accessed on 2022-01-15.

[158] MUI. Material ui for react. https://mui.com/. Last accessed on 2022-01-15.

[159] Wikipedia. Imperative programming. https://en.wikipedia.org/wiki/Imperative_programming. Last accessed on 2021-12-29.

[160] Free Online Dictionary of Computing. Imperative programming. https://foldoc.org/imperative+languages. Last accessed on 2021-12-29.

[161] Free Online Dictionary of Computing. Declarative programming. https://foldoc.org/declarative%20language. Last accessed on 2021-12-29.

[162] Mozilla. Json and javascript. https://developer.mozilla.org/en-US/docs/Learn/JavaScript/Objects/JSON. Last accessed on 2021-12-29.

[163] Apple. Ios build process. https://developer.apple.com/videos/play/wwdc2018/415/. Last accessed on 2021-12-30.

[164] Google. Android build. https://developer.android.com/studio/build. Last accessed on 2021-12-30.

[165] Babel. A javascript compiler. https://babeljs.io/docs/en/index.html. Last accessed on 2021-12-30.

[166] Webpack. A static module bundler for modern javascript applications. https://webpack.js.org/concepts/. Last accessed on 2021-12-30.

Appendix A

Statistical Analysis Code Listings of Responsiveness and Convenience of Use Experiments

Listing A.1: Statistical analysis code of the launch time experiment (in R)

```
library(dplyr)

rawData = read.table("/Users/iman/Desktop/results-responsiveness/launch-time-raw-data.csv
    ↪ ", sep = ",", header = TRUE)
rawData$method <- as.factor(rawData$method)

# ----------------------------------------------------------------------------Android
    ↪ -----------------------------------------#
# ------------------------launchTime-android-low-end-------------------------------#
dataAndroidLow <- filter(rawData, method == "rnAndroidLow" |
                        method == "flutterAndroidLow" |
                        method == "pwaAndroidLow" |
                        method == "nativeAndroidLow")
group_by(dataAndroidLow,method) %>%
  summarise(
    count = n(),
    median = median(ct, na.rm = T),
    std = sd(ct, na.rm = T))
# ranks: rnAndroidLow < flutterAndroidLow < nativeAndroidLow < pwaAndroidLow

rnAndroidLow <- filter(dataAndroidLow, method == "rnAndroidLow")
flutterAndroidLow <- filter(dataAndroidLow, method == "flutterAndroidLow")
nativeAndroidLow <- filter(dataAndroidLow, method == "nativeAndroidLow")
pwaAndroidLow <- filter(dataAndroidLow, method == "pwaAndroidLow")


## rnAndroidLow < flutterAndroidLow
rnAndroidLow_flutterAndroidLow_result <- wilcox.test(rnAndroidLow$ct, flutterAndroidLow$
    ↪ ct, alt="less")
rnAndroidLow_flutterAndroidLow_result

## flutterAndroidLow < nativeAndroidLow
flutterAndroidLow_nativeAndroidLow_result <- wilcox.test(flutterAndroidLow$ct,
    ↪ nativeAndroidLow$ct, alt="less")
flutterAndroidLow_nativeAndroidLow_result

## nativeAndroidLow < pwaAndroidLow
nativeAndroidLow_pwaAndroidLow_result <- wilcox.test(nativeAndroidLow$ct, pwaAndroidLow$
    ↪ ct, alt="less")
nativeAndroidLow_pwaAndroidLow_result

# ------------------------launchTime-android-high-end-------------------------------#
dataAndroidHigh <- filter(rawData, method == "rnAndroidHigh" |
```

```
                            method == "flutterAndroidHigh" |
                            method == "pwaAndroidHigh" |
                            method == "nativeAndroidHigh")
group_by(dataAndroidHigh,method) %>%
  summarise(
    count = n(),
    median = median(ct, na.rm = T),
    std = sd(ct, na.rm = T))
# ranks: nativeAndroidHigh < flutterAndroidHigh < rnAndroidHigh < pwaAndroidHigh

nativeAndroidHigh <- filter(dataAndroidHigh, method == "nativeAndroidHigh")
flutterAndroidHigh <- filter(dataAndroidHigh, method == "flutterAndroidHigh")
rnAndroidHigh <- filter(dataAndroidHigh, method == "rnAndroidHigh")
pwaAndroidHigh <- filter(dataAndroidHigh, method == "pwaAndroidHigh")

## nativeAndroidHigh < flutterAndroidHigh
nativeAndroidHigh_flutterAndroidHigh_result <- wilcox.test(nativeAndroidHigh$ct,
    ↪ flutterAndroidHigh$ct, alt="less")
nativeAndroidHigh_flutterAndroidHigh_result

## flutterAndroidHigh < rnAndroidHigh
flutterAndroidHigh_rnAndroidHigh_result <- wilcox.test(flutterAndroidHigh$ct,
    ↪ rnAndroidHigh$ct, alt="less")
flutterAndroidHigh_rnAndroidHigh_result

## rnAndroidHigh < pwaAndroidHigh
rnAndroidHigh_pwaAndroidHigh_result <- wilcox.test(rnAndroidHigh$ct, pwaAndroidHigh$ct,
    ↪ alt="less")
rnAndroidHigh_pwaAndroidHigh_result

# -------------------------launchTime-android-low-end vs high-end
    ↪ ------------------------------#
## nativeAndroidHigh < nativeAndroidLow
nativeAndroidHigh_nativeAndroidLow_result <- wilcox.test(nativeAndroidHigh$ct,
    ↪ nativeAndroidLow$ct, alt="less")
nativeAndroidHigh_nativeAndroidLow_result

## rnAndroidHigh < rnAndroidLow
rnAndroidHigh_rnAndroidLow_result <- wilcox.test(rnAndroidHigh$ct, rnAndroidLow$ct, alt="
    ↪ less")
rnAndroidHigh_rnAndroidLow_result

## flutterAndroidHigh < flutterAndroidLow
flutterAndroidHigh_flutterAndroidLow_result <- wilcox.test(flutterAndroidHigh$ct,
    ↪ flutterAndroidLow$ct, alt="less")
flutterAndroidHigh_flutterAndroidLow_result

## pwaAndroidHigh < pwaAndroidLow
pwaAndroidHigh_pwaAndroidLow_result <- wilcox.test(pwaAndroidHigh$ct, pwaAndroidLow$ct,
    ↪ alt="less")
pwaAndroidHigh_pwaAndroidLow_result
```

```
# --------------------------------------------------------------------------------iOS
    ↪ ------------------------------------------#
# -------------------------launchTime-ios-low-end--------------------------------#
dataIosLow <- filter(rawData, method == "rnIosLow" |
                     method == "flutterIosLow" |
                     method == "nativeIosLow")
group_by(dataIosLow,method) %>%
  summarise(
    count = n(),
    median = median(ct, na.rm = T),
    std = sd(ct, na.rm = T))
# ranks: nativeIosLow < rnIosLow < flutterIosLow

nativeIosLow <- filter(dataIosLow, method == "nativeIosLow")
rnIosLow <- filter(dataIosLow, method == "rnIosLow")
flutterIosLow <- filter(dataIosLow, method == "flutterIosLow")

## nativeIosLow < rnIosLow
nativeIosLow_rnIosLow_result <- wilcox.test(nativeIosLow$ct, rnIosLow$ct, alt="less")
nativeIosLow_rnIosLow_result

## rnIosLow < flutterIosLow
rnIosLow_flutterIosLow_result <- wilcox.test(rnIosLow$ct, flutterIosLow$ct, alt="less")
rnIosLow_flutterIosLow_result

# -------------------------launchTime-ios-high-end-------------------------------#
dataIosHigh <- filter(rawData, method == "rnIosHigh" |
                      method == "flutterIosHigh" |
                      method == "nativeIosHigh")
group_by(dataIosHigh,method) %>%
  summarise(
    count = n(),
    median = median(ct, na.rm = T),
    std = sd(ct, na.rm = T))
# ranks: rnIosHigh < nativeIosHigh < flutterIosHigh

rnIosHigh <- filter(dataIosHigh, method == "rnIosHigh")
nativeIosHigh <- filter(dataIosHigh, method == "nativeIosHigh")
flutterIosHigh <- filter(dataIosHigh, method == "flutterIosHigh")


## rnIosHigh < nativeIosHigh
rnIosHigh_nativeIosHigh_result <- wilcox.test(rnIosHigh$ct, nativeIosHigh$ct, alt="less")
rnIosHigh_nativeIosHigh_result

## nativeIosHigh < flutterIosHigh
nativeIosHigh_flutterIosHigh_result <- wilcox.test(nativeIosHigh$ct, flutterIosHigh$ct,
    ↪ alt="less")
nativeIosHigh_flutterIosHigh_result

# -------------------------launchTime-ios-low-end vs high-end
    ↪ ------------------------------#
## nativeIosHigh < nativeIosLow
```

```
nativeIosHigh_nativeIosLow_result <- wilcox.test(nativeIosHigh$ct, nativeIosLow$ct, alt="
    ↪ less")
nativeIosHigh_nativeIosLow_result

## rnIosHigh < rnIosLow
rnIosHigh_rnIosLow_result <- wilcox.test(rnIosHigh$ct, rnIosLow$ct, alt="less")
rnIosHigh_rnIosLow_result

## flutterIosHigh < flutterIosLow
flutterIosHigh_flutterIosLow_result <- wilcox.test(flutterIosHigh$ct, flutterIosLow$ct,
    ↪ alt="less")
flutterIosHigh_flutterIosLow_result
```

