Universität Hamburg
DER FORSCHUNG | DER LEHRE | DER BILDUNG

MASTERTHESIS

AI-Assisted Coding and Comparative Performance Analysis of Cross-Platform Mobile Applications: Kotlin Multiplatform vs. React Native

vorgelegt von

Jacqueline Böhm

Abgabedatum: 12.02.2024

MIN-Fakultät

Fachbereich Informatik

Arbeitsbereich House of Computing and Data Science

Studiengang: Informatik

Matrikelnummer: 7329079

Erstgutachter: Dr. Martin Semmann

Zweitgutachter: Prof. Dr. Janick Edinger

Abstract

In today's dynamic landscape of mobile application development, the integration of artificial intelligence (AI) and the exploration of cross-platform mobile development (CPMD) solutions have emerged as promising paths for enhancing efficiency and performance. This thesis presents a comprehensive investigation into AI-assisted coding and comparative performance analysis of CPMD solutions, with a focus on the novel Kotlin Multiplatform (KMP) technology and the established React Native framework.

The research objectives encompass assessing the effectiveness of AI-generated code in aiding mobile application development, identifying key metrics for comparing CPMD solutions, and quantifying performance differences among the evaluated CPMD solutions. Through a systematic exploration of AI-assisted coding experiences, this study provides valuable insights into the efficacy of AI integration. By utilizing metric-based performance comparisons, this study aims to observe the performance nuances between CPMD solutions, namely KMP and React Native, as well as the baseline solution of Android Native.

The research methodology is embedded in a case study design and employs a systematic literature review, the development of sample applications, and a metric-based comparison. The findings highlight the potential of AI-assisted coding in streamlining development workflows and particularly in skill acquisition, which can enhance productivity. Additionally, the findings reflect the challenges encountered when utilizing AI for code creation. The comparative analysis sheds light on performance disparities among the evaluated CPMD solutions.

The implications of this research extend to developers, researchers, and educators in the fields of computer science and software development, offering valuable insights into the integration of AI and the performance measurement of CPMD solutions. By addressing key research objectives and contributing to the ongoing discourse on mobile application development, this thesis contributes to the understanding of AI integration and CPMD solutions, paving the way for future innovations and advancements in the field.

Keywords: Cross-platform mobile development, performance metrics, AI-assisted coding, Kotlin Multiplatform, React Native

Contents

Appendix List

Figures

Tables

Abbreviations

AI	Artificial intelligence
ANOVA	Analysis of variance
AOSP	Android Open Source Project
API	Application programming interface
APK	Android package kit
CPMD	Cross-platform mobile development
CPU	Central Processing Unit
DSL	Domain-specific language
GUI	Graphical user interface
IDE	Integrated development environment
KMP	Kotlin Multiplatform
LLM	Large language model
LOC	Lines of code
OS	Operating system
PWA	Progressive web application
RN	React Native
RQ	Research question
SDK	Software development kit
UI	User interface
WORA	write once – run anywhere

1. Introduction

Since 2017, mobile operating systems, particularly Android and iOS, have dominated web usage, surpassing traditional desktop operating systems. This shift signifies a transition to mobile-centric computing, with smartphones leading in usage compared to other devices. Despite regional variations, the combined market share of Android and iOS accounts for 80% of the global mobile operating system market (Biørn-Hansen, Grønli, & Ghinea, 2018, p. 2). This number is not surprising, considering that the present era is characterized by the so-called *appification*, the continuous process of adapting digital services into mobile applications. This phenomenon facilitates the use of these services on mobile devices such as smartphones and tablets (EASA Inc., 2021). From the corporate perspective, mobile development has become increasingly challenging and costly. This stems from the requirement of supporting various mobile operating systems, at least Android and iOS, when using the traditional Native development approach where applications are created for only one specific platform each (Biørn-Hansen et al., 2020, p. 2998). Thus, cross-platform mobile development (CPMD) technologies that use a unified codebase for multiple platforms can serve as a viable alternative to the Native development approach. They have the potential to be cost and time saving, while simultaneously introducing a particular performance overhead, depending on the specific implemented technology, in comparison to Native (Biørn-Hansen, Grønli, & Ghinea, 2018, p. 21; Biørn-Hansen et al., 2020, p. 3031).

Since there is no silver bullet among technologies, companies are faced with the challenge of selecting the most suitable development approach for their purposes (Biørn-Hansen, Grønli, & Ghinea, 2018, p. 21; Shah et al., 2019, p. 2). Biørn-Hansen et al. (2022, p. 7780) discover that cross-platform applications are particularly prevalent in the business, finance, and education sectors, comprising approximately 20-30% of each. Biørn-Hansen et al. (2022, p. 7779) also highlight that the top three search interests for CPMD technologies on Google were React Native, Xamarin, and Ionic. At the same time, Biørn-Hansen, Grønli, et al. (2019, p. 9) emphasize that React Native and Ionic are increasingly popular for practitioners. Biørn-Hansen, Grønli, et al. (2019, p. 9) also describe that the adoption of CPMD technologies by practitioners does not always align with frequently researched technologies in the science community. They point out that indications from practitioners could guide researchers in their decisions regarding which technologies to include in their research (Biørn-Hansen, Grønli, et al., 2019, p. 9). This study is motivated by this recommendation. The young and innovative CPMD technology Kotlin Multiplatform (KMP) by JetBrains describes itself as very promising (JetBrains, 2023b), while it lacks presence in the academic literature, indicating its novelty. The additional recommendation made by Biørn-Hansen, Grønli and Ghinea (2018, p. 21) to research the emerging technology KMP is taken into account since it represents a research gap in the literature due to the presence of only one academic paper. This single source about KMP is a Master's thesis by Evert (2019), where KMP is compared solely with Native. In response to the recent surge in generative artificial intelligence (AI), notably ChatGPT, the topic has already found its way into research in the field of software development (Becker et al., 2023; Calegario et al., 2023). The current advancement of AI lends itself to being integrated into the coding process as an additional facet of this research. The aim is to utilize AI-enabled code generation as a novel approach to enhance the software development process, in this case for developing the sample applications. This choice is driven by the recognized potential of AI code generation, which could be capable of enhancing the scalability and efficiency of software development, thus opening new paths for exploration and experimentation.

Therefore, in this thesis, the performance of two different CPMD technologies are compared based on their application performance. For that, sample applications with the same features are built using each evaluated technology with the help of AI-generated code. Afterwards, the performance of these sample applications is measured with metrics. The context of this thesis is to provide benchmarking about CPMD solutions that are currently relevant in mobile development practice. To investigate several of the described aspects regarding CPMD and the integration of AI in software development, this thesis aims to specifically address the following research questions:

RQ 1: How effectively can AI-generated code be utilized for a systematic comparison of CPMD solutions?

RQ 2: Which metrics can be identified for a systematic comparison of CPMD solutions with special regard to resource conservation?

RQ 3: What are the quantifiable performance differences observed between the two CPMD technologies KMP and React Native?

RQ 1 explores the potential of AI-generated code in aiding the development of mobile applications using different CPMD solutions. It seeks to discover the extent to which the support of AI can facilitate the coding process. RQ 2 involves the identification of quantifiable metrics for comparing mobile applications built with both CPMD technologies with a particular focus on resource conservation. RQ 3 investigates the performance comparison of both CPMD technologies used in implemented sample applications. Metrics are used to measure and evaluate potential performance differences. The idea of this thesis is to involve KMP as novel technology, comparing it with the well-established CPMD framework React Native, additionally employing a Native application as a baseline reference. The claim that KMP demonstrates remarkable efficiency and closely mirrors Native performance is the subject of investigation. The main goal of the performance comparison, and thus of RQ 3, is to verify and quantitatively assess these performance differences, contributing crucial insights to the ongoing discourse on the efficacy of KMP in mobile application development.

By evaluating the new technology phenomenon KMP, this study can assist in decision-making processes regarding the selection of a cross-mobile development approach. By exploring the experimental implementation of sample applications for performance measurement using KMP in comparison to a more established CPMD framework, this thesis will offer practical insights for developers and researchers interested in cross-platform mobile application development. Additionally, it demonstrates the role of AI-enabled code generation in the scope of this study, potentially offering insights into how such technologies can be incorporated into the development or code learning process. From a scientific perspective, this thesis will contribute to existing knowledge of established CPMD frameworks and will additionally make a novel contribution to the limited existing scientific publications about KMP in practical application. Moreover, it could inspire a new methodological approach in terms of incorporating AI-generated code when developing applications in a scientific study. From the practical standpoint, this thesis provides benchmarking of different CPMD solutions that can help practitioners in choosing a potential fitting solution based on the conducted performance analysis. In addition, it can provide insights for developers regarding the practicality of incorporating AI-enabled code generation in specific scenarios. Evaluating the two chosen CPMD solutions including the KMP technology is relevant to both practice and research due to the novelty of the technology and the lack of scientific papers specifically addressing the performance of KMP. A further research gap is filled by providing a direct comparison between established CPMD frameworks and the emerging KMP technology, quantifying performance differences. Furthermore, this thesis provides a novel insight into the use of AI-generated code in a scientific case study setting and assesses the development experience perceived.

In the following, an overview of the structure of this thesis is provided, which comprises several key chapters exploring AI-assisted coding and comparative performance analysis of CPMD technologies. The stage is set for the research in Chapter 1 by introducing the topic, research questions and objectives. The significance of investigating AI integration and CPMD solutions in the context of mobile application development is highlighted. Moving on to Chapter 2, the theoretical foundations underpinning the research are explored in depth. This chapter delves into mobile platforms, CPMD, and AI-assisted coding, with subsections focusing on key concepts such as Android and iOS platforms, cross-platform vs. Native development, and the role of AI in code generation. Chapter 3 details the methodology employed in the research. This includes the research design, literature review process, and sample application development. The methodology for metric-based comparison and performance analysis is also outlined. In Chapter 4, comparative studies of different CPMD solutions are presented. Drawing insights from existing research, comparison metrics are identified, and the nuances of each solution are explored. Chapter 5 provides insights into the implementation details of the sample applications. Focusing on features, architecture, and variations across technologies, this chapter also delves into the AI-assisted coding experience and its implications for development workflows. Chapter 6 presents the findings of the metric-based performance comparison. In the scope of this comparative analysis, the performance metrics are analyzed, and comparisons between the employed technologies are drawn. In Chapter 7, discussions on the AI-assisted coding experience and metric-based performance comparison take place. Furthermore, key findings, limitations, and challenges encountered throughout the research process are highlighted. Finally, in Chapter 0, the main findings of the research are summarized, implications for future research and practice are discussed, and final remarks are presented. The appendix additionally contains, besides the supplementary material included at the end of this paper, digital data, such as source code and measurement records, as further explained in the appendix section. By following this structured approach, this thesis aims to provide a comprehensive exploration of AI coding integration and comparison of CPMD solutions, contributing to the ongoing discourse on mobile application development. Before delving into the outlined research activities, a solid theoretical foundation serves as the basis for the subsequent research journey.

2. Theoretical Background

The theoretical background context essential for CPMD and the AI integration into coding is explored in this chapter. Beginning with an overview of mobile platforms, the prominent platforms Android and iOS are introduced. Subsequently, CPMD is portrayed, differentiating it from Native development and describing different approaches and technologies to build applications across multiple platforms. Lastly, the innovative domain of AI-assisted coding is addressed, which leverages artificial intelligence to aid in application development.

2.1. Mobile Platforms

Mobile platforms refer to the operating systems for mobile devices, such as smartphones, tablets, and wearables (Silberschatz et al., 2018, p. 41). Silberschatz et al. (2018, p. 3) characterizes an operating system as a software that manages hardware, supports applications, and serves as an intermediary between users and hardware. The global mobile platforms market is dominated by Android and iOS (Silberschatz et al., 2018, p. 42). According to Gartner Inc. (2017), Android and iOS held a market share of 99.6% in global smartphone sales in the fourth quarter of 2017. Alternative mobile platforms exist, for instance Windows and Blackberry, but this study omits these platforms due to their minor presence within the smartphone market, as portrayed in Table 1, and exclusively focuses on Android and iOS.

Table 1: *Worldwide Smartphone Sales to End Users by Operating System in 4th Quarter 2016 (Adapted from Source: Gartner Inc., 2017)*

Mobile Platform	**Sold Items (Thousands of Units)**	**Market Share (%)**
Android	352,669.9	81.72
iOS	77,038.9	17.85
Windows	1,092.2	0.25
Blackberry	207.9	0.05
Other	530.4	0.12
Total	**431,539.3**	**100**

The subsequent subchapters about Android and iOS provide a comprehensive overview of the corresponding operating system, shedding light on their respective role on the market, their licensing approach, the underlying software architecture, primary programming languages used, as well as the distinct process for application publication for each platform.

2.1.1. Android

Android operating system (OS) is an open-source operating system developed and promoted by Google and the global leader of mobile platforms, operating on various mobile device brands (Annuzzi et al., 2016, p. 51; Google, n.d.-a). Opensignal (2015) states in their Android fragmentation report nearly 1300 distinct Android device manufacturers. Even though Google promotes its own mobile device brand, Google Pixel, a substantial number of Android device manufacturers are notably more popular (Chebbi, 2019). According to Statcounter Global Stats (2022b), as of 2022, the three most relevant Android device brands in terms of worldwide vendor market share are Samsung, Xiaomi, and Huawei. Among these, Samsung stands out as the clear market leader among Android device brands, with a global market share of approximately 28% (Statcounter Global Stats, 2022b).

The Android Open Source Project (AOSP), managed by Google, serves as a foundation for the Android OS development, enabling the creation of custom variants (Google, n.d.-a). While the AOSP provides Android OS source code and compatibility guidance for device manufacturers, it does not cover most proprietary Google applications. Device manufacturers have the option to obtain a license for Google Mobile Services, such as Google Play, Gmail, Google Maps, and YouTube, to include on their Android-compatible devices (Annuzzi et al., 2016, p. 54). The implementation of Android OS provided by AOSP is built upon a layered architecture. These layers are visually depicted in Figure 1 and subsequently concisely outlined based on the architecture overview of the official Android OS documentation (Google, 2023b).

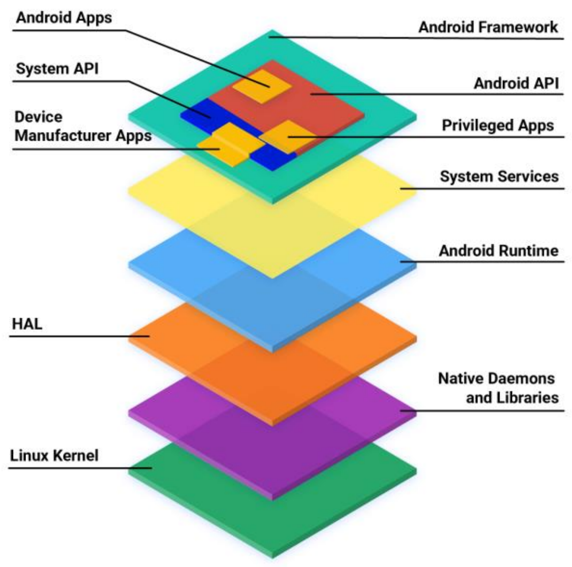


Figure 1: *AOSP software stack architecture*
(Source: Google, 2023b)

The kernel is the core component of an operating system that establishes communication and coordination with the device hardware. AOSP employs a Linux kernel, which is modularized into hardware-independent and vendor-specific components. Native daemons and libraries directly interact with the kernel or other interfaces, operating without relying on services provided by the Hardware Abstraction Layer. The Hardware Abstraction Layer offers a standardized interface for hardware vendors, abstracting the low-level hardware details to enable a hardware-independent implementation. Android runtime is AOSP's Java runtime environment which is responsible for translating an application's bytecode into device-specific instructions (Google, 2023b). A major feature of Android runtime is ahead-of-time compilation, i.e. compiling applications already during installation instead of preceding the start of the application, to reach improved performance (Android, 2022; Annuzzi et al., 2016, p. 67). System services are modular components responsible for fundamental tasks such as system coordination, graphic rendering, and multimedia handling. System services are accessed by applications through the Android framework API to utilize the underlying hardware resources of the device. Android framework consists of Java components designed for building applications. These components operate within each respective application's process. Certain elements are publicly accessible via the Android API, while other components are solely available to authorized partners or device manufacturers through system APIs for integrating certain capabilities into their devices. In general, applications running on Android devices fall in one of the three distinct categories: Android applications, privileged applications, or device manufacturer applications. Android applications are created using the Android API and are commonly acquired by downloading them from the Google Play Store. Privileged applications are built using both Android and system APIs and require preinstallation on the device. Device manufacturer applications are developed using the Android API, system API, and direct Android framework access. They also require preinstallation and, additionally, application updates are dependent on software updates of the device system (Google, 2023b).

To develop Android applications, the Android development toolkit is used, which contains the required Software Development Kit (SDK), debuggers, and emulators (Chebbi, 2019). Regarding the IDE, officially recommended and also widely used is Android Studio, while other well-liked alternatives include Apache Netbeans and IntelliJ IDEA (Chebbi, 2019; Google, n.d.-b). The Android development environment is compatible with Windows, Mac, and Linux operating systems. It is important to introduce the two primary programming languages for Android application development: Java and Kotlin (Chebbi, 2019; Google, 2023a). Originally, Java was the default programming language for Android application development since the introduction of the Android platform in 2008 (Chebbi, 2019). However, it is worth noting that Kotlin has been gaining significant popularity in the Android development since Google declared it to the official programming language and preferred choice for Android programming in 2019 (Chebbi, 2019; Google, 2023a). This means Google prioritizes the development and design of APIs, tools, and resources with a focus on Kotlin users' needs, while still ensuring compatibility and support for Java users (Google, 2023a). In *Android's Kotlin first approach*, Google (2023a) states the reasons for this choice as being Kotlin's concise and expressive nature, code safety, interoperability, and structured concurrency.

After completing the development of an Android application, the topic of publishing the application typically arises. According to Google (2023c), publishing an Android application involves a multi-step process. The first step is preparing the application for release which involves configuring settings, disabling logging, specifying version details, followed by building, signing, and thorough testing the release version on diverse devices, updating resources, and ensuring external server readiness. For the application release, the generation of promotional materials such as screenshots and videos are advised. Application distribution through marketplaces allows for broad audience reach, with Google Play being the prominent marketplace choice for Android. To release the application to Google Play, a configuration of global parameters is required, followed by the uploading resources along with a draft version of the application. After completing these steps, the publication can be initiated, and the application will be published after having passed Google Play review. Alternatively, the application can be published on an independent website in the form of an APK file, although this entails limitations on monetization and licensing. This publication alternative will most likely result in losses in distribution and, furthermore, users are less likely to trust it as they have to explicitly consent to the installation of applications downloaded outside trusted marketplaces (Google, 2023c).

2.1.2. iOS

The mobile platform iOS is developed by Apple Inc. to run exclusively on its proprietary hardware, with iPhones being Apple's developed and manufactured line of smartphones (Apple Inc., 2023b). While iOS as a mobile platform does not dominate the global market, iPhone, as a device brand, shares the position of market leader with Samsung, each holding a global market share of approximately 28% as of 2022 (Statcounter Global Stats, 2022b). Nevertheless, iOS still asserts its dominance in the smartphone market within the United States, controlling a 57% market share in 2022 (Statcounter Global Stats, 2022a).

The license approach of iOS is characterized by a proprietary and closed ecosystem maintained by Apple Inc (Chebbi, 2019). This strategy involves designing company-owned software and hardware components and only selectively supporting internet technologies in order to create a closed and controlled ecosystem for their products (Foremski, 2010). For iOS, this indicates that it is a proprietary OS which is owned, controlled, and maintained exclusively by Apple, not being open to public inspection or modification (Gartner Inc., n.d.). While iOS itself is not open-source, Apple is actively involved in open-source projects and still provides access to certain open-source components, for instance with releasing portions of its code used in macOS, iOS, and Developer tools on GitHub (Apple Inc., 2023d; Peterson, 2021). Moreover, Apple actively supports the development community by releasing featured open-source projects, such as Swift (programming language) and WebKit (web content engine), which find application within the iOS ecosystem (Apple Inc., 2023d).

For building Native iOS applications, developers utilize the iOS SDK, which provides the essential tools, libraries, and frameworks that are specifically designed to run on iOS devices (Apple Inc., 2012, p. 7). The architecture of iOS technologies can be regarded as a layered set presented in Figure 2. The lower layers provide the foundational services and technologies upon which all applications depend, while higher-level layers encompass more sophisticated elements (Apple Inc., 2012, p. 8).

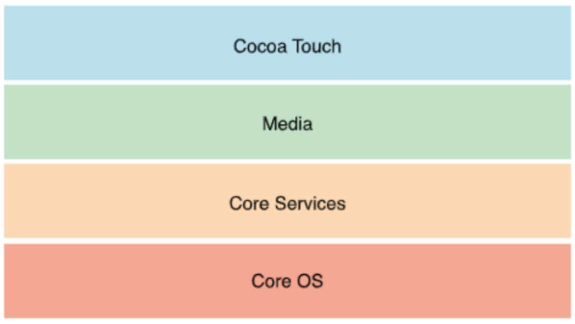


Figure 2: *iOS architecture layers*
(Source: Apple Inc., 2012, p. 8)

Beginning from the basis, the *Core OS* layer provides essential low-level features such as the Security framework along with the system-level kernel environment and hardware drivers (Apple Inc., 2012, pp. 43-45). It is worth noting that iOS overall relies on Darwin OS, which serves as Unix-based foundation and incorporates the so-called XNU kernel (Apple OSS Distributions, 2023). The *Core Services* layer builds upon the foundational infrastructure of the *Core OS* layer, delivering essential system services utilized by all applications. It contains *Core Services* Frameworks for tasks such as data management, contact management, geolocation, and system configuration (Apple Inc., 2012, pp. 30-42). The Media layer is dedicated to graphics, audio, and video technologies, enhancing the multimedia experience on the mobile device, with a focus on simplifying the creation of applications with impressive visual and auditory elements (Apple Inc., 2012, p. 21). Reaching the uppermost layer illustrated, the *Cocoa Touch* layer houses essential application frameworks, defining the basic iOS application infrastructure and facilitating key technologies such as multitasking, touch-based input, and push notifications (Apple Inc., 2012, p. 11). *Cocoa* is generally an application development environment by Apple for both iOS and macOS (formerly OS X), the OS for Apple's Macintosh computer series. It offers object-oriented libraries, a runtime system, and an integrated development environment (IDE) (Apple Inc., 2013). Numerous frameworks from *Cocoa* are shared between macOS and iOS, with the main distinctions primarily at the user interface (UI) level (Apple Inc., 2012, p. 9).

To develop iOS applications, an Intel-based Macintosh computer is required. Additionally, Apple recommends its IDE Xcode created for software development on Apple platforms. Xcode tools include project management, code editing, debugging on virtual and physical devices, executable building, version control, and performance optimization (Apple Inc., 2012, p. 55). To effectively code iOS applications, the primary programming languages Swift and Objective-C are used. Objective-C was Apple's initial language for iOS app development, which remained stable and mature over the years, albeit criticized for its verbose syntax. However, Apple’s introduction of Swift in 2014, followed by major improvements, led to Swift surpassing Objective-C as the preferred language for Native iOS app development due to its simplicity and compactness. While Objective-C and Swift can coexist, Swift has become Apple's default choice for iOS app development (Chebbi, 2019).

To publish an iOS application involves a structured workflow (Apple Inc., 2023c). As a prerequisite, developers have to be enrolled in the paid Apple Developer Program in order for them to distribute their iOS applications on Apple's marketplace, the App Store (Apple Inc., 2012, p. 9; 2023a). It begins with selecting the specific build of the application intended to submit for review. Next, the pricing and tax category for the application needs to be determined, with the option to target specific regions or offer pre-orders. The most crucial step involves submitting the application for review, where all required metadata is provided and the application release method (manual, automatic, or phased) is chosen. While the application is waiting for the review, any issues that arise during the review process must be addressed by reading and responding to App Review communications. After the application is approved, promotion codes can be requested, which can be distributed to users before the application's official release, allowing them to redeem these codes when purchasing the application (Apple Inc., 2023c). The aforementioned enrollment in the Apple Developer Program comes with, besides the application distribution, further exclusive benefits that, for instance, gain developers access to beta software and testing services, application development resources, application analytics, technical support resources, and developer forums, to name a few (Apple Inc., 2023a).

2.2. Cross-Platform Mobile Development

CPMD is still a relatively young and fast-paced field of research (Biørn-Hansen, Grønli, & Ghinea, 2018, pp. 2, 28). It refers to mobile development suited to run across multiple mobile platforms, such as Android and iOS (Heitkötter et al., 2013, p. 299). This chapter entails a comprehensive overview of CPMD, establishing a clear distinction from Native mobile development, outlining different CPMD development approaches, and introducing different concrete CPMD solutions that are viewed as relevant at the present time of writing this thesis.

2.2.1. Cross-Platform vs. Native

When developing applications for multiple mobile platforms, technically three options are available, which are outlined by Heitkötter et al. (2013, pp. p. 301-304) within the scope of their comparison of cross-platform approaches for mobile applications. The first option involves developing Native applications for each platform separately. The second possibility is to create a mobile web application using standardized web technologies. The third alternative is to use CPMD frameworks / technologies, which allows the building of a single application that can be deployed across multiple mobile platforms. Given that this thesis focuses on mobile applications that are downloaded and installed on the mobile device, the second option of mobile web applications is not taken into consideration. Since mobile web applications are implemented as a website optimized for mobile devices and are therefore accessed through the device's web browser, fundamentally different technical considerations apply in comparison to installable mobile applications (Heitkötter et al., 2013, pp. 301-302). This leads to the two remaining options, Native and CPMD, which will be contrasted with each other in the succeeding paragraphs.

Native mobile development utilizes platform-specific SDKs along with the associated development tools (Majchrzak et al., 2015, p. 6). Individual to each mobile platform, specific skills are required, such as the programming language and development environment used. For instance, Android developers need expertise in Kotlin / Java and Android Studio, whereas iOS developers have to be proficient in Swift / Objective-C and Xcode (Biørn-Hansen, Majchrzak, & Grønli, 2018, p. 2; Majchrzak et al., 2015, p. 6). Thus, development is performed separately for each platform, which means that the development effort increases proportionally with each supported platform added. Among typical software engineering activities, only requirements engineering is shared in this case, while design, implementation, and testing are conducted independently, as separate codebases exist for each mobile platform (Biørn-Hansen, Grønli, & Ghinea, 2018, p. 2; Majchrzak et al., 2015, p. 6).

In response to this challenge, there has been a growing exploration of alternative development techniques and tools that aim to complement or even replace the Native approach in order to simplify the development process and enhance compatibility across various platforms (Biørn-Hansen, Grønli, & Ghinea, 2018, p. 2). In an often-cited study by Xanthopoulos and Xinogalos (2013, p. 1), the authors state that "The ultimate goal of cross-platform mobile app development is to achieve native app performance and run on as many platforms as possible." Biørn-Hansen, Grønli and Ghinea (2018, p. 2) identify "budget, time constraints, and knowledge considerations" the most common reasons listed in literature for favoring cross-platform approach over Native. Mobile applications seldom target one single mobile platform, but if they do, Native is the obvious choice. In most cases, at least Android and iOS are supported though. When developing for more than one mobile platform, it is advisable to analyze if a CPMD solution is the superior alternative over Native, when prioritizing time and effort (Shah et al., 2019, p. 2). In contrast to Native, cross-platform development aims for a so-called "write once – run anywhere" (WORA) principle (Dalmasso et al., 2013, p. 324; Huynh & Ghimire, 2017, p. 51; Shah et al., 2019, p. 2; Xanthopoulos & Xinogalos, 2013, p. 214), also called "implement once - run everywhere" principle by Majchrzak et al. (2015, p. 6). The idea of WORA was originally created by Sun Microsystems in the mid-1990s to highlight the platform independency of the Java Platform (Huynh & Ghimire, 2017, p. 51) and is now continuously used in the cross-platform development context, for instance Dalmasso et al. (2013, p. 324) and Shah et al. (2019, p. 2) refer to CPMD tools as WORA tools. Another alternate term for cross-platform development is multi-platform development (Biørn-Hansen, Grønli, & Ghinea, 2018, p. 2). However, Biørn-Hansen, Grønli and Ghinea (2018, p. 2) acknowledge cross-platform development as the predominant term aligned with both literature and industry. For CPMD, various development approaches with different characteristics exist for creating mobile applications. These approaches are elaborated upon in Section 2.2.2. Before delving into these approaches, the advantages and disadvantages of CPMD in contrast to Native are discussed.

