Presence and Performance of Mobile Development Approaches

A thesis submitted for the degree of Doctor of Philosophy

by

Andreas Biørn-Hansen

Department of Computer Science,
Brunel University London

Supervisors:

Professor Dr. Gheorghita (George) Ghinea, Brunel University London, UK

Professor Dr. Tor-Morten Grønli, Kristiania University College, Norway

November 2020

Abstract

At the centre of the multi-trillion dollar mobile app economy, we find the mobile apps – smartphone-optimised software downloadable from platform marketplaces, including Google's Play Store and Apple's App Store. While the development of these apps has traditionally been conducted on a per-platform basis, using the native platform-specific programming languages and architectures, in this thesis, I explore alternative development approaches for cross-platform mobile apps. Original contributions to knowledge include new empirical insight into the presence and performance of these alternative approaches. The background is an identified lack of empirical studies on these topics, which has spawned claims and allegations across practitioners' outlets and scholarly research. This thesis follows the design science research methodology, focusing on the development and evaluation of mobile apps for performance testing, and scripts to facilitate studying the presence of technical frameworks in published apps. In terms of performance, the findings indicate that while the native development approach provides an overall better performance output, cross-platform frameworks can deliver better performance output in certain situations and for specific hardware metrics. Adding to the complexity is the challenge of significant performance variations between the assessed frameworks' generated Android and iOS apps. As for framework presence in published Android apps, the findings indicate that cross-platform frameworks are present in approximately 15% of the sampled dataset ($n = 661\,705$ apps) with adoption fluctuating between app categories and that the choice of technology and framework has a significant impact on compiled app file size.

Acknowledgements

I wish to first and foremost extend my gratitude to my two supervisors, Professor George Ghinea (Brunel, UK) and Professor Tor-Morten Grønli (KUC, Norway), without whom this thesis and the years as a PhD Fellow (KUC) and Doctoral Researcher (Brunel) would not have been the same. All discussions, whether formal towards a panel or submission, or informal chats and meetings, have always been highly appreciated – I am lucky to have been supervised by the both of you. I also want to thank my research development advisor, Professor Xiaohui Liu (Brunel, UK) for his advice and discussions during my years at Brunel University.

To Oda, thank you for sticking with me through years of late nights, vacations and weekends spent preparing lectures, reviews and revisions, and for all the help and discussions during these years. You're one of a kind.

To my family, parents Ingveig and Lars, sister Torunn and her significant other Jostein, nieces and nephew Julie, Eline and Erik – thank you all.

At Kristiania University College, I wish to thank my colleagues for discussions and support throughout the years, especially the Mobile Technology Lab group, head of department Eivind Brevik, and those I have been fortunate enough to share an office with during these years.

To my friends, thank you for not growing too tired of all the academic talk. A special thanks to Daniel who managed to drag me out of the academic bubble now and then, with beers, games and coding.

November 2020 Biørn-Hansen, A.

List of Publications

Peer-Reviewed Journals

1. 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. *Springer Empirical Software Engineering*, 25(4):2997–3040, June 2020. URL `https://doi.org/10.1007/s10664-020-09827-6`

2. 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*, 51(5), November 2018a. URL `http://doi.org/10.1145/3241739`

3. Andreas Biørn-Hansen, Tor-Morten Grønli, and Gheorghita Ghinea. Animations in cross-platform mobile applications: An evaluation of tools, metrics and performance. *MDPI Sensors*, 19(9), May 2019a. URL `https://doi.org/10.3390/s19092081`

Peer-Reviewed Conferences

1. Andreas Biørn-Hansen, Tor-Morten Grønli, and Gheorghita Ghinea. Cross-platform frameworks in google play store: Trends and directions. In *ACM Symposium on Applied Computing (SAC)*. ACM, (In-review)

2. Andreas Biørn-Hansen and Gheorghita Ghinea. Bridging the gap: Investigating Device-Feature Exposure in Cross-Platform development.

In *Proceedings of the 51st Hawaii International Conference on System Sciences*, pages 5717–5724. ScholarSpace, January 2018. URL http://doi.org/10.24251/HICSS.2018.716

3. Andreas Biørn-Hansen, Tor-Morten Grønli, and Gheorghita Ghinea. Baseline requirements for comparative research on Cross-Platform mobile development: A literature survey. In *Proceedings of the 30th Norwegian Informatics Conference*. Bibsys, November 2017. URL http://ojs.bibsys.no/index.php/NIK/article/view/427

List of Figures

List of Tables

Contents

Abbreviations

ADB Android Debug Bridge

API Application Programming Interface

APK Android Package

B2B Business-to-Business

CLI Command-Line Interface

CPU Central Processing Unit

DEX Dalvik Executable

DSL Domain-Specific Language

DSR Design Science Research

FFI Foreign Function Interface

FPS Frames per Second

GPU Graphics Processing Unit

MDD Model-Driven Development

MSR Mining Software Repositories

RAM Random Access Memory

SDK Software Development Kit

SWE Software Engineering

TTC Time to Completion

(G)UI (Graphical) User Interface

UX User Experience

VM Virtual Machine

Chapter 1

Introduction

This chapter is devoted to providing the reader with an overview of the current smartphone ecosystems alongside introducing mobile software development in general. Subsequently, the research motivation is described, along with the thesis aim, research questions and objectives, followed by a list of contributions to the field of practice and the knowledge base.

1.1 Background

The technological advances in ubiquitous and mobile computing over the last two decades have had an immense impact on society, organisations and people. When Weiser (1991) in his seminal essay proposed the idea of *tabs* as centimetre-sized, wearable computing devices, the Motorola MicroTac Classic had just released. This device was rather contrasting from Weiser's characteristics of a *tab*, and what today's smartphone represents, with its extendable antenna, 10-character display and physical T9 keyboard. In the years to follow, IBM would release the Simon Personal Communicator known as the first smartphone to be publicly released (Sager, 2012), Apple and Palm would start producing personal digital assistants (PDAs), Nokia's line of the popular Communicators was released, and devices such as the touchscreen-enabled Nokia 7710 would continue to push in the direction of Weiser's idea. However, close to one and a half decades would pass before the envisioned *tab* – or what became the *smartphone* – would become ubiquitous. Although a handful of smartphone devices such as the LG Prada were released prior to the launch of the Apple iPhone in 2007, the latter is widely recognised as the beginning of the mainstream smartphone era. Approximately a year after the launch of the iPhone, the first Android-based smartphone was released, the HTC Dream. They both represented a shift away from the older generation of mobile phones and the earliest smartphones, along with PDAs, Communicators and physical QWERTY and T9 keyboards, and towards Weiser's idea of a tab device. It is noteworthy that in terms of advances in performance, the Apple iPhone 11's A13 (GadgetVersus) chip is capable of close to 155 times the FLOPS performance of the 1991 supercomputer `C-DAC PARAM 8000` (Singh, 2007, p. 215) released at the time of Weiser's essay – a testament to technological progress; *"Tabs will also take on functions that no computer performs today." - Weiser (1991, p. 3)*

These days, the smartphone market encompasses an estimated 3,5 billion users worldwide (Statista, 2019), using tens of thousands of distinct device models (OpenSignal, 2015), with capabilities ranging from budget phones to high-end phones with performance equivalent to a laptop or desktop computer. Unsurprisingly, the smartphone has manifested itself as indispensable and ubiquitous in everyday life. Perhaps one of the driving forces behind the widespread adoption is that along with the introduction and proliferation of the smartphone, there followed entire ecosystems and platforms of mobile application software (*apps* for short) – easily downloadable and readily available to the end-user. Apps can range from messaging services to games and other computationally demanding forms of entertainment, to device personalisation and photography, to social media and education platforms, to news and industrial utilities – and anyone with the right skillset can develop and publish an app. These mobile platforms and their accompanying apps are at the core of the thesis at hand, as they undergo empirical evaluation and assessment in terms of performance, alongside scrutiny of underlying development approaches and technologies. Notwithstanding that the latest high-end smartphone devices inherit the performance capabilities of 155 supercomputers from 1991, relevant literature suggests that there are significant differences in app performance depending on the technologies used in the development (Biørn-Hansen et al., 2018a, Dhillon and Mahmoud, 2015, Willocx et al., 2016). The literature further suggests that this might impact the fluidity of user interfaces and response time of integrated platform and device features, for instance, geolocation, sensors and file system operations. These subjects are further investigated in this thesis (Chapters 4 and 5), alongside an analysis of the use – or *presence* – of these technologies in published apps available on the Google Play Store app marketplace (Chapter 6). Investigating these subjects are of importance also outside of following up on suggested future work by fellow researchers. When developing mobile apps, practitioners and decision makers are faced with a large pool of development approaches and technologies to choose from (see Table 2.1), decisions which can ultimately impact such factors as product time to market and development price. These factors can be monumental in the development phase, provided that the global app economy accounts for trillions of U.S. dollars (Statista, 2021) reaching billions of users worldwide.

In terms of the situation on mobile platforms and ecosystems, while some have lasted, others were unable to compete for the enormous and continually growing smartphone user base. The 2010s witnessed the rise and fall of Microsoft's Windows Phone and Windows Mobile ecosystem, the Firefox OS smartphone project, Symbian, MeeGo, Ubuntu Touch and others. Based on usage statistics, it is widely recognised that there are currently two leading mobile platforms as we enter the 2020 decade, operated by Apple Inc. and Google LCC. The latter, Google's Android OS, is the leading mobile platform from a global market-share perspective (70.68%), while Apple's iOS is the second-most adopted mobile platform (28.79%) (StatCounter, 2020). Their market-shares do however vary greatly between regions and countries as reported by StatCounter, where the platforms are closer to equally divided in some Western countries (*e.g.*, United Kingdom and Norway). In terms of devices, while Apple's iOS operating system and encompassing ecosystem are only available through their own line of products, Google's Android ecosystem is available to mobile manufacturers through licensing agreements. This means that while Apple has full control of both software and hardware, thus can optimise both accordingly, Google does not hold the same level of control outside their own product line of mobile devices (*i.e.*, the Google Pixel series). A result of the Android platform and ecosystem being available to other manufacturers is the fragmentation of devices and installed Android versions which have become a challenge for developers and companies seeking to launch apps on the platform (Wei et al., 2016). Indeed, a report from OpenSignal (2015) noted more than 24 000 distinct Android-based device models. Although this leaves us with predominantly two mobile ecosystems, the fragmentation and differences between them are vast and many. From a developer's perspective, the differences between iOS and Android do not stop at device and hardware fragmentation. Each of the mobile platforms imposes specific rules and regulations for developers to follow, alongside guidelines, proprietary and protected app distribution channels – marketplaces, programming languages and design practices. The platforms inherit technical characteristics adding to the complexity of app development, as summarised in Table 1.1 and further detailed in the upcoming Section 1.2.

1.2 Mobile Development

The differences between the Android and iOS platforms are rather vast from the perspectives of developers and the platforms' end-users. This section will elaborate primarily through the lens of the app developer, focusing on the inherent challenges related to developing platform-specific and *cross-platform* apps, *i.e.*, apps developed using techniques which allow for publishing across two or more platforms.

1.2.1 Platform Characteristics

The two leading platforms are heterogenous across most technical characteristics. A summary of how they differ is provided below in Table 1.1, and subsequently elaborated on further.

Table 1.1: Characteristics of the Apple iOS and Google Android platforms.

	Platforms	
	iOS	**Android**
Programming languages	Swift, Objective-C, C	Java, Kotlin, C, C++
User interface	Storyboards, SwiftUI, code	XML, code
Development tool (IDE)	Xcode (or third-party)	Android Studio (or third-party)
Design guidelines	Human Interface Guidelines	Material Design
Marketplace	Apple App Store	Google Play Store

Programming languages and user interface development. Both platforms provide the developer with specific options in terms of supported programming languages. The Android platform supports Java and Kotlin when targeting the Android SDK (Software Development Kit), while the Android NDK (Native Development Kit) adds additional support for C and C++. The Apple iOS platform has had Objective-C as its primary language for many years until the addition of the Swift language in 2014. Apple iOS also has support for C and C++ through Objective-C++. However, although both platforms support C and C++, the underlying APIs and rendering mechanisms differ between Android and iOS, making it infeasible to share business logic and user interface code across platforms primarily using C or C++ through the native SDKs or NDKs.

Development tools and requirements. To execute a build process for an iOS app, the developer must do so on a compatible macOS-based machine with the Xcode IDE and CLI installed. Although iOS apps could be developed on Windows-based machines, the build step and binary compilation of the app must be done on a compatible macOS machine. For Android development, the requirements are considerably less rigid. The Android Studio IDE is installable on Windows, macOS, Linux and Chrome OS, all of which enable Android app development.

Design guidelines. Best practices and patterns for designing interactions and layouts are part of the platform-specific design guidelines. Although the same app may exist on both Android and iOS, the user would expect different interaction patterns and aesthetics based on the platform they use. An Android app should not necessarily look and behave identically to an iOS app due to platform differences and user expectations (O'Sullivan, 2015). The Material Design guidelines dictate best practices for designing Android-based apps. The Human Interface Guidelines govern the same principles for the Apple iOS platform, describing the intricacy of designing well-functioning iOS apps.

Marketplaces. Perhaps the driving forces of the smartphone revolution are the marketplaces that host and distribute apps to end-users, enabling developers to reach out to the platform users. These marketplaces allow for uploading of new apps and distribution of updates to existing apps. There are two primary marketplaces: Google Play Store for the Android ecosystem, and the Apple App Store for the iOS ecosystem. Both marketplaces are considered *walled gardens*, where internal systems and procedures decide which apps and app updates are allowed entry, as well as which existing published apps to be removed from public access. Publishing apps to the Google Play Store is free of charge to the developer, while Apple charge $99 USD per year for distribution of apps through their App Store. To be present in both marketplaces can be of importance based on the targeted market; such cross-marketplace availability is referred to as *multi-homing* (Hyrynsalmi et al., 2016). A significant difference between Android and iOS is access to side-loading of apps, *i.e.*, the process of installing apps that are not necessarily part of the official marketplaces. While Android enables users to execute installable app files, named Android Package - APK (`.apk`) - directly on the phone, for instance through an email attachment or via an Internet download, the iOS ecosystem prohibits such activities. The relatively lax regulations from Google for Android app side-loading have spawned various third-party app marketplaces not governed by Google. Perhaps the most famous third-party marketplace is F-Droid, hosting and distributing approximately 2 000 open-source Android apps (Itzchak Rehberg, 2020). In comparison, Google Play Store hosts an approximate 2 570 000 Android apps, while the Apple App Store has 1 840 000 iOS apps as of Q4 2019 (Statista, 2020). For research purposes, third-party marketplaces and repositories have seen considerable use in related work (Coppola et al., 2019, Martin et al., 2017, Zeng et al., 2019). Such is also the case for the thesis experiment presented in Chapter 6, where apps from the official Google Play Store are harvested through a third-party repository, AndroZoo, developed by the University of Luxembourg (Allix et al., 2016) specifically to enable efficient app store research and analysis.

1.2.2 Technical Perspectives

Consumers and developers alike are thus faced with two incompatible and competing mobile ecosystems. Developing multi-homed apps require knowledge of multiple programming languages, understanding of disparate design guidelines, manage fragmented APIs and OS versions, distribution channels (marketplaces), supported development tools, and supported hardware. An app developed for Android cannot be installed or executed on an iOS-based device, or vice versa, an inherent characteristic of the *native* development approach (Heitkötter et al., 2012a). Native development entails that separate code bases are required for multi-homed apps (*e.g.*, for distribution through both Google Play Store and Apple App Store), as illustrated in Figure 1.1.

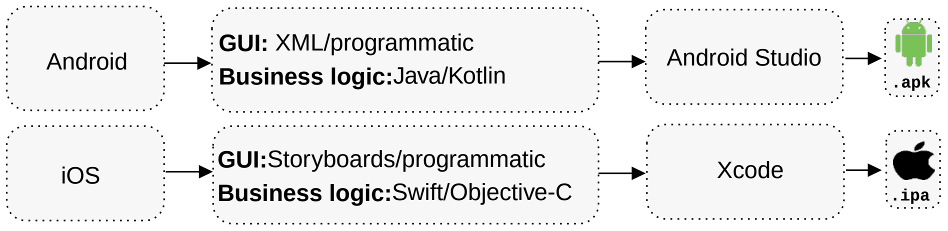

Android
GUI: XML/programmatic
Business logic: Java/Kotlin
Android Studio
.apk
iOS
GUI: Storyboards/programmatic
Business logic: Swift/Objective-C
Xcode
.ipa

Figure 1.1: The platform-specific (native) development approach illustrated.

This lack of interoperability between the leading ecosystems has generated considerable interest from both industry and research (Biørn-Hansen et al., 2018a) in third-party[1] development tools aiding in the creation of

[1]In this context, *third-party* refers to the development tools being developed and supported by other parties than the platform owners, *i.e.*, not by Apple and Google them-

apps executable on both platforms with a high degree of code reuse. This concept is generally referred to as *cross-platform* development. From a holistic perspective, the idea of cross-platform development is to allow developers to target multi-homing as a viable product requirement without dealing with an array of specialised programming- and markup languages (Biørn-Hansen et al., 2018a). Particular tools for developing cross-platform apps can also to a large degree mitigate the need for in-depth implementation experience of platform design guidelines, as the tools take on the responsibility of rendering guideline-adhering user interface elements based on the underlying target platform.

When abstracting away implementation details and technical fundamentals, Figure 1.2 depicts one way of illustrating cross-platform development for mobile. The developer writes code for the graphical user interface and business logic in one or more programming languages usually dictated by the cross-platform tool or framework. For instance, this could be JavaScript, C#, Java, Dart, or Domain-Specific Languages (DSLs). The tool or framework will in turn package the cross-platform code alongside necessary platform-specific resources, and compile the app into a suitable executable file format, for instance `.apk` (Android Package) for Android and `.ipa` (iOS App Store Package) for iOS.

The complexity illustrated in Figure 1.2 increases exponentially with the introduction of implementation details. Such details may include specific programming language(s) for development of business logic and the graphical user interface, the granularity of platform and device control provided

selves.

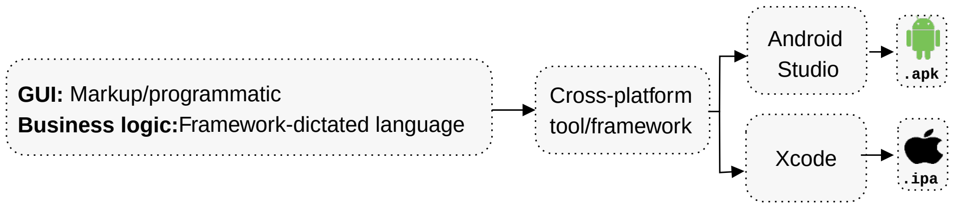

GUI: Markup/programmatic
Business logic: Framework-dictated language
Cross-platform tool/framework
Android Studio
Xcode
.apk
.ipa

Figure 1.2: Holistic workflow illustration of cross-platform development.

by the cross-platform tool or framework, mechanisms for rendering graphical user interfaces, and underlying API access. More in-depth descriptions and illustrations of cross-platform solutions follow in Section 2.1. We can generally place these tools for developing cross-platform apps into an overarching *approach* category, based on how they function and operate on a fundamental level. Section 2.1 further elaborates on implementation details and highlighting differences between the approaches. Adding to the complexity of (cross-platform) mobile development is the vast array of available tools and frameworks aiding in the development process, displayed in Table 2.1 in the literature review, counting over 60 cross-platform tools and frameworks.

1.3 Motivation

The motivation for conducting this research stems from (i) personal interest in unveiling the state of cross-platform development, (ii) suggestions for empirical studies posed by related work and an identified array of unbacked claims and assertions, along with (iii) the increasing industry interest in cross-platform technologies and the potential benefits and drawbacks of such, along with the proliferation in smartphone apps and usage. Each point is further elaborated on below.

Personal motivation. Having a background in software engineering industry targeting mobile and Web apps, I have followed the field from the perspective of a hobby practitioner since 2012, and a professional since 2013 working with Web, native and cross-platform mobile development in Norwegian consultancy companies, startups and for in-house product development teams. The frequent need for publishable apps across several mobile platforms (multi-homing) on limited budgets and with small teams led to an interest in cross-platform development as a potential alternative to the native development approach, and an enabler for rapid application development. It is through industry experience that my interest in cross-platform development grew, and through encountering a frequent notion of dislike and potential misconceptions from industry voices and outlets that my interest towards empirical validation and rigorous research increased further.

Academic motivation. It is suggested in numerous studies to carry out empirical validation of findings, assertions and claims. This concern was explicitly raised for future research by Heitkötter et al. (2012a) in their seminal work comparing cross-platform approaches. Conducting an assessment of performance and user interface responsiveness in cross-platform apps is highlighted as suggestions for further scrutiny by Xanthopoulos and Xinogalos (2013) in their influential comparative study, and again in performance-oriented studies including but not limited to those by Willocx et al. (2016) and Delía et al. (2017). While the assessment of related work in Chapter 2 shows that numerous studies have expanded on these suggestions, there is an identified lack of large-scale studies on performance and framework presence. This thesis is a step towards closing that gap, by providing guiding principles for app development derived from three empirical experiments as further explained in Section 1.4.

Industry motivation. To better understand the industry's perceived issues and challenges with cross-platform mobile development, we surveyed (Biørn-Hansen et al., 2019b) practitioners and developers on their thoughts and experiences with using such technologies. It became evident that the respondents' were primarily concerned with the performance compared to native apps, user experience (UX), the maturity of frameworks and technologies, and user interface development (UI). These results formed the foundation for the work towards this thesis, and will be revisited in Chapters 4, 5 and 6. Additionally, the continuing discourse between cross-platform technologies scrutinised in research and technologies more frequently discussed by industry was another motive and relevance gap identified.

1.4 Aim, Research Questions & Objectives

As can be derived from the thesis motivation and upcoming literature review, there is a notable lack of empirical studies on newer cross-platform mobile development framework performance and app marketplace presence. As a result, a wide array of claims and allegations from both industry and academia is identified, where the existing knowledge base of empirically verified results is inadequate. Consequently, this has spawned the overarching aim and research questions for this PhD thesis and its experiments:

Thesis aim: To provide a set of guiding principles for conducting mobile app development based on results from empirical experiments.

RQ_1: How do apps developed using cross-platform mobile development approaches and associated frameworks perform compared to native mobile apps in terms of hardware and platform utilisation?

- $Sub-RQ_{1.1}$: To what degree do cross-platform mobile development frameworks impose additional performance-related overhead when compared to native mobile development?
- $Sub-RQ_{1.2}$: Do the performance metrics fulfil their purpose in researching animated user interfaces in mobile apps?
- $Sub-RQ_{1.3}$: How well do the official performance insight tools cater to the profiling of animation and transition performance in the cross-platform apps developed?
- $Sub-RQ_{1.4}$: Which of the platforms, iOS or Android, requires the least amount of device- and hardware resources in order to execute and run performant animations and transitions?

RQ_2: How common is the presence of cross-platform development frameworks compared to the native development approach in published mobile apps?

- $Sub-RQ_{2.1}$: What is the distribution of cross-platform development frameworks on the Google Play Store across app categories?
- $Sub-RQ_{2.2}$: How has the use of cross-platform frameworks in deployed apps changed over the last 12 years?

Hypothesis$_1$: Apps developed using the native approach should generate `.apk` files of smaller file size than apps developed using cross-platform development frameworks due to not relying on bundled interpreters, virtual machines or WebView containers.

Lastly, two objectives have been formed based on the above aim and research questions, related to *Performance* and *Presence*, as further detailed in Table 1.2, which details the relevance, objective and purpose of the work conducted, and corresponding thesis chapters.

Table 1.2: Thesis research objectives.

	Objectives *"Empirically assess cross-platform development approaches on their ... "*	
	Performance	**Presence**
Corresponding experiment(s)	Reported in Chapters 4 and 5.	Reported in Chapter 6.
Relevance	Identified lack of newer empirical studies on cross-platform performance, while discussions in industry are plentiful and oftentimes subjective.	Industry discussions on usage (or *presence*) of development approaches and technical frameworks. Lacking an empirical foundation for informed decision making and discussions.
Approach	Empirically assess the performance of technical frameworks aggregated on overarching development approach based on metrics related to hardware and software benchmarking.	Empirically investigate the presence of cross-platform development frameworks in mobile apps published to the Google Play Store marketplace.
Purpose	Highlight potential performance deviation between development approaches and provide a better understanding of performance differences in underlying system components. Derive results to develop empirically validated principles and suggestions for mobile development.	Provide an overview of framework usage in industry, and provide academia and industry with empirical indications to increase scientific relevance in technical discussions and decision making. Derive results to develop empirically validated principles and suggestions for mobile development.

1.5 Contributions

This PhD thesis contributes with the following six components, which are the experiments and their results, and software, data and principles derived from investigating the research questions.

- A set of empirically validated principles and suggestions for (cross-platform) mobile development, derived from the experiments and interpretations inferred from their results and discussions, presented in Section 7.3 (*derived from* RQ_1 *and* RQ_2).

- A large-scale empirical analysis and comparison of the performance of apps developed using cross-platform mobile development frameworks regarding access to- and use of device and platform features *(answers* RQ_1*)*.

- An analysis and comparison of animated user interfaces and their performance in apps developed using cross-platform mobile development frameworks *(answers* RQ_1*)*.

- A large-scale investigation into the presence and trends of cross-platform mobile development frameworks in published Android apps from the Google Play Store *(answers* RQ_2*)*.

- An open-sourced dataset[2] of package names for 99 304 identified cross-platform apps which have been published to the Google Play Store (*derived from* RQ_2).

- Open sourced tools and scripts[2] for data gathering and analysis targeted towards Android apps (*derived from* RQ_1 *and* RQ_2).

[2]Link to tools and scripts repository: `https://github.com/andreasbhansen/phd-thesis-contributions`

1.6 Thesis Structure

Chapter 2 describes the state of research on cross-platform mobile development and further motivates the thesis at hand by highlighting knowledge gaps arising from research and practice.

Chapter 3 dwells on philosophical perspectives, the research methodology and applied research methods.

Chapter 4 describes the experiment targeting the performance of bridge technologies allowing for foreign function invocation or native code invocation in cross-platform technologies.

Chapter 5 describes an experiment exploring the performance of animated elements in user interfaces developed using cross-platform technologies.

Chapter 6 describes an experiment on the presence of cross-platform technologies in published Google Play Store apps.

Chapter 7 discusses the results derived from the three experiments in the context of related work, and compiles the new knowledge into a set of descriptive guiding principles and suggestions for conducting mobile development. Finally, the chapter provides conclusions on the work conducted throughout the thesis, alongside thoughts and suggestions for future work.

Chapter 2

Literature Review

Throughout this chapter, we explore the knowledge base related to native and cross-platform mobile development. The chapter starts by detailing the most frequently encountered approaches and technologies for developing mobile apps. Afterwards, we further investigate existing knowledge on topics kernel to the PhD; user experience, software platform features, performance and app store analysis. Towards the end is a taxonomy of mobile development alongside an executive summary of the state of research.

2.1 Development Approaches

The background section (see Section 1.2) briefly introduced the concept of cross-platform mobile development and the existence of tools, frameworks

Communication of Research: This is an extended version of the publication in the ACM Computing Surveys (2018a). The content and format has been modified to fit the thesis narrative. The literature review has also been updated to reflect the latest advancements to the field.

and categorisation of these into *development approaches*. Most of these approaches are frequently mentioned in the literature (*e.g.*, El-Kassas et al. (2017), Heitkötter et al. (2012a), Smutný (2012)), and a taxonomy has been developed for a consistent language when discussing both with academics and practitioners (see Figure 2.7). Each approach, as further elaborated on below, has its own set of characteristics (El-Kassas et al., 2017). These characteristics are visually presented in Figure 2.7, the taxonomy model constructed to provide a consistent language and shared understanding of the technical aspects of cross-platform development. Technical frameworks for app development can generally belong to a single approach, strictly which depending on such characteristics as further explored both in Table 2.1, and throughout the remainder of this chapter.

The coming subsections introduce the most prevailing cross-platform approaches and associated frameworks, discussing each to a degree where a fundamental understanding of the benefits and challenges they pose should be conceivable. The naming of these approaches tend to vary between authors and studies, but those used throughout the literature review are those most frequently encountered when compared to alternative names, *e.g.*, *"{Native—WebView—Web-to-native} wrapper"* as sometimes seen in-place of *hybrid* (Ribeiro and da Silva, 2012, Willocx et al., 2016).

This is also true for the categorisation of frameworks, *i.e.*, to which approach a framework belongs. An example of this is the NeoMAD framework, which according to Ettifouri et al. (2017) is listed under the cross-compiled approach, while Willocx et al. (2016) argue that it is a source-code translator, the latter an approach not covered by this literature review as it is not commonly encountered in the literature to the best of my knowledge. One framework, specifically MoSync, can be placed into two approaches, which before it was discontinued (MoSync AB, 2015) provided both a C++ based cross-compiled framework, in addition to a JavaScript-based framework for interpreted app development. Each approach poses benefits and challenges, and no single approach is inherently *"best suited for all situations"*. It is important to note that *cross-platform* is an umbrella term for a wide variety of concepts, technologies, approaches, frameworks and libraries. The term is also somewhat context-dependent, meaning that cross-platform development could for instance refer to the development of software across multiple device types, not only mobile, as discussed by Rieger and Majchrzak (2018). Nevertheless, throughout this current literature review, the focus is on mobile smartphones, leaving other smart devices such as cars, smart homes and Internet of Things devices out of scope.

It is important to acknowledge the existence of development approaches not discussed in this review, which covers the hybrid, interpreted, cross-compiled, model-driven and Progressive Web App approaches. These are left out for lack of prevalence in the literature, and no mentions of them in practitioner outlets to the best of my knowledge. Examples of such approaches include Component-Based development (*e.g.*, Escoffier et al. (2015), Perchat et al. (2013)) and Integrated Cross-Platform Mobile Development (El-Kassas et al., 2014), the latter a proposed approach drawing from best practices inherited from other approaches. Notwithstanding their novelty or possible application, no studies and experiments have been identified which include these approaches, except for those previously cited. Thus, not enough data has been generated to sufficiently or with purpose provide an overview of their state of research or use in industry.

In the context of the approaches included in this review, an exhaustive list of identified technical frameworks has been developed. Each framework categorised within an approach best describing their characteristics, and listed in Table 2.1. In the table column *"Hybrid"*, the frameworks using Cordova as their underlying technology, allowing execution of Web-based code on-device, have been marked (*). As the table depicts, the majority of hybrid cross-platform frameworks do rely on Cordova for these tasks, although some outliers exist, namely Capacitor and Trigger.io. Cordova's prevalent position is examined further as part of Section 2.1, detailing the hybrid approach. Furthermore, in the context of Table 2.1, it contains frameworks of all life-cycles, ranging from alpha-stage, through production-ready, to having been discontinued for some time. The purpose of Table 2.1 is to exhaustively introduce cross-platform mobile development frameworks identified in existing research, in industry outlets, and through exploring the field, *i.e.,* regardless of whether a framework is in active development or not. After the table, we continue exploring each development approach in more detail, starting with the hybrid approach.

Table 2.1: An overview of development approaches and an exhaustive list of associated technical frameworks, as published in Biørn-Hansen et al. (2018a, p. 3).

	Development Approaches				
	Hybrid	**Interpreted**	**Cross-compiled**	**Model-Driven**	**Progressive Web Apps**
Technical Frameworks	Cordova *	React Native	Xamarin	MD^2	Ionic
	Capacitor	NativeScript	Flutter	MobML	Zuix
	PhoneGap *	Tabris.js	Codename One	Applause	Mithril
	Ionic *	Fusetools	Xojo Mobile	Mobl	Polymer
	Onsen UI *	MoSync	MonoCross	Mendix	Svelte
	Framework7 *	Titanium	Corona	Appian	Preact
	AppGyver *	Appcelerator	Apportable	MAML	Vue.js
	Quasar Framework *	Adobe AIR	Qt Mobile	Mobia Modeler	Angular
	Sencha Touch *	Smartface Cloud	MoSync	Xmob	React.js
	Intel App Framework *	Weex	RAD Studio (Delphi)	AXIOM	Stencil.js
	Intel XDK *	Kony	Crosslight	MOPPET	Glimmer.js
	RhoMobile	Jasonette	RoboVM	mdsl	Ember.js
	Kony $^?$	LuaView	Marmalade	Automobile	viperHTML
	EvoThings *		Rhodes	WebRatio	Moon.js
	NSB/AppStudio *		DragonRAD	XIS-Mobile	
	Cocoon *			MobDSL	
	Trigger.io				

The Hybrid Approach

This approach allows for the use of standard Web technologies including HTML, CSS, and JavaScript, in an app development context for the implementation of user interfaces, native integrations and business logic. The approach works internally by initialising a new native app project, which includes a WebView component (Latif et al., 2016) as well as code for communication between the WebView and native code (Android Developers, n.d., Gok and Khanna, 2013). As the WebView component is, simply put, an embeddable Web browser (Android Developers, n.d.), it allows for the execution and rendering of HTML, CSS and JavaScript files. These files make up the hybrid app's logic and user interface components, included as part of the app bundle together with the native app project and code. The developer can then point the WebView to render a specific HTML page, programmatically controlling what the component displays to the user (Gok and Khanna, 2013). Accordingly, as a hybrid app developer, one will write the entirety of the front-end and business logic of the app using Web technologies, then have a regular native app wrap and bundle the Web code displaying it through the embedded WebView component. Due to this technique, the hybrid approach is also found referred to as *native-wrapper* (Willocx et al., 2015), as it wraps Web assets into a publishable, deployable native app, installable from any typical mobile app marketplace.

As this requires a fair bit of code and setup, Apache Cordova has become a popular tool and library for initialising new hybrid apps (Willocx et al., 2015). Instead of leaving all the above work to the developer, *i.e.,* the initialisation and setup of WebViews and communication protocols, Cordova will through the use of a command-line tool generate a new native app including a WebView and two-way communication between the WebView and native code (Cordova, n.d.). Such communication referred to as *bridging* (Adinugroho et al., 2015), allows developers to communicate with platform-specific native code from within a non-native environment, such as a JavaScript context. For reference, this technique of executing code between distinct programming languages is also called foreign function interface (FFI). One would typically use bridging and FFIs to, for instance, process or handle tasks deemed too computationally expensive for JavaScript, or to leverage functionality typically only accessible in native environments (Adinugroho et al., 2015, Rieger and Majchrzak, 2016). The bridge is called from the app's JavaScript code, executing a native-code function and potentially returning some value from the native-side back to the JavaScript context, *e.g.,* a GPS coordinate requested from JavaScript fetched from native code. Thus, bridging functionality should be deemed of high importance to ensure a native app-like user experience, as further performance measured in Chapter 4.

In addition to providing trivial hybrid app initialisation, Cordova also provides a plugin system with thousands of available plugins, including camera access, GPS access and contact list access, features requiring the aforementioned bridging system to operate (Corral et al., 2012b). The plugin system also provides a standardised method for the hybrid app developer community to contribute with additional plugins (Biørn-Hansen and Ghinea, 2018), at least for the hybrid frameworks building on Cordova. Since the Cordova library primarily provides the communication layer and command-line tools for (most) of the hybrid app development frameworks (ref. Table 2.1), additional tools and libraries are used to create native-like and native-feeling user interfaces and interactions (Latif et al., 2016). Previous research and search engine queries have identified a wide range of such libraries. Examples include Ionic[1], Framework7[2], Onsen UI[3] and Sencha Touch[4]. They all have in common the focus of facilitating the development of user interfaces for hybrid apps (*e.g.,* Latif et al. (2016)). Because a hybrid app is a Website presented inside a native app through the use of a WebView component, the user interface may look as native-app-like or non-native-app-like as one may wish (Griffith, 2017), depending on the CSS styling. However, developing native-app-like user interfaces which adhere to the interface guidelines of all supported platforms, *e.g.,* Android Material Design and Apple Human Interface Guidelines, may be challenging and time-consuming to do from scratch (Gok and Khanna, 2013, p. 7). This is the power of user interface libraries such as those mentioned above, as they mimic the look of native app user interface components using HTML, CSS and JavaScript (Gronli et al., 2014).

Because the files all contain standard Web technology code, they work in every (embeddable) browser (Adinugroho et al., 2015) and allow for re-use of existing knowledge for Web developers. This reason has made the hybrid approach highly popular amongst cross-platform developers (Ali and Mesbah, 2016, Malavolta et al., 2015a). Because a hybrid app has a native app as its foundation, it can be submitted into app stores the same way as an app developed using the native development approach, and thus differs

[1]http://ionicframework.com/
[2]https://framework7.io/
[3]https://onsen.io/
[4]https://www.sencha.com/products/touch/

significantly from a regular Website or Web app which for now is constrained to the executing browser's implemented device and platform APIs.

The illustration in Figure 2.1 provides an overview of how the hybrid approach works, where Cordova is the controlling entity, managing the execution, rendering and packaging of the app's user interface (HTML, CSS) and the business logic (JavaScript). An alternative to Cordova, named Capacitor, is being developed by the Ionic team. Thus, although the below illustration accurately depicts state of the art in hybrid app development at the time of writing this review, specific frameworks, thereof Ionic, are now incorporating Capacitor as the default communication layer library.

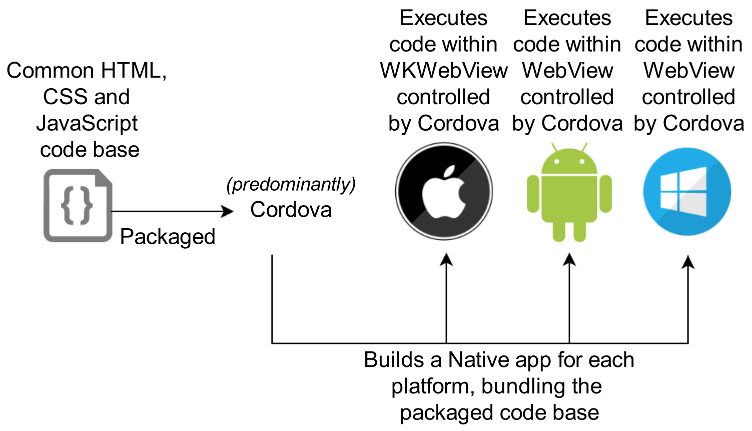

Common HTML, CSS and JavaScript code base
Packaged
(predominantly) Cordova
Executes code within WKWebView controlled by Cordova
Executes code within WebView controlled by Cordova
Executes code within WebView controlled by Cordova
Builds a Native app for each platform, bundling the packaged code base

Figure 2.1: Overview of the hybrid approach build workflow.

The Interpreted Approach

Similar to the hybrid approach previously described, it is common to find development frameworks of the interpreted approach enabling developers to build their apps using the JavaScript language (*e.g.*, Facebook (2018), Telerik (n.d.)), although JavaScript is not a language inherent to the interpreted approach. Apps developed using the interpreted approach are fundamentally different from hybrid apps, as interpreted apps do not rely on a WebView component to render a bundled Website (El-Kassas et al., 2017) (see Figure 2.2). Instead, interpreted apps can render actual native user interface components to the screen, not HTML- and CSS-based views (Dhillon and Mahmoud, 2015), although there are examples of interpreted tools which do not have this as a goal (*e.g.,* the Unity game engine). This works through the use of on-device JavaScript interpreters (Majchrzak et al., 2017), hence the naming of the approach. In terms of code interpreters, JavaScriptCore is the default interpreter on iOS devices (Alcocer, 2013, Facebook, 2017). On Android devices, the interpreter in use differs between frameworks belonging to the approach, but JavaScriptCore and V8 are both frequently used engines (*e.g.,* Alcocer (2013), Facebook (2017)), alongside the newer React Native-based engine Hermes.

Specific frameworks of the interpreted approach, such as Titanium Appcelerator, are occasionally incorrectly associated with the cross-compiled approach (*e.g.,* Escoffier et al. (2015)). While both approaches generate native user interfaces, the interpreted approach does not compile, convert or transpile the codebase into native byte code, which is how the cross-compiled approach works. Indeed, as documented by the Titanium framework, a JavaScript interpreter is required as a layer of abstraction (Appcelerator, n.d.), making Titanium a framework of the interpreted approach.

In order to communicate between the JavaScript layer and native code layer which has access to native device features, the interpreted approach also employs the technique of bridging (with foreign function interfaces), similar to how the hybrid approach facilitate such access (Biørn-Hansen and Ghinea, 2018). An interesting note is that a *"Cordova for interpreted apps"* has yet to be developed, meaning frameworks of the interpreted approach lack the common foundation found in hybrid apps. Thus, plugins or modules for exposing certain features to JavaScript from the native environments belonging to one interpreted framework would not work out of the box in another framework due to differences in implementation and bridging APIs. Examples of such inoperability include plugins in technical frameworks such as Facebook's React Native and Telerik's NativeScript. While they both belong to the same development approach, the underlying framework APIs are of such different nature that a plugin or module developed for one framework cannot currently work in the other. This inoperability fragments frameworks and developer communities of the interpreted approach significantly, whereas, for hybrid development, all frameworks building on top of Cordova could theoretically use the same Cordova plugins.

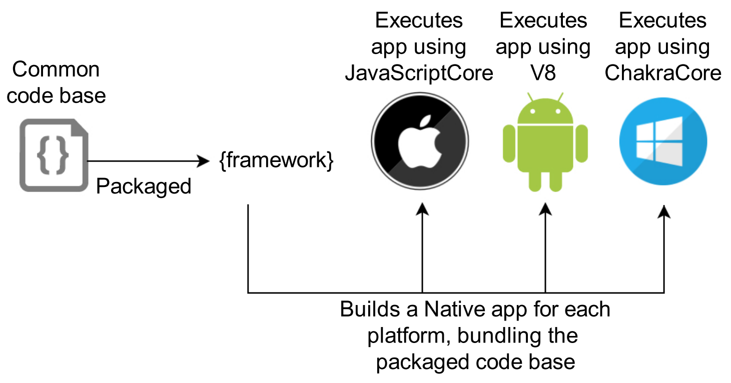

Common code base
Packaged
{framework}
Executes app using JavaScriptCore
Executes app using V8
Executes app using ChakraCore
Builds a Native app for each platform, bundling the packaged code base

Figure 2.2: Overview of the interpreted approach build workflow.