Listing A.2: Statistical analysis code of the navigation time experiment (in R)

```
library(dplyr)

rawData = read.table("/Users/iman/Desktop/results-responsiveness/navigation-time-raw-data
    ↪ .csv", sep = ",", header = TRUE)
rawData$method <- as.factor(rawData$method)

# -----------------------------------------------------------------------------Android
    ↪ -----------------------------------------#
# -------------------------navigation-android-low-end-------------------------------#
dataAndroidLow <- filter(rawData, method == "rnAndroidLow" |
                        method == "flutterAndroidLow" |
                        method == "pwaAndroidLow" |
                        method == "nativeAndroidLow")
group_by(dataAndroidLow,method) %>%
  summarise(
    count = n(),
    median = median(ct, na.rm = T),
    std = sd(ct, na.rm = T))
# ranks: flutterAndroidLow < pwaAndroidLow < rnAndroidLow < nativeAndroidLow

rnAndroidLow <- filter(dataAndroidLow, method == "rnAndroidLow")
flutterAndroidLow <- filter(dataAndroidLow, method == "flutterAndroidLow")
nativeAndroidLow <- filter(dataAndroidLow, method == "nativeAndroidLow")
pwaAndroidLow <- filter(dataAndroidLow, method == "pwaAndroidLow")


## flutterAndroidLow < pwaAndroidLow
flutterAndroidLow_pwaAndroidLow_result <- wilcox.test(flutterAndroidLow$ct, pwaAndroidLow
    ↪ $ct, alt="less")
flutterAndroidLow_pwaAndroidLow_result

## pwaAndroidLow < rnAndroidLow
pwaAndroidLow_rnAndroidLow_result <- wilcox.test(pwaAndroidLow$ct, rnAndroidLow$ct, alt="
    ↪ less")
pwaAndroidLow_rnAndroidLow_result

## rnAndroidLow < nativeAndroidLow
rnAndroidLow_nativeAndroidLow_result <- wilcox.test(rnAndroidLow$ct, nativeAndroidLow$ct,
    ↪  alt="less")
rnAndroidLow_nativeAndroidLow_result
```

```
# ------------------------navigation-android-high-end------------------------------#
dataAndroidHigh <- filter(rawData, method == "rnAndroidHigh" |
                          method == "flutterAndroidHigh" |
                          method == "pwaAndroidHigh" |
                          method == "nativeAndroidHigh")
group_by(dataAndroidHigh,method) %>%
  summarise(
    count = n(),
    median = median(ct, na.rm = T),
    std = sd(ct, na.rm = T))
# ranks: flutterAndroidHigh < rnAndroidHigh < pwaAndroidHigh < nativeAndroidHigh

nativeAndroidHigh <- filter(dataAndroidHigh, method == "nativeAndroidHigh")
flutterAndroidHigh <- filter(dataAndroidHigh, method == "flutterAndroidHigh")
rnAndroidHigh <- filter(dataAndroidHigh, method == "rnAndroidHigh")
pwaAndroidHigh <- filter(dataAndroidHigh, method == "pwaAndroidHigh")

## flutterAndroidHigh < rnAndroidHigh
flutterAndroidHigh_rnAndroidHigh_result <- wilcox.test(flutterAndroidHigh$ct,
    ↪ rnAndroidHigh$ct, alt="less")
flutterAndroidHigh_rnAndroidHigh_result

## rnAndroidHigh < pwaAndroidHigh
rnAndroidHigh_pwaAndroidHigh_result <- wilcox.test(rnAndroidHigh$ct, pwaAndroidHigh$ct,
    ↪ alt="less")
rnAndroidHigh_pwaAndroidHigh_result

## pwaAndroidHigh < nativeAndroidHigh
pwaAndroidHigh_nativeAndroidHigh_result <- wilcox.test(pwaAndroidHigh$ct,
    ↪ nativeAndroidHigh$ct, alt="less")
pwaAndroidHigh_nativeAndroidHigh_result

# ------------------------navigation-android-low-end vs high-end
    ↪ ------------------------------#
## nativeAndroidHigh < nativeAndroidLow
nativeAndroidHigh_nativeAndroidLow_result <- wilcox.test(nativeAndroidHigh$ct,
    ↪ nativeAndroidLow$ct, alt="less")
nativeAndroidHigh_nativeAndroidLow_result

## rnAndroidHigh < rnAndroidLow
rnAndroidHigh_rnAndroidLow_result <- wilcox.test(rnAndroidHigh$ct, rnAndroidLow$ct, alt="
    ↪ less")
rnAndroidHigh_rnAndroidLow_result

## flutterAndroidHigh < flutterAndroidLow
flutterAndroidHigh_flutterAndroidLow_result <- wilcox.test(flutterAndroidHigh$ct,
    ↪ flutterAndroidLow$ct, alt="less")
flutterAndroidHigh_flutterAndroidLow_result

## pwaAndroidHigh > pwaAndroidLow
pwaAndroidHigh_pwaAndroidLow_result <- wilcox.test(pwaAndroidHigh$ct, pwaAndroidLow$ct,
    ↪ alt="greater")
pwaAndroidHigh_pwaAndroidLow_result
```

```
# ------------------------------------------------------------------------------iOS
    ↪ -----------------------------------------#
# ------------------------navigation-ios-low-end------------------------------#
dataIosLow <- filter(rawData, method == "rnIosLow" |
                  method == "flutterIosLow" |
                  method == "pwaIosLow" |
                  method == "nativeIosLow")
group_by(dataIosLow,method) %>%
  summarise(
    count = n(),
    median = median(ct, na.rm = T),
    std = sd(ct, na.rm = T))
# ranks: flutterIosLow < nativeIosLow < rnIosLow < pwaIosLow

nativeIosLow <- filter(dataIosLow, method == "nativeIosLow")
rnIosLow <- filter(dataIosLow, method == "rnIosLow")
flutterIosLow <- filter(dataIosLow, method == "flutterIosLow")
pwaIosLow <- filter(dataIosLow, method == "pwaIosLow")

## flutterIosLow < nativeIosLow
flutterIosLow_nativeIosLow_result <- wilcox.test(flutterIosLow$ct, nativeIosLow$ct, alt="
    ↪ less")
flutterIosLow_nativeIosLow_result

## nativeIosLow < rnIosLow
nativeIosLow_rnIosLow_result <- wilcox.test(nativeIosLow$ct, rnIosLow$ct, alt="less")
nativeIosLow_rnIosLow_result

## rnIosLow < pwaIosLow
rnIosLow_pwaIosLow_result <- wilcox.test(rnIosLow$ct, pwaIosLow$ct, alt="less")
rnIosLow_pwaIosLow_result

# ------------------------navigation-ios-high-end------------------------------#
dataIosHigh <- filter(rawData, method == "rnIosHigh" |
                    method == "flutterIosHigh" |
                    method == "pwaIosHigh" |
                    method == "nativeIosHigh")
group_by(dataIosHigh,method) %>%
  summarise(
    count = n(),
    median = median(ct, na.rm = T),
    std = sd(ct, na.rm = T))
# ranks: nativeIosHigh < rnIosHigh < flutterIosHigh < pwaIosHigh

rnIosHigh <- filter(dataIosHigh, method == "rnIosHigh")
nativeIosHigh <- filter(dataIosHigh, method == "nativeIosHigh")
flutterIosHigh <- filter(dataIosHigh, method == "flutterIosHigh")
pwaIosHigh <- filter(dataIosHigh, method == "pwaIosHigh")

## nativeIosHigh < rnIosHigh
nativeIosHigh_rnIosHigh_result <- wilcox.test(nativeIosHigh$ct, rnIosHigh$ct, alt="less")
nativeIosHigh_rnIosHigh_result
```

```
## rnIosHigh < flutterIosHigh
rnIosHigh_flutterIosHigh_result <- wilcox.test(rnIosHigh$ct, flutterIosHigh$ct, alt="less
    ↪ ")
rnIosHigh_flutterIosHigh_result

## flutterIosHigh < pwaIosHigh
pwaIosHigh_flutterIosHigh_result <- wilcox.test(flutterIosHigh$ct, pwaIosHigh$ct, alt="
    ↪ less")
pwaIosHigh_flutterIosHigh_result

# ------------------------navigation-ios-low-end vs high-end
    ↪ ------------------------------#
## nativeIosHigh < nativeIosLow
nativeIosHigh_nativeIosLow_result <- wilcox.test(nativeIosHigh$ct, nativeIosLow$ct, alt="
    ↪ less")
nativeIosHigh_nativeIosLow_result

## rnIosHigh < rnIosLow
rnIosHigh_rnIosLow_result <- wilcox.test(rnIosHigh$ct, rnIosLow$ct, alt="less")
rnIosHigh_rnIosLow_result

## flutterIosHigh > flutterIosLow
flutterIosHigh_flutterIosLow_result <- wilcox.test(flutterIosHigh$ct, flutterIosLow$ct,
    ↪ alt="greater")
flutterIosHigh_flutterIosLow_result

## pwaIosHigh < pwaIosLow
pwaIosHigh_pwaIosLow_result <- wilcox.test(pwaIosHigh$ct, pwaIosLow$ct, alt="less")
pwaIosHigh_pwaIosLow_result
```