The most obvious advantage and difference of CPMD, as already mentioned before, is the use of a single code base. With that, the application can be deployed across multiple platforms, while Native requires multiple platform-specific code repositories. Consequently, CPMD with one programming language facilitates a straightforward development, whereas Native often needs to rewrite and customize code for different platforms. Following the same logic, when having only one development workforce dedicated to a single CPMD technology, an effective cost reduction can be achieved in comparison to Native with multiple development groups for various platforms (Shah et al., 2019, p. 2). It should still be taken into account, that the integration of CPMD also involves additional time and effort for combining components and adapting applications to platform-specific settings as well as locating problems in the reusable code (Farooq et al., 2022, p. 89). Going further, subsequent updates and bug fixes can be simultaneously rolled out across all platforms with CPMD, instead of asynchronous, platform-specific application revisions (Shah et al., 2019, p. 2). Conversely, Farooq et al. (2022, p. 89) highlight that such a consolidated update mechanism across different platforms can be a drawback because it necessitates all platforms to have the same set of features, potentially limiting the flexibility of individual platforms. Additionally, Shah et al. (2019, p. 2) points out the engaged open-source communities of leading CPMD tools as an advantage. Zohud and Zein (2021, p. 53), on the other hand, identifies limited support from the community of the CPMD tool as a challenge. Furthermore, Biørn-Hansen, Grønli, et al. (2019, p. 9) point out that one of the most common issues of CPMD is the maturity of the technical frameworks. Moving from application development to application use, Native applications undeniably have the smoothest performance for their designated platforms and significantly exceed the performance of cross-platform applications. This leads, consequently, to a superior user experience delivered by native applications over cross-platform applications (Biørn-Hansen, Grønli, et al., 2019, p. 9; Farooq et al., 2022, p. 89; Shah et al., 2019, p. 2). A major setback for CMPD is that issues such as crashes and freezing become more prevalent, as the complexity of applications increases, therefore hindering robust device support (Shah et al., 2019, p. 2). It is worth mentioning that, for small applications, the performance differences between CPMD and Native are not significant and most likely not noticeable (Farooq et al., 2022, p. 89). Furthermore, Native applications benefit from deeper device integration, while accessing device features with CPMD is more limited (Shah et al., 2019, p. 2; Zohud & Zein, 2021, p. 53). Another advantage of CPMD is that applications developed with this approach can face deployment challenges on specific application stores, potentially receiving lower priority and even rejection if they predominantly employ HTML and JavaScript injection within a thin Native container (Shah et al., 2019, p. 2). Other challenges in CPMD, identified by Zohud and Zein (2021, p. 53), arise from the reliance on community-provided plugins and manual testing techniques. Naturally, some of the advantages and disadvantages can vary depending on the chosen cross-platform development approach, which are individually portrayed in the upcoming Section 2.2.2.

2.2.2. Development Approaches

CPMD encompasses a variety of development approaches. Several authors consider traditional mobile web applications accessed through browsers as a standalone development approach of CPMD (Farooq et al., 2022, pp. 86-87; Nunkesser, 2018, p. 216; Pinto & Coutinho, 2018, pp. 670-671; Raj & Tolety, 2012, p. 626; Shah et al., 2019, pp. 2-3; Stanojević et al., 2022, p. 215; Xanthopoulos & Xinogalos, 2013, p. 215). As previously mentioned in Section 2.2.1, this approach is not considered further in this thesis due to the focus on installed mobile applications. However, the novel approach of progressive web applications (PWAs) described by Biørn-Hansen, Grønli and Ghinea (2018, p. 73) is worth portraying because it enables installable applications. The most common identified CPMD approaches besides web applications are hybrid, interpreted, and cross-compiled development approaches (Farooq et al., 2022, pp. 86-89). Already Raj and Tolety (2012, pp. 626-628) outline these approaches and Farooq et al. (2022, pp. 85-86), even though identifying additional approaches, still classify them as the most utilized approaches, which makes them most relevant for practical use. Biørn-Hansen et al. (2020, p. 3003), El-Kassas et al. (2017, p. 168), and Farooq et al. (2022, p. 85) identify the compilation approach as the generic approach, encompassing both the cross-compiled and trans-compiled approaches. In the majority of the literature, researchers mostly refer only to the sub-approach *cross-compiled*, making it predominant and more relevant to practice than grouping it with the trans-compiled approach (Biørn-Hansen, Grønli, & Ghinea, 2018, pp. 8-9; Farooq et al., 2022, pp. 88-89; Raj & Tolety, 2012, pp. 626-627). The last relevant approach presented is the model-driven approach, already described by Xanthopoulos and Xinogalos (2013, p. 216) and further elaborated upon by Heitkötter et al. (2015, pp. 32-33) and Majchrzak et al. (2015, pp. 6-7). Additional approaches are discussed in academic papers but have limited practical relevance. Such approaches include component-based, cloud-based, and merged approach categorized by Farooq et al. (2022, p. 85) and widget-based approach explained by Shah et al. (2019, p. 5). Stanojević et al. (2022, p. 490) contrasts the traditional categorization of CPMD approaches with a novel categorization method proposed by Nunkesser (2018, pp. 215-217) where the various approaches are classified into pandemic, endemic and ecdemic applications. While this represents an interesting alternative method for categorizing CPMD approaches, for the purposes of this paper, the well-established categorization was chosen in order to proceed with the more detailed description of the introduced CPMD development approaches. In Table 2, the key characteristics of the CPMD approaches, derived from the detailed explanations below, are summarized.

Table 2: *Key characteristics of development approaches summarized*

	PWA	**Hybrid**	**Interpreted**	**Cross-compiled**	**Model-driven**
Codebase	web-based		common		DSL
Deployment	downloaded	packaged		generated	
Execution	Offline within web browser	Within Native application container		As Native application	
Platform integration	Service Workers	Bridging via WebView	Bridging via on-device interpreters	-	
Native adaption	No native elements	Native packaging	Native GUI	Native application	

Table 2 lists the most common development approaches, which are discussed in more depth in this section, along with their development infrastructure approach and platform adaption characteristics. The listed development approaches are chosen to be the most relevant based on the studies of Biørn-Hansen, Grønli and Ghinea (2018, pp. 5-11), Biørn-Hansen et al. (2020, pp. 3001-3003), Farooq et al. (2022, pp. 86-89), and Xanthopoulos and Xinogalos (2013, pp. 215-216).

Progressive web applications (PWAs) are web applications with advanced capabilities (Biørn-Hansen, Grønli, & Ghinea, 2018, p. 10). They do not rely on Native elements and are built using web technologies to achieve Native-like features and design. Biørn-Hansen, Grønli and Ghinea (2018, p. 10) describe PWAs as a novel approach, with Biørn-Hansen, Majchrzak and Grønli (2018, p. 66) contributing a valuable introductory publication on this particular approach. Like traditional web applications, they are hosted on web servers and accessed via a web browser. What sets PWAs apart from traditional web applications is providing an application-like experience by mimicking the look and functionality of Native or cross-platform applications. This is established by being installable on users' devices, enabling offline usage, and eliminating standard browser elements, such as the web address bar, settings icon or similar. This way, only the website's content is displayed, and it is not visible to users that the application operates within a browser. When designing the UI of PWAs, similar techniques to hybrid approaches are used, mainly utilizing HTML and CSS for structure and style. Transforming a traditional web application into a PWA involves integrating Service Workers, a JSON-based manifest file, and a set of static UI components, collectively referred to as the Application Shell, managing the application's lifecycle and background processing (Biørn-Hansen, Grønli, & Ghinea, 2018, p. 10). Biørn-Hansen, Majchrzak and Grønli (2018, pp. 81-82) conclude that PWAs hold great potential but are still in an early stage of maturity and have not yet gained widespread adoption in the industry.

In the **hybrid approach**, web assets such as HTML, CSS, and JavaScript are leveraged to create UIs and business logic. It involves packaging web technologies into a Native application, functioning as a wrapping container. This streamlines the process by generating a Native application with an embedded WebView component. This WebView enables the execution and rendering of web code within the Native application environment. Bridging functionality allows communication between the WebView and Native code, enabling access to platform-specific features to achieve Native-like user experiences (Biørn-Hansen, Grønli, & Ghinea, 2018, pp. 5-7; Pinto & Coutinho, 2018, pp. 671-672; Shah et al., 2019, pp. 3-4; Xanthopoulos & Xinogalos, 2013, p. 215).

The **interpreted approach** enables the creation of applications using interpreted scripting languages, most commonly JavaScript. In contrast to the hybrid approach, interpreted applications do not rely on WebView and instead render actual Native UI components. This is facilitated through on-device JavaScript interpreters. Bridging techniques, facilitated by platform-specific JavaScript engines, enable communication with Native code layers to access device features, thus facilitating the generation of a Native GUI. A variety of interpreted CPMD technologies exist, resulting in inoperability of plugins and, consequently, fragmentation of developer communities (Biørn-Hansen, Grønli, & Ghinea, 2018, pp. 7-8; Shah et al., 2019, pp. 4-5; Xanthopoulos & Xinogalos, 2013, p. 215).

The **cross-compiled approach** differs from other development approaches by utilizing compilers instead of interpreters or WebView components. CPMD technologies of this approach compile a common language like C# into Native bytecode executable on the targeted platforms, eliminating the need for a bridging layer. Instead, access to Native device features is facilitated through the CPMD technology's SDK, which directly maps functionality to the underlying platform-specific SDK. This approach ensures that the UI is rendered as Native interface components, enhancing performance and consistency across platforms. Disagreements exist regarding which frameworks belong to this approach, with Xamarin being cited as an example due to its compilation of a common language into Native bytecode (Biørn-Hansen, Grønli, & Ghinea, 2018, pp. 8-9; Xanthopoulos & Xinogalos, 2013, pp. 215-216).

The **model-driven approach** facilitates the generation of UIs and business logic based on constructed models and templates. Model-driven CPMD technologies typically employ a domain-specific language (DSL) to enable both developers and non-developers to build software without requiring knowledge of platform-specific programming languages. These frameworks use generators to convert models or code written in the DSL into Native source code for targeted platforms. While model-driven CPMD technologies vary in their integrated functionality and DSL design, they share the goal of empowering non-technical users to model applications (Biørn-Hansen, Grønli, & Ghinea, 2018, pp. 9-10). Biørn-Hansen, Grønli, et al. (2019, p. 9) highlight the disparity between model-driven frameworks used in academic research and those adopted by practitioners, emphasizing the absence of model-driven CPMD technologies in the industry.

2.2.3. Technologies

After the explanation of the most relevant development approaches, this section delves into their practical application through concrete CPMD technologies. Table 3 provides a mapping of development approaches currently most relevant in practical applications followed by corresponding examples of CPMD technologies. The listed mapped technologies were selected from an exhaustive overview by Biørn-Hansen, Grønli and Ghinea (2018, p. 3) to provide the necessary background context for their subsequent appearance in Chapter 4. As previously mentioned, KMP also plays a crucial role in this thesis. However, due to its unique characteristics, KMP cannot be assigned in the mapping table, as it does not fully comply with any development approach.

Table 3: *Mapping of CPMD technologies to development approaches (Source: Biørn-Hansen, Grønli, & Ghinea, 2018, p. 3)*

Hybrid	Interpreted	Cross-compiled
Ionic	React Native	Flutter
Cordova / PhoneGap	NativeScript	Xamarin
	Fuse	
	Titanium	

Both hybrid technologies Ionic and Cordova / PhoneGap employ web technologies, such as HTML, CSS, and JavaScript, in order to access Native device functionality and package applications for multiple platforms. Adobe PhoneGap and Apache Cordova are closely related CPMD platforms, initiated by the company Nitobi in 2008 and later acquired by Adobe. Both frameworks utilize a WebView for UI rendering and enable access to device functionalities through JavaScript plugins. While Adobe PhoneGap includes proprietary features such as the PhoneGap Build service, Apache Cordova remains true to its open-source roots (Farooq et al., 2022, p. 87; Pinto & Coutinho, 2018, p. 672). Ionic, founded in 2012, offers a rich library of pre-built components and device plugins, leveraging Apache Cordova and Angular (Pinto & Coutinho, 2018, pp. 672-6773). The interpreted technologies React Native by Facebook, NativeScript by Telerik, Fuse by Fusetools, and Titanium by Appcelerator utilize an on-device JavaScript interpreter to produce Native UIs, without any dependency on a web browser (Biørn-Hansen, Majchrzak, & Grønli, 2018, p. 65; Farooq et al., 2022, p. 88). Appcelerator Titanium is a free framework that offers a Model-View-Controller development kit called Alloy, which provides APIs for accessing local UI components and Native device functionality (Farooq et al., 2022, p. 88). React Native is a JavaScript framework built upon ReactJS, initially developed by Facebook to extend the capabilities of web-based interfaces to mobile devices. With its focus on reusability and performance, React Native facilitates CPMD with a high percentage of shared code, utilizing familiar JavaScript syntax for a familiar developer experience similar to web development (Farooq et al., 2022, p. 88; Pinto & Coutinho, 2018, p. 673). The React Native platform officially recommends Expo as the most straightforward start into CPMD using React Native, providing a suite of tools and services centered on React Native (Meta Platforms Inc., n.d.). Expo is an open-source framework for CPMD with React Native, supporting Android, iOS, and web platform. It is supplemented by Expo Application Services, which offer integrated cloud services for building, submitting, and updating React Native applications (Expo, n.d.). Both cross-compiled technologies Flutter and Xamarin implement their own UI component for optimal performance (Farooq et al., 2022, pp. 88-89). Xamarin was founded in 2011 by engineers behind the open specification and technical standard Common Language Infrastructure and was later acquired by Microsoft in 2016. Leveraging a common C# codebase, Xamarin enables the building of Native Android, iOS, and Windows applications with shared codebases. While Xamarin promotes code reuse across platforms, it may require custom code to leverage certain platform-specific capabilities. Significant code reuse, up to 75%, can be achieved, positioning Xamarin as a key platform for CPMD within the Microsoft ecosystem (Farooq et al., 2022, pp. 88-89; Pinto & Coutinho, 2018, p. 672). Flutter is a cross-platform SDK created by Google, designed to simplify the process of building high-performance Native applications for Android, iOS, desktop, and web platforms. Utilizing the Dart programming language, Flutter offers features like Hot Reload for faster development and testing, while providing access to third-party SDKs, APIs, and Native design widgets (Farooq et al., 2022, p. 88).

KMP is an open-source technology by JetBrains designed to streamline cross-platform development by enabling efficient code sharing across multiple platforms, while retaining the benefits of Native programming (JetBrains, 2023b). Initially introduced at KotlinConf 2017, KMP remained in an experimental phase until 2020, during which it was continually shaped by user feedback. Subsequently, the focus of KMP was shifted to code sharing between iOS and Android, recognizing them as the primary use cases (Petrova, 2023). However, it still also supports other platforms, such as desktop, web, and server-side. Compose Multiplatform, a Kotlin-based UI framework, further enhances the CPMD by facilitating the sharing of UI components between Android and iOS, ensuring fully cross-platform application creation. Additionally, KMP supports the development of multiplatform libraries, enabling developers to publish common code with platform-specific implementations for JVM, web, and Native platforms (JetBrains, 2023a, 2023b, n.d.). Compose Multiplatform is currently stable for Android and desktop platforms, with plans to promote iOS support to Beta and web support to Alpha in 2024, though specific stabilization dates have not been announced (JetBrains, n.d.). KMP does not directly fit into any of the outlined development approaches in Section 2.2.2. Although it shares similarities with cross-compilation by compiling code to Native applications, KMP distinguishes itself by leveraging the capabilities of the Kotlin language and compiler to enable multiplatform development. Therefore, KMP can be regarded as its own approach to cross-platform development.

As of the Stack Overflow Developer Survey 2022, Flutter and React Native were the two most popular cross-platform tools, followed by Ionic and Xamarin (Stack Overflow, 2022). The outcomes of the literature review analyzing CPMD comparison studies in Section 4.1 confirm the popularity of these frameworks within academic research as well. In this thesis, the two CPMD technologies React Native and KMP were selected for evaluation. React Native was chosen due to its widespread adoption and established presence in the CPMD landscape, making it one of the most predominant frameworks for CPMD. Both Biørn-Hansen, Grønli, et al. (2019, p. 7) and Zohud and Zein (2021, p. 53) concur that React Native stands out as promising and prevalent CPMD technology in the industry. In contrast, KMP was included as it represents a newer and emerging technology in the CPMD ecosystem, offering unique advantages and potential benefits.

2.3. AI-Assisted Coding

AI-assisted coding represents the use of artificial intelligence (AI), typically large language models (LLMs), to assist in code writing within the domain of computer science (Murillo & D'Angelo, 2023, p. 1). In a recent research paper published by Google, Murillo and D'Angelo (2023, p. 1) emphasize that AI-assisted coding is an emerging paradigm which has the potential to significantly influence the traditional approach to code creation from an engineering perspective. The use of AI-powered code writing assistants aims to enhance development efficiency and productivity by the reduction of workloads, task time, and human errors. The integrated use of AI is additionally reshaping coding approaches with acceleration for quick execution and exploration for planning potential paths (Murillo & D'Angelo, 2023, p. 1). Sergeyuk et al. (2024, p. 1) also points out that the integration of AI into IDEs is anticipated to shift from aiding in coding to providing intelligent insights and recommendations. This evolution of human-AI experience envisions AI as more than a tool but as a collaborative partner, fundamentally altering how users engage with and derive benefits from technological systems (Sergeyuk et al., 2024, p. 1). Calegario et al. (2023, p. 32) likewise recognize the potential of AI-assisted coding in boosting productivity and enhancing quality in software development. Specifically, the beneficial assistance encompasses automating repetitive tasks, generating artifacts, optimizing code, and delivering intelligent assistance throughout the key stages of the software development life cycle. Calegario et al. (2023, p. 33) underscores the importance of methodologies such as prompt engineering, specialized fine-tuning, UX design, and seamless collaboration between humans and AI for achieving optimal results. Becker et al. (2023, p. 502) acknowledges promising opportunities arising from AI-assisted coding, acting as a valuable tool for onboarding users to unfamiliar codebases, enabling non-programmers to craft specifications and make programming more accessible overall. From an educational perspective, a more effective way of learning and teaching coding could be realized with individual code solutions and reviews, generated learning resources and explanations, and new pedagogical approaches, including a focus on higher-level algorithms due to a reduced syntax complexity (Becker et al., 2023, pp. 503-504). In contrast, Vaithilingam et al. (2022, p. 1) did not necessarily observe improvements in task completion time or success rates with AI-assistance in place, but users widely preferred integrating it into their daily programming workflow. According to Vaithilingam et al. (2022, p. 1) as well as Murillo and D'Angelo (2023, p. 1), the AI-assisted coding process also faced challenged related to understanding, editing, and debugging generated code snippets, which restricted task-solving effectiveness and introduced potential difficulties compared to human-written code. Becker et al. (2023, pp. 504-505) highlight several challenges associated with AI-assisted coding, including ethical concerns such as academic misconduct, ownership and licensing issues, as well as sustainability concerns due to substantial energy consumption associated with training large AI models. Additionally, Becker et al. (2023, p. 505) point out challenges related to harmful bias and security issues, ant the risk of coding novices overly relying on AI.

Recent studies have explored various AI-powered code generation tools, including GitHub Copilot, ChatGPT, Amazon CodeWhisperer, DeepMind AlphaCode, Bard, Claude 2, and Tabnine (Becker et al., 2023; Calegario et al., 2023; Hochmair et al., 2024; Murillo & D'Angelo, 2023; Sakib et al., 2023; Vaithilingam et al., 2022; Yetiştiren et al., 2023). Another LLM for code generation is StarCoder developed by the BigCode community, trained on extensive GitHub data, excelling in code generation and surpassing other open LLMs across diverse programming languages (Li et al., 2023, p. 33). Throughout two different studies assessing various LLMs for code generation, ChatGPT consistently emerged as the top-performing assistance tool. Hochmair et al. (2024) examined the generative AI-chatbots ChatGPT, Bard, Claude 2, and Github Copilot for code-related tasks in the geo-science community. Evaluated tasks included coding, spatial computations, time-series forecasting, and toponym recognition. Generally, for all evaluated chatbots, weaknesses were identified in mapping, code generation, and code translation, while strengths were recognized in spatial literacy, GIS theory, interpreting programming code and functions. ChatGPT outperformed its opponents in most task categories, including code generation (Hochmair et al., 2024, p. 1). Yetiştiren et al. (2023) compared the code generation capabilities of GitHub Copilot, Amazon CodeWhisperer, and ChatGPT. Key findings include ChatGPT's higher success rate (65.2%) in generating correct code compared to GitHub Copilot (46.3%) and Amazon CodeWhisperer (31.1%). The study highlights improvement rates in newer versions of GitHub Copilot (18%) and Amazon CodeWhisperer (7%). Additionally, the average technical debt was measured, where ChatGPT has a slightly higher average technical debt (8.9 minutes) compared to Amazon CodeWhisperer (5.6 minutes), yet slightly lower than GitHub Copilot (9.1 minutes) (Yetiştiren et al., 2023, pp. 1-2). In another study by Sakib Sakib et al. (2023), investigating the effectiveness of ChatGPT in solving programming problems, it achieved a success rate of 71.875%. While it shows potential for revolutionizing code generation, challenges in refining solutions, especially in debugging tasks, were noted. The study underscores the necessity for continuous improvement to enhance ChatGPT's coding assistance capabilities (Sakib et al., 2023, p. 1). In the course of the maturation of generative AI and its deepening integration with software engineering, Calegario et al. (2023, p. 33) anticipate a promising future for developers collaborating with AI in creating impactful software.

3. Methodology

Unwrapping the methodological course, the research design serves as the framework for the study, guiding through the specifics of the applied research components, namely literature review, sample application development, and metric-based comparison.

3.1. Research Design

The research paradigm of this thesis adopts a mixed-methods case study design, combining both qualitative and quantitative research approaches. The qualitative component involves a comprehensive exploration and reflection on the AI-assisted coding experience. Moreover, the study integrates experimental elements into the broader research design, enabling quantitative performance measurement in the developed mobile sample applications built with different CPMD solutions. The case study is applied as the research approach because it is used “to generate an in-depth, multi-faceted understanding of a complex issue in its real-life context” (Sarah et al., 2011, p. 1). The thesis aims to provide a holistic understanding of the interplay between CPMD technologies, AI-assisted coding, and performance metrics for mobile applications. A case study design supports investigating these interconnected elements and achieving a contextual understanding of this specific, practical setting.

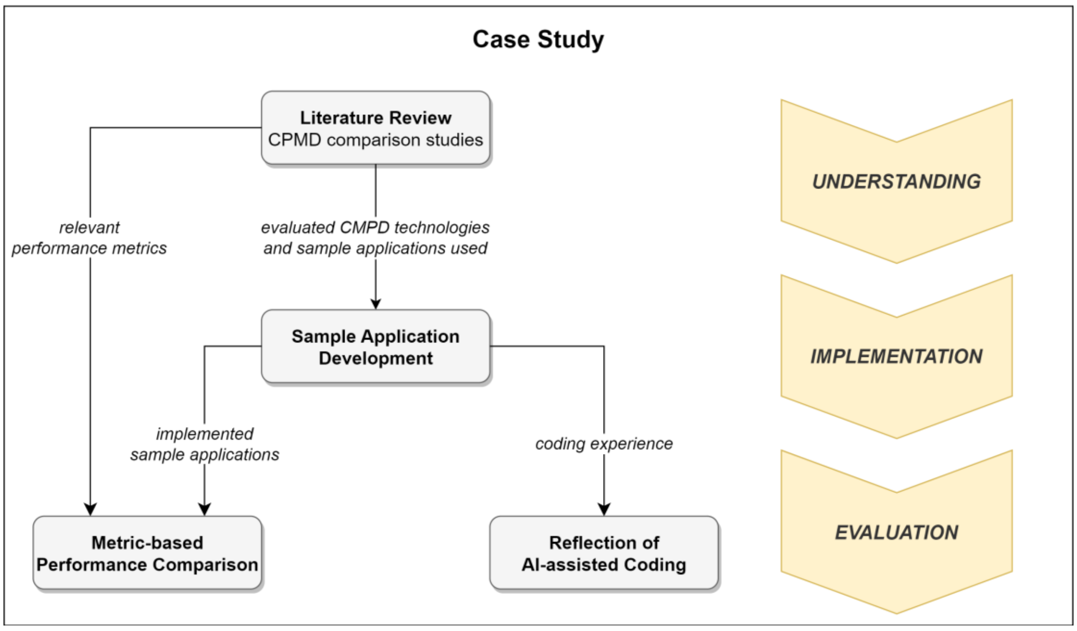


Figure 3: *Research Design*
(Source: Author’s illustration)

This study encompasses various components within the case study framework. A literature review offers an insightful overview of existing cross-platform comparison studies, while identifying appropriate performance metrics essential for evaluation. Next, the development phase involves the creation of sample applications, employing one with the Native approach and two more with distinct

CPMD technologies. This process integrates AI-assisted coding to enhance productivity. Subsequently, a metric-based comparison systematically contrasts the performance of the developed sample applications using predefined metrics. Lastly, the study includes a reflective analysis which critically examines experiences and outcomes related to the incorporation of AI-assisted coding in the development process. Figure 3 illustrates the described research design, which can be categorized into the three phases of understanding, implementation, and evaluation. The following sections provide detailed insights into the individual research steps, encompassing literature review (Section 3.2), sample application development (Section 3.3), and metric-based performance comparison (Section 3.4).

As this thesis employs the case study design, the thesis itself can be considered the case study report. The study follows the *process of building theory from case study research* outlined by Eisenhardt (1989, pp. 533, 536-545). Table 4 offers an overview of the process steps applied throughout this thesis, specifying the corresponding chapters where each step was discussed. These steps demonstrate how the thesis integrates qualitative and quantitative methods to investigate both the AI-assisted coding experience and performance evaluation in the context of mobile applications built with different CPMD technologies.