The Cross-compiled Approach

A core difference between the cross-compiled approach and the aforementioned approaches is that due to being compilers, frameworks and development tools of the cross-compiled approach do not rely on WebView components or on-device (JavaScript) interpreters for the rendering of user interfaces or communication with the underlying platform and device features. Instead, a common language such as C# (Xamarin) is compiled to native byte code executable on targeted platforms (Ciman and Gaggi, 2016) (see Figure 2.3). Thus, the bridging layer known from, for instance, the interpreted and hybrid approaches does not exist in cross-compiled apps. Neither the use of- nor the access to native device features is controlled by such a layer but is exposed rather to the app developer through the framework Software Development Kit (SDK), which accordingly maps functionality to the underlying platforms' SDKs. Another positive consequence of compiling to native byte code is that they render native user interface components (Willocx et al., 2015).

There is a lack of consensus regarding which frameworks belong in the cross-compiled approach. Indeed, the approach is more of a "catch-all" category for frameworks and technologies not belonging to the other approaches. An example of disagreement includes a study by El-Kassas et al. (2017), claiming that the Xamarin cross-platform tool belongs to the *interpreted approach*, together with, for instance, Titanium Appcelerator. On the other hand, Willocx et al. (2015) claim Xamarin belongs to the cross-compiled approach due to how it does not rely on interpreters, as it instead compiles a common language into native byte code (Xamarin, 2017). As such, Xamarin is throughout this thesis considered a tool of the cross-compiled approach, due to Willocx *et al.*'s reasoning.

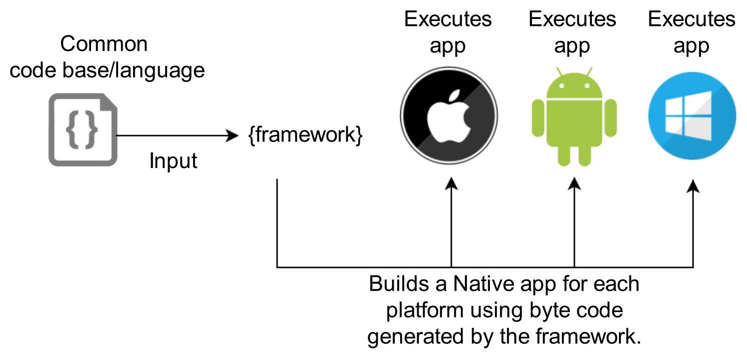

Common
code base/language
Input
{framework}
Executes app
Executes app
Executes app
Builds a Native app for each platform using byte code generated by the framework.

Figure 2.3: Overview of the cross-compiled approach build workflow.

The Model-Driven Approach

This approach draws from the common software development paradigm *Model-Driven Development* (MDD), also referred to as *Model-Driven Software Development* (MDSD) (Heitkötter et al., 2013). While being a somewhat commonplace approach in the relevant body of knowledge (*e.g.,* Heitkötter et al. (2013), Krainz et al. (2016), Ribeiro and Araújo (2016), Rieger (2018)), technical implementations building on the MDD approach are seemingly rare among practitioners and in developer communities, also beyond the context of (cross-platform) mobile development (Gorschek et al., 2014). This lack of use is partially confirmed by Umuhoza and Brambilla (2016) in their survey on MDD approaches for cross-platform mobile development. They categorise technical tools and frameworks into *Research Approaches* and *Commercial Solutions*, where the latter category only lists four tools, none of which frequently cited in the previous literature to the best of my knowledge.

Frameworks of the MDD approach differ in terms of integrated functionality, as discussed by Heitkötter and Majchrzak (2013). The technical framework MD^2, as presented in their study, does not require any knowledge of platform-specific programming languages. According to Ribeiro and da Silva (2012) in their study on cross-platform approaches, these types of abstractions are part of the underlying methodology of MDD development. Frameworks of the MDD approach facilitate the generation of user interfaces and business logic based on constructed models and templates, suitable for mobile app development as they *"[...] allow[s] platform independent modeling, which can later on be transformed to multiple mobile platforms"*, as described by Usman et al. (2017, p. 2).

Common for MDD frameworks is the use of Domain-Specific Languages (DSLs) (Xanthopoulos and Xinogalos, 2013). Developers and non-developers alike are enabled by the framework-provided DSL to build their software. Thus, developing apps across mobile platforms will require knowledge of the DSL rather than Objective-C and Java. Generators will then convert the models/code into native source code for the targeted platforms (Heitkötter and Majchrzak, 2013) (see Figure 2.4). From a developer perspective, most MDD frameworks develop and expose distinct DSLs (Umuhoza and Brambilla, 2016), as illustrated in Le Goaer and Waltham (2013)'s study titled *"Yet Another DSL for Cross-platforms Mobile Development"* in which the authors propose the *Xmob* DSL for mobile development. The distinct nature of the DSLs may render knowledge and code transferability from one MDD-based framework to another low or non-existent. However, MDD framework super-users with expert knowledge of the DSL may develop genuinely native apps using shared codebases (Heitkötter and Majchrzak, 2013). Indeed, one of the philosophies behind the model-driven approach is enabling non-developers and non-technical users with domain expertise to model line-of-business apps based on a provided textual or graphical DSL (Rieger, 2018).

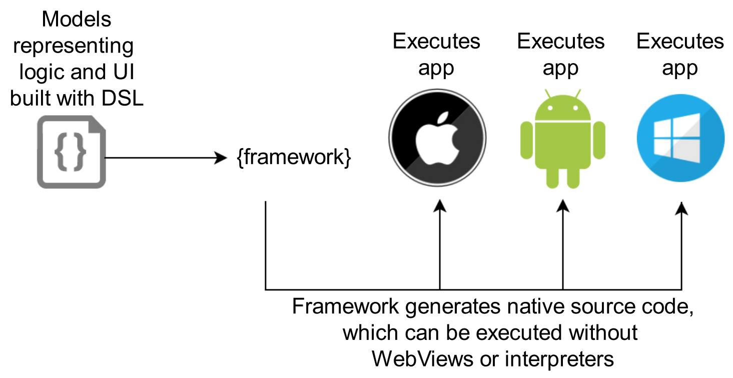

Models representing logic and UI built with DSL
{framework}
Executes app
Executes app
Executes app
Framework generates native source code, which can be executed without WebViews or interpreters

Figure 2.4: Overview of the model-driven development approach build workflow.

The Progressive Web Apps Approach

While still lacking some inherent characteristics of other cross-platform approaches (for instance access to significant device and platform features (Biørn-Hansen et al., 2018b)), Progressive Web Apps – or PWAs for short – have gained popularity amongst practitioners since 2016 (Biørn-Hansen et al., 2018b). At its core, a PWA is a Web app with enhanced capabilities. While being hosted on- and served by- a Web-server to users accessing the Website's URL in a browser, one goal of this novel approach is to allow for Web apps to look and feel like a regular native or cross-platform built app (see Figure 2.5). Due to being Web-based, PWA user interfaces can be designed to look and feel similar to native apps using the same methods as in the hybrid approach, specifically through the use of HTML and CSS for structure and style.

When accessing a PWA-enabled Website, a banner will prompt the user asking them to install the Website onto their phone. Accepting the prompt will download all necessary assets, including JavaScript files, HTML and CSS, images and fonts, and allow for offline use of the Website. The now-offline PWA is added to the user's home screen, similar to any app downloaded from app stores. In Apple's iOS version 11.3, the underlying technology enabling PWAs to function, *i.e.,* Service Workers, has been added support for, allowing iOS users to take advantage of Progressive Web Apps, although with some technical limitations, *e.g.,* lack of state management between sessions, and white screens in the App Switcher, according to Firtman (2018b). Upon launching the PWA from the home screen, an artefact-less browser window will open the previously downloaded assets, and read a manifest file to configure a splash screen, icons and colour configurations. In this context, being artefact-less means no address bar, browser settings (icon) or similar are displayed, only the Website's content – effectively meaning that unless the user knows about the concepts of PWA, they will not know that the app is running within a browser. This can significantly increase the app-like feeling of using a PWA compared to a regular Website browsed using traditional means.

Extending a regular Web app into a Progressive Web App involves the integration of Service Workers, a JSON-based manifest file and a bundle of

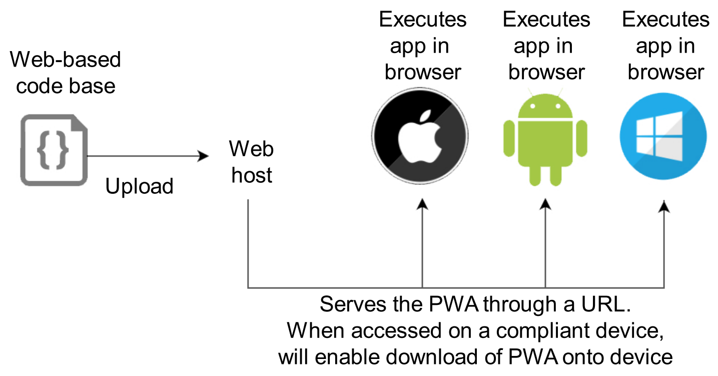

Web-based
code base
Upload
Web
host
Executes
app in
browser
Executes
app in
browser
Executes
app in
browser
Serves the PWA through a URL.
When accessed on a compliant device,
will enable download of PWA onto device

Figure 2.5: Overview of the Progressive Web Apps approach build workflow.

static user interface components not dependant on dynamic content referred to as Application Shell (Biørn-Hansen et al., 2018b). The purpose of the Service Worker is to control and manage the app's lifecycle, business logic for data synchronisation and handling of push notifications, all through a script file written in JavaScript that can execute functionality while the app is in the background or not running on the device (Gaunt, 2018).

The academic body of knowledge on Progressive Web Apps is limited to only a handful of published works (*e.g.,* Biørn-Hansen et al. (2018b), Chan-Jong-Chu et al. (2020), Majchrzak et al. (2018), Malavolta et al. (2017, 2020)). These publications, along with practitioners' outlets such as Google Web Fundamentals and the Google I/O conference, would indeed act as the foundation for further scholarly research. Along with the advent of Progressive Web Apps, research involving technical assessments, conceptual discussions and case studies would be a natural step towards including PWAs in mobile and Web computing research.

2.2 The Research Foundation

While a solid, although ageing, theoretical foundation on cross-platform mobile development is identified, new development frameworks and approaches seeking, for instance, to bring native user experiences to cross-platform apps have emerged and rapidly gained popularity over the last few years (Biørn-Hansen et al., 2018b). When Dalmasso et al. (2013, p. 1) stated that cross-platform apps were “Not as good as Native apps” in terms of user experience and that the quality of such apps ranges between “Medium to low”, these results, due to the age of the study and the dynamic nature of the field, might not reflect the current state of the art. Section 2.3 further explores and discusses studies related to the users’ perspective, as results from newer research have the potential to contradict Dalmasso *et al.*’s findings, a testament to the fast-paced innovation within this field of practice and research.

The literature review at hand is not the first taxonomy or survey on mobile application development, nor is it hopefully the last. A study by Rieger and Majchrzak (2018) explores the vast landscape of app-enabled devices, in which they generated a taxonomy based on dimensions related to media input, output, and device mobility. The nature of their taxonomy is distinctly different from that on hand. While their taxonomy is based on a categorisation of all app-enabled hardware focuses on the aforementioned dimensions, the focus of this literature review is to investigate the state of research on cross-platform app development and discuss areas within the field in need of further work. Nevertheless, there most certainly is an overarching topical link between the taxonomies, being that of *(mobile) apps.* Also identified was the taxonomy by El-Kassas et al. (2017) for cross-platform mobile development. The authors describe and illustrate low-level differences between development approaches in a technical matter, thus a study recommended for those interested in a taxonomy of a more software engineering-oriented fashion compared to the review at hand. Thus, whereas El-Kassas et al. (2017) provide, for instance, code-supported pros and cons of an array of development approaches, as previously mentioned, the aim of this literature review is instead to provide an overview of the state of the art and research.

Several comprehensive studies have been identified as part of the literature search process. One example thereof is Heitkötter et al. (2012b)'s seminal paper on cross-platform approach evaluation, from the early days of cross-platform mobile development research. They present 14 criteria for different perspectives of the development process and proceed to map their results and findings to said criteria. They go into topics such as licensing, platform support, user experience, maintainability and ease of development. They presented possibilities for future work such as empirically verifying their results and ensuring that academia stays up to date with the technological progress in the field. The latter suggestion has been followed up by academia to a varying degree. While cross-platform development is still actively researched, newer studies tend to perhaps draw too much from previous research rather than including innovative and novel technologies in new experiments.

Due to the overarching and broad-covering nature of the seminal paper by Heitkötter et al. (2012b), it does not go in-depth into any of its presented topics or evaluation criteria. The authors predominantly scratch the surface of what they present, but they do lay a foundation for what is important when researching the field. The topics presented could all be of relevance for in-depth research, and the value of the contribution increases accordingly. Due to its impact on the field of research, Rieger and Majchrzak (2016) published a revised version seeking to extend the "cross-platform" term into other contexts such as smart homes, smart cars and smart TVs. By doing so, they have established a foundation for research on cutting-edge novel technologies and use-cases. Nevertheless, researching the suggested contexts is a task for future research, as it extends beyond the scope, motivation and purpose of this thesis.

Furthermore, a variety of studies are from the earlier days of cross-platform research. Perhaps one of the earliest works identified was that of Charland and LeRoux (2011). Theirs is a particularly interesting contribution, as both authors were involved in the creation of the PhoneGap framework during its early days, a commercial distribution of the open-source Cordova framework. Their perspectives may thus differ from practitioners and academics. In their paper, Charland and Leroux discuss differences in how native and Web apps were developed in 2011. An interesting observation is that very little has changed since the paper was published. While development tooling and environments have inherently progressed and become more advanced, Web apps are still predominantly developed using the same languages (HTML, CSS and JavaScript), although WebAssembly is starting to allow for non-traditional languages in front-end and full-stack Web development. Native apps are also still developed the same way – and may still account for most apps in the app stores (Viennot et al., 2014), platform- and device fragmentation (especially on Android) is still very much true (Grønli and Ghinea, 2016, Willocx et al., 2015), and platform conventions (design, user experience) are still a widely discussed subject when debating native versus cross-platform development (Angulo and Ferre, 2014, Mercado et al., 2016). All this makes for possible future research contributions of significant impact. The fact that problem areas identified at the beginning of cross-platform development are still very much tangible and real today shows that not enough research effort has gone towards developing real solutions. The coming section discusses one such frequently mentioned problem area, specifically user experience and perceived quality of cross-platform developed apps.

2.3 User Experience

Are developers biased in their decisions on cross-platform versus native development? Will their decisions inherently be coloured by the fact that they work towards a goal of well-optimised and well-performing apps and user interfaces, regardless of actual user perception? Can end-users notice the difference between cross-platform and native apps, and if they do, does it matter? These questions are interesting in the context of this thesis, as it is common in both academia and industry to find critical and often unsupported claims regarding user experience in cross-platform apps, even regardless of overarching research questions. An example of such is a statement from a well-cited study by Dalmasso et al. (2013, p. 324), reporting that when it comes to the development of user interfaces, cross-platform frameworks and tools cannot provide interface elements comparable to those found in native apps. They also follow up with a statement on the importance of rich user interfaces, and how cross-platform frameworks should indeed implement such.

A study by Boushehrinejadmoradi et al. (2015, p. 441) also claims that the hybrid approach and its associated frameworks will generate apps that are inherently subpar compared to native apps for certain types of app categories, but they do not provide any empirical evidence to back their claim. Similar claims form the basis of some cross-platform approach decision frameworks, thereof studies by Shah et al. (2019), Latif et al. (2016) and Lachgar and Abdali (2017), reporting that user interfaces generated by the hybrid approach are inferior to those in native apps due to execution in the WebView – although they provide no empirical evidence thereof.

The few studies identified that focus on the end-user perception of cross-platform apps employs mostly quantitative methods for data gathering. While one study relies on data from laboratory and longitudinal research methods (Angulo and Ferre, 2014), others focus on mining and analysing quantitative data in the form of app store reviews (Ali and Mesbah, 2016, Malavolta et al., 2015b, Mercado et al., 2016), some through the use of Natural-Language Processing (Mercado et al., 2016). Caulo et al. (2019) interviewed 18 participants to better understand the potential impact and benefits of migrating a hybrid app to a native Android app, finding that the participants liked the migrated Android app better than the hybrid app. These cited studies make up the identified body of knowledge on research on the *user's perspective* of cross-platform apps. Thus, it is evident that a topic this commonly discussed requires additional research and perhaps a more comprehensive array of research methods, for instance, using advanced equipment, including eye trackers and brain-computer interfaces.

A different approach to testing user interfaces was conducted by Huber et al. (2020), in a recent study on the impact of in-app interactions on memory (`RAM`) consumption, `CPU` usage, janky frame rendering and `GPU` usage. Their findings indicate a varying impact on all metrics as they compare a baseline native Android app to React Native and Ionic-Capacitor implementations. For instance, React Native consumes more `CPU` than both the native implementation and the Ionic-Capacitor implementation, and Ionic-Capacitor more than native. In terms of `RAM` usage, the cross-platform frameworks are substantially more requiring across all tests and devices, in one case (React Native) using more than 28 times the memory compared to the native implementation. Frame rendering is found to be inconsistently janky across the implementations, with Ionic-Capacitor rendering the most janky frames. Huber et al. (2020)'s findings are of great importance to understand the trade-offs involved in deciding on a technology and development approach. Assessing `RAM` and `CPU` consumption is followed up in Chapters 4 and 5.

The studies incorporating mining and analysis of app store reviews display interesting quantitative results, but alas, such studies are far between. Nevertheless, an experiment of this kind has been conducted by Ali and Mesbah (2016), presenting a rating scale named *Aggregated User-perceived Rating (AUR)* to attempt to rate user perception based on an app's star rating and the number of reviews. They find that the much-criticised hybrid approach, in their paper represented by the PhoneGap framework, scores the highest on the AUR scale (higher is better), compared to the Titanium Appcelerator and Adobe AIR frameworks, both of which are frameworks of the interpreted approach. The authors also found that within specific app categories such as comics, business, entertainment and finance, hybrid apps received a greater AUR than native apps in the same categories. The authors conclude their study by stating that hybrid apps can be of such quality that they may provide native app-like experiences (Ali and Mesbah, 2016, p. 54).

Their results concur with Malavolta et al. (2015b)'s study on end-user perception of Google Play Store (Android) apps, following much of the same research approach and method for measuring perceived app performance. Indeed, cross-platform apps published to some categories continually score higher than native apps in the same categories – *e.g.,* in categories such as business, medical and lifestyle. These findings also match those presented by Ali and Mesbah (2016). However, both the entertainment and music & video categories vary significantly between the studies. In Ali and Mesbah (2016)'s study, both categories contain seemingly better hybrid apps than native ones. These findings are in stark contrast with results published by Malavolta et al. (2015b). One explanation could be that there were notable variations in the data sets. A total of 3 041 315 reviews from 11 917 apps were the basis of the Malavolta et al. (2015b) study, while 9 948 hybrid and native apps, totalling 19 896 apps, were analysed by Ali and Mesbah (2016). Another explanation for the differences between the studies could be the formula used to calculate rating.

Using Natural-Language Processing to analyse app reviews from Google Play Store and Apple's App Store, Mercado et al.'s (2016) study on development approach choice's impact on end-user experience fills an essential space in the knowledge body. It complements previous studies (Ali and Mesbah, 2016, Malavolta et al., 2015b) with a language processing strategy and model rather than formulas for star reviews and review counts. Their findings indicate that end-users perceive apps developed in different cross-platform approaches differently and that users leave reviews regarding their varying experiences with such apps (Mercado et al., 2016, p. 48). It would be of interest to know if the apps receiving negative user reviews belong in either of the outlier categories as discussed by Ali and Mesbah (2016) and Malavolta et al. (2015b). From a technical perspective, conducting code reviews of the mentioned hybrid apps identified by Mercado et al. (2016) could help answer *why* the apps were rated so poorly. No academic literature has been identified on code-wise optimisation of hybrid apps, thus looking to industry literature makes more sense. Books such as *High Performance Mobile Web* (Firtman, 2016) and *Hacking Web Performance* (Firtman, 2018a) could help in creating a code review baseline for both Progressive Web Apps and apps of the hybrid approach. Nevertheless, the findings quoted above are highly relevant for academia and industry and fill a gap in which much of the previously published work discusses unsupported claims.

In the qualitative research space, only one study of scale directly involving end-users has been identified. A study by Angulo and Ferre (2014) involves both laboratory and longitudinal studies on the end-user perception of cross-platform app quality. To rigorously evaluate and compare such quality, the researchers developed four apps, including two interpreted-approach apps (one for Android and one for iOS) as well a native app for each platform to understand the baseline quality expected. Method-wise, they employed the System Usability Scale for their questionnaire, a tool considered an industry-standard (Usability.gov, 2013). Results indicated more scepticism amongst iOS users towards non-native apps than among Android users (Angulo and Ferre, 2014, p. 7). In their tests, 91% of Android users and 79% of iOS users found the cross-platform app to behave as- or similar to the native baseline app. Their study is the only one identified that leverages not only quantitative methods (*e.g.,* app review analysis), but focuses more on user involvement. Their conclusion, as cited below, is a step towards disconfirming previous studies' unsupported claims regarding user experience in cross-platform apps.

> *"[...] a good level of UX can be obtained if the cross-platform development framework is chosen carefully in terms of providing adapted interaction styles for each platform, and the development team has UX expertise. But there are more possibilities of getting a better UX by maintaining the control over interaction issues that provides the development of an app with native code." - Angulo and Ferre (2014, p. 8)*

Alas, the study did not include any cross-platform apps of the hybrid, cross-compiled or model-driven approaches, the former often more commonly criticised than the others due to depending on the WebView component (*e.g.,* Boushehrinejadmoradi et al. (2015), Latif et al. (2016)). Nevertheless, this only confirms the need for more user-oriented research on cross-platform apps. A study that does assess the user experience of a hybrid PhoneGap app, although with a limited group of subjects, is the seminal paper by Heitkötter et al. (2012a). While they have not conducted any user experiments to scrutinise their implementation objectively, the authors discusses their own experience using the hybrid app, reporting that they found the app to be as responsive as- and to provide comparable performance to- a native application (Heitkötter et al., 2012a, p. 304). Nonetheless, the possibilities of bias in their reporting, further evaluating similar implementations – though with a subjective group of test subjects – can help in clarifying how users perceive cross-platform apps.

2.4 Software Platform Features

In their seminal work, Heitkötter et al. (2012b) state that device and platform feature access would typically include the camera, GPS, push notifications and similar functionality belonging either to the platform itself (Android or iOS operating system) or the devices' hardware, to which the platform SDKs provide access. Having programmatic access to device and platform features is a requirement typically listed as part of cross-platform framework evaluations, and it is common to find comparative studies focusing on the differences between technical frameworks in terms of the number of programmatically accessible features (Heitkötter et al., 2012a). The importance of such access should be paramount to any app development approach, as a (cross-platform) app without access to device and platform features would have significant functional limitations compared to native apps (Luo et al., 2011). Access to these types of features is also commonly listed as a fundamental requirement in decision frameworks targeting the choice of development approach and technical frameworks, examples being that of Latif et al. (2016)'s study, stating that cross-platform frameworks must provide access to all of the features available.

Figure 2.6 shows the connection between the cross-platform app, the intermediary abstraction layer providing programmatic access to device features, and some examples of such features available through the platform and device. What the intermediary layer is and how it works, depend on the development approach. Whereas the hybrid, Progressive Web App and interpreted approaches all will depend on an abstraction layer during runtime, the cross-compiled and model-driven approaches typically do not, as the apps compiled using these approaches will have direct access to the underlying system and platform.

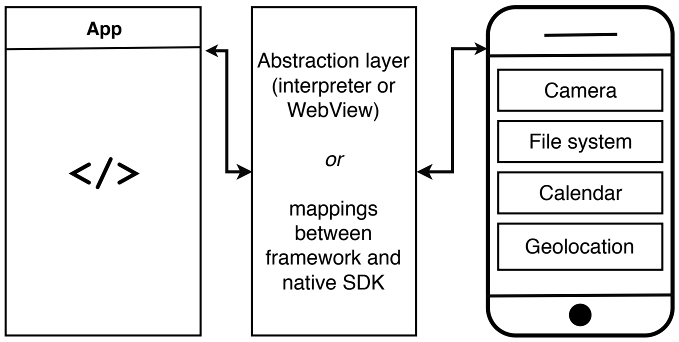

App
</>
Abstraction layer (interpreter or WebView)
or
mappings between framework and native SDK
Camera
File system
Calendar
Geolocation

Figure 2.6: Illustration of connection between app and device-platform features.

The efficiency and ease of device feature access in hybrid and interpreted apps have previously been assessed (Biørn-Hansen and Ghinea, 2018). Both approaches were found to do well at providing easy programmatic access to native features, and no features were identified which could not be integrated and executed from within a cross-platform environment, whether the bridging technology was the interpreted or hybrid approach. This paper is potentially a point of departure for disconfirming previous claims on the subject, *e.g.,* a study from Corral *et al.* discussing advantages and constraints of cross-platform development. In their 2012 paper, they state that cross-platform frameworks do not fully cater programmatic access to certain features (Corral et al., 2012a, p. 1205), without emphasising which features that are commonly missing. Nevertheless, home screen widgets, or app-widgets which can be pinned to the user's home screen, can at the time of writing not be developed using hybrid or interpreted frameworks without implementing this in native code.

A study by Palmieri et al. (2012) compares five major cross-platform development frameworks of different approaches on a variety of parameters, including feature access. Their findings indicate that, while the statement from Corral et al. (2012a) did indeed to a degree report correctly on the then status-quo of feature access, the level of access provided by the frameworks varied greatly. While the hybrid approach, in the paper represented by PhoneGap, provided APIs to all but one of the compared features, the DragonRad framework supported only close to half of the features listed. The same two frameworks are also scrutinised in a study by Ribeiro and da Silva (2012). However, in this study, the list of features used for comparison is less exhaustive than in Palmieri et al. (2012)'s investigation, creating a misconception that DragonRAD can access the majority of such device features.

Another paper exemplifying progress within the field is a study by Smutný (2012), discussing the benefits of *"going native"* in the context of deciding on a development approach. They claim that developing an app using the native approach will allow for the best user experience if the app is used primarily offline (Smutný, 2012, p. 654). However, both cross-platform apps and Progressive Web Apps, as previously discussed, are offline-capable and functional, and should, to a great extent, not differ from the offline experience found in native apps. The *Web* approach does, according to Smutný (2012), not support offline mode, but this is no longer the case due to the possibilities of Progressive Web Apps (Majchrzak et al., 2018).

Progress within device- and platform feature accessibility can be drawn from studies such as that by Escoffier and Lalanda (2015), focusing on *heterogeneity and dynamism* in cross-platform apps. Regarding the state of feature access in 2015, the authors mention that cross-compiled solutions including Xamarin and Rhodes do not expose a feature-set vast enough to cater to the development of complex apps (Escoffier and Lalanda, 2015, p. 75). These findings somewhat contradict those presented in Palmieri et al. (2012)'s article, in which the authors state that Rhodes could access the majority of the features implemented in their study. Also, the Xamarin Website does not seem to agree with the claims presented by Escoffier and Lalanda (2015). This claim from 2015 may no longer apply, as the official statement from Xamarin entirely contradicts it, claiming that Xamarin does indeed provide access to all the platform and device functionality available, and list features such as iBeacons and Android Fragments as relevant examples (Xamarin, n.d.).

Similar claims have also been identified pointing in the direction of the Web approach – thus indirectly Progressive Web Apps. Although regular Web apps and Progressive Web Apps both suffer from limitations imposed by Web browsers, browser vendors and device feature availability exposed through JavaScript and HTML5 (Biørn-Hansen et al., 2018b), it is common to encounter outdated claims related to which device features these development approaches may access. Examples include programmatic access and control of device camera and GPS, which also in newer research (*e.g.,* Lachgar and Abdali (2017)) are reported inaccessible for these approaches. These claims contradict the findings provided by browser feature test platform "Can I Use", stating that both these features can be utilised in Web apps and Progressive Web Apps (Can I Use, n.d.a,n) – the same is also reported by Ciman and Gaggi (2016) in their large-scale energy consumption experiment. Nevertheless, other features are indeed not available from within a browser environment as of the time of writing, examples being those of device calendar and contact list. In an attempt to better the situation for Web apps, the WebAppBooster framework developed and presented by Puder et al. (2014), is an independent service running in the background on a mobile device, listening to connections executed from localhost environments such as a browser. This service allows Websites running in a regular browser to call upon the WebAppBooster through the WebSockets protocol, which in turn executes some functionality only available through native SDKs, including access to contact lists and sending of SMS. Not only is this approach novel and attractive, but the article also represents one of few scholarly research papers in which such a framework is presented outside of the model-driven development sphere, where academic tools seem more prevalent than those originating from practice.

2.5 Performance & Hardware Utilisation

The general perception in academia regarding the performance of apps developed using cross-platform frameworks is that it is inherently inferior to native apps due to abstraction layers such as interpreters, WebView engines and code transpilation- and compilation steps. An example of this perception is found in a study by Huy and vanThanh (2012) on the evaluation of development approaches, claiming that hybrid apps are performance-wise inferior to native apps due to the HTML rendering process that is required to take place in a WebView-based application, and that the result of this is that hybrid apps cannot replace native apps (Huy and vanThanh, 2012, p. 25). Based on this statement, a discussion that could be had is whether or not hybrid – or cross-platform approaches in general – are supposed to *replace* the native approach, or rather provide alternative development techniques and possibilities in contexts benefiting from such.

By traversing the performance-oriented literature, several studies have been identified, most spanning a timeframe from 2012 through 2020. A recently conducted experiment by Dorfer et al. (2020) details the memory (RAM) consumption, CPU usage and battery usage scrolling, maps, location tracking and networking. Their findings indicate a varying impact on all metrics as they compare a baseline native Android app to a React native implementation. On average, the React Native app consumes less memory than the native baseline; however, consumes between two and four times the CPU. In terms of battery, their findings also indicate that the React Native implementation consumes more battery on average. Dorfer et al. (2020)'s findings are of great importance to understand the trade-offs involved in deciding on a technology and development approach. The subjects of `RAM` and `CPU` consumption are followed up in Chapters 4 and 5 in this thesis.

Of other studies identified, we find that of Willocx et al. (2016), in which methods frequently found in performance studies are employed, including measuring memory consumption, disk space use and `CPU` usage on apps developed using a variety of cross-platform frameworks. They conducted a number of tasks while recording device performance, one of which proved to be more efficient in the cross-platform implementation than its native implementation baseline, which was that of page navigation in hybrid apps. One of their most important contributions and findings indicate acceptable performance loss in cross-platform apps, although the penalty introduced depends heavily on approach and framework (Willocx et al., 2016, p. 45). Additionally, they describe differences between high-end and lower-end mobile devices, and that the penalty is specifically acceptable on the higher-end ones.

However, while some experiments, such as the aforementioned performance study find that cross-platform apps generally perform inferior to native apps, one study identified reports the exact opposite. Ahti et al. (2016) found that their hybrid PhoneGap-based app had faster startup-time, consumed less memory and demanded much less disk space on Android compared to their native app. These findings are in stark contrast to most literature, and the authors describe a PhoneGap-based implementation which had its weaknesses related mostly to user experience but provided a *"technically feasible alternative"* to native development (Ahti et al., 2016, p. 47). Their report of weaknesses in user experience and app user interface appearance could be due to their use of a traditional Web interface library, Bootstrap, on top of PhoneGap. There are some Bootstrap alternatives they could have employed which expose HTML-based interface components mimicking both look and behaviour of native mobile interface components (Gronli et al., 2014), two examples being Ionic and Onsen UI. Nevertheless, their performance findings are of utmost interest, and contradict statements such as the aforementioned statement by Huy and vanThanh (2012, p. 25).

In addition to the experiments previously outlined, a comprehensive study by Ciman and Gaggi (2016) on device energy consumption was also identified. In their study, the authors measure the impact on energy consumption in an experiment including a wide array of device functionality and hardware sensors using four cross-platform frameworks – namely PhoneGap, Titanium and MoSync, and a regular Web app. They find that it is of utmost importance to base the choice of development approach and framework on the software specification at hand, but that cross-platform apps developed will regardless introduce some form of an energy consumption overhead. The interpreted approach is found to be more performant than they assumed, although an increase in device CPU usage was noted due to the need for runtime code interpretation. An interesting result from their experiment is that the previous statement also holds for apps of the cross-compiled approach, which are considered *"real native application[s]"* (Ciman and Gaggi, 2016, p. 16).

Drawing from these results, the cross-compiled based Xamarin framework which should generate native apps, do seemingly introduce a performance overhead when compared to the native approach, according to Willocx et al. (2015). While their findings vary significantly between different devices and platforms, the hybrid approach represented by PhoneGap tends to score lower than both Xamarin and the native approach. However, there are test cases in which PhoneGap and Xamarin present close to identical results, being those of CPU usage and on-device installation size. Dhillon and Mahmoud (2015) also report similar findings in their empirical investigation of technical frameworks, in which they find that the cross-compiled MoSync framework performs with notable differences on iOS and Android. While their iOS implementation performed well, the Android app performed subpar to what they expected from a framework of this approach.

Moreover, in the well-cited study by Dalmasso et al. (2013), which scrutinises and performs a comparison of cross-platform development frameworks, the authors discuss device performance using metrics including memory usage, power consumption and CPU usage. Because these metrics are frequently encountered in performance-oriented experiments, studying the progression of cross-platform performance through surveying existing literature could draw from papers such as theirs. The authors identify the hybrid PhoneGap app to be hardware-wise more performant and efficient than the interpreted Titanium app, although the trade-off is that of user experience and interface components which they found lacking in PhoneGap. While the experiment and the reported results contribute to the body of knowledge, a potential weakness in their research design is the lack of a native app from which they could extract a performance baseline for the metrics measured. Such a baseline is present in similar studies, for instance, those by Ciman and Gaggi (2016), Corral et al. (2012b), Willocx et al. (2015). Without a native baseline, it would be inherently difficult to validate whether the performance measurements of the cross-platform apps rendered better or worse results than the performance goal – which should be *native-like performance*, according to several highly-cited studies (Latif et al., 2016, Xanthopoulos and Xinogalos, 2013).

Although the Progressive Web Apps development approach is still in its infancy and thus has not yet seen much academic research effort, two studies were identified scrutinising its performance. In this respect, Malavolta et al. (2017) assess the possibility of an impact on energy consumption due to Service Workers executing and running background tasks. In brief, a Service Worker is a script that can run in the background as part of installed PWAs, conducting tasks such as data synchronisation, push notifications and data caching (Majchrzak et al., 2018). While no significant impact was noted, it was still suggested that the inclusion of a Service Worker is a trade-off in terms of (minimal) energy consumption impact and the additional features it can provide. The latter is at the core of our study as described in Biørn-Hansen et al. (2018b), in which a feature comparison and performance measurement between the native, interpreted, hybrid, and PWA approaches are presented. The results imply app launch- and interface render times on par with apps generated by the hybrid and interpreted approaches while requiring multiple orders of magnitude less disk space.

The earliest performance study identified, an experiment by Corral et al. (2012b) reveals that *native app performance* in cross-platform developed apps was not to be expected at the time of publishing, and reports an overall performance penalty in lapsed time for the measured tasks (for instance writing to and from a disk file, requesting data from the network or GPS module).

However, being that the penalty is only infrequently of any sizeable deviation from the native baseline, the authors also state that the penalty should be satisfactory for what they refer to as *"general-purpose business applications"* (Corral et al., 2012b, p. 742), or line-of-business apps. A refresh and validation of their study would be of immense interest, although extended with more up-to-date frameworks and technologies for greater future validity and industry relevance. Their study included a native app and a hybrid approach PhoneGap app, but the proliferation of cross-compiled and interpreted frameworks should be accounted for in newer studies. Also, a better understanding of differences in user expectation between business apps and regular consumer apps could help in decision-making processes.

Within the model-driven approach, studies targeting business contexts are frequently encountered (*e.g.,* Ernsting et al. (2016), Heitkötter and Majchrzak (2013), Majchrzak and Ernsting (2015), Rieger (2018)), and thus performance testing and surveying of business user app expectations could be of importance to understand the target group. Alas, although there are numerous frameworks of the model-driven approach originating from academia (*e.g.,* Heitkötter et al. (2013), Jia and Jones (2015), Kramer et al. (2010)), there is a general absence of performance discussions and empirical experiments examining the generated apps. A hypothesis for why this topic is so infrequently encountered in the literature could be that the models exposed by the frameworks translate to native platform code, and should thus in theory not differ from native apps (Gaouar et al., 2015). However, in a study by Jia and Jones (2015) on designing Domain-Specific Languages, the authors claim that apps developed using MDD-based frameworks and tools also suffer from performance penalties, something their DSL, *ADSML*, is aiming to improve. Nonetheless, they provide no empirical evidence to verify neither claim – that the average MDD tool suffers from performance constraints, nor that ADSML is *"[...] capable of generating high-performance native applications"* (Jia and Jones, 2015, p. 2). Suggestions for further research on frameworks within the MDD approach often include empirical verification of results (*e.g.,* Umuhoza et al. (2015)). Reports of performance benchmarking of MDD generated apps are yet to be encountered in the literature identified but could help to increase industry adoption of such frameworks, as suggested by Majchrzak and Ernsting (2015), the developers of the MD^2 framework. This suggestion is followed up in the performance experiment described in Chapter 4.

2.6 App Store Analysis

With more than four million apps available across the Google Play Store and Apple App Store (Statista, 2020), these marketplaces are not only distribution channels towards the platforms' end-users, they are also massive software repositories with excellent research potential. The rapid development of the mobile computing field and the enabling factors provided through it have led to considerable academic and industry interest in research and development of mobile apps. Studies from recent years have for instance focused on the impact of programming language on app quality in a study on Java versus Kotlin for development of Android apps (Góis Mateus and Martinez, 2019), assessing and comparing code smells in iOS and Android apps

(Habchi et al., 2017), and providing guidelines for ensuring the development of energy-efficient apps (Cruz and Abreu, 2017). Although app store analysis in the context of cross-platform mobile development is relatively new, descriptive studies have been published and cited frequently since the earliest frameworks were released. For instance, the study by Heitkötter et al. (2012b) shaped much of the research to follow due to actionable suggestions for future research, and for laying the foundation for framework comparisons. Also Charland and LeRoux (2011) discussed early on the possibilities of Web-based mobile apps as a potential challenger to native apps and placed cross-platform development on the research agenda. From a holistic perspective, studies related to cross-platform development and app store analysis often fall within the context of empirical investigation of software and metadata through Mining Software Repositories.

Mining Software Repositories (MSR) has established itself as a mature yet growing field of research within software engineering (Martin et al., 2017, Robles, 2010). While MSR is not a new field, the proliferation of the smartphone led to an increase in MSR research on mobile ecosystems. Mining and analysing mobile app marketplaces have been essential to our understanding of user behaviour on mobile devices (Hoon et al., 2013), the impact of programming languages for app development (Góis Mateus and Martinez, 2019), measuring the presence of hybrid development frameworks (Malavolta et al., 2015a), malware and vulnerability detection (Schütte et al., 2015), the evolution of mobile ecosystems (Wang et al., 2019), library usage in Android apps (Li et al., 2016), and more. The present thesis builds on previous work conducted within the field of MSR, although shifts the focus to cross-platform development, and with a primary focus on using readily available data sources such as AndroZoo rather than mining the Google Play Store directly. The use of MSR techniques within the mobile app ecosystem context was covered by Martin et al. (2017) in their comprehensive survey on the state of research on large app store repositories in the context of software engineering. They point to a growing academic interest towards studying these software repositories, with an increase in published software engineering research analysing mobile apps. However, they do not survey studies related to cross-platform mobile development specifically. App store studies incorporating the context of cross-platform development are nevertheless encountered in the body of knowledge.

One such study is by Mercado et al. (2016), in which the authors leverage user reviews mined from the Google Play Store and the Apple App Store to investigate the impact on the user-perceived quality of apps developed with cross-platform frameworks. It is noteworthy that their findings indicate differences between iOS and Android users regarding the perception of quality concerns in hybrid, interpreted and native apps. This leads to an interesting discussion on the use of cross-platform frameworks. Their findings are of great interest in decision making and better understanding implications related to the choice of technology.

Malavolta et al. (2015b) also scrutinise the user perspective in their study, although looking more specifically at user perception of hybrid mobile apps, also based on reviews mined from the Google Play Store. Similarly to the study described in Chapter 6 of this thesis, they provide an overview of cross-platform framework usage and distribution across the Play Store categories, although with a more limited dataset of $n = 11\,917$ apps and a partially different set of cross-platform frameworks (Cordova, Titanium, PhoneGap, Sencha, Kivy, Rho Mobile, UIU and Enyo). Through comparison of Malavolta et al. (2015b)'s results, indications of how this distribution has changed since the study was published in 2015 can be provided, but also how their findings from 2015 align with the findings on distribution over time (further investigated in Section 6.4.3).

From a broader perspective, Viennot et al. (2014) conducted a measurement study of the Google Play Store, describing their Web crawler architecture along with results on topics including native library usage, advertisement libraries, graphics engine usage, and briefly on cross-platform framework usage. Their results indicate that 9.20% of non-popular apps are built using cross-platform technologies, while 2.60% of popular apps are built the same way.

2.7 Taxonomy & State of Research

Having conducted a comprehensive review of cross-platform mobile development literature, certain gaps in the body of knowledge emerge. In Table 2.2, the state of user experience, software platform features, feature availability, and performance and hardware is briefly summarised based on the previous sections exploring each topic more in-depth.

As can be derived from Table 2.2, several areas of research within cross-platform mobile development either lack research all-together or suffer from predominantly dated knowledge bases. Additionally, the state of research on app store analysis is also described in the literature review, although not included in Table 2.2 as the topic does not fit into the approach-based categorisation used in the table.