Appendix B

Statistical Analysis Code Listings of Resource Use and Performance Experiments

Listing B.1: Statistical analysis code of the CPU-intensive experiment (in R)

```
library(dplyr)

rawData = read.table("/Users/iman/Desktop/results-resource/cpu-intensive-raw-data.csv",
    ↪ sep = ",", header = TRUE)
rawData$method <- as.factor(rawData$method)

# -------------------------android-------------------------------#
dataAndroidLow <- filter(rawData, method == "rnAndroidLow" |
                        method == "flAndroidLow" |
                        method == "pwaAndroidLow" |
                        method == "nativeAndroidLow")
rnAndroidLow <- filter(dataAndroidLow, method == "rnAndroidLow")
flAndroidLow <- filter(dataAndroidLow, method == "flAndroidLow")
pwaAndroidLow <- filter(dataAndroidLow, method == "pwaAndroidLow")
nativeAndroidLow <- filter(dataAndroidLow, method == "nativeAndroidLow")


dataAndroidHigh <- filter(rawData, method == "rnAndroidHigh" |
                        method == "flAndroidHigh" |
                        method == "pwaAndroidHigh" |
                        method == "nativeAndroidHigh")
rnAndroidHigh <- filter(dataAndroidHigh, method == "rnAndroidHigh")
flAndroidHigh <- filter(dataAndroidHigh, method == "flAndroidHigh")
pwaAndroidHigh <- filter(dataAndroidHigh, method == "pwaAndroidHigh")
nativeAndroidHigh <- filter(dataAndroidHigh, method == "nativeAndroidHigh")

# -------------------------ios-------------------------------#
dataIosLow <- filter(rawData, method == "rnIosLow" |
                        method == "flIosLow" |
                        method == "pwaIosLow" |
                        method == "nativeIosLow")
rnIosLow <- filter(dataIosLow, method == "rnIosLow")
flIosLow <- filter(dataIosLow, method == "flIosLow")
pwaIosLow <- filter(dataIosLow, method == "pwaIosLow")
nativeIosLow <- filter(dataIosLow, method == "nativeIosLow")


dataIosHigh <- filter(rawData, method == "rnIosHigh" |
                         method == "flIosHigh" |
                         method == "pwaIosHigh" |
                         method == "nativeIosHigh")
rnIosHigh <- filter(dataIosHigh, method == "rnIosHigh")
flIosHigh <- filter(dataIosHigh, method == "flIosHigh")
```

```
pwaIosHigh <- filter(dataIosHigh, method == "pwaIosHigh")
nativeIosHigh <- filter(dataIosHigh, method == "nativeIosHigh")

# -----------------------------------------------------------------------------Android
    ↪ ------------------------------------------------------------------------------- #
# -------------------------android-low-end-ct-------------------------------#
group_by(dataAndroidLow,method) %>%
  summarise(
    count = n(),
    median = median(ct, na.rm = T),
    std = sd(ct, na.rm = T))
# ranks: nativeAndroidLow < flAndroidLow < pwaAndroidLow < rnAndroidLow

# nativeAndroidLow < flAndroidLow
nativeAndroidLow_flutterAndroidLow_ct_result <- wilcox.test(nativeAndroidLow$ct,
    ↪ flAndroidLow$ct, alt="less")
nativeAndroidLow_flutterAndroidLow_ct_result

# flAndroidLow < pwaAndroidLow
flAndroidLow_pwaAndroidLow_ct_result <- wilcox.test(flAndroidLow$ct, pwaAndroidLow$ct,
    ↪ alt="less")
flAndroidLow_pwaAndroidLow_ct_result

# pwaAndroidLow < rnAndroidLow
pwaAndroidLow_rnAndroidLow_ct_result <- wilcox.test(pwaAndroidLow$ct, rnAndroidLow$ct,
    ↪ alt="less")
pwaAndroidLow_rnAndroidLow_ct_result

# -------------------------android-high-end-ct-------------------------------#
group_by(dataAndroidHigh,method) %>%
  summarise(
    count = n(),
    median = median(ct, na.rm = T),
    std = sd(ct, na.rm = T))
# ranks: pwaAndroidHigh < flAndroidHigh < nativeAndroidHigh < rnAndroidHigh

# pwaAndroidHigh < flAndroidHigh
pwaAndroidHigh_flAndroidHigh_ct_result <- wilcox.test(pwaAndroidHigh$ct, flAndroidHigh$ct
    ↪ , alt="less")
pwaAndroidHigh_flAndroidHigh_ct_result

# flAndroidHigh < nativeAndroidHigh
flAndroidHigh_nativeAndroidHigh_ct_result <- wilcox.test(flAndroidHigh$ct,
    ↪ nativeAndroidHigh$ct, alt="less")
flAndroidHigh_nativeAndroidHigh_ct_result

# nativeAndroidHigh < rnAndroidHigh
nativeAndroidHigh_rnAndroidHigh_ct_result <- wilcox.test(nativeAndroidHigh$ct,
    ↪ rnAndroidHigh$ct, alt="less")
nativeAndroidHigh_rnAndroidHigh_ct_result

# -------------------------android-high-end-low-end-ct-------------------------------#
## nativeAndroidHigh > nativeAndroidLow
```

```
nativeAndroidHigh_nativeAndroidLow_ct_result <- wilcox.test(nativeAndroidHigh$ct,
    ↪ nativeAndroidLow$ct, alt="greater")
nativeAndroidHigh_nativeAndroidLow_ct_result

## rnAndroidHigh < rnAndroidLow
rnAndroidHigh_rnAndroidLow_ct_result <- wilcox.test(rnAndroidHigh$ct, rnAndroidLow$ct,
    ↪ alt="less")
rnAndroidHigh_rnAndroidLow_ct_result

## flAndroidHigh < flAndroidLow
flAndroidHigh_flAndroidLow_ct_result <- wilcox.test(flAndroidHigh$ct, flAndroidLow$ct,
    ↪ alt="less")
flAndroidHigh_flAndroidLow_ct_result

## pwaAndroidHigh < pwaAndroidLow
pwaAndroidHigh_pwaAndroidLow_ct_result <- wilcox.test(pwaAndroidHigh$ct, pwaAndroidLow$ct
    ↪ , alt="less")
pwaAndroidHigh_pwaAndroidLow_ct_result

# -------------------------android-low-end-cpu------------------------------#
group_by(dataAndroidLow,method) %>%
  summarise(
    count = n(),
    median = median(cpu, na.rm = T),
    std = sd(cpu, na.rm = T))
# ranks: nativeAndroidLow < flAndroidLow < pwaAndroidLow < rnAndroidLow

# nativeAndroidLow < flAndroidLow
nativeAndroidLow_flutterAndroidLow_cpu_result <- wilcox.test(nativeAndroidLow$cpu,
    ↪ flAndroidLow$cpu, alt="less")
nativeAndroidLow_flutterAndroidLow_cpu_result

# flAndroidLow < pwaAndroidLow
flAndroidLow_pwaAndroidLow_cpu_result <- wilcox.test(flAndroidLow$cpu, pwaAndroidLow$cpu,
    ↪  alt="less")
flAndroidLow_pwaAndroidLow_cpu_result

# pwaAndroidLow < rnAndroidLow
pwaAndroidLow_rnAndroidLow_cpu_result <- wilcox.test(pwaAndroidLow$cpu, rnAndroidLow$cpu,
    ↪  alt="less")
pwaAndroidLow_rnAndroidLow_cpu_result

# -------------------------android-high-end-cpu------------------------------#
group_by(dataAndroidHigh,method) %>%
  summarise(
    count = n(),
    median = median(cpu, na.rm = T),
    std = sd(cpu, na.rm = T))
# ranks: flAndroidHigh < nativeAndroidHigh < pwaAndroidHigh < rnAndroidHigh

# flAndroidHigh < nativeAndroidHigh
flAndroidHigh_nativeAndroidHigh_cpu_result <- wilcox.test(flAndroidHigh$cpu,
    ↪ nativeAndroidHigh$cpu, alt="less")
flAndroidHigh_nativeAndroidHigh_cpu_result
```

```
# nativeAndroidHigh < pwaAndroidHigh
nativeAndroidHigh_pwaAndroidHigh_cpu_result <- wilcox.test(nativeAndroidHigh$cpu,
    ↪ pwaAndroidHigh$cpu, alt="less")
nativeAndroidHigh_pwaAndroidHigh_cpu_result

# pwaAndroidHigh < rnAndroidHigh
pwaAndroidHigh_rnAndroidHigh_cpu_result <- wilcox.test(pwaAndroidHigh$cpu, rnAndroidHigh$
    ↪ cpu, alt="less")
pwaAndroidHigh_rnAndroidHigh_cpu_result


# -------------------------android-high-end-low-end-cpu-------------------------------#
## nativeAndroidHigh > nativeAndroidLow
nativeAndroidHigh_nativeAndroidLow_cpu_result <- wilcox.test(nativeAndroidHigh$cpu,
    ↪ nativeAndroidLow$cpu, alt="greater")
nativeAndroidHigh_nativeAndroidLow_cpu_result

## rnAndroidHigh > rnAndroidLow
rnAndroidHigh_rnAndroidLow_cpu_result <- wilcox.test(rnAndroidHigh$cpu, rnAndroidLow$cpu,
    ↪  alt="greater")
rnAndroidHigh_rnAndroidLow_cpu_result

## flAndroidHigh > flAndroidLow
flAndroidHigh_flAndroidLow_cpu_result <- wilcox.test(flAndroidHigh$cpu, flAndroidLow$cpu,
    ↪  alt="greater")
flAndroidHigh_flAndroidLow_cpu_result

## pwaAndroidHigh > pwaAndroidLow
pwaAndroidHigh_pwaAndroidLow_cpu_result <- wilcox.test(pwaAndroidHigh$cpu, pwaAndroidLow$
    ↪ cpu, alt="greater")
pwaAndroidHigh_pwaAndroidLow_cpu_result

# -------------------------android-low-end-memory-------------------------------#
group_by(dataAndroidLow,method) %>%
  summarise(
    count = n(),
    median = median(memory, na.rm = T),
    std = sd(memory, na.rm = T))
# ranks: flAndroidHigh < nativeAndroidHigh < pwaAndroidHigh < rnAndroidHigh

# pwaAndroidLow < nativeAndroidLow
pwaAndroidLow_pwaAndroidLow_memory_result <- wilcox.test(pwaAndroidLow$memory,
    ↪ nativeAndroidLow$memory, alt="less")
pwaAndroidLow_pwaAndroidLow_memory_result

# nativeAndroidLow < flAndroidLow
nativeAndroidLow_flAndroidLow_memory_result <- wilcox.test(nativeAndroidLow$memory,
    ↪ flAndroidLow$memory, alt="less")
pwaAndroidLow_pwaAndroidLow_memory_result

# flAndroidLow < rnAndroidLow
flAndroidLow_rnAndroidLow_memory_result <- wilcox.test(flAndroidLow$memory, rnAndroidLow$
    ↪ memory, alt="less")
```

```
flAndroidLow_rnAndroidLow_memory_result