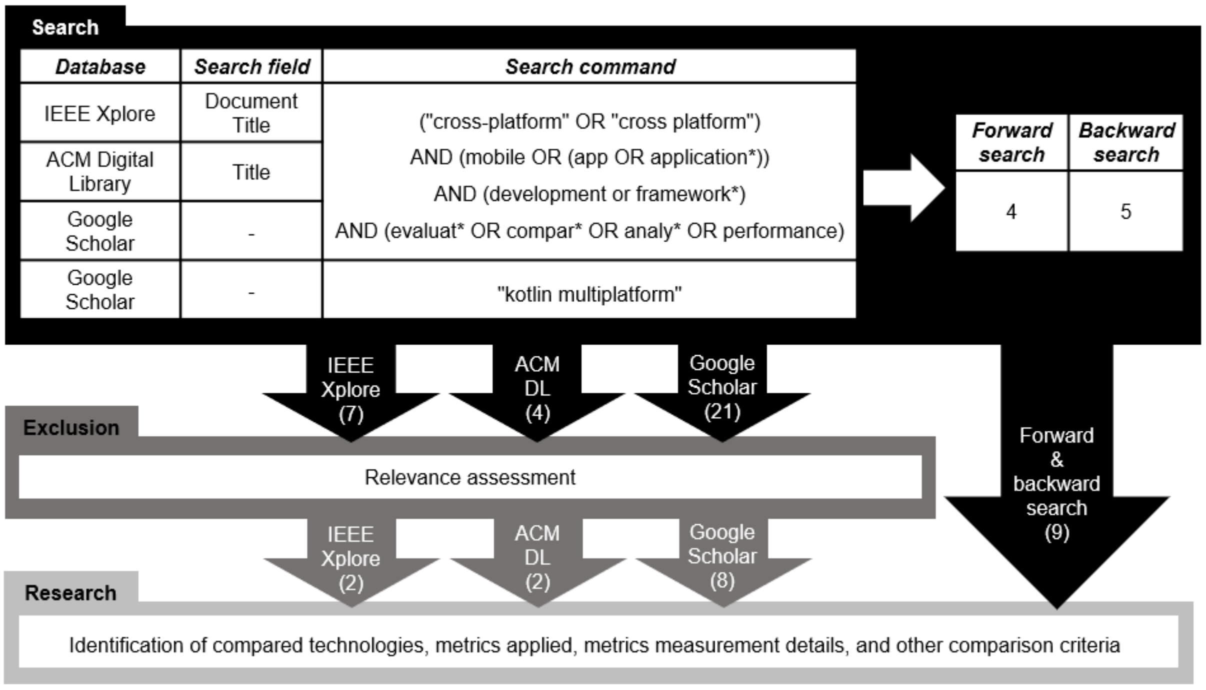
Table 4: *Case study research steps mapped to thesis*
(Adapted from Source: Eisenhardt, 1989, p. 533)

Step	Activity	Chapter
Getting Started	Definition of research questions A priory constructs: Theoretical Background	1 2
Selecting Cases	Specified, purposeful sampling: Mobile applications built with different CPMD technologies	3.3
Crafting Instruments and Protocols	Multiple data collection methods: - Quantitative assessment: Performance metrics - Qualitative insights: Overview of existing CPMD comparison studies, Reflection about AI-assisted coding	3.1 3.4 3.2 3.3
Entering the Field	Overlap data collection and analysis: Adapting data collection based on emerging and unforeseen trends	4–6
Analyzing the Data	- Within-case analysis: Examining development process and performance metrics of each CPMD technology individually - Cross-case pattern search: Identifying commonalities and differences in application development and performance across different CPMD technologies	5–7
Shaping Hypotheses	Search evidence for the why behind relationships: Investigating the underlying reasons behind observed performance differences or similarities	7
Enfolding Literature	Comparison with conflicting and similar literature regarding AI-assisted coding and performance of applied CPMD technologies	
Reaching Closure	Stop iterating between theory and data	8

3.2. Literature Review

In the context of this thesis, and with a specific focus on addressing RQ 2, a literature review was conducted, following a systematic search process advocated by Webster and Watson (2002, p. 16) and Brocke et al. (2009, pp. 9-10). Given the crucial role of documenting the literature search process, as highlighted by Brocke et al. (2009, p. 11), this subchapter provides a detailed account of the search methodology.

In the process of the literature review, the databases IEEE Xplore and ACM Digital Library (digital libraries for research articles in engineering and technology), as well as Google Scholar (academic search engine), were explored. Google Scholar was included in the search process due to its broader scope, as the outcomes in the two databases with technology focus were relatively limited. The search scope across all databases was constrained to publications published from 2018 to 2023. In IEEE Xplore and ACM Digital Library, the search was conducted for the title field. Elaborate details of the literature search process, including the precise search terms and publication counts at each stage, are available in Figure 4. The primary emphasis of the search was to pinpoint publications engaging in comparative studies of CPMD technologies. Additionally, a targeted exploration for Kotlin Multiplatform was undertaken, considering its limited coverage in the existing literature. Therefore, an extra search employing the term Kotlin Multiplatform in the arXiv database for scientific research articles pre-peer review was performed, but not included in the figure as it yielded no results.



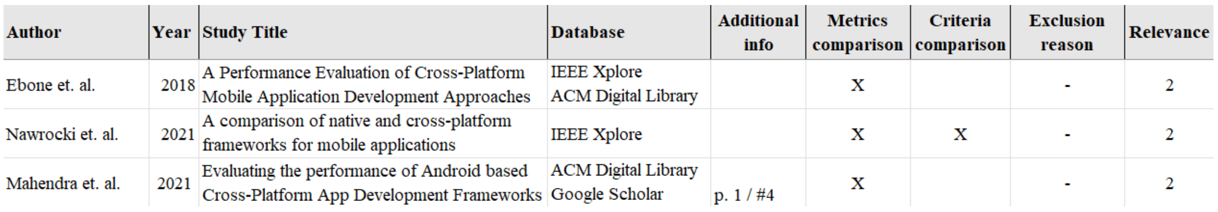
Database	*Search field*	*Search command*
IEEE Xplore	Document Title	("cross-platform" OR "cross platform") AND (mobile OR (app OR application*)) AND (development or framework*) AND (evaluat* OR compar* OR analy* OR performance)
ACM Digital Library	Title	
Google Scholar	-	
Google Scholar	-	"kotlin multiplatform"

Forward search	*Backward search*
4	5

IEEE Xplore (7)
ACM DL (4)
Google Scholar (21)
Exclusion
Relevance assessment
Forward & backward search (9)
IEEE Xplore (2)
ACM DL (2)
Google Scholar (8)
Research
Identification of compared technologies, metrics applied, metrics measurement details, and other comparison criteria

Figure 4: *Literature search process*
(Source: Author's illustration)

Applying the concept matrix by Watson and Webster (2020, pp. 17-18), the publications from the search results were listed in tabular form and sorted based on their respective relevance for this study. As this work focuses on the comparison of CPMD technologies, the publications were analyzed to determine whether the concepts *metrics comparison* and/or *criteria comparison* were addressed in the study, marked with a cross when applicable. For *metrics comparison*, a study compares CPMD technologies using quantifiable metrics, while for *criteria comparison*, additional criteria are used for comparison. If neither applies, a reason for exclusion was provided. A segment of the literature table used for relevance classification is shown in Figure 5, and the complete version can be found in Appendix B1.



Author	Year	Study Title	Database	Additional info	Metrics comparison	Criteria comparison	Exclusion reason	Relevance
Ebone et. al.	2018	A Performance Evaluation of Cross-Platform Mobile Application Development Approaches	IEEE Xplore ACM Digital Library		X		-	2
Nawrocki et. al.	2021	A comparison of native and cross-platform frameworks for mobile applications	IEEE Xplore		X	X	-	2
Mahendra et. al.	2021	Evaluating the performance of Android based Cross-Platform App Development Frameworks	ACM Digital Library Google Scholar	p. 1 / #4	X		-	2

Figure 5: *Concept Matrix segment*

The relevance assessment is based on a scale of 0 to a maximum of 2, determined by analyzing the covered topics:

- 0: Coverage of neither concept
- 1: Sole coverage of the *criteria comparison* concept
- 2: Sole coverage of the *metrics comparison* concept or coverage of both concepts.

The relevance was assessed this way because the concept *metrics comparison* is necessary to answer RQ 2. Duplicates and publications classified with relevance 0 were sorted out. In total, 21 out of the original 38 publications were excluded. Additionally, through forward and backward searches, 8 relevant publications were identified. Therefore, the literature for investigating the comparison studies in this work consists of 29 publications. These studies will be analyzed in detail in Chapter 4.

3.3. Sample Application Development

In order to comprehensively evaluate the performance and derive meaningful implications regarding the employed CPMD technologies, the creation of sample mobile applications is essential. These applications serve as practical real-world scenarios for the chosen technologies to be evaluated and provide controlled conditions for the systematic measurement and assessment of performance metrics. Three distinct sample applications were developed, using the following technologies:

1. Native Android
2. KMP
3. React Native.

Functioning as a baseline, a Native Android application implemented with Kotlin provides a reference point for assessing potential advantages or trade-offs when compared with CPMD technologies. KMP was selected as a young and innovative CMPD technology with a novel approach that has not been extensively established in practice, while claiming to preserve Native performance. As a prominent and widely established CPMD framework, React Native has been identified as currently one of the most popular and relevant options based on the insights gained from the literature review about CPMD comparison studies in Section 4.1.

As for the technical setup, version control and two distinct IDEs were in use. Android Studio, the officially recommended IDE for Android development, was employed for Native Android and KMP, offering seamless integration with KMP. Notably, besides Android, Android Studio facilitated the execution of the iOS application execution as well, given that Xcode was set up properly. Gradle is the build toolkit utilized in Android Studio, enabling automated and customizable build processes, with the

Android Gradle plugin providing specific configurations for building and testing the Android applications (Google, 2024b). For KMP, the Java Development Kit JDK, Cocoa Pods and the Android Studio plugin *Kotlin Multiplatform Mobile* had to be installed. The React Native application was developed using *Visual Studio Code*, a lightweight IDE by Microsoft that offers robust built-in support for JavaScript, aligning with the React Native framework. For React Native development setup, the JavaScript runtime environment *Node.js* and the Node Package Manager *npm* had to be installed. The Expo development could be started using the `npx expo` command. To enable development and ensure application executions for both platforms, a Macintosh device, specifically a *MacBook Pro*, was utilized due to its compatibility with iOS development requirements. Dedicated Git repositories, using GitHub, were established for each of the three sample applications, facilitating organized code management, version tracking, and functioning as a precautionary measure against data loss. Regular commits throughout the development lifecycle provided a granular overview of code changes and ensured a traceable development history.

All three sample applications meet identical application requirements, sharing a unified layout and set of functionalities. The application is designed with three main features, encompassing essential functionalities:

1. Currency conversion: Users can enter values into input fields, including the amount and currency codes for both the source and target currencies. Upon pressing the convert button, the application triggers currency conversion and displays the result on the screen.
2. Export to CSV File: The application allows users to export currency conversions, performed in the current session, to a CSV file. By pressing the export button, a CSV file is generated, capturing the conversion details for external usage.
3. Reset conversion fields: A reset button is integrated to clear all conversion fields when clicked, providing users with a convenient way to start a new conversion or correct entries.

The layout components envisioned in the application are illustrated in Figure 6, offering an overview of the UI design and showcasing the positioning of the key features.

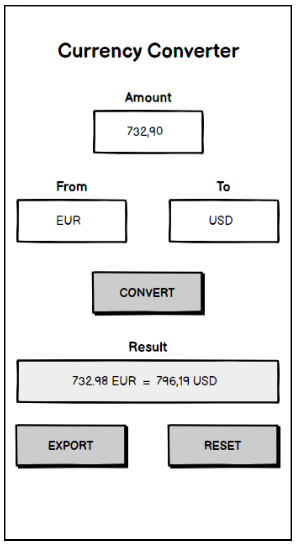


Figure 6: *Mockup of application UI design*
(Source: Author's illustration)

For the coding process, a suitable AI platform capable of aiding in code production had to be selected. After a brief experimentation period, ChatGPT was selected as the preferred AI-assisted coding tool. While StarCoder was also explored in an experimental manner in the early stage, it did not exhibit accuracy comparable to ChatGPT (refer to Section 5.2 for detailed reasoning). The decision to try StarCoder stemmed from the absence of performance evaluations by third-party studies at the time (see Section 2.3). For each sample application, a chat session was initiated, user prompts were entered, and responses were utilized to generate code, whether by copying and pasting generated code snippets or adjusting existing code based on instructions in the responses. To maintain a record for documentation and traceability purposes, chat sessions were saved in HTML format in order to preserve the original formatting of the chat history. This involved generating a shareable link within ChatGPT, accessing the link, and downloading the HTML file along with its associated resources.

With this outlined methodological approach, the sample applications were, accompanied by AI-assisted coding, precisely developed. In-depth details regarding the realized implementation are thoroughly discussed in Chapter 5.

3.4. Metric-Based Comparison

Within the scope of the metric-based comparison, the selection and measurement of the metrics as well as the employed statistical analysis components are described, outlining the approach to evaluating the sample applications.

3.4.1. Metrics Measurement

The following metrics for the comparative analysis were chosen, primarily based on the results of the literature review presented in Section 4.2:

- CPU usage
- memory usage
- battery usage
- startup time
- network traffic
- package file size
- installed application size
- lines of code (LOC).

Network traffic was specifically included due to its relevance to the main feature of the sample applications, involving web API calls to retrieve the exchange rate for the currency conversion. Despite its less frequent use in the examined comparison studies, battery usage was also added, to intensify the focus on resource conservation. The outcomes of the metric measurements will be detailed in Chapter 6, except for the LOC metric, which is not technically classified as an application performance metric but rather as a code metric and, therefore, is showcased in Chapter 5, Section 5.1. To measure the LOC metric, the open-source console application *cloc* was utilized (Danial, 2023). To enable categorization based on code type, the project was divided into configuration files and source code specifically for the metric measurement. For the CPU usage, the peak CPU usage as percentage is measured. The memory usage is quantified by the peak resident memory allocations in megabytes, which corresponds to the amount of physical memory the process is using for normal memory allocations. The battery usage is determined by the peak instantaneous current in microamps. Peak values are utilized and considered sufficient for reporting performance metrics because they indicate the highest resource consumption during the test scenarios, as emphasized and employed by Biørn-Hansen et al. (2020, p. 3011). The startup time is quantified by the time of application launch in milliseconds. The network traffic is measured as the sum of received and sent traffic in kilobytes. The package file size corresponds to the file size of the application package file (in the case of Android, APK file), retrieved from the hardware's file system in megabytes. The installed application size refers to the size of the installed application on the mobile device in megabytes. LOC represents the number of code lines of the source code projects, excluding comments and empty lines.

The performance testing was exclusively conducted on Android devices due to two main reasons. The primary constraint was time, which prevented the development of a Native iOS application, thus lacking a baseline for comparison. Additionally, the React Native application, though functional on the iOS simulator, faced deployment challenges on actual iOS devices. A detailed account of the challenges encountered is provided in Section 7.3. Faced with the unavailability of two out of the three intended comparison technologies on the iOS platform, the decision was made to exclude iOS from the testing scope. To compensate for this exclusion, multiple testing devices were utilized on the Android side to ensure a comprehensive evaluation. Details about the mobile test devices used can be found in Table 5, sorted by hardware capacity.

Table 5: *Specifications of mobile test devices*

Brand	Model	Release	RAM	Storage	Processors	Android version
Motorola	Moto G10	Feb 2021	4 GB	64 GB	Qualcomm Snapdragon 460 1.8 + 1.6 GHz, 64-bit Octa-Core, 11 nm	11
Google	Pixel 4a	Aug 2020	6 GB	128 GB	Qualcomm Snapdragon 730G 2.2 + 1.8 GHz, 64-bit Octa-Core, 8 nm	13
Motorola	Moto G54	Sep 2023	8 GB	256 GB	MediaTek Dimensity 7020 2.2 + 2.0 GHz, 64-bit Octa-Core, 6 nm	13

Manual testing was employed for the evaluation process due to time constraints, preventing the creation of automated test cases. This decision aligns with insights from Mota and Martinho (2021, p. 154), who highlight limitations of automation tools. These limitations include their inability to support testing in release mode and the requirement to install additional applications on mobile devices, with unknown impacts on results. Another critical factor is the need for automation tools to start applications under testing mode, introducing overhead, and therefore not reflecting common performance circumstances of the mobile application from a regular user's perspective. Consequently, the testing process demanded considerable effort and precision, additionally given the subsequent analysis and translation of all values into conclusions. An essential aspect of this process was the execution phase, which entailed a substantial time investment. In the selection of evaluation tools, preference was given to the official testing tools provided by the platform. Consequently, for most metrics measurements, *systrace* files were directly recorded on the device and subsequently analyzed in Android Studio and Perfetto Trace Viewer to extract pertinent values (Google, 2024c). Additionally, for the retrieval of network traffic data, the open-source network monitoring application *PCAPdroid* was chosen (Faranda, 2024). In the case of this third-party application, the measured metric value was not impacted by the potential introduced overhead, as the measurement with this tool was not performed simultaneously with the capture of the *systrace* file.

The test sequences employed a systematic approach to assess various performance metrics of the sample applications. For CPU usage, memory usage, battery usage, and startup time, three subsequent scenarios were executed and measured distinctly. Scenario 1 focused on the startup phase. Scenario 2 involved entering two conversions, while introducing a typo in the second conversion, correcting the typo, and converting again. Scenario 3 encompassed exporting the two performed conversions of the current session. Measuring the network traffic included two subsequent scenarios: the initial conversion (Scenario 1) and subsequent conversions (Scenario 2). The input values mirrored those in the first test sequence, excluding the typo. Each test sequence was executed 15 times for each of the three sample applications on every designated test device. At the beginning of each test sequence on every specified device, the particular sample application was installed in release mode using the corresponding APK file, all of which are available in Appendix B2. After each test iteration in both test sequences, the application was forced to quit, and all storage data was cleared to maintain a consistent cold start condition between test runs (Google, 2024a).

Table 6: *Test sequence #1*

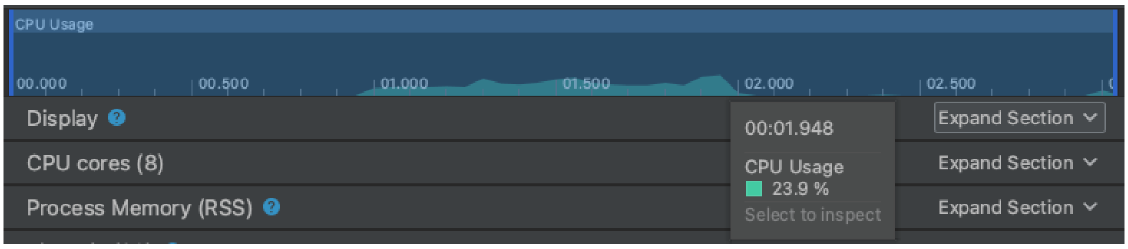
Scenario	Steps
1) Startup	- Start application
2) Conversions	- Enter values - *Amount*: 123.45 - *From*: CAD - *To*: USD - Press *Convert* button - Press *Reset* button - Enter values - *Amount*: 789 - *From*: MAD - *To*: EUU - Press *Convert* button - Correct value - *To*: EUR - Press *Convert* button
3) Export	- Press *Export* button

Table 7: *Test sequence #2*

Scenario	Steps
1) Initial conversion	- Enter values - *Amount*: 123.45 - *From*: CAD - *To*: USD - Press *Convert* button
2) Subsequent Conversion	- Press *Reset* button - Enter values - *Amount*: 789 - *From*: MAD - *To*: EUR - Press *Convert* button

Tables 6 and 7 provide a detailed, step-by-step listing of both described test sequences. Screen recordings for both test sequences are provided in Appendix B3. It is noteworthy that these screen recordings were created for demonstration purposes only and were not part of the data collection process. In the first test sequence, the trace recording was started at the beginning for each of the three scenarios and stopped upon reaching the end of the scenario. In the second test sequence, the recording of network traffic was started initially, and screenshots of the values in the network monitoring application were taken after each scenario.

Some of the captured metrics underwent an extensive extraction process, using various methods. For CPU, memory, and battery usage, the recorded systrace files were opened in Android Studio to analyze and capture peak values in screenshots, as performed by Işitan and Koklu (2020, pp. 278-279) in their study. An example of a screenshot capturing the peak CPU usage is provided in Figure 7. For the startup time, systrace files of only the first scenario, containing the application launch, were opened in Perfetto Trace Viewer and the startup duration values were directly copied, see the presented example in Figure 8 (Google, 2024c). All extracted values from Android Studio, the network monitoring application, and Perfetto Trace Viewer were meticulously compiled and transferred to dedicated Excel files for each metric. This thorough process ensured accurate and comprehensive documentation of the performance data, laying the foundation for subsequent analysis and interpretation. All captured measurement data that served as basis for subsequent data extraction can be found in Appendix B4, including the recorded trace files and described screenshots from Android Studio and *PCAPdroid*.

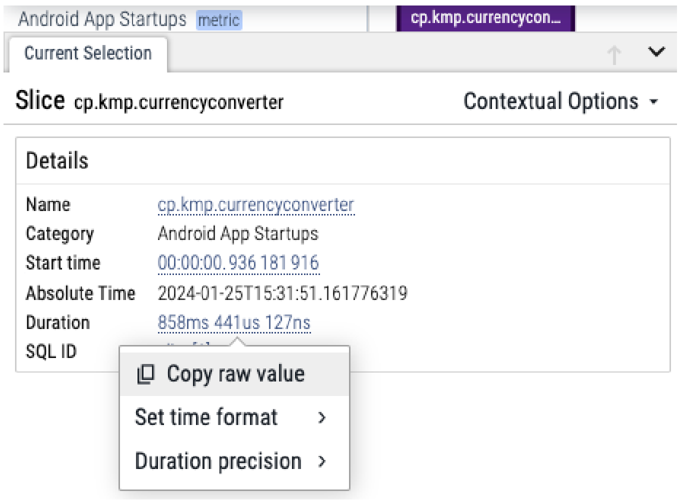


00:01.948

CPU Usage

🟩 23.9 %

Figure 7: *Screenshot of peak CPU usage in Android Studio*



Slice cp.kmp.currencyconverter

Details

Name	cp.kmp.currencyconverter
Category	Android App Startups
Start time	00:00:00.936 181 916
Absolute Time	2024-01-25T15:31:51.161776319
Duration	858ms 441us 127ns
SQL ID	

Figure 8: *Startup time in Perfetto Trace Viewer*

3.4.2. Statistical Comparison

Of the described metrics used, APK file size, installed application size, and LOC cannot be compared with statistical methods due to their static nature, meaning their values remain constant across multiple test runs. Thus, the differences of these static metrics were presented in a descriptive comparison. In the comparative analysis of the dynamic metrics, following statistical measures were applied:

- Compute the arithmetic mean and standard deviation from n iterations, where each iteration corresponds to a specific metric for a sample application on a particular device.
- Conduct one-way analysis of variance (ANOVA) to identify statistically significant differences between each pair of the three applications.
- Calculate the effect size using omega squared to quantify the magnitude of differences between the sample applications.

The arithmetic mean and standard deviation provide a comprehensive overview of the value amounts within a dataset. The arithmetic mean represents the average of a dataset. It is "a measure of center" (Kormanik, 2016, p. 93) that considers every value in a data set, sums them up, and divides the total by the sample size n. The arithmetic mean is defined as

$$\bar{x} = \frac{\sum_{i=1}^{n} x_i}{n}$$

where $\bar{x}$ refers to the mean value and x_i to the measured individual values of the data set (Kormanik, 2016, pp. 14-15). The standard deviation σ quantifies the average deviation of values from the mean and is defined as

$$\sigma = \sqrt{\frac{\sum_{i=1}^{n} (x_i - \bar{x})^2}{n}}$$

and equals the square root of the variance σ^2 (Kormanik, 2016, p. 94).

The size of the datasets is $n = 15$ (as described in Section 3.4.1), since this sample size is considered robust for one-way ANOVA when comparing less than 10 groups (Minitab, 2023). One-way ANOVA is utilized as a statistical test to evaluate differences of means in the dependent variable, representing the measured metrics, with respect to the independent variable, which is the technology utilized in the sample application. ANOVA is applied as one-way because it involves only one independent variable. The requirements for ANOVA are fulfilled, as the dependent variable is continuous, approximately normally distributed, and demonstrating homogeneous variances within each group. Additionally, the independent variable is categorical and consists of mutually exclusive values. The null hypothesis in ANOVA states that there are no significant differences among the means of all groups being compared, while the alternative hypothesis suggests that at least two group means are significantly different. Therefore, the null hypothesis is rejected in favor of the alternative hypothesis in case the latter applies (Kormanik, 2016, p. 59). One-way ANOVA serves as a suitable method for comparing means across three or more independent groups, while in case of involving only two groups, the outcomes of a t-test (significance test between the means of two groups) and one-way ANOVA are identical. In instances where significant results arise from a one-way ANOVA across multiple groups, it signifies that at least two groups exhibit statistically significant differences, without indicating which groups specifically. Therefore, post-hoc tests, e.g. pairwise t-tests, are crucial to determine the specific group differences. However, due to the low number of compared groups, this thesis deviates from this conventional approach by directly conducting pairwise ANOVA comparisons, being equivalent to t-test outcomes (Hemmerich, 2024). The preference for ANOVA over t-test in this context arises from the necessity of ANOVA result parameters for the subsequent omega squared calculation. In the scope of ANOVA, the following parameters are calculated: Sum of squares (SS), degrees of freedom (df), mean squares (MS), F-statistic (F), p-value (p), and critical F-value. The specific definitions and formulas for all these parameters exceed the scope of this thesis; more in-depth details can be found in the statistics fundamentals book by Kormanik (2016, pp. 59-63). Notably, emphasis is placed on the p-value. The p-value indicates the probability of differences in means occurring solely by random chance (Kroes & Finley, 2023, p. 1). A significance level of $p < .05$ indicates whether the difference in the effect of the used technology on the measured metric is statistically significant. However, the statistical significance alone does not provide information about the magnitude of the effect (Kroes & Finley, 2023, p. 3). To calculate the so-called effect size, omega squared (ω^2) can be utilized, which is the superior measure of effect size compared to eta squared due to the correction of bias (Kroes & Finley, 2023, pp. 2, 5-7). Since various ways to calculate omega squared exist, Kroes and Finley (2023, p. 17) strongly recommend to report the specific formula used for calculating effect sizes in a study. In the case of this thesis, the formula for standard omega squared for any size between-subjects design with fixed factors was applied (Kroes & Finley, 2023, pp. 7-8):

$$\omega^2 = \frac{SS_{between} - (df_{between} \times MS_{within})}{SS_{total} + MS_{within}}$$

The effect size computed with omega squared consistently falls within the range of 0 to 1, indicating the magnitude of observed differences; the closer the effect size approaches 1, the larger the observed differences (Kroes & Finley, 2023, p. 16). Contrary to the recommendation made by Kroes and Finley (2023, p. 16) regarding reporting effect sizes with three decimal places, this thesis opted to round effect sizes to one decimal place after the comma, given that the observed values fell within the range of 0.1 to 1 and this level of precision effectively depicted the magnitude of differences observed, having differences greater than 0.1. In cases where the effect size was listed as 1.0, it is due to rounding, as the effect size would never be exactly 1. Kroes and Finley (2023, p. 17) emphasize the inadequacy of standardized classifications for effect sizes as small, medium, or large, since they vary depending on research areas. Therefore, in this thesis, effect size classification was developed based on the individual range of the measured values.

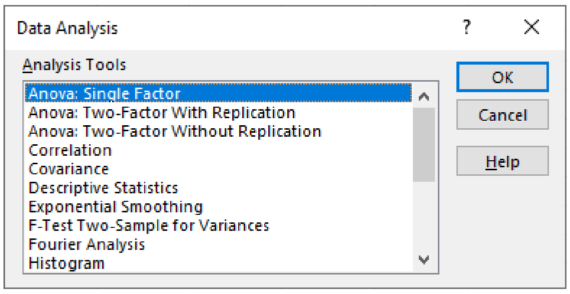
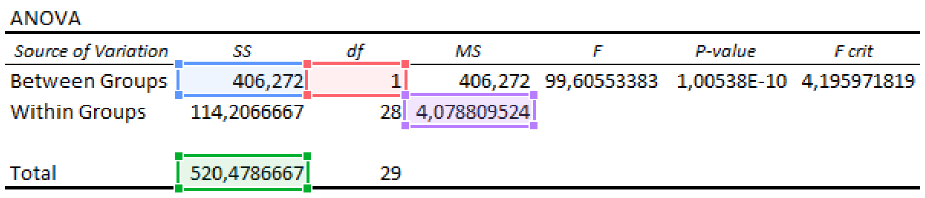


Figure 9: *Data Analysis feature in Excel*

The computations of ANOVA and omega squared were performed using *Microsoft Excel 365*. With the *Data Analysis* feature (see Figure 9), comprehensive ANOVA calculations can be executed. The subsequent omega squared calculation utilized parameters obtained from the ANOVA result table. Figure 10 illustrates an example of such ANOVA table, where the specific parameters used for the omega squared calculation are framed in color.


ANOVA

Source of Variation	SS	df	MS	F	P-value	F crit
Between Groups	406,272	1	406,272	99,60553383	1,00538E-10	4,195971819
Within Groups	114,2066667	28	4,078809524			
Total	520,4786667	29				

Figure 10: *Example of ANOVA table*

4. Comparative Studies of Cross-Platform Mobile Development Solutions

A review of various studies comparing diverse CPMD solutions offers valuable insights for this thesis. In Section 4.1, an overview of comparison studies of CPMD technologies is provided, detailing the CPMD technologies examined, the specifics of implemented sample applications, and how the comparison conducted. In Section 4.2, a compilation is drawn regarding which metrics were utilized in the analyzed comparison studies.