Table 2.2: Summary of state of research on cross-platform development.

	Hybrid	**Interpreted**	**Cross-compiled**	**Model-Driven**	**Progressive Web Apps**
User Experience	Quantitative. Lacking qualitative.	Some qualitative. Insufficient.	Insufficient research.	Insufficient research.	No identified research.
Software Platform Features	Well-researched.	Well-researched.	Some research. Conclusion lacking.	Some research. Conclusion lacking.	Some research. Conclusion known.
Feature availability	*Debated. Likely not limited.*	*Debated. Likely not limited.*	*Debated. Likely not limited.*	*Likely not limited.*	*Browser constrained.*
Performance and Hardware	Well-researched. Dated.	Well-researched. Dated.	Well-researched. Dated.	Insufficient research.	Insufficient research.

Specifically, the research gaps addressed by this thesis are those of hardware consumption and app performance generated by cross-platform frameworks (as described in Table 2.2), and the app store presence of these technologies (described in Section 2.6). There is a pressing need for updated research on both topics, as the industry moves forward with new frameworks, thoughts and technologies. Much of the academic efforts towards investigating cross-platform apps were conducted between 2011 and 2017, after which the number of new studies declined, particularly within the topics covered in this literature review.

In addition to the condensed findings presented in Table 2.2, a technical

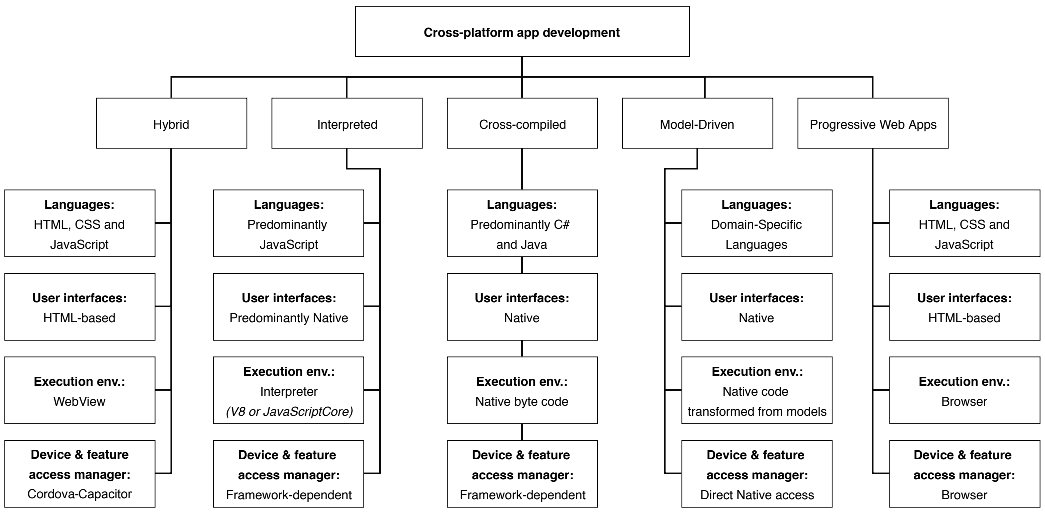

Cross-platform app development
Hybrid
Interpreted
Cross-compiled
Model-Driven
Progressive Web Apps
Languages: HTML, CSS and JavaScript
User interfaces: HTML-based
Execution env.: WebView
Device & feature access manager: Cordova-Capacitor
Languages: Predominantly JavaScript
User interfaces: Predominantly Native
Execution env.: Interpreter (V8 or JavaScriptCore)
Device & feature access manager: Framework-dependent
Languages: Predominantly C# and Java
User interfaces: Native
Execution env.: Native byte code
Device & feature access manager: Framework-dependent
Languages: Domain-Specific Languages
User interfaces: Native
Execution env.: Native code transformed from models
Device & feature access manager: Direct Native access
Languages: HTML, CSS and JavaScript
User interfaces: HTML-based
Execution env.: Browser
Device & feature access manager: Browser

Figure 2.7: Taxonomy of cross-platform development.

overview is provided, addressing the five development approaches discussed throughout this review. Figure 2.7 depicts the current state of the art in terms of predominant programming languages, app execution environments, how user interfaces are rendered, and which controlling entity is managing access to the device- and platform features.

Having assessed previous literature and through that identified two significant research gaps related to framework presence and performance, the coming chapter discusses methods and models for carrying out and evaluating the three thesis experiments addressing these gaps.

Chapter 3

Research Methods

This chapter is devoted to exploring philosophical perspectives of science, the design science research methodology, and the experimental design and evaluation in order to contextualise the thesis as a whole. More in-depth details on each conducted experiment is presented throughout the subsequent thesis chapters.

3.1 Philosophical Perspective

There is a vast array of assumptions, paradigms and perspectives involved in conducting scientific work. Epistemological discussions are frequently encountered in computer science and software engineering, as they are in, for instance, the natural and social sciences. This section discusses among other the epistemological perspective subscribed to while conducting the experiments and overall thesis – specifically positivism. As a point of departure for the discussion to come, the thesis' related field of study is considered to be somewhere between computer science and software engineering, specifically applied computer science; wherein there is a focus on developing and measuring software solutions with an impact on both practice and academia. Rather than using mathematical proofs, models and formal methods for validation, my interest is more towards an objective and statistical interpretation of performance measurements. Rather than building or expanding on existing software solutions to attempt improving algorithmic throughput or a reduction in a software's hardware requirements, my interest is on building relatively isolated technical artefacts for the purpose of reporting on what is state of practice in performance and presence of cross-platform frameworks. Based on the discussion by Eden (2007) in their examination of philosophical disputes in computer science and software engineering, the work presented in this thesis is considered to belong to the technocratic paradigm, which is described as the idea that computer science is:

> *"[...] an engineering discipline, treats programs as mere data, and seeks probable, a posteriori knowledge about their reliability empirically using testing suites." - Eden (2007, p. 3)*

The data used to support the findings reported in Chapters 4, 5 and 6 are predominantly quantitative and statistical. The data has been extracted based on the use of scripts and test suites in an exploratory and experimental approach, deriving knowledge *a posteriori* – through observations and experiments. Although the knowledge body on cross-platform mobile development is quite robust as illustrated in Chapter 2, due to the variations in experimental design, physical test devices, cross-platform frameworks, and the rapid development of the field, the work conducted is still more exploratory than confirmatory. Results from the experiments conducted are based on measurements using objective metrics with an emphasis on reproducibility of results, focusing on detached interpretation of results, subscribing to the idea that a single reality exists although with inherent contextually differing world-states. Indeed, these are all characteristics of the positivistic scientific perspective (Vaishnavi and Kuechler, 2015, p. 31).

3.2 Design Science Research

As the overarching research methodology, design science research (DSR) has been used. The purpose of DSR is to guide projects aiming at conducting rigorous and relevant design-driven research, where *design* in our context represents *implemented and evaluable technical artefacts*. Through this section, the DSR methodology is explored and discussed, alongside its use in this thesis.

3.2.1 An Overview of the DSR Methodology

While there are numerous takes on what design science research is and is not (Baskerville, 2008), Hevner et al. (2004) ultimately describe it as a problem-solving methodology; input is gained from industry and gaps in the body of knowledge, research is conducted using rigorous methods, and output is commonly relevant to both practitioners, organisations and science. It is a *science of the artificial* (Simon, 1996), and revolves around the design and evaluation of novel artefacts, whether being code, systems or concepts

(Baskerville, 2008). For this thesis, the technical artefacts designed and evaluated are based on executable code; scripts and mobile apps.

Much of the seminal work on DSR is written in the context of Information Systems (IS) research, perhaps most notably studies including Nunamaker et al. (1990), March and Smith (1995), Hevner et al. (2004), Peffers et al. (2007), Iivari (2007) and Gregor and Hevner (2013). Although the emphasis in these studies is on IS research, the significance for systems development both as an activity and a methodology is clear. Accordingly, Nunamaker et al. (1990) describe the importance of prototyping technical artefacts and the subsequent experiments and evaluation, for instance, in terms of performance. They further describe observing the impact on organisation and individuals as a consecutive step, not as a necessity for conducting design science research.

A recent study by Engström et al. (2020) investigates the use of DSR in software engineering and systems development literature. The authors map 38 software engineering papers published in the International Conference on Software Engineering (ICSE) to a proposed design science research template. Their findings indicate that although few papers explicitly mention the use of DSR, there is an implicit link to DSR in the software engineering (SWE) projects through the problem formulation and proposed solution in the context of rigour, relevance and novelty. Thus, while the fundamentals of DSR originated in IS research, the findings by Engström et al. (2020) indicate that although its use in related fields is equally suitable, the methodology is rarely explicitly mentioned in published studies within SWE. For this thesis, the background on- and understanding of DSR lean on both the traditional literature from an IS perspective, accompanied by newer literature also encompassing DSR in SWE, such as studies and books by Wieringa (2014), Vaishnavi and Kuechler (2015) and Engström et al. (2020).

Several models have been developed for explaining, conducting and evaluating DSR projects over the past decades. This thesis leans on two specific models for the methodological grounding of the project. The first model is by Hevner (2007), the *"Three Cycle View"* (see Figure 3.1), illustrating the interconnectivity and dependencies on rigour and relevance in a design science research project. It was indeed this focus which was particularly vital in the choice of methodology for the thesis at hand.

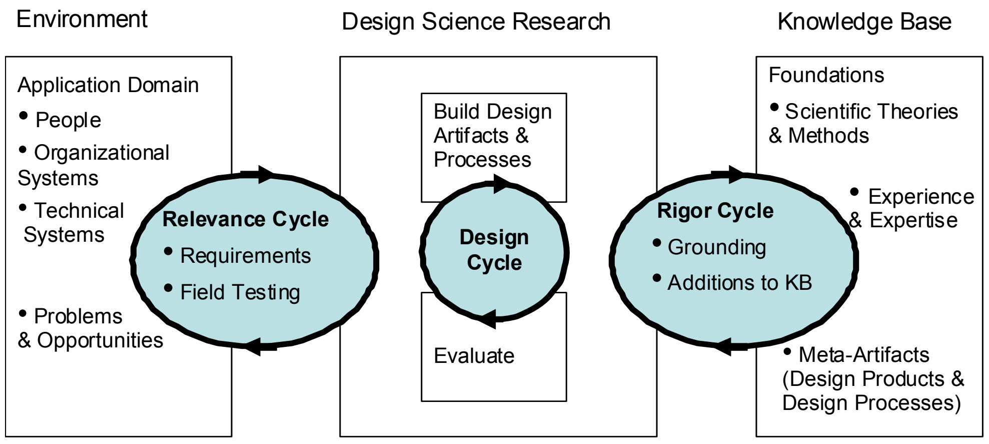


Figure 3.1: Hevner's Three Cycle View of Design Science Research. Reprinted from "A Three Cycle View of Design Science Research" by Hevner (2007, p. 2). © Copyright 2007 by BePress.

The three blocks and accompanying cycles make up Hevner's DSR model, describing the overarching concepts of how a DSR project is influenced by both a surrounding environment and existing rigor. While the *"Three Cycle View"* provides a holistic perspective of the main components of DSR, it does not by itself provide guidance for the actual implementation and execution of a DSR project. For this, a process model proposed by Vaishnavi and Kuechler (2015) has been followed.

3.2.2 Adopted DSR Process Model

It is important to note that several process models for DSR projects exist, which is also acknowledged in Vaishnavi and Kuechler (2015)'s model description – explaining their model as an iteration of that previously proposed by Peffers et al. (2007) (Vaishnavi and Kuechler, 2015, p. 19).

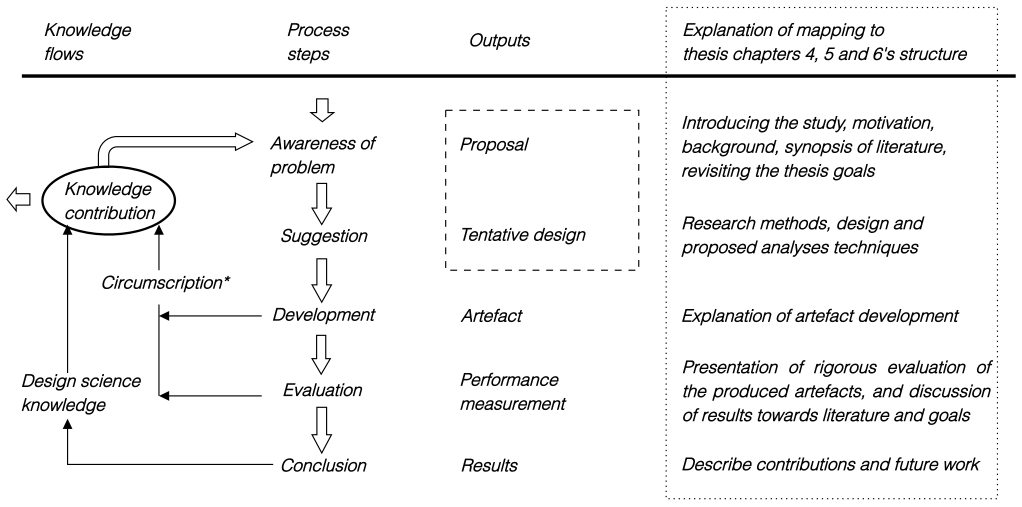

Knowledge flows
Process steps
Outputs
Explanation of mapping to thesis chapters 4, 5 and 6's structure
Knowledge contribution
Awareness of problem
Proposal
Introducing the study, motivation, background, synopsis of literature, revisiting the thesis goals
Suggestion
Tentative design
Research methods, design and proposed analyses techniques
Circumscription*
Development
Artefact
Explanation of artefact development
Design science knowledge
Evaluation
Performance measurement
Presentation of rigorous evaluation of the produced artefacts, and discussion of results towards literature and goals
Conclusion
Results
Describe contributions and future work

Figure 3.2: Design Science Research Process Model adapted from "Design Science Research Methods and Patterns", by Vaishnavi and Kuechler (Vaishnavi and Kuechler, 2015, p. 15). © Copyright 2015 by Taylor & Francis Group, LLC.; Modified to include a description of each step's relevance to the structure of Chapters 4, 5 and 6 (inside the dotted box to the right).

The process model by Vaishnavi and Kuechler (2015) was chosen for primarily two reasons: (i) it provides guidance and a clear structure along with a set of actionable steps to follow, and (ii) each step maps well to the structure and narrative of the thesis experiment chapters. Indeed the five steps derived from the process model – *Awareness of problem, Suggestion, Development, Evaluation,* and *Conclusion* – have been used to structure the experiments presented in this thesis, respectively Chapters 4, 5 and 6. The model, as illustrated in Figure 3.2, consists of five specific steps, each generating a particular output while contributing to knowledge through circumscription. The use of the model and its steps in this project have been included to the right in Figure 3.2, thus the original figure has been modified to contain thesis-specific details.

The short description of each of the five steps to follow are based primarily on those provided by the process model creators, Vaishnavi and Kuechler (2015, p. 14-17), alongside own interpretations and comments. Starting at the **Awareness of problem** phase, the research gap is identified and highlighted (derived from the literature review in Chapter 2). A design science problem can originate from either academia, industry or a combination. The basis of the aim, objectives and research questions introduced in Chapter 1, is the combination of an academic literature gap and an industry need. The output, a research proposal, is the anchoring to relevance and state of the art. The following step, **Suggestion**, is the anchoring to rigour, describing and proposing, for instance, an experimental design, analysis techniques and other components related to method and methodology. Thus, this step in regards to this thesis is related to Chapters 2 and 3, and the suggestion of methods and evaluation accompanying the experiment chapters. Following is the **Development** step, in which the artefact development is detailed, for instance, the development and use of specific algorithms, technical tools and frameworks, user interfaces and systems. For this step, the authors stress that an artefact implementation *"need not involve novelty beyond the state-of-practice for the given artefact"* (Vaishnavi and Kuechler, 2015, p. 16). This has been the case for two of the experiments towards this thesis, respectively in Chapters 4 and 5, in which the novelty is not the technical artefacts themselves, but rather the performance measurement outputs and subsequent analysis of the experiments. The **Evaluation** step is where either tentative or concluding results from experiments and observations are brought forward and discussed in light of research questions, hypotheses and the existing knowledge body. For the experiments described in Chapters 4, 5 and 6, the evaluations are primarily based on statistical, descriptive and partially prescriptive knowledge from performance measurements and software repository mining. Finally, the **Conclusion** draw on all previous steps but focuses on communicating the results of the former evaluation step, and the overall contributions, whether exploratory or confirmatory.

3.2.3 Alternatives to DSR

As previously indicated, design science research is a natural methodological choice for conducting the type of research presented in this thesis. Beyond the scope of this thesis, a natural extension of the conducted experiments would be to measure their impact in the context of organisations and people. Conducting for instance Action Research (AR) and Action Design Research (ADR) as vehicles for engaged scholarship could be feasible approaches (Moloney and Church, 2012), merging artefact development in

DSR with the interdependence on organisational needs and policies. Another possibility could be to conduct case studies with companies either developing cross-platform apps, or who are looking into developing a new product from scratch or migrating from or to a native or cross-platform approach. Such work could provide valuable knowledge to researchers and framework developers on intricacies and requirements. As gauged by Engström et al. (2020), work in software engineering is often conducted without explicitly mentioning the use of an overarching methodological framework. Thus, an alternative to DSR for the type of work conducted in this thesis could also be the absence of a methodology. However, this could affect both the structure and scientific rigidness of the thesis, which are heavily influenced by DSR theory and models.

3.3 Experiments

The three experiments conducted towards this thesis, as to be described in Chapters 4, 5 and 6, all rely on a combination of artefact development and statistical and descriptive evaluation. The technical artefacts are primarily mobile apps developed for performance evaluation, and programs designed to harvest and analyse large datasets. This section outlines the development and evaluation of these technical artefacts and on the mining and analyses of software and metadata from the Google Play Store and Androzoo.

3.3.1 Artefact Development

At the core of the design science research methodology is the design, creation and evaluation of *IT artefacts*. Both March and Smith (1995) and Hevner et al. (2004, p. 77) describe four types of IT artefacts or outputs from a DSR project, specifically *Constructs, Methods, Instantiations* and *Models*. This list is extended by Vaishnavi and Kuechler (2015, p. 20) with *Frameworks*, *Architectures*, *Design principles* and *Design theories*. Four of these are of direct relevance to this thesis and its outputs, which are summarised in Table 3.1.

Table 3.1: Design Science Research artefact categories by Hevner et al. (2004, p. 77) and Vaishnavi and Kuechler (2015, p. 20), extended with a mapping of relevance to this thesis.

Artefact Type	**Description**	**Thesis Relevance**
Constructs	*Vocabulary and symbols*	The taxonomy of cross-platform mobile development presented in Chapter 2.
Instantiations	*Implemented and prototype systems*	Mobile apps implemented for use in data gathering, statistical analyses and performance evaluation in Chapters 4 and 5.
Methods	*Algorithms and practices*	The cross-platform framework recognition algorithm developed towards the study on framework presence in Chapter 6.
Frameworks	*Guiding principles, real or conceptual*	The final outcome of this thesis as presented in Chapter 7.

A great deal of the artefact development has been of the *Instantiations* type, where mobile apps have been developed for the sake of performance benchmarking and extraction of subsequent results. A *Construct* in the shape of a taxonomy was developed and published as part of the literature review, while a *Method* in the form of a framework identification algorithm was developed for the Google Play Store analysis experiment in Chapter 6. As for the final thesis contribution presented in Chapter 7, a *Framework* type artefact has been developed as a set of guiding principles based on the empirical experiments and their outputs, in the context of the thesis' aim and objectives.

3.3.2 Artefact Evaluation

While the design and creation of IT artefacts are integral parts of the design science research methodology, so is the evaluation of said artefacts. Considerable effort has been put into the artefact evaluation and development of reproducible research designs, through open sourcing of results, datasets and technical artefacts. Hevner et al. (2004, p. 85) list eight attributes on which an IT artefact can be evaluated, of which the *accuracy* and *performance* attributes are made use of in the thesis experiments. Combining the performance attribute from Hevner et al. (2004) with the *benchmarking* attribute proposed by Vaishnavi and Kuechler (2015, p. 282) describes well the experiments and evaluations described in Chapters 4 and 5 wherein the performance of native and cross-platform mobile apps is benchmarked in terms of native code communication (bridging) and animations, on metrics including time-to-completion (`TTC`), `CPU` usage, `RAM` consumption, `GPU RAM` consumption, and Frames-per-Second (FPS). The physical mobile devices used in the data extraction for the experiments described in Chapters 4 and 5 include both low-, mid- and high-end devices to target a complete range. This was deemed necessary to increase the validity and generalisability of results, and the choice of testing on physical devices rather than emulators – although more complex and time-consuming – was deemed equally important.

The *accuracy* attribute was also a central part of one experiment, respectively in Chapter 6 in which a proposed algorithm was designed to identify apps in the Google Plays Store which have possibly been developed using cross-platform development framework(s). Although evaluating the accuracy by itself is not the main contribution of the experiment, it was an integral part in the iterative development of the algorithm, ensuring that the algorithm would to a satisfactory degree correctly identify the use of cross-platform frameworks. The subsequent evaluation in the experiment is more statistical and descriptive, contributing to the overall thesis aim.

Each experiment chapter will further detail the artefact development and evaluation and tie the results and discussions back to the thesis research questions posed in Chapter 1.

3.3.3 Ethics

Results and findings presented in this thesis are based on performance measurements and data from public and available repositories (Google Play Store and AndroZoo) and has not relied on human or organisational involvement. Limitations are raised as part of the published manuscripts and in Chapter 7 to highlight known and potential shortcomings and threats to validity of results and findings. For this PhD thesis project, an application was sent to the Brunel Research Ethics Online (BREO) office for ethical consideration, and was approved by the appointed ethics committee (approval form included in Appendix E).

Chapter 4

Bridge Performance of Mobile Development Approaches

This experiment investigates an essential part of cross-platform mobile apps' performance, namely the bridge providing access to underlying platform and hardware APIs. The chapter firstly revisits the problem described previously in the thesis introduction, along with research questions and methods of data gathering and data analyses. Subsequently, the evaluation of bridge performance is presented, statistically analysing time to completion for the benchmarked tasks, memory consumption and `CPU` usage. A weighting of the benchmarked frameworks is then provided, summarising the chapter findings.

Communication of Research: This is an extended version of the publication in Springer Empirical Software Engineering (2020). The content and format has been modified to fit the thesis narrative. Replication package available in Appendix B.

4.1 Awareness of Problem: A Potential Performance Overhead

Revisiting the Objective. The performance of cross-platform frameworks is often discussed in both academic and industry outlets, as previously described in Chapter 2 on related work. The purpose of the experiment described in this chapter is to investigate the potential performance overhead imposed through the use of such frameworks and their bridging mechanisms for communication between the cross-platform and native environments, by comparing performance results from five frameworks to a native Android app baseline implementation. This investigation is conducted towards answering the first thesis research question (RQ_1) regarding performance as described in Section 1.4. Derived from this experiment, a weighting of the benchmarked frameworks is developed based on their overall relative performance output as measured through a series of benchmark tests.

Background and Literature Synopsis. Certain studies indicate an inherent performance loss in cross-platform apps, although end-users may not negatively experience this during everyday use (Angulo and Ferre, 2014). Nevertheless, the choice of a suitable technical development framework has been found to matter a great deal in terms of expected performance (Corbalan et al., 2018). Besides, it is not clear if cross-platform apps are inherently subpar to native apps in terms of performance output: using frameworks that generate native apps might yield code that outperforms hand-written code due to optimisation; interpreted apps could undergo runtime optimisation that leads to better performance than apps optimised at compile-time.

Ciman and Gaggi (2016)'s comprehensive evaluation of energy consumption for multiple cross-platform frameworks reports significant differences in hardware performance between the evaluated frameworks. They also observed differences between programming languages, *e.g.*, how their C++ based artefact was outperformed by an artefact designed using JavaScript, although both artefacts were built using the MoSync development framework that supports both programming languages. In a recent related study, Corbalan et al. (2018) focus on the increase in energy consumption caused by cross-platform frameworks, although their results and methods differed compared to those from Ciman and Gaggi (2016). Their findings showed that whereas Apache Cordova handled processing and audio playback well, it performed poorly in their video playback measurement. The Corona framework (renamed to Solar2D in 2020), on the other hand, performed well at video playback but had issues with intensive processing – illustrating possible trade-offs when deciding on a development framework.

The results presented by Ciman and Gaggi (2016) share similarities with those reported in related performance studies, *e.g.*, by Willocx et al. (2015, 2016) focusing on the hardware impact imposed by a variety of frameworks and implementations. Whereas accelerometer and GPS sensor values, camera, and network access represent standard evaluation criteria, many platform features have already been covered as well as deliberate restrictions to in-app computations – although also implemented using a set of cross-platform frameworks (Delía et al., 2017). For instance, access to platform features in PhoneGap/Apache Cordova was approximately twice as slow compared to native apps and going up to a factor of 20 for file system access and beyond for GPS sensor usage (Corral et al., 2012b). However, more recent studies indicate that the framework is still more resource-intensive; load times are 40% slower than for native apps (Que et al., 2016), and sometimes it even outperforms native implementations (Delía et al., 2017). Performance overhead is further investigated in this current experiment to better understand the state of performance in cross-platform apps compared to native apps.

4.2 Suggestion: Measure Performance Output

To better understand the potential performance overhead imposed through the use of cross-platform mobile development frameworks, technical artefacts were developed to undergo performance scrutiny. This section describes the research question, objective and technologies involved in this experiment.

4.2.1 Objective and Research Question

The objective is to measure the performance of native-side access to platform features through invoking- and having data returned over a bridge (communication layer)- from foreign function interfaces (FFIs) or framework- and approach-specific equivalents. The main contribution from this experiment is an in-depth investigation of the technologies enabling cross-platform development frameworks to provide functionality similar to what is found in native app development by measuring the performance-oriented impact of individual hardware or platform features and to empirically assess the performance of cross-platform app development to this extent. Evidently, the tangible contribution is a weighting of the assessed frameworks on their relative performance output and hardware consumption. The results produced go towards answering the first overarching thesis research question,

RQ_1: How do apps developed using cross-platform mobile development approaches and associated frameworks perform compared to native mobile apps in terms of hardware and platform utilisation?

For this experiment, one research question has been formed based the knowledge gap surfacing from the literature review and background section: that with the introduction of newer frameworks and approaches, the deprecation of older technologies, and the lack of large-scale empirical studies on bridge performance has spawned an array of assertions and claims in both academia and industry.

***Sub−RQ*$_{1.1}$:** To what degree do cross-platform mobile development frameworks impose additional performance-related overhead when compared to native mobile development?

4.2.2 Technologies

A total of six artefacts have been developed using five cross-platform frameworks and the native development approach, and profiling tools have been used to measure the performance of these apps. Unlike the majority of studies in which performance of cross-platform development frameworks is investigated, the artefacts developed in this current experiment use a wide array of technologies, including frameworks of the model-driven development approach, hybrid approach, interpreted approach, cross-compiled approach, and the native approach.

Table 4.1 lists six technologies which have been used in the development of the artefacts. Of these, one belongs to the native approach, *i.e.*, it does not support cross-platform deployment. It serves as the Android baseline benchmark. The remaining five technologies allow for the creation of iOS and Android apps based on a common code base, although for this experiment, only the Android platform has been assessed. They vary in terms of programming language, associated development approach, industry adoption, among other aspects.

Table 4.1: List of technologies included in the experiment.

Framework	**Version**	**Associated Approach**	**Programming Language**	**APK Size**
Ionic	v3.9.2	Hybrid *(Cordova-based)*	TypeScript	10.3MB
React Native	v0.53.2	Interpreted	JavaScript	9.7MB
NativeScript	v3.4.1	Interpreted	JavaScript	30.2MB
Flutter	v0.5.1	Cross-compiled	Dart	32.8MB
MAML / MD^2	v2.0.0	Model-Driven Development	DSL	3.2MB
Native Android	-	Native	Java	2.7MB

4.3 Development of Artefacts & Methods

This section further elaborates on the rigorous artefact implementation work conducted towards this experiment, describing benchmarked features and the data gathering process, alongside detailing the data analysis conducted.

4.3.1 Artefact Design and Implementation

The focus in this experiment is on the capabilities of the cross-platform frameworks to provide access to the device hardware and operating system features of the underlying platform, and not the frameworks' capabilities to render user interfaces[1]. A unified representation across the different framework implementations is essential, and use separate views for each task (see next

[1]In fact, studying the visual capabilities in the light of a debate about native look and feel (Heitkötter et al., 2012b, Majchrzak and Heitkötter, 2014) would be an interesting idea for an empirical research paper, ideally conducted with real users.

Section 4.3.2) which can be selected from an introductory start screen. Although the number of runs for a specific benchmark can be specified, this feature was in the end not made use of, as instead only a single benchmark run was executed before restarting the app and starting over (see Figure 4.1). When executing the benchmark through a `Start Benchmark` button in the user interface, the app will initialise the benchmark run of the respective feature by measuring the time until the value of the platform feature is retrieved. This value (time-to-completion, `TTC`) is printed to the screen and manually transferred into an external datasheet. Section 4.3.3 describes the gathering of measurement results from the remaining metrics.

4.3.2 Benchmark Features and Tasks

A plethora of hardware features exist that invite for benchmarking, including sensors (accelerometer, gyroscope, compass), network (cellular, WiFi, Bluetooth, NFC), native events (hardware `back` button, volume keys), device information such as battery status, and many more. Additionally, features of the operating system, storage databases, contact lists, or notifications, can all be accessed by native apps and therefore also through bridge components in cross-platform frameworks.

Comprehensive sensor usage statistics are not available and can only be approximated through requested app permissions. From the 42 Android apps in the Google Play Store that have more than 1 billion installations

(Androidrank, 2019), 37 apps request (external) file system access, 33 access contact lists, 26 use GPS location, 24 ask for image capture permissions, 23 read out the phone status, 18 request microphone access, eight read or modify calendar entries, eight want to read SMS, and one app accesses body sensors. However, not all features require explicit permissions by the user, *e.g.*, the accelerometer sensor, and every app has access to some software features such as a database.

From this extensive list of possible features, five benchmarks were designed. These relate to both hardware and software capabilities. Features were selected to reflect the most common use cases (based on the usage statistics presented) while at the same time being present on many devices – in contrast to specialised sensors, which only a few devices provide and few people use. Moreover, benchmarks were chosen to be executable mostly in isolation, avoiding elaborate multi-device set-ups that rely on external factors such as network quality beyond a developer's control, helping to provide objective and reproducible results.

The features implemented as tasks for benchmarking are as following:

Accelerometer: The accelerometer sensor captures data on the acceleration force applied to the device in all spacial axes in m/s^2. It is mostly used for simple routine tasks such as device orientation changes but can be employed for complex activities, for examples in augmented reality (AR) settings. The benchmark requests these three values (x, y, z

axes) for the next update. To measure the minimum reaction time, the sensor sampling rate is set to the mode *SENSOR_DELAY_FASTEST*, which avoids artificial delays intended to reduce processor load and power consumption.

Contacts: Contact lists are routinely utilised by almost all users of smartphones. The contacts benchmark involves creating and inserting a new contact into the device's contact list. In terms of the contacts' information, each contact object was provided with a name and a mobile phone number. This was deemed the minimum amount of data needed to store a new contact in a real-world context.

File system: Similarly, reading files stored on the file system is evaluated. The test is conducted using a benchmark PNG image of 528 x 528 pixels and a size of 613 kB. In order to separate the device access from the UI representation, only the time between the base64-encoded string is decoded in memory and ready for assignment to a view element (but excluding the actual image rendering to the screen) is measured.

Geolocation: Finally, accessing location information via GPS sensor or network-based positioning mechanisms is another common use case of mobile-specific functionality. It is used for routing, to provide location-based services and hints, and for other localisation purposes. This benchmark retrieves the longitude and latitude values of the device's current location based on the vendor-recommended location retrieval mechanism, as the geolocation implementation between the frameworks differed in terms of options granularity.

4.3.3 Data Gathering

For this experiment, data was gathered on time-to-completion (`TTC`), `CPU` usage, idle-state `RAM` occupancy (`PreRAM`), and busy-state memory occupancy (`RAM`). All measurements, except time-to-completion, were recorded using the Android Studio profiler tool using the default Java sampling method for data collection, which captures values using a frequent sampling interval of 1 ms. Because more accurate trace-based inspection of method calls impacts runtime performance but provides no additional value in terms of the above metrics, the configuration is sufficient for this experiment. Specifically `CPU` and memory usage (`RAM`) are metrics also included in previous performance studies, including Dalmasso et al. (2013) and Willocx et al. (2015). The `TTC` metric is provided in milliseconds, and describe the duration of time between invoking a benchmark task in the cross-platform context and having the results returned from the app's native-side. An example of this is the time it takes from requesting accelerometer data until the values are provided back to the cross-platform context, ready to be displayed to the user. The `CPU` usage is the percentage of processing power consumed at peak during benchmarking. Within the Android Studio profiler, values are provided for the specific app; thus, the single highest consumption observed for the given benchmark is gathered. Idle-state `RAM` consumption (`PreRAM`) is the observed memory consumption in megabytes when the app is running on a device just before executing a benchmark task. This facilitates the analysis of fundamental memory requirements among the frameworks included in the experiment. The busy-state `RAM` consumption (`RAM`) is the observed peak of memory consumption in megabytes during the execution of a benchmark task. Specifically, it is the difference between the `RAM` and `PreRAM` variables (denoted as `ComputedRAM`) that will assist in understanding the actual impact on memory consumption caused by each specific benchmark task and framework.

All performance benchmarks were conducted on physical mobile devices rather than on emulated hardware due to subtle differences in receiving realistic sensor input and effects of continued physical execution (Joorabchi et al., 2013) – to the extent that evasive malware can use a multitude of heuristics to detect emulators (Mutti et al., 2015). Furthermore, the heterogeneity of devices, including attributes such as processor and memory, needs to be taken into account as stressed by Noei et al. (2017) in their research on user perception of software quality versus device and app attributes. All the Android APK app installation files were built for release rather than debug mode. This was especially required for specific cross-platform frameworks, as, *e.g.*, Flutter limits the performance of apps built and compiled using debug mode. Nevertheless, to extract information on app-specific usage and utilisation of `CPU` and memory on-device, APKs built for release must include a `debuggable` property in their Gradle configuration (Google LLC, 2019a). This is done to enable Android Studio's profiler tool to gather necessary profiling data for inspection.

First, the time-to-completion feature benchmark was conducted using APKs without the `debuggable` property. Secondly, the APKs were re-compiled, this time including `debuggable`, and conducted the profiling using the Android Studio profiler environment. This approach should allow the time-to-completion benchmark to produce results unaffected by potential monitoring overhead, while afterwards being able to retrieve `CPU` and `RAM` data using the means available. Both processes are illustrated in Figure 4.1, displaying the extraction of results from within the app while running on a device, and results from Android Studio.

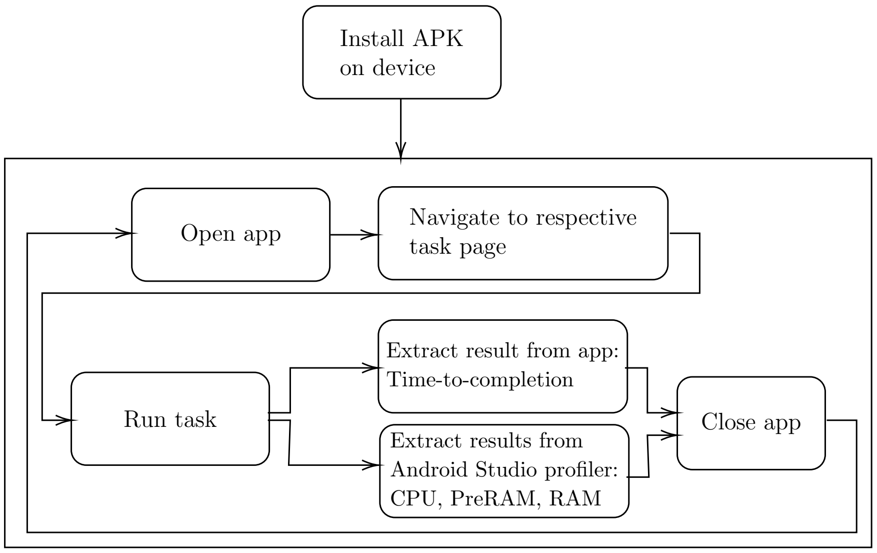

Install APK on device
Open app
Navigate to respective task page
Run task
Extract result from app: Time-to-completion
Extract results from Android Studio profiler: CPU, PreRAM, RAM
Close app

Figure 4.1: The process of gathering performance measurements.

Furthermore, Figure 4.1 illustrates the effort put into the data gathering process. For each loop, as illustrated in the figure, only one ($n = 1$) benchmark run was executed. This process was then repeated $n = 30$ times for each combination of device, feature and performance metric. In order to extract results on time-to-completion (`TTC`), a task would be executed, and upon completion, the result (in milliseconds) would be displayed within the app's user interface, after which the result of the benchmark would be transferred manually from the app into a datasheet. As recent research indicates a non-trivial energy consumption overhead related to the use of automation frameworks (Cruz and Abreu, 2019), process automation related to data gathering was avoided. This holds true for extracting results on `CPU`, `PreRAM` and `RAM`, all of which were manually extracted from the Android Studio profiler. The process involved starting the Android Studio profiler and letting it record the preferred metrics, executing the benchmark on-device, then manually inspecting the recorded event timeline to identify the impact on the metrics caused by the execution of the benchmark. A separation was made between the process of extracting the time-to-completion metric, and the process of extracting the remaining metrics; `CPU`, `PreRAM`, and `RAM`. Thus, all benchmark tests were executed twice to gather the aforementioned results using different processes: once for `TTC`, and once more for the remaining three metrics. There were no automated services or processes involved in the data gathering process. Every interaction with the app, every extraction of data, whether from within the app or from the Android Studio profiler, were done manually.

Benchmarking the JavaScript-based implementations (*i.e.*, Ionic, React Native and NativeScript) leverages the `Date` API for calculating elapsed time towards the `TTC` metric. While there are other timer units available for JavaScript, including `DOMHighResTimeStamp` (W3C, 2018), no other timer than `DateTime` for the Dart-based Flutter implementation was identified (Google LLC, 2019b). Thus, in an attempt to harmonise the timer units, the JavaScript `Date` API was used in place of `DOMHighResTimeStamp`. These differences in time units and their resolutions could be seen as a technical limitation inherently affecting the obtained results. Nevertheless, the lack of a unified cross-language timer implementation ultimately led to this decision.

Preparation of the included devices was a significant step towards ensuring a unified hardware baseline. Thus, prior to the benchmarking, networking features including WiFi access, Bluetooth connectivity, and mobile data were turned off, limiting external interference. All background apps were terminated, and the benchmark started after ensuring in the Android Studio profiler that overall processor load had abated. After each completed benchmark run (*i.e.*, retrieval of one set of results, either `CPU`, `RAM` and `PreRAM`, or time-to-completion (`TTC`)), the app would be terminated and restarted, then proceed to the next benchmark run, and repeat the process in order to avoid distorted results from warm starts of the app screens which already reside in memory (Singh, 2017). Restarting the app from scratch also limits caching of app content and executing previously just-in-time compiled code by the Android Runtime (Google LLC, 2019c) – although these operating system level optimisations are generally beyond the control of the app developer, it was aimed for realistic real-world behaviour of the device.

For this experiment, the performance measurements were conducted on a total of six mobile devices, as further described in Table 4.2. The devices included in this experiment represent a wide range of hardware, including both budget smartphones and state of the art. The number of devices is also above or on par with most recent related work, for instance Huber et al.

(2020) (three devices), Barros et al. (2020) (two devices), Willocx et al. (2016) (four devices).

Table 4.2: List of devices used for measuring performance.

Model	**CPU** (Cores)	**Memory**	**OS**
Samsung S8	2.3+1.7 GHz (Octa)	4GB	Android 8.1
Samsung A3 (2016)	1.5 GHz (Quad)	1.5GB	Android 7.0
Huawei Mate 10 Pro	2.36+1.8 GHz (Octa)	6GB	Android 8.0
Huawei Mate 8	2.3+1.8 Ghz (Octa)	3GB	Android 6.0
LG Nexus 5X	1.4+1.8 GHz (Hexa)	2GB	Android 8.1.0
Sony Xperia Z5	1.5+2.0 GHz (Octa)	3GB	Android 6.0

4.3.4 Data Analysis

In total, $n = 16\,290$ individual data points were manually gathered for this experiment. Of this, 4 320 data points are related to time-to-completion (`TTC`) metric, 3 990 to `CPU` load, 3 990 to idle-state memory usage (`PreRAM`), and 3 990 to memory usage during benchmarking (`RAM`). Additionally 3 990 data points were analysed, which are the arithmetic computations of `RAM` subtracted from `PreRAM`, providing the actual memory impact of executing a given benchmark (`ComputedRAM`). The difference in n data points between time-to-completion and the other metrics is due to issues with the NativeScript-based implementation on one of the benchmark devices for which detailed performance profiling was unavailable.

The resulting raw data was then statistically analysed in order to identify differences between the frameworks. The native Java implementation served as a baseline to which all other frameworks are compared. By respecting current best practices and state-of-the-art system APIs, one could assume a high performance (*i.e.*, low utilisation of resources and fast execution times) for this implementation. However, frameworks might utilise highly optimised modules that do not make use of the Android (Java) SDK but rely on low-level C++ code which can potentially outperform the native baseline.

In the following section, descriptive statistics such as mean, minimum, and maximum values as well as the respective standard deviation are provided for each combination of feature and framework. For assessing a potential significance of variance in the results, ANOVA tests ($\alpha = .05$) along with effect sizes using omega squared (ω^2) were performed, following the interpretations provided by Kirk (1996). The effect size provides an indication of what percentage of the variance between two groups can be explained by the independent variable. While the ANOVA can provide on whether or not two (or more) groups are statistically significantly different from each other, the test does not help in determining where those significant differences are to be found. Thus, where the tests reported of statistical significance, this was followed up using Tukey post-hoc tests between each individual framework with results from the native implementation as a baseline standard. In each table provided in the subsections to follow, the p value column indicates the level of statistical significance to the native implementation results provided by the Tukey test.