# -------------------------android-high-end-memory-------------------------------#
group_by(dataAndroidHigh,method) %>%
  summarise(
    count = n(),
    median = median(memory, na.rm = T),
    std = sd(memory, na.rm = T))
# ranks: nativeAndroidHigh < pwaAndroidHigh < flAndroidHigh < rnAndroidHigh

# nativeAndroidHigh < pwaAndroidHigh
nativeAndroidHigh_pwaAndroidHigh_memory_result <- wilcox.test(nativeAndroidHigh$memory,
    ↪ pwaAndroidHigh$memory, alt="less")
nativeAndroidHigh_pwaAndroidHigh_memory_result

# pwaAndroidHigh < flAndroidHigh
pwaAndroidHigh_flAndroidHigh_memory_result <- wilcox.test(pwaAndroidHigh$memory,
    ↪ flAndroidHigh$memory, alt="less")
pwaAndroidHigh_flAndroidHigh_memory_result

# flAndroidHigh < rnAndroidHigh
flAndroidHigh_rnAndroidHigh_memory_result <- wilcox.test(flAndroidHigh$memory,
    ↪ rnAndroidHigh$memory, alt="less")
flAndroidHigh_rnAndroidHigh_memory_result

# -------------------------android-high-end-low-end-memory
    ↪ -------------------------------#
## nativeAndroidHigh > nativeAndroidLow
nativeAndroidHigh_nativeAndroidLow_memory_result <- wilcox.test(nativeAndroidHigh$memory,
    ↪  nativeAndroidLow$memory, alt="greater")
nativeAndroidHigh_nativeAndroidLow_memory_result

## rnAndroidHigh > rnAndroidLow
rnAndroidHigh_rnAndroidLow_memory_result <- wilcox.test(rnAndroidHigh$memory,
    ↪ rnAndroidLow$memory, alt="greater")
rnAndroidHigh_rnAndroidLow_memory_result

## flAndroidHigh > flAndroidLow
flAndroidHigh_flAndroidLow_memory_result <- wilcox.test(flAndroidHigh$memory,
    ↪ flAndroidLow$memory, alt="greater")
flAndroidHigh_flAndroidLow_memory_result

## pwaAndroidHigh > pwaAndroidLow
pwaAndroidHigh_pwaAndroidLow_memory_result <- wilcox.test(pwaAndroidHigh$memory,
    ↪ pwaAndroidLow$memory, alt="greater")
pwaAndroidHigh_pwaAndroidLow_memory_result


# -----------------------------------------------------------------------------iOS
    ↪ ----------------------------------------------------------------------------- #
# -------------------------ios-low-end-ct-------------------------------#
group_by(dataIosLow,method) %>%
  summarise(
    count = n(),
```

```
    median = median(ct, na.rm = T),
    std = sd(ct, na.rm = T))
# ranks: flIosLow < pwaIosLow < nativeIosLow < rnIosLow

## flIosLow < pwaIosLow
flIosLow_pwaIosLow_result_ct <- wilcox.test(flIosLow$ct, pwaIosLow$ct, alt="less")
flIosLow_pwaIosLow_result_ct

## pwaIosLow < nativeIosLow
pwaIosLow_nativeIosLow_result_ct <- wilcox.test(pwaIosLow$ct, nativeIosLow$ct, alt="less"
    ↪ )
pwaIosLow_nativeIosLow_result_ct

## nativeIosLow < rnIosLow
nativeIosLow_rnIosLow_result_ct <- wilcox.test(nativeIosLow$ct, rnIosLow$ct, alt="less")
nativeIosLow_rnIosLow_result_ct

# -------------------------ios-high-end-ct-------------------------------#
group_by(dataIosHigh,method) %>%
  summarise(
    count = n(),
    median = median(ct, na.rm = T),
    std = sd(ct, na.rm = T))
# ranks: nativeIosHigh < flIosHigh < pwaIosHigh < rnIosHigh

## nativeIosHigh < flIosHigh
nativeIosHigh_flIosHigh_result_ct <- wilcox.test(nativeIosHigh$ct, flIosHigh$ct, alt="
    ↪ less")
nativeIosHigh_flIosHigh_result_ct

## flIosHigh < pwaIosHigh
flIosHigh_pwaIosHigh_result_ct <- wilcox.test(flIosHigh$ct, pwaIosHigh$ct, alt="less")
flIosHigh_pwaIosHigh_result_ct

## pwaIosHigh < rnIosHigh
pwaIosHigh_rnIosHigh_result_ct <- wilcox.test(pwaIosHigh$ct, rnIosHigh$ct, alt="less")
pwaIosHigh_rnIosHigh_result_ct

# -------------------------ios-high-end-low-end-ct-------------------------------#
## nativeIosHigh < nativeIosLow
nativeIosHigh_nativeIosLow_result_ct <- wilcox.test(nativeIosHigh$ct, nativeIosLow$ct,
    ↪ alt="less")
nativeIosHigh_nativeIosLow_result_ct

## rnIosHigh < rnIosLow
rnIosHigh_rnIosLow_result_ct <- wilcox.test(rnIosHigh$ct, rnIosLow$ct, alt="less")
rnIosHigh_rnIosLow_result_ct

## flIosHigh < flIosLow
flIosHigh_flIosLow_result_ct <- wilcox.test(flIosHigh$ct, flIosLow$ct, alt="less")
flIosHigh_flIosLow_result_ct

## pwaIosHigh < pwaIosLow
pwaIosHigh_pwaIosLow_result_ct <- wilcox.test(pwaIosHigh$ct, pwaIosLow$ct, alt="less")
```

```
pwaIosHigh_pwaIosLow_result_ct

# -------------------------ios-low-end-cpu-------------------------------#
group_by(dataIosLow,method) %>%
  summarise(
    count = n(),
    median = median(cpu, na.rm = T),
    std = sd(cpu, na.rm = T))
# ranks: pwaIosLow < nativeIosLow < flIosLow < rnIosLow

## pwaIosLow < nativeIosLow
pwaIosLow_nativeIosLow_result_cpu <- wilcox.test(pwaIosLow$cpu, nativeIosLow$cpu, alt="
    ↪ less")
pwaIosLow_nativeIosLow_result_cpu

## nativeIosLow < flIosLow
nativeIosLow_flIosLow_result_cpu <- wilcox.test(nativeIosLow$cpu, flIosLow$cpu, alt="less
    ↪ ")
nativeIosLow_flIosLow_result_cpu

## flIosLow < rnIosLow
flIosLow_rnIosLow_result_cpu <- wilcox.test(flIosLow$cpu, rnIosLow$cpu, alt="less")
flIosLow_rnIosLow_result_cpu

# -------------------------ios-high-end-cpu-------------------------------#
group_by(dataIosHigh,method) %>%
  summarise(
    count = n(),
    median = median(cpu, na.rm = T),
    std = sd(cpu, na.rm = T))
# ranks: pwaIosHigh < rnIosHigh < nativeIosHigh < flIosHigh

## pwaIosHigh < rnIosHigh
pwaIosHigh_rnIosHigh_result_cpu <- wilcox.test(pwaIosHigh$cpu, rnIosHigh$cpu, alt="less")
pwaIosHigh_rnIosHigh_result_cpu

## rnIosHigh < nativeIosHigh
rnIosHigh_nativeIosHigh_result_cpu <- wilcox.test(rnIosHigh$cpu, nativeIosHigh$cpu, alt="
    ↪ less")
rnIosHigh_nativeIosHigh_result_cpu

## nativeIosHigh < flIosHigh
nativeIosHigh_flIosHigh_result_cpu <- wilcox.test(nativeIosHigh$cpu, flIosHigh$cpu, alt="
    ↪ less")
nativeIosHigh_flIosHigh_result_cpu

# -------------------------ios-high-end-low-end-cpu-------------------------------#
## nativeIosHigh < nativeIosLow
nativeIosHigh_nativeIosLow_result_cpu <- wilcox.test(nativeIosHigh$cpu, nativeIosLow$cpu,
    ↪  alt="less")
nativeIosHigh_nativeIosLow_result_cpu

## rnIosHigh < rnIosLow
rnIosHigh_rnIosLow_result_cpu <- wilcox.test(rnIosHigh$cpu, rnIosLow$cpu, alt="less")
```

```
rnIosHigh_rnIosLow_result_cpu

## flIosHigh > flIosLow
flIosHigh_flIosLow_result_cpu <- wilcox.test(flIosHigh$cpu, flIosLow$cpu, alt="greater")
flIosHigh_flIosLow_result_cpu

## pwaIosHigh < pwaIosLow
pwaIosHigh_pwaIosLow_result_cpu <- wilcox.test(pwaIosHigh$cpu, pwaIosLow$cpu, alt="less")
pwaIosHigh_pwaIosLow_result_cpu

# -------------------------ios-low-end-memory------------------------------#
group_by(dataIosLow,method) %>%
  summarise(
    count = n(),
    median = median(memory, na.rm = T),
    std = sd(memory, na.rm = T))
# ranks: nativeIosLow < rnIosLow < pwaIosLow < flIosLow

## nativeIosLow < rnIosLow
nativeIosLow_rnIosLow_result_memory <- wilcox.test(nativeIosLow$memory, rnIosLow$memory,
    ↪ alt="less")
nativeIosLow_rnIosLow_result_memory

## rnIosLow < pwaIosLow
rnIosLow_pwaIosLow_result_memory <- wilcox.test(rnIosLow$memory, pwaIosLow$memory, alt="
    ↪ less")
rnIosLow_pwaIosLow_result_memory

## pwaIosLow < flIosLow
pwaIosLow_flIosLow_result_memory <- wilcox.test(pwaIosLow$memory, flIosLow$memory, alt="
    ↪ less")
pwaIosLow_flIosLow_result_memory

# -------------------------ios-high-end-memory------------------------------#
group_by(dataIosHigh,method) %>%
  summarise(
    count = n(),
    median = median(memory, na.rm = T),
    std = sd(memory, na.rm = T))
# ranks: nativeIosHigh < rnIosHigh < pwaIosHigh < flIosHigh

## nativeIosHigh < rnIosHigh
nativeIosHigh_rnIosHigh_result_memory <- wilcox.test(nativeIosHigh$memory, rnIosHigh$
    ↪ memory, alt="less")
nativeIosHigh_rnIosHigh_result_memory

## rnIosHigh < pwaIosHigh
rnIosHigh_pwaIosHigh_result_memory <- wilcox.test(rnIosHigh$memory, pwaIosHigh$memory,
    ↪ alt="less")
rnIosHigh_pwaIosHigh_result_memory

## pwaIosHigh < flIosHigh
pwaIosHigh_flIosHigh_result_memory <- wilcox.test(pwaIosHigh$memory, flIosHigh$memory,
    ↪ alt="less")
```

```
pwaIosHigh_flIosHigh_result_memory

# -------------------------ios-high-end-low-end-memory-------------------------------#
## nativeIosHigh > nativeIosLow
nativeIosHigh_nativeIosLow_result_memory <- wilcox.test(nativeIosHigh$memory,
    ↪ nativeIosLow$memory, alt="greater")
nativeIosHigh_nativeIosLow_result_memory

## rnIosHigh < rnIosLow
rnIosHigh_rnIosLow_result_memory <- wilcox.test(rnIosHigh$memory, rnIosLow$memory, alt="
    ↪ less")
rnIosHigh_rnIosLow_result_memory

## flIosHigh > flIosLow
flIosHigh_flIosLow_result_memory <- wilcox.test(flIosHigh$memory, flIosLow$memory, alt="
    ↪ greater")
flIosHigh_flIosLow_result_memory

## pwaIosHigh > pwaIosLow
pwaIosHigh_pwaIosLow_result_memory <- wilcox.test(pwaIosHigh$memory, pwaIosLow$memory,
    ↪ alt="greater")
pwaIosHigh_pwaIosLow_result_memory
```