4.1. Comparison Studies

As result of the systematic literature review detailed in Section 3.2, Table 8 presents the 21 CPMD comparison studies assessed as relevant. Moreover, the table specifies the CPMD technologies examined in each study, Native applications included as baseline, the platforms used for testing, the sample application type, and the comparison approach.

Table 8: *Overview of various CPMD comparison studies and corresponding comparison details*

Study	CPMD technology evaluated						Native		Tested platform		Application type	Comp.	
	Flutter	Ionic	KMP	React Native	Xamarin	Other	Android	iOS	Android	iOS		Metrics	Qualitative
Biørn-Hansen et al. (2020)	x	x		x		x	x		x		Benchmarking	x	
Biørn-Hansen, Grønli and Ghinea (2019)		x		x	x		x	x	x	x	Benchmarking	x	
Dalmasso et al. (2013)						x			x		Benchmarking	x	
Ebone et al. (2018)					x	x	x	x	x	x	Benchmarking	x	x
El Tom et al. (2023)				x	x				x	x	Task list	x	
Evert (2019)			x				x	x	x	x	Information retrieval	x	
Fentaw (2020)	x			x					x	x	Information retrieval	x	x
Goetz and Li (2018)					x	x			x	x	Online retail	x	x
Heitkötter et al. (2013)						x					-		x
Huber et al. (2020)		x		x			x		x		Contacts	x	
Işitan and Koklu (2020)	x			x	x	x			x		Benchmarking	x	
Lachgar et al. (2022)											-		x
Mahendra and Anggorojati (2021)	x			x			x		x		Social media	x	
Majchrzak et al. (2017)		x		x		x			x	x	Social media		x
Mota and Martinho (2021)	x			x					x	x	Benchmarking	x	
Nawrocki et al. (2021)	x			x	x		x	x	x	x	Benchmarking	x	x
Rieger and Majchrzak (2019)				x		x	x	x			-		x
Shah et al. (2019)											-		x
Willocx et al. (2015)					x	x	x	x	x	x	Online retail	x	
Wu (2018)	x			x					x		Information retrieval	x	
You and Hu (2021)	x			x	x		x		x		Blockchain wallet	x	x
Totals	8	4	1	13	8	8	10	6	17	10	-	16	10

Numerous research studies have been conducted comparing various CPMD technologies. While identifying the CPMD technologies examined in the studies outlined in Table 8, special attention was given to the CPMD frameworks React Native, Flutter, Ionic, and Xamarin due to their present significance, and KMP because of its novelty, as outlined in Section 2.2.3. CPMD technologies investigated in the comparison studies, which did not fall within this group, were categorized as *Other* and were considered negligible in the context of this thesis, as they are either outdated or not established. These other technologies included Cordova / Phonegap (Dalmasso et al., 2013; Ebone et al., 2018; Goetz & Li, 2018; Heitkötter et al., 2013; Willocx et al., 2015), Titanium (Dalmasso et al., 2013; Ebone et al., 2018; Heitkötter et al., 2013), NativeScript (Biørn-Hansen et al., 2020), MAML/MD (Biørn-Hansen et al., 2020), and Fuse (Majchrzak et al., 2017). Furthermore, a distinction was made between metrics and qualitative comparison. Qualitative comparisons encompassed both descriptive comparisons of qualitative aspects and more structured evaluations utilizing systematic frameworks. It is noticeable that the studies by Lachgar et al. (2022) and Shah et al. (2019) lack specification of CPMD technologies in their evaluations. Lachgar et al. (2022) develop a ranking-based decision framework for selecting suitable CPMD technologies, wherein the evaluated technologies remain unnamed, being only referred to with variable names. Similarly, Shah et al. (2019) compare CPMD approaches instead of specific technologies, employing a descriptive analysis based on predefined comparison parameters. Regarding the evaluated CPMD technologies across all studies, React Native emerges as the clear winner, followed by Xamarin and Flutter, while Ionic was rarely compared and KMP is even assessed only once, exclusively against Native. Approximately half of the studies conducted comparisons against Native in order to establish a comparative baseline, where Native developments were observed either on both platforms or exclusively on Android. A parallel trend is noted concerning the tested platforms, where evaluations were performed either across both platforms or solely on Android. Remarkably, the opposite scenario, exclusive testing on iOS, is entirely absent. Concerning the sample applications developed, pure benchmarking applications were the most frequent choice. Those benchmarking applications were designed for specific tasks or operations, that systematically target concrete performance aspects for measurement and comparison. Examples of benchmarking functionalities employed in the studies include animations (Biørn-Hansen, Grønli, & Ghinea, 2019, pp. 8-10; Mota & Martinho, 2021, p. 153), various web requests (Dalmasso et al., 2013, p. 26; Nawrocki et al., 2021, p. 21), list elements (Işitan & Koklu, 2020, p. 276; Mota & Martinho, 2021, p. 152; Nawrocki et al., 2021, p. 21), local file access (Biørn-Hansen et al., 2020, p. 3011; Mota & Martinho, 2021, p. 153), and CPU-intensive calculations (Nawrocki et al., 2021, p. 21). The remaining sample applications encompassed more real-world scenarios representing practical use cases, such as information retrieval from the web (Evert, 2019, pp. 20-21; Fentaw, 2020, p. 40; Wu, 2018, p. 13), task management (El Tom et al., 2023, p. 6947), contacts management (Huber et al., 2020, p. 45), social media platform (Mahendra & Anggorojati, 2021, p. 33; Majchrzak et al., 2017, p. 6162), online retail platforms (Goetz & Li, 2018, p. 11; Willocx et al., 2015, p. 457), or blockchain wallet (You & Hu, 2021, pp. 4-5).

Sixteen of the comparative studies utilized metrics measurements, which are further elaborated upon in Section 4.2. Alongside metric measurement, some studies additionally incorporate the qualitative comparison aspect of subjective development experience (Goetz & Li, 2018, p. 16; Majchrzak et al., 2017, pp. 6167-6168; Nawrocki et al., 2021, p. 25; You & Hu, 2021, p. 7). Besides development experience, Majchrzak et al. (2017, pp. 6168-6169) also describe aspects related to code reusability, community and popularity of the respective CPMD technology. Other studies focus on employing a systematic approach to comparison criteria. In their analysis of approaches, Shah et al. (2019, pp. 5-6) highlight 11 comparison parameters for descriptive comparison. Heitkötter et al. (2013, pp. 300-301) create a comparison framework for CPMD technologies, which includes infrastructure and development perspectives, each encompassing seven sub-criteria. Rieger and Majchrzak (2019, pp. 179-186) design an extended criteria catalog across the four perspectives: infrastructure, development, application, and usage, comprising a total of 33 criteria. El Tom et al. (2023, pp. 6948-6949) apply the criteria catalogue developed by Rieger and Majchrzak (2019) additionally to their metrics comparison. Lachgar et al. (2022, p. 12) develop a decision framework for selecting the suitable CPMD technology, incorporating four main criteria: business environment perspective, infrastructure perspective, development perspective, and application perspective, with a total of 16 sub-criteria combined. Fentaw (2020, pp. 68-72) similarly proposes a guideline for framework selection.

Based on the analyzed comparison studies, the general observation can be drawn that CPMD technologies inherently exhibit performance overhead compared to Native applications (Biørn-Hansen et al., 2020, p. 3031; Dalmasso et al., 2013, p. 328; Huber et al., 2020, p. 55; Mahendra & Anggorojati, 2021, p. 37; Nawrocki et al., 2021, p. 26; Willocx et al., 2015, p. 460; You & Hu, 2021, p. 8). Given the varying degree of performance overhead across different domains, no universal victor among the CPMD solutions can be designated, and the optimal choice in a particular use case depends on different factors, such as the specific application features as well as the programming skills and experience of the developers (Biørn-Hansen, Grønli, & Ghinea, 2019, pp. 17-18; Biørn-Hansen et al., 2020, p. 3031; Ebone et al., 2018, p. 92; Fentaw, 2020, p. 74; Goetz & Li, 2018, p. 17; You & Hu, 2021, pp. 8-9). Some studies collectively conclude that Flutter and React Native emerge as preferred choices. Wu (2018, p. 25) underscore the great value provided by both React Native and Flutter, affirming their effectiveness in CPMD. Işitan and Koklu (2020, p. 273) emphasize Flutter and React Native as superior options compared to Xamarin and NativeScript. Fentaw (2020, p. 74) recommend React Native for applications requiring real-time communication and social interaction functionalities, and suggest Flutter for those with a focus on extensive search and filtering capabilities. Mahendra and Anggorojati (2021, p. 37) highlight Flutter's superior performance over React Native in resource utilization, while Mota and Martinho (2021, p. 156) note the superiority in visual effects. Nawrocki et al. (2021, p. 26) advocate for Flutter as the preferred choice. However, in their conclusion, the relevant deciding factor was the development experience, with React Native being the next highest rated.

In sharp contrast to the relatively extensive amount of comparison studies involving established CPMD frameworks, there is a notable lack of studies incorporating KMP. The sole identified academic paper on this subject was the Master’s thesis by Evert (2019), which evaluated the performance of KMP in comparison to Native. For this thesis, the evident research gap regarding KMP encourages its inclusion as a CPMD technology to evaluate. React Native is considered as most established framework, evidenced by its frequent evaluation within the literature. Incorporating Native as a baseline for comparison aims to facilitate a comprehensive evaluation, with special attention to the expected overhead in CPMD technologies. Limitations necessitated testing solely on Android, an approach frequently observed in the analyzed studies. The chosen sample application type, a currency conversion application, serves as the foundation for evaluation, encompassing API requests and file access. This choice complies with the absence of such applications in the analyzed studies. The study involves both quantitative metrics analysis and a qualitative examination of development experience, with a specific focus on the incorporation of AI assistance in coding.

4.2. Comparison Metrics

To determine the metrics suitable for evaluating CPMD technologies, the metrics employed in the comparison studies elucidated in Section 4.1 are analyzed. Table 9 offers an overview of the frequency of the metrics used. CPU usage, memory usage, and application package file size are the most predominantly utilized metrics. Execution time, frame rate, LOC, installed application size, and startup time were consistently but less frequently used. Battery usage, GPU usage, and response time were only sporadically in use, while build time and network traffic were applied solely once.

Table 9: *Overview of metrics used in various CPMD comparison studies*

Study	Battery usage	Build time	CPU usage	Execution time	Frame rate	GPU usage	LOC	Memory usage	Network traffic	Response time	Size of package	Size installed	Startup time
Biørn-Hansen et al. (2020)			x	x				x					
Biørn-Hansen, Grønli and Ghinea (2019)			x		x	x		x					
Dalmasso et al. (2013)	x		x					x					
Ebone et al. (2018)								x		x	x		
El Tom et al. (2023)							x				x		
Evert (2019)		x					x					x	x
Fentaw (2020)			x			x		x					
Goetz and Li (2018)			x	x			x	x			x		x
Huber et al. (2020)			x			x		x					
Işitan and Koklu (2020)	x		x					x	x		x	x	
Mahendra and Anggorojati (2021)			x		x			x		x	x	x	
Mota and Martinho (2021)			x	x	x			x					
Nawrocki et al. (2021)			x					x			x		x
Willocx et al. (2015)			x					x		x	x	x	x
Wu (2018)				x	x								
You and Hu (2021)			x				x	x			x		
Totals	2	1	12	4	4	3	4	13	1	3	8	4	4

In the following, the named metrics are briefly defined. Battery usage refers to the amount of electrical power consumed by the mobile application (Dalmasso et al., 2013, p. 327). CPU usage is the percentage of time that the central processing unit is active, relative to the total processing capacity available (Biørn-Hansen et al., 2020, p. 3011). The execution time is the duration the application needs to complete a specific task or operation (Goetz & Li, 2018, p. 11). The frame rate is the rate at which consecutive frames are displayed by the application, influencing visual fluidness in the user experience (Biørn-Hansen, Grønli, & Ghinea, 2019, p. 6). GPU usage is the utilization degree of the graphics processing unit by the application, for graphical operations (Biørn-Hansen, Grønli, & Ghinea, 2019, pp. 7, 19). LOC are the total number of code lines in the application's source code (You & Hu, 2021, p. 5). Memory usage is the amount of device memory consumed by the application during execution (Biørn-Hansen, Grønli, & Ghinea, 2019, p. 7). Response time is the time needed by the application to respond to a specific event (Mahendra & Anggorojati, 2021, p. 34). The application package size refers to the package file for installation, which is downloaded from the application marketplace, while the size of the installed application refers to the occupied device storage after installation. Biørn-Hansen et al. (2020, p. 3009) emphasized that the application package size is a highly essential metric because it might have direct impact on application adoption across marketplaces as it represents the download size for users, potentially influencing their decision to install the app. Startup time is the duration the application takes to launch and become operational after initiation (Goetz & Li, 2018, p. 11). Network traffic is the amount of data transmitted from or to the mobile application over an established network connection (Işitan & Koklu, 2020, pp. 279-280). Build time is the duration to compile the mobile application from its source code (Evert, 2019, p. 23). Both LOC and build time are metrics that measure efficiency within the development cycle instead of the application's performance.

For CPU and memory usage, Biørn-Hansen, Grønli and Ghinea (2019, p. 11), Biørn-Hansen et al. (2020, p. 3011), Fentaw (2020, p. 54), Nawrocki et al. (2021, p. 22), Işitan and Koklu (2020, pp. 277-279) measure peak values, whereas Dalmasso et al. (2013, p. 327), Huber et al. (2020, pp. 49-50), Mahendra and Anggorojati (2021, p. 34), and Willocx et al. (2015, p. 457) measure averages. While providing averages offers a better indication of total usage over time, peak values represent the maximum resource utilization, which are more crucial for performance analysis (Biørn-Hansen et al., 2020, p. 3011). As previously mentioned, performance outcomes vary significantly depending on the measured scenario and metric, leaving little room for generalizations. Nonetheless, some valuable insights regarding metrics have emerged from the studies. El Tom et al. (2023, pp. 6950-6951) observe that React Native generates smaller APK files compared to Xamarin. Evert (2019, p. 36) note a significant delay in the startup time of KMP in comparison to Android Native, as well as a larger installed application size. The latter could be attributed to the initial size overhead, which might diminish in significance with application scaling. Both Huber et al. (2020, p. 55) and Nawrocki et al. (2021, p. 26) agree on the relatively high CPU and memory demands of React Native, particularly when contrasted with Native. Nawrocki et al. (2021, p. 26) further highlight a notably poorer development experience with React Native compared to Native or Flutter. Regarding the identified research gap concerning KMP, Evert (2019, p. 37) stress the need for more detailed research on KMP performance, advocating for a wider variety of metrics in the evaluation.

Drawing from the analysis of employed metrics in the considered comparison studies, the selection of metrics for this thesis is explained in the following, after they were already briefly introduced in Section 3.4.1. From a sustainability and resource conservation standpoint, metrics such as battery, CPU, memory and GPU usage, as well as network traffic play a role. Therefore, CPU and memory usage were included for this reason and due to their high frequency of use. The application package size was also covered due to its frequent usage, along with the installed application size, which is an easily retrievable value for contrasting with the package size. Startup time was also selected as it is a crucial metric for user experience. Although execution and response time as well as frame rate are also relevant to user experience, they were excluded because it was assumed that the application consisted of scenarios too small to significantly impact execution or response time and lacked special graphics or animations to affect frame rate. For the latter reason, GPU was also not included. The code metric LOC was reflected upon in the implementation chapter due to its moderate frequency of use, but build time was not included, as the focus was placed on application performance rather than development performance. Battery usage and network traffic were employed due to their energy consumption aspect. Although network traffic was utilized only once in the comparison studies, it represented a relevant metric in the context of the developed sample application in this case study, as it encompasses network communication via API interactions.

5. AI-Assisted Implementation of Sample Applications

The sample applications were implemented, utilizing AI to aid during the development process. Section 5.1 discusses the technical details of the implementations, while Section 5.2 provides a detailed account of the incorporation of AI-assisted coding.

5.1. Implementation Details

In the following, the application features and architecture, which apply in all technologies, are addressed, followed by implementation variations depending on the technology, including the project structure and size, the layout, essential libraries, and the creation of release APK files.

5.1.1. Application Features and Architecture

The implementation of the three core features outlined in Section 3.3 was carried out, accompanied by input validation and user feedback mechanisms.

1. Currency conversion: Triggered by pressing the convert button, the input field values, including amount, base and target currency codes, are fetched, and upon validation, an API call is made to retrieve the current exchange rate for the specified currencies. The API calls are handled in the `ApiService` class, using the free version of *currencyconverterapi.com* (Vergel, 2014). After extracting the exchange rate from the API response, the currency conversion is calculated with the required parameters and displayed on the user screen. The currency conversion process is illustrated in Figure 11 as a UML sequence diagram. The base and target currency codes are entered as letters in a text field, and it is acknowledged that this is not the most optimal or user-friendly approach. In a real-world application, additional development resources would have been allocated to enhance the user experience, potentially incorporating features like a dropdown menu or autocomplete functionality. However, due to time limitations, these enhancements were not implemented in the sample applications, as the main focus was on functionality and the ability to compare the applications' performances. In order to utilize the specified API, a user account had to be created to acquire the API key. With the chosen free plan, the number of requests is restricted to 100 per hour, which made extensive testing involving API requests unfeasible. The received API key must be renewed monthly using the renewal link sent via email. This also implies that, once the API key expires, it requires code modifications and new application deployment in order to maintain a functional app, as the API key is stored in a configuration file.

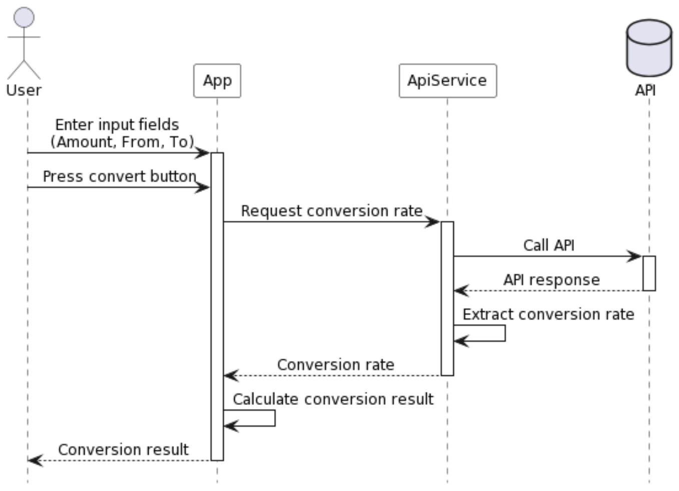


Figure 11: *UML sequence diagram of currency conversion*
(Source: Author's illustration)

2. Export to CSV file: Upon pressing the export button, a CSV file containing the currency conversions from the current session is created and saved on the mobile device. The export operation is managed within the `ExportService` class, and success of the export is displayed as user feedback on the screen. The export procedure is presented in the UML sequence diagram in Figure 12. In the exported CSV file, the following columns are filled: Timestamp of the conversion, base code, base amount, target code, and target amount. An illustrative screenshot of an example CSV export can be found in Figure 13, while the CSV file itself is included in Appendix B5.

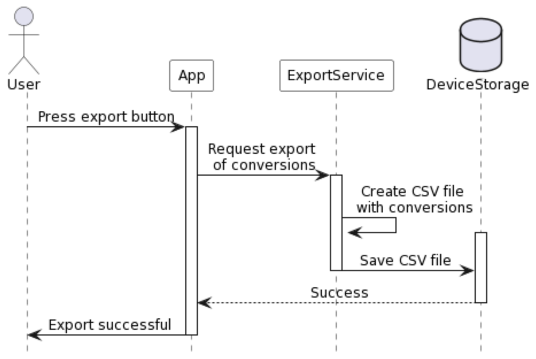
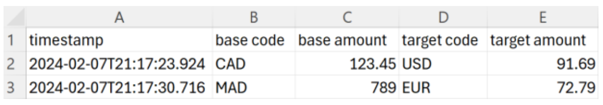


Figure 12: *UML sequence diagram of export*
(Source: Author's illustration)



	A	B	C	D	E
1	timestamp	base code	base amount	target code	target amount
2	2024-02-07T21:17:23.924	CAD	123.45	USD	91.69
3	2024-02-07T21:17:30.716	MAD	789	EUR	72.79

Figure 13: *Screenshot of exported CSV file*

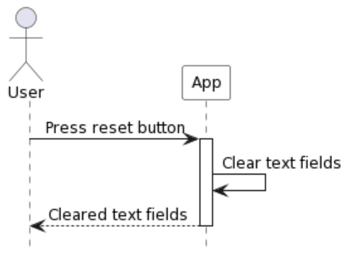


Figure 14:*UML sequence diagram of conversions field reset*
(Source: Author's illustration)

3. Reset conversion fields: When the reset button is pressed, the contents of the input fields and the result field are set empty. The conversion field reset is depicted in the UML sequence diagram in Figure 14.

Validation and user feedback: The application incorporates a validation mechanism for user input parameters, providing immediate feedback to the user to correct any errors. In the amount field, it is validated whether the entered input is in a numerical format. If not, a popup banner is displayed upon pressing the convert button, alerting the user to the invalid amount value. In both currency code fields, the entered codes are compared against the supported existing currency codes to ensure their validity. As a preemptive measure, the character length of these input fields has been restricted to 3 and letters are automatically entered uppercase, aligning with the ISO format of currency codes. Should at least one of the two currency code fields be invalid, the user is notified through a popup banner. In addition to validation alerts, popup banners also serve as user feedback for both successful and failed currency conversions or exports.

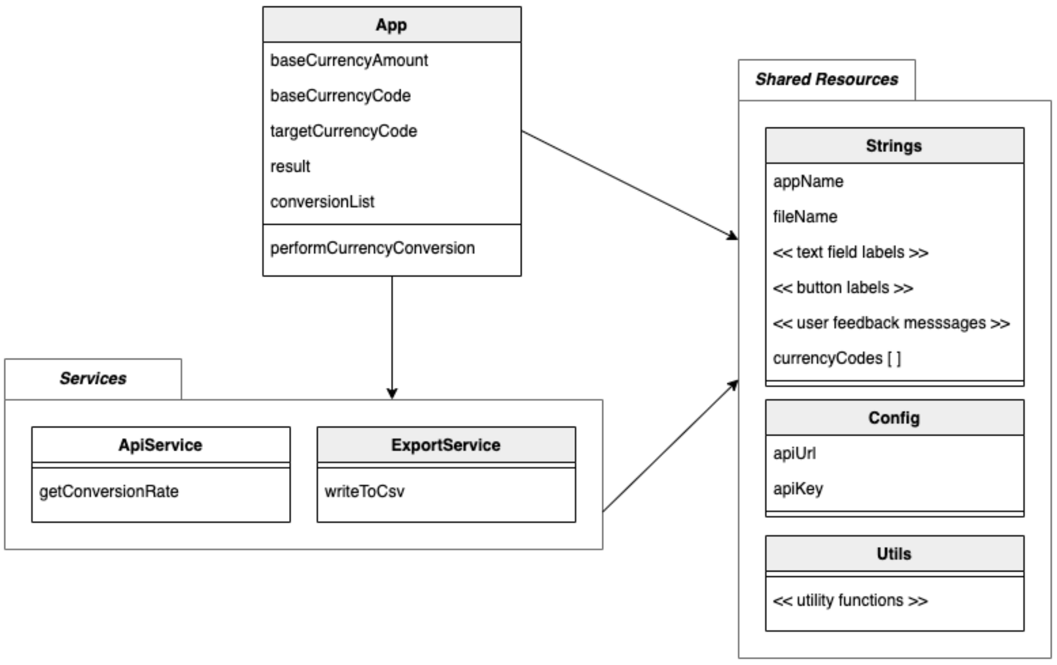


Figure 15: *Simplified UML class diagram of sample application*
(Source: Author's illustration)

Figure 15 presents a summarized UML class diagram, which is simplified and applicable to all sample applications. The main class `App` constitutes the entry point for the application. Since the application consists of one single screen only, the core method for currency conversion, performCurrencyConversion, is located here. The variables baseCurrencyAmount, baseCurrencyCode, and targetCurrencyCode are obtained from the input fields. The result variable contains the currency conversion outcome, while the conversionList maintains the parameters of successful conversions, serving as the basis for potential exports. The App class interacts with two services, ApiService and ExportService, upon request through button press. ApiService is responsible for fetching the exchange rate via the getConversionRate method, while ExportService manages the saving of currency conversions in a CSV file through the writeToCSV method. Additionally, the App class handles the reset of conversion fields, as it is a fairly straightforward operation. Both the main class and the service classes access shared resources. The Utils class offers helper methods for different operations, such as specific date or number formatting. The Config class holds the apiUrl and apiKey variables, which are crucial parameters for the API request. The Strings class holds appName, fileName for the exported file, all labels for text fields and buttons and all user feedback messages. Lastly, all supported 3-letter ISO currency codes for validating the input currency codes are stored there.

5.1.2. Variations across Technologies

As described in Section 2.2.3, the programming language for both Native and KMP was Kotlin, with Native requiring XML coding in addition, while React Native was implemented using JavaScript and embedded JavaScript XML for the layout definition. The project structure between Native and KMP are fairly similar, whereas React Native follows a significantly different and much simpler approach. The distinct folder structures are depicted in Figure 16 in a simplified manner, containing only the relevant folders for the following explanations and excluding configuration files. The source code for all projects can be found in Appendix B6.

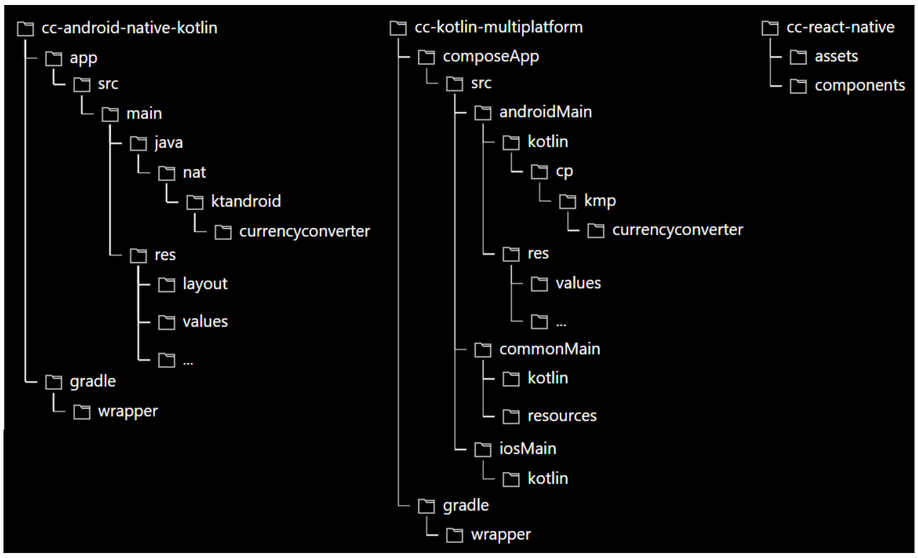


Figure 16: *Code projects directories application*
(Source: Author's illustration)

In Android Native projects, the main source code is typically located in app/src/main directory. Within the java directory, the Java or Kotlin source code files can be found, organized according to their package names. In this case, the package name nat.ktandroid.currencyconverter represents a hierarchical naming convention for organizing code, where nat denotes the top-level domain, ktandroid indicates a specific project or organization, and currencyconverter represents the application name. In this Native project, all Kotlin files are located within the currencyconverter directory. Directly within the main directory, the crucial configuration file AndroidManifest.xml contains application details for the Android system, including package name, permissions, and other configurations. Within the res directory, various resources are housed, such as the layout and values folders, as well as different drawable-* folders containing graphics and mipmap-* folders containing launcher icons in different resolutions. The UI structures in this project were all defined in activity_main.xml within the layout directory. In the values folder, strings and colors were defined in XML files. Further configuration files, primarily associated with Gradle, can be found at the root level as well as in the app and gradle/wrapper directories. In KMP, the main source code resides in the composeApp/src directory. Within this directory, there are the three folders commonMain for shared code as well as androidMain and iosMain for the platform-specific code, respectively. Most of the code logic could be implemented in the shared section, except for file access for the CSV export, which had to be implemented platform-specifically for Android and iOS. The androidMain directory mirrors the structure of Android Native projects as described above, while commonMain contains kotlin and resources folders, and iosMain solely contains the former. The Gradle configuration files also resemble those of Android Native projects. Additionally, in the commonMain/resources directory, the compose-multiplatform.xml file serves as a configuration file for Jetpack Compose, enabling centralized definitions of Compose-related resources shared across different platforms. Within this KMP project, the string values utilized and the API configuration values were directly defined in the commonMain/kotlin directory, alongside the other Kotlin source code files. In this React Native project, there is only one folder hierarchy. Within the root folder, the assets directory contains static files such as images, while the components directory holds reusable UI components. The main source code JavaScript files reside at the root folder level. API configuration and string values were stored in JavaScript files along with other source code files.