Regarding the memory usage before (`PreRAM`) and during (`RAM`) the benchmarks, the ANOVA ($\alpha = .05$) was used, alongside Spearman's rank-order test to reveal any correlation between idle-state memory usage (`PreRAM`) and `ComputedRAM`.

4.4 Evaluation of Bridge Performance

This section firstly assesses overall performance results independent of individual tasks and devices. Consequently, the bigger picture is explored: how do the frameworks perform in terms of overall time-to-completion, `CPU` usage, as well as idle-state and during-task memory occupancy. For Figures 4.2, 4.3, 4.4, 4.5, 4.6 and 4.7, regular outliers are denoted by a black circle (●), while more extreme outliers by an asterisk (*). Subsequently, each individual task is assessed on its performance across the frameworks scrutinised.

4.4.1 Time-to-completion

The time-to-completion (`TTC`) metric describe the duration between when a foreign function call is invoked from the cross-platform context of the application, and when the result of the call is returned via the bridging layer from the native side. From this metric, the speed-wise performance of each cross-platform framework can be evaluated, with the native implementation results as a baseline.

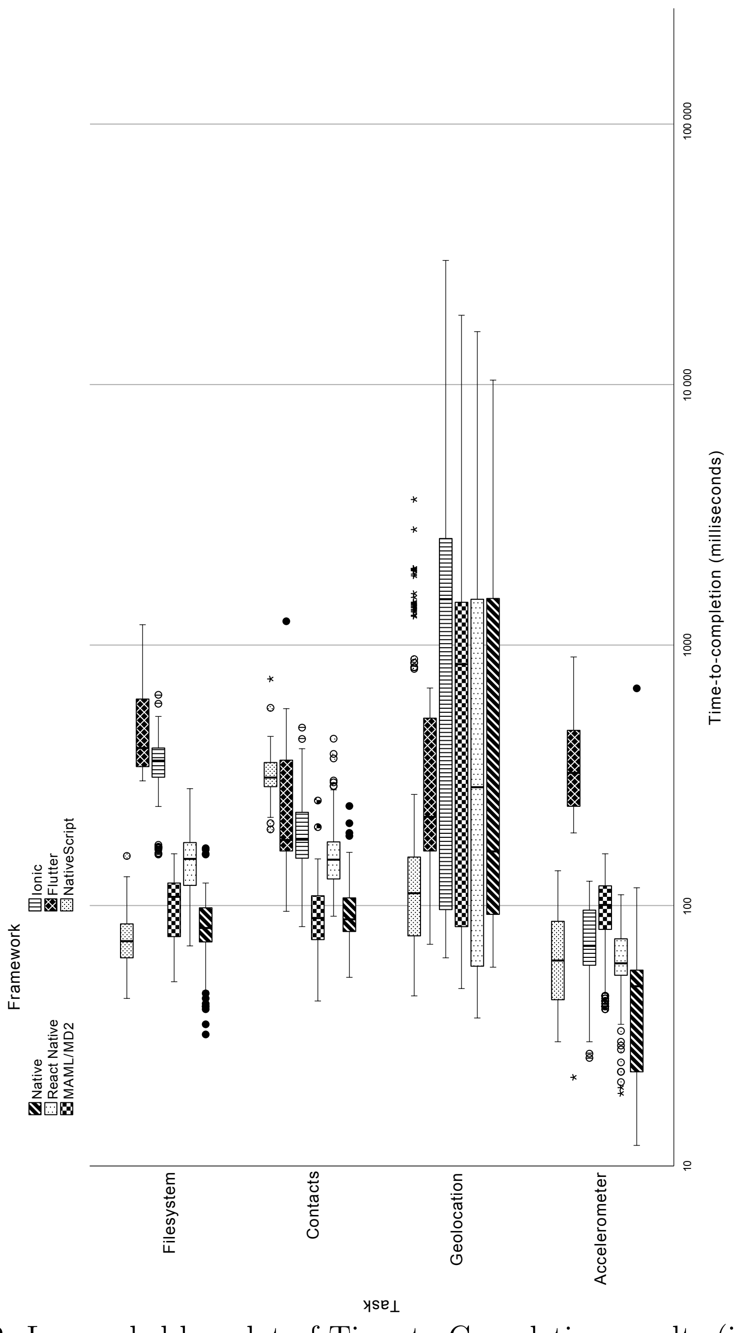


Figure 4.2: Log-scaled boxplot of Time-to-Completion results (in ms) per framework per task.

Figure 4.2 shows a boxplot of time-to-completion per framework per task and Figure 4.3 (p. 94) depicts a boxplot illustrating the time-to-completion between the different frameworks, regardless of task.

From a visual assessment of the results, a large number of outliers in the dataset are found. This could indicate that for the majority of the implementations, time-to-completion is highly fluctuating. Only NativeScript and Flutter did not to the same degree show the same fluctuating results; however, the Flutter implementation has an overall higher mean `TTC` than the other frameworks. Nevertheless, results from the Ionic benchmarks indicate that the framework may cross the 10 000 ms mark for fetching geolocation data more often than the other implementations.

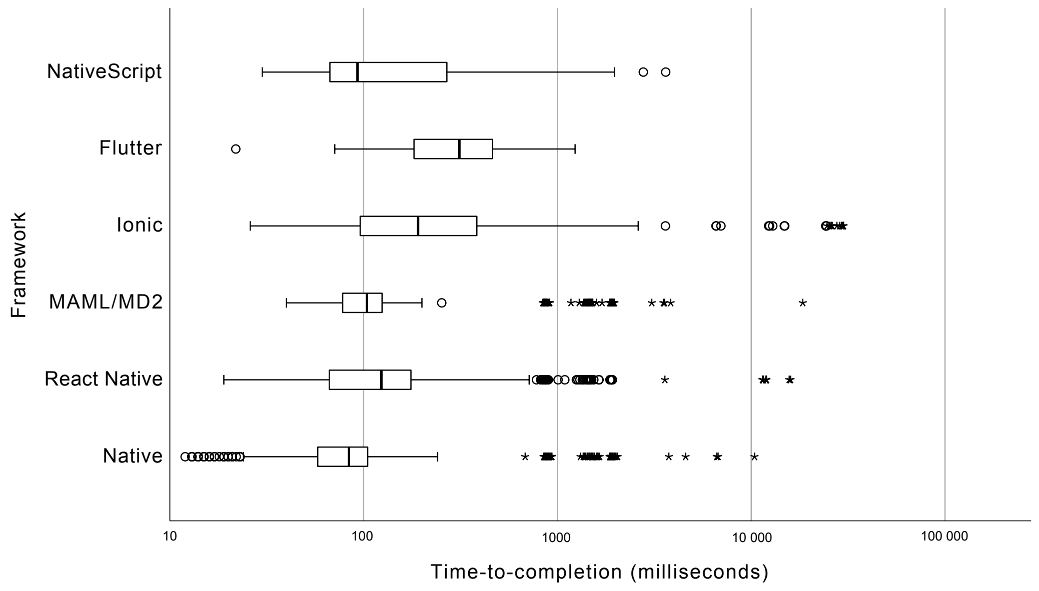


Figure 4.3: Log-scaled boxplot of Time-to-Completion results (in ms) per framework independent of task and device type.

In order to determine if differences in time-to-completion are statistically significant between the technical frameworks, a one-way ANOVA was conducted individually per framework with the native implementation as the baseline. As depicted in Table 4.3 (p. 95), the low values for $p < 0.01$ for all cross-platform frameworks indicate significance except for MAML/MD^2. This, however, aligns well with the intention of a model-driven framework that generates source code ideally indistinguishable from a native implementation.

Table 4.3: Overview of overall Time-to-Completion performance results.

	Descriptives				Analysis against native		
Framework	**Mean**	**SD**	**max**	**min**	**ANOVA** p	ω^2	**Tukey** p
Native	278.72	714.98	10401	12	-	-	-
React Native	656.54	2253.47	15968	19	< .001	.012	= .002
MAML/MD^2	295.20	846.71	18453	40	= .690	-.001	= 1.0
Ionic	1021.04	3762.86	29969	26	< .001	.018	< .001
Flutter	354.50	211.88	1234	22	= .006	.004	= .971
NativeScript	200.34	322.80	3620	30	= .007	.004	= .967

4.4.2 Memory Consumption

This experiment differentiates between the general memory usage as occupied by the app in an idle state (`PreRAM`), and the memory usage during benchmarking (`RAM`). In particular, the actual impact on memory usage caused by the task benchmarked can be assessed by subtracting the latter value from the former. That is if an app consumes 85MB `PreRAM` in idle state, and 100MB `RAM` during a task run, the memory impact for that task is 15MB – which is what the `ComputedRAM` metric reflects. Depending on the programming style and framework architecture, an app might seemingly use little additional memory for executing tasks but require much idle memory, *e.g.*, to continually hold some data structures in memory.

In Tables 4.4, 4.5 and 4.6, summaries of the overall memory usage are provided, in terms of `PreRAM`, `RAM` and `ComputedRAM` in that respective order. These results are independent of specific features and devices, and instead, provide a holistic perspective of the state of memory usage in the technical artefacts benchmarked. In Figure 4.4, a per-feature boxplot is provided, separated on framework. While still providing an overview, the boxplot also sheds light upon the differences in memory usage in more detail than the tables do. For instance, through visual assessment of Figure 4.4, the Ionic framework in general uses the most `ComputedRAM` memory, but also has the greatest variance.

The idle-state memory usage metric, `PreRAM` (cf. Table 4.4), is the

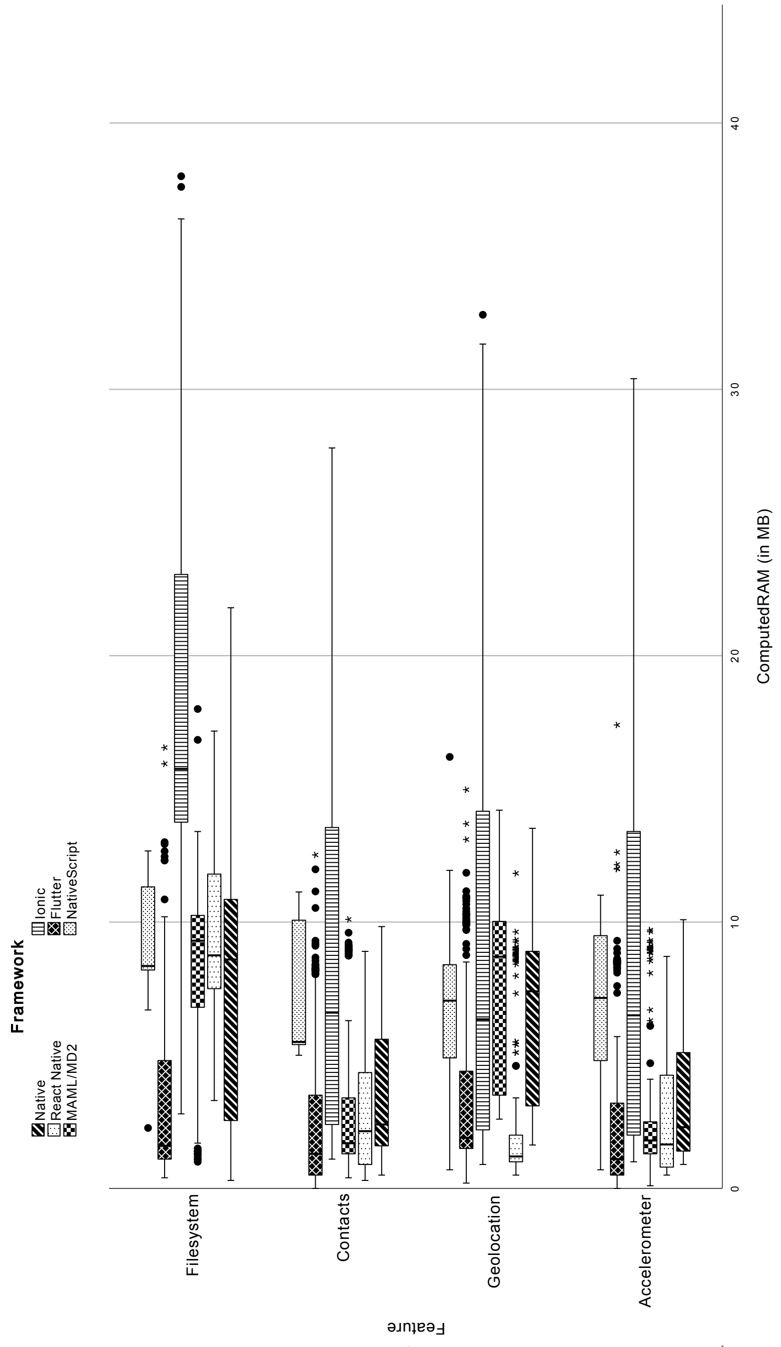


Figure 4.4: Linearly scaled boxplot of `ComputedRAM` results (in MB) per framework per task.

Table 4.4: Overview of `PreRAM` performance results.

	Descriptives				Analysis against native		
Framework	**Mean**	**SD**	**max**	**min**	**ANOVA** p	ω^2	**Tukey** p
Native	49.53	17.03	84.80	15.82	-	-	-
React Native	57.68	17.30	92.80	25.39	$< .001$	.053	$< .001$
MAML/MD^2	51.96	16.47	85.60	26.43	$= .001$	.005	$= .233$
Ionic	93.78	20.75	125.40	20.75	$< .001$	.576	$< .001$
Flutter	101.67	31.51	168.20	29.94	$< .001$	.514	$< .001$
NativeScript	63.93	15.48	87.60	37.03	$< .001$	.147	$< .001$

Table 4.5: Overview of `RAM` performance results.

	Descriptives				Analysis against native		
Framework	**Mean**	**SD**	**max**	**min**	**ANOVA** p	ω^2	**Tukey** p
Native	55.01	16.88	95.70	29.34	-	-	-
React Native	62.22	17.21	99.50	29.99	$< .001$	.042	$< .001$
MAML/MD^2	57.39	16.40	96.50	36.59	$= .007$	.004	$= .269$
Ionic	105.72	24.08	156.50	50.79	$< .001$	.598	$< .001$
Flutter	104.95	30.50	176.20	40.79	$< .001$	.506	$< .001$
NativeScript	71.49	13.79	92.60	41.52	$< .001$	.197	$< .001$

profiler-reported usage when the app is running on the device and have navigated to the respective benchmark's view, but prior to running the benchmark. This way, any potential overhead that cross-platform frameworks impose on the memory occupancy at runtime when compared with the native baseline results can be measured. As expected, the native baseline has the

Table 4.6: Overview of `ComputedRAM` performance results.

	Descriptives				Analysis against native		
Framework	**Mean**	**SD**	**max**	**min**	**ANOVA p**	ω^2	**Tukey p**
Native	5.48	4.13	21.80	0.30	-	-	-
React Native	4.53	4.13	17.17	0.30	$< .001$	.012	$= .006$
MAML/MD^2	5.42	3.84	18.00	0.10	$= .777$	-.001	$= 1.0$
Ionic	11.93	9.05	38.00	0.90	$< .001$	.173	$< .001$
Flutter	3.27	3.48	17.40	0.00	$< .001$	.076	$< .001$
NativeScript	7.56	2.61	16.20	0.70	$< .001$	.067	$< .001$

lowest reported `RAM` (cf. Table 4.5) usage. Across all tasks, Flutter has the highest idle-state memory usage of the studied frameworks, up to a tenfold increase for the geolocation task compared to native.

To account for possible correlations between `PreRAM` and `ComputedRAM` (cf. Table 4.6), Spearman's rank-order correlation coefficient test was conducted for each cross-platform implementation against the native baseline. While all results are statistically significant, the size of correlation varies, as presented in Table 4.7. To discuss the strength of the correlation size, the *rule of thumb interpretation* by Hinkle et al. (1988) is followed. From the results in Table 4.7, we find that Ionic is the only framework with a positive correlation, although less than $r_s = .3$ which according to Hinkle et al. should be interpreted as a negligible correlation. The only non-negligible correlation identified is that between native and Flutter, where Flutter has a low negative correlation. Looking at Flutter's results in Tables 4.5 and 4.6, this could indicate that while Flutter has a high `PreRAM`, the impact on memory usage caused by executing the benchmark task – `ComputedRAM` – is low.

Table 4.7: Results from Spearman's Rank-Order Correlation Coefficient tests on `PreRAM` and `ComputedRAM` against the native baseline implementation results.

Framework	**Observations** (n)	**Correlation coefficient** (r_s)	**Significance level** (p)
React Native	$r_s(1440)$	-.166	$< .001$
MAML/MD^2	$r_s(1440)$	-.162	$< .001$
Ionic	$r_s(1440)$	.286	$< .001$
Flutter	$r_s(1440)$	-.424	$< .001$
NativeScript	$r_s(1100)$	-.117	$< .001$

Figure 4.5 shows the linearly scaled boxplot for `ComputedRAM` usage in megabytes across all tests and devices per framework. We can observe from the figure that Flutter has a consistent low memory usage, although with a significant amount of high outliers. Also, results for Ionic show a considerable variation. NativeScript has the second-highest mean `RAM` usage, but with the lowest standard deviation. Only MAML/MD^2 exhibits no significant deviation from the native implementation.

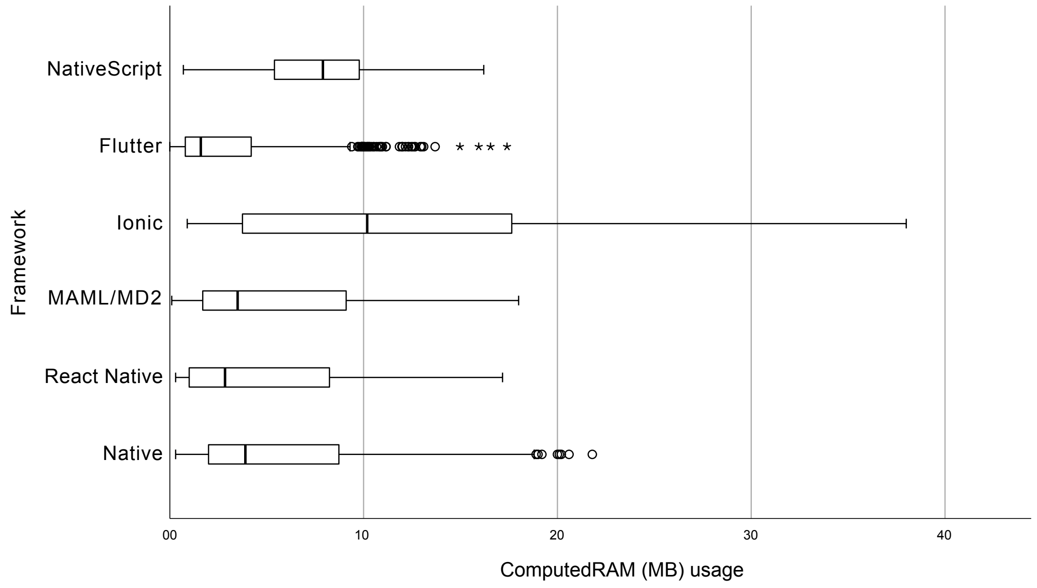


Figure 4.5: Linearly scaled boxplot of `ComputedRAM` usage (in MB) across all tests and devices per framework.

4.4.3 CPU Usage

In this experiment, the `CPU` usage across all the frameworks as laid out in Table 4.8 is measured. The mean values were quite concentrated, with React Native and Ionic standing out as less efficient. Although `CPU` usage will be heavily framework dependent, it introduces a possibility to see the impact of the individual frameworks from the tests.

Figure 4.6 – the boxplot of `CPU` usage – shows the use in percent across all tests and devices per framework. MAML/MD^2 comes out as the winner, although quite a few outliers can be observed. NativeScript performs with the most concentrated values and no particular outliers exposed. The mean

Table 4.8: Overview of CPU performance results.

	Descriptives				Analysis against native		
Framework	**Mean**	**SD**	**max**	**min**	**ANOVA** p	ω^2	**Tukey** p
Native	17.50	8.24	49.60	5.90	-	-	-
React Native	23.17	13.57	66.60	1.80	$< .001$	.059	$= .000$
MAML/MD^2	16.11	8.12	45.90	5.90	$= .001$	.006	$= .101$
Ionic	22.35	11.11	59.93	0.09	$< .001$	.057	$= .000$
Flutter	19.60	9.64	56.75	0.00	$< .001$	.013	$= .001$
NativeScript	15.71	8.38	35.73	5.00	$= .001$	.010	$= .060$

values of NativeScript and MAML/MD^2 again outperform the native baseline implementation; however, differences are not significant according to the ANOVA test.

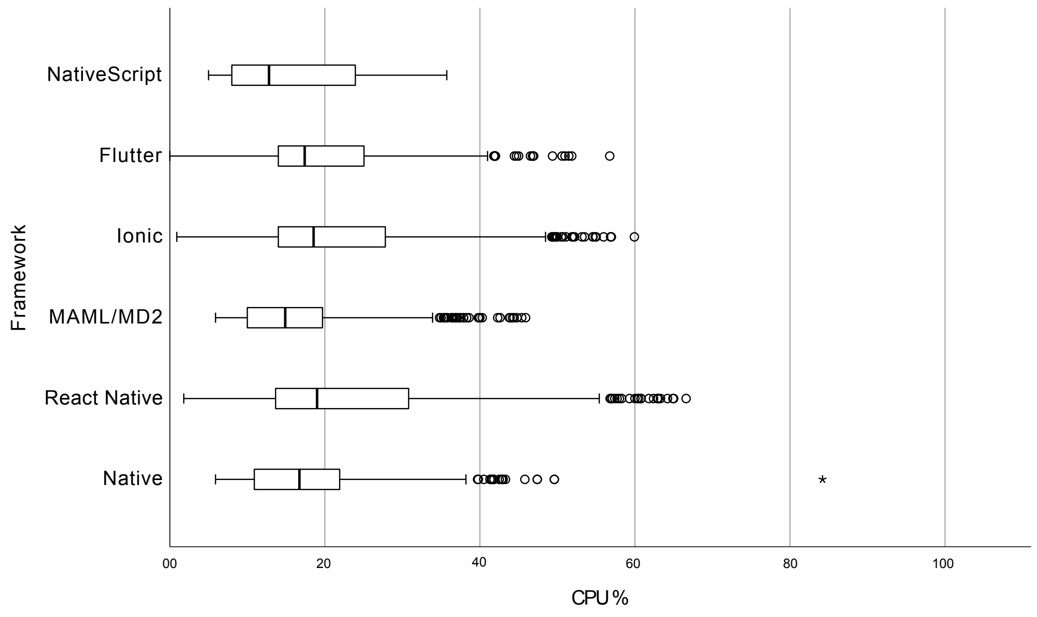


Figure 4.6: Linearly scaled boxplot of `CPU` usage (in %) across all tests and devices per framework.

While Table 4.8 and Figure 4.6 both provide `CPU` usage results independent of device and feature, Figure 4.7 provides a more detailed look into the various frameworks' per-feature `CPU` usage performance. Through visual assessment of Figure 4.7, especially the Contacts API had highly fluctuating `CPU` usage across all but the NativeScript-based implementation, indicating that for specific tasks, cross-platform frameworks may outperform the native baseline implementation in terms of reliability and consistency of performance results.

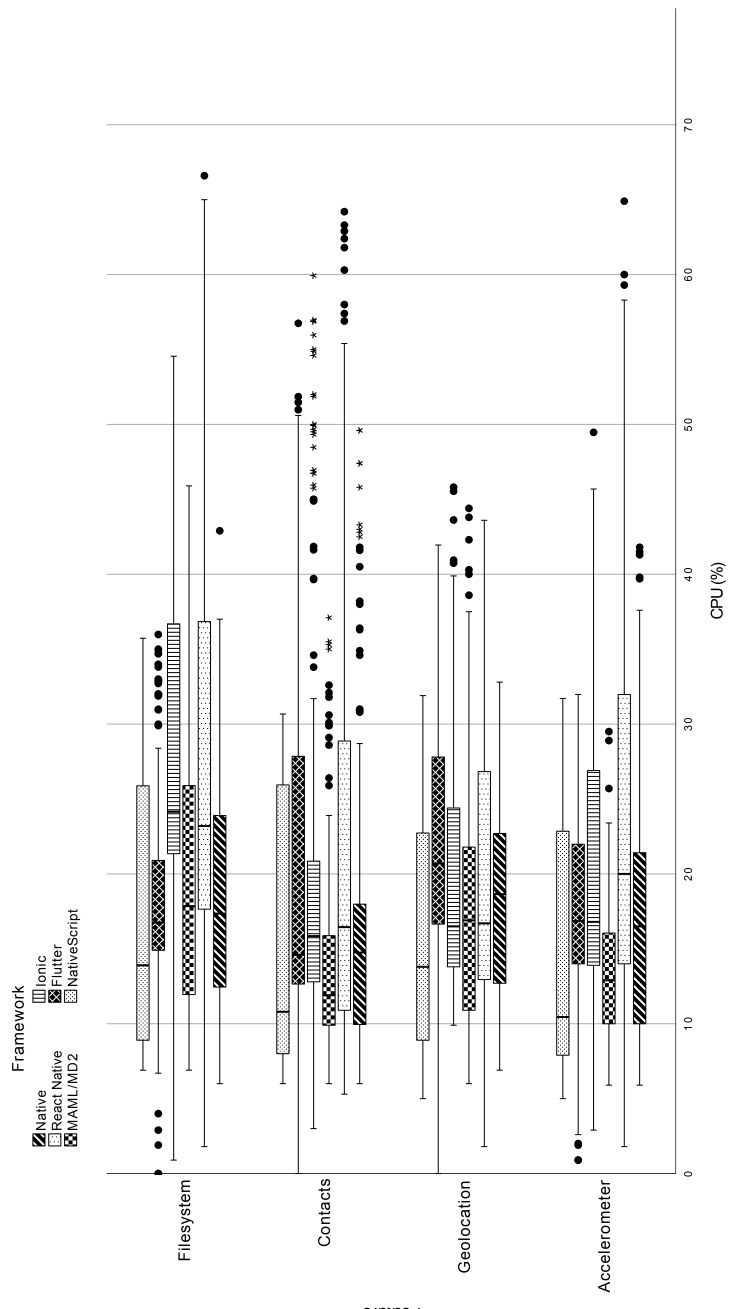


Figure 4.7: Linearly scaled boxplot of CPU results (in %) per framework per task.

4.4.4 Accelerometer

On Android, it is not possible to query the current value of the accelerometer through a platform-provided API call. Instead, the sensor sends system events when changes are detected, which can then be handled by appropriate event listeners. For the native application, the benchmark activity can directly register an event listener on benchmark start and access the sensor values from an upcoming update event. This results in the by far lowest time to completion. In contrast, MAML/MD^2 apps internally use a custom event-action cycle to handle the separation of user interface or data changes and their effect. Requesting the sensor value requires registering for an update of the SensorProvider, which in turn needs to wait for an upcoming sensor value update. Therefore, more time is required for the additional cycle, which is reflected in a two times slower retrieval as depicted in Table 4.9.

For the React Native implementation, it was experimented using the option `updateInterval` provided by the observable-based accelerometer plugin. While the default interval was 100ms, it was found to directly effect the benchmark results, consistently reporting ~100ms results. Lowering the interval to 0ms, the app would become unresponsive. However, at a 50ms interval, the accelerometer benchmark reported values both above and below the set interval, rendering it more similar to the other implementations.

Results from benchmarking the accelerometer sensor indicate that the framework choice has a statistically significant impact on performance across all considered metrics, although with a varying effect size. Below, each metric is investigated in detail to uncover differences in performance impact between the cross-platform frameworks using results from the native implementation as the baseline.

Time-to-Completion for Accelerometer

By inspecting the Tukey post-hoc results in Table 4.9, it is evident that MAML/MD^2, Ionic, and Flutter are all statistically significantly different from the native implementation in terms of time-to-completion results for the accelerometer task. React Native and NativeScript are reported as non-significant in the same context. Based on the descriptive statistics, the native implementation has the lowest mean and the lowest reported minimum value but also exhibits the second-highest standard deviation. React Native and NativeScript both have low mean and standard deviation values which indicate a consistent accelerometer performance among the implementations benchmarked. The Flutter implementation is the furthest away from the native implementation for this feature, with a seven-fold mean value compared to native, the highest standard deviation, and the maximum value in absolute terms.

$$Time : F(1074, 5) = 657.217, p < .001, \omega^2 = .752 \tag{4.1}$$

Table 4.9: Results per framework on accelerometer performance, metric: Time-to-Completion (ms).

Framework	**Mean**	**SD**	**max**	**min**	**Tukey *p***
Native	48.1	51.87	682	12	-
React Native	65.1	20.26	110	19	$= .095$
MAML/MD^2	97.3	30.48	158	40	$< .001$
Ionic	76.3	23.75	124	26	$< .001$
Flutter	357.2	133.25	900	22	$< .001$
NativeScript	65.5	24.30	136	30	$= .082$

CPU Usage for Accelerometer

Looking to the Tukey post-hoc results in Table 4.10, the benchmark results from React Native and MAML/MD^2 are statistically significantly different from the native implementation, while Ionic, Flutter, and NativeScript are non-significant. Results from the MAML/MD^2 implementation indicate that it has a lower mean `CPU` usage, lower standard deviation, and a lower maximum value than any other implementation, including the native baseline. React Native, on the other hand, uses more `CPU` capacity than all other considered frameworks, both in mean and maximum values. The most native-like results are here provided by Flutter, with a mean value, standard deviation, and maximum value close to the native implementation, although with a lower minimum value.

$$CPU : F(984, 5) = 29.527, p < .001, \omega^2 = .126 \tag{4.2}$$

Table 4.10: Results per framework on accelerometer performance, metric: `CPU` (%).

Framework	**Mean**	**SD**	**max**	**min**	**Tukey *p***
Native	17.2	8.52	41.8	5.9	-
React Native	23.6	13.00	64.9	1.8	$< .001$
MAML/MD^2	13.2	4.65	29.5	5.9	$< .001$
Ionic	20.0	8.80	49.5	2.9	$= .027$
Flutter	18.4	7.30	31.9	.9	$= .791$
NativeScript	14.6	8.30	31.7	5.0	$= .227$

PreRAM Usage for Accelerometer

The Tukey test in Table 4.11 indicates that all but one implementation are statistically significantly different from the native implementation results. MAML/MD^2 has the most native-like usage of `PreRAM`. On the contrary, Flutter has the highest mean, highest standard deviation, and the highest maximum and minimum values. Closely following Flutter is the Ionic implementation, which shares similarities regarding high values across all statistical columns.

$$PreRAM : F(984, 5) = 224.237, p < .001, \omega^2 = .530 \qquad (4.3)$$

Table 4.11: Results per framework on accelerometer performance, metric: `PreRAM` (MB).

Framework	**Mean**	**SD**	**max**	**min**	**Tukey *p***
Native	46.95	15.80	78.9	23.3	-
React Native	58.60	17.55	92.8	37.8	$< .001$
MAML/MD^2	50.98	16.83	81.7	27.0	$= .434$
Ionic	95.06	21.44	125.4	59.8	$< .001$
Flutter	101.01	30.23	168.2	62.9	$< .001$
NativeScript	61.05	15.93	81.9	37.0	$< .001$

ComputedRAM Usage for Accelerometer

The Tukey tests in Table 4.12 reports that React Native, MAML/MD^2, and Flutter are statistically non-significant compared to native, while Ionic and NativeScript are significantly different. Both Ionic and NativeScript have higher means, although only Ionic has a higher standard deviation, and a three-fold maximum value compared to native. React Native and MAML/MD^2 are comparable in performance, although their mean values are lower than the native implementation, the standard deviation comparable, and maximum values lower. The mean values indicate that Flutter has the lowest mean `ComputedRAM` usage, although this needs to be seen in relation to the highest mean `PreRAM` usage discussed in the previous section.

$$ComputedRAM : F(984, 5) = 69.015, p < .001, \omega^2 = .256 \qquad (4.4)$$

Table 4.12: Results per framework on accelerometer performance, metric: `ComputedRAM` (MB).

Framework	**Mean**	**SD**	**max**	**min**	**Tukey *p***
Native	3.58	2.76	10.1	.9	-
React Native	3.00	2.80	8.7	.5	= .815
MAML/MD^2	2.99	2.83	9.7	.1	= .808
Ionic	9.59	8.31	30.4	1.0	< .001
Flutter	2.77	3.42	17.4	.0	= .506
NativeScript	7.09	2.45	11.0	.7	< .001

4.4.5 Contacts

As for implementation and benchmarking challenges, when benchmarking on devices without SIM cards, the process of creating and saving new contacts could fail without any exceptions thrown by the development framework. Using the `adb logcat` CLI tool to inspect an unfiltered stream of logs from the device over USB, it was possible to manually look for relevant silent failures. The lack of a signed-in Google account was identified as the primary reason why the contacts API did not function as expected. This exception was mitigated by ensuring the device had indeed access to a signed-in Google account.

Results from benchmarking contacts performance indicate that the framework employed has a statistically significant impact on performance across all metrics included, although with a varying effect size. Below, each metric is investigated in detail to uncover differences in performance impact between the cross-platform frameworks using results from the native implementation as the baseline.

Time-to-Completion for Contacts

As reported in Table 4.13, all but the MAML/MD^2 based implementation are statistically significantly different from the native baseline results. For this benchmark, MAML/MD^2's native-like performance is indicated by a Tukey p close to 1.0. Mean-wise, the NativeScript implementation has the highest (worst) score with a three-fold increase in `TTC` compared to the native baseline, while MAML/MD^2's performance is, in fact, better than the native baseline by about 2 (two) milliseconds. While NativeScript has the highest mean value, Flutter's performance is seemingly the least consistent framework in this test, with a `TTC` varying from 95ms to 1 234ms.

$$Time : F(1074, 5) = 275.815, p < .001, \omega^2 = .560 \tag{4.5}$$

Table 4.13: Results per framework on contact performance, metric: Time-to-Completion (ms).

Framework	**Mean**	**SD**	**max**	**min**	**Tukey** p
Native	95.43	26.93	241	53	-
React Native	159.61	52.40	437	91	< .001
MAML/MD^2	93.06	26.55	253	43	= .0.999594
Ionic	193.58	64.70	482	83	< .001
Flutter	242.31	135.64	1234	95	< .001
NativeScript	321.07	61.41	741	196	< .001

CPU Usage for Contacts

Looking at the descriptive statistics in Table 4.14, only two implementations have statistically significantly different means compared to the native implementation, namely React Native and Ionic. In this benchmark, the MAML/MD^2 implementation has a lower mean `CPU` utilisation, lower standard deviation, lower maximum value and equal minimum value to the native implementation.

$$CPU : F(984, 5) = 13.407, p < .001, \omega^2 = .059 \tag{4.6}$$

Table 4.14: Results per framework on contact performance, metric: `CPU` (%).

Framework	**Mean**	**SD**	**max**	**min**	**Tukey** p
Native	16.69	10.06	49.6	6.0	-
React Native	22.10	15.48	64.2	5.0	$< .001$
MAML/MD^2	13.69	6.80	37.1	6.0	$= .155$
Ionic	21.35	13.46	59.9	3.0	$= .003$
Flutter	19.09	12.64	56.7	0	$= .381$
NativeScript	15.61	8.54	30.7	6.0	$= .981$

PreRAM for Contacts

From Table 4.15, we find that the native implementation has the lowest mean, maximum and minimum values. As indicated by the Tukey post-hoc test, results from the MAML/MD^2 implementation closely resemble those of the native counterpart at a highly non-significant level of difference. All other implementations are statistically significantly different from the native baseline, with Flutter furthest away with the highest results across all metrics – in several cases a two-fold increase. The lowest standard deviation is found in the NativeScript implementation, although the mean memory usage is higher than in native, React Native and MAML/MD^2.

$$PreRAM : F(984, 5) = 215.704, p < .001, \omega^2 = .520 \qquad (4.7)$$

Table 4.15: Results per framework on contact performance, metric: `PreRAM` (MB).

Framework	**Mean**	**SD**	**max**	**min**	**Tukey *p***
Native	48.18	17.30	79.7	24.8	-
React Native	58.43	17.89	87.2	30.5	$< .001$
MAML/MD^2	50.08	16.24	84.6	28.1	$= .954$
Ionic	93.50	21.48	119.1	37.3	$< .001$
Flutter	101.35	29.88	154.9	57.7	$< .001$
NativeScript	62.36	15.60	83.0	38.0	$< .001$

ComputedRAM for Contacts

While Flutter has the lowest reported mean and minimum values (cf. Table 4.16), the standard deviation is slightly higher than what is found in the native baseline results. Both React Native and MAML/MD^2 have results indicating native-like performance, while Ionic and NativeScript are both statistically significantly different from native. Ionic has the highest values across all metrics in this test.

$$ComputedRAM : F(984, 5) = 75.003, p < .001, \omega^2 = .272 \qquad (4.8)$$

Table 4.16: Results per framework on contact performance, metric: `ComputedRAM` (MB).

Framework	**Mean**	**SD**	**max**	**min**	**Tukey *p***
Native	3.72	2.81	9.8	.5	-
React Native	3.13	2.74	8.9	.3	= .749
MAML/MD^2	3.15	2.84	10.1	.4	= .780
Ionic	9.46	7.50	27.8	1.1	< .001
Flutter	2.75	3.19	12.5	0	= .224
NativeScript	7.08	2.37	11.1	5.0	< .001

4.4.6 File System Access

File system access frequently occurs when data such as images are stored on-device or on the external flash storage (as opposed to the system-provided database which can be used to store structured data in the order of magnitude below 1MB per entry). Typically, file system access is performed asynchronously to avoid blocking the main UI thread until the data is persisted or retrieved. For this task, it is worth noting that no asynchronous interface was available in NativeScript for accessing the file system. Thus, the only option was to make use of the synchronous interface, rendering the implementation of the app somewhat different than those for the other apps.

Time-to-Completion for File System Access

As reported in Table 4.17, NativeScript has the most native-like performance, showing even lower mean, standard variation, and maximum values compared to native. The MAML/MD^2 implementation has the third-best performance in the test and cannot be regarded as statistically different from native. Results indicate that Flutter has the overall highest values across all metrics, with a six-fold increase in mean time-to-completion compared to the baseline.

$$Time : F(1074, 5) = 459.062, p > .001, \omega^2 = .680 \tag{4.9}$$

Table 4.17: Results per framework on file system performance, metric: Time-to-Completion (ms).

Framework	**Mean**	**SD**	**max**	**min**	**Tukey** p
Native	82.34	22.93	166	32	-
React Native	154.74	45.79	281	70	< .001
MAML/MD^2	103.38	25.84	158	51	= .520
Ionic	360.35	87.84	644	157	< .001
Flutter	528.02	263.91	1197	301	< .001
NativeScript	75.58	19.18	155	44	= .994

CPU Usage for File System Access

As reported in Table 4.18, across MAML/MD^2, Flutter, and NativeScript, the differences from the native baseline are minimal. NativeScript has the lowest mean and maximum values, although higher standard deviation and minimum values than native. Both React Native and Ionic are statistically significantly different from the native implementation, with the former having the lowest mean usage, but the highest standard deviation (cf. Table 4.18).

$$CPU : F(984, 5) = 38.270, p < .001, \omega^2 = .158 \tag{4.10}$$

Table 4.18: Results per framework on file system performance, metric: `CPU` (%).

Framework	**Mean**	**SD**	**max**	**min**	**Tukey** p
Native	18.20	7.33	42.9	6	-
React Native	27.45	14.38	66.6	1.8	< .001
MAML/MD^2	19.68	9.53	45.9	6.9	= .739
Ionic	28.19	10.82	54.6	.9	< .001
Flutter	18.93	7.51	36.0	0	= .984
NativeScript	17.00	9.24	35.7	6.9	= .944

PreRAM Usage for File System Access

By inspecting the mean variation in Table 4.19, we find significant differences in minimal and maximal memory requirements between the frameworks. While React Native, MAML/MD^2 and NativeScript are relatively close to the native implementation in terms of `PreRAM` usage, both Ionic and Flutter consume a statistically significantly larger amount of memory, the latter close to a two-fold increase compared to native.

$$PreRAM : F(984, 5) = 180.822, p < .001, \omega^2 = .476 \tag{4.11}$$

Table 4.19: Results per framework on file system performance, metric: `PreRAM` (MB).

Framework	**Mean**	**SD**	**max**	**min**	**Tukey *p***
Native	54.08	17.59	84.8	26.6	-
React Native	57.37	16.84	87.7	36.1	= .685
MAML/MD^2	55.87	16.43	83.8	30.2	= .968
Ionic	93.91	19.67	120.1	64.7	< .001
Flutter	103.01	33.56	167.6	29.9	< .001
NativeScript	63.98	14.67	83.9	43.0	= .004

ComputedRAM Usage for File System Access

For this test, Ionic particularly stands out as performing relatively subpar when compared to the native implementation, with significant variations from max 38.0 to min 2.8, observing a standard deviation of 7.92 more than double the size relative to all other frameworks but the native performance (cf. Table 4.20) The MAML/MD^2 implementation has the closest-to-native performance, with a Tukey p of 1.0. While both Ionic and Flutter are statistically significantly different from the native baseline on mean values, the Flutter implementation has a much lower mean than native.

$$ComputedRAM : F(984, 5) = 174.612, p < .001, \omega^2 = .467 \quad (4.12)$$

Table 4.20: Results per framework on file system performance, metric: `ComputedRAM` (MB).