Listing B.2: Statistical analysis code of the memory-intensive experiment (in R)

```
library(dplyr)

rawData = read.table("/Users/iman/Desktop/results-resource/memory-intensive-raw-data.csv"
    ↪ , sep = ",", header = TRUE)
rawData$method <- as.factor(rawData$method)

# -------------------------android-------------------------------#
dataAndroidLow <- filter(rawData, method == "rnAndroidLow" |
                        method == "flAndroidLow" |
                        method == "pwaAndroidLow" |
                        method == "nativeAndroidLow")
rnAndroidLow <- filter(dataAndroidLow, method == "rnAndroidLow")
flAndroidLow <- filter(dataAndroidLow, method == "flAndroidLow")
pwaAndroidLow <- filter(dataAndroidLow, method == "pwaAndroidLow")
nativeAndroidLow <- filter(dataAndroidLow, method == "nativeAndroidLow")


dataAndroidHigh <- filter(rawData, method == "rnAndroidHigh" |
                        method == "flAndroidHigh" |
                        method == "pwaAndroidHigh" |
                        method == "nativeAndroidHigh")
rnAndroidHigh <- filter(dataAndroidHigh, method == "rnAndroidHigh")
flAndroidHigh <- filter(dataAndroidHigh, method == "flAndroidHigh")
pwaAndroidHigh <- filter(dataAndroidHigh, method == "pwaAndroidHigh")
nativeAndroidHigh <- filter(dataAndroidHigh, method == "nativeAndroidHigh")

# -------------------------ios-------------------------------#
dataIosLow <- filter(rawData, method == "rnIosLow" |
                        method == "flIosLow" |
                        method == "pwaIosLow" |
```

```
                        method == "nativeIosLow")
rnIosLow <- filter(dataIosLow, method == "rnIosLow")
flIosLow <- filter(dataIosLow, method == "flIosLow")
pwaIosLow <- filter(dataIosLow, method == "pwaIosLow")
nativeIosLow <- filter(dataIosLow, method == "nativeIosLow")


dataIosHigh <- filter(rawData, method == "rnIosHigh" |
                         method == "flIosHigh" |
                         method == "pwaIosHigh" |
                         method == "nativeIosHigh")
rnIosHigh <- filter(dataIosHigh, method == "rnIosHigh")
flIosHigh <- filter(dataIosHigh, method == "flIosHigh")
pwaIosHigh <- filter(dataIosHigh, method == "pwaIosHigh")
nativeIosHigh <- filter(dataIosHigh, method == "nativeIosHigh")

# ----------------------------------------------------------------------------Android
    ↪ ---------------------------------------------------------------------------- #
# -------------------------android-low-end-ct-------------------------------#
group_by(dataAndroidLow,method) %>%
  summarise(
    count = n(),
    median = median(ct, na.rm = T),
    std = sd(ct, na.rm = T))
# ranks: nativeAndroidLow < pwaAndroidLow < rnAndroidLow < flAndroidLow

# nativeAndroidLow < pwaAndroidLow
nativeAndroidLow_pwaAndroidLow_ct_result <- wilcox.test(nativeAndroidLow$ct,
    ↪ pwaAndroidLow$ct, alt="less")
nativeAndroidLow_pwaAndroidLow_ct_result

# pwaAndroidLow < rnAndroidLow
pwaAndroidLow_rnAndroidLow_ct_result <- wilcox.test(pwaAndroidLow$ct, rnAndroidLow$ct,
    ↪ alt="less")
pwaAndroidLow_rnAndroidLow_ct_result

# rnAndroidLow < flAndroidLow
rnAndroidLow_flAndroidLow_ct_result <- wilcox.test(rnAndroidLow$ct, flAndroidLow$ct, alt=
    ↪ "less")
rnAndroidLow_flAndroidLow_ct_result

# -------------------------android-high-end-ct-------------------------------#
group_by(dataAndroidHigh,method) %>%
  summarise(
    count = n(),
    median = median(ct, na.rm = T),
    std = sd(ct, na.rm = T))
# ranks: nativeAndroidHigh < pwaAndroidHigh < rnAndroidHigh < flAndroidHigh

# nativeAndroidHigh < pwaAndroidHigh
nativeAndroidHigh_pwaAndroidHigh_ct_result <- wilcox.test(nativeAndroidHigh$ct,
    ↪ pwaAndroidHigh$ct, alt="less")
nativeAndroidHigh_pwaAndroidHigh_ct_result
```

```
# pwaAndroidHigh < rnAndroidHigh
pwaAndroidHigh_rnAndroidHigh_ct_result <- wilcox.test(pwaAndroidHigh$ct, rnAndroidHigh$ct
    ↪ , alt="less")
pwaAndroidHigh_rnAndroidHigh_ct_result

# rnAndroidHigh < flAndroidHigh
rnAndroidHigh_flAndroidHigh_ct_result <- wilcox.test(rnAndroidHigh$ct, flAndroidHigh$ct,
    ↪ alt="less")
rnAndroidHigh_flAndroidHigh_ct_result

# -------------------------android-high-end-low-end-ct------------------------------#
## nativeAndroidHigh < nativeAndroidLow
nativeAndroidHigh_nativeAndroidLow_ct_result <- wilcox.test(nativeAndroidHigh$ct,
    ↪ nativeAndroidLow$ct, alt="less")
nativeAndroidHigh_nativeAndroidLow_ct_result

## rnAndroidHigh < rnAndroidLow
rnAndroidHigh_rnAndroidLow_ct_result <- wilcox.test(rnAndroidHigh$ct, rnAndroidLow$ct,
    ↪ alt="less")
rnAndroidHigh_rnAndroidLow_ct_result

## flAndroidHigh < flAndroidLow
flAndroidHigh_flAndroidLow_ct_result <- wilcox.test(flAndroidHigh$ct, flAndroidLow$ct,
    ↪ alt="less")
flAndroidHigh_flAndroidLow_ct_result

## pwaAndroidHigh < pwaAndroidLow
pwaAndroidHigh_pwaAndroidLow_ct_result <- wilcox.test(pwaAndroidHigh$ct, pwaAndroidLow$ct
    ↪ , alt="less")
pwaAndroidHigh_pwaAndroidLow_ct_result

# -------------------------android-low-end-cpu------------------------------#
group_by(dataAndroidLow,method) %>%
  summarise(
    count = n(),
    median = median(cpu, na.rm = T),
    std = sd(cpu, na.rm = T))
# ranks: rnAndroidLow < flAndroidLow < nativeAndroidLow < pwaAndroidLow

# rnAndroidLow < flAndroidLow
rnAndroidLow_flAndroidLow_cpu_result <- wilcox.test(rnAndroidLow$cpu, flAndroidLow$cpu,
    ↪ alt="less")
rnAndroidLow_flAndroidLow_cpu_result

# flAndroidLow < nativeAndroidLow
flAndroidLow_nativeAndroidLow_cpu_result <- wilcox.test(flAndroidLow$cpu,
    ↪ nativeAndroidLow$cpu, alt="less")
flAndroidLow_nativeAndroidLow_cpu_result

# nativeAndroidLow < pwaAndroidLow
nativeAndroidLow_pwaAndroidLow_cpu_result <- wilcox.test(nativeAndroidLow$cpu,
    ↪ pwaAndroidLow$cpu, alt="less")
nativeAndroidLow_pwaAndroidLow_cpu_result
```

```
# -------------------------android-high-end-cpu-------------------------------#
group_by(dataAndroidHigh,method) %>%
  summarise(
    count = n(),
    median = median(cpu, na.rm = T),
    std = sd(cpu, na.rm = T))
# ranks: rnAndroidHigh < pwaAndroidHigh < flAndroidHigh < nativeAndroidHigh

# rnAndroidHigh < pwaAndroidHigh
rnAndroidHigh_pwaAndroidHigh_cpu_result <- wilcox.test(rnAndroidHigh$cpu, pwaAndroidHigh$
    ↪ cpu, alt="less")
rnAndroidHigh_pwaAndroidHigh_cpu_result

# pwaAndroidHigh < flAndroidHigh
pwaAndroidHigh_flAndroidHigh_cpu_result <- wilcox.test(pwaAndroidHigh$cpu, flAndroidHigh$
    ↪ cpu, alt="less")
pwaAndroidHigh_flAndroidHigh_cpu_result

# flAndroidHigh < nativeAndroidHigh
flAndroidHigh_nativeAndroidHigh_cpu_result <- wilcox.test(flAndroidHigh$cpu,
    ↪ nativeAndroidHigh$cpu, alt="less")
flAndroidHigh_nativeAndroidHigh_cpu_result

# -------------------------android-high-end-low-end-cpu-------------------------------#
## nativeAndroidHigh > nativeAndroidLow
nativeAndroidHigh_nativeAndroidLow_cpu_result <- wilcox.test(nativeAndroidHigh$cpu,
    ↪ nativeAndroidLow$cpu, alt="greater")
nativeAndroidHigh_nativeAndroidLow_cpu_result

## rnAndroidHigh > rnAndroidLow
rnAndroidHigh_rnAndroidLow_cpu_result <- wilcox.test(rnAndroidHigh$cpu, rnAndroidLow$cpu,
    ↪  alt="greater")
rnAndroidHigh_rnAndroidLow_cpu_result

## flAndroidHigh > flAndroidLow
flAndroidHigh_flAndroidLow_cpu_result <- wilcox.test(flAndroidHigh$cpu, flAndroidLow$cpu,
    ↪  alt="greater")
flAndroidHigh_flAndroidLow_cpu_result

## pwaAndroidHigh < pwaAndroidLow
pwaAndroidHigh_pwaAndroidLow_cpu_result <- wilcox.test(pwaAndroidHigh$cpu, pwaAndroidLow$
    ↪ cpu, alt="less")
pwaAndroidHigh_pwaAndroidLow_cpu_result

# -------------------------android-low-end-memory-------------------------------#
group_by(dataAndroidLow,method) %>%
  summarise(
    count = n(),
    median = median(memory, na.rm = T),
    std = sd(memory, na.rm = T))
# ranks: pwaAndroidHigh < nativeAndroidHigh < rnAndroidHigh < flAndroidHigh

# pwaAndroidHigh < nativeAndroidHigh
pwaAndroidLow_nativeAndroidHigh_memory_result <- wilcox.test(pwaAndroidHigh$memory,
```

```
    ↪ nativeAndroidHigh$memory, alt="less")
pwaAndroidLow_nativeAndroidHigh_memory_result

# nativeAndroidHigh < rnAndroidHigh
nativeAndroidHigh_rnAndroidHigh_memory_result <- wilcox.test(nativeAndroidHigh$memory,
    ↪ rnAndroidHigh$memory, alt="less")
nativeAndroidHigh_rnAndroidHigh_memory_result

# rnAndroidHigh < flAndroidHigh
rnAndroidHigh_flAndroidHigh_memory_result <- wilcox.test(rnAndroidHigh$memory,
    ↪ flAndroidHigh$memory, alt="less")
rnAndroidHigh_flAndroidHigh_memory_result

# -------------------------android-high-end-memory-------------------------------#
group_by(dataAndroidHigh,method) %>%
  summarise(
    count = n(),
    median = median(memory, na.rm = T),
    std = sd(memory, na.rm = T))
# ranks: pwaAndroidHigh < nativeAndroidHigh < rnAndroidHigh < flAndroidHigh

# pwaAndroidHigh < nativeAndroidHigh
pwaAndroidHigh_nativeAndroidHigh_memory_result <- wilcox.test(pwaAndroidHigh$memory,
    ↪ nativeAndroidHigh$memory, alt="less")
pwaAndroidHigh_nativeAndroidHigh_memory_result

# nativeAndroidHigh < rnAndroidHigh
nativeAndroidHigh_rnAndroidHigh_memory_result <- wilcox.test(nativeAndroidHigh$memory,
    ↪ rnAndroidHigh$memory, alt="less")
nativeAndroidHigh_rnAndroidHigh_memory_result

# rnAndroidHigh < flAndroidHigh
rnAndroidHigh_flAndroidHigh_memory_result <- wilcox.test(rnAndroidHigh$memory,
    ↪ flAndroidHigh$memory, alt="less")
rnAndroidHigh_flAndroidHigh_memory_result

# -------------------------android-high-end-low-end-memory
    ↪ -------------------------------#
## nativeAndroidHigh > nativeAndroidLow
nativeAndroidHigh_nativeAndroidLow_memory_result <- wilcox.test(nativeAndroidHigh$memory,
    ↪  nativeAndroidLow$memory, alt="greater")
nativeAndroidHigh_nativeAndroidLow_memory_result

## rnAndroidHigh > rnAndroidLow
rnAndroidHigh_rnAndroidLow_memory_result <- wilcox.test(rnAndroidHigh$memory,
    ↪ rnAndroidLow$memory, alt="greater")
rnAndroidHigh_rnAndroidLow_memory_result

## flAndroidHigh > flAndroidLow
flAndroidHigh_flAndroidLow_memory_result <- wilcox.test(flAndroidHigh$memory,
    ↪ flAndroidLow$memory, alt="greater")
flAndroidHigh_flAndroidLow_memory_result

## pwaAndroidHigh > pwaAndroidLow
```

```
pwaAndroidHigh_pwaAndroidLow_memory_result <- wilcox.test(pwaAndroidHigh$memory,
    ↪ pwaAndroidLow$memory, alt="greater")
pwaAndroidHigh_pwaAndroidLow_memory_result

# ----------------------------------------------------------------------------iOS
    ↪ ------------------------------------------------------------------------------ #
# -------------------------ios-low-end-ct-------------------------------#
group_by(dataIosLow,method) %>%
  summarise(
    count = n(),
    median = median(ct, na.rm = T),
    std = sd(ct, na.rm = T))
# ranks: rnIosLow < flIosLow < nativeIosLow

## rnIosLow < flIosLow
rnIosLow_flIosLow_result_ct <- wilcox.test(rnIosLow$ct, flIosLow$ct, alt="less")
rnIosLow_flIosLow_result_ct

## flIosLow < nativeIosLow
flIosLow_nativeIosLow_result_ct <- wilcox.test(flIosLow$ct, nativeIosLow$ct, alt="less")
flIosLow_nativeIosLow_result_ct

# -------------------------ios-high-end-ct-------------------------------#
group_by(dataIosHigh,method) %>%
  summarise(
    count = n(),
    median = median(ct, na.rm = T),
    std = sd(ct, na.rm = T))
# ranks: pwaIosHigh < flIosHigh < rnIosHigh < nativeIosHigh

## pwaIosHigh < flIosHigh
pwaIosHigh_flIosHigh_result_ct <- wilcox.test(pwaIosHigh$ct, flIosHigh$ct, alt="less")
pwaIosHigh_flIosHigh_result_ct

## flIosHigh < rnIosHigh
flIosHigh_rnIosHigh_result_ct <- wilcox.test(flIosHigh$ct, rnIosHigh$ct, alt="less")
flIosHigh_rnIosHigh_result_ct

## rnIosHigh < nativeIosHigh
rnIosHigh_nativeIosHigh_result_ct <- wilcox.test(rnIosHigh$ct, nativeIosHigh$ct, alt="
    ↪ less")
rnIosHigh_nativeIosHigh_result_ct

# -------------------------ios-high-end-low-end-ct-------------------------------#
## nativeIosHigh < nativeIosLow
nativeIosHigh_nativeIosLow_result_ct <- wilcox.test(nativeIosHigh$ct, nativeIosLow$ct,
    ↪ alt="less")
nativeIosHigh_nativeIosLow_result_ct

## rnIosHigh < rnIosLow
rnIosHigh_rnIosLow_result_ct <- wilcox.test(rnIosHigh$ct, rnIosLow$ct, alt="less")
rnIosHigh_rnIosLow_result_ct

## flIosHigh < flIosLow
```

```
flIosHigh_flIosLow_result_ct <- wilcox.test(flIosHigh$ct, flIosLow$ct, alt="less")
flIosHigh_flIosLow_result_ct

# -------------------------ios-low-end-cpu-------------------------------#
group_by(dataIosLow,method) %>%
  summarise(
    count = n(),
    median = median(cpu, na.rm = T),
    std = sd(cpu, na.rm = T))
# ranks: nativeIosLow < flIosLow < rnIosLow

## nativeIosLow < flIosLow
nativeIosLow_flIosLow_result_cpu <- wilcox.test(nativeIosLow$cpu, flIosLow$cpu, alt="less
    ↪ ")
nativeIosLow_flIosLow_result_cpu

## flIosLow < rnIosLow
flIosLow_rnIosLow_result_cpu <- wilcox.test(flIosLow$cpu, rnIosLow$cpu, alt="less")
flIosLow_rnIosLow_result_cpu

# -------------------------ios-high-end-cpu-------------------------------#
group_by(dataIosHigh,method) %>%
  summarise(
    count = n(),
    median = median(cpu, na.rm = T),
    std = sd(cpu, na.rm = T))
# ranks: pwaIosHigh < nativeIosHigh < flIosHigh < rnIosHigh

## pwaIosHigh < nativeIosHigh
pwaIosHigh_nativeIosHigh_result_cpu <- wilcox.test(pwaIosHigh$cpu, nativeIosHigh$cpu, alt
    ↪ ="less")
pwaIosHigh_nativeIosHigh_result_cpu

## nativeIosHigh < flIosHigh
nativeIosHigh_flIosHigh_result_cpu <- wilcox.test(nativeIosHigh$cpu, flIosHigh$cpu, alt="
    ↪ less")
nativeIosHigh_flIosHigh_result_cpu

## flIosHigh < rnIosHigh
flIosHigh_rnIosHigh_result_cpu <- wilcox.test(flIosHigh$cpu, rnIosHigh$cpu, alt="less")
flIosHigh_rnIosHigh_result_cpu

# -------------------------ios-high-end-low-end-cpu-------------------------------#
## nativeIosHigh < nativeIosLow
nativeIosHigh_nativeIosLow_result_cpu <- wilcox.test(nativeIosHigh$cpu, nativeIosLow$cpu,
    ↪  alt="less")
nativeIosHigh_nativeIosLow_result_cpu

## rnIosHigh < rnIosLow
rnIosHigh_rnIosLow_result_cpu <- wilcox.test(rnIosHigh$cpu, rnIosLow$cpu, alt="less")
rnIosHigh_rnIosLow_result_cpu

## flIosHigh < flIosLow
flIosHigh_flIosLow_result_cpu <- wilcox.test(flIosHigh$cpu, flIosLow$cpu, alt="less")
```

```
flIosHigh_flIosLow_result_cpu

# -------------------------ios-low-end-memory------------------------------#
group_by(dataIosLow,method) %>%
  summarise(
    count = n(),
    median = median(memory, na.rm = T),
    std = sd(memory, na.rm = T))
# ranks: nativeIosLow < rnIosLow < flIosLow

## nativeIosLow < rnIosLow
nativeIosLow_rnIosLow_result_memory <- wilcox.test(nativeIosLow$memory, rnIosLow$memory,
    ↪ alt="less")
nativeIosLow_rnIosLow_result_memory

## rnIosLow < flIosLow
rnIosLow_flIosLow_result_memory <- wilcox.test(rnIosLow$memory, flIosLow$memory, alt="
    ↪ less")
rnIosLow_flIosLow_result_memory

# -------------------------ios-high-end-memory------------------------------#
group_by(dataIosHigh,method) %>%
  summarise(
    count = n(),
    median = median(memory, na.rm = T),
    std = sd(memory, na.rm = T))
# ranks: nativeIosHigh < rnIosHigh < flIosHigh < pwaIosHigh

## nativeIosHigh < rnIosHigh
nativeIosHigh_rnIosHigh_result_memory <- wilcox.test(nativeIosHigh$memory, rnIosHigh$
    ↪ memory, alt="less")
nativeIosHigh_rnIosHigh_result_memory

## rnIosHigh < flIosHigh
rnIosHigh_flIosHigh_result_memory <- wilcox.test(rnIosHigh$memory, flIosHigh$memory, alt=
    ↪ "less")
rnIosHigh_flIosHigh_result_memory

## flIosHigh < pwaIosHigh
flIosHigh_pwaIosHigh_result_memory <- wilcox.test(flIosHigh$memory, pwaIosHigh$memory,
    ↪ alt="less")
flIosHigh_pwaIosHigh_result_memory

# -------------------------ios-high-end-low-end-memory------------------------------#
## nativeIosHigh < nativeIosLow
nativeIosHigh_nativeIosLow_result_memory <- wilcox.test(nativeIosHigh$memory,
    ↪ nativeIosLow$memory, alt="less")
nativeIosHigh_nativeIosLow_result_memory

## rnIosHigh < rnIosLow
rnIosHigh_rnIosLow_result_memory <- wilcox.test(rnIosHigh$memory, rnIosLow$memory, alt="
    ↪ less")
rnIosHigh_rnIosLow_result_memory
```

```
## flIosHigh < flIosLow
flIosHigh_flIosLow_result_memory <- wilcox.test(flIosHigh$memory, flIosLow$memory, alt="
    ↪ less")
flIosHigh_flIosLow_result_memory
```

