Empirical Software Engineering
https://doi.org/10.1007/s10664-020-09827-6

# An empirical investigation of performance overhead in cross-platform mobile development frameworks

**Andreas Biørn-Hansen[1,2] · Christoph Rieger[3] · Tor-Morten Grønli[1] · Tim A. Majchrzak[4] · Gheorghita Ghinea[1,2]**

Published online: 09 June 2020
© The Author(s) 2020

## Abstract

The heterogeneity of the leading mobile platforms in terms of user interfaces, user experience, programming language, and ecosystem have made cross-platform development frameworks popular. These aid the creation of mobile applications – *apps* – that can be executed across the target platforms (typically Android and iOS) with minimal to no platform-specific code. Due to the cost- and time-saving possibilities introduced through adopting such a framework, researchers and practitioners alike have taken an interest in the underlying technologies. Examining the body of knowledge, we, nonetheless, frequently encounter discussions on the drawbacks of these frameworks, especially with regard to the performance of the apps they generate. Motivated by the ongoing discourse and a lack of empirical evidence, we scrutinised the essential piece of the cross-platform frameworks: the bridge enabling cross-platform code to communicate with the underlying operating system and device hardware APIs. The study we present in the article benchmarks and measures the performance of this bridge to reveal its associated overhead in Android apps. The development of the artifacts for this experiment was conducted using five cross-platform development frameworks to generate Android apps, in addition to a baseline native Android app implementation. Our results indicate that – for Android apps – the use of cross-platform frameworks for the development of mobile apps may lead to decreased performance compared to the native development approach. Nevertheless, certain cross-platform frameworks can perform equally well or even better than native on certain metrics which highlights the importance of well-defined technical requirements and specifications for deliberate selection of a cross-platform framework or overall development approach.

**Keywords** Cross-platform development · Mobile app · Performance benchmark · Development approaches

Communicated by: Meiyappan Nagappan

✉ Andreas Biørn-Hansen
andreas.biorn-hansen@kristiania.no

Extended author information available on the last page of the article.

## 1 Introduction

As a consequence of the introduction of *app-enabled* smartphones (Macedonia 2007), mobile software applications – so-called *apps* – enjoy tremendous popularity from developers and end-users alike. This can especially be noted by the phenomenal growth in user acquisition, app downloads, and sales experienced by the two leading mobile app marketplaces, *Google Play Store* and *Apple App Store* (Jansen and Bloemendal 2013). Following several years of consolidation, the duopoly of both operating systems accounts for more than 99% of smartphone sales in 2018 (Statista Inc 2018a). The two ecosystems combined generated an estimated revenue of $58.6 billion USD in 2017 alone, an increase of 30% compared to the previous year (Nelson 2018). More than 5.8 million smartphone-specific software applications are available throughout the numerous app marketplaces, as part of the estimated $6.3 trillion dollar app economy (Statista Inc 2018b). App-enablement is an ongoing process, blurring the lines between smartphones and tablets, consumer electronics, the Internet-of-things (IoT) and even cars (Rieger and Majchrzak 2018). This also corresponds to the ubiquitous availability of smartphones and the sheer amount of smartphone users. According to recent reports, 3.8 billion unique users are estimated to have access to a smartphone and further growth is expected (Statista Inc 2016). For individuals and companies to be part of the app economy, presence in the mobile app stores is required, consequently in the form of a mobile app.

Traditionally, the creation of mobile apps has been conducted on a per-platform basis, meaning that an app cannot be deployed to a platform it was not specifically developed for. This type of development is commonly referred to as the *native* development approach, because the apps are written using tools and languages specifically designed for (i.e., *native to*) each platform. A considerable and inherent consequence of this development approach is that if an app should reach a multi-platform audience, the entirety of the app must be written twice: once for Android using Android Studio and Java, Kotlin, or C++, and a second time for iOS using Xcode and Objective-C or Swift (Grønli et al. 2014). Overall, this requires knowledge about multiple programming languages, different user interface and experience design guidelines, development environments, ecosystems, and so on. The problem is far worsened by *device fragmentation*: Especially Android exists not only in many versions but with vendor-specific changes; operating system APIs evolve over time and devices come with all kinds of capabilities and differences in hardware specifications (Wei et al. 2016). There is no guarantee that an app developed in a generally acknowledged way will truly be useful on the multitude of devices that offer compatibility in theory.

Due to the costs and knowledge requirements associated with the native development approach (Heitkötter and Majchrzak 2013), numerous alternative approaches are available (El-Kassas et al. 2017). These are commonly referred to as *cross-platform mobile development*, an umbrella term covering a wide array of conceptual development approaches and technical frameworks (Rieger and Majchrzak 2019) to simplify the creation of apps (Biørn-Hansen et al. 2018). Typically, a single codebase can be used to specify or *generate* apps deployable across several platforms with little to no platform-specific modifications, although the level of codesharing across platform differs between frameworks and approaches. However, while cross-platform development frameworks can aid in the development of apps executable across multiple platforms, certain case studies show that companies and industry practitioners leverage cross-platform frameworks for the development of single-platform apps. One such recent case study describe the online art platform Artsy making use of Facebook's React Native framework to write mobile apps using JavaScript, as that is where their internal technical competency was strongest (Therox 2019).

An even more recent case study from the online travel platform Townske describe the same situation: single-platform development using a cross-platform framework, with developer experience and internal JavaScript knowledge being core reasons to choose this over the native approach (McPherson 2019).

Thus, developing mobile apps using cross-platform frameworks does not inherently mean that app availability on multiple platforms is the ultimate goal – it could just as well be other factors such as technical in-house competency. There are also numerous studies exploring cross-platform development approaches and technologies from a single-platform perspective, such as performance testing non-functional requirements on Android by Corbalán et al. (2019), investigating cross-platform bridge security on Android by Bai et al. (2019), and impact on energy consumption in Progressive Web Apps on Android by Malavolta et al. (2017). For the study at hand, we make use of cross-platform frameworks for single-platform development; each codebase is built for Android, then the benchmarking is performed on an array of Android phones. We further elaborate on this in the study design, and suggest conducting a similar study also for iOS as future work. Whereas apps generated using such approaches are still not as commonly encountered in the app stores as native apps (Viennot et al. 2014), a clear industry interest in cross-platform technologies has been noted through the introduction of frameworks by leading technology companies, including Facebook (React Native), Google (Flutter), and Progress (NativeScript).

Both in academia and industry, the performance of apps developed using cross-platform frameworks is frequently discussed. Some studies indicate an inherent performance loss in such apps, although end users may not negatively experience this in everyday usage (Angulo and Ferre 2014a). 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, whether an overhead of app developed in a cross-platform fashion is an inherent fate is not clear: Using frameworks that generate native apps might yield code that outperforms hand-written code due to optimization; interpreted apps could undergo runtime optimization that leads to better performance than apps optimized at compile-time.

We have set out to substantiate the debate and our main motivation is the frequent encounter of claims regarding the efficiency of cross-platform technologies (e.g. Latif et al. 2016a; Ahti et al. 2016; Ribeiro and da Silva 2012; Delía et al. 2017). It often is argued that a performance overhead is introduced by bridge components between framework and native device access (Latif et al. 2017). Result from a recent study exploring the industry perspective of cross-platform mobile development, indicate that the loss of performance when compared to native apps is in fact the topmost perceived challenge of cross-platform apps (Biørn-Hansen et al. 2019). The advice is then against employing cross-platform frameworks in development projects. However, from our survey of the related literature, this performance parameter has yet to be measured and empirically evaluated, leaving an interesting gap in the body of knowledge.

Throughout our study, we investigate five technologies for cross-platform mobile app development, along with the native development approach for comparison and analysis purposes. We measure elapsed time from execution of a set of typical tasks, to the return of their results. Examples of such tasks include the programmatic retrieval of a file from the device's file system, the querying for device GPS coordinates, or listening for accelerometer sensor data.

We thus investigate technologies which developers can leverage in the efficient design and development of mobile applications across multiple platforms, effectively allowing for participation in these vast ecosystems without re-developing the same application multiple times from scratch. More specifically, we measure the performance

overhead introduced by multiple app development tools and frameworks following different development approaches which we contrast with native app development. Our research question is as follows:

**RQ:** To what degree do cross-platform mobile development frameworks impose additional performance-related overhead when compared to native mobile development?

Our main contribution from this study 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 feature and to empirically assess the performance of cross-platform app development to this extent. We have developed a total of six artifacts, and used profiling tools to measure the performance of these apps that were generated using six technical development frameworks. Unlike the majority of studies in which performance of cross-platform development frameworks is investigated, we have developed artifacts using a wide array of technologies, including frameworks of the *Model-Driven* development approach, *Hybrid* approach, *Interpreted* approach, *Cross-compiled* approach, and the *Native* approach, as further elaborated in Section 3.1. This includes recently published frameworks which claim to combine interpreted business logic with native user interface (UI) components and have not yet been studied thoroughly by previous literature. The broad spectrum of approaches is in line with our aim to focus on the validity and generalization of results beyond individual implementations through the inclusion of a wide array of smartphone devices, frameworks, and measurable features.

Besides the core contribution, our article also describes our replicable method, which can serve as a blueprint for further performance studies. Moreover, we contextualize and discuss our results to enrich the body of theory and share insights of performance comparisons for cross-platform frameworks.

The remainder of this article is structured as follows. In Section 2, we highlight and discuss related work in the context of our investigation and review the current state of literature on cross-platform framework performance. Section 3 presents the research method and design employed to conduct the study. Our findings are then presented in Section 4, before we discuss the findings with respect to related work and general research on the subject in Section 5, together with thoughts on limitations and directions for research. A conclusion of our work is then presented in Section 6.

## 2 Background and Related Work

Measuring performance and the overhead associated with the use of cross-platform mobile development frameworks has been the focus of prior work. In this section, we first draw the background by giving an overview about different cross-platform app development approaches. Subsequently, we present related work on performance measurement based on an extensive literature review.

## 2.1 Overview of Cross-Platform Development Approaches

*Native apps* are developed individually for each targeted platform using vendor-provided software development kits (SDK). The user has a limited choice of supported programming languages, which differ per target platform. For instance, Java, Kotlin, and C++ are supported by Android, and Objective-C and Swift by iOS. The advantage of having full control over the platform APIs serves as the baseline against the development of apps using different cross-platform techniques.

In general, cross-platform development uses a single code base that can be executed on multiple platforms. Platforms in this sense typically refer to different operating systems provided by software or hardware vendors, e.g., Android, or iOS. In addition, device fragmentation might cause different versions of the same underlying operating system to be considered as distinct platforms. For example, Android is often tailored by hardware vendors to specific devices or substantial changes to user interfaces (e.g., Android material design) and technical APIs can occur when platforms evolve over time (Li et al. 2018; Wei et al. 2016; Scalabrino et al. 2019). According to classifications by El-Kassas et al. (2017) and Heitkötter et al. (2013), several approaches can be distinguished to achieve this aim. We use the remainder of this section to introduce each category from Fig. 1 in detail. Beware that although this classification helps to distinguish the main characteristics, some frameworks merge different approaches (e.g., ICPMD El-Kassas et al. 2016) or investigate completely different approaches such as by Tang et al. (2011) for thin-client apps based on a cloud infrastructure. Furthermore, similar functionality might be provided using very different techniques.

**(Progressive) Web Apps:** A mobile web app is essentially a web application developed using web technologies such as HTML, CSS, and JavaScript. It is optimized for smartphone (and tablet) screen resolutions. Consequently, the app cannot be installed on the device but is executed within the respective platform browser. With the rising standardization and support of various APIs by mobile browser environments in the past years, it is possible to access device features such as location and data storage. To compensate for the – typically undesired (Heitkötter et al. 2013b) – look and feel of a web site, Progressive Web Apps (PWA) were recently introduced by Google. PWAs improve traditional web apps with so-called service workers (to allow for running code in a background thread), a web app manifest (to provide metadata), off-line capabilities, and an

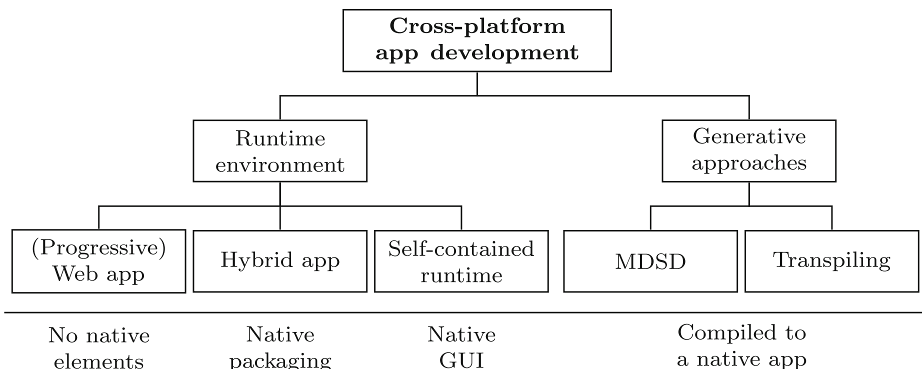


Fig. 1 Categorization of Cross-Platform Approaches adapted from Majchrzak et al. (2015)

installation-like user experience. This has evidently led to possibilities not previously available for web apps, with Progressive Web Apps performing on a par with regular native apps (Archibald 2016; Biørn-Hansen et al. 2018). While a PWA has access to device and platform privileges beyond what is typical for a web app, they are still limited in terms of feature access – a PWA cannot access device or platform features not exposed through the web browser it runs within.