Framework	**Mean**	**SD**	**max**	**min**	**Tukey p**
Native	8.07	5.46	21.8	.3	-
React Native	9.41	3.59	17.2	3.3	= .101
MAML/MD^2	8.14	3.23	18.0	1.0	= 1.000
Ionic	18.45	7.92	38.0	2.8	< .001
Flutter	3.83	3.72	16.6	.4	< .001
NativeScript	9.30	1.84	12.7	2.3	= .378

4.4.7 Geolocation

In contrast to the accelerometer sensor, the GPS module in Android is not just event-based, but access is provided through an intermediate location manager (`FusedLocationProvider` as used by the native implementation or `LocationManager`), which aggregates different location providers such as GPS or a WiFi network. The manager object directly supports querying for the last known location. To retrieve up-to-date location values and avoid using cached values, which are updated according to a system-controlled rate, the time to request a new location value through a high accuracy request is measured. Nevertheless, the origin of the retrieved value may be based on previous network information, the GPS sensor, or a fused value based on different sources and varying accuracy.

Time-to-Completion for Geolocation

There is an internal process of waking up the GPS sensor for power reasons which leads to multi-second delays until a location value is retrieved from the hardware sensor. This occurs independently of the data collection method of completely closing and restarting the app as Android's location manager applies its own criteria on when the GPS sensor is queried. Consequently, minimum and maximum values in time-to-completion depicted in Table 4.21 exhibit a wide variation for all frameworks.

The most prominent outlier is the Ionic-based implementation. As the code already implements the geolocation provider options as suggested by the Ionic team (Lynch, 2018), no further optimisation of code or the underlying geolocation plug-in was implemented. Implementing optimisations for one framework would necessitate similar optimisations also in the remaining artefacts, but this was not the objective of this experiment. From searching for information on the issue, numerous questions regarding the geolocation feature in Ionic were encountered, which is also noted as the motivation behind the work of Lynch (2018). Thus, the results should be treated as what one would expect of the framework without any optimisations.

$$Time : F(1074, 5) = 23.997, p < .001, \omega^2 = .096 \tag{4.13}$$

Table 4.21: Results per framework on geolocation performance, metric: Time-to-Completion (ms).

Framework	**Mean**	**SD**	**max**	**min**	**Tukey p**
Native	889.05	1244.50	10401	58	-
React Native	2246.74	4122.66	15968	37	= .002
MAML/MD^2	887.07	1551.66	18453	48	= 1.000
Ionic	3453.94	6991.73	29969	63	< .001
Flutter	291.18	165.98	684	71	= .560
NativeScript	339.24	588.07	3620	45	= .560

CPU Usage for Geolocation

Drawing from the results presented in Table 4.22, only Flutter is statistically significantly different from the native implementation in terms of mean performance. While MAML/MD^2 has the most native-like performance, with a Tukey p value of 1.0, NativeScript has a lower mean CPU usage than native although with a slightly higher standard deviation. Minimum and maximum values of the NativeScript implementation closely resemble those also reported by the native implementation.

$$CPU : F(1014, 5) = 9.273, p < .001, \omega^2 = .041 \tag{4.14}$$

Table 4.22: Results per framework on geolocation performance, metric: CPU (%).

Framework	**Mean**	**SD**	**max**	**min**	**Tukey p**
Native	17.94	6.64	32.8	6.9	-
React Native	19.58	9.58	43.6	1.8	= .458
MAML/MD^2	17.83	8.73	44.4	6.0	= 1.000
Ionic	19.84	8.56	45.8	9.9	= .290
Flutter	22.00	9.80	41.9	0	< .001
NativeScript	15.65	7.55	31.9	5.0	= .215

PreRAM Usage for Geolocation

Drawing from the results in Table 4.23, MAML/MD^2 is relatively close to the native implementation in terms of `PreRAM` usage, although having a slightly higher mean. Both Ionic and Flutter require significantly more memory than native, at close to- or above a twofold increase. React Native and NativeScript are closer to native than the two frameworks previously mentioned.

$$PreRAM : F(1014, 5) = 207.898, p < .001, \omega^2 = .504 \tag{4.15}$$

Table 4.23: Results per framework on geolocation performance, metric: `PreRAM` (MB).

Framework	**Mean**	**SD**	**max**	**min**	**Tukey *p***
Native	48.91	16.66	79.1	15.8	-
React Native	56.34	16.97	86.1	25.4	= .009
MAML/MD^2	50.94	15.90	85.6	26.4	= .940
Ionic	92.66	20.49	125.1	20.5	< .001
Flutter	101.34	32.46	165.4	37.0	< .001
NativeScript	67.24	15.24	87.6	38.0	< .001

ComputedRAM Usage for Geolocation

As reported in Table 4.24, for this task, the implementations written in both React Native and Flutter are statistically significantly different from the native baseline in a positive fashion, as they both consume less memory. NativeScript's memory requirement resembles that of the native implementation, while both MAML/MD^2 and Ionic used more memory than the aforementioned frameworks, although the latter used significantly more.

$$ComputedRAM : F(1014, 5) = 59.239, p < .001, \omega^2 = .222 \tag{4.16}$$

Table 4.24: Results per framework on geolocation performance, metric: `ComputedRAM` (MB).

Framework	**Mean**	**SD**	**max**	**min**	**Tukey** p
Native	6.56	2.92	13.5	1.6	-
React Native	2.60	2.91	11.8	0.5	< .001
MAML/MD^2	7.40	3.20	14.2	2.6	= .542
Ionic	10.26	9.18	32.8	.9	< .001
Flutter	3.74	3.48	16.0	.2	< .001
NativeScript	6.97	2.84	16.2	.7	= .976

4.5 Discussion

To provide a general overview of the development technologies' performance, they were weighted based on each measurement metric's mean value. The framework with the lowest mean is assigned a score of 6 (best), the highest mean a score of 1 (worst), and the remaining scores are assigned in the same order to the remaining frameworks. This inverse ranking of the six considered technologies allows us to identify the overall performance-wise best- and worst-scoring technologies in this investigation. For this weighting, there is no separation between the different mobile devices or benchmarking tasks (*e.g.*, geolocation, accelerometer, *etc.*). Instead, the weighting is based on the results presented in Sections 4.4.1, 4.4.2 and 4.4.3.

Table 4.25: Weighting of frameworks on bridge performance, ordered by sum $\sum$ (higher is better).

Framework	`TTC`	`CPU`	`PreRAM`	`RAM`	`ComputedRAM`	$\sum$
Native	5	4	6	6	3	24
MAML/MD^2	4	5	5	5	4	23
NativeScript	6	6	3	3	2	20
React Native	2	1	4	4	5	16
Flutter	3	3	1	2	6	15
Ionic (Cordova)	1	2	2	1	1	7

Performance-wise, the model-driven MAML/MD^2 framework closely resembles the overall $\sum$ of the native development approach, according to Table 4.25. This is unlike several of the more industry-adopted frameworks scoring lower on the weighting. Indeed it is surprising that there is a seeming lack of industry adoption of model-driven frameworks (Biørn-Hansen et al., 2019b) when they perform so close to the native performance baseline. Thus, findings derived from this current experiment may imply that practitioners and industry decision-makers should look more towards the model-driven approach, based on the performance results presented in this experiment.

As Table 4.25 illustrates, the hybrid approach-based Ionic framework (relying on Cordova for native access) has the overall lowest score. It ranks lowest on three of five metrics, however, still outperforms React Native on `CPU` usage and Flutter on `PreRAM` usage by one point each. The framework's overall ranking is in line with previous studies on performance in cross-platform applications, including El-Kassas et al. (2016), Katevas et al. (2016), and Abousaleh et al. (2014).

Flutter has `PreRAM` and `RAM` scores comparable to the Ionic framework's results, however has the best score for `ComputedRAM`. This could indicate that while Flutter has high overall memory requirements, the effect on the memory usage when executing a task is lower than for the other frameworks. In terms of `TTC` and `CPU`, Flutter has an average score. No previous academic efforts to empirically investigating the performance of the Flutter framework have been identified.

What Table 4.25 best illustrates is some of the trade-off developers face. If time-to-completion is the most critical metric, thus adopting NativeScript, this will come at the cost of `ComputedRAM`. If minimising `ComputedRAM` is important, Flutter scores the highest, but also has the lowest `PreRAM` score, meaning it overall consumes the most memory prior to executing a task (geolocation, contacts, *etc.*). As a developer or decision-maker deciding on a cross-platform development framework, having a clear idea of product requirements and specifications is of paramount importance.

4.6 Conclusion

In the experiment described in this chapter, the performance overhead imposed by cross-platform mobile development frameworks in Android apps compared to the native development approach has been investigated. Specifically, the performance of native-side foreign function interface calls from a cross-platform context has been addressed, to invoke and run device and platform functionality, including geolocation API, contacts API, file system integration, and accelerometer integration. Data has been gathered on five metrics: `CPU` usage, idle-state memory consumption (`PreRAM`), during-benchmark memory consumption (`RAM`), the difference between `RAM` and `PreRAM` (`ComputedRAM`), and the lapsed time from invoking a benchmark task until data is returned from the native side (time-to-completion (`TTC`)).

In total, $n = 16\,290$ individual data points related to these metrics were manually gathered for analysis through a rigorous and time-consuming data collection process. Both the statistical analysis and a weighted evaluation of the results are used to investigate how well the developed artefacts perform compared to a developed native baseline artefact.

$Sub - RQ_{1.1}$**:** To what degree do cross-platform mobile development frameworks impose additional performance-related overhead when compared to native mobile development?

Evidently, the results presented in Table 4.25 suggest that using one of the cross-platform frameworks tested will impose additional performance overhead compared to native in the context of executing native-side functionality. The severity of this overhead, however, ranges from rather small in the case of MAML/MD^2, to more than threefold in the case of Ionic when compared on the final weighting $\sum$. Nevertheless, while the native approach has the highest overall $\sum$ (best performance output), other cross-platform frameworks were found to possibly be more performant on specific metrics, such as NativeScript's time-to-completion and `CPU` usage, and Flutter's minimal increase in memory usage during task benchmarking, as reported in the `ComputedRAM` column.

The results indicate that the use of cross-platform frameworks for the development of mobile apps may lead to decreased performance compared to the native development approach. Nevertheless, the results also indicate that specific cross-platform frameworks can perform equally well or even better than native on specific metrics but no framework scores best across all features in this experiment. These findings reinforce the importance of well-defined technical requirements and specifications, without which deciding on a cross-platform framework or overall development approach can potentially lead to underperforming apps. In the upcoming chapter, the performance of cross-platform apps is further scrutinised within the context of animations and user interfaces.

Chapter 5

Animation Performance of Mobile Development Approaches

This experiment investigates the performance of animated user interfaces in native and cross-platform mobile apps. The chapter firstly summarises the problem statement, research questions, and the experimental setup. Subsequently, the artefact development is described, followed by the evaluation of animation performance and a thorough discussion of results, metrics and tools.

5.1 Awareness of Problem: Performance of Cross-Platform User Interfaces

Revisiting the Objective. In the previous experiment described in Chapter 4, the performance of the underlying communication bridges which cross-platform frameworks use to communicate with native code and functionality was assessed. In the experiment described in the current chapter, the performance of cross-platform frameworks and apps is further scrutinised towards answering the first thesis research question (RQ_1) regarding performance as described in Section 1.4, but change the focus to the user interface, and the fluidity of animations. The review of related work in Chapter 2 indicated that user-focused research within cross-platform development is sparse and in need of additional empirical efforts to extend the body of knowledge.

Background and Literature Synopsis. Nowadays, it is common to encounter animated graphical user interfaces in mobile apps (Trapp and Yasmin, 2013), with use cases including those of branding purposes, communication of information and interaction patterns, training, and to enhance overall user experience (Huhtala et al., 2010). Thus, it should be safe to assume that animations and transitions are integral parts of modern user interfaces, performance- and view-estate constrained handheld mobile devices.

Communication of Research: This is an extended version of the publication in MDPI Sensors - Special Issue on Mobile Computing and Ubiquitous Networking (2019a). The content and format has been modified to fit the thesis narrative. Replication package is available in Appendix C.

Although the importance of high-performing user interfaces is clear (Reddi et al., 2018), little empirical research has been conducted on the performance of animated user interfaces in cross-platform apps. A longitudinal study on user experience and user perspectives by Angulo and Ferre (2014) indicated that there were differences between Android and iOS users on how they perceived cross-platform apps. Indeed, 91% of Android users and 79% of iOS users found the cross-platform app to behave as- or similar to- the native baseline app. This could, for instance, indicate that cross-platform frameworks do a better job implementing Android-specific user interface guidelines than for iOS. We find that mentions of the importance of user experience and native-like user interfaces flourish in the literature (*e.g.*, Heitkötter et al. (2012a), Lachgar and Abdali (2017), Latif et al. (2016)), alas research – especially of empirical nature – has not been identified to the same extent. This experiment targets parts of this gap, being that of empirical evaluation of the hardware-wise performance outputted by cross-platform development frameworks, and the tools used for analysing and gathering of performance measurement data.

5.2 Suggestion: Measure Animation Performance

This section is dedicated to presenting the research questions and the research methods employed, thereby the cross-platform frameworks scrutinised, equipment utilised, and an overview of performance metrics and tools used for measurement and data gathering. In order to investigate the performance of user interfaces in cross-platform frameworks, a systematic approach for conducting the experiments and data gathering is employed. The user interface performance of three cross-platform frameworks is benchmarked, with two native apps for the baseline performance comparisons. To ensure verification of results and the possibility of re-using technical artefacts in future research, the technical development conducted as part of this experiment has been open-sourced on Github (see Appendix C).

5.2.1 Research Questions

Based on the assessment of existing research on performance in cross-platform development, as presented in Chapter 2, three research questions for this experiment have been formed. These cover the suitability of the metrics included in the experiment, limitations imposed by profiling tools, and differences in performance between the mobile platforms. Together, they provide new knowledge used towards answering the first thesis research question,

***RQ_1*:** How do apps developed using cross-platform mobile development approaches and associated frameworks perform compared to native mobile apps in terms of hardware and platform utilisation?

This experiment also assesses the metrics used, the tools used for profiling and insight, alongside the performance measurements. The research questions are answered as part of the discussion in Section 5.5, where each question is re-iterated and answered throughout the subsections. The three research questions for this experiment are as follows,

$Sub - RQ_{1.2}$**:** Do the performance metrics fulfil their purpose in researching animated user interfaces in mobile apps?

$Sub - RQ_{1.3}$**:** How well do the official performance insight tools cater to the profiling of animation and transition performance in the cross-platform apps developed?

$Sub - RQ_{1.4}$**:** Which of the platforms, iOS or Android, requires the least amount of device- and hardware resources in order to execute and run performant animations and transitions?

5.2.2 Technologies

To answer the research questions previously presented, a comprehensive assessment of performance has been conducted. A total of eight apps have been developed using the frameworks and technologies listed in Table 5.1: two native apps in addition to the three cross-platform frameworks which each generate an Android and an iOS app to ensure cross-platform deployability. The two native apps developed enable gathering of baseline performance results to accurately understand and measure how the cross-platform apps perform in comparison. Making any performance-oriented comparisons without the presence of baseline performance results would be inherently challenging. The use of native baselines is also found in other studies, such as in both Ciman and Gaggi (2016)'s energy consumption analysis study, and in Willocx et al. (2015)'s performance study.

Table 5.1: Frameworks and technologies scrutinised for animation performance.

Technology	**Version #**	**Approach**	**Language**	**IDE**
Native Android	26 (SDK)	Native	Kotlin	Android Studio
Native iOS	Xcode 9	Native	Swift	Xcode
React Native	0.49.5	Interpreted	JavaScript	Text editor
Ionic	3.15.2	Hybrid	JavaScript	Text editor
Xamarin Forms	2.4.0.38779	Cross-Compiled	C#	Xamarin Studio

It is important to note that no optimisation was conducted to any of the implementations or codebases. While one could do extensive work in order to optimise performance in all frameworks and technologies included in this experiment, such work is beyond the scope of the experiment. Only the performance delivered out of the box is of interest in this experiment, as this is what developers can expect to work with after initialising a new project.

5.2.3 Experimental Setup

The tests were conducted using the mobile devices listed in Table 5.2. At the time of the experiment, both devices had installed the latest operating system updates available. During the tests, both devices were set to maximum screen brightness, had mobile data, GPS and Bluetooth turned off, and did not have any other apps running in the background simultaneously. Each app was restarted prior to each experiment and task, this to help us generate results unaffected by previous benchmark runs.

Table 5.2: List of mobile devices included in the animation experiment.

Model	**Released**	**OS (version)**	**Memory**	**Processor**
LG Nexus 5X	2015	Android (8.0)	2GB	Snapdragon 808
Apple iPhone 6	2014	iOS (11.1.1)	1GB	Apple A8

5.2.4 Metrics and Data Gathering Tools

The following list of metrics has been recorded in this study, deemed of importance to properly scrutinise the performance and possible penalties introduced with the use of cross-platform development frameworks.

Metrics Measured

Frames-per-second (`FPS`). Suggested as an important metric for measuring fluidity of movement in graphical user interfaces (Basques, 2018, Reddi et al., 2018). Animations and transitions that are not displayed in a constant close-to-60 `FPS` can be perceived as *"janky"* according to some studies (*e.g.*, Lewis (2017)), a term used to describe partially unresponsive interfaces, or interfaces that underperform. Measuring jank is also suggested by Reddi et al. (2018). Contrastingly, at 60 FPS, an interface and its animated elements are fluently displayed (Lewis, 2017). The importance of measuring `FPS` is put to the test throughout this experiment. During the experiments, each animation and transition is executed twice and the data from the second run is used, this due to constraints in certain frameworks as loading the Lottie animation file contributed to some perceivable lag.

`CPU` usage. Apps generated in a cross-platform fashion have been reported to differ in terms of `CPU` usage between the various development frameworks (Ciman and Gaggi, 2016, Willocx et al., 2015). This work extends such previous work, and apply methods for measuring `CPU` usage specifically in the context of animations. Thus, this experiment fills a knowledge gap in terms of user experience of cross-platform generated apps, and the frequently encountered inclusion of `CPU` as a mean for measurement makes including it here as well ideal, also to discuss the suitability of the metric in Section 5.5.

Device memory (`RAM`) usage. Mobile devices are significantly and inherently constrained by the performance of the underlying hardware. Especially the availability of memory can hamper the user experience; hence the inclusion of the `RAM` usage measurement in the tests conducted.

`GPU` memory usage. Data on `GPU` memory usage was extracted for Android only. It was found that on iOS devices, `GPU` and `CPU` seem to share the same memory (Unity, n.d.) – alas, no method for proper extraction of `GPU`-only memory usage was identified. As such, Table 5.4 displaying the performance results does not present `GPU` memory usage for the iOS apps.

Tools Overview

A handful of tools were selected to measure the four performance-oriented metrics listed previously. For measuring iOS performance, all but one metric could be measured using *Instruments*, the official profiling tool from Apple. The one metric which could not be measured using Instruments was `GPU` memory usage, as briefly explained in the last section of 5.2.4. Measuring Android performance involved the use of both command-line interfaces (CLIs) and a graphical profiling tool, the former being `adb (Android Debug Bridge)` and the latter being Android Studio, the official development IDE by Google. Android Studio had previously been equipped with a `GPU` Monitor as part of its array of profiling tools, but it was deprecated and removed from the latest version of the IDE as of the time of writing (Android Developer, n.d.), hence the need for the CLIs to gather additional insights not provided by Android Studio.

Table 5.3: Tools associated with animation performance metrics.

Platforms	**Metrics**	**Tools**
Android	`FPS`	`adb systrace`
	`RAM`	Android Studio Profiler
	`CPU` %	Android Studio Profiler
	`GPU RAM`	`adb dumpsys`
iOS	`FPS`	Instruments: Core Animation
	`RAM`	Instruments: VM Tracker
	`CPU` %	Instruments: Time Profiler
	`GPU RAM`	-

5.3 Development of Artefacts

This section is dedicated to presenting the three animations and transitions implemented as parts of the artefacts, and an overview of the artefacts' visual design baseline. This experiment strives to present each animation and transition implementation in a level of detail rendering reproducibility of the artefacts possible[1].

[1]Note that all artefacts are open-sourced (see Appendix C).

5.3.1 Lottie Star Animation

Lottie is a library developed by the engineering team at Airbnb for the rendering of JSON-defined animations exported from Adobe After Effects. The library is cross-platform compatible, and can display animations in both native Android and iOS apps, as well as in Ionic[2], React Native[3], Xamarin[4] and more. An animation by Michael Harvey on Lottiefiles.com[5] was identified as the most popular open-source Lottie animation at the time of writing (33kB), containing multiple stages and elements, as depicted in Figure 5.1. The animation involves movement, appearing and disappearing elements, and is of a fast-paced nature. The animation was deemed attractive for scrutiny purposes due to the variety of elements and its popularity.

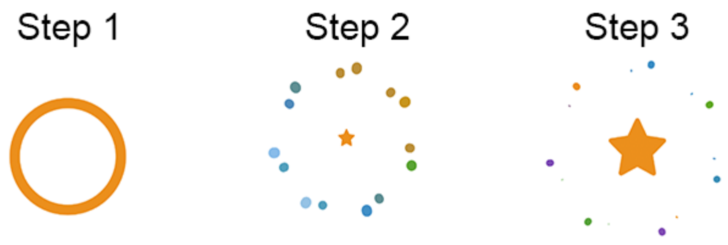


Figure 5.1: Three main steps of the Lottie animation.

[2]https://github.com/chenqingspring/ng-lottie
[3]https://github.com/airbnb/lottie-react-native
[4]https://github.com/martijn00/LottieXamarin
[5]https://www.lottiefiles.com/72-favourite-app-icon

5.3.2 Navigation Transition

When navigating between pages and views in mobile apps, transition animations are executed to provide a visual indication of an ongoing page (or context) switch. Such transitions vary between platforms and the type of navigation executed, for instance, a modal pop-up, a page switch adding to the navigation stack, or a replacement of the current view. In this experiment, the focus is on performance testing the transition animation going from one page to another. Inherently, the result from this is coloured by the actual navigation event's potential performance.

The frameworks and native technologies included in this study, as listed in Table 5.1, did, for the most part, integrate navigation functionality. Only React Native required a third-party navigation library, and there were several open-source alternatives. While both React Navigation[6] and React Native Navigation[7] provided easy navigation APIs, the latter was the only library exposing actual native navigation APIs and transitions, not JavaScript-based native-*like* mimics. As such, to make a fair comparison to the other frameworks tested, React Native Navigation was used for navigation. For further research, it could be of interest to study the differences between JavaScript-based and native transitions in the context of both device resource usage and user experience.

[6] https://reactnavigation.org/
[7] https://github.com/wix/react-native-navigation

5.3.3 Side Menu Animation

The opening of the side menu and its associated opening animation are typically triggered whenever a user presses the *hamburger menu icon*, *i.e.*, the icon in the very left image, consisting of three stacked horizontal lines. This navigation pattern is common in real-world implementations (Majchrzak et al., 2017).

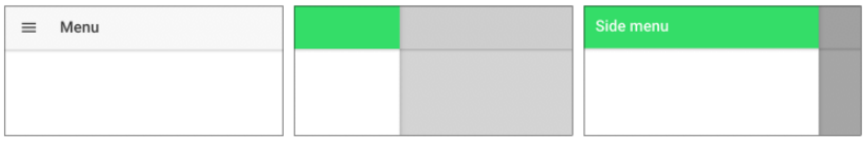


Figure 5.2: Example of a side-menu drawer opening sequence.

Achieving the intended behaviour using the frameworks at hand resulted in a variety of implementations, each one seemingly unique to the respective framework, as reported below.

Xamarin. An official example code base[8] for implementation of side menus was made use of, created by David Britch at Xamarin.

React Native. The React Native Navigation third-party library was used to enable view navigation. A side menu implementation was also available in the library, which was subsequently used in the implementation.

Ionic. Upon starting a new Ionic project, their CLI prompts the possibility

[8]Xamarin side menu repository on Github: https://developer.xamarin.com/samples/xamarin-forms/Navigation/MasterDetailPage/

of scaffolding a side menu-based project, which adds some required and related boilerplate code.

Native Android. The side menu-enabled template which was available upon project initialisation in the Android Studio IDE was utilised.

Native iOS. The side menu navigation pattern is not native to the iOS platform; thus, no such component existed in the Xcode development environment by default. A third-party library was found to compensate, which adds an easy-to-use API for providing side menus in Swift iOS, called SideMenu[9] written by developer Jon Kent.

A Visual Overview

Each of the applications developed inherit the same user interface layout and visual design. In Figure 5.3, the React Native application running on the Android platform is depicted to illustrate the visuals of the finished artefacts. At the top, a navigation bar including a "hamburger menu button" displays the title of the current view. Not depicted is the side menu drawer which will open upon pressing the hamburger menu button, but an example of this is illustrated in Figure 5.2. Below the navigation bar, following in a vertical stack-wise layout are the animation view (yellow star), a button to execute a play-through of the animation, and at the end a button to navigate to another page.

[9]SideMenu repository on Github: `https://github.com/jonkykong/SideMenu`

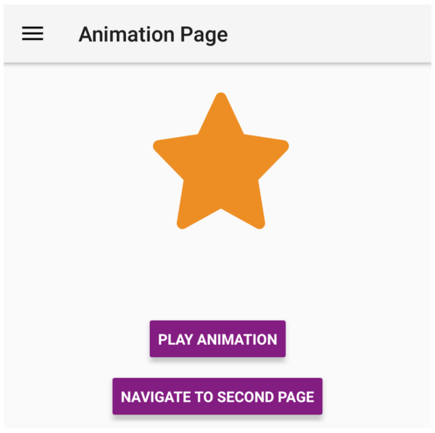


Figure 5.3: Cropped example of a developed artefact.

5.4 Evaluation of Animation Performance

The findings from the experiments have been condensed into Table 5.4, with additional notes on the findings located below the table. When encountering arrows ($\rightarrow$ *and* $\leftarrow$) in the table, these represent the value displayed in the direction of the arrow. To exemplify, in the case of the native iOS app during the Lottie animation playing, memory usage was reported to be 49.08MB before (pre), during and after (post) the execution of the animation. Thus, only the value *during* the animation is reported in numbers, with arrows replacing the value in the *pre* and *post* fields.

For those tasks whose running-time could not be programmatically calculated, such as the opening time of the React Native side menu on iOS, each framework's source code was traversed to extract the defined transition or animation duration. These extracts have been carefully marked using a question mark in the `FPS` column in Table 5.4.

A challenge encountered during the data-gathering phase was the differences in the apps' performance before conducting the benchmarks. An example of this from Table 5.4 is the performance of the native Android app, which prior to starting the Lottie animation reported of memory consumption of 62.34MB, while the reported consumption was 49.46MB prior to the navigation transition benchmark. Below follows a brief explanation of annotations in Table 5.4's header:

`FPS`. "*Count*$^{Dur.}$ *(jank)*" refers to the overall reported *count* of frames rendered, the ***dur****ation* of the animation in those cases where running time was less than a full second, and the number of *janky* frames as reported by the Android profiler tool.

`CPU` Peak. The results of this metric's measurements are reported in percentage, as they are by the profiling tools on both platforms.

Memory (`RAM`) Peak. Describes the `RAM` consumption prior to the animation running *(pre)*, *during* the animation, and after the animation is complete *(post)*.

Table 5.4: Results from animation performance tests.

Technology	FPS *Count$^{Dur.}$* *(jank)*	CPU **Peak**	**Memory Peak (MB)** *(pre) during (post)*	GPU **Memory** *(Android)*
		Lottie Star Animation Performance		
Nexus 5X (Android)				
Native	60 (16)	25.58%	(62.34) 68.49 (64.14)	2.33MB
React Native	60 (18)	21.62%	(73.47) 93.72 (~90)	3.45MB
Ionic	59 (26)	29.93%	(93.39) 124.27 (~116)	4.55MB
Xamarin Forms	60 (37)	19.95%	(116.7) 125.34 (117.08)	6.47MB
iPhone 6 (iOS)				
Native	30-23	40%	(→) 49.08 (←)	-
React Native	46-5	100%	(51.96) 53.56 (←)	-
Ionic	17-23	30%	(→) 67.75 (67.90)	-
Xamarin Forms	48-5	70%	(→) 91.67 (101.95)	-
		Navigation Transition Performance		
Nexus 5X (Android)				
Native	5^{328ms} (3)	29.34%	(49.46) 73.5 (59.06)	2.36MB
React Native	18^{340ms} (2)	22.82%	(68.84) 82.04 (80.40)	1.08MB
Ionic	21^{482ms} (14)	21.78%	(91.45) 132.76 (104.75)	4.55MB
Xamarin Forms	18^{377ms} (9)	20%	(105.78) 108.71 (107.21)	6.48MB
iPhone 6 (iOS)				
Native	18^{200ms}	100%	(→) 49.80 (←)	-
React Native	$28^{250ms?}$	100%	(52.14) 53.38 (←)	-
Ionic	$36^{500ms?}$	50%	(→) 66.65 (←)	-
Xamarin Forms	23-6^{586ms}	100%	(→) 103.23 (←)	-
		Side Menu Performance		
Nexus 5X (Android)				
Native	27^{443ms} (1)	7.93%	(64.53) 64.92 (64.67)	2.29MB
React Native	35^{573ms} (0)	11.73%	(69.20) 69.70 (69.60)	1.08MB
Ionic	32^{617ms} (8)	21.97%	(93.79) 114.79 (109.84)	4.55MB
Xamarin Forms	28^{458ms} (0)	13.86%	(103.55) ~87 (84)	6.36MB
iPhone 6 (iOS)				
Native	30-^{350ms}	67%	(37.63) 41.13 (←)	-
React Native	6-$4^{200ms?}$	70%	(→) 76.70 (←)	-
Ionic	14-$8^{280ms?}$	60%	(→) 67.14 (←)	-
Xamarin Forms	7-$3^{300ms?}$	75%	(→) 102.45 (←)	-

5.4.1 Lottie Star Animation Performance

All implementations on Android, including native, reported large numbers of janky frames during the execution and play-through of the Lottie animation. The native implementation saw the fewest janky frames rendered, used more `CPU` than React Native and Xamarin, but saw only a minor increase in `RAM` consumption. For this task, the increase in `RAM` consumption through all cross-platform frameworks illustrate the penalty introduced by using cross-platform technologies. Ionic's jump from 93.39MB to 124.27MB `RAM` consumption upon Lottie execution display that there are certainly trade-offs involved when developing cross-platform applications. Another negative aspect of the penalty is how memory consumption after specific tasks does not decrease to its original value. This is true for all the apps generated using cross-platform technologies, indicating a potential memory leak. While Xamarin produced the highest amount of janky frames (more than half of the attempted rendered frames), it used the least `CPU` but consumed the most `RAM`.

5.4.2 Navigation Transition Performance

For Android, it is interesting to note how the native app relies the heaviest on the `CPU`, in fact consuming 9.35% more `CPU` than the lowest-consuming app being the Xamarin-based implementation, although the latter having a much higher impact on RAM consumption at 108.71MB versus 73.5MB for the native app. The highest-consuming implementation in terms of RAM on Android is the Ionic app at 132.76MB, seeing a significant impact upon execution of the navigation transition task, increasing RAM consumption by 41.31MB. However, Ionic was also the framework to render the most frames of all the Android implementations, although with a high count of reportedly janky frames. It also peaked at a lower CPU consumption than the native app, although 1.78% higher than the Xamarin app.

On the other hand, Ionic on iOS outperformed both the native baseline and the other cross-platform frameworks on the FPS and CPU usage metrics while consuming more RAM (at 66.65MB) than native (49.80MB) and React Native (53.38MB), but far less than that of the Xamarin implementation at 103.23MB. However, one thing to note is that the native app conducts the navigation in 200ms, whereas the Ionic app – based on Ionic's source code – uses 500ms for the same task. Thus, the FPS count is higher for the native app due to the short play-through time, assuming that the length of the navigation is correctly reported.

5.4.3 Side Menu Performance

While the Ionic framework on iOS required the least amount of CPU activity, it rendered only half of the frames compared to the native approach – which used 7% more CPU than Ionic. However, the native approach consumed

26.01MB less RAM than Ionic. Comparing Android to iOS on side menu navigation, the Ionic framework consumed 114.79MB RAM on Android, while 67.14MB on iOS. These deviations in performance between the platforms result in difficulties when deciding on a particular framework and technology.

Perhaps the most interesting discrepancy is that of native iOS versus native Android, in terms of CPU consumption. At 7.94% CPU usage, Android consumed 59.07% less CPU than its iOS counterpart. However, Android still consumed 23.79MB more memory than the iOS app.

By empirically comparing the performance of the cross-platform frameworks, React Native on Android rendered the most frames without any reported jank while using only marginally more CPU than the native app and consuming only a few MB more RAM, and using the least amount of GPU RAM. In fact, RAM consumption was only marginally impacted by the execution of the side menu task, which is comparable to that of the native app's performance. React Native on iOS did, however, not render a number of frames comparable to the native app, although using the second most GPU and CPU.

Xamarin on iOS, similar to React Native, render few frames, but consumes more RAM and CPU than the other frameworks. On Android, Xamarin renders consistent frames without jank, at a lower CPU and RAM usage than Ionic. Still, React Native on Android performs better at this task than both Xamarin and Ionic. On iOS, Ionic was the most performant and hardware-consumption friendly framework, rendering more frames than the other cross-platform frameworks, relying less on CPU power than native, and consuming the least amount of RAM compared to the other frameworks.

5.4.4 Additional Observations

For the iOS apps, far fewer drastic increases in hardware penalties were recorded than for the Android apps. Most of the iOS implementations did not report any increases in consumption of any of the metrics. This is in stark contrast to Android, where the most significant increase in hardware consumption is that of the Ionic app during navigation transition. In one instance, RAM consumption *decreased* upon execution of the task, being Xamarin forms during the side menu test – starting at 103.55MB RAM consumption, decreasing to 87MB.

5.5 Discussion

This section considers each of the experiment's research questions in turn, discussing investigated performance metrics, profiling tools, and deviations in performance between the Android and iOS platforms.

5.5.1 Performance Metrics

$Sub - RQ_{1.2}$: Do the performance metrics fulfil their purpose in researching animated user interfaces in mobile apps?

While the frames-per-second metric can be of great value for measuring interface fluidity in games or apps with long-running animations, it did not provide the same value when measuring the fast-paced short-running animations. The other metrics, including `CPU`, `RAM` and `GPU RAM` usage, are common in performance-oriented research (*e.g.*, Dalmasso et al. (2013), Willocx et al. (2015, 2016)). Nevertheless, the extraction of `GPU` memory statistics from the iOS profiling tool was not found to be possible, rendering it less usable for a *cross*-platform comparison. Also, the `GPU` memory results generated per framework were found to only have had minor to no variations between the test, *e.g.*, Ionic which used 4.55MB regardless of the task or React Native outputting 3.45MB during the Lottie animation, and otherwise 1.08MB during the two other tasks. Due to the lack of significant differences between the benchmark tasks, the `GPU` memory metric was not found to be of particular importance for comparative studies on cross-platform development. `CPU` and `RAM` metrics were found to provide insightful information on potential performance penalties introduced by cross-platform frameworks.

The difficulty of reasoning about user interface performance based on the `FPS` metric led us to believe that user-centric studies on performance and perceived performance is perhaps more generalisable than profiling-based empirical assessments. As an example, as displayed in Table 5.4, the native Android app rendered only five frames during the 328ms long navigation transition. The reported jank could nevertheless be an indication of unoptimised frame rendering. Further research is needed on the correlation between `FPS`, reported jank and end-users' performance perception, as even the Xamarin Lottie animation reporting 37 janky frames could be challenging to distinguish from the other frameworks reporting of fewer unoptimised frames.

5.5.2 Evaluation of Performance Profiling Tools

$Sub - RQ_{1.3}$**:** How well do the official performance insight tools cater to the profiling of animation and transition performance in the cross-platform apps developed?

Both platforms provided their own tooling for performance profiling. In the case of Android profiling, three distinct tools had to be used – effectively increasing the overhead of the task. Nevertheless, all the tools employed provided highly granular and understandable data. In the case of the iOS apps, the data generated by the profiling tools were found to be less granular, and no `GPU RAM` measuring tool was identified. Both platform providers could have created performance profiling experiences requiring less overhead while providing more data. The Android profiling experience, while requiring several tools and the use of CLIs, outputted the most insightful and actionable data. Below follows a discussion in more detail on the experiences of gathering data using the profiling tools provided through Android Studio, `adb` and Xcode's Instruments.

5.5.3 Android Data Gathering

It was attempted to use third-party programs to measure and gather data on the frames-per-second performance of the artefacts' user interfaces. GameBench and FPS Meter were frequently mentioned in practitioners' forums; however, both were seemingly unable to report the performance of the implementations properly, both varying significantly from the results outputted by the official profiling tools, and often not producing any output at all. Also, as both GameBench and FPS Meter are supposed to run on the device in the background while measurements are recorded, they were deemed to potentially impact the performance results.

A combination of tools was required to retrieve data from the metrics included, as presented previously in Table 5.3. During the initial data-gathering phase, experimentation with a variety of third-party profiling tool alternatives was conducted. Due to the experiences from using these tools, the use of them was avoided for data gathering, as the app's performance was impacted by the tools. An example of such a tool is the built-in performance monitor for React Native (Ramos and Lemos, 2017), which was found to increase the RAM consumption of the app. In these tests, the initialisation and continuous profiling of the React Native monitor increased the RAM consumption from 80.23MB to approximately 106MB, with 110MB at peak consumption during initialisation. Such knowledge is invaluable for conducting proper performance testing, as incorrectly reported results could quickly be introduced into the data sets if control measurements are ignored.

Another obstacle encountered while performance testing the applications, explicitly for measuring CPU usage, was Android 8 (Oreo)'s then newly introduced security measures (Anonymous, 2017), disallowing third-party applications to gain access to CPU data. Subsequently, the official Android Studio profiler as well as `adb systrace` (see below) were the only viable options for data gathering from the Android-based apps. Thus, no verification of the results using third-party software was possible. For the sake of security, this newly implemented measure might have a positive effect, but for the sake of research and system performance insight, it severely limits the possibilities of results verification and ease of access to third-party system monitoring.

On the positive side, the Android Debugging Bridge (`adb`) was able to generate highly detailed performance reports using the following command (henceforth referred to as `adb systrace`):

```
$ python systrace.py --time=5 -o trace_<framework>.html
    ↪  sched gfx view -a app_package_name
```

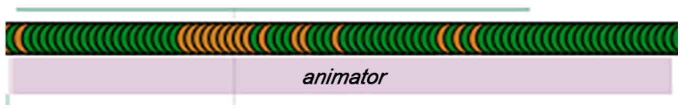


Figure 5.4: Example of how FPS is reported using `adb systrace` for Android apps.

By using the above `adb systrace` command, a five-second snapshot of the specified app's performance is recorded and outputted in HTML format, which can be opened in a browser and used to drill down into any single-frame performance issues. Figure 5.4 illustrates the part of the generated report displaying data on FPS. Every individual dot in Figure 5.4 represents a single frame, where green ones (darkest) have been optimally rendered, while orange or red ones (lighter colour) represent a frame that has been rendered in a suboptimal fashion. This could be due to the rendering time exceeding 16.6ms per frame, the time available to do any frame-specific calculations and rendering in order to keep a user interface at a stable 60 frames-per-second (Maust, 2015).

For animations and transitions lasting less than a second, being the menu opening animation and the page navigation transition, only the performance measurements reported during that given sequence was included alongside the measured millisecond count in superscript. For the Lottie animation lasting more than a second, only the first 1000ms of the animation's measurements are included.

In terms of retrieval of GPU memory usage statistics, the following command line tool was used:

```
$ adb shell dumpsys gfxinfo app_package_name
```

The tool was executed directly after each animation event was complete, as the tool outputted data limited to the last 128 rendered frames (note that in periods without changes to the interface, no frames were logged by the tool).

5.5.4 iOS Data Gathering

For the iOS apps, using the official performance profiling tool developed by Apple, Xcode Instruments, granted access to all the data and metrics needed, except for GPU memory usage. The Instruments tool implemented several *profiling instruments*, several of which reported on slightly similar metrics. Practical suggestions were also drawn from previous research by Willocx et al. (2015), which listed the types of instruments they included and mapped them to specific metrics. Some deviations from their suggestions were made, such as using the Time Profiler instruments rather than Activity Monitor for CPU measurement, as the reported output rendered results that were less complex to interpret, but delivered the same relevant data quality. For measuring device memory usage, the VM Tracker Instruments-exposed memory metric *Resident Size* was used, described to be the actual device memory consumed by the targeted application (Stack Overflow, 2013).

Lastly, to measure FPS, the Core Animation instrument was used. Alas, no instruments for reporting on janky frames were identified, rendering investigation of frame-specific problems more difficult than in Android profiling. Due to the granularity level and lack of drilling capabilities of the profiling data and instrument, *cross-second* `FPS` counts are included. This process is illustrated in Figure 5.5, where each blue bar represents a second of on-screen activity, such as displaying the Lottie animation. The empty spaces between the bars represent time passed without any frame activity, *i.e.,* no frames were (re-)drawn during those seconds. If an animation is executed well into an already-begun second (Instruments-wise), Instruments will start reporting performance during that second and finish sometime during the next second slot. Thus, this is the reason for *cross-second* `FPS` reporting, in Table 5.4 illustrated with a dash between the reported `FPS`, *e.g.,* "30-23" in the iOS native app `FPS` results during a running Lottie animation.

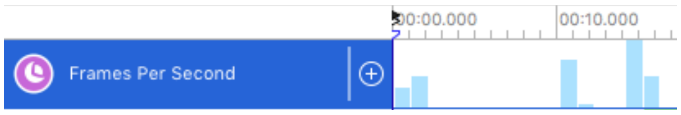


Figure 5.5: Example of measuring `FPS` in Xcode's Core Animations Instruments tool.

5.5.5 Platform Performance Deviations

$Sub-RQ_{1.4}$: Which of the platforms, iOS or Android, requires the least amount of device- and hardware resources in order to execute and run performant animations and transitions?

The performance, as reported by the profiling tools illustrate an interesting discrepancy, in that the reported `CPU` usage on Android is consistently lower than that reported on iOS across all three tasks and regardless of technical framework. In fact, `CPU` usage as reported during the navigation transition task is at 100% on iOS for all but the hybrid app, a finding correlating to previous research (Willocx et al., 2016) also stating that navigation transition is identified to be more performant in hybrid apps than in *e.g.,* native due to browser navigation optimisation. The only occurrence of both platforms performing similarly is in the case of the Ionic apps running the Lottie animation, where `CPU` is peaking at 29.93% on Android, and 30% on iOS.