Listing B.3: Statistical analysis code of the network-intensive experiment (in R)

```
library(dplyr)

rawData = read.table("/Users/iman/Desktop/results-resource/network-intensive-raw-data.csv
    ↪ ", sep = ",", header = TRUE)
rawData$method <- as.factor(rawData$method)

# -------------------------android-------------------------------#
dataAndroidLow <- filter(rawData, method == "rnAndroidLow" |
                        method == "flAndroidLow" |
                        method == "pwaAndroidLow" |
                        method == "nativeAndroidLow")
rnAndroidLow <- filter(dataAndroidLow, method == "rnAndroidLow")
flAndroidLow <- filter(dataAndroidLow, method == "flAndroidLow")
pwaAndroidLow <- filter(dataAndroidLow, method == "pwaAndroidLow")
nativeAndroidLow <- filter(dataAndroidLow, method == "nativeAndroidLow")


dataAndroidHigh <- filter(rawData, method == "rnAndroidHigh" |
                        method == "flAndroidHigh" |
                        method == "pwaAndroidHigh" |
                        method == "nativeAndroidHigh")
rnAndroidHigh <- filter(dataAndroidHigh, method == "rnAndroidHigh")
flAndroidHigh <- filter(dataAndroidHigh, method == "flAndroidHigh")
pwaAndroidHigh <- filter(dataAndroidHigh, method == "pwaAndroidHigh")
nativeAndroidHigh <- filter(dataAndroidHigh, method == "nativeAndroidHigh")

# -------------------------ios-------------------------------#
dataIosLow <- filter(rawData, method == "rnIosLow" |
                        method == "flIosLow" |
                        method == "pwaIosLow" |
                        method == "nativeIosLow")
rnIosLow <- filter(dataIosLow, method == "rnIosLow")
flIosLow <- filter(dataIosLow, method == "flIosLow")
pwaIosLow <- filter(dataIosLow, method == "pwaIosLow")
nativeIosLow <- filter(dataIosLow, method == "nativeIosLow")


dataIosHigh <- filter(rawData, method == "rnIosHigh" |
                         method == "flIosHigh" |
                         method == "pwaIosHigh" |
                         method == "nativeIosHigh")
rnIosHigh <- filter(dataIosHigh, method == "rnIosHigh")
flIosHigh <- filter(dataIosHigh, method == "flIosHigh")
pwaIosHigh <- filter(dataIosHigh, method == "pwaIosHigh")
nativeIosHigh <- filter(dataIosHigh, method == "nativeIosHigh")

# ----------------------------------------------------------------------------Android
    ↪ ------------------------------------------------------------------------------ #
```

```
# -------------------------android-low-end-ct------------------------------#
group_by(dataAndroidLow,method) %>%
  summarise(
    count = n(),
    median = median(ct, na.rm = T),
    std = sd(ct, na.rm = T))
# ranks: flAndroidLow < nativeAndroidLow < pwaAndroidLow < rnAndroidLow <

# flAndroidLow < nativeAndroidLow
flAndroidLow_nativeAndroidLow_ct_result <- wilcox.test(flAndroidLow$ct, nativeAndroidLow$
    ↪ ct, alt="less")
flAndroidLow_nativeAndroidLow_ct_result

# nativeAndroidLow < pwaAndroidLow
nativeAndroidLow_pwaAndroidLow_ct_result <- wilcox.test(nativeAndroidLow$ct,
    ↪ pwaAndroidLow$ct, alt="less")
nativeAndroidLow_pwaAndroidLow_ct_result

# pwaAndroidLow < rnAndroidLow
pwaAndroidLow_rnAndroidLow_ct_result <- wilcox.test(pwaAndroidLow$ct, rnAndroidLow$ct,
    ↪ alt="less")
pwaAndroidLow_rnAndroidLow_ct_result

# -------------------------android-high-end-ct------------------------------#
group_by(dataAndroidHigh,method) %>%
  summarise(
    count = n(),
    median = median(ct, na.rm = T),
    std = sd(ct, na.rm = T))
# ranks: flAndroidHigh < pwaAndroidHigh < rnAndroidHigh < nativeAndroidHigh

# flAndroidHigh < pwaAndroidHigh
flAndroidHigh_pwaAndroidHigh_ct_result <- wilcox.test(flAndroidHigh$ct, pwaAndroidHigh$ct
    ↪ , alt="less")
flAndroidHigh_pwaAndroidHigh_ct_result

# pwaAndroidHigh < rnAndroidHigh
pwaAndroidHigh_rnAndroidHigh_ct_result <- wilcox.test(pwaAndroidHigh$ct, rnAndroidHigh$ct
    ↪ , alt="less")
pwaAndroidHigh_rnAndroidHigh_ct_result

# rnAndroidHigh < nativeAndroidHigh
rnAndroidHigh_nativeAndroidHigh_ct_result <- wilcox.test(rnAndroidHigh$ct,
    ↪ nativeAndroidHigh$ct, alt="less")
rnAndroidHigh_nativeAndroidHigh_ct_result

# -------------------------android-high-end-low-end-ct------------------------------#
## nativeAndroidHigh < nativeAndroidLow
nativeAndroidHigh_nativeAndroidLow_ct_result <- wilcox.test(nativeAndroidHigh$ct,
    ↪ nativeAndroidLow$ct, alt="less")
nativeAndroidHigh_nativeAndroidLow_ct_result

## rnAndroidHigh < rnAndroidLow
rnAndroidHigh_rnAndroidLow_ct_result <- wilcox.test(rnAndroidHigh$ct, rnAndroidLow$ct,
```

```
    ↪ alt="less")
rnAndroidHigh_rnAndroidLow_ct_result

## flAndroidHigh < flAndroidLow
flAndroidHigh_flAndroidLow_ct_result <- wilcox.test(flAndroidHigh$ct, flAndroidLow$ct,
    ↪ alt="less")
flAndroidHigh_flAndroidLow_ct_result

## pwaAndroidHigh < pwaAndroidLow
pwaAndroidHigh_pwaAndroidLow_ct_result <- wilcox.test(pwaAndroidHigh$ct, pwaAndroidLow$ct
    ↪ , alt="less")
pwaAndroidHigh_pwaAndroidLow_ct_result

# -------------------------android-low-end-cpu-------------------------------#
group_by(dataAndroidLow,method) %>%
  summarise(
    count = n(),
    median = median(cpu, na.rm = T),
    std = sd(cpu, na.rm = T))
# ranks: nativeAndroidLow < flAndroidLow < pwaAndroidLow < rnAndroidLow

# nativeAndroidLow < flAndroidLow
nativeAndroidLow_flAndroidLow_cpu_result <- wilcox.test(nativeAndroidLow$cpu,
    ↪ flAndroidLow$cpu, alt="less")
nativeAndroidLow_flAndroidLow_cpu_result

# flAndroidLow < pwaAndroidLow
flAndroidLow_pwaAndroidLow_cpu_result <- wilcox.test(flAndroidLow$cpu, pwaAndroidLow$cpu,
    ↪  alt="less")
flAndroidLow_pwaAndroidLow_cpu_result

# pwaAndroidLow < rnAndroidLow
pwaAndroidLow_rnAndroidLow_cpu_result <- wilcox.test(pwaAndroidLow$cpu, rnAndroidLow$cpu,
    ↪  alt="less")
pwaAndroidLow_rnAndroidLow_cpu_result

# -------------------------android-high-end-cpu------------------------------#
group_by(dataAndroidHigh,method) %>%
  summarise(
    count = n(),
    median = median(cpu, na.rm = T),
    std = sd(cpu, na.rm = T))
# ranks: nativeAndroidHigh < pwaAndroidHigh < flAndroidHigh < rnAndroidHigh

# nativeAndroidHigh < pwaAndroidHigh
nativeAndroidHigh_pwaAndroidHigh_cpu_result <- wilcox.test(nativeAndroidHigh$cpu,
    ↪ pwaAndroidHigh$cpu, alt="less")
nativeAndroidHigh_pwaAndroidHigh_cpu_result

# pwaAndroidHigh < flAndroidHigh
pwaAndroidHigh_flAndroidHigh_cpu_result <- wilcox.test(pwaAndroidHigh$cpu, flAndroidHigh$
    ↪ cpu, alt="less")
pwaAndroidHigh_flAndroidHigh_cpu_result
```

```
# flAndroidHigh < rnAndroidHigh
flAndroidHigh_rnAndroidHigh_cpu_result <- wilcox.test(flAndroidHigh$cpu, rnAndroidHigh$
    ↪ cpu, alt="less")
flAndroidHigh_rnAndroidHigh_cpu_result

# -------------------------android-high-end-low-end-cpu------------------------------#
## nativeAndroidHigh < nativeAndroidLow
nativeAndroidHigh_nativeAndroidLow_cpu_result <- wilcox.test(nativeAndroidHigh$cpu,
    ↪ nativeAndroidLow$cpu, alt="less")
nativeAndroidHigh_nativeAndroidLow_cpu_result

## rnAndroidHigh > rnAndroidLow
rnAndroidHigh_rnAndroidLow_cpu_result <- wilcox.test(rnAndroidHigh$cpu, rnAndroidLow$cpu,
    ↪  alt="greater")
rnAndroidHigh_rnAndroidLow_cpu_result

## flAndroidHigh > flAndroidLow
flAndroidHigh_flAndroidLow_cpu_result <- wilcox.test(flAndroidHigh$cpu, flAndroidLow$cpu,
    ↪  alt="greater")
flAndroidHigh_flAndroidLow_cpu_result

## pwaAndroidHigh < pwaAndroidLow
pwaAndroidHigh_pwaAndroidLow_cpu_result <- wilcox.test(pwaAndroidHigh$cpu, pwaAndroidLow$
    ↪ cpu, alt="less")
pwaAndroidHigh_pwaAndroidLow_cpu_result

# -------------------------android-low-end-memory------------------------------#
group_by(dataAndroidLow,method) %>%
  summarise(
    count = n(),
    median = median(memory, na.rm = T),
    std = sd(memory, na.rm = T))
# ranks: nativeAndroidHigh < flAndroidHigh < rnAndroidHigh < pwaAndroidHigh

# nativeAndroidHigh < flAndroidHigh
nativeAndroidHigh_flAndroidHigh_memory_result <- wilcox.test(nativeAndroidHigh$memory,
    ↪ flAndroidHigh$memory, alt="less")
nativeAndroidHigh_flAndroidHigh_memory_result

# flAndroidHigh < rnAndroidHigh
flAndroidHigh_rnAndroidHigh_memory_result <- wilcox.test(flAndroidHigh$memory,
    ↪ rnAndroidHigh$memory, alt="less")
flAndroidHigh_rnAndroidHigh_memory_result

# rnAndroidHigh < pwaAndroidHigh
rnAndroidHigh_pwaAndroidHigh_memory_result <- wilcox.test(rnAndroidHigh$memory,
    ↪ pwaAndroidHigh$memory, alt="less")
rnAndroidHigh_pwaAndroidHigh_memory_result

# -------------------------android-high-end-memory------------------------------#
group_by(dataAndroidHigh,method) %>%
  summarise(
    count = n(),
    median = median(memory, na.rm = T),
```

```
    std = sd(memory, na.rm = T))
# ranks: nativeAndroidHigh < flAndroidHigh < rnAndroidHigh < pwaAndroidHigh

# nativeAndroidHigh < flAndroidHigh
nativeAndroidHigh_flAndroidHigh_memory_result <- wilcox.test(nativeAndroidHigh$memory,
    ↪ flAndroidHigh$memory, alt="less")
nativeAndroidHigh_flAndroidHigh_memory_result

# flAndroidHigh < rnAndroidHigh
flAndroidHigh_rnAndroidHigh_memory_result <- wilcox.test(flAndroidHigh$memory,
    ↪ rnAndroidHigh$memory, alt="less")
flAndroidHigh_rnAndroidHigh_memory_result

# rnAndroidHigh < pwaAndroidHigh
rnAndroidHigh_pwaAndroidHigh_memory_result <- wilcox.test(rnAndroidHigh$memory,
    ↪ pwaAndroidHigh$memory, alt="less")
rnAndroidHigh_pwaAndroidHigh_memory_result

# -------------------------android-high-end-low-end-memory
    ↪ -------------------------------#
## nativeAndroidHigh < nativeAndroidLow
nativeAndroidHigh_nativeAndroidLow_memory_result <- wilcox.test(nativeAndroidHigh$memory,
    ↪  nativeAndroidLow$memory, alt="less")
nativeAndroidHigh_nativeAndroidLow_memory_result

## rnAndroidHigh < rnAndroidLow
rnAndroidHigh_rnAndroidLow_memory_result <- wilcox.test(rnAndroidHigh$memory,
    ↪ rnAndroidLow$memory, alt="less")
rnAndroidHigh_rnAndroidLow_memory_result

## flAndroidHigh < flAndroidLow
flAndroidHigh_flAndroidLow_memory_result <- wilcox.test(flAndroidHigh$memory,
    ↪ flAndroidLow$memory, alt="less")
flAndroidHigh_flAndroidLow_memory_result

## pwaAndroidHigh > pwaAndroidLow
pwaAndroidHigh_pwaAndroidLow_memory_result <- wilcox.test(pwaAndroidHigh$memory,