**Hybrid apps:** The hybrid approach has allowed primarily web developers to develop mobile apps using the same set of knowledge they would use for the development of web sites. Cordova, the open-source core of the framework which became known as PhoneGap (now representing a commercial branch with additional features), was an early representative of this approach and enabled the packaging of HTML, CSS, and JavaScript files into an installable app. These files are subsequently rendered using a WebView – an embeddable browser window which hides its typical controls, e.g., address bar, bookmarks and settings. Standardized JavaScript APIs can be used – if provided by the respective browser engine – to access device-specific functionality similar to web apps. Additionally, functionality including contact lists, Bluetooth, GPS, and network connectivity is provided via JavaScript APIs by the framework, each of which acts as a Foreign Function Interface (FFIs) between the WebView component and the underlying native code (e.g., Java or Objective-C). A hybrid app can be downloaded from the app stores, installed on the phone, and used offline, equivalently to a native app. Thus, the combination of access to device and platform functionality, ease of user interface implementation through widely known web languages, and native behaviour has rendered the hybrid approach popular amongst both practitioners and researchers. Especially prominent proponents of this approach include Ionic, Onsen UI, Quasar Framework, Cordova / PhoneGap, and Framework7. A possible drawback of this approach, which calls for additional research and scrutiny, is the HTML-based user interfaces, and how they might behave differently from native user interface elements even if styled to adhere to platform design guidelines.

**Runtime-based and Interpreted apps:** In contrast to the hybrid approach, which reuses the device's browser engine through a WebView, apps built with this approach ship with a self-contained runtime component (Corbalan et al. 2018). This approach is typically referred to as interpreted approach, web-native approach, runtime-based approach, or JavaScript-to-Native for the JavaScript-based implementations. The framework vendor needs to develop the runtime layer for all targeted platforms and app developers can then use a common API to access the underlying functionality. Typically, the application code is written using a programming language such as JavaScript (e.g., in React Native and NativeScript), C# (Xamarin), or custom markup (e.g., Qt). Also, instead of providing access to native functionality through a Cordova-controlled WebView component, frameworks of this approach typically expose proprietary plugin-based bridging systems that allow for invocation of foreign function interfaces in native code. React Native and NativeScript are exemplary frameworks backed by companies such as Facebook and Progress (formerly Telerik). They use a combination of natively rendered user interfaces in combination with a runtime for the JavaScript-based business logic. This is possible through the use of on-device language interpreters, e.g., JavaScriptCore and V8, which interpret markup language and returns platform-specific interface components. One drawback of this approach relates to the fragmented space containing numerous frameworks and tools, each with their own underlying plugin architecture. Consequently, a plugin developed for React Native will not work in NativeScript out of the box, and vice versa. Thus, in situations where custom plugins are developed as part of a project,

changing out the cross-platform framework at a later stage will inherently mean rewriting not only user interfaces and business logic, but also plugins and similar custom native bridging infrastructure.

**Model-driven software development:** The model-driven paradigm has been used for many years in software engineering for the purpose of managing variability. It focuses on the model as abstract representation of (possibly a part of) a system from which actual software artifact is derived (Stahl and Völter 2006). In the mobile computing context, this approach allows for the development of cross-platform apps using an higher level of abstraction than source code, often-so through the use of textual or graphical domain-specific languages (DSLs) or general-purpose modelling notations such as UML. Subsequently, code generators (one per target platform) transform the platform-agnostic model into platform-specific source code, which can then be compiled and built to each mobile platform supported by the framework. The resulting apps can therefore exploit the full potential of the platform as they are – ideally – indistinguishable from native apps. Commercial frameworks include WebRatio Mobile, BiznessApps, and Bubble (WebRatio Srl 2015; Bizness Apps 2019; Bubble Group 2019), whereas in academia the focus on domain-specific frameworks is more prevalent, e.g., $MD_2$ (Heitkötter and Majchrzak 2013), MAML (Rieger and Kuchen 2018), and applause (applause 2015). While there are numerous frameworks from both industry and academia, one drawback of this approach is the infrequent encounter of the model-driven development approach in practitioners' outlets, which are usually more concerned with the hybrid and interpreted development approaches.

**Compilation-based:** Compilation-based approaches (so-called cross- or trans-compilers) aim for reusing a native application by mapping the input application to a target representation. This can happen on the level of bytecode or the high-level programming code. Because of the complexity on a low level of abstraction as well as differences between the respective platforms, compilation-based approaches typically focus on specific aspects of an application such as the business logic and need manual additions to replicate the full app functionality. Examples include XMLVM (Antebi et al. 2012), J2ObjC (Google LLC 2019c) and Google's Flutter framework. Of these, Flutter is the most recent addition to the compilation-based approach to the best of our knowledge. The main differentiator between Flutter and the interpreted approach is that it does not render native user interface components. Instead, Flutter leaves all rendering to the Skia Graphics Engine, which is able to re-create the look and feel of native user interfaces through a Skia Canvas. When building in debug mode, a Flutter app additionally contains the Dart VM needed for enhanced developer experience, including functionality such as hot reload. Business logic, i.e., Dart code, along with the Flutter SDK are ahead-of-time compiled to native (ARM/x86) libraries, avoiding the use of interpreters (Flutter Developers 2019). A drawback of the compilation-based approach seen from a developer's perspective, relates to the ease of pushing updates to end-users without going through the Google Play Store or Apple App Store. While both the hybrid and interpreted approaches can use tools such as Microsoft CodePush to push JavaScript-based bundles as updates circumventing the app marketplaces, the compilation-based approach, using Flutter as an example, cannot (Seidel 2018).

## 2.2 Performance Evaluations of Cross-Platform Frameworks

Previous research has targeted both hardware and software perspectives, looking at the feasibility of employing such frameworks, especially when compared to the performance baseline of traditional native apps.

Searching Scopus in July 2019 resulted in 516 hits with the following – intentionally broad – search query for cross-platform framework evaluations:

```
TITLE-ABS-KEY(("cross platform" OR cross-platform OR
"multi platform" OR multi-platform) AND (mobile OR app
OR application) AND (framework OR library OR approach*)
AND (performance OR evaluate* OR assess*)).
```

In addition, a forward search on the papers by Heitkötter et al. (2012, 2013) was performed. These studies provided an early systematic selection of app development frameworks for smartphones and have been used by many authors as a basis for further research on apps.[1] Abstracts were manually filtered and the paper contents screened to target actual framework comparisons in contrast to purely abstract considerations regarding the development approaches. Also, at least two implementations should be compared; for instance considering only the sensor power consumption on iOS (Katevas et al. 2016) or focusing on the battery component but only for Android (Abousaleh et al. 2014) allows for targeted optimizations but was not further considered for the scope of this study. Furthermore, studies included in the selection should contain evaluations of functional characteristics in contrast to the performance of UI rendering or visualizations which are usually subjectively assessed by users (e.g., Kromer et al. 2016; Angulo and Ferre 2014b). Consequently, a list of 18 articles is considered closely related work as depicted in Table 1 (p. 10).

Notable details of the identified literature are presented in the following. Ciman and Gaggi's (2017) comprehensive evaluation of energy consumption for multiple cross-platform frameworks reports major differences in hardware performance between the evaluated frameworks. They also observed differences between programming languages, e.g., how their C++ based artifact was outperformed by an artifact designed using JavaScript, although both artifacts were built using the underlying MoSync development framework that supports both programming languages. In a recent study related to the work of Ciman and Gaggi (2017), 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 (2017). Whereas Ciman and Gaggi (2017) measure energy consumption during the execution of device features such as accelerometer and GPS, Corbalan et al. (2018) measured energy consumption during three usage scenarios, being intensive processing, audio playback, and video playback. Their findings showed that whereas Apache Cordova handled processing and audio playback well, it performed poorly in their video playback measurement. Corona on the other hand performed well at video playback, but had issues with intensive processing.

In contrast to these studies, our focus lies on measuring the performance of native-side access to platform features through invoking and having data returned from Foreign Function Interfaces (FFIs) or framework- and approach-specific equivalents. The invocation of FFIs and the measurement of bridge performance is also mentioned in a study by Biørn-Hansen and Ghinea (2018), although this part of their study is stated as preliminary and in need of further work and verification. Thus, our current study greatly extends on their findings, as we now focus solely on the performance of bridges and their equivalents across multiple frameworks.

The results presented by Ciman and Gaggi (2017) share similarities with those reported in related performance studies, e.g., by Willocx et al. (2015, 2016) focusing on the hardware

[1] They together have 268 citations according to Google Scholar as of 2019-07-15.

**Table 1** Literature on cross-platform App development performance evaluations

| Paper & Year | Evaluated tools | Evaluated App features and performance metrics |
|---|---|---|
| Biørn-Hansen and Ghinea (2018) | Ionic, React Native | Duration of file system access |
| Corbalan et al. (2018) | Apache Cordova, Corona, Native app, NativeScript, Titanium, Xamarin | Energy consumption, CPU utilization, and duration of calculations and Audio/Video playback |
| Delia et al. (2018) | Apache Cordova, Corona, Native app, NativeScript, Titanium, Web app, Xamarin | Execution time of calculations |
| Ferreira et al. (2018) | Native App, PhoneGap, Sensa Touch, Titanium | Camera, GPS, and calculations; duration of scenarios (including a sequence of activities) and RAM utilization |
| Jia et al. (2018) | Apache Cordova, Native App, Titanium, Xamarin | Building time, rendering time, UI response time for screen content |
| Biørn-Hansen et al. (2017) | Ionic, PWA, React Native | No specific platform features; size of installation, launch time until first activity, and launch time until UI rendering |
| Que et al. (2017) | Apache Cordova, Native app | Accelerator, camera, GPS, media player; measuring of installation and startup time, CPU and RAM utilization, battery temperature, and network flow |
| Ahti et al. (2016) | Native app, PhoneGap | File size, starting time, and RAM utilization for a sample app with network access |
| Mercado et al. (2016) | Multiple frameworks | Measuring the complaint density in reviews of 50 actual apps in the app store |
| Willocx et al. (2016) | Adobe Air, Famo.us, Intel App Framework, Ionic, jQueryMobile, Mgwt, Native app, NeoMAD, PhoneGap, Sencha Touch, Titanium, Xamarin | GPS, network access; Measuring launch time, CPU and RAM utilization, package and file size |
| Ciman and Gaggi (2015) | PhoneGap, Titanium | Measuring energy consumption of accelerometer, camera, compass, GPS, and microphone |
| Dhillon and Mahmoud (2015) | Adobe Air, Native app, PhoneGap, Titanium, WebWorks | Execution times of contact list access, microphone, and calculations |
| Willocx et al. (2015) | Native app, PhoneGap, Xamarin | GPS, network access; Measuring launch time, CPU and RAM utilization, package and file size |
| Ciman and Gaggi (2014) | PhoneGap, Titanium | Measuring energy consumption of accelerometer, camera, compass, GPS, and microphone |
| Perchat et al. (2014) | COMMON framework, Native app | Computations, GPS, network access; measuring of execution times, installation size, RAM utilization |

**Table 1** (continued)

| Paper & Year | Evaluated tools | Evaluated App features and performance metrics |
|---|---|---|
| Rösler et al. (2014) | IBM Worklight (Cordova) with jQuery Mobile, Native app | Start-up and loading time, network stability, computations, file system throughput, RAM utilization |
| Dalmasso et al. (2013) | jQuery Mobile, PhoneGap, Sencha Touch, Titanium | CPU and RAM utilization, and power consumption of simple app with web service communication |
| Corral et al. (2012) | Native app, PhoneGap | Execution times of accelerometer, contact list, file system access (read/write), GPS, network information, sound notification, and vibrator |
| Ohrt and Turau (2012) | Flash Builder, Illumination Software Creator, LiveCode, Marmalade, MoSync, OpenPlugStudio, PhoneGap, Rhodes, Titanium | File size, launch time, and RAM utilization for a simple app skeleton |

impact imposed by a variety of frameworks and implementations. The studies presented in Table 1 evaluate a heterogeneous variety of platform characteristics. Whereas accelerometer and GPS sensor values, camera, and network access represent common 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). More exotic studies use secondary data taken from app store reviews in order to detect performance issues in actual apps (Mercado et al. 2016).