Table 10: *LOC and size of projects per technology*

Technology	LOC			Project size (KB)
	Configuration files	**Source code**	**Total**	
Native	313	875	1,188	132
KMP	420	784	1,204	194
React Native	13,999	454	14,453	642

Table 11: *KMP – LOC of shared and platform-specific code*

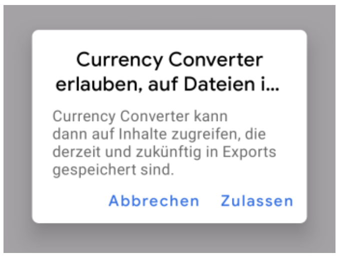
Source code	LOC	%
Shared	490	62,5
Platform-specific	294	37,5
Total	784	100

After highlighting the differences in project structure, the sizes of the projects are contrasted with each other. Table 10 lists the code metric LOC, split into configuration files and source code files, as well as the project size in kilobytes. It is noticeable that React Native has the fewest lines of source code, but a significantly larger number of code lines in the configuration files. This can be primarily attributed to the package-lock.json configuration file, which alone accounts for 13,935 LOC. The `package-lock.json` file provides a detailed record of the exact versions of dependencies installed in the React Native project, ensuring reproducible builds across different environments. As a result of these circumstances, the React Native project is the largest with 642 KB. The project size of KMP is on a similar level to Native, yet larger. KMP has slightly more total LOC with fewer LOC in the source code, but more in the configuration files. In total, Native is the slightly superior winner, both in terms of LOC and KB size. Focusing solely on the shared and platform-specific code of KMP, as displayed in Table 11, it is noticeable that the platform-specific segment of the code constitutes 37.5%, even though only the file access part of the export service required platform-specific implementation. This can be accounted for by the platform-specific code being twice as extensive, given that it had to be developed for both Android and iOS platforms.

Table 12: *Libraries used for HTTP client and file access per technology*

	HTTP client	File access
Native	`okhttp3`	`android.os.Environment`
KMP	`ktor-client`	`android.os.Environment` (Android) `platform.Foundation` (iOS)
React Native	`expo-file-system`	-

After discussing the structure and size of the projects, it is also worth highlighting the most essential libraries utilized for each technology. Crucial to the two implemented services were libraries for HTTP request handling in the API service and for file access in the export service. A concise overview of the utilized libraries for these purposes can be found in Table 12. While these were the primary libraries for the services, several additional libraries were required, specific to the respective technologies. In the API service, Native employed the single library `okhttp3` as the HTTP client. However, in the export service, alongside `android.os.Environment` for file access, Native also employed `java.io.File` and `java.io.FileWriter` for file representation and file writing, and `com.opencsv.CSVWriter` for CSV file creation. In the API service, KMP required, besides `ktor-client` for HTTP request handling, also `kotlinx.serialization.json` so that JSON data format could be handled from the API response. In the export service, platform-specific implementations were needed for Android and iOS. Similar to Native, Android utilized `android.os.Environment` for file system access and `java.io.File` for file representation. In the iOS implementation of KMP, `platform.Foundation` was used for file system access, and `platform.UIKit` was required for UI components and system-level interactions. The latter was required to request the save location of the export file, as iOS applications do not have direct permission to choose a storage location, which is a security measure on the iOS platform. For the export service, React Native utilized the `expo-file-system` library for file access. Analogous to iOS applications, React Native applications cannot directly access file storage due to security restrictions imposed by Android. Therefore, the user must grant file access permission, similar to the process in KMP on iOS. Figure 17 displays a screenshot of such file access request from the React Native sample application, where the user is prompted to grant storage permissions for a folder named *Exports*.

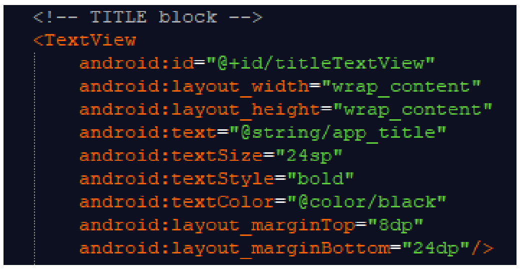


Currency Converter erlauben, auf Dateien i...

Currency Converter kann dann auf Inhalte zugreifen, die derzeit und zukünftig in Exports gespeichert sind.

Figure 17: *Screenshot of file access request in React Native sample application*

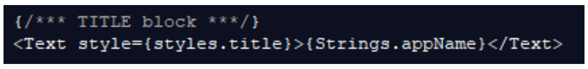
Moving on to the UI of the sample applications, the layout implementation differed across all three technologies. In Native, the layout was specified in the `activity_main.xml` file, while in KMP it was defined in the main class `App.kt` within the `Surface()` method, which is a Compose component for defining a surface and rendering UI components within it. In React Native, the layout was defined in `App.js` in the `return()` statement, defining the UI elements in JavaScript XML structure, and further specified in the `StyleSheet` object. To illustrate the variations in layout implementation, Figures 18–20 display the respective implementations of the title component in Native, KMP, and React Native.



```
<!-- TITLE block -->
<TextView
    android:id="@+id/titleTextView"
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    android:text="@string/app_title"
    android:textSize="24sp"
    android:textStyle="bold"
    android:textColor="@color/black"
    android:layout_marginTop="8dp"
    android:layout_marginBottom="24dp"/>
```

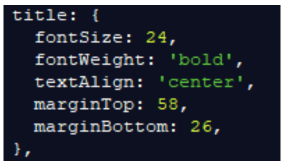
Figure 18: *Code segment of title UI component in Native*

return() statement:



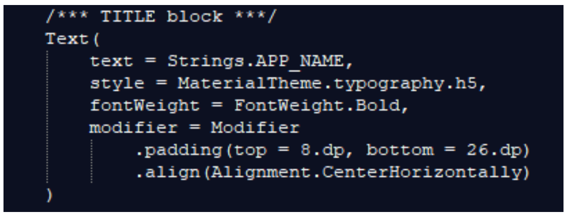
```
{/*** TITLE block ***/}
<Text style={styles.title}>{Strings.appName}</Text>
```

StyleSheet:



```
title: {
  fontSize: 24,
  fontWeight: 'bold',
  textAlign: 'center',
  marginTop: 58,
  marginBottom: 26,
},
```

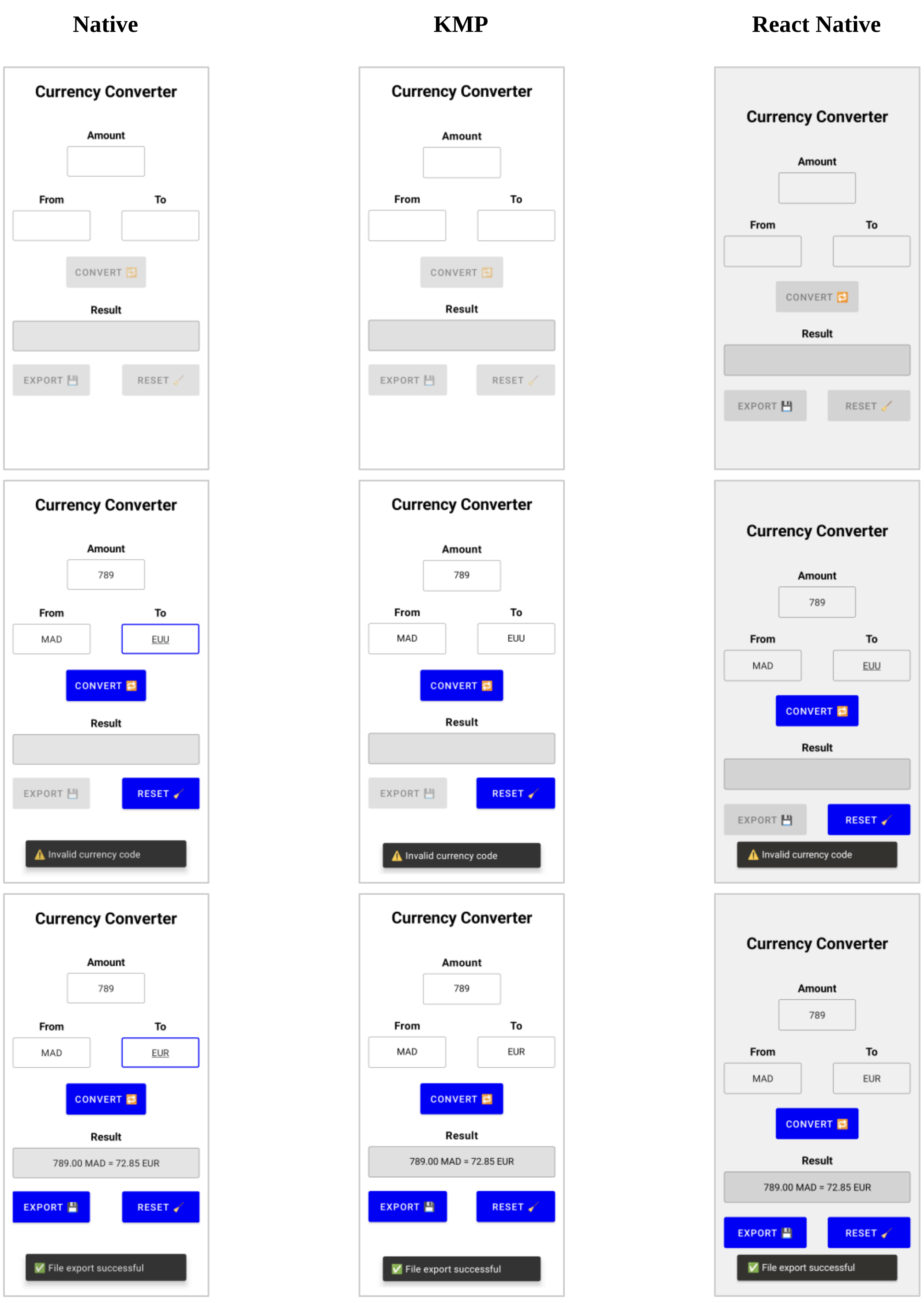
Figure 19: *Code segment of title UI component in React Native*



```
/*** TITLE block ***/
Text(
    text = Strings.APP_NAME,
    style = MaterialTheme.typography.h5,
    fontWeight = FontWeight.Bold,
    modifier = Modifier
        .padding(top = 8.dp, bottom = 26.dp)
        .align(Alignment.CenterHorizontally)
)
```

Figure 20: *Code segment of title UI component in KMP*

Despite the various implementations of the layout, it was possible to build the UI according to the sketch presented in Section 3.3. The layout was implemented in a vertically linear manner and segmented into six blocks, within which the individual layout elements were positioned. These blocks encompass the title block featuring the title label, the amount block comprising the amount label and text field, the from/to block housing base and target currency labels and text fields, the convert block containing the convert button, the result block holding result label and text field, and the export/reset block incorporating export and reset buttons. Figure 21 showcases the application screens of all three implemented sample applications, illustrating the following three distinct states: initial app screen after opening, conversion attempt with invalid currency code, and successful export confirmation.



Invalid currency code

789.00 MAD = 72.85 EUR

File export successful

Figure 21: *Screenshots of the three sample applications in different operational states*

When investigating the individual screenshots of the different sample applications, it is evident that a striking similarity in appearance could be accomplished. Nevertheless, subtle differences can be detected. For instance, the React Native application features a default light gray background, which curiously appeared white when launched for debugging, but transitioned to gray in the release version of the application. Also, in React Native, a larger margin is noticeable at the top of the screen compared to Native and KMP, moderately shifting all layout elements downward. This margin was on a smaller scale during debugging and more similar to the other two applications during that phase. This suggests that unexpected alterations may occur during the generation of the release version in React Native. Another disparity is that upon pressing the convert button, the text field remains active in Native and React Native, whereas it loses focus in KMP.

In order to install the applications on Android devices in their release versions, APK files had to be created. The process was identical for both Native and KMP. It was imperative that the APKs were signed. Signing an APK file involves attaching a digital signature to verify the authenticity and integrity of the application, ensuring no modifications were made during distribution. A cryptographic key pair with a validity of 10,000 days was generated using the `keytool` command. Following this, the `build.gradle` file was modified by adding the signing configurations with the keystore information. This ensured that the release build type utilized these configurations to sign the APK during the build process. The generated keystores and the corresponding `keytool` commands and outputs are provided in Appendix B7. The APK files were created in Android Studio via the build menu. The creation of the signed APK file in React Native had to be managed through the Expo account. The free plan in Expo includes 30 APK builds per month, which was sufficient for this thesis. For the build of the release APK file, the command `expo build:android` had to be executed in the terminal. Expo generated the APK file in its cloud with proper signing and provided a download link once the build was complete. As previously mentioned, all APK files are available in Appendix B2.

5.2. AI-Assisted Coding Experience

In this thesis, ChatGPT 3.5 was utilized as AI to assist in the coding process. StarCoder was also experimentally tested in the initial stage. Unfortunately, it was found that StarCoder could not keep up with ChatGPT. While it was able to answer or explain technical code questions just as effectively, it struggled to maintain the desired context in a chat conversation. This means that while it could generate small code segments correctly, it often required repeatedly providing context regarding the application, technology, version, etc., resulting in excessive effort. While StarCoder may be a viable alternative in other circumstances, it appeared to be inferior to ChatGPT in the use case of this thesis. ChatGPT was also chosen over an AI-powered IDE-plugin, given that all technologies were entirely new to the author. The absence of prior knowledge of the project structure, syntax, and technology stack would have limited the effectiveness of an IDE plugin, as it heavily relies on contextual familiarity to provide accurate suggestions. In contrast, ChatGPT's natural language processing capabilities allowed for more flexible and conversational interactions, enabling users to seek assistance without needing extensive project-specific knowledge. This approach was considered particularly beneficial for novices in a specific technology, as it provides a more accessible and user-friendly means of accessing coding assistance.

For traceability reasons, all chat conversations with ChatGPT were exported in HTML format to maintain the original structure. New conversations were started, as ChatGPT's response time slowed down with longer conversations, likely due to the consideration of the entire or a substantial portion of the chat history as context. The HTML files containing the chat logs were named based on the utilized technology and sequentially numbered. In subsequent descriptions, the files are referenced by their names, such as *Native-v1* or *KMP-v3*. All chat logs are provided in their entirety in Appendix B8. In subsequent descriptions Table 13 offers an overview of the coding activities assisted in each chat log for reference. In the following, different aspects of the coding experience are illuminated, including the skill building and troubleshooting process, as well as weaknesses and errors encountered during the development with ChatGPT.

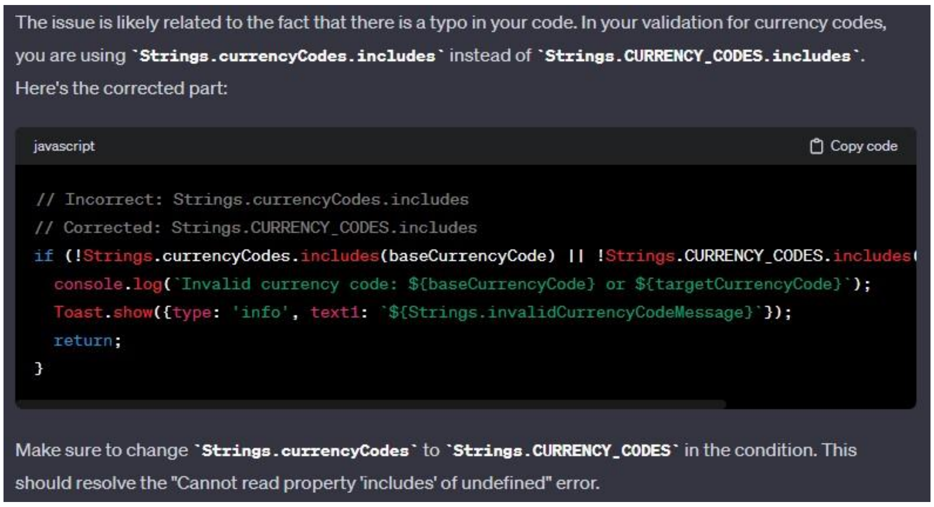
Table 13: *Coding activities in respective Chat GPT chat histories*

	Native			**KMP**				**React Native**			
	v1	**v2**	**v3**	**v1**	**v2**	**v3**	**v4**	**v1**	**v2**	**v3**	**v4**
Basic UI setup	x			x				x			
UI enhancements		x	x	x		x	x		x		
Input validation		x	x			x			x		
Notification banner		x	x			x			x		
Utils		x		x					x		
API service	x			x	x				x		
Export service		x	x			x	x			x	x
Reset of fields			x						x		

In the initial stages of encountering a completely unfamiliar technology, a systematic procedure was developed to build skills effectively. After recognizing that attempts to generate code for an entire mobile application, even when only encompassing a moderate feature scope, would not produce functioning code results, which, due to the lack of understanding, could not be corrected autonomously, an incremental approach was adopted. First, a minimal introduction to the topic was sought through resources independent from AI, such as tutorial videos for a basic introduction into the technology domain. Armed with a basic understanding, the AI-assisted coding journey was initiated by creating an empty project as a starting point. From there, the project was, with the help of AI, gradually extended to the specified sample application. New functionalities were integrated step by step and at each stage, assistance from AI was utilized. However, it was observed that requesting too many details from ChatGPT at once could overwhelm its ability to incorporate them all in its response. For instance, in *KMP-v1*, the entire XML layout from the Native app was fed into ChatGPT, with a request to translate it into Compose components for KMP. However, this attempt failed, as it did not produce functional code. As a result, the process was restarted, beginning with a single component, similar to the approach in Native, and gradually extending the layout by small portions. When working in such small increments, functional code was generated very frequently. Instances where this was not the case are addressed later on.

Skill building in the AI-assisted coding process was supported in several ways, despite encountering some obstacles. Due to the mentioned constraints on input complexity and the necessity for step-by-step progression, a systematic extension approach was required, which encouraged autonomous decision-making to determine each next logical step. At each step of the coding process, the AI provided assistance, often accompanied by brief technical explanations to aid comprehension. Therefore, knowledge was acquired automatically by making active adjustment in order to integrate the generated code snippets. Moreover, in case of uncertainties, it was always possible to inquire again, for instance about guidance on where exactly to integrate generated code or clarification on the functionality of certain methods or objects. This proved to be very valuable, especially when encountering difficulties with the previous response, as it facilitated troubleshooting and skill development through hands-on problem-solving. Thus, especially when faced with responses that did not produce the desired output, significant skill-building occurred during the process of self-directed troubleshooting. In such instances, the need to independently diagnose and resolve issues fostered a deeper understanding of the technology.

The troubleshooting process during the AI-assisted coding involved various steps to resolve issues efficiently. Upon encountering error messages after inserting generated code, inputting these into ChatGPT often resulted in either solving or at least revealing the issue. There were also instances where a larger code segment from the current project was provided, and ChatGPT identified a typo, which turned out to be the problem. For instance, in *ReactNative-v2*, the entire code of the `App` class was input and ChatGPT found a typo in a variable name (`Strings.CURRENCY_CODES` instead of `Strings.currencyCodes`), illustrated in Figure 22. In other cases, following up with additional inquiries helped, whether to clarify further, remind of specific requirements or circumstances relevant to the task, even if they had been communicated previously. For example, in *Native-v1*, it had been clarified earlier that the way of accessing UI elements is done via View Binding, but in a later code generation, the less modern approach of `findViewById` was used instead of the initialized `binding` instance. After a brief reminder, the correct code using View Binding was generated. In cases where the problem persisted despite multiple inquiries, it was transitioned to searching for solutions online, often referring to resources such as official documentation or Stack Overflow, a widely used Q&A platform for programmers. Such searches for solutions online occasionally led to a resolution on its own. Alternatively, the retrieved external resources were re-entered into ChatGPT to receive assistance in creating a concrete code solution that is compatible with the existing code and meets the specific requirements of the application. Examples illustrating these situations can be found in *KMP-v4*, regarding the iOS file access issue, and in *ReactNative-v1*, concerning the auto-capitalize topic for text fields.


The issue is likely related to the fact that there is a typo in your code. In your validation for currency codes, you are using **`Strings.currencyCodes.includes`** instead of **`Strings.CURRENCY_CODES.includes`**. Here's the corrected part:

```javascript
// Incorrect: Strings.currencyCodes.includes
// Corrected: Strings.CURRENCY_CODES.includes
if (!Strings.currencyCodes.includes(baseCurrencyCode) || !Strings.CURRENCY_CODES.includes(
  console.log(`Invalid currency code: ${baseCurrencyCode} or ${targetCurrencyCode}`);
  Toast.show({type: 'info', text1: `${Strings.invalidCurrencyCodeMessage}`});
  return;
}
```

Make sure to change **`Strings.currencyCodes`** to **`Strings.CURRENCY_CODES`** in the condition. This should resolve the "Cannot read property 'includes' of undefined" error.

Figure 22: *Screenshot of ChatGPT identifying typo*

Ultimately, weaknesses and errors encountered during coding with ChatGPT are elaborated. Generally, the encountered issues can be sorted into the following categories:

- contextual incompleteness,
- versioning and compatibility issues,
- occasional lack of contextual recall, and
- unnecessary complexity.

One notable concern was the incompleteness of code suggestions, particularly regarding import statements. ChatGPT sometimes omitted the corresponding import statements, leaving gaps in the code that needed to be addressed manually, by requesting the missing statements or following import recommendations made by the IDE. Examples of this issue can be found in *KMP-v1*, where import statements had to be requested afterwards for the Preview UI feature. Another area of concern revolved around versioning and compatibility. Due to the restricted knowledge of ChatGPT-3.5, extending only up to January 2022, it frequently recommends outdated versions of libraries. In reaction, each instance was cross-referenced with the latest versions available in the official documentation of the corresponding library, ensuring accurate and up-to-date versions. Furthermore, ChatGPT occasionally disregards modern approaches within its knowledge and instead resorts to less modern alternatives, for example in *Native-v1*, regarding the previously mentioned View Binding incident, or in *Native-v2*, continuously using the wrong syntax for dependencies defined in the `build.gradle.kts` file, specifically `implementation '...'` instead of `implementation("...")`. This led to difficulties in maintaining compatibility with the latest standards and practices and, therefore, with other components of the existing codebase. Moreover, contextual recall emerged as both a strength and also an occasional weakness of the AI system. On one hand, ChatGPT demonstrated impressive memory capabilities, recalling variable names and method or class content from previous interactions. However, there were instances where it failed to remember previously used approaches, requiring reminders. ChatGPT also sporadically, during troubleshooting, tended to supply redundant suggestions, as it circled back to previously suggested solutions that had already been deemed ineffective. For instance, in *KMP-v1* and *KMP-v3*, it persistently suggested Java libraries, despite multiple prior communication that these are not supported within the shared code section in KMP. A similar situation occurred in *ReactNative-v3*, where libraries for file access were repeatedly suggested, despite their incompatibility with Expo and the repeated feedback that they do not work. Lastly, unnecessary complexity was observed in some of the code suggestions provided by ChatGPT. For instance, in *ReactNative-v1*, instead of simply setting the `autoCapitalize` property, a more complex implementation of the `handleTextChange` method was suggested for capitalizing letters in text fields. Beginners might not always recognize when overly complex options are suggested and may unintentionally pursue them. Another example occurred for *KMP-v1* and *KMP-v2*, where ChatGPT suggested unnecessary platform-specific implementations in KMP, despite the possibility of implementing them in the shared section of the codebase, which deviates from the principles of the cross-platform approach.

6. Metric-Based Performance Comparison

After performing measurements with the implemented sample applications on the specified devices and following the defined test sequences as detailed in Section 3.4.1, the results are presented below. A compilation of the measured performance metrics is summarized in table form, including means and standard deviations (represented with M and SD) across the different technologies and devices. Furthermore, bar charts depict the mean performance for every metric of the technologies on each device, in order to visualize the presence or absence of differences. The outcomes of ANOVA and effect size calculations are also portrayed in tables, offering a comprehensive overview of the comparative performance characteristics. In the ANOVA tests, statistically significant differences are present when the ANOVA p-value is below 0.05. For simplicity, in the upcoming tables, binary indicators (checkmark or cross) are used to denote the presence of significant differences. In cases of statistical distinctions, the extent of the differences is indicated by the effect size, which is rounded to one decimal place. In ANOVA and effect size tables, the results values are unified in one cell when they align across all devices. In case of mismatch, the cell is divided into three, and the results for the three devices are recorded in the sequence of Motorola G10, Google Pixel 4a, Motorola G54. The interpretation of effect sizes was conducted in the context of the measurements and the applied scale classifies effects as small (< 0.3), medium (≥ 0.3), and large (≥ 0.6) in the subsequent descriptions. The highest recorded effect size in all measurements was 1, the smallest 0.1. For space-saving purposes, the CPMD technology React Native has been abbreviated as RN. For the same reasons, the device names were abbreviated as well, omitting the brand name and only specifying the model.

Package file size and installed application size

Table 14 provides insights into the APK file size and installed application size in megabytes for each sample application built with each technology. Since testing was exclusively carried out on the Android devices, the package files come in the form of APK files. The installed application size values represent means across the three distinct devices. Variations in device storage allocations, and hardware configurations contribute to slight differences in installed application sizes.

Table 14: *APK file size and installed application size*

Technology	APK file size (MB)	Installed application size (MB)
Native	6.19	18.97
KMP	6.06	23.27
RN	26.46	55.77

Native and KMP demonstrate the best efficiency in APK file size, with KMP slightly edging out Native. In contrast, React Native's APK file size surpasses the other two by more than four times. A similar trend, though not proportionate, is observed in the installed application size. Native and KMP maintain striking similarities, Native being notably smaller this time by 18%. On the other hand, React Native's installed application size exceeds the other two by over twice as much.

Startup time

Table 15: *Startup time – Mean and standard deviation by technology and device*

Startup time (ms)						
Device	Native		KMP		React Native	
	M	SD	M	SD	M	SD
Moto G10	618.32	23.44	839.69	10.62	706.80	52.04
Pixel 4a	344.46	51.02	480.61	53.38	411.35	19.58
Moto G54	376.30	19.52	458.43	16.99	420.20	18.68

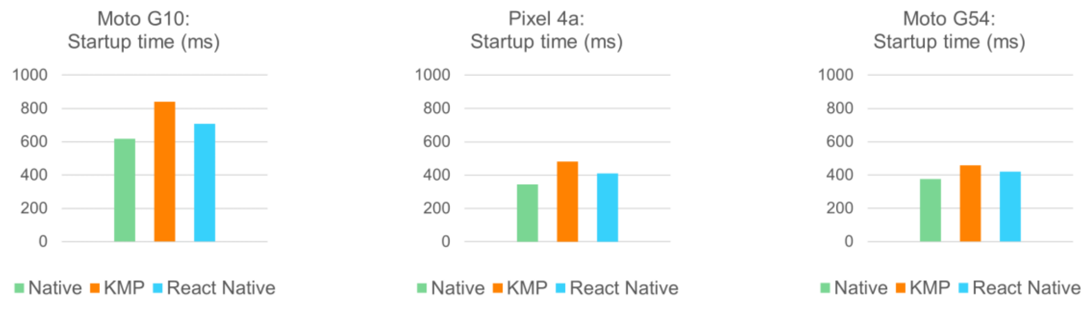


Figure 23: *Startup time – Means per technology grouped by devices*

Table 16: *Startup time – ANOVA and effect size results by technology pair and across devices*

Startup time		
Comparison subjects	**ANOVA-p**	**Effect size**
Native / KMP	✓	0.5 – 1.0
Native / RN	✓	0.5 – 0.8
KMP / RN	✓	0.4 – 0.6

Table 15 displays the means and standard deviations of the measured startup times in milliseconds per technology and device, while Figure 23 provides a visual representation of the means grouped by technology and segmented by devices. Moto G10 exhibits remarkably higher startup times in general, while Pixel 4a and Moto G54 show similar lower startup times, approximately 300 ms less on average than Moto G10. KMP records the highest startup time across all devices, whereas Native has the lowest. The variations in startup times on Moto G54 are minor, whereas Moto G10 shows the highest differences. Based on ANOVA calculations presented in Table 16, significant differences in startup time are observed among all technologies on all devices, with effect sizes ranging from small to large.