Nevertheless, it is also essential to acknowledge the discrepancy between the native baseline implementations, especially that of `CPU` consumption during the side menu task – 7.93% on Android versus 67% on iOS. While one could speculate that this is the result of iOS's lack of a native side menu component (as mentioned, the implementation relied on a third-party component), one must also note that iOS, in general, has a higher `CPU` usage, but lower `RAM` consumption. A recent effort by Afjehei et al. (2019) has brought forth `IPerfDetector`, a tool for detecting performance anti-patterns related to memory consumption, updates to the user interface, and threading in iOS apps. Applying such a tool to the native baseline implementations could potentially further describe the reasoning behind the significant difference in `CPU` consumption between the two platforms.

To fully answer this section's research question, one metric is missing from the data set, being that of janky frames in the iOS implementations. Conducting a measurement-based comparison without the presence of this metric, renders an empirical-backed conclusion challenging, as working towards eliminating janky frames is deemed essential for the user experience (Reddi et al., 2018). Nonetheless, it is noted that iOS apps are more consistent in the consumption of `RAM`, while on Android, the apps' consumption tends to fluctuate.

While there are notable differences between the Android and iOS platforms, the same is also true for the cross-platform frameworks scrutinised. While a specific framework can be performant on Android, it is not necessarily the most optimal framework on iOS – and vice versa. An example of this is the Ionic framework, which on Android consumes the most `RAM` in two out of three tests (at the same time also being close to consuming the most also in the third), while on iOS it varies between the second- to third most `RAM`-efficient framework. From previous research, similar results in terms of the performance penalty introduced by the WebView component in hybrid-based apps are found (Latif et al., 2016). Overall, the nature of these platform variations renders cross-platform development additionally challenging, as one framework can be most performant on iOS, while on Android, a different framework can be beneficial to make use of instead. These discrepancies were also true for non-user-interface research, for instance, as reported by Willocx et al. (2016) in their performance analysis of cross-platform frameworks, displaying similar results. Also, specific product requirements may call for the need of a specific framework, *e.g.*, if Lottie animations are more critical than side menu navigation, then one framework may cater better to the requirements than the alternative technologies.

Nevertheless, on Android, the interpreted-based React Native framework was deemed the most performant cross-platform framework among those included in this experiment. The penalties introduced in terms of hardware consumption are less severe than those caused by Ionic and Xamarin, while also outputting a higher `FPS` than the other frameworks. Also on iOS does React Native produce better results than the alternative frameworks in terms of `RAM` consumption, but varies between the tasks on `CPU` usage and consistency of rendered `FPS`. In fact, the hybrid-based Ionic framework does overall consume between second- and third most `RAM`, but is the most `CPU` efficient in two of the tasks – also more so than native, and it renders an `FPS` that, in two of the tasks, are of a higher count than the other frameworks. The only exception is during the Lottie animation, in which Ionic rendered 11-13 fewer frames than React Native and Xamarin respectively.

5.6 Conclusion

The frameworks included for assessment in the experiment described in this chapter are React Native, Ionic and Xamarin Forms. In addition to the cross-platform apps, one native app for each platform, respectively iOS and

Android, are implemented for gathering baseline performance results for comparison purposes. The amount of available performance measurement tools, both official and third-party, lead to some confusion in terms of *picking the right ones*, mainly as some could – and inherently would – introduce additional overhead in terms of device resource consumption. A key takeaway from this experiment is that numerous trade-offs must be accounted for when choosing cross-platform over native development. For instance, one cross-platform framework may deliver good performance results on Android, while delivering subpar results on iOS – and *vice versa*. React Native was found to deliver the most performant user interface animations among the set of cross-platform frameworks, although Ionic on iOS was more `CPU` efficient than both frameworks and the native baseline implementation. Thus having some knowledge of the end-users' hardware may help decide on an approach or framework for app development. The results indicate that there is no *silver bullet* among the technologies benchmarked. Having now scrutinised the performance of both the bridge implementations and user interface fluidity of a set of cross-platform frameworks towards answering the first thesis research question (RQ_1), the following chapter describes the experiment towards answering the second thesis research question (RQ_2) and objective, investigating the presence of these frameworks in apps published to the Google Play Store.

Chapter 6

Presence of Mobile Development Approaches

This experiment investigates the presence of cross-platform mobile development frameworks in published Android apps. The chapter firstly summarises the problem statement, research questions and hypothesis, and the experimental setup. Subsequently, it describes the development of the app harvesting and framework identification algorithm, followed by the evaluation and discussion on framework usage across marketplace categories, changes in framework usage over time, app file sizes, and a discussion on trends from an industry perspective.

6.1 Awareness of Problem: Cross-Platform Framework Usage in Published Apps

Revisiting the Objective. In the previous two experiments, the performance of cross-platform frameworks and their generated apps has been scrutinised on a variety of metrics including, but not limited to `CPU%`, `RAM` consumption, and time-to-completion `(TTC)`. In this experiment, the usage, or presence, of these frameworks in published apps available on the Google Play Store is investigated, using a sample dataset of $n = 661\,705$ apps, along with metadata for each app. The distribution of framework usage across Play Store categories is assessed, along with trends over the last decade, and app file sizes. The objective is to uncover the actual use of cross-platform frameworks, and discuss how this differs from trends encountered in industry outlets. This investigation is conducted towards answering the second thesis research question (RQ_2) regarding the presence of cross-platform frameworks in published apps, as described in Section 1.4.

Background and Literature Synopsis. A significant body of knowledge exists on mobile app store analysis considering the field is only a decade old at the time of writing. There is a wide range of topics covered in the literature, as surveyed by Martin et al. (2017); however, only a handful of identified papers cover cross-platform development. Several of these focus on mining

Communication of Research: This is an extended version of the publication currently in-review. The content and format has been modified to fit the thesis narrative. Replication package is available in Appendix D.

and analysing user reviews, for instance Malavolta et al. (2015b) studying cross-platform framework usage and user perception of apps generated using a set of eight frameworks. Their findings indicate that hybrid apps account for 3.73% of their investigated dataset ($n = 11\,917$) and that the majority of these are found in the Play Store categories Finance, Medical, and Travel and Local. According to the authors, native apps on average receive better ratings than hybrid apps, although, on perceived performance, the results vary significantly between the various app categories. Mercado et al. (2016) describes a similar study, wherein the authors investigate how the choice of development approach impacts user reviews for $n = 50$ apps. For instance, their results indicate that for iOS users, interpreted apps are perceived more performant and usable than native apps and that Android users had a more pessimistic view on hybrid apps than on native apps. Although there are a handful of studies looking into the presence of cross-platform apps, the experiment to follow includes a broader array of frameworks in the identification algorithm, and a large sample dataset ($n = 661\,705$). There is also additional emphasis specifically on challenges related to `.apk` size and the impact this has on potential app downloads and user acquisition, as highlighted by Tolomei (2017) of the Google Play Store team.

6.2 Suggestion: Analyse Harvested Apps & Metadata

Harvesting Android installation files and associated metadata directly from the Google Play Store has previously been reported as a task of considerable complexity due to implemented security measures. Instead of circumventing these measures, as has previously been described by Li et al. (2017), readily available open and paid services providing access to Android installation files and metadata have been used.

6.2.1 Research Questions and Hypothesis

As introduced in Section 1.4, the overarching thesis research question related to the presence of cross-platform frameworks in deployed apps reads as follows:

RQ_2: How common is the presence of cross-platform development frameworks compared to the native development approach in published mobile apps?

Derived from this research question, two sub-research questions and one hypothesis have been formed:

$Sub-RQ_{2.1}$: What is the distribution of cross-platform development frameworks on the Google Play Store across app categories?

$Sub-RQ_{2.2}$: How has the use of cross-platform frameworks in deployed apps changed over the last 12 years?

$Hypothesis_1$: Apps developed using the native approach should generate `.apk` files of smaller file size than apps developed using cross-platform development frameworks due to not relying on bundled interpreters, virtual machines or WebView containers.

6.2.2 Harvesting Android installation files

Through the AndroZoo project, the University of Luxembourg is providing researchers with a massive binary-based dataset for Android app analysis. AndroZoo is a repository of Android Package (`.apk`) files and a service through which researchers can download such files if granted access (in the form of an API key). What the AndroZoo service provides is direct access to app installation files that either are or have been published to the Google Play Store or similar app marketplaces. Thus, access to published, real-world Android apps has been drastically simplified compared to scraping the Google Play Store directly.

6.2.3 Harvesting app metadata

A frequent concern when harvesting data from third-party data providers who do not expose their data through APIs, is the risk of being IP banned after an unknown number of Web page requests. Such data harvesting is typically referred to as Web Scraping and relies on extracting data from Websites based on the Website's HTML structure, tags and IDs. The first approach to gathering data from the Google Play Store attempted various methods of Web Scraping. However, the risk of getting IP banned for making repetitive requests to the Google Play Store Website was deemed unnecessary so much so that the data gathering instead went through a paid third-party API[1], hosted and made available through the API service RapidAPI. The API should be considered a middleware data provider, placed in between the Google Play Store and us, mitigating the risk of IP bans.

6.3 Development of Harvesting Mechanism & Identification Algorithm

This experiment is of an exploratory and deductive nature, with analyses based on quantitative data. The procedure for conducting the analysis is twofold: firstly, details on the process of gathering the data used for analy-

[1]https://rapidapi.com/maxcanna/api/google-play-store?endpoint=5414be4de4b031830c0100b8

ses are presented, namely binary `.apk` files from AndroZoo and associated metadata from the Google Play Store. Following that is a description of the framework identification algorithm used to search for pre-defined rules and patterns in the `.apk` files, thus identifying which (if any) cross-platform framework(s) have been used in the development of the apps.

6.3.1 Data Gathering

To gather the Android installation files (`.apk`) needed to answer the research questions, access to the AndroZoo service was granted, a large dataset made available by researchers at the University of Luxembourg (as described in detail by Allix et al. (2016)). Data gathering through AndroZoo instead of the Google Play Store was a deliberate choice to increase the reproducibility and availability of the dataset and results, as Google Play Store mining is unnecessarily complicated and time-consuming (Allix et al., 2016) when more accessible services exist, especially so when these were developed specifically for the benefit of researchers. Due to the structure and services provided by AndroZoo, researchers interested in replicating or continuing the work presented in the current experiment may study the data gathering process and open-sourced dataset (see Section 1.5). The AndroZoo dataset was filtered on apps published in the Google Play Store, as the dataset also contains apps from various other sources including, but not limited to PlayDrone, AppChina and F-Droid. The `.apk` files were downloaded onto a LaCie 12big 120TB hard drive using the `GNU Parallel` package (Tange, 2018), then further analysed using on-premise hardware. The dataset consisting of $n = 661\,705$ compiled (`.apk`) apps require 9.2TB of hard drive space. After downloading the `.apk` files, associated metadata was gathered from the Google Play Store through the RapidAPI service, and the `.apk` installation files were piped through the cross-platform framework identification algorithm (described in Section 6.3.2). The data gathering process is illustrated in Figure 6.1.

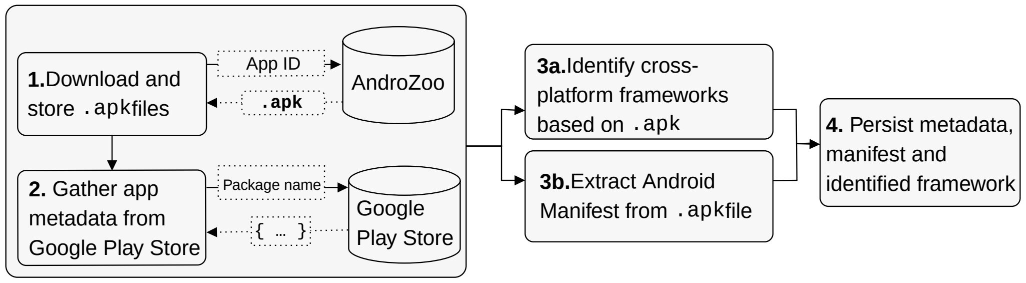

1.Download and store .apkfiles
App ID
.apk
AndroZoo
2. Gather app metadata from Google Play Store
Package name
{ … }
Google Play Store
3a.Identify cross-platform frameworks based on .apk
3b.Extract Android Manifest from .apkfile
4. Persist metadata, manifest and identified framework

Figure 6.1: Data harvesting process from Google Play Store, AndroZoo and Android Manifests.

6.3.2 Framework Identification Algorithm

In order to identify the use of a cross-platform framework based on an `.apk` file, a pattern matching algorithm was developed, looking for specific string-based values, files, folders, *etc.* As the technical landscape of cross-platform development is vast, with more than 60 individual frameworks listed as part of the literature review in Chapter 2 (Table 2.1), it was necessary to decide on a specific set of frameworks for which condition-based rules could be created. Table 6.1 contains the 13 frameworks for which rules were developed to analyse the `.apk` files. All four major development approaches are accounted for in the list. In terms of sample size, Martin et al. (2017) found that studies on app store analyses wherein analysed apps have a median sample size of 1 679 and a mean of 44 807 apps. The sample size of $n = 661\,705$ `.apk` files puts the experiment in the upper percentile in terms of the number of assessed apps (Martin et al., 2017, Fig. 4).

The framework identification algorithm is inspired by Ali and Mesbah (2016)'s study on characterising hybrid apps based on an algorithm accounting for three frameworks, which in this current experiment is extended to 13 frameworks across three development approaches. The algorithm traverses a directory containing N `.apk` files to identify the various frameworks, searching for information for- and in meta tags, manifest files, text- and binary files. The following two rulesets were developed to identify the various frameworks. A mapping between rulesets and frameworks is found in Table 6.1.

1. File/folder search in the `.apk` 's extracted `/assets/` folder.
2. String search in extracted `AndroidManifest.xml`.

The algorithm was validated by downloading and testing `.apk` files associated with apps from each framework's showcase page. The development of the algorithm was highly iterative. The version of the algorithm used to extract the dataset presented in this experiment managed to identify all the tested showcase apps correctly.

Table 6.1: List of the 13 technologies included in the identification algorithm. Table grouped by approach.

Technology	**Approach**	**Programming language**	**Release year**	**Ruleset**
Native*	Native	Numerous	2007/08	-
Adobe Air	Interpreted	Numerous	2008	(1,2)
NativeScript	Interpreted	JavaScript	2014	(2)
Qt (Mobile)	Interpreted	C++/QML	2013	(2)
Fuse	Interpreted	JavaScript/C#	2012	(2)
Titanium	Interpreted	JavaScript	2009	(2)
React Native	Interpreted	JavaScript	2015	(1,2)
Weex	Interpreted	JavaScript	2016	(2)
Codename One	Cross-compiled	Java/Kotlin	2012	(2)
Flutter	Cross-compiled	Dart	2017	(2)
Xamarin	Cross-compiled	C#	2011	(2)
Capacitor	Hybrid	JavaScript	2017	(1,2)
Cordova *(+PhoneGap)*	Hybrid	JavaScript	2009	(1,2)
Ionic *(Cordova-based)*	Hybrid	JavaScript	2013	(1,2)

As a note on the native category: in this study, to categorise apps as belonging to the native development approach, the requirement was that the `.apk` did not match any of the rules specified for the cross-platform frameworks included. Hereinafter, these are described as Unidentified or native, as the algorithm could not identify the use of a framework, nor necessarily the lack thereof. This is a limitation to the experiment.

While Robles (2010) previously reported that studies within Mining Software Repositories (MSR) had been found to incorporate a low level of replicability due to the lack of available datasets, the identification algorithm and dataset with extracted data (`.csv`) from this current experiment have been made available through GitHub[2], along with necessary instructions. The

[2]GitHub repository for scripts and dataset: https://github.com/andreasbhansen/phd-thesis-contributions/

binary `.apk` files, however, were not uploaded as these can be retrieved through the AndroZoo service[3] based on their unique identification string found in the `.csv` dataset.

6.4 Evaluation and Discussion

This section details findings related to framework usage and distribution, app file size, and a discussion on technology trends versus adoption. It begins the discussion and presentation of findings by analysing the dataset in light of the research questions and hypothesis. Subsequently follows a more holistic discussion on the newly derived knowledge, looking at differences between actual technology adoption and perceived interest from practitioners based on search trends.

6.4.1 Framework Distribution for Apps on Google Play

The discussion on findings starts by providing an overview of the adoption of cross-platform frameworks in apps published to the Google Play Store. The current section and the section to follow both discuss results in light of the experiment's first research question (RQ_1):

[3]AndroZoo repository Website: https://androzoo.uni.lu/

> $Sub - RQ_{2.1}$: What is the distribution of cross-platform development frameworks on the Google Play Store across app categories?

Although industry outlets have set their focus on more recent frameworks including React Native and Flutter (*e.g.*, Greif et al. (2018), Skuza et al. (2019)), we can look to Figure 6.2 to find indications of numerous frameworks having considerably larger market shares of published apps on the Google Play Store as per the sample dataset. In terms of development approaches, native apps account for the majority of the analysed apps, according to Figure 6.4. In numbers, the analysed dataset consists of $n = 562\,401$ native apps ($\approx$ 85%) and $n = 99\,304$ ($\approx$ 15%) cross-platform apps. Each of the top three cross-platform frameworks listed in Figure 6.2 belongs to different approaches, respectively hybrid (Cordova), cross-compiled (Xamarin) and interpreted (Adobe AIR). Grouping the count of apps on development approach (see Table 6.1) rather than on framework, the hybrid approach accounts for $n = 48\,371$ apps, interpreted for $n = 27\,468$ apps, and cross-compiled for $n = 23\,484$ apps. The hybrid approach is thus the largest of the development approaches in terms of published apps on the Google Play Store.

Specifically, the number of hybrid apps is in rather stark contrast to the Gartner (2013) prediction that by 2016, a majority of mobile apps will be based on cross-platform technologies. Based on a thorough search yielding no results, Gartner does not seem to have provided similar predictions for 2020

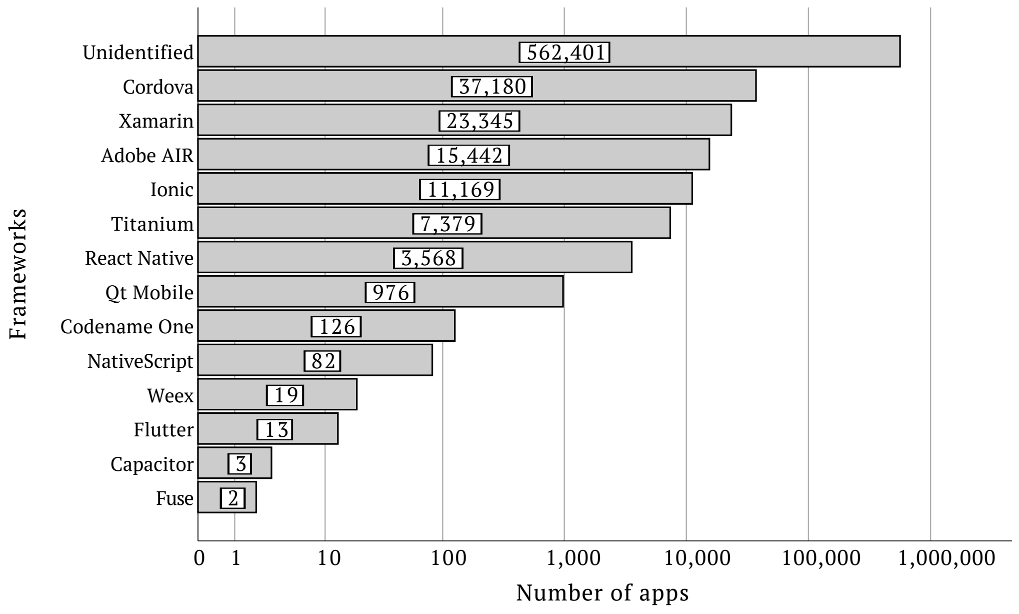


Figure 6.2: Log-scaled distribution of cross-platform frameworks and native development.

or later, neither have such predictions been identified by similar companies or organisations. Nevertheless, looking to related studies, these numbers are somewhat higher than what has previously been reported by Viennot et al. (2014) (9.20% of non-popular apps specifically). However, the number of hybrid apps identified is more extensive in this experiment (8.67%) than in Malavolta et al. (2015b), where the hybrid approach accounted for 3.73% of the apps analysed. This discrepancy could be the result of framework identification technique or sample size, as Malavolta et al. (2015b) analysed $n = 11\,917$ apps, while for this current experiment $n = 661\,705$ apps were analysed. It may also be an indication of growing industry interest in the hybrid development approach. However, statistics on the number of downloads of the Cordova tool is approaching an all-time low according to the Website npm-stat (Vorbach, 2019) covering download statistics for JavaScript third-party packages.

6.4.2 Framework Distribution Across Google Play Categories

Investigating the distribution of cross-platform framework usage across app categories on Google Play Store can indicate how practitioners and industry have made use of these technologies for targeting various types of apps, for instance, the particular presence of cross-platform frameworks in one category, and the absence of such in another. It could as such also give an indication of which categories have been found particularly suitable or unsuitable for cross-platform technologies, for instance, due to specific requirements such as complex graphics processing or reliance on low-level C or C++ code, situations in which cross-platform frameworks may not be beneficial.

The aggregated results are presented in Figure 6.3 and Table 6.2. The former is a log-scaled visualisation of the use of development technologies across the top four Play Store categories, the latter a contingency table displaying a tabularised distribution of apps across development technologies grouped by Play Store categories. Due to the amount of Play Store categories, only the top four Google Play Store categories based on the number of apps regardless of development approach have been visualised, which according to

Table 6.2 (column "$\sum$ w/ native") are Games[4] (n = 136 316), Education (n = 57 662) and Lifestyle (n = 46 675). In terms of cross-platform framework usage, looking to Table 6.2 (column "$\sum$ w/o native"), Games (n = 13 248), Business (n = 12 426) and Education (n = 12 091) are the top three Play Store categories with the most identified apps. Thus, the Lifestyle category was replaced with Education when excluding the native development approach, and instead focusing specifically on cross-platform apps, hence the top four categories in Figure 6.3.

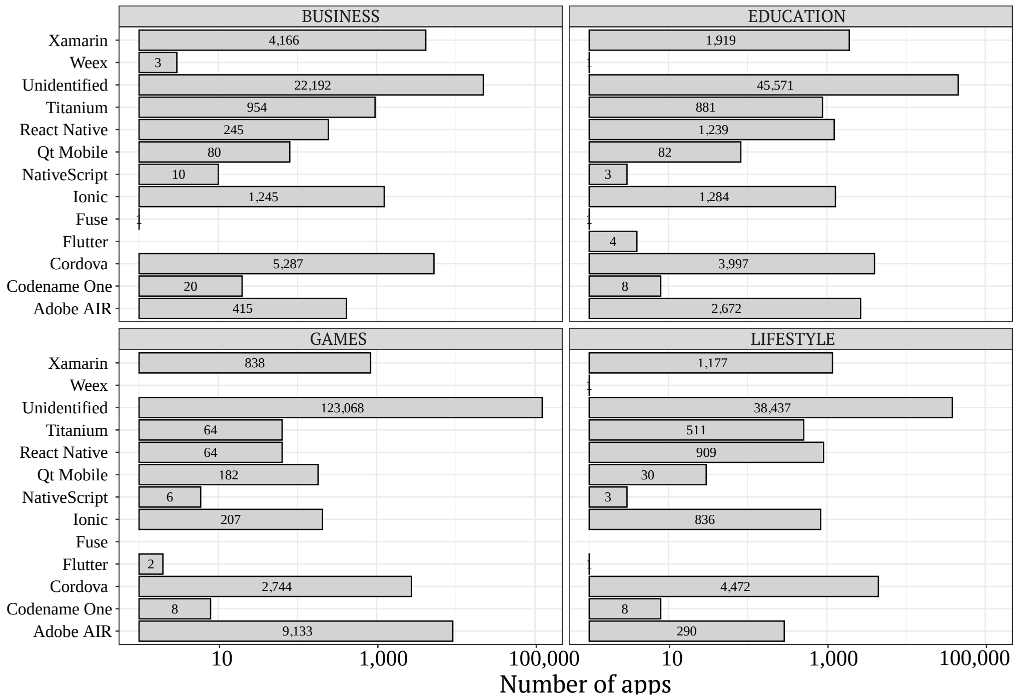


Figure 6.3: Log-scaled distribution of native and cross-platform frameworks across top four Play Store categories. Ordered by framework name.

Of the n = 13 248 games identified, Adobe AIR (n = 9 133) and Cordova

[4]The Games category is an aggregation of all the sub-categories on Play Store within gaming, for instance, adventure, puzzle and strategy.

(n = 2 744) together account for 89.65% of the cross-platform apps in the category. Although Adobe AIR markets itself as a cross-platform framework for both app and game development, 59.19% of the identified Adobe AIR apps are in the games category, a higher percentage than any other framework. This presence could indicate that while Adobe AIR can cater to both gaming and non-gaming apps, it is a technology preferred by game developers more so than by non-game app developers.

Regarding the Business category, B2B apps might prioritise differently in terms of user experience and functionality than apps developed for regular consumers (Anglin and Telerik, 2014). Given that the hybrid approach (thus Cordova) is frequently perceived as being nearly incapable of achieving predictable native-like user experience across platforms and older devices (*e.g.*, Ahti et al. (2016), Dhillon and Mahmoud (2015), Nunkesser (2018)), apps in the Business category might skew towards functionality over user experience.

Table 6.2: Distribution of apps per framework grouped by Play Store category. Table ordered by category.

Category	Adobe Air	Capacitor	Codename One	Cordova	Flutter	Fuse	Ionic	Native-Script	Qt Mobile	React Native	Titanium	Weex	Xamarin	Native*	Σ w/ native	Σ w/o native
ART AND DESIGN	44	0	2	61	0	0	20	0	7	5	7	0	51	3843	4040	197
AUTO AND VEHICLES	16	0	1	144	0	0	48	0	4	10	46	0	286	2483	3038	555
BEAUTY	12	0	0	86	0	0	21	0	2	15	17	0	110	1573	1836	263
BOOKS AND REFERENCE	255	0	6	1667	0	0	840	11	14	35	183	1	641	35888	39541	3653
BUSINESS	415	0	20	5287	0	1	1245	10	80	245	954	3	4166	22192	34618	12426
COMICS	36	0	0	22	0	0	17	0	0	0	1	0	21	1725	1822	97
COMMUNICATION	86	0	4	618	0	0	253	4	23	34	85	0	565	9590	11262	1672
DATING	7	0	0	118	0	0	7	0	1	2	4	0	22	412	573	161
EDUCATION	2672	0	8	3997	4	1	1284	3	82	1239	881	1	1919	45571	57662	12091
ENTERTAINMENT	697	0	5	2052	0	0	525	4	33	86	203	2	1019	33029	37655	4626
EVENTS	16	0	0	112	0	0	65	3	2	29	46	0	271	671	1215	544
FINANCE	93	0	3	1850	0	0	443	2	7	69	755	1	1237	13125	17585	4460
FOOD AND DRINK	90	0	1	766	0	0	338	1	5	36	307	0	559	5632	7735	2103
GAMES	9133	0	8	2744	2	0	207	6	182	64	64	0	838	123068	136316	13248
HEALTH AND FITNESS	182	3	7	1397	1	0	629	1	32	59	1013	0	958	14024	18306	4282
HOUSE AND HOME	16	0	0	141	0	0	49	1	8	14	28	0	141	1963	2361	398
LIBRARIES AND DEMO	15	0	0	61	0	0	29	1	7	5	19	0	60	1439	1636	197
LIFESTYLE	290	0	8	4472	1	0	836	3	30	909	511	1	1177	38437	46675	8238
MAPS AND NAVIGATION	20	0	1	649	0	0	183	1	20	25	106	1	433	6502	7941	1439
MEDICAL	109	0	4	1086	0	0	299	4	22	44	395	0	761	6685	9409	2724
MUSIC AND AUDIO	167	0	3	831	0	0	174	1	42	18	147	0	409	35705	37497	1792
NEWS AND MAGAZINES	81	0	2	791	0	0	368	3	19	74	174	1	534	15872	17919	2047
PARENTING	28	0	0	33	0	0	13	0	2	3	1	0	44	489	613	124
PERSONALISATION	10	0	0	40	0	0	23	0	0	1	8	0	48	31290	31420	130
PHOTOGRAPHY	53	0	0	87	0	0	26	0	11	12	13	0	70	10890	11162	272
PRODUCTIVITY	181	0	13	1352	1	0	568	6	112	80	224	2	1737	15250	19526	4276
SHOPPING	29	0	5	998	0	0	374	5	4	77	168	3	952	8447	11062	2615
SOCIAL	54	0	5	658	0	0	325	1	7	77	140	0	477	7173	8917	1744
SPORTS	138	0	0	1302	0	0	351	4	16	66	190	0	949	10528	13544	3016
TOOLS	275	0	12	1260	0	0	582	4	149	78	203	1	1657	34412	38633	4221
TRAVEL AND LOCAL	142	0	6	2243	3	0	913	2	30	144	435	2	1119	17582	22621	5039
VIDEO PLAYERS	59	0	1	125	1	0	33	0	18	8	29	0	45	4209	4528	319
WEATHER	9	0	1	107	0	0	74	1	5	5	18	0	62	2442	2724	282
Σ	15430	3	126	37157	13	2	11162	82	976	3568	7375	19	23338	562141	661392	99251

Looking at previous research on Google Play Store category distribution of cross-platform apps, Malavolta et al. (2015b) had the Business category listed as the seventh most cross-platform populated Play Store category. As their study is from 2015, an explanation of this discrepancy could be a shift towards more massive investments in business digitisation, leading to an increase in B2B and line of business apps, and thus in the use of Cordova. However, the results presented by Ali and Mesbah (2016) align much closer to the distribution presented in Table 6.2, with the Business category as the predominant outlet for hybrid apps. As the results indicate, the hybrid-based Cordova framework is the most used cross-platform framework within the Business category, accounting for 42.55% of cross-platform apps in the category. Do note that the far-right sum ($\sum$) columns in Table 6.2 differs from the number of identified cross-platform apps due to parsing and extraction issues with a small number of apps' Play Store category.

By inspecting the Google Play Store's page on top apps in the Education category, we find Massive Open Online Courses (MOOC) from MIT, Udemy and Coursera, alongside apps for Learning Management Systems (LMS), interactive language courses such as Duolingo, and note-taking apps. The strong presence of cross-platform apps in this category could indicate that cross-platform frameworks can cater to varying degrees of complexity and requirements, as the aforementioned apps range from predominantly list-view-based apps (note-taking) to more performance-demanding products focusing on video rendering and animated content.

Other specific categories also stand out, but rather due to the relatively considerable absence of cross-platform apps. Personalisation is one extreme case of such, where cross-platform frameworks account for 0.41% of the category's identified apps. Apps in the Personalisation category include, for instance, custom Android launchers, custom keyboards, icon and wallpaper packs, apps focusing on home screen widgets, and ringtone providers. While apps focusing on content delivery, including icons, ringtones and wallpapers, may not require deep integration into native device and platform functionality, creating custom Android launchers and keyboards do require such integrations. Those types of apps may require a higher degree of platform-specific native code than, for instance, a Business category app, as the former's predominant focus is on modifying the underlying Android system front-end. Another category lacking the presence of cross-platform apps is Photography, wherein such apps account for 2.74%. It could be argued that possible reasons are similar to those of the Personalisation category; a significant need for platform-specific native code for communication with the underlying platform and device features, such as to perform face recognition and tracking with superimposed filters in real-time (*e.g.*, adding dog ears to a video feed and changing facial features).

6.4.3 Framework Distribution Over Time

A challenge with analysing framework distribution and usage over time stems from a side-effect of obfuscation during `.apk` compilation (Kambourakis et al., 2017). Specific measures and obfuscation techniques will render an app's compilation time (Dalvik executable files' creation dates, `DEX_DATE`) unusable as the date is set to the 1970s, 1980s or an arbitrary year; 2001, 2046 or 2059 which were all observed in the dataset. Prior to downloading `.apk` files from the AndroZoo repository, a filter was applied to only download files with a seemingly usable (*i.e.*, non-obfuscated) date between 2008 and 2019. In Figure 6.4, a log-scaled distribution of framework usage between year 2008 and 2019 is depicted. Native apps were excluded from the figure to focus exclusively on cross-platform presence. This section is concerned with addressing the experiment's second research question (RQ_2).

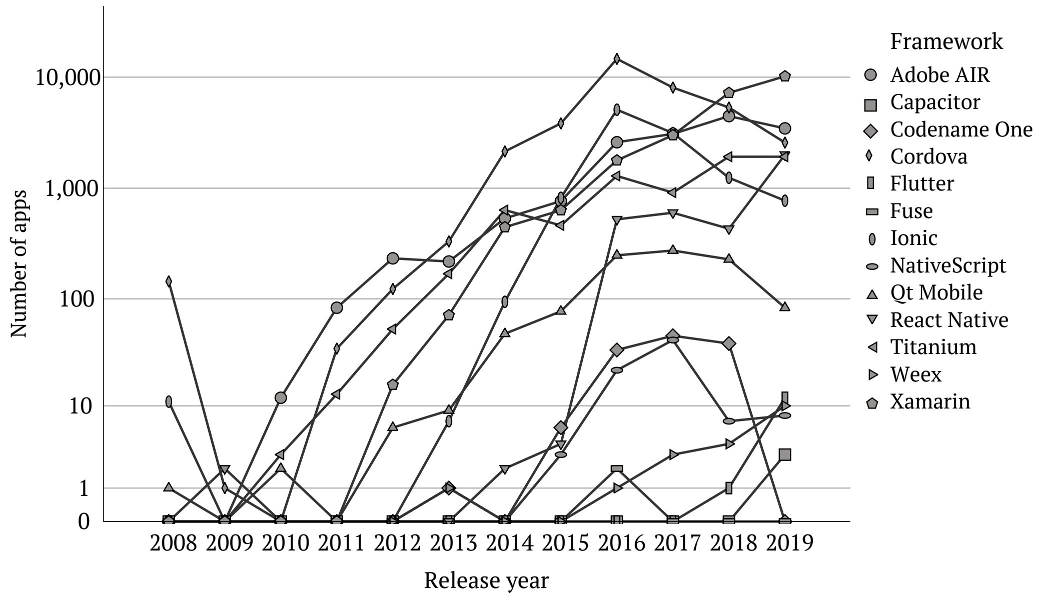


Figure 6.4: Log-scaled framework distribution over time from 2008 to 2019.

$Sub-RQ_{2.2}$: How has the use of cross-platform frameworks in deployed apps changed over the last 12 years?

The year 2016 was particularly interesting, considering the findings in Figure 6.4, as all the assessed cross-platform frameworks increased in adoption that year. One possible explanation for this is the industry-generated hype surrounding the first release of React Native in 2015 (Alpert, 2015), an implementation which turned more stable in 2016. The possible newfound interest for cross-platform development could have led industry practitioners to also look for alternatives to React Native, resulting in higher adoption and usage across all frameworks.

Up until 2018, the dataset indicates that Cordova was for five consecutive years the dominating cross-platform development framework. Cordova peaked in terms of deployed apps in 2016 ($n = 16\,225$). However, in 2018 Xamarin surpassed Cordova for the first time since 2011. Based on data from the Stack Overflow developer surveys from 2017 and 2018 (Stack Overflow, 2017, 2018), Xamarin is more appreciated than Cordova for both years, with a slight increase in developer appreciation for both frameworks. The Xamarin growth in 2018, according to the data, could possibly be related to the Microsoft acquisition of Xamarin in 2016. Xamarin grew simultaneous to the decrease in Cordova adoption. The decreasing use of Cordova is also reflected in the decline of desktop installs of the Cordova tool, as illustrated by the download statistics provided by the npm-stat Website (Vorbach, 2019).

Based on discussions from industry outlets, Facebook's React Native has gained quite some traction among practitioners (Skuza et al., 2019). As shown in Figure 6.4 however, an earlier implementation of the interpreted approach does still see more usage in published apps – namely the Titanium framework from Appcelerator. While React Native had a spike after its release in 2015, the Titanium framework was still found to be more popular for published apps up until 2019 where React Native saw a considerable increase in adoption, from $n = 431$ in 2018 to $n = 2\,004$ in 2019, whereas Titanium in 2019 was at $n = 1\,922$.

Looking to Figure 6.4, Cordova was used in 144 of the sample size apps in 2008, an outlier compared to the remaining frameworks. However, the increase in the use of Cordova has continued since then, with the exception of 2009 and 2010. Interestingly to observe, is that for several of the frameworks with high usage volumes, such as Xamarin, Adobe AIR and Cordova, the trend for the last two years, 2018 / 2019, shows a decline. It will be interesting to see whether this trend continues, and could be an indication of a swing in the pendulum between native apps and cross-platform apps, where the former has an indication to be on the rise at the moment.

6.4.4 Impact on `APK` File Size

The importance of keeping the compiled binary `.apk` size as small as possible is stressed by the Google Play Store team's research. Their findings indicate a 1% decrease in app downloads per 6MB increase in `.apk` size. Additionally, apps beyond 100MB in size are more prone to see a cancellation of the Play Store download by an increase of 30% compared to apps less than 100MB in size (Tolomei, 2017). Thus, investigating the potential impact which cross-platform frameworks impose on the compiled `.apk` size is essential. Prior to the investigation, a Levene's test was conducted to check for homogeneity of variance in the dataset ($\alpha = .05$) (Field and Hole, 2003). This is important in of terms uncovering possible violations of assumptions for follow-up tests, for instance in the case of the ANOVA which assumes normal distribution. The test reported of statistically significant variance, thus violating the assumption of homogeneity of variance in ANOVA, at $F(13, 661691) = 318.927, p < .0005$. Due to the number of outliers (see Figure 6.5), high standard deviation and sample variance of identified apps between the categories of development approaches (see Table 6.3), ranging from two (2) apps in the case of Fuse up to 37 180 apps in the case of Cordova, and further 562 401 unidentified/native apps, it was deemed infeasible to conduct hypothesis testing using an analysis of variance. Instead, the focus is on reporting and interpreting results based on descriptive statistics to discuss Hypothesis 1 (H_1), and highlight mean ($\bar{x}$) `.apk` size alongside standard deviation (σ), maximum and minimum values in Table 6.3 accompanied by a boxplot of `.apk` size per cross-platform framework.

The mean `.apk` size generated by the various cross-platform frameworks can have immediate implications for decision making and requirements engineering. Developing apps for storage-constrained devices requires caution and thought, and the appropriateness of an app's file size depends on factors such as primary market (*e.g.*, Western countries versus developing regions) and significance for end-user (*e.g.*, daily use versus one-time use). An app targeting a developing country may need to put additional emphasis on file size to avoid unnecessary cost and download time-related to the acquirement of the app (see for example Google's case study on the Ola ridesharing app (Google, 2017)). Developing an app meant for daily use may grant the developer fewer constraints in terms of file size, while an app used only once, for instance, a public transportation app needed during a holiday, can perhaps face additional scrutiny by the end-user for being unnecessarily large. Findings and results derived from the current experiment can thus potentially be of great value to practitioners during processes of technical decision making. This section is concerned with addressing the experiment's hypothesis (H_1).

Hypothesis$_1$: Apps developed using the native approach should generate `.apk` files of smaller file size than apps developed using cross-platform development frameworks due to not relying on bundled interpreters, virtual machines or WebView containers.

The results presented in Table 6.3 and Figure 6.5 indicate that the vast majority of the frameworks generate apps with a higher mean `.apk` size than the native development approach. Nevertheless, three frameworks were found to generate lower means. On the grounds of the results presented in Table 6.3, Hypothesis 1 (H_1) can be rejected; native apps are not inherently smaller in file size than cross-platform apps. Nevertheless, the only three exceptions to the hypothesis were Fuse, Cordova and Codename One. The remaining frameworks were shown to produce apps of larger mean file size than the native approach. It is important to note that both Fuse and Codename One have a significantly lower number of identified apps used for comparison, while Cordova has a more comparable size.

Table 6.3: Overview of mean `.apk` size per framework. Table ordered by mean `.apk` size.