Although the individual study results cannot easily be compared due to the dependency on the sample app scenario, test devices, and potential external factors, it can be derived that individual frameworks have evolved over time. Indeed, the lack of comparability is a general limitation in the young field. For instance, access to platform features in PhoneGap/Apache Cordova was roughly 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. 2012). However, more recent studies indicate that the framework is still more resource-intensive but for example load times are "only" 40% slower than for native apps (Que et al. 2017), and sometimes it even outperforms native implementations (Delia et al. 2018).

From the identified studies it can also be seen that there is no clear *winner* among the evaluated frameworks and approaches. Generally, it can be observed that JavaScript-based frameworks and interpreted apps encounter performance penalties more frequently. Countering intuition, Ahti et al. (2016) find in their evaluation that the native implementation is slower than the PhoneGap app which might be caused by additional third-party libraries. Novel interpreted/web-native approaches claiming a near-native performance of user interfaces do not necessarily perform well with regard to device access compared to hybrid frameworks which exist for multiple years and have undergone continuous optimization. In case of the preliminary results by Biørn-Hansen and Ghinea (2018), file system access in React Native was slower by a factor of 5 compared to Ionic. Highly generalized assumptions – for instance that there is "no choice other than native for performance" (Hudli et al. 2015) – are therefore not without their caveats.

Also, using only an app skeleton without actual content (e.g., Ohrt and Turau 2012) or pure calculations without accessing platform features (e.g., Delia et al. 2018) make the

results even less comparable among the studies and hardly transferable to the development of real apps. Our study, thus, aims for a clear separation by benchmarking the considered platform features in isolation as described in Section 3. Moreover, it sets out to arrive at the current status – as evident from above considerations, the literature lacks in respect of development of cross-platform technology.

In addition, several articles touch the topic of cross-platform framework performance but rest on a qualitative level of argumentation based on the underlying approach (e.g., assuming that interpreted apps have an inferior performance compared to native apps) or discussing subjective experiences with the general performance of a sample application (Humayoun et al. 2013; Palmieri et al. 2012; Latif et al. 2016b; El-Kassas et al. 2017; Lachgar and Abdali 2017; Rieger and Majchrzak 2016). For example, Botella et al. (2016) found that application load durations are significantly worse for apps built with Sencha Touch compared to Ionic, although both are developed using web technologies. Sommer and Krusche (2013) explain that PhoneGap's limited JavaScript performance is fast enough in many cases but identify deteriorating performances for Rhodes and Sencha Touch with complex DOM operations resulting in view changes of multiple seconds. Especially when it comes to animations, the experienced performance of browser-based approaches degrades considerably (Heitkötter et al. 2012; Ciman et al. 2014). Our focus, however, lies on the quantification of the overhead by the frameworks' bridges.

The findings in the identified related work are of fundamental importance for professionals and researchers alike, as results clearly indicate that there is no framework which is superior for *all* contexts or needs. Moreover, performance particularities have not been reported uniformly let alone have systematic, broad assessments been conducted. Neither is it possible to describe *why* performance deviations occur.

Our work therefore aims to extend previous studies and especially to shed more light on the current state of platform feature access across several frameworks from very different approaches. Besides the ability to access these platform features (which has increased over the past years), the performance of the frameworks' abstraction layers is of high practical importance as it enables cross-platform development to be an alternative approach to native app development.

# 3 Research Method

In this section we present the research methodology applied to investigate the varying performance among different cross-platform frameworks. We elaborate on the framework selection, platform features to be benchmarked, the artifact design, and the process of data gathering using six Android devices. Our research has been designed with the explicit goal of *reproducibility* to overcome the incoherence of the existing literature on cross-platform development performance.

## 3.1 Technical Frameworks

From our discussion on related work (Table 1), we find that few previous studies have comprehensively included frameworks and tools from the most-encountered development approaches, namely Hybrid, Interpreted, Model-Driven Software Development, Cross-compiled, and Native. Instead, previous studies often choose to implement artifacts using frameworks belonging to only *some* of these approaches (such as work by Que et al. 2017 and Corral et al. 2012). None of the articles in Table 1 makes use of the model-driven

**Table 2** List of technologies included in the study

| 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 |

approach. This exclusion is especially striking regarding the trend of modern frameworks to generate at least the user interface for performance reasons (e.g., React Native, NativeScript). Whereas the overall choice of a cross-platform framework depends on many criteria (cf., e.g., Rieger and Majchrzak 2019), performance comparisons are a relevant factor to assess the proposition of near-native performance. In addition, particular operating system or hardware features should be evaluated in isolation in order to determine the net effects of their implementation across frameworks in contrast to an app representing a coherent scenario. In general, studying performance in the area of cross-platform development needs to be attested low maturity.

With this study, we aim to broaden the scope of what is typically encountered in similar studies, and we do so through the effort of artifact implementation. We thus include a wide range of cross-platform development frameworks and tools which have partly been considered in related work (cf. Table 1). The selection reflects a combination of industry standards and cutting-edge, popular frameworks (AppBrain 2019; Stack Exchange Inc 2019). In particular, we aimed to cover the major cross-platform approaches and selected appropriate frameworks.

Table 2 lists six technologies which have been used in the development of the artifacts. Of these, one belongs to the Native approach, i.e., it does not support cross-platform deployment. It serves as the baseline benchmark. The remaining five technologies allow for the creation of iOS and Android apps based on a common code base. They vary in terms of programming language, associated development approach, industry adoption, among other aspects.

- The *Ionic* framework is a representative of the hybrid app approach and itself based on the long-lasting Apache Cordova framework. Ongoing popularity can be seen with the project accumulating 39 000 stars on Github.
- *React Native* ($\tilde{8}$2 000 stars) and *NativeScript* ($\tilde{1}$7 000 stars) can both be categorized to the runtime-based/interpreted approach although also generating parts of the user interface.
- The cross-compiled *Flutter* framework has not yet been scrutinized by related literature. It has, however, attracted huge interest among practitioners (as can be seen from the 77 000 stars on the Github project) and reflects the current trend towards generated app components in cross-platform tools.
- *MAML / $MD_2$* stand out from this list as the frameworks originate from an academic context.[2] As already noted before, model-driven approaches have a comparatively low

[2]For full disclosure it should be noted that MAML and $MD_2$ have been co-developed by one respectively two of the authors.

adoption resulting from a lack of commercial applications. Nevertheless, the framework was chosen in order to cover a broad range of development approaches.

From the presentation of approaches in Section 2.1, the Progressive Web App approach is the only approach missing from our study. We decided against the inclusion of this approach due to limited access to device- and platform functionality, including the file system API and contacts API which we rely on extensively for our study. A Progressive Web App is also executed in a blackbox (Web browser) where the communication between JavaScript code and native functionality is abstracted and unavailable to the developer. Thus, when needing access to the underlying native code, e.g., for debugging or implementing custom code, which is a straightforward process in the other development approaches we have investigated, this would not be feasible for a Progressive Web App.

Thus, the list of technologies should provide the means for a comprehensive evaluation and discussion throughout the remainder of the article. In Table 2, we also list the compiled APK size (`MB`) for each generated app. This metric can be of utmost importance in the context of technical decision making. Considering the ”Next Billion Users”, a term referring to the increase in connected users in emerging markets (Google LLC 2019e), additional effort should be put towards building mobile software targeting end-users regardless of their socioeconomic and demographic situation. Compiled APK size is a metric with direct impact on an app's adoption across markets wherein network availability, data usage costs and available hardware are important factors.

### 3.2 Artifact Design and Implementation

The focus in our study is on the underlying capabilities of the cross-platform frameworks to provide access to device hardware and operating system features of the platform – and not the frameworks' capabilities to render (nice) user interfaces.[3] We, hence, kept the visual aesthetics of the artifacts intentionally functional to measure the objective feature performance. However, we aim for a similar representation across the different framework implementations and use separate views for each task (see next Section 3.3) which can be selected from an introductory start screen. While the number of runs for a specific benchmark can be specified, we did not make use of this feature, as we always executed only a single benchmark run before restarting the app and starting over (see Fig. 2). When pressing the `Start benchmark` button, the app will initialize 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) is printed to the screen, and manually transferred into an external data sheet. A replication package containing all source code and compiled APK files is available in the project Github repository.[4]

### 3.3 Benchmark Features and Tasks

A plethora of hardware features exist that invite for benchmarking, including sensors (accelerometer, gyroscope, compass), network connection (cellular, WiFi, Bluetooth, NFC), native events (hardware `back` button, volume keys), device information such as battery

[3]In fact, studying the visual capabilities in the light of a debate about native look&feel (Majchrzak and Heitkötter 2014; Heitkötter et al. 2013a) would be an interesting idea for an empirical research paper, ideally conducted with real users.

[4]Open source replication package: https://github.com/mobiletechlab/EMSE-D-19-00180-replication-package.

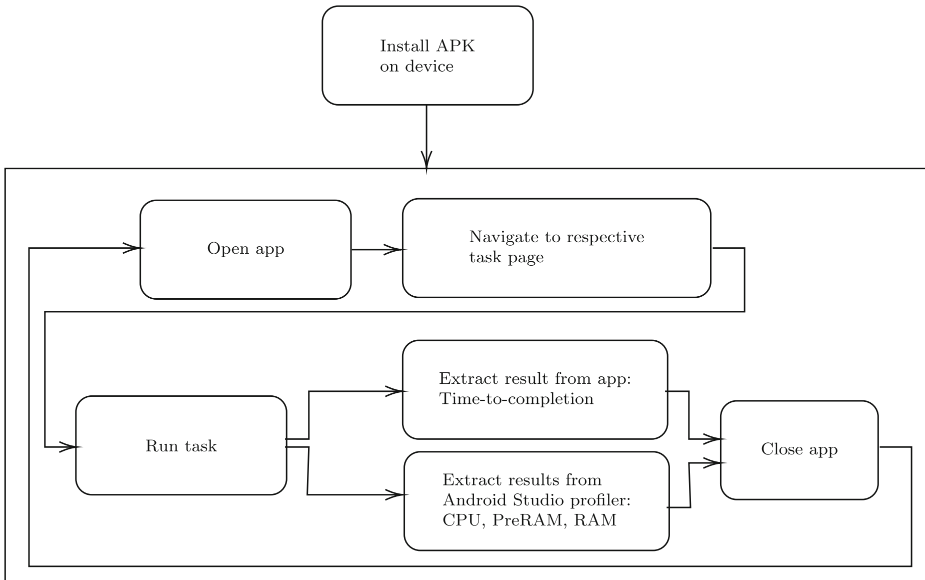


**Fig. 2** The data gathering process

status, and many more. In addition, operating system features such as storage databases, contact lists, or notifications are provided, which can be accessed by native apps and therefore also through bridge components in cross-platform frameworks.

From this extensive list of possible features we designed five benchmarks. These relate to both hardware and software capabilities. Features were selected to reflect the presumably most common use cases while at the same time being present on many devices – in contrast to specialized sensors, which only a few devices provide and few people use. Moreover, benchmarks were chosen to be executable mostly in isolation, avoiding complex multi-device set-ups that rely on external factors such as network quality beyond our control. This should help providing objective and reproducible results.

Unfortunately, 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, 8 read or modify calendar entries, 8 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.

The features implemented as tasks for benchmarking are as following:[5]

[5] Although we deem the device's camera as important use cases for smartphone hardware access, the encapsulation of functionality in several frameworks posed major difficulties with regard to automating the process of taking pictures without manual user interaction (e.g., tapping a capture button). We also found differences between the frameworks regarding how deeply integrated and configurable their camera plugins were in terms of being able to create an equally complex and testable integration between all the developed apps. Therefore, this feature could not be assessed objectively and we left it out of the present study.