Network traffic

Table 17: *Network traffic – Mean and standard deviation by technology and device*

	Network traffic (MB)						
	Device	**Native**		**KMP**		**RN**	
		M	**SD**	**M**	**SD**	**M**	**SD**
Scenario 1	Moto G10	8.42	0.11	8.77	0.11	8.79	0.07
	Pixel 4a	8.20	0.11	8.23	0.14	8.21	0.11
	Moto G54	8.21	0.05	8.26	0.06	8.23	0.09
Scenario 2	Moto G10	8.15	0.12	0.67	0.06	0.69	0.07
	Pixel 4a	8.07	0.13	0.72	0.08	0.71	0.09
	Moto G54	8.08	0.11	0.70	0.08	0.68	0.08

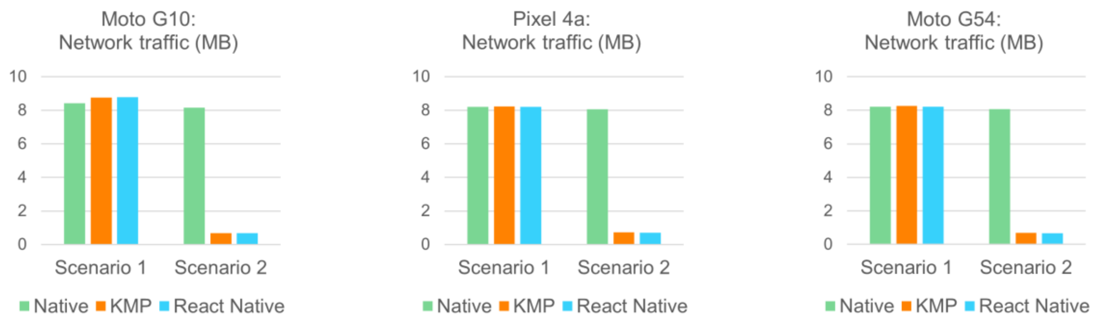


Figure 24: *Network traffic – Means per technology grouped by devices*

Table 18: *Network traffic – ANOVA and effect size results by technology pair and across devices*

Network traffic								
Comparison subjects	**Scenario 1**						**Scenario 2**	
	ANOVA			**Effect size**			**ANOVA**	**Effect size**
Native / KMP	✓	x	✓	0.7	-	0.8	✓	1.0
Native / RN	✓	x	x	0.2	-	-	✓	1.0
KMP / RN	x			-			x	-

The means and standard deviations of network traffic in megabytes across technologies and devices for each of the two scenarios are summarized in Table 17, complemented by the graphical representation in Figure 24 with the average of the metrics grouped by technology and divided by devices. It immediately stands out that Native, in scenario 2, the subsequent conversion, produces a very similar amount of network traffic across all devices during the subsequent conversion as it did for the initial one. Conversely, KMP and Native generate minimal network traffic in the second conversion on all devices, reaching less than a tenth of the traffic generated in the initial conversion. The ANOVA calculations provided in Table 18 indicate that there are no significant differences between KMP and React Native across all devices in both scenarios. When comparing Native to both KMP and React Native in the second scenario, strong statistically significant differences are observed on all devices, with maximum effect sizes. In the comparison of Native to both KMP and React Native in the first scenario, however, significant differences only partially occurred, varying across devices. On the Moto G10, both CPMD technologies significantly differed from Native, with Native performing superiorly, whereas on the Pixel 4a, no significant distinctions were observed. On the Moto G54, significant differences were noticeable solely between Native and KMP.

CPU usage

Table 19: *CPU usage – Mean and standard deviation by technology and device*

	CPU usage (%)						
	Device	Native		KMP		React Native	
		M	SD	M	SD	M	SD
Scenario 1	Moto G10	19.03	2.93	23.46	0.87	31.58	1.57
	Pixel 4a	11.20	0.56	16.09	1.71	25.33	1.84
	Moto G54	14.99	1.73	18.22	1.26	26.03	1.26
Scenario 2	Moto G10	18.44	1.68	31.81	1.87	19.33	1.57
	Pixel 4a	13.87	2.04	19.79	4.02	15.69	1.84
	Moto G54	14.77	5.16	25.39	4.02	16.15	1.26
Scenario 3	Moto G10	9.03	2.57	18.77	2.16	24.97	0.05
	Pixel 4a	5.09	0.80	10.95	1.05	18.31	2.66
	Moto G54	7.24	2.84	13.16	2.52	15.69	5.35

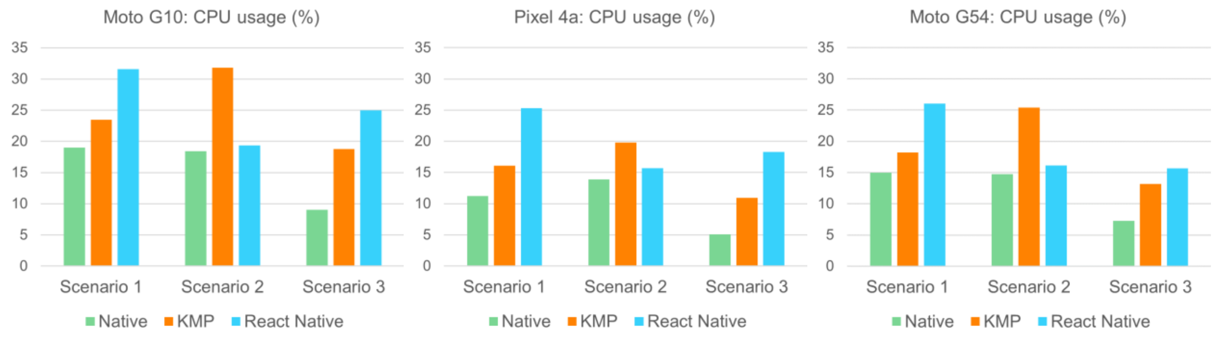


Figure 25: *CPU usage – Means per technology grouped by devices*

Table 20: *CPU usage – ANOVA and effect size results by technology pair and across devices*

CPU usage																		
Comparison subjects	Scenario 1						Scenario 2						Scenario 3					
	ANOVA			Effect size			ANOVA			Effect size			ANOVA			Effect size		
Native / KMP	✓			0.5	0.8	0.5	✓			0.9	0.5	0.6	✓			0.8	0.9	0.5
Native / RN	✓			0.9	1.0	0.9	x	✓	x	-	0.1	-	✓			1.0	0.9	0.5
KMP / RN	✓			0.9			✓			0.9	0.3	0.7	✓	✓	x	0.8	0.8	-

In Table 19, the mean values and standard deviations corresponding to CPU usage in percent across different technologies and devices can be found, with Figure 25 offering a graphical illustration of these means categorized by technology and partitioned by devices. A clear pattern is visible regarding the CPU performance in each scenario across all devices. Native demonstrates notably superior performance in Scenarios 1 and 3, while React Native performs the least effectively. However, in

Scenario 2, KMP distinctly falls behind, while Native and React Native exhibit similar performance levels, with Native taking the lead. As per ANOVA calculations listed in Table 20, statistical differences are found among all technologies on all devices in Scenario 1, characterized by medium to large effect sizes. On the Pixel 4a, notably, significant differences are observed among all technologies in all three scenarios. Conversely, in scenario 2, on the Moto G60 and Moto G54, performance variations between Native and React Native lack statistical significance. In Scenario 3, only on the Moto G54, no significant disparities are observed between the two CMPD technologies. Additionally, alongside the CPU usage comparison outlined above, complete CPU charts from a single instance are available in Appendix A1 in order to gain profound insights beyond the peak value consideration. The illustrative comparison of CPU charts, considering only one single instance, reveals that in scenario 1, an evenly spread level of CPU usage was observed across the technologies, with React Native slightly exhibiting the highest usage. However, in scenario 2, there is a notable difference, with KMP consistently consuming more CPU than both Native and React Native. In scenario 3, React Native demonstrates the lowest overall CPU consumption, although with a minor peak, while Native and KMP exhibit consistent usage throughout.

Memory usage

Table 21: *Memory usage – Mean and standard deviation by technology and device*

	Memory usage (MB)						
	Device	Native		KMP		React Native	
		M	SD	M	SD	M	SD
Scenario 1	Moto G10	30.31	0.89	29.70	0.32	61.07	0.26
	Pixel 4a	26.15	0.11	26.68	1.55	51.64	0.19
	Moto G54	45.13	0.35	45.49	0.30	70.36	2.33
Scenario 2	Moto G10	33.95	0.16	44.88	2.24	63.50	0.26
	Pixel 4a	35.87	0.25	52.01	1.19	70.97	0.19
	Moto G54	50.67	0.45	59.58	1.54	83.21	2.33
Scenario 3	Moto G10	35.29	0.88	44.01	1.99	72.37	0.54
	Pixel 4a	36.22	0.21	44.83	0.62	69.67	1.50
	Moto G54	51.14	1.32	53.89	0.70	79.95	2.26

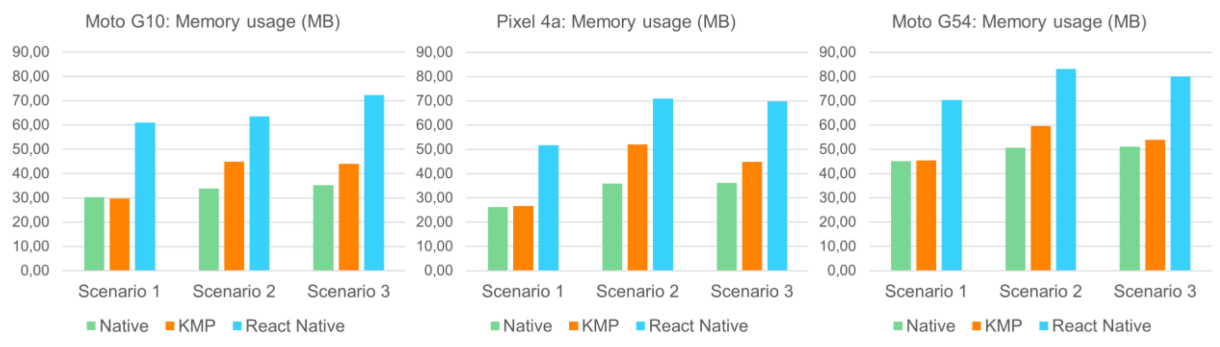


Figure 26: *Memory usage – Means per technology grouped by devices*

Table 22: *Memory usage – ANOVA and effect size results by technology pair and across devices*

Memory usage						
Comparison subjects	**Scenario 1**		**Scenario 2**		**Scenario 3**	
	ANOVA	**Effect size**	**ANOVA**	**Effect size**	**ANOVA**	**Effect size**
Native / KMP	✓ / x / ✓	0.2 / - / 0.2	✓	0.9 / 1.0 / 0.9	✓	0.9 / 1.0 / 0.6
Native / RN	✓	1.0	✓	1.0	✓	1.0
KMP / RN	✓	1.0	✓	1.0	✓	1.0

Table 21 presents the mean and standard deviation measurements for memory usage in megabytes across technologies and devices, while Figure 26 visually represents the means per technology, distinguished by devices. React Native consistently emerges as the underperformer in memory usage across all scenarios and devices, while Native is clearly leading. In Scenario 1, KMP closely matches Native and even slightly outperforms it once, on Moto G10. The ANOVA and effect size calculations presented in Table 22 confirm the observations made. In scenario 1, either significant differences with small effect sizes are noted between Native and KMP (Moto G10, Moto G54), or no significant differences are observed (Pixel 4a). Otherwise, statistically significant differences with large effect sizes can be consistently observed between all technologies across all scenarios and devices.

Battery usage

Table 23: *Battery usage – Mean and standard deviation by technology and device*

	Battery usage (MA)						
	Device	Native		KMP		React Native	
		M	SD	M	SD	M	SD
Scenario 1	Moto G10	-518,48	61,76	-547,95	80,98	-615,12	108,65
	Pixel 4a	-793,93	123,52	-865,23	102,08	-969,44	194,60
	Moto G54	-600,81	121,53	-644,83	121,17	-691,48	99,12
Scenario 2	Moto G10	-502,97	65,14	-566,03	70,84	-537,44	108,65
	Pixel 4a	-767,84	108,84	-986,53	109,84	-821,33	194,60
	Moto G54	-659,56	80,26	-633,48	409,54	-678,79	99,12
Scenario 3	Moto G10	-450,97	43,29	-439,37	64,75	-615,69	138,47
	Pixel 4a	-626,42	86,60	-711,12	157,16	-789,61	161,42
	Moto G54	-601,98	110,70	-615,19	137,21	-695,57	461,29

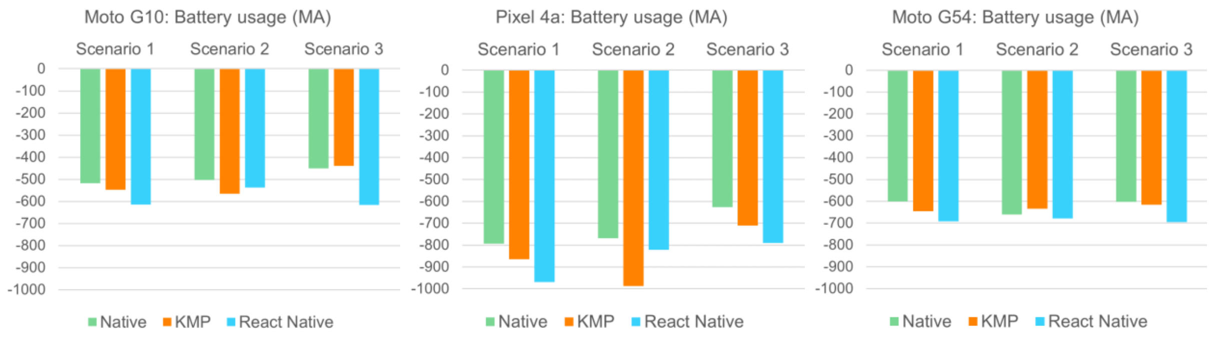


Figure 27: *Battery usage – Means per technology grouped by devices*

Table 24: *Battery usage – ANOVA and effect size results by technology pair and across devices*

Battery usage																		
Comparison subjects	Scenario 1						Scenario 2						Scenario 3					
	ANOVA			Effect size			ANOVA			Effect size			ANOVA			Effect size		
Native / KMP	x			-			✓	✓	x	0.2	0.5	-	x			-		
Native / RN	✓			0.2	0.2	0.1	x			-			✓	✓	x	0.4	0.3	-
KMP / RN	x			-			x	✓	x	-	0.3	-	✓	x	x	0.4	-	-

Table 23 exhibits the averages and standard deviations for battery usage in megaampere measured across technologies and devices, complemented by the graphical depiction of these averages organized by technology and grouped by devices in Figure 27. In most cases, Native tends to a slightly superior performance across all scenarios and devices, while React Native shows the lowest performance. Exceptions are scenario 2 on Moto G10 and Pixel 4a, where KMP is weakest performing, and scenario 3 on Moto G10 and scenario 2 on Moto G54, where Native only ranks as second-best performer. In summary, all differences between applied technologies appear mostly marginal. Remarkably, the Pixel 4a consistently demonstrated higher battery consumption overall, while the least consumption was observed on the Moto G10, with the Moto G54 performing slightly less efficiently. The ANOVA calculations in Table 24 show that over half of the comparisons do not exhibit statistically significant differences. The only significant differences can be observed in the following instances: between Native and React Native in scenario 1 across all devices and in scenario 3 on the Pixel 4a and Moto G54, between Native and KMP exclusively in scenario 2 on the Moto G10 and Pixel 4a, and between KMP and React Native in scenario 1 on the Pixel 4a and in Scenario 3 on the Moto G10. Overall, the effect sizes of these significant differences range from small to moderate.

As previously described, the presented values were aggregated to the arithmetic mean, ensuring a comprehensive overview. The individual measurements for each iteration per metric can be found in Appendix A2, while the precise results of ANOVA and effect size calculations can be found in Appendix A3. Fully detailed data and calculations, including all ANOVA results tables, are enclosed as Excel files in Appendix B9 for reference.

7. Discussion

Following the successful implementation of mobile sample applications with AI assistance and their evaluation through metric-based comparison, this discussion delves into the comprehensive findings and implications across these multifaceted domains. Furthermore, limitations encountered and implications for future research are elaborated upon.

7.1. AI-Assisted Coding Experience

The development process encompassed three distinct technologies, all of which were entirely new to the author, with a primary focus on skill-building. With the assistance of AI, it was found to be feasible to develop the sample applications according to the defined requirements, achieving very similar look and functionality across all technologies. Notably, React Native presented the most obstacles due to dealing with incompatibilities and limitations arising from the Expo framework, as detailed in Section 7.3. However, despite these challenges, React Native also facilitated the fastest application development process. Additionally, React Native was noted for its simple project structure and layout implementation, which made it easier and faster to learn compared to Android and KMP. These development experiences could be the reason why React Native is found to be the most popular approach in practice, as discussed in Section 2.2.3. Regarding the efficiency of the project size, KMP fell short in supporting 100% shared code, necessitating platform-specific implementation for one service, resulting in a 37.5% shared code portion. This contradicts JetBrains's claims regarding KMP achieving 100% shared code when certain scenarios lack shared support.

Various key findings regarding the effectiveness, efficiency, and user experience of using AI in coding tasks emerged from this research. Development without prior knowledge of the technology was perceived as more efficient compared to traditional tutorials, which do not precisely align with one's concrete objectives. AI assistance offered distinct advantages over resources such as Stack Overflow or generic tutorials by providing tailored code snippets. However, multiple inquiries were frequently required, demanding both patience and a solid understanding of the required code, which is not self-evident among beginners. This personalized approach fostered higher learning motivation, enabling users to build custom tutorials for their specific purposes with AI assistance. Despite the support by AI in skill building and accelerating development for technology novices, it demonstrated several weaknesses, necessitating technology understanding and troubleshooting. This aligns with studies from Section 2.3 evaluating code generation success rates, with ChatGPT achieving a success rate of 65%, which only amounts to slightly more than half. Given that other AI tools exhibited lower performance in those studies, ChatGPT had proven to be the most optimal option by being the best performer for AI-powered code generation. The interactions with ChatGPT clearly demonstrated the importance of the provided input and context by the user. This aligns with the significance of prompt engineering described by Becker et al. (2023, p. 504) in the context of LLMs. Additionally, the encountered weaknesses of ChatGPT that lead to troubleshooting had the positive side effect that the learning effect was greatest in these situations. Estimating the degree of time saved due to AI assistance in comparison to traditional methods is challenging, as there was no direct comparison available. Therefore, the exact volume to which development time would have been extended without AI assistance remains unclear. Moreover, it is noteworthy that certain errors or challenges encountered with AI might have similarly occurred in conventional tutorials, including issues associated with outdated versioning and compatibility.

This study also identified potential risks associated with AI's improving accuracy and generating flawless code, which could lead to a passive development and learning approaches characterized by code copying without comprehension. Instances of this phenomenon were observed in minor aspects in this project when code was working perfectly. This highlights the importance of active engagement in the learning process. Furthermore, this study revealed that while AI assistance facilitated the development process, it was never a completely smooth process, often requiring autonomous initiative. Becker et al. (2023) emphasize the potential for AI to enable a more effective approach to learning and teaching coding. This study echoes their findings by showing that customized tutorials tailored to individual skill levels and interests could enhance learning effectiveness. In this study, the AI support enabled a hands-on learning approach by coding with limited prior knowledge of the technologies, fostering skill building, especially during troubleshooting. However, caution is warranted to prevent overreliance on AI-generated code, particularly in time-sensitive scenarios.

Practical implications extend to developers, researchers, and educators in the computer science and software development domain. The study underscores the role of AI in supporting tailored application development while highlighting encountered difficulties. These insights are valuable for guiding future strategies in integrating AI into educational and professional settings, maximizing its benefits. The expectation is that AI will continue to improve in accuracy, therefore the heightened risk of potential misuse emerges, and the skills normally acquired through troubleshooting would decrease as a result of less troubleshooting in general. In an educational context, this could entail a significant shift in teaching tasks, as Becker et al. (2023) already suggested. The experiences gained with AI hold critical significance not only for computer science educators and students but also for the broader field of software development, given the inevitable integration of AI into our everyday lives. In various contexts, such as industry, private, or academia, this study provides evidence of the effectiveness of AI in supporting application development tailored to the needs of programmers, while also illuminating encountered difficulties. These insights serve as valuable lessons to learn from, enabling the prevention of similar issues in future application.

7.2. Metric-Based Performance Comparison

In the metric-based performance comparison of the sample applications, each metric exhibited varying performance, with occasionally differing scenarios as well. Nevertheless, general trends can be detected. Neglecting some individualities in devices and scenarios, a simplified ranking can be derived from the results. To provide a rough overview of the overall performance of the respective technologies, Table 25 presents, in a simplified manner, how the CPMD technologies performed overall, indicated by a ranking. If the differences were not deemed remarkable, the ranking number was shared.

Table 25: *Simplified performance ranking of CPMD technologies for each metric*

	Native	**KMP**	**RN**
Package file size	1	1	2
Installed file size	1	2	3
Startup time	1	3	2
Network traffic	1	1	1
CPU usage	1	2	3
Memory usage	1	2	3
Battery usage	1	2	3

In terms of APK file size and installed application size, React Native unquestionably trails behind in both metrics. Native and KMP exhibit similarity in APK file size, while Native excels in installed application size. Regarding startup time, Native demonstrates the best performance, with React Native showing better results than KMP. This is the only instance where React Native outperforms KMP overall. While the level of network traffic remains relatively consistent across all technologies, Native stands out for consistently establishing a new connection during the second conversion, resulting in significantly higher network traffic compared to the CPMD technologies. This suggests that optimizations present in the libraries used by the CPMD technologies may be lacking in Native or would have required manual optimization. The majority findings of the comparison studies in Section 4.2, stating that React Native is typically highly CPU- and memory intensive, are supported through this performance analysis. React Native mostly exhibits the highest CPU usage, during startup and export and consistently underperforms for memory usage. KMP demonstrates the highest CPU consumption for API requests during conversions. While KMP incurs memory usage overhead compared to Native, it is not excessively high. Concerning battery usage, Native solutions generally consume the least amount of battery, while React Native tends to consume the most. However, there are instances where KMP has the highest battery usage during conversions, consistent with its CPU usage pattern. KMP does not consistently outperform React Native across all areas. Clear evidence of performance overhead with KMP is apparent in startup time and CPU usage during conversions. The former align with the findings by Evert (2019, p. 32). In summary, KMP exhibits, in most cases, a slight yet significant overhead, typically lower than that of React Native. Differences independent of the technologies related to the devices were also observed. On the oldest device model Moto G10, an overall longer startup time and higher CPU usage can be observed in comparison to the other two devices. This suggests that the device exhibits inefficient resource utilization due to outdated hardware, ultimately resulting in lower overall performance. Notably, the Pixel 4a and Moto G54 devices exhibit very similar levels across all metrics. Furthermore, the Pixel 4a showed a considerably higher battery consumption compared to the other two devices. This possibly denotes that the Pixel 4a device may have inefficient software or operating system optimizations that lead to increased energy consumption. Additional background processes from installed applications on the user's device could also increase battery usage compared to newly set-up Motorola devices with factory settings. Mahendra and Anggorojati (2021, p. 37) encourage the exploration of other frameworks and metrics measurement on real devices instead of emulators, while You and Hu (2021, p. 9) advocate for the use of multiple test devices. With the methodology pursued, all of these suggestions were followed.

ANOVA occasionally reveals significant differences due to a small overlap of the values distributions, but in practice, they can be considered on the same level, even if statistically significant. For the performance metrics comparison, all findings could already be identified based on the descriptive depiction, perceptible from the bar charts. While ANOVA confirmed these observations, it is questionable whether the effort required for extensive calculations is justified. The time spent on these calculations could be invested more wisely in other areas. The metrics for the evaluation in this study were selected based on an extensive literature review in Section 4.2, hence the applied metrics are considered good choices that resulted in interesting and diverse outcomes. Equally beneficial was the decision to divide the test sequences into scenarios, allowing for a distinct examination of the application’s various activities. Since only one KMP comparison study could be identified in the literature review, which explicitly encouraged the use of more metrics, this study represents a valuable contribution to the scientific understanding of KMP.

7.3. Limitations and Future Research

In conducting this study, several limitations were encountered, which are acknowledged to provide context and transparency. The encountered limitations are as follows: a low number of comparative technologies, application scope and metrics limitations, API testing restrictions, device and platform limitations, manual measurement risks, and limited skill set. The study primarily focused on comparing KMP with React Native as the sole established CPMD technology. Alternative CPMD technologies with varying development approaches could have been considered, broadening the scope of the comparison. The performance measurement was restricted to the selected metrics and application features implemented in the sample applications. This limitation suggests a potential for future studies to explore a broader range of metrics and features, or to develop different types of applications for benchmarking purposes. Due to limitations in free API availability, extensive testing scenarios were constrained because of the limited number of available API requests. Future research could involve using alternative API providers to conduct more comprehensive testing. The study was also limited to testing on a specific set of devices and the Android platform only. Future research could expand the testing scope to include a wider variety of devices and platforms, such as iOS, to provide a more comprehensive analysis. The evidence provided by Biørn-Hansen, Grønli and Ghinea (2019, p. 16) and Evert (2019, p. 32) illustrates that differing performance levels can arise between the Android and iOS platforms. Since all measurements were performed manually, and thus required a high level of accuracy, there is a risk of human error. However, considering that the data does not contain notable outliers and that observable patterns exist throughout, this risk is assessed as relatively low. Still, future studies might want to explore automated testing methods to enhance accuracy and efficiency, while taking into account the necessary time and effort. Furthermore, the lack of previous skill regarding all employed technologies, the learning curve to compensate the lack of skill, and AI-introduced challenges encountered during development led to slower progress, increased debugging time, and potentially more errors in implementation. In future studies, it is generally advisable to possess prior foundational knowledge to mitigate the degree of unpredictability.

Several challenges, including underestimated high manual effort and technical hurdles, were encountered during the execution of this study, which impacted the methodology and outcomes. The manual effort required for measurement and testing was substantial, leading to time constraints, which consequently, for example, prevented the implementation of a Native iOS sample application, another CPMD technology for comparison, and the evaluation on an additional device. Future research should consider allocating time for implementing automated testing scripts to mitigate manual effort and improve reliability. Technical issues, especially on the React Native platform, such as build problems with React Native and limitations in Android Studio, posed challenges throughout the development and testing phases. It is advisable to initially undergo the process within marginal scope by building a release APK of a minimal application, in order to prevent and detect deployment issues early on. During this project, the build and deployment toolset Expo did not support generating local builds on the development machine, leading to the dependency on the Expo cloud service for application deployment, which was also limited in frequency. Moreover, deploying the iOS application via Expo required membership of the Apple Developer Program, which was not accessible. Consequently, only one iOS sample application with KMP was available, disqualifying meaningful comparison against any other iOS sample applications. Thus, this contributed to the decision to exclude the iOS platform from the evaluation. Android Studio's limitations included the absence of an export function for the values depicted in charts within the Android Profiler. Consequently, resorting to the manual screenshot capturing method was necessary. Addressing these challenges may necessitate dedicated time and resources in future research endeavors or should ideally be actively avoided if possible.