Technology	N	**Mean (kB)**	**SD (σ)**	**Max (kB)**	**Min (kB)**
Xamarin	23 345	34 142,45	20 579,42	198 528,01	2 510,52
Weex	19	33 439,57	26 852,28	97 842,74	930,33
Flutter	13	31 657,93	26 088,96	94 889,04	7 200,16
Titanium	7 379	28 380,40	24 104,87	102 230,71	804,60
React Native	3 568	27 239,81	14 494,94	112 823,37	3 734,31
Adobe AIR	15 442	23 556,74	17 359,39	107 526,09	32,64
Capacitor	3	21 841,82	75,41	21 885,39	21 754,74
Qt Mobile	976	20 706,93	14 849,85	104 086,80	73,58
NativeScript	82	19 314,41	12 166,07	85 992,08	6 741,01
Ionic	11 169	15 273,53	14 355,10	104 716,90	315,74
Unidentified	562 401	13 776,04	17 490,73	176 575,43	3,38
Fuse	2	12 675,19	2 584,90	14 502,99	10 847,39
Cordova	37 180	12 527,55	13 151,81	106 686,63	95,72
Codename One	126	8 011,64	7 934,04	38 516,81	1 895,09
Total	*661 705*	*14 924,16*	*17 953,00*	*198 528,01*	*3,38*

On the opposite side of Codename One is Xamarin with the highest mean `kB` size. Looking at the overview of the number of apps per framework per Play Store category in Table 6.2, Xamarin is mainly present in the Business category ($n = 4\,166$). While it could be that apps in this category are in general larger in size, also the presence of Cordova in the Business category is substantially larger than in any other category ($n = 5\,287$), yet has a mean `kB` size close to three times smaller than the average Xamarin app. These find-

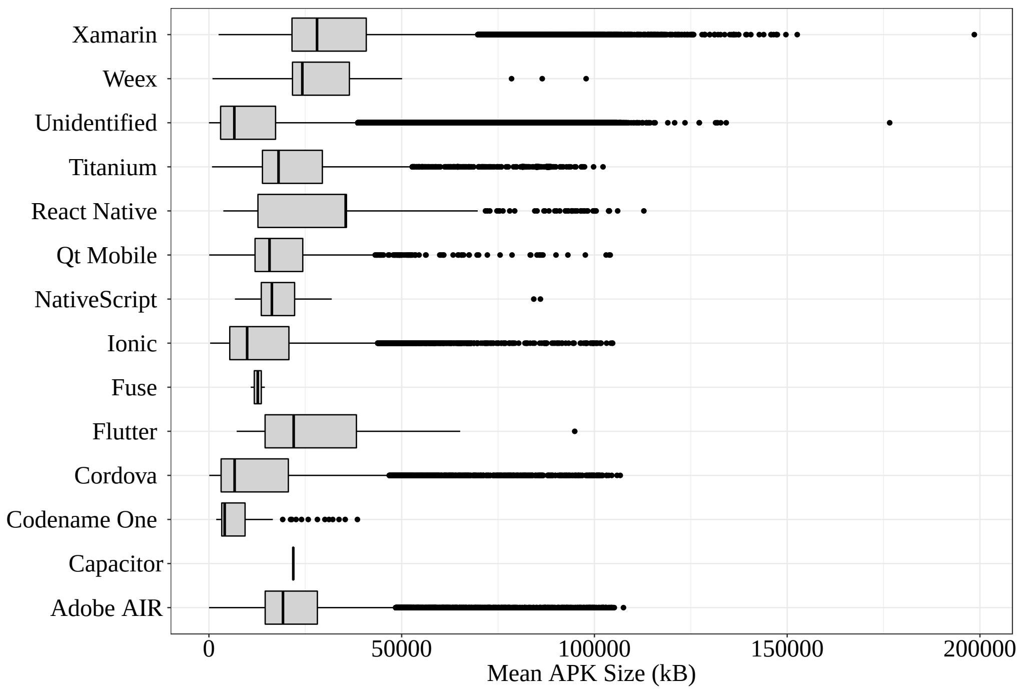


Figure 6.5: Boxplot of `.apk` size per framework.

ings contradict those presented by Corbalán et al. (2019), where three types of apps were developed with- and compared across five cross-platform frameworks involved. The authors report Xamarin to output average-sized apps when compared to the alternative frameworks, whereas Corona, Titanium, and NativeScript produce larger apps, and Cordova and native Android produce smaller apps. While the Corona (now Solar2D) framework is not part of the identification framework, Table 6.3 may indicate that both Titanium and NativeScript produce smaller mean file sizes than Xamarin.

These results are more in line with those presented by Willocx et al. (2015), in which the native development approach is compared to apps generated using PhoneGap and Xamarin through a software engineering effort. The authors find that Xamarin for Android generates apps approximately five times the size of the native baseline, and three times the size of PhoneGap. However, for the iOS counterpart, Xamarin had approximately half the file size footprint the Android app had, suggesting that a study similar to the one at hand should be carried out also for the Apple App Store and compared. Such an effort could be of value and importance to practitioners using cross-platform development frameworks for the sake of platform multi-homing (Hyrynsalmi et al., 2016), *i.e.*, the process of developing and publishing a given app to both the Android and iOS ecosystems.

6.4.5 Analysing Trends and Adoption from an Industry Perspective

By looking to Google Trends, we can compare the search interest for cross-platform frameworks with the framework distribution over time from Figure 6.4. By doing so, possible discrepancies in framework adoption versus what is frequently searched for on Google can be identified. Raw data from Google Trends does not include search volume, but instead a range from 0 to 100 per keyword illustrating search interest relative to the other keywords, a process further detailed by Google (n.d.). The Google Trends data was harnessed and processed using the open-source Python library `gsvi` (Pirchner, 2020). This approach was chosen over exporting data directly from Google Trends due to the service's maximum limitation of five keywords and the search trend relativity challenge, a limitation and challenge `gsvi` circumvented due to clever processing and normalisation of the Google Trends data. Due to the possibility of keyword ambiguity, for instance, the meaning of Ionic which could refer to the app development framework, but also to the chemical process "ionic bonding", or to the Fitbit Ionic wearable, the results from Google Trends were filtered based on the category "Programming" (code 31).

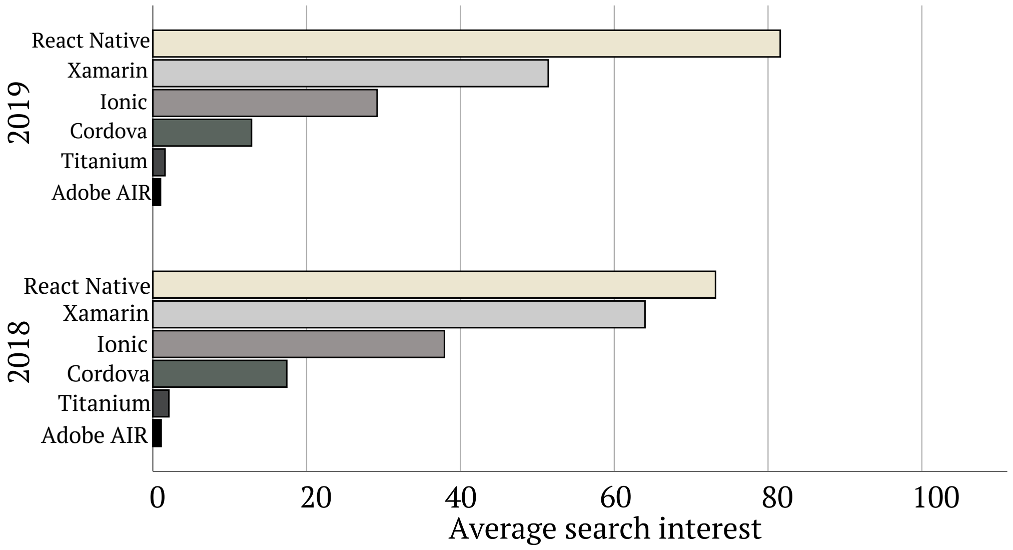


Figure 6.6: Search interest for React Native, Xamarin, Ionic, Cordova, Titanium and Adobe AIR. Data source: Google Trends (https://www.google.com/trends).

The results of this procedure are illustrated in Figure 6.6, in which the search trends for the top six frameworks from 2018 and 2019 as displayed in Figure 6.4 are compared, specifically Xamarin, Ionic, Cordova, React Native, Adobe AIR, and Titanium. We can conclude from Figure 6.6 that Google Trends provides indications that React Native has been searched for more often than the other frameworks listed both in 2018 and 2019. The two other interpreted approach frameworks, *i.e.*, the React Native alternatives – Titanium and Adobe AIR – are to be found at the bottom of the chart. React Native, Ionic and Cordova all saw an increase in search interest from 2018 to 2019, while Xamarin and Titanium both decreased. Adobe AIR saw little search interest during both years. To compare these trends to the results of this experiment, the data from Figure 6.4 and Figure 6.6 has been tabularised and ranked, as can be seen in Table 6.4.

Table 6.4: Comparison of Google Trends from 2018 and 2019 (Fig. 6.6) to the findings (Fig. 6.4).

#	**Google Trends 2018 & 2019**	**Results 2018 (Fig. 6.4)**	**Results 2019 (Fig. 6.4)**
1	React Native	Xamarin (7 219)	Xamarin (10 178)
2	Xamarin	Cordova (5 329)	Adobe AIR (3 457)
3	Ionic	Adobe AIR (4 461)	Cordova (2 573)
4	Cordova	Titanium (1 921)	React Native (2 004)
5	Titanium	Ionic (1 241)	Titanium (1 922)
6	Adobe AIR	React Native (431)	Ionic (772)

Once the Google Trends data and the experiment results are seen side by side, certain discrepancies between search trends and framework adoption in both years can be noted. Particularly React Native is seeing the most industry interest relative to the other frameworks based on the Google Trends data, while based on the experiment results it was the sixth most adopted framework in 2018 and fourth in 2019. This discrepancy could indicate that mobile developers are interested and curious regarding new technologies, although the interest does not necessarily translate into actual adoption and use in published projects. From the developer survey questionnaire on perceived issues and challenges with cross-platform development published in Biørn-Hansen et al. (2019b), framework maturity was highlighted as the third (of eight) most frequently perceived challenge with cross-platform development, which could provide some context to the experiment results. It is also noteworthy that Adobe AIR, the third (2018) and second (2019) most used framework in published Android apps according to Figure 6.4, does not see much search interest at all during either 2018 or 2019, as seen in Figure 6.6. This could potentially indicate that there is a stable community of Adobe AIR developers who keep publishing apps, while the more trending frameworks including React Native are less stable but generates comparably more search interest. While the Google Trends search interests are equal between 2018 and 2019, the framework adoption differs between the two years. Cordova was used in considerably fewer apps in 2019 than in 2018, so much so that Adobe AIR ranks second in adoption in 2019 whereas Cordova was the second most adopted in 2018. It is also noteworthy the quintuple increase in React Native apps between the two years, which could potentially be seen in correlation to the increase in search interest for React Native (see Figure 6.6).

Another industry-based source of data for gauging of framework interest is the State of JavaScript survey from 2018 (n > 20 000 respondents) conducted by Greif et al. (2018). In their survey, React Native is the clear outlier in terms of developer interests, with 53.2% of respondents stating they are interested in learning it. While React Native enjoys the interest of developers, NativeScript is on the other side of the presented quadrant, between the categories “Assess” and “Avoid” due to varying satisfaction. Both Cordova and Ionic are also listed in the survey and quadrant, but both frameworks get a significant percentage of votes for the survey option “Heard of it, not interested”. This contrasts to the presented findings on framework adoption in this current experiment. Noteworthy is also the absence of Xamarin, Adobe AIR and Titanium Appcelerator from the survey, three of the more popular frameworks in use on Google Play Store according to Table 6.4, also more popular than React Native in 2018. Results from previous studies provided indications that Adobe AIR might outperform the native approach on specific tasks according to Dhillon and Mahmoud (2015), and so its absence from developer-oriented studies should perhaps be reviewed. Based on these findings, a suggestion for future surveys similar to what was conducted by Greif et al. (2018), is to include a more comprehensive list of technologies for participants to choose between, possibly leading to a more detailed and accurate view of the use of technologies.

6.5 Conclusion

The presented findings indicate that cross-platform apps account for approximately 15% of the explored dataset. Contrary to the industry-generated hype surrounding certain technologies and frameworks, the hybrid development approach has for years been the most popular path for developing cross-platform apps, only to be challenged by the cross-compiled approach in 2018. While there was an increase in the adoption of React Native and NativeScript in 2016 – frameworks that are frequently discussed in industry outlets, both frameworks saw a decrease during the two years to follow, before React Native had a considerable rise again in 2019. This could indicate that practitioners were particularly interested in hands-on experience with new technologies at the time of release, but did not opt to adopt these until the technologies increased in maturity. In terms of Google Play Store categories, Business, Education and Games all saw plenty of cross-platform apps, unlike categories such as Personalisation and Photography, which both saw minimal use of cross-platform frameworks. Due to the amount of available data for this experiment, some degree generalisation of results is achieved. With a population size of approximately 2.7 million apps (Statista, 2020) in the Google Play Store, the experiment's sample size of $n = 661\,705$ `.apk` files results in a 99% confidence level of accuracy, with an approximate $0,255\%$ margin of error. Thus, the experiment should provide a representative indication of the state of cross-platform framework usage in the Google Play Store.

Chapter 7

Conclusions & Future Work

This chapter synthesizes the new knowledge derived from Chapters 4, 5 and 6, while providing answers to the thesis research questions, and revisiting the thesis aim introduced in Chapter 1. A list of empirically validated guiding principles for conducting mobile development is then presented, after which the research impact, limitations and validity are discussed.

7.1 Performance of Cross-Platform Frameworks

RQ$_1$**:** How do apps developed using cross-platform mobile development approaches and associated frameworks perform compared to native mobile apps in terms of hardware and platform utilisation?

Mobile developers have previously reported that reuse of code for building apps across platform has been challenging (Ahmad et al., 2018). Considering that such is currently not possible in the case of native mobile development, cross-platform approaches and technologies have seen much interest from both academia and industry the last decade. However, these technologies are by practitioners often associated with a performance output inferior to that of a native app (Ahmad et al., 2018, Biørn-Hansen et al., 2019b, Patkar et al., 2020). To investigate the matter of performance of native-side communication and user interfaces, this PhD thesis project has employed a design science research approach to investigate actual performance output, through development and performance measuring of mobile apps and underlying cross-platform frameworks alongside native baseline implementations.

As uncovered in Chapters 4 and 5, there is no "silver bullet" framework or technology for ensuring optimal performance in all situations. Looking at the overall results of Chapter 4 on bridge performance, it is clear that the native Android baseline implementation was the most consistently performant out of the artefacts produced, with the cross-platform approaches either in close proximity or with a significant performance overhead. However, further investigating the details of the findings shows that for instance the time-to-completion metric (`TTC`) on retrieving geolocation coordinates in Ionic is significantly slower than the native baseline implementation by close to a factor of four (3 453,94ms versus 889,05ms), but also that Flutter was close to twice as fast (291,18ms) as the native baseline. In terms of `ComputedRAM` in the geolocation experiment, React Native had the lowest memory consumption across all artefacts (mean of 2,60MB versus the native baseline's 6,56MB). These results, indicating that cross-platform frameworks can in certain situations outperform native, align with previous research, for instance, the performance comparisons reported of by Willocx et al. (2016) in which their native baseline was running on a high-end iOS device consumed the most memory when compared to the performance results of seven out of ten frameworks. For file access and retrieval, NativeScript was the fastest to complete the task (75,58ms versus native's 82,34ms) and had the lowest `CPU%` consumption (17,00% versus native's 18,20%), but consumed second-most `ComputedRAM` (9,30MB versus native's 8,07MB). Thus, there are clear performance trade-offs involved in deciding on mobile development approach and potentially a cross-platform framework, although the differences may be potentially trivial.

As for the animation performance reported in Chapter 5, React Native was found the most performant cross-platform framework out of those included in the benchmark (React Native, Ionic and Xamarin Forms). Different from the bridge performance experiment in Chapter 4 where the Android platform was scrutinised, in this current experiment, both iOS and Android were included. Ionic did also provide good results on specific metrics, including on iOS where it was more CPU efficient than native and the other frameworks tested. However, with measurement data from both Android and iOS, we find an additional layer of complexity in technical decision making: while Ionic on iOS consumed the least CPU in several cases, it consumed the most on Android in all but one case being that of navigation transition performance where it (21,78%) was more performant than both the native baseline (29,34%) and React Native (22,82%). That hybrid-based cross-platform frameworks perform well or even outperform native on in-app page navigation and transitions align with previous research (Willocx et al., 2016), although recent research indicates that they consume more CPU and RAM than native and React Native on user interface interaction tasks (Huber and Demetz, 2019).

In summary, the performance of cross-platform frameworks has been scrutinised both throughout this PhD project and in related work. Generating prescriptive knowledge, for instance, a decision framework, based on the empirical work is seemingly infeasible due to the fast-paced nature of the field and the sheer number of available devices, frameworks, features and other product-specific requirements. Nevertheless, aggregated findings from the conducted experiments indicate that cross-platform frameworks can provide performance benefits for specific tasks and metrics. Ultimately, this challenges the practitioner to consider parameters including but not limited to end-users' device and available hardware, level of performance needed in user interface rendering and native-side bridge communication, and the necessity of native integrations.

7.2 Presence of Cross-Platform Frameworks in the Google Play Store

RQ_2: How common is the presence of cross-platform development frameworks compared to the native development approach in published mobile apps?

From the experiment presented in Chapter 6, we see that cross-platform apps account for approximately 15% ($n = 99\,304$) of the investigated dataset ($n = 661\,705$). This number is significantly lower than Gartner (2013)'s prediction stating that by 2016, half of all mobile apps will be developed using a cross-platform framework, or more specifically, the hybrid development approach. Nevertheless, 15% still display that the use of cross-platform development frameworks is somewhat common, although the native development approach is by far the most common in use. These results align with those presented in the seminal Google Play measurement study by Viennot et al. (2014), in which the authors reported of cross-platform framework usage in approximate 15% of what they categorised as non-popular apps, and in 3%

of popular apps. Their study investigated the use of PhoneGap, Adobe AIR and Titanium, a list further extended in Chapter 6. Ali and Mesbah (2016) also investigated the use of the same list of frameworks as Viennot et al. (2014). Their dataset (n = 80 000) consisted of approximately 20% cross-platform apps, thus a 5% positive difference from both Viennot et al. (2014) and the results from Chapter 6.

The trade-offs discussed in the previous section regarding the first thesis research question (RQ_1) on performance are further fuelled by the results from this experiment on framework presence, as it is also clear that certain frameworks are often significantly more used within specific app categories. Business (~36%), Finance (~25%) and Education (~21%) are examples of categories seeing an above-average number of published apps developed using cross-platform frameworks. This is in contrast to the categories Photography (~2,44%) and Personalisation (~0,41%), where significantly fewer apps than on average are developed using cross-platform frameworks. These results may provide an indication of the suitability of using cross-platform frameworks when developing towards these categories. We know from previous research by Mercado et al. (2016) that end-users may also be impacted by choice of a cross-platform framework as derived by their finding on hybrid app performance where Android users tend to have a more negative perception than iOS users. Also, it may be that the average business or finance app requires fewer native integrations (hence less bridge performance requirements) and perhaps cater more to business-to-business or other end-users who are more focused on functionality than on fancy and animated user interfaces. Indeed, findings presented by Francese et al. (2017) indicate that usability is more critical in consumer-apps than in business-to-business apps, potentially explaining the large number of business and finance apps developed using cross-platform frameworks. Within Photography and Personalisation, however, access to more computational power (photo and video editing) and native integrations (device and home-screen personalisation) may be significant product requirements.

In summary, the presence of cross-platform apps on the Google Play Store is noted, accounting for approximate 15% of the investigated sampled dataset. However, significant differences in the use of cross-platform frameworks emerge when investigating usage between the Play Store app categories. The obvious example is the Business and Finance categories which see more use of cross-platform frameworks than Personalisation and Photography, which are essentially void of cross-platform apps.

7.3 Empirically Validated Principles for Mobile Development

Thesis aim: To provide a set of guiding principles for conducting mobile app development based on results from empirical experiments.

Based on the experiments conducted, a set of empirically verified guiding principles ($GP_{1...5}$) for mobile apps development have been formed. These should be considered descriptive rather than prescriptive. Reasons for this are elaborated on in the upcoming section on limitations and threats to validity.

GP_1: The native development approach is a safe choice in terms of performance of user interfaces and animations, and platform and device feature access, as can be derived from the results of the conducted experiments. Nevertheless, certain functionality and context are found to benefit from the use of cross-platform frameworks in terms of performance, outperforming the native approach. From the animations experiment in Chapter 5, it becomes evident that cross-platform frameworks may differ in performance between Android and iOS – thus where one framework can provide optimal performance on one platform, the performance is less optimal on the other platform. This, in turn, adds an additional layer of complexity for choosing a framework for app development across multiple platforms.

GP_2: According to research by the Google Play team, there is a correlation between `.apk` size and app installs. They report an average 1% decrease in app installations per 6MB increase in `.apk` size, with the impact being more severe in emerging markets (Tolomei, 2017). They also report a 30% higher chance of app download cancellation for apps above 100MB. This is noteworthy, as developing space-conscious apps can be a challenge, especially with cross-platform development frameworks. Indeed, as identified in Chapter 6, most cross-platform frameworks will compile binaries (`.apk`) of significantly larger size than the native approach, which had a mean size of 13 776,04kB. The hybrid development approach was found to generate binaries of approximately the same size as the native apps, with Cordova apps averaging 12 527,55kB and Ionic apps averaging 15 273,53kB. At approximately 2.4x the average size of a native app, we find Xamarin apps averaging 34 142,45kB. Thus, the choice of a cross-platform framework can directly impact the end-user's willingness to download an app due to the generated `.apk` size.

GP_3: It was discovered during both performance experiments (Chapters 4 and 5) that on-device performance measurement tools and automation tools may have an impact on performance readings, through imposing additional performance overhead. This impacted the research design and data gathering methods for both experiments, as automated tools were replaced with manual performance data extraction and data entry to spreadsheets. Thus, it is essential for researchers who make use of- or develop tools for performance measuring to know of- and reflect upon – the indications that such monitoring tools may themselves introduce additional performance overhead. Reddi et al. (2018) calls for the construction of new methods and tools for performance benchmarking on mobile, for which the findings in this thesis further illustrate the need. Indeed, using certain measuring tools could severely impact results from performance measurements if control measurements are not conducted, especially if using framework-specific measurement tools –

i.e., using the built-in monitor in React Native (which may introduce overhead) versus the Android Studio profiler for an Ionic app (which may not introduce overhead). The same also goes for automation tools (Cruz and Abreu, 2019), where recent research has indicated an imposed performance overhead as a result of using such tools. Thus, conducting performance research using a combination of task automation tools along with specific performance measurement tools can lead to skewed and incorrect results.

GP_4: An app's target marketplace category may significantly influence the choice between developing a native app or make use of a specific cross-platform framework, as derived from Chapter 6. A prime example of this from the experiment is the significant difference in the use of cross-platform frameworks between published business apps (35.9%) and personalisation (0.41%) or photography apps (2.74%). There are likely logical reasons for these variations, for instance, computational processing requirements and the significance of native access, and investigating the state of practice can uncover such patterns.

GP_5: Basing technical decisions on industry hype (*hype-driven development*) may not be representative of the actual use of technologies, as investigated in Chapter 6. Whereas React Native enjoys the majority of industry interest, Xamarin, Cordova and Adobe AIR are all significantly more used in recently published apps. This could indicate that many developers will default to technologies of a particular maturity, and instead investigate more novel technologies in unpublished projects.

Thus, accurate technical decision making requires additional parameters, for instance, empirical insights as those generated through Chapter 6.

7.4 Implications for Industry & Practitioners

Discussions on the feasibility and suitability of cross-platform development technologies are frequently encountered in industry outlets and forums. Often, we find these discussions to lack an informed base. This has motivated the work at hand, seeking to provide empirically backed insight and generate new knowledge of how cross-platform technologies perform compared to native apps. For industry, mainly results on bridge performance, animation performance, generated `.apk` sizes, and distribution of cross-platform frameworks across Google Play categories can assist in making informed technical decisions and as background for discussion.

In the performance experiment presented in Chapter 4, a handful of device and platform features were measured using a set of performance-related metrics. These findings indicate significant differences between the frameworks across all metrics, and more intricately, which framework performs the most optimal differs between the tested features as described in Section 7.1 discussing the first thesis research question (RQ_1). This may significantly impact technical decision making for practitioners, as there is seemingly no "silver bullet" in terms of framework or technology.

7.5 Implications for Research

For researchers, the findings on framework usage can be a point of departure for the selection of relevant tools and frameworks for further research. If conducting research which should be of relevance also to practitioners, it should be of great interest to do so using the tools and frameworks practitioners use, alongside exciting and novel frameworks yet to be fully adopted by either practitioners or researchers.

A finding with a direct impact on how we conduct research involving performance measurements is described in GP_3, and relates to the performance impact imposed through the use of automation and performance profiling tools. This also has clear implications for peer review of newer research, and assessment of previous research: has this performance overhead been taken into account in the research design and analysis?

For educators, a better understanding of the popularity and adoption of tools can aid in choosing technologies for hands-on training in (cross-platform) mobile development courses. The results and approach to measuring performance may also be interesting course material for discussing drawbacks and possibilities of cross-platform and native mobile app development.

7.6 Limitations & Validity

Software. A limitation to the thesis experiments is that of the technologies involved in the design and implementation of the technical artefacts used for performance evaluation. The sheer number of technologies available (see Table 2.1) is of such magnitude that it is infeasible to test them all. Thus, for studies conducting artefact implementation using such frameworks and technologies, it has been essential to highlight the specific technologies involved. Provided that two of three experiments focus exclusively on Android, and one experiment on both Android and iOS, findings regarding cross-platform performance and presence on iOS are more limited than results for Android. Although more results from the iOS ecosystem would have been interesting to analyse, we know from practice that cross-platform frameworks are also adopted for other purposes than developing cross-platform apps (as described in Biørn-Hansen et al. (2020)). For instance, they introduce a different mental model than native development and may enable, for instance, Over-the-Air updates and other knowledge requirements, *e.g.*, programming languages. Another threat to the validity is the conscious decision not to conduct any code-wise optimisation of the developed apps for the experiments in Chapters 4 and 5. Thus, the results should reflect the expected performance of a newly initialised app in each of the technologies scrutinised.

Hardware. The significant fragmentation in available Android-based hardware makes it infeasible to conduct performance evaluations on even a statistically representative sample size of devices. While a limitation to this study, it is equally a challenge for practitioners and industry – it is infeasible to maintain a testbed of devices of such magnitude. Thus, a smaller number of devices have been used for performance testing and evaluation. It has been of utmost importance to use a range of devices representing a snapshot of the state of the market during the time of conducting the studies. This includes high-end, mid-range and low-end devices to ensure validity across the device spectrum, and also the inclusion of numerous operating system versions due to the version fragmentation on Android (StatCounter Global Stats, 2020).

Methods and Findings. As for the generalisability of the findings presented throughout this thesis, the previous sections regarding software and hardware limitations are essential. The degree of generalisability varies between the conducted experiments and their derived findings. While the performance experiments, in particular, do not provide prescriptive knowledge applicable to "any and all" contexts or situations, the results can be taken as guidance and could provide empirical insights constrained by the set of employed cross-platform frameworks and mobile devices. Technical decision making involves great complexity and parameters beyond the scope of this thesis; for instance, demographics of the end-user, developer team size, and required time to market. As for the reliability of the results, datasets and algorithms have been open-sourced to ensure repeatability and transparency, employed frameworks and devices are explicitly listed throughout the experiments, and two of three thesis experiments as well as the literature review have undergone peer review and been published in high-ranking outlets, while the last experiment as described in Chapter 6 is in-review as of thesis submission. Statistical tests have been applied, although for future work it would be suggested to lean on the teachings of Fenton and Bieman (2015) into research involving software metrics, for instance the robustness of the median (replacing or in addition to the mean) value in non-normally distributed metrics-based datasets (Fenton and Bieman, 2015, p. 284).

7.7 Summary of Contributions

This thesis contributes to knowledge and practice by introducing a set of empirically validated guiding principles for conducting mobile development. Not only should these principles be of interest to practitioners and fellow researchers, they may also help inform processes such as academic peer review of experimental designs and results. These principles have been derived from three empirical experiments investigating the presence and performance of cross-platform frameworks, two of which have at the time of thesis completion been peer-reviewed and published, with the third study in-review. Alongside this core contribution, datasets, tools, apps and scripts have been open-sourced for increased reliability, reproducibility and to provide a point of departure for future research. A detailed overview of contributions are listed in Section 1.5.

7.8 Suggestions for Future Work

Motivated by ongoing discussions in industry and academia on the performance and usage of cross-platform frameworks, this thesis makes an effort to investigate both matters empirically using a design science research approach. Through two experiments focusing on the performance (RQ_1) of cross-platform apps, it is found that contrary to frequently encountered claims and allegations, the performance of apps generated using cross-platform frameworks is not inherently inferior to that of a native app. The presented findings indicate that while the native development approach overall scores highest in terms of performance output, cross-platform frameworks and approaches can score better on specific metrics and in certain situations. The presence (RQ_2) of cross-platform apps in the Google Play Store is evident, accounting for approximately 15% of the sampled dataset ($n = 661\,705$). It is noteworthy that there are significant variations in the adoption of cross-platform frameworks between the Google Play Store categories, indicating that adopting a cross-platform framework may be better suited for certain types of apps, for instance in the business, education and lifestyle segments as per the findings. A set of empirically verified principles for (cross-platform) mobile development has been presented as a part of the thesis contributions, aiming to provide actionable insights to practitioners and researchers in industry and academia. The proliferation of apps, smartphones, novel device types, technologies and frameworks calls for continuous research going forward, particularly qualitatively investigating the performance of cross-platform apps from a user perspective.

Looking to Table 2.2 we find that the state of research on cross-platform development demands a particular focus on user experience and security going forward. There is an evident absence of qualitative studies on user experience, investigating the actuality of users' perception of user interfaces and usability of cross-platform apps, hereunder also accessibility. This is most useful, as allegations and claims without empirical backing are frequently encountered in both scholarly research and in practitioners' outlets.

Due to the nature of mobile development and the pace at which innovation and development take place, there is a need for continuous research going forward. Not only does the pace of innovation equal new and novel device types, but it also means novel frameworks and technologies for app development. While numerous cross-platform technologies and frameworks have been released over the last few years, *e.g.*, React Native, NativeScript, Platform Uno, .NET MAUI, Flutter *etc.*, there are good reasons to believe that innovation in tooling and technologies will continue demanding the attention of researchers. As novel frameworks are introduced, older ones deprecate, such as the recently announced discontinuation of support for the PhoneGap framework (Adobe I/O, 2020) – leaving developers looking for guidance on choosing the next technology for (cross-platform) mobile development.

Bibliography

Mustafa Abousaleh, David Yarish, Deepali Arora, Stephen W Neville, and Thomas E Darcie. Determining per-mode battery usage within non-trivial mobile device apps. *Proceedings - International Conference on Advanced Information Networking and Applications, AINA*, pages 202–209, May 2014. ISSN 1550445X. URL https://doi.org/10.1109/AINA.2014.29.

Timothy Yudi Adinugroho, Reina, and Josef Bernadi Gautama. Review of multi-platform mobile application development using WebView: Learning management system on mobile platform. In *Procedia Computer Science*, volume 59, pages 291–297. Elsevier, August 2015. URL http://doi.org/10.1016/j.procs.2015.07.568.

Adobe I/O. Update for customers using PhoneGap and PhoneGap build. https://blog.phonegap.com/update-for-customers-using-phonegap-and-phonegap-build-cc701c77502c, August 2020. Accessed: 2020-8-12.

Sara Seif Afjehei, Tse-Hsun (peter) Chen, and Nikolaos Tsantalis. iPerfDetector: Characterizing and detecting performance anti-patterns in iOS applications. *Empirical Software Engineering*, 24(6):3484–3513, December 2019. URL https://doi.org/10.1007/s10664-019-09703-y.

Arshad Ahmad, Kan Li, Chong Feng, Syed Mohammad Asim, Abdallah Yousif, and Shi Ge. An empirical study of investigating mobile applications development challenges. *IEEE Access*, 6:17711–17728, March 2018. URL http://doi.org/10.1109/ACCESS.2018.2818724.

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*, CompSysTech '16, pages 41–48, New York, NY, USA, June 2016. ACM. URL http://doi.org/10.1145/2983468.2983484.

Ricardo Alcocer. But I thought titanium was cross platform!?, 23 July 2013. URL http://www.appcelerator.com/blog/2013/07/but-i-thought-titanium-was-cross-platform/. Accessed: 2017-8-3.

Mohamed Ali and Ali Mesbah. Mining and characterizing hybrid apps. In *Proceedings of the International Workshop on App Market Analytics*, WAMA 2016, pages 50–56, New York, NY, USA, November 2016. ACM. URL http://doi.org/10.1145/2993259.2993263.

Kevin Allix, Tegawendé F Bissyandé, Jacques Klein, and Yves Le Traon. AndroZoo: Collecting millions of android apps for the research community. In *2016 IEEE/ACM 13th Working Conference on Mining Soft-*

ware Repositories (MSR), pages 468–471. IEEE, May 2016. URL http://doi.org/10.1145/2901739.2903508.

Sophie Alpert. Introducing react native. https://reactjs.org/blog/2015/03/26/introducing-react-native.html, March 2015. Accessed: 2019-9-12.

Android Developer. GPU monitor, n.d. URL https://developer.android.com/studio/profile/am-gpu.html. Accessed: 2018-2-25.

Android Developers. WebView, n.d. URL https://developer.android.com/reference/android/webkit/WebView.html. Accessed: 2017-8-3.

Androidrank. Free android market data, history, ranking, 2019. URL https://www.androidrank.org/.

Todd Anglin and Telerik. Web, native and cross-platform - three approaches to mobile app development. Technical report, Telerik, 2014. URL https://www.telerik.com/docs/default-source/whitepapers/choose-right-approach-mobile-app-developmentbb581d10116543e79a9febdb187fd0a3.pdf?sfvrsn=0.

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. ACM, September 2014. URL https://doi.org/10.1145/2662253.2662280.

Anonymous. Google issue TrackerAndroid O prevents access to /proc/stat, March 2017. URL https://issuetracker.google.com/issues/37140047. Accessed: 2018-1-23.

Appcelerator. Appcelerator platform, n.d. URL http://docs.appcelerator.com/platform/latest/#!/guide/Titanium_Platform_Overview. Accessed: 2018-2-1.

Lucas Pugliese Barros, Flávio Medeiros, Eduardo Cardoso Moraes, and Anderson Feitosa Júnior. Analyzing the performance of apps developed by using Cross-Platform and native technologies. In *SEKE 2020 Proceedings*, December 2020. URL https://doi.org/10.18293/SEKE2020-122.

Richard Baskerville. What design science is not. *European Journal of Information Systems*, 17(5):441–443, October 2008. URL http://doi.org/10.1057/ejis.2008.45.

Kayce Basques. Get started with analyzing runtime performance, January 2018. URL https://developers.google.com/web/tools/chrome-devtools/evaluate-performance/. Accessed: 2018-3-12.

Andreas Biørn-Hansen and Gheorghita Ghinea. Bridging the gap: Investigating Device-Feature Exposure in Cross-Platform development. In *Proceedings of the 51st Hawaii International Conference on System Sciences*, pages 5717–5724. ScholarSpace, January 2018. URL http://doi.org/10.24251/HICSS.2018.716.

Andreas Biørn-Hansen, Tor-Morten Grønli, and Gheorghita Ghinea. Cross-platform frameworks in google play store: Trends and directions. In *ACM Symposium on Applied Computing (SAC)*. ACM, (In-review).

Andreas Biørn-Hansen, Tor-Morten Grønli, and Gheorghita Ghinea. Baseline requirements for comparative research on Cross-Platform mobile development: A literature survey. In *Proceedings of the 30th Norwegian Informatics Conference*. Bibsys, November 2017. URL http://ojs.bibsys.no/index.php/NIK/article/view/427.

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*, 51(5), November 2018a. URL http://doi.org/10.1145/3241739.

Andreas Biørn-Hansen, Tim A Majchrzak, and Tor-Morten Grønli. Progressive web apps for the unified development of mobile applications. In *Web Information Systems and Technologies*, volume 322 of *Lecture Notes in Business Information Processing*. Springer, July 2018b. URL http://doi.org/10.1007/978-3-319-93527-0.

Andreas Biørn-Hansen, Tor-Morten Grønli, and Gheorghita Ghinea. Animations in cross-platform mobile applications: An evaluation of tools, metrics and performance. *MDPI Sensors*, 19(9), May 2019a. URL https://doi.org/10.3390/s19092081.

Andreas Biørn-Hansen, Tor-Morten Grønli, Gheorghita Ghinea, and Sahel Alouneh. An empirical study of cross-platform mobile development in industry. *Wiley Hindawi Wireless Communications and Mobile Computing*, 2019, January 2019b. URL https://doi.org/10.1155/2019/5743892.

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. *Springer Empirical Software Engineering*, 25(4):2997–3040, June 2020. URL https://doi.org/10.1007/s10664-020-09827-6.

Nader Boushehrinejadmoradi, Vinod Ganapathy, Santosh Nagarakatte, and Liviu Iftode. Testing Cross-Platform mobile app development frameworks (t). In *2015 30th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 441–451. IEEE, November 2015. URL http://doi.org/10.1109/ASE.2015.21.

Can I Use. Geolocation API, n.d.a. URL https://caniuse.com/#feat=geolocation. Accessed: 2018-2-16.

Can I Use. getUserMedia/Stream API, n.d.b. URL https://caniuse.com/#feat=stream. Accessed: 2018-2-16.

Maria Caulo, Rita Francese, Giuseppe Scanniello, and Antonio Spera. Does the migration of cross-platform apps towards the android platform matter? an approach and a user study. In *Product-Focused Software Process Improvement*, pages 120–136. Springer International Publishing, November 2019. URL https://doi.org/10.1007/978-3-030-35333-9_9.

Kwame Chan-Jong-Chu, Tanjina Islam, Miguel Morales Exposito, Sanjay Sheombar, Christian Valladares, Olivier Philippot, Eoin Martino Grua, and Ivano Malavolta. Investigating the correlation between performance scores and energy consumption of mobile web apps. In *Proceedings of the Evaluation and Assessment in Software Engineering*, EASE '20, pages

190–199, New York, NY, USA, April 2020. Association for Computing Machinery. URL https://doi.org/10.1145/3383219.3383239.

Andre Charland and Brian LeRoux. Mobile application development: Web vs. native. *Queue*, 9(4):20, April 2011. URL http://doi.org/10.1145/1966989.1968203.

Matteo Ciman and Ombretta Gaggi. An empirical analysis of energy consumption of cross-platform frameworks for mobile development. *Pervasive and Mobile Computing*, October 2016. URL https://doi.org/10.1016/j.pmcj.2016.10.004.

Riccardo Coppola, Luca Ardito, and Marco Torchiano. Characterizing the transition to kotlin of android apps: a study on F-Droid, play store, and GitHub. In *Proceedings of the 3rd ACM SIGSOFT International Workshop on App Market Analytics - WAMA 2019*, pages 8–14, New York, New York, USA, August 2019. ACM Press. URL https://doi.org/10.1145/3340496.3342759.

Leonardo Corbalan, Juan Fernandez, Alfonso Cuitiño, Lisandro Delia, Germán Cáseres, Pablo Thomas, and Patricia Pesado. Development frameworks for mobile devices: A comparative study about energy consumption. In *Proceedings of the 5th International Conference on Mobile Software Engineering and Systems*, MOBILESoft '18, pages 191–201, New York, NY, USA, May 2018. ACM. URL https://doi.org/10.1145/3197231.3197242.

Leonardo Corbalán, Pablo Javier Thomas, Lisandro Delia, Germán Cáseres, Patricia Pesado, and others. A study of non-functional requirements in apps for mobile devices. In *Cloud Computing and Big Data*, volume 1050 of *Communications in Computer and Information Science*, pages 125–136. Springer, July 2019. URL https://doi.org/10.1007/978-3-030-27713-0_11.

Cordova. Architectural overview of cordova platform, n.d. URL https://cordova.apache.org/docs/en/latest/guide/overview/index.html. Accessed: 2017-8-3.

Luis Corral, Andrea Janes, and Tadas Remencius. Potential advantages and disadvantages of multiplatform development Frameworks–A vision on mobile environments. In *Procedia Computer Science*, volume 10, pages 1202–1207. SciVerse ScienceDirect, August 2012a. URL https://doi.org/10.1016/j.procs.2012.06.173.

Luis Corral, Alberto Sillitti, and Giancarlo Succi. Mobile multiplatform development: An experiment for performance analysis. *Procedia Computer Science*, 10:736–743, January 2012b. URL https://doi.org/10.1016/j.procs.2012.06.094.

Luis Cruz and Rui Abreu. Performance-Based guidelines for energy efficient mobile applications. In *Proceedings of the 2017 IEEE/ACM 4th International Conference on Mobile Software Engineering and Systems (MOBILESoft)*, pages 46–57. IEEE/ACM, May 2017. URL https://doi.org/10.1109/MOBILESoft.2017.19.

Luis Cruz and Rui Abreu. On the energy footprint of mobile testing frameworks. *IEEE Transaction on Software Engineering*, October 2019. URL https://doi.org/10.1109/TSE.2019.2946163.

Isabelle Dalmasso, Soumya Kanti Datta, Christian Bonnet, and Navid Nikaein. Survey, comparison and evaluation of cross platform mobile application development tools. In *Wireless Communications and Mobile Computing Conference (IWCMC), 2013 9th International*, pages 323–328. IEEE, July 2013. URL http://doi.org/10.1109/IWCMC.2013.6583580.

Lisandro Delía, Nicolás Galdamez, Leonardo Corbalan, Patricia Pesado, and Pablo Thomas. Approaches to mobile application development: Comparative performance analysis. In *Proceedings of the 2017 Computing Conference*, pages 652–659, July 2017. URL https://doi.org/10.1109/SAI.2017.8252165.

Sunny Dhillon and Qusay H Mahmoud. An evaluation framework for cross-platform mobile application development tools. *Software: Practice and Experience*, 45(10):1331–1357, October 2015. URL http://doi.org/10.1002/spe.2286.

Thomas Dorfer, Lukas Demetz, and Stefan Huber. Impact of mobile cross-platform development on CPU, memory and battery of mobile devices when using common mobile app features. *Procedia Computer Science*, 175:189–196, August 2020. URL https://doi.org/10.1016/j.procs.2020.07.029.

Amnon H Eden. Three paradigms of computer science. *Minds and Machines*, 17(2):135–167, July 2007. URL http://doi.org/10.1007/s11023-007-9060-8.

Wafaa S El-Kassas, Bassem A Abdullah, Ahmed H Yousef, and Ayman M Wahba. ICPMD: Integrated cross-platform mobile development solution. In *2014 9th International Conference on Computer Engineering Systems (ICCES)*, pages 307–317. IEEE, December 2014. URL http://doi.org/10.1109/ICCES.2014.7030977.

Wafaa S El-Kassas, Bassem A Abdullah, Ahmed H Yousef, and Ayman M Wahba. Enhanced code conversion approach for the integrated cross-platform mobile development (icpmd). *IEEE Transactions on Software Engineering*, 42(11):1036–1053, March 2016. URL https://doi.org/10.1109/TSE.2016.2543223.

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, June 2017. URL https://doi.org/10.1016/j.asej.2015.08.004.

Emelie Engström, Margaret-Anne Storey, Per Runeson, Martin Höst, and Maria Teresa Baldassarre. How software engineering research aligns with design science: a review. *Empirical Software Engineering*, pages 2630––2660, April 2020. URL https://doi.org/10.1007/s10664-020-09818-7.