- **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 utilized 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, we provide each contact object with a name and a mobile phone number. We deemed this the minimum amount of data needed to store a new contact in a real-world context. Nevertheless, there is little reason to expect additional information to be stored to have a significant impact on performance.
- **File system:** Similarly, reading files stored on the file system is evaluated. File system access is particularly required for bulk data access. We, therefore, use 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, we measure the time until the base64-encoded string is decoded in memory and ready for assignment to a view element (but excluding the actual rendering).
- **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 localization purposes. This benchmark retrieves the longitude and latitude values of the device's current location based on the vendor-recommended location retrieval mechanism.

### 3.4 Data Gathering

For this study, we gathered data on time-to-completion (`TTC`), `CPU` usage, idle-state RAM occupancy (`PreRAM`), and busy-state memory occupancy (`RAM`). All measurements were taken with 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. Within the Android Studio profiler, performance values are provided for the specific app during testing; thus, we report on the single highest consumption (peak value) observed for the given benchmark. These peak values indicate which frameworks during which benchmarks may require the most of the on-device hardware. Because more accurate trace-based inspection of method calls impacts runtime performance but provides no additional value in terms of the above metrics, this configuration is sufficient for our purposes. Specifically `CPU` and memory usage are metrics included also in previous performance studies, including Dalmasso et al. (2013) and Willocx et al. (2015). The `TTC` metric is provided in milliseconds, and reports on the duration of time between invoking a benchmark task, and having the results available. 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 available processing power consumed at peak during benchmarking. Within the Android Studio profiler, values are provided for the specific app; thus, we report on the single highest consumption observed for the given benchmark. 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 study. 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.

We conducted all performance tests on physical mobile devices rather than on emulated hardware. Again, this was a deliberate choice, facing that the effort for running, monitoring, and debugging on real devices is tremendously higher than using emulation. However, this is particularly important due to subtle differences in receiving realistic sensor input and effects of continued physical execution (Joorabchi et al. 2013) – to an 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. Additionally, we ensured that all the Android APK app installation files were built for release rather than debug mode. This was especially required for certain cross-platform frameworks, e.g., Flutter limits the performance of apps built in debug mode, consequently rendering any performance comparisons in debug mode invalid. Nevertheless, to extract information on app-specific usage and utilization of `CPU` and memory on-device, APKs built for release must include a `debuggable` property in their Gradle configuration (Google LLC 2019d). This is done to enable Android Studio's profiler tool to gather necessary profiling data for inspection.

First, we conducted the time-to-completion feature benchmark using APKs without the `debuggable` property. Secondly, we re-compiled the APKs, this time including `debuggable`, and conducted the profiling using the Android Studio profiler environment. To the best of our knowledge, 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 Fig. 2. It illustrates the extraction of results from within the app while running on device, and results from Android Studio.

Furthermore, Fig. 2 illustrate the effort put in to the data gathering process. For each loop as illustrated in the figure, only one ($n = 1$) benchmark run was executed. 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 indicate a non-trivial energy consumption overhead related to the use of automation frameworks (Cruz and Abreu 2019), we avoided automating any processes related to data gathering. This is also 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 inspect the recorded event timeline to identify the impact on the metrics caused by the execution of the benchmark. We separated 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 in two separate rounds to gather the aforementioned results using different processes: first round for `TTC`, and second round for the remaining three metricises which were gathered simultaneously. We relied on no automated services or processes for the data gathering. Every interaction with the app, every extraction of data – whether from within the app or from the Android Studio profiler – were done manually. This resulted in tens of thousands of manual interactions with the physical mobile devices and the apps deployed to them, being navigation to respective benchmark task page, executing the benchmark, closing the

app, restarting the app, and so on, for every benchmark run recorded. This was a rigorous and time-consuming process.

Benchmarking the JavaScript-based implementations (i.e., Ionic, React Native and NativeScript) leverages the `Date` API for calculating elapsed time. While there are other time units available for JavaScript, including `DOMHighResTimeStamp` (W3C 2018), we identified no other timer than `DateTime` for the Dart-based Flutter implementation (Google LLC 2019b). Thus, in an attempt to harmonize the timer units, we decided on the JavaScript `Date` API over the `DOMHighResTimeStamp`. These differences in time units and their resolutions could be seen as a technical limitation inherently affecting the obtained results (cf. Section 5). Nevertheless, the lack of a unified cross-language timer implementation ultimately led to this decision.

We focused on the preparation of devices included in this study, ensuring to the best of our abilities that the hardware on which the benchmarking tasks were executed kept to the same 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 also 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 we would 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 contents and executing previously just-in-time compiled code by the Android Runtime (Google LLC 2019a) – although these operating system level optimizations are generally beyond the control of the app developer and we aim for a realistic real-world behavior of the device.

In our research, we conducted the performance measurements on a total of six mobile devices, as further described in Table 3. With a previously identified population of $n = 24\,093$ distinct Android device models (OpenSignal 2015), the market fragmentation is too severe to conduct the benchmarks on a sample-wise representative number of consumer devices ($n = 379$). Nevertheless, the devices included in this study represent a wide range of hardware, including both budget smartphones and the state of the art, as well as devices representative for the current consumer and business hardware on the market.

### 3.5 Data Analysis

In total, $n = 16\,290$ individual data points were manually gathered for this study. 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

**Table 3** List of devices used for measuring performance

| Model | CPU (Cores) | Memory | OS |
|---|---|---|---|
| Samsung S8 | 2.3+1.7 GHz (Octa) | 4GiB LPDDR4 | Android 8.1 |
| Samsung A3 (2016) | 1.5 GHz (Quad) | 1.5GiB | Android 7.0 |
| Huawei Mate 10 Pro | 2.36+1.8 GHz (Octa) | 6GiB | Android 8.0 |
| Huawei Mate 8 | 2.3+1.8 Ghz (Octa) | 3GiB | Android 6.0 |
| LG Nexus 5X | 1.4+1.8 GHz (Hexa) | 2GiB | Android 8.1.0 |
| Sony Xperia Z5 | 1.5+2.0 GHz (Octa) | 3GiB | Android 6.0 |

(RAM). In addition, we have analyzed 3 990 data points which are the arithmetic computations of RAM subtracted from PreRAM, providing the actual memory impact of executing a given benchmark (ComputedRAM). The difference in $n$ 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 profiling was unavailable.

This raw data was then statistically analyzed in order to identify differences across the frameworks under test. We designed the native Java implementation as baseline to which all other frameworks are compared. By respecting current best practices and state-of-the-art system APIs we assume a high performance (i.e., low utilization of resources and fast execution times) for this implementation. However, frameworks might utilize highly optimized 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. To assess whether the observed variance of results is significant, we perform ANOVA tests ($\alpha = .05$) along with effect sizes using omega squared ($\omega^2$), following the interpretations provided by Kirk (1996). The effect size provides an indication of how many percent of the variance between two groups can be explained by the independent variable. While the ANOVA can provide data 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 we identify statistical significance, we follow up using Tukey post-hoc tests between each individual framework with results from the native implementation as 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, we also make use of ANOVA ($\alpha = .05$), along with Spearman's rank-order test to report on the correlation between idle-state memory usage (PreRAM) and ComputedRAM.

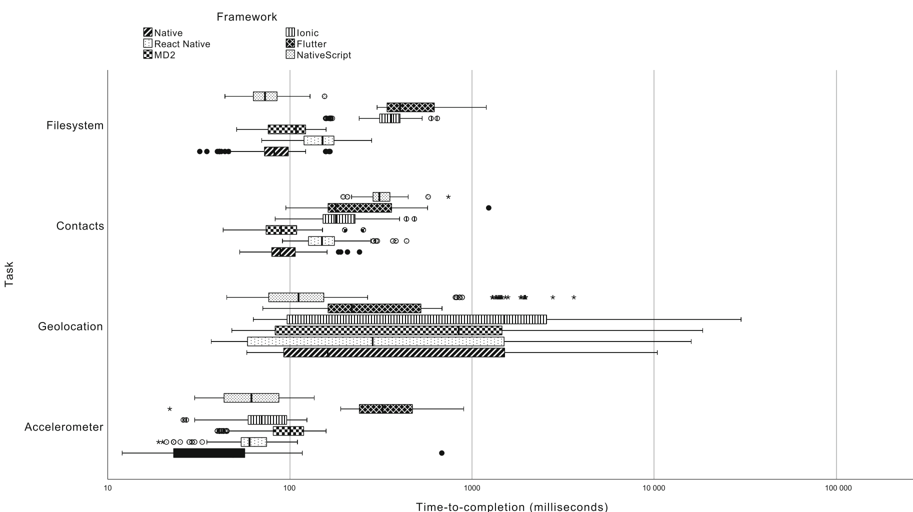


**Fig. 3** Log-scaled Boxplot of Time-to-Completion Results (in ms) per Framework per Task

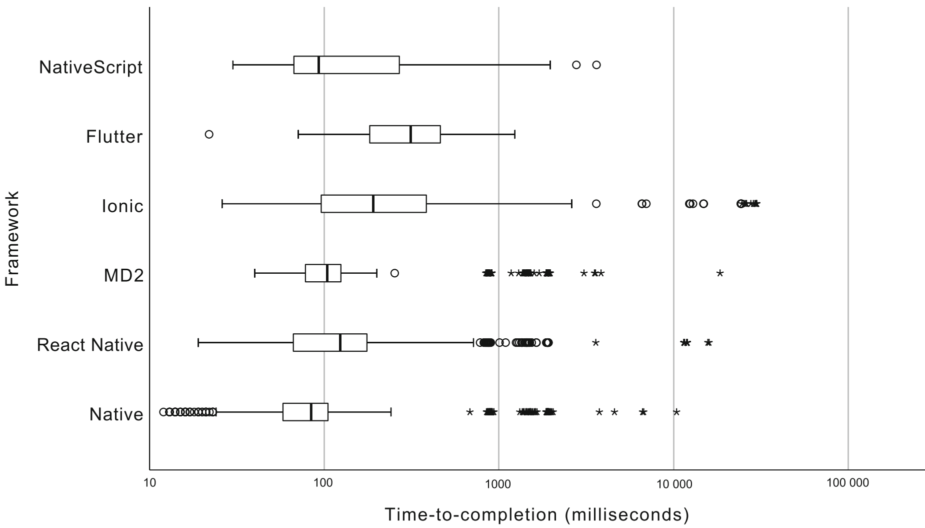


**Fig. 4** Log-scaled Boxplot of Time-to-Completion Results (in ms) per Framework Independent of Task and Device Type

# 4 Results

In this section, we firstly assess overall performance results independent of individual tasks and devices. We are interested in exploring the bigger picture: 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 Figs. 3, 4, 5, 6, 7 and 8 regular outliers are denoted by a black circle (●), while more extreme outliers by an asterisk (*). Subsequently, we assess each individual task on its performance across the frameworks scrutinized.

## 4.1 Time-to-Completion

The time-to-completion (`TTC`) metric reports on the duration between when a foreign function call is invoked on the front-end of the application, and when the result of the call is

**Table 4** Overview of overall time-to-completion performance results

| | Descriptives | | | | Analysis against native | | |
|---|---|---|---|---|---|---|---|
| Framework | Mean | SD | Max | Min | ANOVA $p$ | $\omega^2$ | 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 | $= .967$ |

**Table 5** Overview of PreRAM performance results

| Framework | Descriptives | | | | Analysis against native | | |
|---|---|---|---|---|---|---|---|
| | Mean | **SD** | Max | Min | ANOVA $p$ | $\omega^2$ | 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 | $< .001$ |

returned from the native side. From this metric, we can evaluate the speed-wise performance of each cross-platform framework, with the native implementation results as baseline.

Figure 3 (next page) shows a boxplot of time-to-completion per framework per task and Fig. 4 (p. 23) depicts a boxplot illustrating the time-to-completion between the different frameworks, regardless of task.

From a visual assessment of the results, we find a large amount of outliers in the dataset. 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.

In order to determine if differences in time-to-completion are statistically significant between the technical frameworks, we conducted a one-way ANOVA individually per framework with the native implementation as baseline. As depicted in Table 4 (p. 23), 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.

## 4.2 Memory Consumption

In this study, we differentiate between the general memory usage as occupied by the app in an idle state (`PreRAM`), and the memory usage during benchmarking (`RAM`). In particular,

**Table 6** Overview of RAM performance results

| Framework | Descriptives | | | | Analysis against native | | |
|---|---|---|---|---|---|---|---|
| | Mean | SD | Max | Min | ANOVA $p$ | $\omega^2$ | 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 | $< .001$ |

**Table 7** Overview of ComputedRAM performance results

| | Descriptives | | | | Analysis against native | | |
|---|---|---|---|---|---|---|---|
| Framework | Mean | SD | Max | Min | ANOVA $p$ | $\omega^2$ | 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 | $< .001$ |

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 calculated usage for that task is 15MB – which is what the metric `ComputedRAM` 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 constantly hold some data structures in memory.