Despite the encountered limitations and challenges, this study opens avenues for future research in several directions, such as quantifying value added by AI-assisted coding, evaluation focused on user perception studies or development performance. While this thesis focused on the qualitative reflective analysis of AI integration in coding, future studies could benefit from adopting a more systematic and potentially quantifiable approach to better assess the impact of AI assistance. Leveraging documented AI interactions to analyze success rates and user interactions more precisely could enhance the evaluation of AI-assisted coding tools. It could also be beneficial to compare development productivity when undertaking a specific task, both with and without AI support. Developing methodologies to objectively assess the effectiveness of these tools beyond subjective reflections could contribute to future research efforts. Regarding the comparison of CPMD technologies, future research could focus on understanding the impact of performance differences on user experience. Investigating discrepancies between measured performance metrics and actual user perception, particularly in visually oriented and interaction-based applications, could provide valuable insights, as highlighted by Biørn-Hansen, Grønli and Ghinea (2019, p. 18). There is also the potential for exploring the development performance aspects of the different CPMD technologies, such as ease of development, build time, coding efficiency, and considering the development lifecycle.

Conclusion

The central theme of this study was a comprehensive exploration of AI-assisted coding and a comparative performance analysis of cross-platform mobile applications developed using KMP and React Native, as well as Android Native serving as baseline. Quantitative metrics provided a lens through which to assess the performance of sample applications, while qualitative reflections on AI-assisted coding enriched the understanding of the development experience. This dual approach illuminates the multifaceted nature of CPMD, emphasizing the importance of considering both numerical outcomes and nuanced perspectives. The findings encapsulate valuable insights into the effectiveness of AI in coding tasks and the performance disparities among various CPMD technologies. When reflecting on the findings and indications of this study, it becomes evident that the landscape of both domains encompasses challenges as well as opportunities.

Throughout the exploration of AI-assisted coding, a complex interplay of strengths and weaknesses emerged, underscoring the nuanced nature of this transformative and fast-paced technology. While AI exhibited decent potential in streamlining the development process, it also revealed several weaknesses, presenting opportunities for improvement. Issues such as code comprehension and the risk of passive development emphasize the need for a balanced approach to AI integration. On the other hand, it demonstrated enormous potential for skill development, particularly through the numerous interactions that promote tailored code generation and understanding of relationships. These findings highlight the high stakes and potential associated with AI incorporation, calling for careful consideration and continual refinement of AI-assisted coding tools.

In the area of CPMD performance comparison, the landscape is characterized by diversity, with no single technology emerging in the literature as the definitive leader across all aspects, only in certain scenarios and when considering corresponding contextual priorities. In this study, there was a tendency for KMP to outperform React Native across most metrics. As a side effect, it was observed that different device models generally exhibited variations in certain metrics independently from the CPMD technology, particularly startup time, CPU usage, and battery consumption.

The journey through AI-assisted coding and comparative performance analysis of CPMD technologies has not only uncovered valuable insights but also addressed the research objectives outlined at the onset of this study. Through the exploration of AI-assisted coding, this study has shed light on the potential and challenges of AI-generated code in aiding the development of mobile applications using different CPMD solutions. While it can be asserted that AI was effectively employed to streamline the coding process of the CPMD sample applications, thereby enhancing productivity and skill acquisition, the precise degree of improvement cannot be definitively quantified. The identification of quantifiable metrics for comparing mobile applications built with different CPMD solutions has been a focal point of this research. By taking resource conservation into account, this study has delineated key metrics for evaluating the performance of CPMD solutions implemented in sample applications, providing valuable benchmarks for future research endeavors. The performance comparison of different CPMD solutions, encompassing KMP, React Native, and Android Native, has been meticulously conducted in this study. By quantitatively assessing performance differences and inspecting various metrics, this research has contributed crucial insights into the efficacy of KMP, largely superior to React Native.

Navigating through the challenges encountered in this study, it is imperative to recognize them not as barriers but as learning opportunities leading to further investigation and refinement. Future research efforts should strive to expand the scope of comparative technologies, explore alternative API providers, and employ automated testing methods to enhance accuracy and efficiency. Moreover, efforts to mitigate technical hurdles, such as deployment limitations, are essential for ensuring the robustness and reliability of future research outcomes. Looking to the future, there are several promising directions for future research in both AI-assisted coding and CPMD. A systematic and potentially quantifiable approach to assessing the impact of AI assistance in coding tasks could provide deeper insights into its effectiveness and value proposition. Additionally, comparative studies focusing on development productivity with and without AI support could offer an avenue for understanding the true benefits of AI integration in software development workflows. In the realm of CPMD, future research could delve into user perception studies to understand the impact of performance differences on user experience.

In conclusion, this study serves as a steppingstone towards a deeper understanding of AI-assisted coding and CPMD evaluation. Especially for the relatively underexplored KMP technology up to this point, the scientific body is expanded with this study. The study also provides practitioners with insights into both KMP and React Native technologies. In these rapidly changing and continuously developing research areas, it is encouraged to remain committed to pushing the boundaries of knowledge and leveraging technology to shape a more efficient and impactful future for CPMD and AI-assisted coding.

Bibliography

Android. (2022). *Android Runtime (ART) and Dalvik*. Android Developers. Retrieved August 20, 2023 from https://source.android.com/docs/core/runtime

Annuzzi, J., Darcey, L., & Conder, S. (2016). *Introduction to Android application development: Android essentials* (5 ed.). Addison-Wesley.

Apple Inc. (2012). *iOS Technology Overview (iOS 6.0)*. Apple Developer. Retrieved August 25, 2023 from https://docplayer.net/docview/28/12576512/

Apple Inc. (2013). *What Is Cocoa?* Cocoa Fundamentals Guide. Apple Developer. Retrieved September 20, 2023 from https://developer.apple.com/library/archive/documentation/Cocoa/Conceptual/CocoaFundamentals/WhatIsCocoa/WhatIsCocoa.html

Apple Inc. (2023a). *Apple Developer Program*. Retrieved September 11, 2023 from https://developer.apple.com/programs/

Apple Inc. (2023b). *iOS & iPadOS Release Notes*. Apple Developer. Retrieved August 25, 2023 from https://developer.apple.com/documentation/ios-ipados-release-notes

Apple Inc. (2023c). *Manage your app's availability. Overview of publishing your app*. Apple Developer. Retrieved September 11, 2023 from https://developer.apple.com/help/app-store-connect/manage-your-apps-availability/overview-of-publishing-your-app/

Apple Inc. (2023d). *Open Source at Apple*. Apple Open Source. Retrieved September 3, 2023 from https://opensource.apple.com/

Apple OSS Distributions. (2023, June 24). *apple-oss-distributions/xnu*. Github. Retrieved Sep 10, 2023 from https://github.com/apple-oss-distributions/xnu

Becker, B. A., Denny, P., Finnie-Ansley, J., Luxton-Reilly, A., Prather, J., & Santos, E. A. (2023). Programming Is Hard - Or at Least It Used to Be: Educational Opportunities and Challenges of AI Code Generation. (pp. 500-506). *Proceedings of the 54th ACM Technical Symposium on Computer Science Education V. 1*, Toronto ON, Canada. https://doi.org/10.1145/3545945.3569759

Biørn-Hansen, A., Grønli, T.-M., & Ghinea, G. (2018). A Survey and Taxonomy of Core Concepts and Research Challenges in Cross-Platform Mobile Development. *ACM Computing Surveys (CSUR), 51*(5), 1-34. https://doi.org/10.1145/3241739

Biørn-Hansen, A., Grønli, T.-M., & Ghinea, G. (2019). Animations in cross-platform mobile applications: An evaluation of tools, metrics and performance. *Sensors, 19*(9), 2081. https://doi.org/10.3390/s19092081

Biørn-Hansen, A., Grønli, T.-M., Ghinea, G., & Alouneh, S. (2019). An empirical study of cross-platform mobile development in industry. *Wireless Communications and Mobile Computing, 2019*. https://doi.org/10.1155/2019/5743892

Biørn-Hansen, A., Grønli, T.-M., Majchrzak, T. A., Kaindl, H., & Gheorghita, G. (2022). The Use of Cross-Platform Frameworks for Google Play Store Apps. *Proceedings of the 55th Hawaii International Conference on System Sciences (HICSS),* Honolulu, USA. https://doi.org/10.24251/HICSS.2022.934

Biørn-Hansen, A., Majchrzak, T. A., & Grønli, T.-M. (2018). *Progressive Web Apps for the Unified Development of Mobile Applications*. In T. A. Majchrzak, P. Traverso, K.-H. Krempels, & V. Monfort (Eds.), Web Information Systems and Technologies. WEBIST 2017. Lecture Notes in Business Information Processing (Vol. 322, pp. 64-86). Springer. https://doi.org/10.1007/978-3-319-93527-0_4

Biørn-Hansen, A., Rieger, C., Grønli, T.-M., Majchrzak, T. A., & Ghinea, G. (2020). An empirical investigation of performance overhead in cross-platform mobile development frameworks. *Empirical Software Engineering, 25,* 2997-3040. https://doi.org/10.1007/s10664-020-09827-6

Brocke, J. v., Simons, A., Niehaves, B., Niehaves, B., Reimer, K., Plattfaut, R., & Cleven, A. (2009). Reconstructing the Giant: On the Importance of Rigour in Documenting the Literature Search Process. *ECIS 2009 Proceedings, 161*. https://aisel.aisnet.org/ecis2009/372

Calegario, F., Burégio, V., Erivaldo, F., Andrade, D. M. C., Felix, K., Barbosa, N., da Silva Lucena, P. L., & França, C. (2023). *Exploring the intersection of Generative AI and Software Development*. arXiv [cs.SE]. http://arxiv.org/abs/2312.14262

Chebbi, A. (2019, July 1). Choosing the best programming language for mobile app development. *IBM Developer*. Retrieved August 8, 2023 from https://developer.ibm.com/articles/choosing-the-best-programming-language-for-mobile-app-development/

Dalmasso, I., Datta, S. K., Bonnet, C., & Nikaein, N. (2013). Survey, comparison and evaluation of cross platform mobile application development tools. (pp. 323-328). *Proceedings of the 2013 9th International Wireless Communications and Mobile Computing Conference (IWCMC),* Sardinia, Italy. https://doi.org/10.1109/IWCMC.2013.6583580

Danial, A. (2023). *cloc (version 1.98) [Console application]*. Retrieved February 3, 2024 from https://doi.org/10.5281/zenodo.8266258

EASA Inc. (2021). What is appification? Retrieved February 12, 2024 from https://www.easasoftware.com/democratization/what-is-appification/

Ebone, A., Tan, Y., & Jia, X. (2018). *A Performance Evaluation of Cross-Platform Mobile Application Development Approaches*. In 2018 IEEE/ACM 5th International Conference on Mobile Software Engineering and Systems (MOBILESoft) (pp. 92-93). https://doi.org/10.1145/3197231.3197252

Eisenhardt, K. M. (1989). Building Theories from Case Study Research. *The Academy of Management Review, 14*(4), 532-550. https://doi.org/10.2307/258557

El-Kassas, W. S., Abdullah, B. A., Yousef, A. H., & Wahba, A. M. (2017). Taxonomy of cross-platform mobile applications development approaches. *Ain Shams Engineering Journal, 8*(2), 163-190. https://doi.org/10.1016/j.asej.2015.08.004

El Tom, A., Bogdan, C., Majchrzak, T. A., & Grønli, T.-M. (2023). Criteria Based Evaluation of Cross-Platform Development Frameworks. *Proceedings of the 2023 56th Hawaii International Conference on System Sciences (HICSS)*, Lahaina, HI, USA. https://hdl.handle.net/10125/103475

Evert, A.-K. (2019). *Cross-Platform Smartphone Application Development with Kotlin Multiplatform: Possible Impacts on Development Productivity, Application Size and Startup Time* [Master's thesis, KTH Royal Institute of Technology]. https://www.diva-portal.org/smash/get/diva2:1368323/FULLTEXT01.pdf

Expo. (n.d.). *Core concepts. An overview of Expo tools, features and services*. Expo Docs. Retrieved February 12, 2024 from https://docs.expo.dev/core-concepts/

Faranda, E. (2024). *PCAPdroid (version 1.6.9) [Mobile Android application].* GitHub repository. Retrieved January 30, 2024 from https://github.com/emanuele-f/PCAPdroid

Farooq, M. S., Riaz, S., Alvi, A., Ali, A., & Rehman, I. U. (2022). Cross-Platform Mobile Development Approaches and Frameworks. *VFAST Transactions on Software Engineering, 10*(2), 79-93. https://doi.org/doi.org/10.21015/vtse.v10i2.978

Fentaw, A. E. (2020). *Cross platform mobile application development: a comparison study of React Native Vs Flutter* [Master's thesis, University of Jyväskylä]. https://jyx.jyu.fi/handle/123456789/70969

Foremski, T. (2010, January 28). *Apple becomes more closed with each new device*. Retrieved September 3, 2023 from https://www.zdnet.com/article/apple-becomes-more-closed-with-each-new-device/

Gartner Inc. (2017, February 15). *Gartner Says Worldwide Sales of Smartphones Grew 7 Percent in the Fourth Quarter of 2016* [Press release]. https://www.gartner.com/en/newsroom/press-releases/2017-02-15-gartner-says-worldwide-sales-of-smartphones-grew-7-percent-in-the-fourth-quarter-of-2016

Gartner Inc. (n.d.). Proprietary Software. In *Gartner Glossary*. Retrieved September 3, 2023 from https://www.gartner.com/en/information-technology/glossary/proprietary-software

Goetz, J., & Li, Y. (2018). Evaluation of Cross-Platform Frameworks for Mobile Applications. *International Journal of Engineering and Innovative Technology (IJEIT), 8*(3), 10-17. https://doi.org/10.17605/OSF.IO/27SWU

Google. (2023a). *Android's Kotlin-first approach*. Android Developers. Retrieved August 24, 2023 from https://developer.android.com/kotlin/first

Google. (2023b). *Architecture overview*. Android Developers. Retrieved August 19, 2023 from https://source.android.com/docs/core/architecture

Google. (2023c). *Publish your app*. Android Developers. Retrieved August 24, 2023 from https://developer.android.com/studio/publish

Google. (2024a). *App startup time*. Android Developers. Retrieved January 30, 2024 from https://developer.android.com/topic/performance/vitals/launch-time

Google. (2024b). *Configure your build*. Android Developers. Retrieved February 10, 2024 from https://developer.android.com/build

Google. (2024c). *Overview of system tracing*. Android Developers. Retrieved January 30, 2024 from https://developer.android.com/topic/performance/tracing

Google. (n.d.-a). *Android Open Source Project*. Android Developers. Retrieved August 3, 2023 from https://source.android.com/

Google. (n.d.-b). *Android Studio*. Android Developers. Retrieved September 16, 2023 from https://developer.android.com/studio

Heitkötter, H., Hanschke, S., & Majchrzak, T. A. (2013). *Comparing cross-platform development approaches for mobile applications*. In J. Cordeiro & K. Krempels (Eds.), 2012 8th International Conference on Web Information Systems and Technologies (WEBIST), Porto, Portugal, Revised Selected Papers, Springer (pp. 120-138). Springer. https://doi.org/10.1007/978-3-642-36608-6_8

Heitkötter, H., Kuchen, H., & Majchrzak, T. A. (2015). Extending a model-driven cross-platform development approach for business apps. *Science of Computer Programming, 97*, 31-36. https://doi.org/10.1016/j.scico.2013.11.013

Hemmerich, W. (2024). *Einfaktorielle ANOVA: Einführung*. StatistikGuru, Version 1.96. Retrieved February 6, 2024 from https://statistikguru.de/spss/einfaktorielle-anova/einfuehrung.html

Hochmair, H. H., Juhasz, L., & Kemp, T. (2024). *Correctness Comparison of ChatGPT-4, Bard, Claude-2, and Copilot for Spatial Tasks*. arXiv [cs.CY]. http://arxiv.org/abs/2401.02404

Huber, S., Demetz, L., & Felderer, M. (2020). *Analysing the performance of mobile cross-platform development approaches using UI interaction scenarios*. In M. van Sinderen & L. Maciaszek (Eds.), Software Technologies. ICSOFT 2019. Communications in Computer and Information Science (Vol. 1250, pp. 40-57). Springer. https://doi.org/10.1007/978-3-030-52991-8_3

Huynh, M. Q., & Ghimire, P. (2017). Browser app approach: Can it be an answer to the challenges in cross-platform app development? *Journal of Information Technology Education. Innovations in Practice, 16*, 47. https://doi.org/10.28945/3667

Işitan, M., & Koklu, M. (2020). Comparison and Evaluation of Cross Platform Mobile Application Development Tools. *International Journal of Applied Mathematics Electronics and Computers, 8*(4), 273-281. https://doi.org/10.18100/ijamec.832673

JetBrains. (2023a). *FAQ*. Kotlin Multiplatform overview. Retrieved February 12, 2024 from https://www.jetbrains.com/help/kotlin-multiplatform-dev/faq.html

JetBrains. (2023b). *Kotlin Multiplatform*. Retrieved February 12, 2024 from https://kotlinlang.org/docs/multiplatform.html

JetBrains. (n.d.). *Compose Multiplatform*. Retrieved February 12, 2024 from https://www.jetbrains.com/lp/compose-multiplatform/

Kormanik, K. A. (2016). *Statistics Fundamentals Succinctly*. Syncfusion.

Kroes, A. D. A., & Finley, J. R. (2023, Jul 20). Demystifying omega squared: Practical guidance for effect size in common analysis of variance designs. *Psychol Methods, Advance online publication*. https://doi.org/10.1037/met0000581

Lachgar, M., Hanine, M., Benouda, H., & Ommane, Y. (2022). Decision Framework for Cross-Platform Mobile Development Frameworks Using an Integrated Multi-Criteria Decision-Making Methodology. *International Journal of Mobile Computing and Multimedia Communications (IJMCMC), 13*(1), 1-22. https://doi.org/10.4018/IJMCMC.297928

Li, R., Allal, L. B., Zi, Y., Muennighoff, N., Kocetkov, D., Mou, C., Marone, M., Akiki, C., Li, J., & Chim, J. (2023). *StarCoder: may the source be with you!* arXiv [cs.CL]. https://doi.org/10.48550/arXiv.2305.06161

Mahendra, M., & Anggorojati, B. (2021). Evaluating the performance of Android based Cross-Platform App Development Frameworks. (pp. 32–37). *Proceedings of the 6th International Conference on Communication and Information Processing,* Tokyo, Japan. https://doi.org/10.1145/3442555.3442561

Majchrzak, T. A., Biørn-Hansen, A., & Grønli, T.-M. (2017). Comprehensive Analysis of Innovative Cross-Platform App Development Frameworks. *Proceedings of the 50th Hawaii International Conference on System Sciences*, Waikoloa Village, HI, USA. http://hdl.handle.net/10125/41909

Majchrzak, T. A., Ernsting, J., & Kuchen, H. (2015, 01/01). Achieving Business Practicability of Model-Driven Cross-Platform Apps. *Open Journal of Information Systems (OJIS), 2*, 3-14. https://www.researchgate.net/publication/280310211

Meta Platforms Inc. (n.d.). *Setting up the development environment*. Retrieved February 12, 2024 from https://reactnative.dev/docs/environment-setup?guide=quickstart

Minitab, L. (2023). *Data considerations for One-Way ANOVA*. Minitab 21 Support. Retrieved January 29, 2024 from https://support.minitab.com/en-us/minitab/21/help-and-how-to/statistical-modeling/anova/how-to/one-way-anova/before-you-start/data-considerations/

Mota, D., & Martinho, R. (2021). An Approach to Assess the Performance of Mobile Applications: A Case Study of Multiplatform Development Frameworks. (Vol. 1, pp. 150-157). *Proceedings of the 16th International Conference on Evaluation of Novel Approaches to Software Engineering (ENASE)*, Virtual Event. https://doi.org/10.5220/0010497401500157

Murillo, A., & D'Angelo, S. (2023). An Engineering Perspective on Writing Assistants for Productivity and Creative Code. *Google*. https://research.google/pubs/an-engineering-perspective-on-writing-assistants-for-productivity-and-creative-code/

Nawrocki, P., Wrona, K., Marczak, M., & Sniezynski, B. (2021). A comparison of native and cross-platform frameworks for mobile applications. *Computer, 54*(3), 18-27. https://doi.org/10.1109/MC.2020.2983893

Nunkesser, R. (2018). Beyond web/native/hybrid: a new taxonomy for mobile app development. (pp. 214-218). *Proceedings of the 5th International Conference on Mobile Software Engineering and Systems*, Gothenburg, Sweden. https://doi.org/10.1145/3197231.3197260

Opensignal. (2015). *Android Fragmentation 2015*. https://cdn.opensignal.com/public/data/reports/global/data-2015-08/2015_08_fragmentation_report.pdf

Peterson, M. (2021, Dec 8). Apple debuts new Open Source website, will release projects on GitHub. *AppleInsider*. Retrieved September 10, 2023 from https://appleinsider.com/articles/21/12/08/apple-debuts-new-open-source-website-will-release-projects-on-github

Petrova, E. (2023). Update on the Name of Kotlin Multiplatform. *The Kotlin Blog*. JetBrains. Retrieved February 12, 2024 from https://blog.jetbrains.com/kotlin/2023/07/update-on-the-name-of-kotlin-multiplatform/

Pinto, C. M., & Coutinho, C. (2018). From native to cross-platform hybrid development. (pp. 669-676). *2018 International Conference on Intelligent Systems (IS)*, https://doi.org/10.1109/IS.2018.8710545

Raj, C. R., & Tolety, S. B. (2012). A study on approaches to build cross-platform mobile applications and criteria to select appropriate approach. (pp. 625-629). *2012 Annual IEEE India Conference (INDICON)*, https://doi.org/10.1109/indcon.2012.6420693

Rieger, C., & Majchrzak, T. A. (2019). Towards the definitive evaluation framework for cross-platform app development approaches. *Journal of Systems and Software, 153*(C), 175–199. https://doi.org/10.1016/j.jss.2019.04.001

Sakib, F. A., Khan, S. H., & Karim, A. H. M. R. (2023). *Extending the Frontier of ChatGPT: Code Generation and Debugging*. arXiv [cs.SE]. http://arxiv.org/abs/2307.08260

Sarah, C., Kathrin, C., Ann, R., Guro, H., Anthony, A., & Aziz, S. (2011). The case study approach. *BMC Med Res Methodol, 11*(100). https://doi.org/10.1186/1471-2288-11-100

Sergeyuk, A., Titov, S., & Izadi, M. (2024). *In-IDE Human-AI Experience in the Era of Large Language Models; A Literature Review*. arXiv [cs.SE]. http://arxiv.org/abs/2401.10739

Shah, K., Sinha, H., & Mishra, P. (2019). Analysis of cross-platform mobile app development tools. (pp. 1-7). *2019 IEEE 5th International Conference for Convergence in Technology (I2CT)*, Bombay, India. https://doi.org/10.1109/I2CT45611.2019.9033872

Silberschatz, A., Gagne, G., & Galvin, P. B. (2018). *Operating System Concepts* (10 ed.). Wiley.

Stack Overflow. (2022). *2022 Developer Survey*. https://survey.stackoverflow.co/2022/

Stanojević, J., Šošević, U., Minović, M., & Milovanović, M. (2022). An Overview of Modern Cross-platform Mobile Development Frameworks. (pp. 489-497). *Proceedings of the Central European Conference on Information and Intelligent Systems*, Dubrovnik, Croatia. https://www.proquest.com/openview/1e0058c90b20fc50ef286e5fc98da4b5/

Statcounter Global Stats. (2022a). *Mobile Vendor Market Share United States Of America Jan - Dec 2022*. Retrieved August 27, 2023 from https://gs.statcounter.com/vendor-market-share/mobile/united-states-of-america/2022

Statcounter Global Stats. (2022b). *Mobile Vendor Market Share Worldwide Jan - Dec 2022*. Retrieved August 27, 2023 from https://gs.statcounter.com/vendor-market-share/mobile/worldwide/2022

Vaithilingam, P., Zhang, T., & Glassman, E. L. (2022). Expectation vs. Experience: Evaluating the Usability of Code Generation Tools Powered by Large Language Models. (pp. 1-7). *CHI '22 Extended Abstracts. Conference on Human Factors in Computing Systems*, New Orleans, LA, USA. https://doi.org/10.1145/3491101.3519665

Vergel, M. (2014). *The Free Currency Converter API*. Retrieved February 7, 2024 from https://free.currencyconverterapi.com/

Watson, R. T., & Webster, J. (2020). Analysing the past to prepare for the future: Writing a literature review a roadmap for release 2.0. *Journal of Decision Systems, 29*(3), 129-147. https://doi.org/https://doi.org/10.1080/12460125.2020.1798591

Webster, J., & Watson, R. T. (2002). Analyzing the Past to Prepare for the Future: Writing a Literature Review. *MIS Quarterly*, *26*(2), xiii-xxiii. http://www.jstor.org/stable/4132319

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) (pp. 454-461). https://doi.org/10.1109/MobServ.2015.68

Wu, W. (2018). *React Native vs Flutter, Cross-platform mobile application frameworks* [Bachelor's thesis, Metropolia University of Applied Sciences]. https://www.theseus.fi/bitstream/handle/10024/146232/thesis.pdf

Xanthopoulos, S., & Xinogalos, S. (2013). A comparative analysis of cross-platform development approaches for mobile applications. (pp. 213-220). *Proceedings of the 6th Balkan Conference in Informatics*, Thessaloniki, Greece. https://doi.org/10.1145/2490257.2490292

Yetiştiren, B., Özsoy, I., Ayerdem, M., & Tüzün, E. (2023). *Evaluating the Code Quality of AI-Assisted Code Generation Tools: An Empirical Study on GitHub Copilot, Amazon CodeWhisperer, and ChatGPT*. arXiv [cs.SE]. http://arxiv.org/abs/2304.10778

You, D., & Hu, M. (2021). A Comparative Study of Cross-platform Mobile Application Development. (Vol. 66). *Proceedings of the 12th Annual CITRENZ Conference*, Wellington, New Zealand. https://www.researchgate.net/publication/357898491

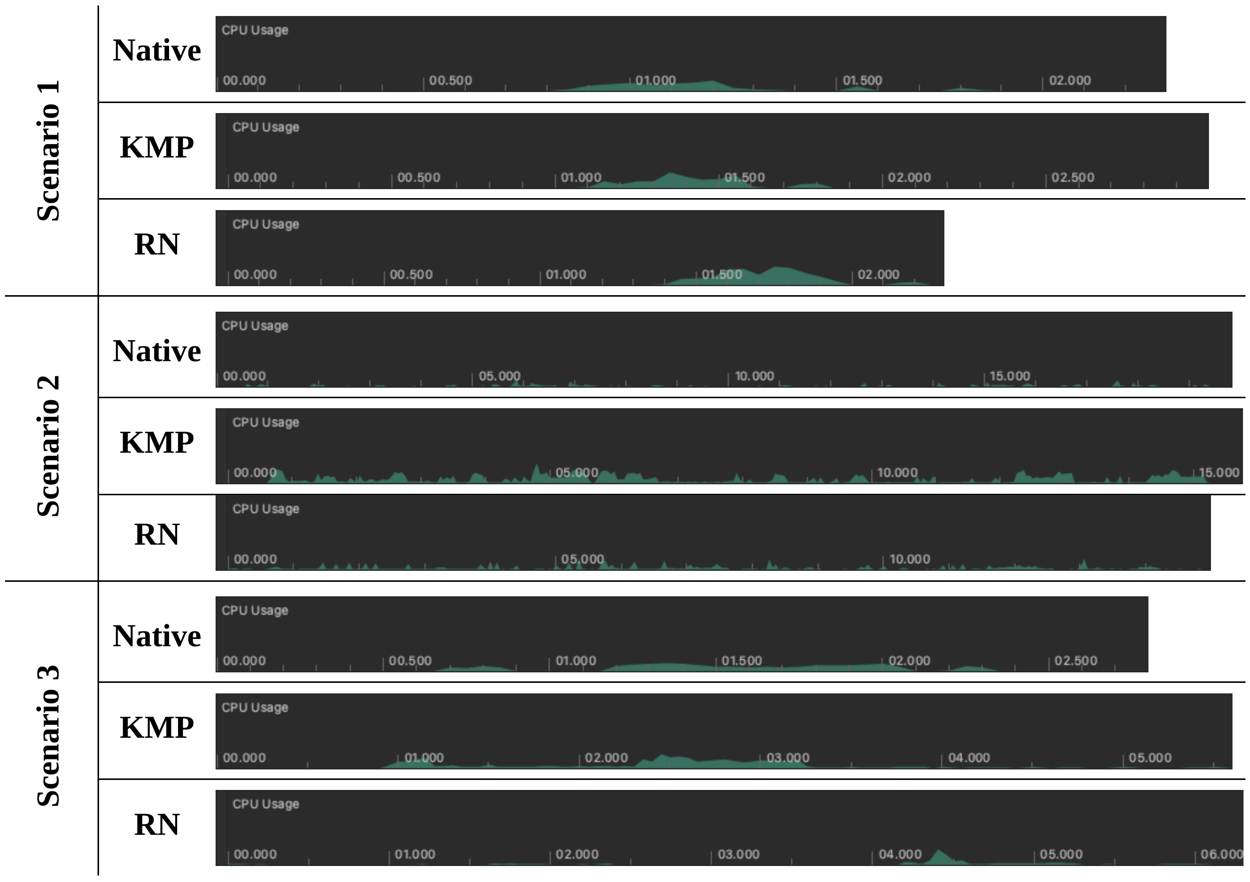
Zohud, T., & Zein, S. (2021). Cross-platform mobile app development in industry: A multiple case-study. *International Journal of Computing, 20*(1), 46-54. https://doi.org/10.47839/ijc.20.1.2091

Appendix

Note: Appendix A contains directly attached materials in this document. Appendix B designates supplementary digital data, with the corresponding locations specified via directory paths within the digital attachment. Each entry in Appendix B consists of one or more files, which are organized into subfolders when necessary.