Jan Ernsting, Christoph Rieger, Fabian Wrede, and Tim A Majchrzak. Refining a reference architecture for Model-Driven business apps. In *12th*

International Conference on Web Information Systems and Technologies, pages 307–316. Scitepress, April 2016. URL https://doi.org/10.5220/0005862103070316.

Clément Escoffier and Philippe Lalanda. Managing the heterogeneity and dynamism in hybrid mobile applications. In *2015 IEEE International Conference on Services Computing*, pages 74–81. IEEE, June 2015. URL http://doi.org/10.1109/SCC.2015.20.

Clément Escoffier, Philippe Lalanda, and Ozan Gunalp. A component model to manage the heterogeneity and dynamism in mobile applications. In *Proceedings of the 18th International ACM SIGSOFT Symposium on Component-Based Software Engineering*, CBSE '15, pages 85–90, New York, NY, USA, May 2015. ACM. URL http://doi.org/10.1145/2737166.2737178.

El Hassane Ettifouri, Abdelkader Rhouati, Jamal Berrich, and Toumi Bouchentouf. Toward a merged approach for cross-platform applications (web, mobile and desktop). In *Proceedings of the 2017 International Conference on Smart Digital Environment*, ICSDE '17, pages 207–213, New York, NY, USA, July 2017. ACM. URL http://doi.org/10.1145/3128128.3128160.

Facebook. JavaScript environment, 7 June 2017. URL https://facebook.github.io/react-native/docs/javascript-environment.html. Accessed: 2017-8-3.

Facebook. React native, 2018. URL https://facebook.github.io/react-native/. Accessed: 2018-NA-NA.

Norman E Fenton and James Bieman. *Software Metrics: A Rigorous and Practical Approach, Third Edition*. Taylor & Francis Group, 2015.

Andy Field and Graham J Hole. *How to Design and Report Experiments*. SAGE Publications Ltd, 1 edition, February 2003.

Maximiliano Firtman. *High Performance Mobile Web*. O'Reilly Media, 2016. ISBN 9781491912553. URL http://shop.oreilly.com/product/0636920035060.do.

Maximiliano Firtman. *Hacking Web Performance*. O'Reilly Media, May 2018a. ISBN 9781492039396.

Maximiliano Firtman. Progressive web apps on iOS are here. https://medium.com/@firt/progressive-web-apps-on-ios-are-here-d00430dee3a7, March 2018b. URL https://medium.com/@firt/progressive-web-apps-on-ios-are-here-d00430dee3a7. Accessed: 2018-6-21.

Rita Francese, Carmine Gravino, Michele Risi, Giuseppe Scanniello, and Genoveffa Tortora. Mobile app development and management: Results from a qualitative investigation. In *2017 IEEE/ACM 4th International Conference on Mobile Software Engineering and Systems (MOBILESoft)*, pages 133–143. IEEE/ACM, May 2017. URL https://doi.org/10.1109/MOBILESoft.2017.33.

GadgetVersus. Apple A13 bionic specs. https://gadgetversus.com/processor/apple-a13-bionic-specs/. Accessed: 2020-5-28.

Lamia Gaouar, Abdelkrim Benamar, and Fethi Tarik Bendimerad. Model driven approaches to cross platform mobile development. In *Proceedings of the International Conference on Intelligent Information Processing, Security and Advanced Communication*, IPAC '15, pages 19:1–19:5, New York, NY, USA, November 2015. ACM. URL http://doi.org/10.1145/2816839.2816882.

Gartner. Gartner says by 2016, more than 50 percent of mobile apps deployed will be hybrid. http://www.gartner.com/newsroom/id/2324917, February 2013. Accessed: 2016-1-12.

Matt Gaunt. Service workers: an introduction, January 2018. URL https://developers.google.com/web/fundamentals/primers/service-workers/. Accessed: 2018-1-22.

Bruno Góis Mateus and Matias Martinez. An empirical study on quality of android applications written in kotlin language. *Empirical Software Engineering*, (24):3356–3393, June 2019. URL https://doi.org/10.1007/s10664-019-09727-4.

Nizamettin Gok and Nitin Khanna. *Building Hybrid Android Apps with Java and JavaScript*. O'Reilly Media, Incorporated, 2013. ISBN 9781449361914. URL http://shop.oreilly.com/product/0636920028994.do.

Google. Ola drives mobility for a billion indians with progressive web app.

https://developers.google.com/web/showcase/2017/ola, November 2017. Accessed: 2020-4-24.

Google. FAQ about google trends data. https://support.google.com/trends/answer/4365533?hl=en, n.d. Accessed: 2020-4-29.

Google LLC. Profile and debug pre-build APKs, 2019a. URL https://developer.android.com/studio/debug/apk-debugger. Accessed: 2019-5-2.

Google LLC. DateTime class - dart:core library - Dart API, 2019b. URL https://api.dartlang.org/stable/2.3.0/dart-core/DateTime-class.html. Accessed: 2019-5-21.

Google LLC. Benchmark app code, 2019c. URL https://developer.android.com/studio/profile/benchmark. Accessed: 2019-5-21.

Tony Gorschek, Ewan Tempero, and Lefteris Angelis. On the use of software design models in software development practice: An empirical investigation. *Journal of Systems and Software*, 95:176–193, September 2014. URL http://doi.org/10.1016/j.jss.2014.03.082.

Shirley Gregor and Alan R Hevner. Positioning and presenting design science research for maximum impact. *MIS Quarterly*, 37(2):337–356, June 2013. URL https://doi.org/10.25300/MISQ/2013/37.2.01.

Sacha Greif, Raphaël Benitte, and Michael Rambeau. Mobile & desktop - overview. https://2018.stateofjs.com/mobile-and-desktop/overview/, November 2018. Accessed: 2019-9-18.

Chris Griffith. *Mobile App Development with Ionic2: Cross-Platform Apps with Ionic 2, Angular 2, and Cordova*. O'Reilly Media, April 2017. ISBN 9781491937785. URL http://shop.oreilly.com/product/0636920044710.do.

Tor-Morten Grønli and Gheorghita Ghinea. Meeting quality standards for mobile application development in businesses: A framework for Cross-Platform testing. In *2016 49th Hawaii International Conference on System Sciences (HICSS)*, pages 5711–5720, January 2016. URL http://doi.org/10.1109/HICSS.2016.706.

Tor-Morten Gronli, Jarle Hansen, Gheorghita Ghinea, and Muhammad Younas. Mobile application platform heterogeneity: Android vs windows phone vs iOS vs firefox OS. In *2014 IEEE 28th International Conference on Advanced Information Networking and Applications*, pages 635–641. IEEE, May 2014. URL https://doi.org/10.1109/AINA.2014.78.

Sarra Habchi, Geoffrey Hecht, Romain Rouvoy, and Naouel Moha. Code smells in iOS apps: How do they compare to android? In *Proceedings of the 2017 IEEE/ACM 4th International Conference on Mobile Software Engineering and Systems (MOBILESoft)*, pages 110–121. IEEE/ACM, May 2017. URL https://doi.org/10.1109/MOBILESoft.2017.11.

Henning Heitkötter and Tim A Majchrzak. Cross-Platform development of business apps with MD2. In *Design Science at the Intersection of Physical and Virtual Design*, Lecture Notes in Computer Science, pages 405–411. Springer, Berlin, Heidelberg, June 2013. URL http://doi.org/10.1007/978-3-642-38827-9_29.

Henning Heitkötter, Sebastian Hanschke, and Tim A Majchrzak. Comparing cross-platform development approaches for mobile applications. In *Proceedings of the 8th International Conference on Web Information Systems and Technologies (WEBIST)*, pages 299–311. SciTePress, April 2012a. URL https://doi.org/10.5220/0003904502990311.

Henning Heitkötter, Sebastian Hanschke, and Tim A Majchrzak. Evaluating Cross-Platform development approaches for mobile applications. In *Web Information Systems and Technologies*, Lecture Notes in Business Information Processing, pages 120–138. Springer Berlin Heidelberg, April 2012b. URL https://doi.org/10.1007/978-3-642-36608-6_8.

Henning Heitkötter, Tim A Majchrzak, and Herbert Kuchen. Cross-platform model-driven development of mobile applications with md2. In *Proceedings of the 28th Annual ACM Symposium on Applied Computing*, SAC '13, pages 526–533, New York, NY, USA, March 2013. ACM. URL https://doi.org/10.1145/2480362.2480464.

Alan R Hevner. A three cycle view of design science research. *Scandinavian Journal of Information Systems*, 19(2):4, January 2007. URL http://aisel.aisnet.org/sjis/vol19/iss2/4.

Alan R Hevner, Salvatore T March, Jinsoo Park, and Sudha Ram. Design science in information systems research. *MIS Quarterly*, 28:75–105, March 2004. URL https://doi.org/10.2307/25148625.

Dennis E Hinkle, William Wiersma, Stephen G Jurs, and Others. *Applied* *statistics for the behavioral sciences*, volume 5. Houghton Mifflin Boston, 1988. URL https://doi.org/10.2307/1164825.

Leonard Hoon, Rajesh Vasa, Jean-Guy Schneider, John Grundy, and Others. An analysis of the mobile app review landscape: trends and implications. Technical report, Swinburne University of Technology, July 2013.

Stefan Huber and Lukas Demetz. Performance analysis of mobile cross-platform development approaches based on typical ui interactions. In *Proceedings of the 14th International Conference on Software Technologies*, pages 40–48. INSTICC, SciTePress, July 2019. ISBN 978-989-758-379-7. URL https://doi.org/10.5220/0007838000400048.

Stefan Huber, Lukas Demetz, and Michael Felderer. Analysing the performance of mobile cross-platform development approaches using UI interaction scenarios. In *Communications in Computer and Information Science*, pages 40–57. Springer International Publishing, July 2020. URL https://doi.org/10.1007/978-3-030-52991-8_3.

Jussi Huhtala, Ari-Heikki Sarjanoja, Jani Mäntyjärvi, Minna Isomursu, and Jonna Häkkilä. Animated UI transitions and perception of time: A user study on animated effects on a mobile screen. In *Proceedings of the 28th SIGCHI Conference on Human Factors in Computing Systems*, CHI '10, pages 1339–1342, New York, NY, USA, April 2010. ACM. URL http://doi.org/10.1145/1753326.1753527.

Ngu Phuc Huy and Do vanThanh. Evaluation of mobile app paradigms. In *Proceedings of the 10th International Conference on Advances in Mobile*

Computing & Multimedia, pages 25–30. ACM, December 2012. URL http://doi.org/10.1145/2428955.2428968.

Sami Hyrynsalmi, Arho Suominen, and Matti Mäntymäki. The influence of developer multi-homing on competition between software ecosystems. *Journal of Systems and Software*, 111:119–127, January 2016. URL https://doi.org/10.1016/j.jss.2015.08.053.

Juhani Iivari. A paradigmatic analysis of information systems as a design science. *Scandinavian journal of information systems*, January 2007. URL http://aisel.aisnet.org/cgi/viewcontent.cgi?article=1018&context=sjis.

Andreas Itzchak Rehberg. F-Droid main repository. https://apt.izzysoft.de/fdroid/?repo=main, February 2020. Accessed: 2020-2-25.

Xiaoping Jia and Christopher Jones. Design of adaptive domain-specific modeling languages for model-driven mobile application development. In *2015 10th International Joint Conference on Software Technologies (ICSOFT)*, volume 1, pages 413–418. IEEE/ScitePress, July 2015. URL https://doi.org/10.5220/0005557404130418.

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. CPS, October 2013. URL http://doi.org/10.1109/ESEM.2013.9.

Georgios Kambourakis, Asaf Shabtai, Constantinos Kolias, and Dimitrios

Damopoulos. *Intrusion Detection and Prevention for Mobile Ecosystems*. CRC Press, September 2017. ISBN 9781138033573.

Kleomenis Katevas, Hamed Haddadi, and Laurissa Tokarchuk. Sensing kit: Evaluating the sensor power consumption in iOS devices. *Proceedings - 12th International Conference on Intelligent Environments, IE 2016*, pages 222–225, September 2016. URL https://doi.org/10.1109/IE.2016.50.

Roger E Kirk. Practical significance: A concept whose time has come. *Educational and Psychological Measurement*, 56(5):746–759, October 1996. URL https://doi.org/10.1177/0013164496056005002.

Elmar Krainz, Johannes Feiner, and Martin Fruhmann. Accelerated development for accessible apps – model driven development of transportation apps for visually impaired people. In *Human-Centered and Error-Resilient Systems Development*, Lecture Notes in Computer Science, pages 374–381. Springer, Cham, August 2016. URL http://doi.org/10.1007/978-3-319-44902-9.

Dean Kramer, Tony Clark, and Samia Oussena. MobDSL: A domain specific language for multiple mobile platform deployment. In *2010 IEEE International Conference on Networked Embedded Systems for Enterprise Applications*, pages 1–7. IEEE, November 2010. URL http://doi.org/10.1109/NESEA.2010.5678062.

Mohamed Lachgar and Abdelmounaïm Abdali. Decision framework for mobile development methods. *International Journal of Advanced Computer*

Science and Applications, 8(2):110–118, 2017. URL http://doi.org/10.14569/IJACSA.2017.080215.

Mounaim Latif, Younes Lakhrissi, El Habib Nfaoui, and Najia Es-Sbai. Cross platform approach for mobile application development: A survey. In *2016 International Conference on Information Technology for Organizations Development (IT4OD)*, pages 1–5. IEEE, March 2016. URL http://doi.org/10.1109/IT4OD.2016.7479278.

Olivier Le Goaer and Sacha Waltham. Yet another DSL for cross-platforms mobile development. In *Proceedings of the First Workshop on the Globalization of Domain Specific Languages*, GlobalDSL '13, pages 28–33, New York, NY, USA, July 2013. ACM. URL https://doi.org/10.1145/2489812.2489819.

Paul Lewis. Rendering performance, September 2017. URL https://developers.google.com/web/fundamentals/performance/rendering/. Accessed: 2017-11-3.

Li Li, Tegawendé F Bissyandé, Jacques Klein, and Yves Le Traon. An investigation into the use of common libraries in android apps. In *2016 IEEE 23rd International Conference on Software Analysis, Evolution, and Reengineering (SANER)*, volume 1, pages 403–414. IEEE, March 2016. URL https://doi.org/10.1109/SANER.2016.52.

Li Li, Jun Gao, Médéric Hurier, Pingfan Kong, Tegawendé F Bissyandé, Alexandre Bartel, Jacques Klein, and Yves Le Traon. AndroZoo++: Collecting millions of android apps and their metadata for the research community. *arXiv [cs. SE]*, September 2017. URL https://arxiv.org/abs/1709.05281.

Tongbo Luo, Hao Hao, Wenliang Du, Yifei Wang, and Heng Yin. Attacks on WebView in the android system. In *Proceedings of the 27th Annual Computer Security Applications Conference*, pages 343–352. ACM, December 2011. URL http://doi.org/10.1145/2076732.2076781.

Max Lynch. Testing geolocation on Android, January 2018. URL https://blog.ionicframework.com/testing-geolocation-on-android/. Accessed: 2018-8-23.

Tim A Majchrzak and Jan Ernsting. Achieving business practicability of model-driven cross-platform apps. *Open Journal of Information Systems*, 2(2):3–14, 2015. URL http://hdl.handle.net/11250/2392249.

Tim A Majchrzak and Henning Heitkötter. Status Quo and Best Practices of App Development in Regional Companies. In Karl-Heinz Krempels and Alexander Stocker, editors, *Revised Selected Papers Web Information Systems and Technologies (WEBIST) 2013*, volume 189 of *Lecture Notes in Business Information Processing (LNBIP)*, pages 189–206. Springer, July 2014. URL https://doi.org/10.1007/978-3-662-44300-2_12.

Tim A Majchrzak, Andreas Biørn-Hansen, and Tor-Morten Grønli. Comprehensive analysis of innovative Cross-Platform app development frameworks. In *Proceedings of the 50th Hawaii International Conference on System Sciences*, pages 6162–6171. ScholarSpace, January 2017. URL https://doi.org/10.24251/HICSS.2017.745.

Tim A Majchrzak, Andreas Biørn-Hansen, and Tor-Morten Grønli. Progressive web apps: the definite approach to Cross-Platform development? In *Proceedings of the 51st Hawaii International Conference on System Sciences*, pages 5735–5745. ScholarSpace, January 2018. URL https://doi.org/10.24251/HICSS.2018.718.

Ivano Malavolta, Stefano Ruberto, Tommaso Soru, and Valerio Terragni. Hybrid mobile apps in the google play store: An exploratory investigation. In *Proceedings of the Second ACM International Conference on Mobile Software Engineering and Systems*, MOBILESoft '15, pages 56–59, Piscataway, NJ, USA, May 2015a. IEEE. URL https://doi.org/10.1109/MobileSoft.2015.15.

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, June 2015b. URL https://doi.org/10.1109/MobServ.2015.14.

Ivano Malavolta, Giuseppe Procaccianti, Paul Noorland, and Petar Vukmirović. Assessing the impact of service workers on the energy efficiency of progressive web apps. In *Proceedings of the 4th International Conference on Mobile Software Engineering and Systems*, MOBILESoft '17, pages 35–45, Piscataway, NJ, USA, May 2017. IEEE. URL https://doi.org/10.1109/MOBILESoft.2017.7.

Ivano Malavolta, Katerina Chinnappan, Lukas Jasmontas, Sarthak Gupta, and Kaveh Ali Karam Soltany. Evaluating the Impact of Caching on the Energy Consumption and Performance of Progressive Web Apps. In *7th*

IEEE/ACM International Conference on Mobile Software Engineering and Systems 2020, MOBILESoft '20, pages 109–119, July 2020. URL https://doi.org/10.1145/3387905.3388593.

Salvatore T March and Gerald F Smith. Design and natural science research on information technology. *Decision Support Systems*, 15(4):251–266, December 1995. URL https://doi.org/10.1016/0167-9236(94)00041-2.

William Martin, Federica Sarro, Yue Jia, Yuanyuan Zhang, and Mark Harman. A survey of app store analysis for software engineering. *IEEE Transaction on Software Engineering*, 43(9):817–847, December 2017. URL https://doi.org/10.1109/TSE.2016.2630689.

Shawn Maust. What the jank?, August 2015. URL https://afasterweb.com/2015/08/29/what-the-jank/. Accessed: 2018-2-27.

Iván Tactuk Mercado, Nuthan Munaiah, and Andrew Meneely. The impact of cross-platform development approaches for mobile applications from the user's perspective. In *Proceedings of the International Workshop on App Market Analytics*, WAMA 2016, pages 43–49, New York, NY, USA, November 2016. ACM. URL https://doi.org/10.1145/2993259.2993268.

Maria Moloney and Liam Church. Engaged scholarship: Action design research for new software product development. In *Thirty Third International Conference on Information Systems*. AIS, December 2012. URL https://doi.org/10.2139/ssrn.2227590.

MoSync AB. MoSync, 2015. URL https://github.com/MoSync/MoSync.

Simone Mutti, Yanick Fratantonio, Antonio Bianchi, Luca Invernizzi, Jacopo Corbetta, Dhilung Kirat, Christopher Kruegel, and Giovanni Vigna. Baredroid: Large-scale analysis of android apps on real devices. In *Proceedings of the 31st Annual Computer Security Applications Conference*, ACSAC 2015, pages 71–80. ACM, December 2015. ISBN 978-1-4503-3682-6. URL https://doi.org/10.1145/2818000.2818036.

Ehsan Noei, Mark D Syer, Ying Zou, Ahmed E Hassan, and Iman Keivanloo. A study of the relation of mobile device attributes with the user-perceived quality of android apps. *Empirical Software Engineering*, 22(6):3088–3116, December 2017. URL https://doi.org/10.1007/s10664-017-9507-3.

Jay F Nunamaker, Minder Chen, and Titus D M Purdin. Systems development in information systems research. *Journal of Management Information Systems*, 7(3):89–106, December 1990. URL https://doi.org/10.1109/HICSS.1990.205401.

Robin Nunkesser. Beyond Web/Native/Hybrid: A new taxonomy for mobile app development. In *2018 IEEE/ACM 5th International Conference on Mobile Software Engineering and Systems (MOBILESoft)*, pages 214–218. IEEE/ACM, May 2018. URL https://doi.org/10.1145/3197231.3197260.

OpenSignal. Android fragmentation visualized. Technical report, August 2015. URL https://opensignal.com/legacy-assets/pdf/reports/2015_08_fragmentation_report.pdf.

Chris O'Sullivan. A tale of two platforms: Designing for both android and iOS. https://webdesign.tutsplus.com/articles/a-tale-of-two-platforms-designing-for-both-android-and-ios--cms-23616, April 2015. Accessed: 2020-7-15.

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, October 2012. URL http://doi.org/10.1109/ICIN.2012.6376023.

Nitish Patkar, Mohammad Ghafari, Oscar Nierstrasz, and Sofija Hotomski. Caveats in eliciting mobile app requirements. In *Proceedings of the Evaluation and Assessment in Software Engineering*, EASE '20, pages 180–189, New York, NY, USA, April 2020. ACM. URL https://doi.org/10.1145/3383219.3383238.

Ken Peffers, Tuure Tuunanen, Marcus Rothenberger, and Samir Chatterjee. A design science research methodology for information systems research. *Journal of Management Information Systems*, 24(3):45–77, December 2007. URL http://doi.org/10.2753/MIS0742-1222240302.

Joachim Perchat, Mikael Desertot, and Sylvain Lecomte. Component based framework to create mobile cross-platform applications. In *Procedia Computer Science*, volume 19, pages 1004–1011. ScienceDirect, June 2013. URL https://doi.org/10.1016/j.procs.2013.06.140.

Andreas Pirchner. gsvi. https://pypi.org/project/gsvi/, January 2020. Accessed: 2020-4-29.

Arno Puder, Nikolai Tillmann, and Michał Moskal. Exposing native device APIs to web apps. In *Proceedings of the 1st International Conference on Mobile Software Engineering and Systems, MOBILESOFT '14*, pages 18–26. ACM, June 2014. URL http://doi.org/10.1145/2593902.2593908.

Peixin Que, Xiao Guo, and Maokun Zhu. A comprehensive comparison between hybrid and native app paradigms. *Proceedings - 2016 8th International Conference on Computational Intelligence and Communication Networks, CICN 2016*, pages 611–614, December 2016. URL https://doi.org/10.1109/CICN.2016.125.

Héctor Ramos and Bruno Lemos. React native performance, December 2017. URL https://facebook.github.io/react-native/docs/performance.html. Accessed: 2018-1-23.

Vijay Janapa Reddi, Hongil Yoon, and Allan Knies. Two billion devices and counting. *IEEE Micro*, 38(1):6–21, January 2018. URL https://doi.org/10.1109/MM.2018.011441560.

André Ribeiro and Alberto Rodrigues da Silva. Survey on Cross-Platforms and languages for mobile apps. In *2012 Eighth International Conference on the Quality of Information and Communications Technology*, pages 255–260. IEEE, September 2012. URL http://doi.org/10.1109/QUATIC.2012.56.

António Nestor Ribeiro and Costa Rogério Araújo. An automated model based approach to mobile UI specification and development. In *Human-Computer Interaction. Theory, Design, Development and Practice*, Lecture

Notes in Computer Science, pages 523–534. Springer, Cham, July 2016. URL http://doi.org/10.1007/978-3-319-39510-4.

Christoph Rieger. Evaluating a graphical Model-Driven approach to codeless business app development. In *Proceedings of the 51st Hawaii International Conference on System Sciences*, pages 5725–5735. ScholarSpace, January 2018. URL https://doi.org/10.24251/HICSS.2018.717.

Christoph Rieger and Tim A Majchrzak. Weighted evaluation framework for Cross-Platform app development approaches. In Stanislaw Wrycza, editor, *Information Systems: Development, Research, Applications, Education*, Lecture Notes in Business Information Processing, pages 18–39. Springer International Publishing, September 2016. URL https://doi.org/10.1007/978-3-319-46642-2_2.

Christoph Rieger and Tim A Majchrzak. A taxonomy for App-Enabled devices: Mastering the mobile device jungle. In *Lecture Notes in Business Information Processing: Web Information Systems and Technologies*, volume 322, pages 202–220. Springer International Publishing, June 2018. URL http://doi.org/10.1007/978-3-319-93527-0_10.

Gregorio Robles. Replicating MSR: A study of the potential replicability of papers published in the mining software repositories proceedings. In *2010 7th IEEE Working Conference on Mining Software Repositories (MSR 2010)*, pages 171–180. IEEE, May 2010. URL https://doi.org/10.1109/MSR.2010.5463348.

Ira Sager. Before IPhone and android came simon, the first smartphone.

https://www.bloomberg.com/news/articles/2012-06-29/before-iphone-and-android-came-simon-the-first-smartphone, June 2012. Accessed: 2020-5-29.

Julian Schütte, Rafael Fedler, and Dennis Titze. ConDroid: Targeted dynamic analysis of android applications. In *2015 IEEE 29th International Conference on Advanced Information Networking and Applications*, pages 571–578. IEEE, March 2015. URL https://doi.org/10.1109/AINA.2015.238.

Kewal Shah, Harsh Sinha, and Payal Mishra. Analysis of Cross-Platform mobile app development tools. In *2019 IEEE 5th International Conference for Convergence in Technology (I2CT)*, pages 1–7. IEEE, March 2019. URL https://doi.org/10.1109/I2CT45611.2019.9033872.

Herbert A Simon. The science of design: Creating the artificial. In *The Sciences of the Artificial*, pages 111–138. MITP, 1996. ISBN 9780262257008.

Ashok Kumar Singh. *Science & Technology For Civil Service Examination*. Tata McGraw-Hill Education, July 2007. ISBN 9789352605705.

Garima Singh. Android app performance optimization, 2017. URL https://medium.com/mindorks/android-app-performance-optimization-cdccb422e38e. Accessed: 2019-5-21.

Bartosz Skuza, Agnieszka Mroczkowska, and Damian Włodarczyk. Flutter vs react native – what to choose in 2019? https://www.thedroidsonroids.com/blog/flutter-vs-react-native-what-to-choose-in-2019, September 2019. Accessed: 2019-9-27.

Pavel Smutný. Mobile development tools and cross-platform solutions. In *Proceedings of the 13th International Carpathian Control Conference (ICCC)*, pages 653–656. IEEE, May 2012. URL http://doi.org/10.1109/CarpathianCC.2012.6228727.

Stack Overflow. What is resident and dirty memory of iOS?, October 2013. URL https://stackoverflow.com/a/19238896/1028722. Accessed: 2017-11-17.

Stack Overflow. Developer survey 2017. https://insights.stackoverflow.com/survey/2017, 2017. Accessed: 2021-3-13.

Stack Overflow. Developer survey 2018. https://insights.stackoverflow.com/survey/2018, 2018. Accessed: 2021-3-13.

StatCounter. Mobile operating system market share worldwide. https://gs.statcounter.com/os-market-share/mobile/worldwide/2019, April 2020. Accessed: 2020-5-18.

StatCounter Global Stats. Mobile & tablet android version market share worldwide (june 2019 - june 2020). https://gs.statcounter.com/android-version-market-share/mobile-tablet/worldwide, June 2020. Accessed: 2020-7-15.

Statista. Smartphone users worldwide 2020. https://www.statista.com/statistics/330695/number-of-smartphone-users-worldwide/, September 2019. Accessed: 2020-5-18.

Statista. App stores: number of apps in leading app stores 2019. https:

//www.statista.com/statistics/276623/number-of-apps-available-in-leading-app-stores/, January 2020. Accessed: 2020-2-26.

Statista. Global app economy size 2021. https://www.statista.com/statistics/267209/global-app-economy/, November 2021. Accessed: 2021-3-11.

Ole Tange. *Gnu Parallel 2018*. Zenodo, 2018. URL http://doi.org/10.5281/zenodo.1146014.

Telerik. NativeScript, n.d. URL https://www.nativescript.org/. Accessed: 2015-10-21.

Sam Tolomei. Shrinking APKs, growing installs. https://medium.com/googleplaydev/shrinking-apks-growing-installs-5d3fcba23ce2, November 2017. Accessed: 2020-9-1.

Marcus Trapp and René Yasmin. Addressing animated transitions already in mobile app storyboards. In *Design, User Experience, and Usability. Web, Mobile, and Product Design*, Lecture Notes in Computer Science, pages 723–732. Springer, Berlin, Heidelberg, July 2013. URL http://doi.org/10.1007/978-3-642-39253-5_81.

Eric Umuhoza and Marco Brambilla. Model driven development approaches for mobile applications: A survey. In *Mobile Web and Intelligent Information Systems*, Lecture Notes in Computer Science, pages 93–107. Springer, Cham, August 2016. URL https://doi.org/10.1007/978-3-319-44215-0_8.

Eric Umuhoza, Hamza Ed-douibi, Marco Brambilla, Jordi Cabot, and Aldo Bongio. Automatic code generation for cross-platform, multi-device mobile apps: Some reflections from an industrial experience. In *Proceedings of the 3rd International Workshop on Mobile Development Lifecycle*, MobileDeLi 2015, pages 37–44, New York, NY, USA, October 2015. ACM. URL http://doi.org/10.1145/2846661.2846666.

Unity. Unity - manual: iOS hardware guide, n.d. URL https://docs.unity3d.com/Manual/iphone-Hardware.html. Accessed: 2017-11-17.

Usability.gov. System usability scale (SUS), 6 September 2013. URL https://www.usability.gov/how-to-and-tools/methods/system-usability-scale.html. Accessed: 2017-8-17.

Muhammad Usman, Muhammad Zohaib Iqbal, and Muhammad Uzair Khan. A product-line model-driven engineering approach for generating feature-based mobile applications. *Journal of Systems and Software*, 123:1–32, January 2017. URL http://doi.org/10.1016/j.jss.2016.09.049.

Vijay K Vaishnavi and William Kuechler. *Design Science Research Methods and Patterns: Innovating Information and Communication Technology, 2nd Edition.* CRC Press, Florida, 6 May 2015. ISBN 9781498715256. URL https://www.crcpress.com/Design-Science-Research-Methods-and-Patterns-Innovating-Information-and/Vaishnavi-Vaishnavi-Kuechler/9781498715256.

Nicolas Viennot, Edward Garcia, and Jason Nieh. A measurement study of google play. In *The 2014 ACM international conference on Measurement* *and modeling of computer systems*, volume 42, pages 221–233, New York, NY, USA, June 2014. ACM. URL http://doi.org/10.1145/2637364.2592003.

Paul Vorbach. Cordova download statistics. https://npm-stat.com/charts.html?package=cordova&from=2014-01-01&to=2019-09-12, September 2019. Accessed: 2019-9-13.

W3C. High resolution time level 2, 2018. URL https://www.w3.org/TR/hr-time-2/. Accessed: 2019-5-21.

Haoyu Wang, Hao Li, and Yao Guo. Understanding the evolution of mobile app ecosystems: A longitudinal measurement study of google play. In *WWW '19: The World Wide Web Conference*, WWW '19, pages 1988–1999, New York, NY, USA, May 2019. ACM. URL https://doi.org/10.1145/3308558.3313611.

Lili Wei, Yepang Liu, and Shing-Chi Cheung. Taming android fragmentation: characterizing and detecting compatibility issues for android apps. In *Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering*, ASE 2016, pages 226–237, New York, NY, USA, August 2016. Association for Computing Machinery. URL https://doi.org/10.1145/2970276.2970312.

Mark Weiser. The computer for the 21st century. *Scientific American*, 265 (3):94–104, September 1991. URL https://doi.org/10.1145/329124.329126.

Roel J Wieringa. *Design Science Methodology for Information Systems and Software Engineering*. Springer, Berlin, Heidelberg, 2014. ISBN 9783662438381. URL https://doi.org/10.1007/978-3-662-43839-8.

Michiel Willocx, Jan Vossaert, and Vincent Naessens. A quantitative assessment of performance in mobile app development tools. In *Mobile Services (MS), 2015 IEEE International Conference on*, pages 454–461. IEEE, June 2015. URL http://doi.org/10.1109/MobServ.2015.68.

Michiel Willocx, Jan Vossaert, and Vincent Naessens. Comparing performance parameters of mobile app development strategies. In *2016 IEEE/ACM International Conference on Mobile Software Engineering and Systems (MOBILESoft '16)*, pages 38–47. IEEE, May 2016. URL http://doi.org/10.1109/MobileSoft.2016.028.

Xamarin. How does xamarin work?, March 2017. URL https://developer.xamarin.com/guides/cross-platform/getting_started/introduction_to_mobile_development/#How_Does_Xamarin_Work. Accessed: 2018-2-13.

Xamarin. Mobile application development to build apps in c#, n.d. URL https://www.xamarin.com/platform. Accessed: 2017-10-20.

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*, BCI '13, pages 213–220. ACM, September 2013. URL https://doi.org/10.1145/2490257.2490292.

Yi Zeng, Jinfu Chen, Weiyi Shang, and Tse-Hsun (peter) Chen. Studying the characteristics of logging practices in mobile apps: a case study on F-Droid. *Empirical Software Engineering*, 24(6):3394–3434, December 2019. URL https://doi.org/10.1007/s10664-019-09687-9.

Appendices

Appendix A

Retrieving Google Play Store Metadata

Appendix type: Computer Program

Purpose: Retrieve Google Play Store metadata for apps based on their package name (`pkg_name`). The API allows for a list of 100 `pkg_name`s to be retrieved simultaneously. The list of `pkg_name`s are exported from a MySQL table into a `.txt` file, which is then consumed by the script below.

Language and Environment: JavaScript (TypeScript), Node.js

Dependencies: Lodash, Axios, Async, TypeORM, Reflect-Metadata

Retrieved metadata properties: Play Store availability, author, category, changelog, content rating, date published, date published ISO, description, editors choice, file size, icon, name, number of downloads, operating systems, physical address, package name, price, store category, support email, support URL, top developer, version name, screenshots, rating.

```
[import statements removed for brevity]
async function run() {
  try {
      const connection: Connection = await createConnection
         ↪ ({...});
      // Read all pkg_names from file
      const filelistPath = ('<PATH>');
      fs.readFile(filelistPath, async function (err, data) {
        if (err) throw err;
        const pkgnamesArray = data.toString().split("\n");
        // Split pkg_names array into sub-arrays of 100
           ↪ elements
        const subPkgnamesArray = _.chunk(pkgnamesArray, 100);
        // For each sub-array, create a POST request containing
           ↪  the 100 pkg_names
        asyncModule.eachLimit(subPkgnamesArray, 1, async
           ↪ function (listOf100PackageNames, callback) {
           const response = await axios({
               "method": "POST",
               "url": "https://gplaystore.p.rapidapi.com/
                  ↪ applicationDetails",
               "headers": {
                 "content-type": "application/json",
                 "x-rapidapi-host": "gplaystore.p.rapidapi.com
                     ↪ ",
                 "x-rapidapi-key": "<API_KEY>",
                 "accept": "application/json"
             }, "data": {
                 "lang": "en",
                 "ids": listOf100PackageNames
             }
           });
           // Extract response data and map each entity (app
               ↪ metadata) to Marketplace entity
           let marketplaceData = [];
           for (let app in response.data) {
               if (response.data[app] === "Sorry, I cannot find
                  ↪  that!") {
                 let entity = new Marketplace();
                 entity.pkg_name = app;
                 entity.scraped_timestamp = new Date();
                 entity.gplay_available = false;
                 marketplaceData.push(entity);
               } else {
                 let metadata = response.data[app] as
                    ↪ Marketplace;
                 let entity = new Marketplace();
                 entity.scraped_timestamp = new Date();
                 entity.gplay_available = true;
                 [Mapping of metadata properties to ORM entity
                    ↪ model
                     removed for brevity]
                 marketplaceData.push(entity);
               }
           }
           // run createQueryBuilder
           const insertResults = await connection.
               ↪ createQueryBuilder().insert().into(
               ↪ Marketplace).values(marketplaceData).execute
               ↪ ();
           callback();
       }, function (err) { ... Error handling ... });
    });
  } catch (e) { ... Error handling ... }
}
```

Appendix B

Replication Package for Experiment “Bridge Performance”

Appendix type: Computer Programs / Mobile Apps

Purpose: Replication package for software (source code and binaries) for Chapter 4.

Language and Environment: Android, Flutter, Ionic, MAML/MD^2, NativeScript, React Native.

Source code and binaries: `https://github.com/mobiletechlab/EMSE-D-19-00180-replication-package`

Appendix C

Replication Package for Experiment "Animation Performance"

Appendix type: Computer Programs / Mobile Apps

Purpose: Replication package for software (source code and binaries) for Chapter 5.

Language and Environment: Android, iOS, Ionic, Xamarin, React Native.

Source code and binaries: https://github.com/andreasbhansen/sensors_journal_animations_performance

Appendix D

Replication Package for Experiment "Presence of Frameworks"

Appendix type: Computer Program

Purpose: Replication package for software (source code and binaries) for Chapter 6.

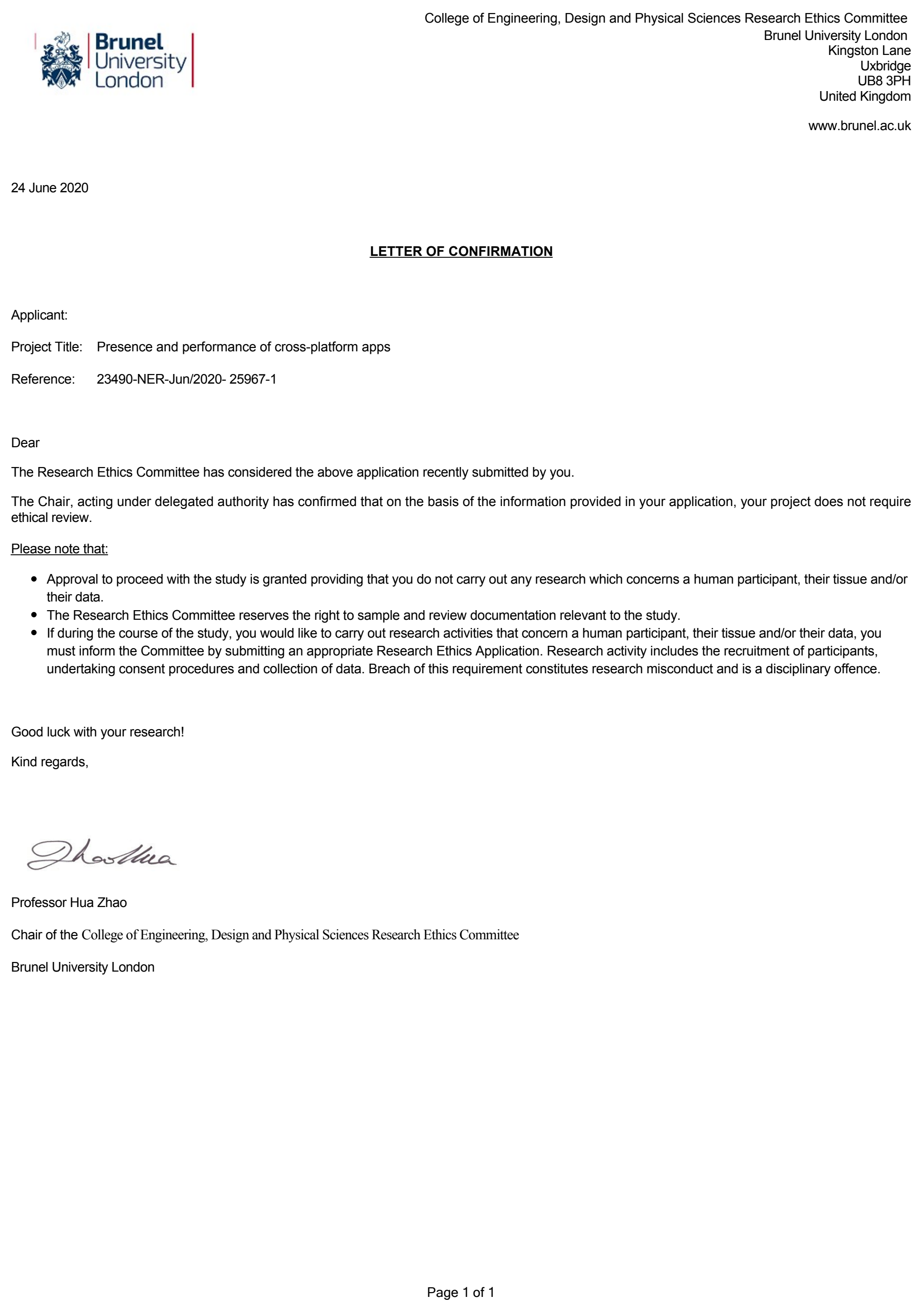
Language and Environment: JavaScript (TypeScript), Node.js

Source code and binaries: https://github.com/andreasbhansen/phd-thesis-contributions/tree/master/apk-framework-identifier-software

Appendix E

Brunel University Ethical Approval

Appendix type: Document



College of Engineering, Design and Physical Sciences Research Ethics Committee
Brunel University London
Kingston Lane
Uxbridge
UB8 3PH
United Kingdom

www.brunel.ac.uk

24 June 2020

LETTER OF CONFIRMATION

Applicant:

Project Title: Presence and performance of cross-platform apps

Reference: 23490-NER-Jun/2020- 25967-1

Dear

The Research Ethics Committee has considered the above application recently submitted by you.

The Chair, acting under delegated authority has confirmed that on the basis of the information provided in your application, your project does not require ethical review.

Please note that:

- Approval to proceed with the study is granted providing that you do not carry out any research which concerns a human participant, their tissue and/or their data.
- The Research Ethics Committee reserves the right to sample and review documentation relevant to the study.
- If during the course of the study, you would like to carry out research activities that concern a human participant, their tissue and/or their data, you must inform the Committee by submitting an appropriate Research Ethics Application. Research activity includes the recruitment of participants, undertaking consent procedures and collection of data. Breach of this requirement constitutes research misconduct and is a disciplinary offence.

Good luck with your research!

Kind regards,

[illegible]

Professor Hua Zhao

Chair of the College of Engineering, Design and Physical Sciences Research Ethics Committee

Brunel University London

Page 1 of 1