In Tables 5, 6 and 7, we summarize the overall memory usage 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 artifacts benchmarked. In Fig. 5, a per-feature boxplot is provided, separated on framework. While still providing overview, the boxplot also shed light upon the differences in memory usage in more detail than the tables do. For instance, through visual assessment of Fig. 5, we find that the Ionic framework in general uses the most `ComputedRAM` memory, but also has the greatest variance.

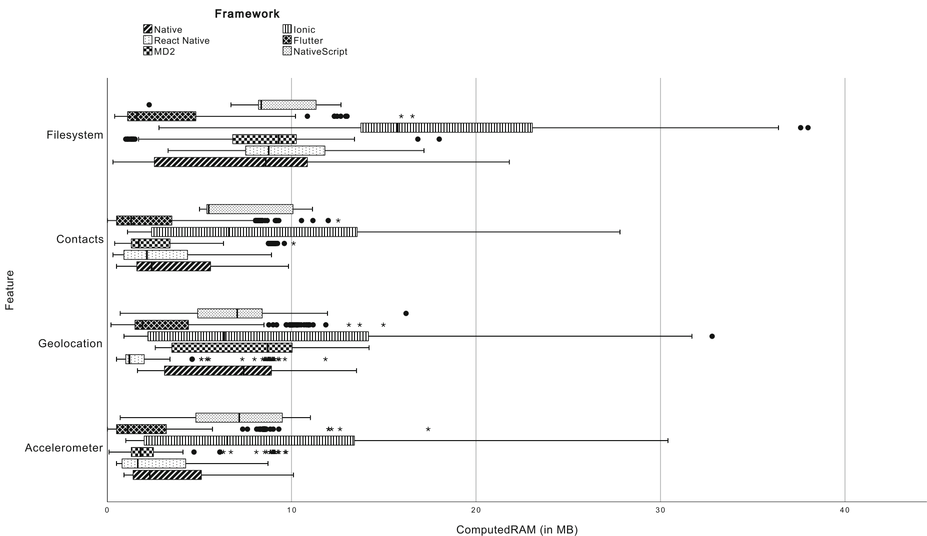


**Fig. 5** Linearly Scaled Boxplot of ComputedRAM Results (in MB) per Framework per Task

**Table 8** Results from Spearman's rank-order correlation coefficient tests on PreRAM and ComputedRAM against the native implementation baseline 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 |

The idle-state memory usage metric, `PreRAM` (cf. Table 5), is the profiler-reported usage when the app is running on the device and navigated to the respective test's view, but prior to running the benchmark. This way, we can measure any potential overhead that cross-platform frameworks impose on the memory occupancy at runtime when compared with the native baseline results. As expected, the native baseline has the lowest reported `RAM` (cf. Table 6) 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 7), 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 8. To discuss the strength of the correlation size, we follow *rule of thumb interpretation* by Hinkle et al. (1988). From the results in Table 8, 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 to Flutter's results in Tables 6 and 7, this could indicate that while Flutter has a high `PreRAM`, the impact on memory usage caused by executing the benchmark task – `ComputedRAM` – is low.

Figure 6 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

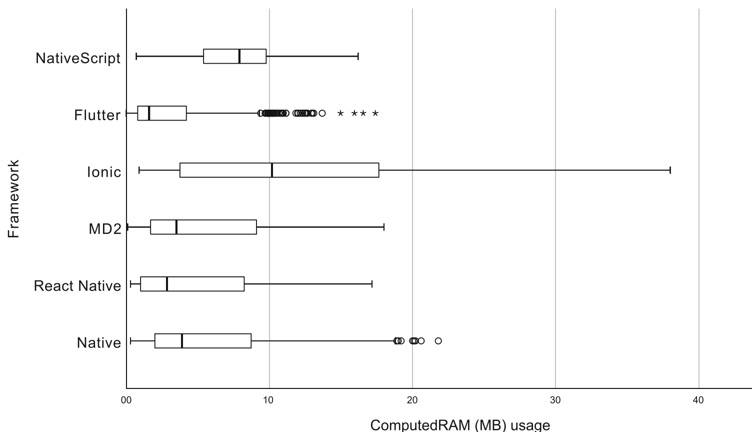


**Fig. 6** Linearly scaled boxplot of RAM Usage (in MB) across all tests and devices per framework

**Table 9** Overview of CPU performance results

| | Descriptives | | | | Analysis against native | | |
|---|---|---|---|---|---|---|---|
| Framework | Mean | SD | Max | Min | ANOVA $p$ | $\omega^2$ | 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 | $= .060$ |

a consistent low memory usage, although with a significant amount of high outliers. Also, results for Ionic show a huge variation. NativeScript has the second-highest mean usage, but with the lowest standard deviation. Only $MD_2$ exhibits no significant deviation from the native implementation. Interestingly, React Native undercuts the mean memory usage of the baseline. This is caused by alternative module implementations for the different features that deviate from recommended practices for a hand-written implementation (e.g., the geolocation module which performs slower and less accurate Facebook Inc 2019, cf. Section 4.7).

## 4.3 CPU Usage

In this study, we measured the CPU usage across all the frameworks as laid out in Table 9. The mean values were quite concentrated, with React Native and Ionic standing out as less effective. Although CPU usage will be heavily framework dependent, it introduces a possibility to see the impact of the individual frameworks from the tests.

Figure 7 – the boxplot of CPU usage – shows the use in percent across all tests and devices per framework. $MD_2$ comes out as the winner, although quite a few outliers can be observed. NativeScript performs with the most concentrated values and no particular

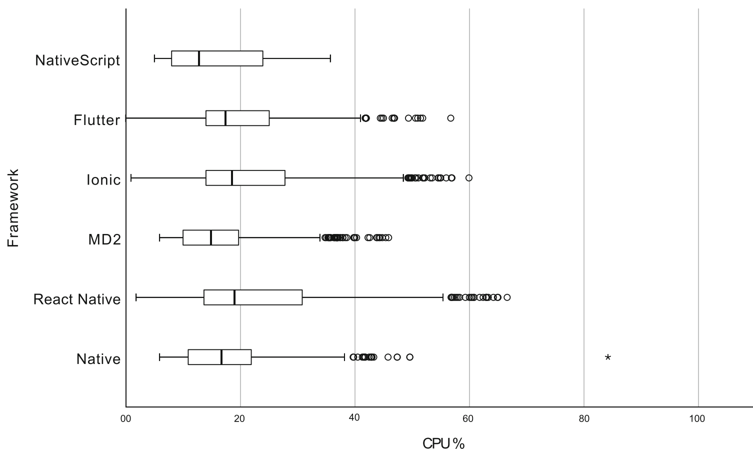


**Fig. 7** Linearly scaled boxplot of CPU usage (in %) across all tests and devices per framework

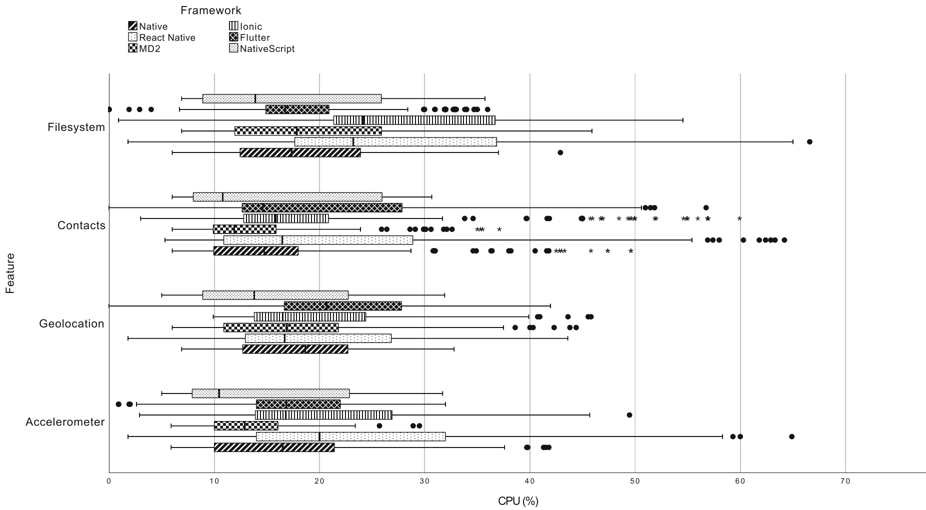


**Fig. 8** Linearly scaled boxplot of CPU results (in %) per framework per task

outliers exposed. The mean values of NativeScript and $MD_2$ again outperform the native baseline implementation, however, differences are not significant according to the ANOVA test. While Table 9 and Fig. 7 both provide `CPU` usage results independent of device and feature, Fig. 8 provide a more detailed look into the various frameworks' per-feature `CPU` usage performance. Through a visual assessment of Fig. 8, we find that especially the Contacts API had highly fluctuating `CPU` usage across all but the NativeScript-based implementation, indicating that for certain tasks, cross-platform frameworks may outperform the native baseline implementation in terms of reliability and consistency of performance results.

## 4.4 Accelerometer

In 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 UI 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 2 times slower retrieval as depicted in Table 10.

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

**Table 10** 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 | $= .082$ |

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, we investigate each metric in detail to uncover differences in performance impact between the cross-platform frameworks using results from the native implementation as baseline.

#### 4.4.1 Time-to-Completion for Accelerometer

Inspecting the Tukey post-hoc results in Table 10, we find 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, we find that 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 indicates 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.

$$TTC : F(1074, 5) = 657.217, p < .001, \omega^2 = .752 \tag{1}$$

The $\omega^2$ indicates that a large amount of 75.2% of the variation in accelerometer time-to-completion performance can be explained by the framework.

#### 4.4.2 CPU Usage for Accelerometer

Looking to the Tukey post-hoc results in Table 11, we find that 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 native. 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.

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

The $\omega^2$ indicates that 12.6% of the variation in accelerometer `CPU` usage can be explained by the framework.

**Table 11** 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 | $= .227$ |

### 4.4.3 PreRAM Usage for Accelerometer

The Tukey test in Table 12 indicate 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 highest maximum and minimum values. Closely following Flutter is the Ionic implementation, which share similarities regarding high values across all statistical columns.

$$PreRAM : F(984, 5) = 224.237,\ p < .001,\ \omega^2 = .530 \tag{3}$$

The $\omega^2$ indicates that 53.0% of the variation in accelerometer `PreRAM` performance can be explained by the framework.

### 4.4.4 ComputedRAM Usage for Accelerometer

From the Tukey test results in Table 13, we find 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 means are lower than the native implementation, the standard deviation comparable, and maximum values lower. The means 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.

$$ComputedRAM : F(984, 5) = 69.015,\ p < .001,\ \omega^2 = .256 \tag{4}$$

The $\omega^2$ indicates that 25.6% of the variation in accelerometer `ComputedRAM` performance can be explained by the framework.

**Table 12** 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 | $< .001$ |

**Table 13** 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 | $< .001$ |

## 4.5 Contacts

In terms of implementation and benchmarking challenges, we found that when benchmarking on devices without SIM cards, the process of creating and saving new contacts could end in failure without any exceptions thrown by the development framework. Using the `adb logcat` CLI tool, we could inspect an unfiltered stream of logs from the device over USB, thus manually seek out the 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.

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, we investigate each metric in detail to uncover differences in performance impact between the cross-platform frameworks using results from the native implementation as baseline.

### 4.5.1 Time-to-Completion for Contacts

As reported in Table 14, 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.

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

**Table 14** 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 | $< .001$ |

**Table 15** 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 | $= .981$ |

The $\omega^2$ indicates that 56.0% of the variation in contacts time-to-completion performance can be explained by the framework.

### 4.5.2 CPU Usage for Contacts

From the descriptive statistics in Table 15, we find that only two implementations have statistically significantly different means compared to the native implementation, namely React Native and Ionic. In this benchmark, we find that the MAML/$MD_2$ implementation has both a lower mean CPU utilization, lower standard deviation, lower maximum value and equal minimum value to the native implementation.

$$CPU : F(984, 5) = 13.407,\ p < .001,\ \omega^2 = .059 \tag{6}$$

The $\omega^2$ indicates that 5.9% of the variation in contacts CPU performance can be explained by the framework.

### 4.5.3 PreRAM for Contacts

From Table 16, 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/$MD_2$.

$$PreRAM : F(984, 5) = 215.704,\ p < .001,\ \omega^2 = .520 \tag{7}$$

**Table 16** 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 | $< .001$ |

**Table 17** 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 | $< .001$ |

The $\omega^2$ indicates that 52.0% of the variation in contacts `PreRAM` performance can be explained by the framework.