Appendix A

Appendix A1: CPU charts for each scenario and technology of iteration 1 on Moto G54



Appendix A2: Summary of all metrics measurements

Startup time (ms)

Device	n	Native	KMP	RN
Motorola Moto G10	1	673.906	858.441	728.85
	2	670.301	822.779	886.996
	3	595.883	841.74	689.029
	4	616.429	833.002	712.106
	5	621.829	842.531	677.794
	6	603.203	822.067	692.76
	7	623.925	825.672	683.66
	8	612.896	835.445	675.564
	9	616.563	846.03	675.536
	10	619.333	840.585	704.059
	11	604.333	846.596	680.73
	12	600.836	839.851	700.449
	13	607.587	855.406	704.959
	14	608.996	840.823	689.111
	15	598.764	844.291	700.434
	x̄	618.319	839.684	706.803
	σ	23.438	10.616	52.039
Google Pixel 4a	1	516.41	453.397	383.504
	2	334.259	452.427	409.563
	3	330.978	490.352	391.684
	4	321.11	661.659	388.398
	5	303.746	441.806	438.367
	6	344.571	468.847	399.239
	7	344.444	498.791	402.293
	8	323.447	443.211	385.45
	9	326.005	457.704	426.355
	10	319.713	471.336	424.838
	11	341.826	501.336	415.884
	12	315.783	451.409	427.732
	13	380.944	469.922	447.627
	14	315.884	473.187	412.358
	15	347.748	473.785	416.927
	x̄	344.458	480.611	411.348
	σ	51.017	53.379	19.577
Motorola Moto G54	1	406.472	483.35	423.569
	2	402.342	471.082	437.066
	3	359.625	442.522	410.475
	4	347.791	445.881	407.246
	5	381.955	438.726	419.481
	6	370.857	453.174	466.832
	7	349.542	493.922	419.566
	8	409.828	466.042	387.329
	9	388.725	457.125	430.093
	10	373.288	451.866	400.166
	11	372.105	454.378	411.961
	12	360.427	468.444	412.498
	13	385.99	429.994	440.49
	14	374.116	451.586	414.723
	15	361.458	468.308	421.501
	x̄	376.302	458.427	420.2
	σ	19.517	16.987	18.678

Network traffic (MB)

Device	n	Scenario 1			Scenario 2		
		Native	KMP	RN	Native	KMP	RN
Motorola Moto G10	1	8.6	8.6	8.8	8	0.7	0.8
	2	8.3	8.9	8.7	8.3	0.6	0.7
	3	8.4	8.9	8.9	8.1	0.7	0.6
	4	8.6	8.8	8.8	8.1	0.7	0.7
	5	8.3	8.7	8.9	8.2	0.7	0.6
	6	8.4	8.6	8.8	8.2	0.7	0.7
	7	8.5	8.8	8.8	8.1	0.6	0.7
	8	8.6	8.7	8.7	7.9	0.6	0.8
	9	8.4	8.8	8.8	8.2	0.7	0.6
	10	8.4	9	8.7	8.2	0.7	0.8
	11	8.3	8.7	8.8	8.2	0.7	0.6
	12	8.3	8.8	8.9	8.1	0.6	0.7
	13	8.4	8.7	8.8	8.4	0.8	0.7
	14	8.4	8.8	8.7	8.1	0.7	0.7
	15	8.4	8.7	8.8	8.2	0.6	0.6
	x̄	8.42	8.77	8.79	8.15	0.67	0.69
	σ	0.11	0.11	0.07	0.12	0.06	0.07
Google Pixel 4a	1	8.2	8.3	8.2	8.1	0.6	0.7
	2	8	8.3	8.2	8	0.7	0.6
	3	8.2	8.1	8.2	8.2	0.7	0.7
	4	8.2	8.6	8.3	8.2	0.6	0.8
	5	8.1	8.1	8.5	7.8	0.8	0.6
	6	8.2	8.4	8.1	8.3	0.6	0.8
	7	8.3	8.2	8.3	8	0.8	0.8
	8	8.2	8.1	8.1	8.1	0.8	0.7
	9	8.5	8.1	8.2	7.9	0.8	0.7
	10	8.2	8.1	8.3	8.2	0.7	0.8
	11	8.2	8.3	8.2	8	0.7	0.8
	12	8.2	8.3	8.1	8	0.8	0.6
	13	8.1	8.3	8.1	8	0.8	0.6
	14	8.2	8.1	8.1	8.1	0.7	0.6
	15	8.2	8.2	8.2	8.1	0.7	0.8
	x̄	8.2	8.23	8.21	8.07	0.72	0.71
	σ	0.11	0.14	0.11	0.13	0.08	0.09
Motorola Moto G54	1	8.2	8.3	8.1	8	0.6	0.7
	2	8.3	8.3	8.1	8	0.8	0.6
	3	8.2	8.3	8.2	8.2	0.7	0.8
	4	8.3	8.2	8.1	8	0.6	0.7
	5	8.2	8.3	8.2	8.2	0.7	0.6
	6	8.2	8.3	8.2	7.9	0.7	0.6
	7	8.2	8.1	8.2	8.2	0.6	0.6
	8	8.2	8.2	8.3	8.2	0.7	0.7
	9	8.2	8.3	8.2	8	0.8	0.8
	10	8.2	8.2	8.4	8	0.7	0.6
	11	8.2	8.3	8.2	8.1	0.7	0.7
	12	8.1	8.2	8.3	8	0.8	0.6
	13	8.2	8.3	8.3	8.3	0.7	0.7
	14	8.2	8.3	8.3	8	0.8	0.7
	15	8.2	8.3	8.3	8.1	0.6	0.8
	x̄	8.21	8.26	8.23	8.08	0.7	0.68
	σ	0.05	0.06	0.09	0.11	0.08	0.08

CPU usage (%)

Device	n	Scenario 1			Scenario 2			Scenario 3		
		Native	KMP	RN	Native	KMP	RN	Native	KMP	RN
Motorola Moto G10	1	16.7	23.9	28.9	16.7	32.5	16.4	7	18.8	24.9
	2	22.3	24	30	21	28.4	17.9	6.9	14.9	24.9
	3	14.9	23	30.7	19.3	31	18.1	8	18.7	24.9
	4	15.9	24	31	19.4	32.9	17.4	6	18.9	25
	5	21.9	25	30.9	18.9	29.5	21.5	11	19.8	25
	6	20.7	22	30.9	18	31.5	18.7	11.8	22.7	25
	7	20.4	21.8	30.9	21.3	29.8	21.8	7	15	25
	8	16	23.9	30	19.5	32.7	19.7	11	16.9	25
	9	17	23.9	30.9	17.5	33.4	19.5	12	20.2	25
	10	17.9	22.7	34	16.5	33.7	20.6	10.9	20.8	25
	11	20.8	23.9	32.7	17.6	31.7	19.5	12.7	19.6	25
	12	21.6	22.8	33.8	15.9	31.4	15.8	6.4	17	25
	13	21	24	33.9	17	35.3	21	6	17.8	24.9
	14	14.9	23	32.5	20.4	29.9	19.9	11.8	20.9	25
	15	23.4	24	32.6	17.6	33.4	22.1	7	19.6	25
	x̄	19.03	23.46	31.58	18.44	31.81	19.33	9.03	18.77	24.97
	σ	2.93	0.87	1.57	1.68	1.87	1.93	2.57	2.16	0.05
Google Pixel 4a	1	10	14.9	25.9	13.2	15.9	14.8	5	10.9	9.9
	2	12	17	23	12.5	25.8	15.3	6	11	17.9
	3	12	14.9	23	14.8	17.1	13.5	5.9	11.9	16.8
	4	11	15	26.9	10.9	18.6	14.8	5	11.9	17
	5	11	19.8	25	10.9	18	14.4	5.7	11.9	19.7
	6	11	15	24	12.7	19.6	14.6	6	11.9	21
	7	12	14	29	14.8	20.2	15.9	4	8.9	20.6
	8	11	19	24	13.4	18	15.2	4.9	12	18.7
	9	11	18	24.9	16.8	31.8	13.7	4.9	10	19.8
	10	11	14.9	24.9	11.8	20.7	16	6	11	18.9
	11	11	15	28.7	13.6	17	17.1	4	9	17
	12	12	17	26.9	16.4	18.8	22.2	5	10	19.8
	13	11	15	23.9	13.1	17.8	16.4	5.9	11	19.8
	14	11	16	25	17.1	18.7	14.8	4	11.9	18.8
	15	11	15.9	24.9	16	18.8	16.7	4	10.9	18.9
	x̄	11.2	16.09	25.33	13.87	19.79	15.69	5.09	10.95	18.31
	σ	0.56	1.71	1.84	2.04	4.02	2.08	0.8	1.05	2.66
Motorola Moto G54	1	14	20.9	24.9	12.9	26.3	13.6	5	18.7	21
	2	13.9	16.9	23.9	8.6	21.8	19.8	11	17	19.9
	3	15	17.9	26.9	13.5	28.5	18.9	7	12	9.7
	4	13	19.9	26.7	12.5	29	16.3	8.9	13	8.7
	5	13	18	23.9	26.6	28	16.8	3	12	19.8
	6	14.9	17	26	22.6	20.3	15.3	9.9	14.9	8.6
	7	13.9	17.9	28	17.3	17.4	14.7	4	9	18.9
	8	19	17.9	24.8	9.1	26.7	16.6	9	12	18.8
	9	13.9	17.8	26.8	17.7	23.8	11.8	3	12	7
	10	17.7	18	27.8	8	29.2	15.8	9	15	8.9
	11	15.8	16.9	25.6	16.4	26.3	12	5.8	12	17
	12	14.9	19.7	26.7	12.7	28.4	19.1	10	13.9	19.6
	13	17	19.8	25.8	10.8	28.9	17.2	10	9.9	18
	14	15	16.9	25.7	17.5	18.5	17.9	9	14	21.6
	15	13.9	17.8	26.9	15.3	27.7	16.5	4	12	17.8
	x̄	14.99	18.22	26.03	14.77	25.39	16.15	7.24	13.16	15.69
	σ	1.73	1.26	1.26	5.16	4.02	2.39	2.84	2.52	5.35

Memory usage (MB)

Device	n	Scenario 1			Scenario 2			Scenario 3		
		Native	KMP	RN	Native	KMP	RN	Native	KMP	RN
Motorola Moto G10	1	29	29	60.5	33.4	37	63.5	33.5	37	72.5
	2	31	29.5	61	34	45.5	63.5	36	44	72.5
	3	30.5	29.5	61	34	44.9	64	36.4	44.4	72.5
	4	31	29.5	61	34	45	63	34.9	44	72.5
	5	30	29.5	61	34	45.5	63	36.5	44	72
	6	29.5	29.5	61	34	46	63.5	35.5	45	72
	7	31.5	29.5	61	34	45	63.5	34.5	44.4	72
	8	31	30	61	33.9	44.9	63.5	35.5	44.5	73.5
	9	29.5	30	61	34	45.5	63.5	35.5	45.5	71.9
	10	31.5	30	61	34	46	63.5	35.5	44.5	72.5
	11	31.5	30	61	34	45.9	63.5	35.5	44	72
	12	29.9	30	61	34	46.5	64	34.5	44.4	73.5
	13	29.9	29.5	61.5	34	44.5	63.5	34.5	44.5	71.7
	14	29	30	61.5	34	45.5	63.5	36.5	45	72.5
	15	29.9	30	61.5	34	45.5	63.5	34.5	45	72
	x̄	30.31	29.7	61.07	33.95	44.88	63.5	35.29	44.01	72.37
	σ	0.89	0.32	0.26	0.16	2.24	0.27	0.88	1.99	0.54
Google Pixel 4a	1	26.1	27.1	52	35.5	52.6	73.7	36.1	44.6	74.6
	2	26.1	27.2	51.6	35.5	49	69	36.1	45.2	69
	3	26.2	21.1	51.6	36.2	50.2	69	36.5	44.2	69.2
	4	26.1	27	51.7	36.1	52.6	69	36.1	45	69.2
	5	26.1	27.1	51.7	36	51.7	71.2	36.1	45.4	69
	6	26.1	27.1	51.7	36	53.7	70.7	36.1	43.9	69.1
	7	26.2	27.1	51.6	35.5	53.2	72.7	36	44	69.2
	8	26.1	27	51.7	36	51.6	71	36.5	44.5	69.5
	9	26.1	27	51.5	36	53.4	69.6	36.1	44.4	69
	10	26.1	27.1	51.5	35.5	52	72.2	36.5	45.9	71.5
	11	26.1	27	51.6	36	51.7	70.2	36.1	45.5	69.1
	12	26.5	27.1	51.6	35.7	51.7	71.5	36.5	44.4	69
	13	26.2	27.2	51.6	36	52.1	72	36.1	44.5	69.1
	14	26.2	27	51.2	36	52.2	72.1	36	45.4	69.1
	15	26.1	27.1	52	36	52.5	70.6	36.5	45.5	69.5
	x̄	26.15	26.68	51.64	35.87	52.01	70.97	36.22	44.83	69.67
	σ	0.11	1.55	0.19	0.25	1.19	1.44	0.21	0.62	1.5
Motorola Moto G54	1	44.5	45.2	74.5	50.5	58.1	88.5	47.1	55.5	84.4
	2	44.9	45.6	72.4	50.4	59.7	85.6	52.6	53.9	82.7
	3	45.5	45.3	73.7	50.2	62.6	85.7	51.1	54.5	83
	4	45	45.6	73.6	51.5	58	86.2	53.6	53.9	82.5
	5	45.4	45.5	72.6	50.6	58.7	84.1	51	53.3	81.1
	6	45	45.7	68.6	51.5	58.1	82.7	51.2	53.1	79
	7	45.7	46.2	69.7	51	60.2	81.7	51.1	53.4	80.2
	8	45.2	45.4	68.6	50.6	59.1	83.1	51.2	53.4	78.7
	9	45	45.6	70.5	50	59.5	82.1	51.1	53	79
	10	45.2	45.7	68.1	50.5	60	81.7	51.2	53.7	78
	11	45.1	45.1	68.1	50.5	60	81.1	51.2	54.5	78.2
	12	45	45	68.4	50.5	60.1	81.5	51.1	53.5	77.1
	13	45.3	45.6	68.2	50.9	58.5	81.5	51.3	53.6	78.2
	14	44.5	45.2	68.9	50.2	58.1	81.1	51.1	54.9	77.6
	15	45.6	45.6	69.5	51.1	63	81.5	51.2	54.1	79.6
	x̄	45.13	45.49	70.36	50.67	59.58	83.21	51.14	53.89	79.95
	σ	0.35	0.3	2.33	0.45	1.54	2.29	1.32	0.7	2.26

Battery usage (MA)

Device	n	Scenario 1			Scenario 2			Scenario 3		
		Native	KMP	RN	Native	KMP	RN	Native	KMP	RN
Motorola Moto G10	1	-498.962	-498.962	-706.482	-549.621	-543.671	-628.662	-522.308	-480.804	-480.804
	2	-471.344	-471.344	-835.266	-389.862	-746.002	-743.408	-441.589	-411.377	-411.377
	3	-477.295	-477.295	-531.921	-588.074	-529.48	-503.845	-440.368	-514.679	-514.679
	4	-473.175	-473.175	-528.107	-484.314	-664.673	-421.295	-472.717	-413.971	-413.971
	5	-477.447	-477.447	-516.815	-382.232	-601.196	-613.556	-422.973	-519.256	-519.256
	6	-511.169	-511.169	-807.953	-619.354	-578.308	-421.142	-463.867	-568.085	-568.085
	7	-587.006	-587.006	-604.553	-511.474	-539.703	-571.136	-406.341	-456.085	-456.085
	8	-500.336	-500.336	-518.036	-497.284	-493.011	-573.12	-479.126	-348.205	-348.205
	9	-548.859	-548.859	-596.313	-511.932	-622.711	-490.265	-424.347	-359.802	-359.802
	10	-548.096	-548.096	-571.594	-501.251	-556.183	-470.428	-536.041	-436.859	-436.859
	11	-597.076	-597.076	-527.038	-459.595	-515.9	-511.474	-496.521	-393.982	-393.982
	12	-454.712	-454.712	-728.607	-524.292	-553.894	-470.581	-400.696	-391.083	-391.083
	13	-671.387	-671.387	-671.844	-570.831	-569.458	-625	-438.995	-490.417	-490.417
	14	-449.066	-449.066	-525.055	-465.698	-507.66	-579.376	-426.33	-437.317	-437.317
	15	-511.322	-511.322	-557.251	-488.739	-468.597	-438.385	-392.303	-368.652	-368.652
	x̄	-518.483	-518.483	-615.122	-502.97	-566.03	-537.445	-450.968	-439.372	-439.372
	σ	61.764	61.764	108.646	65.139	70.837	91.88	43.291	64.751	64.751
Google Pixel 4a	1	-989.99	-989.99	-536.194	-730.896	-846.863	-769.653	-612.793	-626.526	-626.526
	2	-694.885	-694.885	-834.046	-730.286	-852.356	-599.67	-599.365	-587.158	-587.158
	3	-819.702	-819.702	-1029.969	-737.61	-1056.824	-704.346	-684.814	-648.499	-648.499
	4	-870.361	-870.361	-1132.813	-633.85	-879.517	-1091.004	-668.945	-838.623	-838.623
	5	-606.384	-606.384	-859.985	-1003.723	-1184.998	-848.084	-648.499	-1087.952	-1087.952
	6	-704.651	-704.651	-1173.401	-789.795	-1136.17	-882.874	-416.26	-794.983	-794.983
	7	-876.16	-876.16	-817.566	-720.215	-993.958	-788.879	-781.25	-443.115	-443.115
	8	-781.86	-781.86	-974.121	-907.288	-946.045	-860.596	-580.749	-552.978	-552.978
	9	-567.932	-567.932	-972.901	-679.626	-924.072	-677.185	-611.877	-777.283	-777.283
	10	-877.991	-877.991	-978.394	-650.33	-1067.2	-1042.176	-632.629	-636.902	-636.902
	11	-752.896	-752.896	-1065.674	-740.662	-968.323	-922.547	-723.877	-836.792	-836.792
	12	-899.048	-899.048	-679.321	-648.193	-1145.935	-876.77	-701.599	-569.153	-569.153
	13	-718.689	-718.689	-1073.609	-809.326	-990.296	-628.357	-561.218	-699.768	-699.768
	14	-775.757	-775.757	-1201.477	-804.749	-928.345	-822.144	-637.512	-803.223	-803.223
	15	-972.595	-972.595	-1212.159	-931.091	-877.075	-805.664	-534.973	-763.855	-763.855
	x̄	-793.927	-793.927	-969.442	-767.843	-986.532	-821.33	-626.424	-711.121	-711.121
	σ	123.518	123.518	194.603	108.844	109.84	137.754	86.597	157.162	157.162
Motorola Moto G54	1	-486.3	-486.3	-605.9	-688.1	-645.2	-867	-607.8	-430.7	-430.7
	2	-674.2	-674.2	-833.7	-793.9	-667	-677.3	-494.2	-591.5	-591.5
	3	-470	-470	-847.1	-576.9	-685.7	-527.4	-511.1	-734	-734
	4	-575.7	-575.7	-486.9	-633.7	-707.5	-732.8	-714.7	-416.8	-416.8
	5	-661.6	-661.6	-787.8	-595.7	-823.5	-659.8	-597.5	-837.4	-837.4
	6	-722.6	-722.6	-673.1	-594.4	-763	-706.9	-595.7	-654.9	-654.9
	7	-464	-464	-607.8	-572.2	-579.4	-761.3	-523.8	-578.8	-578.8
	8	-421.1	-421.1	-686.3	-765.5	-564.3	-678.5	-665.8	-402.3	-402.3
	9	-451.3	-451.3	-810.8	-650.1	-852.5	-681.5	-616.2	-613.3	-613.3
	10	-582.4	-582.4	-682.7	-742.5	-793.2	-634.9	-528	-746.1	-746.1
	11	-618	-618	-702.7	-722	-836.2	-588.4	-503.3	-462.2	-462.2
	12	-625.3	-625.3	-699.6	-755.8	802.9	-552.8	-643.5	-670	-670
	13	-688.1	-688.1	-585.4	-548.5	-705	-824.1	-752.2	-761.9	-761.9
	14	-812	-812	-655.5	-595	-907.5	-704.4	-430.1	-600.5	-600.5
	15	-759.5	-759.5	-706.9	-659.1	-775.1	-584.8	-845.8	-727.4	-727.4
	x̄	-600.807	-600.807	-691.48	-659.56	-633.48	-678.793	-601.98	-615.187	-615.187
	σ	121.53	121.53	99.124	80.256	409.536	94.794	110.697	137.205	137.205

Appendix A3: Summary of ANOVA-p and ω^2 effect size calculations

Startup time			
Device	Comparison subjects	ANOVA-p	ω^2 effect size
Motorola Moto G10	Native / KMP	4.49E-24	0.97
	Native / React Native	1.81E-06	0.54
	KMP / React Native	1.92E-10	0.76
Google Pixel 4a	Native / KMP	9.02E-08	0.63
	Native / React Native	5.63E-05	0.42
	KMP / React Native	5.99E-05	0.41
Motorola Moto G54	Native / KMP	8.42E-13	0.83
	Native / React Native	8.34E-07	0.56
	KMP / React Native	2.64E-06	0.53

Network traffic					
Device	Comparison subjects	Scenario 1		Scenario 2	
		ANOVA-p	ω^2 effect size	ANOVA-p	ω^2 effect size
Motorola Moto G10	Native / KMP	2.14E-09	0.71	8.72E-47	1.00
	Native / React Native	7.45E-12	0.81	4.13E-46	1.00
	KMP / React Native	0.439	-	0.591	-
Google Pixel 4a	Native / KMP	0.479	-	4.89E-45	1.00
	Native / React Native	0.868	-	1.36E-44	1.00
	KMP / React Native	0.574	-	0.664	-
Motorola Moto G54	Native / KMP	0.013	0.17	3.29E-46	1.00
	Native / React Native	0.443	-	3.76E-46	1.00
	KMP / React Native	0.245	-	0.480	-

CPU usage							
Device	Comparison subjects	Scenario 1		Scenario 2		Scenario 3	
		ANOVA-p	ω^2 effect size	ANOVA-p	ω^2 effect size	ANOVA-p	ω^2 effect size
Motorola Moto G10	Native / KMP	5.17E-06	0.50	1.81E-18	0.93	6.77E-12	0.81
	Native / React Native	1.22E-14	0.88	0.190	-	3.02E-20	0.95
	KMP / React Native	1.26E-16	0.91	6.33E-17	0.91	8.97E-12	0.80
Google Pixel 4a	Native / KMP	3.04E-11	0.79	2.20E-05	0.45	2.12E-16	0.91
	Native / React Native	3.2E-22	0.96	0.022	0.14	3.32E-17	0.92
	KMP / React Native	2.32E-14	0.87	0.002	0.27	1.01E-10	0.77
Motorola Moto G54	Native / KMP	2.76E-06	0.53	8.55E-07	0.56	1.64E-06	0.54
	Native / React Native	4.15E-18	0.93	0.353	-	9.29E-06	0.48
	KMP / React Native	2.83E-16	0.91	2.55E-08	0.66	0.109	-

Memory usage							
Device	Comparison subjects	Scenario 1		Scenario 2		Scenario 3	
		ANOVA-p	ω^2 effect size	ANOVA-p	ω^2 effect size	ANOVA-p	ω^2 effect size
Motorola Moto G10	Native / KMP	0.018	0.15	1.96E-17	0.92	2.68E-15	0.89
	Native / React Native	2.27E-40	1.00	3.28E-53	1.00	2.18E-41	1.00
	KMP / React Native	1.49E-50	1.00	1.42E-23	0.97	1.08E-29	0.99
Google Pixel 4a	Native / KMP	0.199	-	3.18E-29	0.99	4.29E-29	0.99
	Native / React Native	1.38E-55	1.00	1.91E-36	1.00	2.01E-35	1.00
	KMP / React Native	1.54E-31	0.99	4.97E-26	0.98	5.63E-31	0.99
Motorola Moto G54	Native / KMP	0.005	0.21	5.62E-19	0.94	1.02E-07	0.62
	Native / React Native	1.14E-26	0.98	7.34E-30	0.99	5.36E-27	0.98
	KMP / React Native	1.56E-26	0.98	5.02E-24	0.97	5.18E-27	0.98

Battery usage							
Device	Comparison subjects	Scenario 1		Scenario 2		Scenario 3	
		ANOVA-p	ω^2 effect size	ANOVA-p	ω^2 effect size	ANOVA-p	ω^2 effect size
Motorola Moto G10	Native / KMP	0.272	-	0.017	0.15	0.569	-
	Native / React Native	0.006	0.21	0.246	-	1.44E-04	0.38
	KMP / React Native	0.065	-	0.348	-	1.19E-04	0.39
Google Pixel 4a	Native / KMP	0.096	-	7.54E-06	0.49	0.078	-
	Native / React Native	0.006	0.20	0.248	-	0.002	0.27
	KMP / React Native	0.077	-	0.001	0.29	0.188	-
Motorola Moto G54	Native / KMP	0.329	-	0.811	-	0.774	-
	Native / React Native	0.033	0.12	0.554	-	0.451	-
	KMP / React Native	0.258	-	0.680	-	0.523	-

Appendix B: Digital

Appendix B1: Literature analysis	*1 Literature review \*
Appendix B2: APK files	*2 Application development \\ APK files \*
Appendix B3: Test sequences	*3 Performance comparison \\ Test sequences \*
Appendix B4: Measurements	*3 Performance comparison \\ Measurements \*
Appendix B5: Example CSV export	*2 Application development \\ CSV export \*
Appendix B6: Source code	*2 Application development \\ Source code \*
Appendix B7: Keystores	*2 Application development \\ Keystores \*
Appendix B8: ChatGPT transcripts	*2 Application development \\ ChatGPT transcripts \*
Appendix B9: Statistics	*3 Performance comparison \\ Statistics \*

Eidesstattliche Erklärung

Hiermit versichere ich an Eides statt, dass ich die vorliegende Arbeit im Masterstudiengang Informatik selbstständig verfasst und keine anderen als die angegebenen Hilfsmittel – insbesondere keine im Quellenverzeichnis nicht benannten Internet-Quellen – benutzt habe. Alle Stellen, die wörtlich oder sinngemäß aus Veröffentlichungen entnommen wurden, sind als solche kenntlich gemacht. Ich versichere weiterhin, dass ich die Arbeit vorher nicht in einem anderen Prüfungsverfahren eingereicht habe.

[illegible]

Hamburg, den 12.02.2024

Einstellung in die Bibliothek

Ich stimme der Einstellung der Arbeit in die Bibliothek des Fachbereichs Informatik zu.

[illegible]

Hamburg, den 12.02.2024