    ↪ pwaAndroidLow$memory, alt="greater")
pwaAndroidHigh_pwaAndroidLow_memory_result

# -------------------------android-low-end-network-------------------------------#
group_by(dataAndroidLow,method) %>%
  summarise(
    count = n(),
    median = median(network, na.rm = T),
    std = sd(network, na.rm = T))
# ranks: pwaAndroidHigh < nativeAndroidHigh < rnAndroidHigh < flAndroidHigh

# pwaAndroidHigh < nativeAndroidHigh
pwaAndroidHigh_nativeAndroidHigh_network_result <- wilcox.test(pwaAndroidHigh$network,
    ↪ nativeAndroidHigh$network, alt="less")
pwaAndroidHigh_nativeAndroidHigh_network_result

# nativeAndroidHigh < rnAndroidHigh
```

```
nativeAndroidHigh_rnAndroidHigh_network_result <- wilcox.test(nativeAndroidHigh$network,
    ↪ rnAndroidHigh$network, alt="less")
nativeAndroidHigh_rnAndroidHigh_network_result

# rnAndroidHigh < flAndroidHigh
rnAndroidHigh_flAndroidHigh_network_result <- wilcox.test(rnAndroidHigh$network,
    ↪ flAndroidHigh$network, alt="less")
rnAndroidHigh_flAndroidHigh_network_result

# -------------------------android-high-end-network-------------------------------#
group_by(dataAndroidHigh,method) %>%
  summarise(
    count = n(),
    median = median(network, na.rm = T),
    std = sd(network, na.rm = T))
# ranks: pwaAndroidHigh < nativeAndroidHigh < rnAndroidHigh < flAndroidHigh

# pwaAndroidHigh < nativeAndroidHigh
pwaAndroidHigh_nativeAndroidHigh_network_result <- wilcox.test(pwaAndroidHigh$network,
    ↪ nativeAndroidHigh$network, alt="less")
pwaAndroidHigh_nativeAndroidHigh_network_result

# nativeAndroidHigh < rnAndroidHigh
nativeAndroidHigh_rnAndroidHigh_network_result <- wilcox.test(nativeAndroidHigh$network,
    ↪ rnAndroidHigh$network, alt="less")
nativeAndroidHigh_rnAndroidHigh_network_result

# rnAndroidHigh < flAndroidHigh
rnAndroidHigh_flAndroidHigh_network_result <- wilcox.test(rnAndroidHigh$network,
    ↪ flAndroidHigh$network, alt="less")
rnAndroidHigh_flAndroidHigh_network_result

# -------------------------android-high-end-low-end-network
    ↪ -------------------------------#
## nativeAndroidHigh > nativeAndroidLow
nativeAndroidHigh_nativeAndroidLow_network_result <- wilcox.test(nativeAndroidHigh$
    ↪ network, nativeAndroidLow$network, alt="greater")
nativeAndroidHigh_nativeAndroidLow_network_result

## rnAndroidHigh > rnAndroidLow
rnAndroidHigh_rnAndroidLow_network_result <- wilcox.test(rnAndroidHigh$network,
    ↪ rnAndroidLow$network, alt="greater")
rnAndroidHigh_rnAndroidLow_network_result

## flAndroidHigh > flAndroidLow
flAndroidHigh_flAndroidLow_network_result <- wilcox.test(flAndroidHigh$network,
    ↪ flAndroidLow$network, alt="greater")
flAndroidHigh_flAndroidLow_network_result

## pwaAndroidHigh > pwaAndroidLow
pwaAndroidHigh_pwaAndroidLow_network_result <- wilcox.test(pwaAndroidHigh$network,
    ↪ pwaAndroidLow$network, alt="greater")
pwaAndroidHigh_pwaAndroidLow_network_result
```

```
# ----------------------------------------------------------------------------iOS
    ↪ ---------------------------------------------------------------------------- #
# ------------------------ios-low-end-ct------------------------------#
group_by(dataIosLow,method) %>%
  summarise(
    count = n(),
    median = median(ct, na.rm = T),
    std = sd(ct, na.rm = T))
# ranks: pwaIosLow < nativeIosLow < rnIosLow

## pwaIosLow < nativeIosLow
pwaIosLow_nativeIosLow_result_ct <- wilcox.test(pwaIosLow$ct, nativeIosLow$ct, alt="less"
    ↪ )
pwaIosLow_nativeIosLow_result_ct

## nativeIosLow < rnIosLow
nativeIosLow_rnIosLow_result_ct <- wilcox.test(nativeIosLow$ct, rnIosLow$ct, alt="less")
nativeIosLow_rnIosLow_result_ct

# ------------------------ios-high-end-ct------------------------------#
group_by(dataIosHigh,method) %>%
  summarise(
    count = n(),
    median = median(ct, na.rm = T),
    std = sd(ct, na.rm = T))
# ranks: pwaIosHigh < rnIosHigh < nativeIosHigh

## pwaIosHigh < rnIosHigh
pwaIosHigh_rnIosHigh_result_ct <- wilcox.test(pwaIosHigh$ct, rnIosHigh$ct, alt="less")
pwaIosHigh_rnIosHigh_result_ct

## rnIosHigh < nativeIosHigh
rnIosHigh_nativeIosHigh_result_ct <- wilcox.test(rnIosHigh$ct, nativeIosHigh$ct, alt="
    ↪ less")
rnIosHigh_nativeIosHigh_result_ct

# ------------------------ios-high-end-low-end-ct------------------------------#
## nativeIosHigh < nativeIosLow
nativeIosHigh_nativeIosLow_result_ct <- wilcox.test(nativeIosHigh$ct, nativeIosLow$ct,
    ↪ alt="less")
nativeIosHigh_nativeIosLow_result_ct

## rnIosHigh < rnIosLow
rnIosHigh_rnIosLow_result_ct <- wilcox.test(rnIosHigh$ct, rnIosLow$ct, alt="less")
rnIosHigh_rnIosLow_result_ct

## pwaIosHigh < pwaIosLow
pwaIosHigh_pwaIosLow_result_ct <- wilcox.test(pwaIosHigh$ct, pwaIosLow$ct, alt="less")
pwaIosHigh_pwaIosLow_result_ct

# ------------------------ios-low-end-cpu------------------------------#
group_by(dataIosLow,method) %>%
  summarise(
    count = n(),
```

```
    median = median(cpu, na.rm = T),
    std = sd(cpu, na.rm = T))
# ranks: pwaIosLow < nativeIosLow < rnIosLow

## pwaIosLow < nativeIosLow
pwaIosLow_nativeIosLow_result_cpu <- wilcox.test(pwaIosLow$cpu, nativeIosLow$cpu, alt="
    ↪ less")
pwaIosLow_nativeIosLow_result_cpu

## nativeIosLow < rnIosLow
nativeIosLow_rnIosLow_result_cpu <- wilcox.test(nativeIosLow$cpu, rnIosLow$cpu, alt="less
    ↪ ")
nativeIosLow_rnIosLow_result_cpu

# -------------------------ios-high-end-cpu-------------------------------#
group_by(dataIosHigh,method) %>%
  summarise(
    count = n(),
    median = median(cpu, na.rm = T),
    std = sd(cpu, na.rm = T))
# ranks: pwaIosHigh < nativeIosHigh < rnIosHigh

## pwaIosHigh < nativeIosHigh
pwaIosHigh_nativeIosHigh_result_cpu <- wilcox.test(pwaIosHigh$cpu, nativeIosHigh$cpu, alt
    ↪ ="less")
pwaIosHigh_nativeIosHigh_result_cpu

## nativeIosHigh < rnIosHigh
nativeIosHigh_rnIosHigh_result_cpu <- wilcox.test(pwaIosHigh$cpu, nativeIosHigh$cpu, alt=
    ↪ "less")
nativeIosHigh_rnIosHigh_result_cpu

# -------------------------ios-high-end-low-end-cpu-------------------------------#
## nativeIosHigh < nativeIosLow
nativeIosHigh_nativeIosLow_result_cpu <- wilcox.test(nativeIosHigh$cpu, nativeIosLow$cpu,
    ↪  alt="less")
nativeIosHigh_nativeIosLow_result_cpu

## rnIosHigh < rnIosLow
rnIosHigh_rnIosLow_result_cpu <- wilcox.test(rnIosHigh$cpu, rnIosLow$cpu, alt="less")
rnIosHigh_rnIosLow_result_cpu

## pwaIosHigh < pwaIosLow
pwaIosHigh_pwaIosLow_result_cpu <- wilcox.test(pwaIosHigh$cpu, pwaIosLow$cpu, alt="less")
pwaIosHigh_pwaIosLow_result_cpu

# -------------------------ios-low-end-memory-------------------------------#
group_by(dataIosLow,method) %>%
  summarise(
    count = n(),
    median = median(memory, na.rm = T),
    std = sd(memory, na.rm = T))
# ranks: nativeIosLow < rnIosLow < pwaIosLow
```

```
## nativeIosLow < rnIosLow
nativeIosLow_rnIosLow_result_memory <- wilcox.test(nativeIosLow$memory, rnIosLow$memory,
    ↪ alt="less")
nativeIosLow_rnIosLow_result_memory

## rnIosLow < pwaIosLow
rnIosLow_pwaIosLow_result_memory <- wilcox.test(rnIosLow$memory, pwaIosLow$memory, alt="
    ↪ less")
rnIosLow_pwaIosLow_result_memory

# -------------------------ios-high-end-memory-------------------------------#
group_by(dataIosHigh,method) %>%
  summarise(
    count = n(),
    median = median(memory, na.rm = T),
    std = sd(memory, na.rm = T))
# ranks: nativeIosHigh < rnIosHigh < pwaIosHigh

## nativeIosHigh < rnIosHigh
nativeIosHigh_rnIosHigh_result_memory <- wilcox.test(nativeIosHigh$memory, rnIosHigh$
    ↪ memory, alt="less")
nativeIosHigh_rnIosHigh_result_memory

## rnIosHigh < pwaIosHigh
rnIosHigh_pwaIosHigh_result_memory <- wilcox.test(rnIosHigh$memory, pwaIosHigh$memory,
    ↪ alt="less")
rnIosHigh_pwaIosHigh_result_memory

# -------------------------ios-high-end-low-end-memory-------------------------------#
## nativeIosHigh < nativeIosLow
nativeIosHigh_nativeIosLow_result_memory <- wilcox.test(nativeIosHigh$memory,
    ↪ nativeIosLow$memory, alt="less")
nativeIosHigh_nativeIosLow_result_memory

## rnIosHigh < rnIosLow
rnIosHigh_rnIosLow_result_memory <- wilcox.test(rnIosHigh$memory, rnIosLow$memory, alt="
    ↪ less")
rnIosHigh_rnIosLow_result_memory

## pwaIosHigh > pwaIosLow
pwaIosHigh_pwaIosLow_result_memory <- wilcox.test(pwaIosHigh$memory, pwaIosLow$memory,
    ↪ alt="greater")
pwaIosHigh_pwaIosLow_result_memory

# -------------------------ios-low-end-network-------------------------------#
group_by(dataIosLow,method) %>%
  summarise(
    count = n(),
    median = median(network, na.rm = T),
    std = sd(network, na.rm = T))
# ranks: pwaIosLow < nativeIosLow < rnIosLow

## pwaIosLow < nativeIosLow
pwaIosLow_nativeIosLow_result_network <- wilcox.test(pwaIosLow$network, nativeIosLow$
```

```
    ↪ network, alt="less")
pwaIosLow_nativeIosLow_result_network

## nativeIosLow < rnIosLow
nativeIosLow_rnIosLow_result_network <- wilcox.test(nativeIosLow$network, rnIosLow$
    ↪ network, alt="less")
nativeIosLow_rnIosLow_result_network

# -------------------------ios-high-end-network-------------------------------#
group_by(dataIosHigh,method) %>%
  summarise(
    count = n(),
    median = median(network, na.rm = T),
    std = sd(network, na.rm = T))
# ranks: pwaIosHigh < nativeIosHigh < rnIosHigh

## pwaIosHigh < nativeIosHigh
pwaIosHigh_nativeIosHigh_result_network <- wilcox.test(pwaIosHigh$network, nativeIosHigh$
    ↪ network, alt="less")
pwaIosHigh_nativeIosHigh_result_network

## nativeIosHigh < rnIosHigh
nativeIosHigh_rnIosHigh_result_network <- wilcox.test(nativeIosHigh$network, rnIosHigh$
    ↪ network, alt="less")
nativeIosHigh_rnIosHigh_result_network

# -------------------------ios-high-end-low-end-network-------------------------------#
## nativeIosHigh > nativeIosLow
nativeIosHigh_nativeIosLow_result_network <- wilcox.test(nativeIosHigh$network,
    ↪ nativeIosLow$network, alt="greater")
nativeIosHigh_nativeIosLow_result_network

## rnIosHigh > rnIosLow
rnIosHigh_rnIosLow_result_network <- wilcox.test(rnIosHigh$network, rnIosLow$network, alt
    ↪ ="greater")
rnIosHigh_rnIosLow_result_network

## pwaIosHigh > pwaIosLow
pwaIosHigh_pwaIosLow_result_network <- wilcox.test(pwaIosHigh$network, pwaIosLow$network,
    ↪  alt="greater")
pwaIosHigh_pwaIosLow_result_network
```