#### 4.5.4 ComputedRAM for Contacts

While Flutter has the lowest reported mean and minimum values (cf. Table 17), 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.

$$ComputedRAM : F(984, 5) = 75.003,\ p < .001,\ \omega^2 = .272 \tag{8}$$

The $\omega^2$ indicates that 27.2% of the variation in contacts `ComputedRAM` performance can be explained by the framework.

### 4.6 File System Access

File system access occurs frequently when data such as images is stored on the device itself or 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 1 MB 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.

#### 4.6.1 Time-to-Completion for File System Access

As reported in Table 18, 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 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.

$$Time : F(1074, 5) = 459.062,\ p > .001,\ \omega^2 = .680 \tag{9}$$

The $\omega^2$ indicates that 68.0% of the variation in file system time-to-completion can be explained by the framework.

**Table 18** 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 | $= .994$ |

### 4.6.2 CPU Usage for File System Access

As reported in Table 19, 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 highest standard deviation (cf. Table 19).

$$CPU : F(984, 5) = 38.270,\ p < .001,\ \omega^2 = .158 \tag{10}$$

The $\omega^2$ indicates that 15.8% of the variation in file system `CPU` performance can be explained by the framework.

### 4.6.3 PreRAM Usage for File System Access

By inspecting the mean variation in Table 20, 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.

$$PreRAM : F(984, 5) = 180.822,\ p < .001,\ \omega^2 = .476 \tag{11}$$

The $\omega^2$ indicates that 47.6% of the variation in file system `PreRAM` performance can be explained by the framework.

**Table 19** 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 | $= .944$ |

**Table 20** 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 | $= .004$ |

#### 4.6.4 ComputedRAM Usage for File System Access

For this test, Ionic particularly stands out as performing relatively bad when compared to the native implementation, with huge 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 21) 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 native on mean values, the Flutter implementation has a much lower mean than native.

$$ComputedRAM : F(984, 5) = 174.612, p < .001, \omega^2 = .467 \tag{12}$$

The $\omega^2$ indicates that 46.7% of the variation in file system `ComputedRAM` performance can be explained by the framework.

### 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 management object directly supports querying for the last known location. To actually retrieve up-to-date location values and avoid using cached values, which are updated according to a system-controlled rate, we measure the time to request a new value. 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.

**Table 21** 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 | $= .378$ |

**Table 22** 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 | $= .560$ |

### 4.7.1 Time-to-Completion for Geolocation

Apparently, 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 independent of our 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 22 exhibit a wide variation for all frameworks.

The most prominent outlier is the Ionic-based implementation. As our code already implements the geolocation provider options as suggested by the Ionic team (Lynch 2018), we were hesitant to attempt any further optimization of code or the underlying geolocation plug-in. From our search for information on the issue, we encountered numerous questions regarding the geolocation feature in Ionic, as also noted as the motivation behind the work of Lynch (2018). Thus, we treat the results as what should be expected of the framework without any optimizations.

$$Time : F(1074, 5) = 23.997, p < .001, \omega^2 = .096 \tag{13}$$

The $\omega^2$ indicates that only 9.6% of the variation in geolocation time-to-completion performance can be explained by the framework.

### 4.7.2 CPU Usage for Geolocation

Drawing on the results presented in Table 23, we find that 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

**Table 23** 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 | $= .215$ |

**Table 24** 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 | $< .001$ |

deviation. Minimum and maximum values of the NativeScript implementation closely resemble those reported also by the native implementation.

$$CPU : F(1014, 5) = 9.273, p < .001, \omega^2 = .041 \tag{14}$$

The $\omega^2$ indicates that 4.1% of the variation in geolocation `CPU` performance can be explained by the framework.

#### 4.7.3 PreRAM Usage for Geolocation

Drawing from the results in Table 24, we can see that 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.

$$PreRAM : F(1014, 5) = 207.898, p < .001, \omega^2 = .504 \tag{15}$$

The $\omega^2$ indicates that 50.4% of the variation in geolocation `PreRAM` performance can be explained by the framework.

#### 4.7.4 ComputedRAM Usage for Geolocation

As reported in Table 25, for this test the implementations written in both React Native and Flutter are statistically significantly different from native in a positive fashion, as both require less memory. NativeScript's memory requirement resemble that of the native

**Table 25** 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 | $= .976$ |

**Table 26** Weighting of frameworks, ordered by 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 | 1 | 2 | 2 | 1 | 1 | 7 |

implementation, while both MAML/MD$_2$ and Ionic used more memory than the aforementioned frameworks, although the latter used significantly more.

$$ComputedRAM : F(1014, 5) = 59.239, p < .001, \omega^2 = .222 \tag{16}$$

The $\omega^2$ indicates that 22.2% of the variation in geolocation `ComputedRAM` performance can be explained by the framework.

## 5 Discussion

In this section, we present our weighted overview of the technical frameworks and technologies, and discuss it in the context of our research question. Then we present immediate implications of our findings, followed by an overview of limitations to the study, and a comprehensive outlook and directions for further research.

### 5.1 Overview

To provide a general overview of the development technologies' performance, we have weighted them based on each measurement metric's mean value. We assign the framework with the lowest mean a score of 6 (highest), the highest mean a score of 1 (lowest), 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 our study. For this weighting, we do not separate between the different mobile devices or benchmarking tasks (e.g., geolocation, accelerometer, etc.). Instead, we base the weighting on the results presented in Sections 4.1, 4.2 and 4.3. Also, what Table 26 does not take into account is standard deviation or variation, so even if a framework is given a high or low weight, it does not necessarily reflect a consistently high or low performance.

Performance-wise, the model-driven MAML/MD$_2$ framework closely resemble the overall $\sum$ of the native development approach, according to Table 26. As model-driven frameworks generate platform-specific source code, this is perhaps not by itself a surprising finding. Possibly more surprising is the seeming lack of industry adoption of model-driven cross-platform development frameworks (Biørn-Hansen et al. 2019) when the results so closely resemble native performance, unlike several of the more industry-adopted frameworks scoring lower on the weighting. Thus, our findings may imply that practitioners and industry decision-makers should look more towards the model-driven approach, based on the performance results presented in this study.

As Table 26 illustrates, the hybrid approach-based Ionic framework has the overall lowest score. It ranks lowest on three of five metrics, however 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) and 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. We have not identified previous academic studies empirically investigating the performance of the Flutter framework.

What Table 26 best illustrates is the cross-platform trade-off developers face. If time-to-completion is the most important metric, thus adopting NativeScript, this will come at the cost of ComputedRAM. If minimizing 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.

**RQ:** 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 26 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 $\sum$, other cross-platform frameworks were found to possibly be more performant on certain metrics, such as NativeScript's time-to-completion and CPU usage, and Flutter's minimal increase in memory usage during task benchmarking, as seen in the ComputedRAM column.

## 5.2 Implications

In terms of practical implications, a major finding is the importance of having a technical specification or set of requirements as the foundation for deciding on a cross-platform development framework and overarching development approach. While the weighting of frameworks in Table 26 presents native as the overall best performing approach, other frameworks score higher in terms of certain metrics, e.g., NativeScript which scores highest on time-to-completion and CPU usage during task execution, or Flutter's lowest ComputedRAM usage. We do not believe there is a silver bullet among the frameworks included in our evaluation, however the weighting of frameworks and results from our experiment can hopefully help in future technical decision-making processes. Another implication for practice is Table 26, which could aid in decision-making where device attributes (e.g., processor and memory capabilities) are critical, for example in developing countries and emerging markets where typical mobile devices might be more on the lower end hardware-wise. Thus, our results suggest a varying degree of performance overhead

imposed by the cross-platform frameworks tested, although holistically the native approach provides an overall better performance when ranked on mean result values.

As for implications for future research, this paves the way for a discussion about the role and possible neglect of cross-platform frameworks and their bad reputation. Our methodology and through this, our results, highlight a baseline for comparison of such frameworks in future studies. Further, we add to the body of knowledge with a baseline of reference for future research projects and mobile performance studies.

### 5.3 Limitations

There are several threats to validity when conducting a software engineering experiment, and the study at hand is no exception. Differences in software architecture, programming languages, and cross-platform framework capabilities and matureness were some of the topics in question when designing the experiment. We put a great deal of effort into harmonizing the technical artifacts prior to the benchmarking. This was indeed important, as different developers worked on feature implementation, potentially resulting in skewed benchmarks as a result of differences in the artifacts. However, across or even within one framework or programming language, the same effect can be implemented by various means. We aimed for an efficient implementation of each artifact, although we did not explicitly test for differences within one framework.

Nevertheless, inevitable limitations of the study are the inherent differences between the frameworks, especially between different approaches to achieve cross-platform capabilities. For this reason, we focused on the programmatic access to the platform functionalities and deliberately disregarded their representation on screen to avoid interconnecting device access and UI rendering. Further research is possible to focus on the UI performance aspect of cross-platform benchmarks, although some approaches such as the reactive nature of React Native impede an efficient measurement of rendering performance. Additionally, as discussed in Section 3.4 on data gathering, timer APIs and their precision differed especially between the programming languages, rendering the timer unit in the Flutter implementation different from that in, e.g., the React Native artifact. This is a consideration that must be accounted for in the data analysis.

To the best of our abilities, we also ensured a common baseline between the devices, e.g., that WiFi and mobile data would be turned off, and that the apps would be terminated and restarted after each benchmark run ($n = 1$ run), ensure no bias through singular effects. Nonetheless, we are aware of the possibility of platform- and hardware-level measures out of our control, possibly impacting the results produced and reported on. This includes background tasks executed by other apps or the operating system, and also peaks in the execution time of the first run although each run is set up identically and resets all values after the measurement stops.

In terms of executing the benchmarks on-device, there is a possible performance impact when pressing the “Start benchmark” button in the graphical user interface. It should be assumed that pressing said button would require some processing power, which we do not account for in our current study. The same could also be said for re-rendering the screen with results, which is likely to require additional processing power. Nevertheless, this is true for all the benchmarks we’ve executed on all devices in all frameworks. These are covariates one could account for in future studies. Investigating the potential impact of running the Android Studio profiler on the development machine while executing the benchmarks on the mobile device (connected via USB), would also make for an interesting study. To the best of our knowledge, there is no other possible approach to conduct the gathering of

CPU and RAM data with minimal impact on benchmark performance. Also, the Android Studio profiler is the official profiling tool, thus the context would be similar to that of a practitioner's experience when profiling Android apps.

Finally, the combination of operating system version, framework version, and device capabilities has an immediate impact on the benchmark outcome and its reproducibility. To compensate for this problem, we have selected the devices to cover a variety of Android versions (version 6.0.1 to 8.0) which together account for 66.4% of currently installed Android operating systems (Google LLC 2018). In addition, the chosen list of devices includes recent devices with state-of-the-art computing power as well as older devices (cf. Table 3).

With regard to generalizability, we selected four commonly used device features for our study (cf. Section 3.3), i.e., accelerometer, contact list, file system, and geolocation. This selection covers both hardware and software features but of course further extension to other device capabilities is desirable. In particular, the device camera is arguably an important sensing component which could not yet be covered due to the aforementioned problems concerning comparability.

### 5.4 Outlook and Directions for Research

The implications of our study leave no doubt that much further research is needed. A framing is provided by the limitations, which at least partly ought to be overcome to make further contributions to theory. We additionally propose steps and ideas for future research as follows.

(i) A meta study investigating both technical and non-technical metrics and criteria could lead to a complete decision framework for cross-platform frameworks and approaches. While this current study investigates the performance of native function calls from cross-platform frameworks, it does not weigh or rank non-technical metrics or criteria including developer experience and happiness, tooling, number of supported platforms, extensibility, and more. These are indeed often-cited reasons to adopt a cross-platform development approach, even for single-platform development (McPherson 2019; Therox 2019). While such a meta study would be a grand undertaking, especially in a field of such rapid innovation, the outcome could be equally grand, and have impact and practical implications also outside of academia.

(ii) An investigation of the impact on user experience would be equally valuable. While the metrics we have included for this study, being time-to-completion (TTC), CPU, PreRAM, RAM, and ComputedRAM, can provide an overview of system performance, they do not inherently help in exploring user experience. We suggest developing user interface-rich mobile apps using the same set of cross-platform frameworks as included in this current study, and rather measure the users' experience and perception of quality, usability, and possibilities of developing high-quality user interfaces. Can an Ionic app, scoring the lowest on TTC, RAM, and ComputedRAM still provide a user experience on a par with the frameworks scoring higher on these metrics?

(iii) With the introduction of novel device classes, including foldable and flexible touch screens on mobile devices, smart car displays, smart wearables and more, what are the current possibilities and boundaries of cross-platform frameworks in these contexts? Also, are there unexpected differences in app performance between different Android

operating system versions? Does perhaps a newer version of the operating system allow cross-platform apps better performance than previous versions?

(iv) Development of a testbed which reduce the workload related to benchmarking and profiling. I.e., automated extraction and aggregation of CPU/RAM data from on-device apps executing benchmarks, and better so with multi-device support to minimize time spent waiting for results before switching devices to perform the exact same benchmarks again.

(v) In this current study, we have intentionally left out Apple iOS devices. This was done due to the already comprehensive data material and extent of the paper. For future research, we deem it only natural that a similar study will focus on Apple's iOS ecosystem.

(vi) With the increased focus on Progressive Web Apps, we urge the inclusion of this approach in future studies of similar nature. Looking into the potential enabling factors and drawbacks of this Web-native approach would be nothing short of timely and relevant to both practice and research.

## 6 Conclusion

In this study, we have investigated the performance overhead imposed by cross-platform mobile development frameworks in Android apps compared to the native development approach. Specifically, we have focused on the performance of native-side foreign function interface calls from a cross-platform context, to invoke and run device and platform functionality, including geolocation API, contacts API, file system integration, and accelerometer integration. We have gathered data 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. Based on statistical analysis and a weighted evaluation of the results, we investigate how well our developed artifacts perform compared to a developed native baseline artifact. Our 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 certain cross-platform frameworks can perform equally well or even better than native on certain metrics but no framework scores best across all features in this study. 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.

**Open Access** This article is licensed under a Creative Commons Attribution 4.0 International License, which permits use, sharing, adaptation, distribution and reproduction in any medium or format, as long as you give appropriate credit to the original author(s) and the source, provide a link to the Creative Commons licence, and indicate if changes were made. The images or other third party material in this article are included in the article's Creative Commons licence, unless indicated otherwise in a credit line to the material. If material is not included in the article's Creative Commons licence and your intended use is not permitted by statutory regulation or exceeds the permitted use, you will need to obtain permission directly from the copyright holder. To view a copy of this licence, visit http://creativecommonshorg/licenses/by/4.0/.

## References


Abousaleh M, Yarish D, Arora D, Neville S, Darcie T (2014) Determining per-mode battery usage within non-trivial mobile device apps. In: Proceedings - international conference on advanced information networking and applications (AINA), pp 202–209. https://doi.org/10.1109/AINA.2014.29

Ahti V, Hyrynsalmi S, Nevalainen O (2016) An evaluation framework for cross-platform mobile app development tools: a case analysis of Adobe PhoneGap framework. In: ACM International conference proceeding series, p 1164. https://doi.org/10.1145/2983468.2983484

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

Angulo E, Ferre X (2014a) A case study on cross-platform development frameworks for mobile applications and UX. In: Proceedings of the XV international conference on human computer interaction. ACM, p 27. https://doi.org/10.1145/2662253.2662280

Angulo E, Ferre X (2014b) A case study on cross-platform development frameworks for mobile applications and UX. In: Proceedings of the XV international conference on human computer interaction, Interaccion'14. ACM, New York, pp 27:1–27:8. https://doi.org/10.1145/2662253.2662280

Antebi O, Neubrand M, Puder A (2012) Cross-compiling android applications to windows phone 7. In: Zhang JY, Wilkiewicz J, Nahapetian A (eds) Mobile computing, applications, and services. Springer, Berlin, pp 283–302

AppBrain (2019) Android app frameworks. https://www.appbrain.com/stats/libraries/tag/app-framework/android-app-frameworks, accessed: 2019-10-23

applause (2015) applause. https://github.com/applause/

Archibald J (2016) Instant loading: Building offline-first progressive web apps. https://www.youtube.com/watch?v=cmGr0RszHc8&feature=youtu.be&t=2605

Bai J, Wang W, Qin Y, Zhang S, Wang J, Pan Y (2019) BridgeTaint: a bi-directional dynamic taint tracking method for JavaScript bridges in android hybrid applications. IEEE Trans Inform Forens Secur 14(3):677–692

Biørn-Hansen A, Ghinea G (2018) Bridging the gap: investigating device-feature exposure in cross-platform development. In: Proceedings of the 51st Hawaii international conference on system sciences. ScholarSpace, pp 5717–5724

Biørn-Hansen A, Majchrzak T, Grøli TM (2017) Progressive web apps: the possible web-native unifier for mobile development. In: WEBIST 2017 - proceedings of the 13th international conference on web information systems and technologies, pp 344–351

Biørn-Hansen A, Grønli TM, Ghinea G (2018) A survey and taxonomy of core concepts and research challenges in cross-platform mobile development. ACM Comput Surv 51(5):108:1–108:34

Biørn-Hansen A, Grønli TM, Ghinea G, Alouneh S (2019) An empirical study of cross-platform mobile development in industry. Wirel Commun Mob Comput 2019

Bizness Apps (2019) Mobile app maker — bizness apps. http://biznessapps.com/

Biørn-Hansen A, Majchrzak TA, Grønli TM (2018) Progressive web Apps for the unified development of mobile applications. In: Majchrzak TA, Traverso P, Krempels K, Monfort V (eds) Revised selected papers WEBIST 2017, springer, lecture notes in business information processing (LNBIP), vol 322, pp 64–86

Botella F, Escribano P, Peñalver A (2016) Selecting the best mobile framework for developing web and hybrid mobile apps. In: Proceedings of the XVII international conference on human computer interaction, Interacción '16. ACM, New York, pp 40:1–40:4, https://doi.org/10.1145/2998626.2998648

Bubble Group (2019) Bubble - visual programming. https://bubble.is/

Ciman M, Gaggi O (2014) Evaluating impact of cross-platform frameworks in energy consumption of mobile applications, vol 1, pp 423–431

Ciman M, Gaggi O (2015) Measuring energy consumption of cross-platform frameworks for mobile applications. LNBIP 226:331–346. https://doi.org/10.1007/978-3-319-27030-2_21

Ciman M, Gaggi O (2017) An empirical analysis of energy consumption of cross-platform frameworks for mobile development. Pervasive and Mobile Computing. https://doi.org/10.1016/j.pmcj.2016.10.004

Ciman M, Gaggi O, Gonzo N (2014) Cross-platform mobile development: a study on apps with animations. In: Proc. ACM symposium on applied computing. https://doi.org/10.1145/2554850.2555104

Corbalan L, Fernandez J, Cuitiño A, Delia L, Cáseres G, Thomas P, Pesado P (2018) 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. ACM, New York, pp 191–201, https://doi.org/10.1145/3197231.3197242

Corbalán L, Thomas P, Delía L, Cáseres G, Sosa JF, Tesone F, Pesado P (2019) A study of non-functional requirements in apps for mobile devices. In: Cloud computing and big data. Revised selected papers. Springer International Publishing, pp 125–136

Corral L, Sillitti A, Succi G (2012) Mobile multiplatform development: an experiment for performance analysis. Procedia Comput Sci 10:736–743. https://doi.org/10.1016/j.procs.2012.06.094

Cruz L, Abreu R (2019) On the energy footprint of mobile testing frameworks. IEEE Trans Software Eng, 12

Dalmasso I, Datta SK, Bonnet C, Nikaein N (2013) Survey, comparison and evaluation of cross platform mobile application development tools. In: 2013 9th International on wireless communications and mobile computing conference (IWCMC). IEEE, pp 323–328. https://doi.org/10.1109/IWCMC.2013.6583580

Delía L, Galdamez N, Corbalan L, Pesado P, Thomas P (2017) Approaches to mobile application development: Comparative performance analysis. In: 2017 Computing conference, pp 652–659. https://doi.org/10.1109/SAI.2017.8252165

Delia L, Galdamez N, Corbalan L, Pesado P, Thomas P (2018) Approaches to mobile application development: comparative performance analysis. In: Proceedings of computing conference 2017 2018-January, pp 652–659. https://doi.org/10.1109/SAI.2017.8252165

Dhillon S, Mahmoud QH (2015) An evaluation framework for cross-platform mobile application development tools. Softw Pract Exp 45(10):1331–1357. https://doi.org/10.1002/spe.2286

El-Kassas WS, Abdullah BA, Yousef AH, Wahba AM (2016) Enhanced code conversion approach for the integrated cross-platform mobile development (icpmd). IEEE Trans Softw Eng 42(11):1036–1053. https://doi.org/10.1109/TSE.2016.2543223

El-Kassas WS, Abdullah BA, Yousef AH, Wahba AM (2017) Taxonomy of cross-platform mobile applications development approaches. Ain Shams Eng J 8(2):163–190

Facebook Inc (2019) Getting started - react native. https://facebook.github.io/react-native/docs/getting-started

Ferreira C, Peixoto M, Duarte P, Torres A, Júnior M, Rocha L, Viana W (2018) An evaluation of cross-platform frameworks for multimedia mobile applications development. IEEE Lat Am Trans 16(4):1206–1212. https://doi.org/10.1109/TLA.2018.8362158

Flutter Developers (2019) Flutter FAQ. https://flutter.dev/docs/resources/faq

Google LLC (2018) Distribution dashboard. https://developer.android.com/about/dashboards/, 2018-9-14

Google LLC (2019a) Benchmark app code. https://developer.android.com/studio/profile/benchmark, accessed: 2019-5-21

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

Google LLC (2019c) J2ObjC. http://j2objc.org/

Google LLC (2019d) Profile and debug pre-build APKs. https://developer.android.com/studio/debug/apk-debugger, accessed: 2019-5-2

Google LLC (2019e) Build for the next billion users. https://developer.android.com/distribute/best-practices/develop/build-for-the-next-billion, accessed: 2012-1-07

Grønli TM, Hansen J, Ghinea G, Younas M (2014) 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. IEEE, pp 635–641

Heitkötter H, Majchrzak TA (2013) Cross-platform development of business apps with MD2. In: Design science at the intersection of physical and virtual design. Lecture notes in computer science. Springer, Berlin, pp 405–411, https://doi.org/10.1007/978-3-642-38827-9_29

Heitkötter H, Hanschke S, Majchrzak TA (2012) Comparing cross-platform development approaches for mobile applications. In: Proceedings 8th WEBIST. SciTePress, pp 299–311

Heitkötter H, Hanschke S, Majchrzak TA (2013) Evaluating Cross-Platform development approaches for mobile applications. In: Web information systems and technologies, lecture notes in business information processing. Springer, Berlin, pp 120–138, https://doi.org/10.1007/978-3-642-36608-6_8

Heitkötter H, Hanschke S, Majchrzak TA (2013a) Evaluating cross-platform development approaches for mobile applications. In: Cordeiro J, Krempels K (eds) Revised selected papers WEBIST 2012, lecture notes in business information processing (LNBIP), vol 140. Springer, pp 120–138

Heitkötter H, Majchrzak TA, Kuchen H (2013b) Cross-platform model-driven development of mobile applications with $MD^2$. In: Proceedings of the 28th annual ACM symposium on applied computing (SAC). ACM, pp 526–533

Hinkle DE, Wiersma W, Jurs SG et al (1988) Applied statistics for the behavioral sciences, vol 5. Houghton Mifflin, Boston

Hudli A, Hudli S, Hudli R (2015) An evaluation framework for selection of mobile app development platform. In: Proc. 3rd MobileDeLi. https://doi.org/10.1145/2846661.2846678

Humayoun S, Ehrhart S, Ebert A (2013) Developing mobile apps using cross-platform frameworks: a case study. Lecture Notes in Computer Science 8004 LNCS(PART 1):371–380. https://doi.org/10.1007/978-3-642-39232-0_41

Jansen S, Bloemendal E (2013) Defining app stores: the role of curated marketplaces in software ecosystems. In: Herzwurm G, Margaria T (eds) Software business. From physical products to software services and

solutions. ICSOB 2013, lecture notes in business information processing, vol 150. Springer, pp 195–206, https://doi.org/10.1007/978-3-642-39336-5_19

Jia X, Ebone A, Tan Y (2018) A performance evaluation of cross-platform mobile application development approaches. In: 5th International conference on mobile software engineering and systems (MOBILESoft). ACM, pp 92–93, https://doi.org/10.1145/3197231.3197252

Joorabchi ME, Mesbah A, Kruchten P (2013) Real challenges in mobile app development. In: 2013 ACM/IEEE international symposium on empirical software engineering and measurement (ESEM), pp 15–24, https://doi.org/10.1109/ESEM.2013.9