Appendix C

Statistical Analysis Code Listing of Developer Experience Experiments

Listing C.1: Statistical analysis code of the build time experiment (in R)

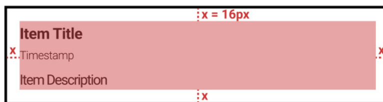
```
library(dplyr)

rawData = read.table("/Users/iman/Desktop/results-developer-experience/buid-time-raw-data
    ↪ .csv", sep = ",", header = TRUE)
rawData$method <- as.factor(rawData$method)

# -----------------------------------------------------------------------------iOS
    ↪ ------------------------------------------#
dataIos <- filter(rawData, method == "rnIos" |
                  method == "flutterIos" |
                  method == "iosNative" |
                  method == "pwa")
group_by(dataIos,method) %>%
  summarise(
    count = n(),
    median = median(ct, na.rm = T),
    std = sd(ct, na.rm = T))

# ranks: iosNative < pwa < flutterIos < rnIos
iosNative <- filter(dataIos, method == "iosNative")
pwa <- filter(dataIos, method == "pwa")
flutterIos <- filter(dataIos, method == "flutterIos")
rnIos <- filter(dataIos, method == "rnIos")

## iosNative < pwa
iosNative_pwa_result <- wilcox.test(iosNative$ct, pwa$ct, alt="less")
iosNative_pwa_result

## pwa < flutterIos
pwa_flutterIos_result <- wilcox.test(pwa$ct, flutterIos$ct, alt="less")
pwa_flutterIos_result

## flutterIos < rnIos
flutterIos_rnIos_result <- wilcox.test(flutterIos$ct, rnIos$ct, alt="less")
flutterIos_rnIos_result


# -----------------------------------------------------------------------------Android
    ↪ ------------------------------------------#
dataAndroid <- filter(rawData, method == "rnAndroid" |
                      method == "flutterAndroid" |
                      method == "androidNative" |
                      method == "pwa")
```

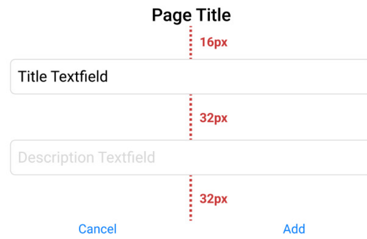
```
group_by(dataAndroid,method) %>%
  summarise(
    count = n(),
    median = median(ct, na.rm = T),
    std = sd(ct, na.rm = T))

# ranks: androidNative < pwa < flutterAndroid < rnAndroid
androidNative <- filter(dataAndroid, method == "androidNative")
pwa <- filter(dataAndroid, method == "pwa")
flutterAndroid <- filter(dataAndroid, method == "flutterAndroid")
rnAndroid <- filter(dataAndroid, method == "rnAndroid")

## androidNative < pwa
androidNative_pwa_result <- wilcox.test(androidNative$ct, pwa$ct, alt="less")
androidNative_pwa_result

## pwa < flutterAndroid
pwa_flutterAndroid_result <- wilcox.test(pwa$ct, flutterAndroid$ct, alt="less")
pwa_flutterAndroid_result

## flutterAndroid < rnAndroid
flutterAndroid_rnAndroid_result <- wilcox.test(flutterAndroid$ct, rnAndroid$ct, alt="less
    ↪ ")
flutterAndroid_rnAndroid_result
```

Appendix D

Listing D.1: The custom LOC (Lines of Code) counter script used for the experiments in the Chapter 6

```
grep -o '[^\t ].*' file | grep -E -v "^import|^}$|^}.*;$|^})$|^)$|^).*;$|^).*,$|^].*,$|^>|^export default|^return \($|^return;$|^return$|^});$|^})}$|^)}$|^},$|^@override$|^@Override$|^@Expose$|^package|^\/\/|^\/\*|^\*|^/>$|^</|^<\!--" | grep -c ^
```



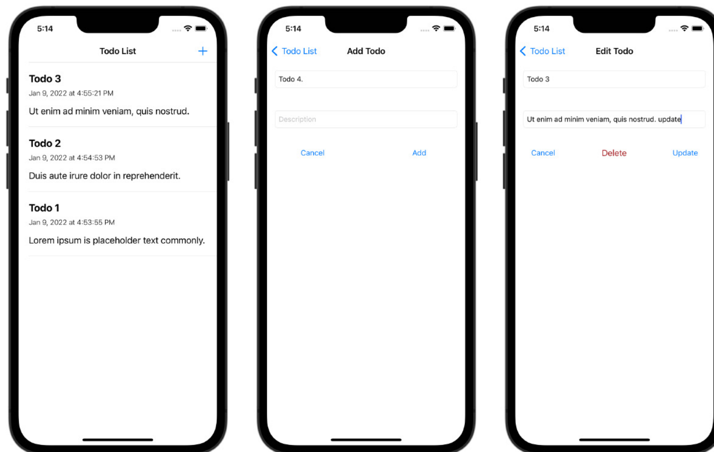


Figure D.1: The style guidelines of the app designed in Chapter 6

File Name	LOC	Cyclomatic Complexity
App.js	6	1
todoApi.js	32	7
Todo.js	6	1
TodoList.js	27	7
TodoList.module.css	26	
EditTodo.js	75	23
EditTodo.module.css	13	

Table D.1: Lines of code and cyclomatic complexity results for the PWA app used in the experiments of Chapter 6

File Name	LOC	Cyclomatic Complexity
App.js	12	1
todoApi.js	32	7
Todo.js	6	1
TodoListPage.js	42	9
EditTodoPage.js	82	22

Table D.2: Lines of code and cyclomatic complexity results for the React Native app used in the experiments of Chapter 6

File Name	LOC	Cyclomatic Complexity
main.dart	8	3
todoApi.dart	40	10
todo.dart	16	3
todolist_page.dart	68	8
edittodo_page.dart	92	18

Table D.3: Lines of code and cyclomatic complexity results for the Flutter app used in the experiments of Chapter 6

File Name	LOC	Cyclomatic Complexity
MainActivity.java	8	2
Todo.java	18	5
TodoService.java	11	0
RetrofitClientInstance.java	10	2
TodoAdapter.java	46	6
TodoListFragment.java	44	11
EditTodoFragment.java	102	37
main_activity.xml	7	
todo_item.xml	36	
todo_list_fragment.xml	26	
edit_todo_fragment.xml	63	

Table D.4: Lines of code and cyclomatic complexity results for the native Android app used in the experiments of Chapter 6

File Name	LOC	Cyclomatic Complexity
API.swift	65	16
Todo.swift	5	0
TodoListTableViewController.swift	57	11
EditTodoViewController.swift	80	26
Main.storyboard	145	

Table D.5: Lines of code and cyclomatic complexity results for the native iOS app used in the experiments of Chapter 6

App	Tool Name: URL
PWA	SonarQube: `https://docs.sonarqube.org/latest/user-guide/metric-definitions/`
React Native	SonarQube: `https://docs.sonarqube.org/latest/user-guide/metric-definitions/`
Flutter	Dart Code Metrics: `https://dartcodemetrics.dev/docs/metrics/cyclomatic-complexity`
Native Android	SonarQube: `https://docs.sonarqube.org/latest/user-guide/metric-definitions/`
Native iOS	Lizard: `https://github.com/terryyin/lizard/blob/master/lizard_languages/swift.py`

Table D.6: Web URL of the tools adopted to measure cyclomatic complexity of the apps in Chapter 6