Katevas K, Haddadi H, Tokarchuk L (2016) Sensing kit: evaluating the sensor power consumption in iOS devices. In: Proceedings - 12th international conference on intelligent environments, IE 2016, pp 222–225. https://doi.org/10.1109/IE.2016.50

Kirk RE (1996) Practical significance: a concept whose time has come. Educ Psychol Meas 56(5):746–759

Kromer L, Wagner M, Blumenstein K, Rind A, Aigner W (2016) Performance comparison between unity and d3.js for cross-platform visualization on mobile devices. CEUR Workshop Proc 1734:47–52

Lachgar M, Abdali A (2017) Decision framework for mobile development methods. Int J Adv Comput Sci Appl (IJACSA) 8:2. https://doi.org/10.14569/IJACSA.2017.080215

Latif M, Lakhrissi Y, Nfaoui EH, Es-Sbai N (2016a) Cross platform approach for mobile application development: a survey. In: 2016 International conference on information technology for organizations development (IT4OD). IEEE, pp 1–5. https://doi.org/10.1109/IT4OD.2016.7479278

Latif M, Lakhrissi Y, Nfaoui EH, Es-Sbai N (2016b) Cross platform approach for mobile application development: a survey. In: 2016 International conference on information technology for organizations development, IT4OD 2016. https://doi.org/10.1109/IT4OD.2016.7479278

Latif M, Lakhrissi Y, Nfaoui EH, Es-Sbai N (2017) Review of mobile cross platform and research orientations. In: 2017 International conference on wireless technologies, embedded and intelligent systems (WITS). IEEE, pp 1–4, https://doi.org/10.1109/WITS.2017.7934674

Li L, Bissyandé TF, Wang H, Klein J (2018) Cid: automating the detection of api-related compatibility issues in android apps. In: Proceedings of the 27th ACM SIGSOFT international symposium on software testing and analysis ISSTA 2018. ACM, New York, pp 153–163. https://doi.org/10.1145/3213846.3213857

Lynch M (2018) Testing geolocation on Android. https://blog.ionicframework.com/testing-geolocation-on-android/, accessed: 2018-8-23

Macedonia M (2007) iPhones target the tech elite. Computer 40:94–95

Majchrzak TA, Heitkötter H (2014) Status quo and best practices of app development in regional companies. In: Krempels K, Stocker A (eds) Revised selected papers WEBIST 2013, lecture notes in business information processing (LNBIP), vol 189. Springer, pp 189–206

Majchrzak TA, Ernsting J, Kuchen H (2015) Achieving business practicability of model-driven cross-platform apps. OJIS 2(2):3–14

Malavolta I, Procaccianti G, Noorland P, Vukmirović P (2017) 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. IEEE Press, Piscataway, pp 35–45

McPherson D (2019) Townske app in react native. https://hackernoon.com/townske-app-in-react-native-6ad557de7a7c, accessed: 2019-10-23

Mercado IT, Munaiah N, Meneely A (2016) The impact of cross-platform development approaches for mobile applications from the user's perspective. In: WAMA 2016 - Proceedings of the international workshop on app market analytics, co-located with FSE 2016. https://doi.org/10.1145/2993259.2993268

Mutti S, Fratantonio Y, Bianchi A, Invernizzi L, Corbetta J, Kirat D, Kruegel C, Vigna G (2015) Baredroid: large-scale analysis of android apps on real devices. In: Proceedings of the 31st annual computer security applications conference, ACSAC 2015. ACM, pp 71–80, https://doi.org/10.1145/2818000.2818036

Nelson R (2018) Global app revenue grew 35% in 2017 to nearly $60 billion. https://sensortower.com/blog/app-revenue-and-downloads-2017, accessed: 2018-2-8

Noei E, Syer MD, Zou Y, Hassan AE, Keivanloo I (2017) A study of the relation of mobile device attributes with the user-perceived quality of android apps. Empir Softw Eng 22(6):3088–3116. https://doi.org/10.1007/s10664-017-9507-3

Ohrt J, Turau V (2012) Cross-platform development tools for smartphone applications. Computer 45(9):72–79. https://doi.org/10.1109/MC.2012.121

OpenSignal (2015) Android fragmentation visualized. Tech. rep., OpenSignal. https://www.opensignal.com/sites/opensignal-com/files/data/reports/global/data-2015-08/2015_08_fragmentation_report.pdf

Palmieri M, Singh I, Cicchetti A (2012) Comparison of cross-platform mobile development tools. In: Proc. 16th ICIN. IEEE, pp 179–186, https://doi.org/10.1109/ICIN.2012.6376023

Perchat J, Desertot M, Lecomte S (2014) Common framework: a hybrid approach to integrate cross-platform components in mobile application. J Comput Sci 10(11):2164–2180. https://doi.org/10.3844/jcssp.2014.2164.2180

Que P, Guo X, Zhu M (2017) A comprehensive comparison between hybrid and native app paradigms. In: Proceedings - 2016 8th international conference on computational intelligence and communication networks, CICN 2016, pp 611–614. https://doi.org/10.1109/CICN.2016.125

Ribeiro A, da Silva AR (2012) Survey on Cross-Platforms and languages for mobile apps. In: 2012 Eighth international conference on the quality of information and communications technology. IEEE, pp 255–260, https://doi.org/10.1109/QUATIC.2012.56

Rieger C, Kuchen H (2018) A process-oriented modeling approach for graphical development of mobile business apps. Comput Lang Syst Struct 53:43–58. https://doi.org/10.1016/j.cl.2018.01.001

Rieger C, Majchrzak TA (2016) Weighted evaluation framework for cross-platform app development approaches. In: Wrycza S (ed) Information systems: development, research, applications, education: 9th SIGSAND/PLAIS EuroSymposium 2016, Gdansk, Poland, September 29, 2016, Proceedings. Springer, pp 18–39, https://doi.org/10.1007/978-3-319-46642-2_2

Rieger C, Majchrzak TA (2018) A taxonomy for app-enabled devices: mastering the mobile device jungle. In: Majchrzak TA, Traverso P, Krempels KH, Monfort V (eds) Web information systems and technologies. Springer International Publishing, Cham, pp 202–220

Rieger C, Majchrzak TA (2019) Towards the definitive evaluation framework for cross-platform app development approaches. J Syst Softw (JSS) 153:175–199. https://doi.org/10.1016/j.jss.2019.04.001

Rösler F, Nitze A, Schmietendorf A (2014) Towards a mobile application performance benchmark. In: ResearchGate

Scalabrino S, Bavota G, Linares-Vásquez M, Lanza M, Oliveto R (2019) Data-driven solutions to detect api compatibility issues in android: an empirical study. In: 2019 IEEE/ACM 16th international conference on mining software repositories (MSR), pp 288–298, https://doi.org/10.1109/MSR.2019.00055

Seidel E (2018) Hot update / out of band updates · issue #14330 · flutter/flutter. https://github.com/flutter/flutter/issues/14330

Singh G (2017) Android app performance optimization. https://medium.com/mindorks/android-app-performance-optimization-cdccb422e38e, accessed: 2019-5-21

Sommer A, Krusche S (2013) Evaluation of cross-platform frameworks for mobile applications. LNI P-215

Stack Exchange Inc (2019) Stack overflow trends. https://insights.stackoverflow.com/trends?tags=nativescript. accessed: 2019-10-23

Stahl T, Völter M (2006) Model-driven software development. Wiley, Chichester

Statista Inc (2016) Number of smartphone users worldwide 2014-2020. https://www.statista.com/statistics/330695/number-of-smartphone-users-worldwide/, accessed: 2018-2-8

Statista Inc (2018a) Global mobile OS market share in sales to end users from 1st quarter 2009 to 1st quarter 2018. https://doi.org/https://www.statista.com/statistics/266136/global-market-share-held-by-smartphone-operating-systems/

Statista Inc (2018b) Global app economy size 2021. https://www.statista.com/statistics/267209/global-app-economy/, accessed: 2019-7-25

Tang W, Lee Jh, Song B, Islam M, Na S, Huh EN (2011) Multi-platform mobile thin client architecture in cloud environment. Procedia Environ Sci 11:499–504. https://doi.org/10.1016/j.proenv.2011.12.079

Therox O (2019) React native at artsy, 3 years later. https://artsy.github.io/blog/2019/03/17/three-years-of-react-native/, accessed: 2019-10-23

Viennot N, Garcia E, Nieh J (2014) A measurement study of google play. In: The 2014 ACM international conference on Measurement and modeling of computer systems, vol 42. ACM, New York, pp 221–233, https://doi.org/10.1145/2637364.2592003

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

WebRatio Srl (2015) Rapid mobile app and web application development platform. http://www.webratio.com/site/content/en/home

Wei L, Liu Y, Cheung SC (2016) 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. ACM, New York, pp 226–237. https://doi.org/10.1145/2970276.2970312

Willocx M, Vossaert J, Naessens V (2015) A quantitative assessment of performance in mobile app development tools. In: 2015 IEEE International conference on mobile services (MS). IEEE, pp 454–461, https://doi.org/10.1109/MobServ.2015.68

Willocx M, Vossaert J, Naessens V (2016) Comparing performance parameters of mobile app development strategies. In: 2016 IEEE/ACM international conference on mobile software engineering and systems (MOBILESoft). IEEE, pp 38–47, https://doi.org/10.1109/MobileSoft.2016.028

**Publisher's note** Springer Nature remains neutral with regard to jurisdictional claims in published maps and institutional affiliations.

**Andreas Biørn-Hansen** is a Ph.D. Research Fellow with the Department of Technology at Kristiania University College (KUC), Norway, and a Doctoral Researcher at the Department of Computer Science at Brunel University London, UK. His research interests include empirical software engineering within the Web and mobile software sphere, mining and analyzing software repositories, and software architecture. He obtained his MSc from Westerdals Oslo School of Arts, Communication and Technology, Norway, and his BSc from the Norwegian School of IT, Norway. At KUC, he is affiliated with the Mobile Technology Lab (MOTEL).

**Christoph Rieger** received his PhD in Information Systems at the University of Münster where he worked as research assistant at the practical computer science group. His research interests are in the areas of model-driven software development and domain-specific languages, particularly related to their application to current and novel app-enabled mobile devices.

**Tor-Morten Grønli** is a professor at the Department of Technology, Kristiania University College, Norway. He graduated with a bachelor's degree in IT from the Norwegian IT College, a master's in distributed systems from Brunel University, UK and a doctorate in computer science from the same place. He is a visiting professor at Copenhagen Business School and has over 90 peer-reviewed publications. Tor-Morten is the founding director of the research group for applied computer science and the Mobile Technology Lab at the Institute of Technology. His primary research interests are mobile computing, Internet of Things and software architectures.

**Tim A. Majchrzak** is professor in Information Systems at the University of Agder (UiA) in Kristiansand, Norway. He also is a member of the Centre for Integrated Emergency Management (CIEM) at UiA. Tim received BSc and MSc degrees in Information Systems and a PhD in economics (Dr. rer. pol.) from the University of Münster, Germany. His research comprises both technical and organizational aspects of Software Engineering, typically in the context of Mobile Computing. He has also published work on diverse interdisciplinary Information Systems topics, most notably targeting Crisis Prevention and Management. Tim's research projects typically have an interface to industry and society. He is a senior member of the IEEE and the IEEE Computer Society, and a member of the Gesellschaft für Informatik e.V.

**Prof. Gheorghita Ghinea** is a Professor of Mulsemedia Computing in the Department of Computer Science at Brunel University. Prof. Ghinea heads the Interactive Multimedia Systems Group at Brunel, boasting strong research ambitions and a track record of innovative solutions as well as a hands-on approach where the developed solutions are validated with real systems. He has over 300 publications in peer-reviewed journals and conferences, and has consulted for both the public and private sector in his areas of expertise. Prof. Ghinea plays an active part on the international academic scene, being regularly invited to edit journal special issues and to be a programme committee member of leading conferences worldwide.

## Affiliations

**Andreas Biørn-Hansen[1,2] · Christoph Rieger[3] · Tor-Morten Grønli[1] · Tim A. Majchrzak[4] · Gheorghita Ghinea[1,2]**

Christoph Rieger
christoph.rieger@uni-muenster.de

Tor-Morten Grønli
tor-morten.gronli@kristiania.no

Tim A. Majchrzak
timam@uia.no

Gheorghita Ghinea
george.ghinea@brunel.ac.uk

1 Mobile Technology Lab, Kristiania University College, Oslo, Norway
2 Brunel University, London, UK
3 University of Münster, Münster, Germany
4 University of Agder, Kristiansand, Norway