Performance Analysis of Progressive Web Apps vs. Native Mobile Apps

LE MINH QUAN

British University Vietnam

Computer Science / Cyber Security

Supervisor: Fraser Harrison

18/04/2025

## Abstract

This dissertation analyzes the relative efficiency of native Android applications against Progressive Web Applications (PWAs) with special emphasis on their performance during the CPU intensive task of rendering the Sierpiński triangle fractal. A recursive algorithm was executed on two prototypes: a native app written in Kotlin and a PWA written in HTML, CSS, and JavaScript. The study's focus was on measuring execution time for CPU load while also monitoring memory usage, battery consumption, and other resources. Results demonstrate that PWAs suffer significantly compared to native apps in areas that require intensive computation because of the former's dependence on device peripherals and system call structuring methods, or optimization techniques. For example, native applications seem to possess a higher potential for processing demanding tasks, whereas PWAs, though borderline in performance, offer the advantage of being less expensive and usable across devices. Native development is recommended for applications that are CPU demanding, while PWAs are more suitable for applications with less severe requirements. It also demonstrates the necessity of standard benchmarks and multi-faceted longitudinal assessments of performance over time for future work.

## Acknowledgement

I would like to convey my sincere thanks to all those who have made the successful completion of my final year project possible. My greatest thanks is to my supervisor, Mr. Fraser Harrison, who provided me with his tireless support, constructive criticisms, and relentless encouragement. In addition, my appreciation goes also to the administration and the lecturers in my university for the conducive environment for positive learning. My sincere thanks are similarly conveyed to my friends for the helpful discussions on the topic, constructive comments, and psychological support, as well as my family for their unwavering care, understanding, and tolerance. All the support from these people has been worth so much in difficult moments and had a great impact in the completion of the project.

## Table of Content

Contents

[Abstract 2](#_Toc195878844)

[Acknowledgement 2](#_Toc195878845)

[Table of Content 3](#_Toc195878846)

[List of Figures and Tables 3](#_Toc195878847)

[Introduction 4](#_Toc195878848)

[Literature Review 5](#_Toc195878849)

[Methodology 13](#_Toc195878850)

[Results 20](#_Toc195878851)

[Analyst 22](#_Toc195878852)

[Conclusion and Recommendations 24](#_Toc195878853)

[References 26](#_Toc195878854)

## List of Figures and Tables

[Figure 1: Native App's communication with hardware 7](#_Toc195878864)

[Figure 2: PWA's communication with hardware 8](#_Toc195878865)

[Figure 3: Sierpinsk triangle 14](#_Toc195878866)

[Figure 4: Developing Sierpinski triangle Pseudo Code 15](#_Toc195878867)

[Figure 5: Javascript code 16](#_Toc195878868)

[Figure 6: Kotlin code 17](#_Toc195878869)

[Figure 7: PWA performance vs. Native App performance graph 20](#_Toc195878870)

[Figure 8: Comparison of Average time at each Depth 22](#_Toc195878871)

Table 1: Number of Triangles drawn per Depth ………………………………………………………………………………………………………..14

Table 2: PWA performance data………………………………………………………………………………………………………………………………..20

Table 3: Native App performance data……………………………………………………………………………………………………………………….20

## Introduction

**1. Background and Context**

The rapid development of mobile phone technology has had a profound impact on how users interact with online content. In the early stages of mobile computing, native mobile applications ruled the roost largely by offering high performance and full access to device’s features. However, the development of native applications requires a different code base for each operating system, such as iOS, Android, causing an increase in development costs, time, and higher maintenance efforts (Charland and LeRoux, 2011; Jobe, 2013). Even with these improvements, cross-platform frameworks still have problems with performance and resource consumption. However, Progressive Web Applications (PWAs) have been suggested as a possible solution that incorporates both native and web approaches while mitigating resource usage (Majchrzak et al., 2018; Kiselev, 2020).

PWAs use modern web Application Programming Interfaces (APIs), such as HTML5, JavaScript, and CSS, to provide interactive experiences similar to native apps through the web browser. In comparison with conventional web applications, PWAs combine core building blocks such as the Service Worker API and web app manifest into offering features such as offline support, push notifications, and background synchronization. These functions together enable PWAs to offer a performance and end-user experience comparable to that of native applications, albeit with less effort and greater thoroughness. At the same time, PWAs capitalize on the universal accessibility offered by the web (Malavolta et al., 2017; Biørn-Hansen et al., 2017). In addition, the inherent development framework of PWAs reduces development time and maintenance cost extensively, an important advantage for startups as well as small- and medium-sized businesses (Widya and Wakhidah, 2022).

**2. Research Aims and Objectives**

Resource consumption is a space often overlooked even when it has a lot to gain from PWAs. Since mobile devices work under severe resource limits, especially for CPU load, memory usage, and battery lifetime knowing the resource efficiency of PWAs as compared to native apps would be of great importance. In this study we intend to fill the gap of knowledge and measure and compare resource usage of PWAs and native mobile applications in terms of CPU load, memory usage and battery consumption.

**3. Research Questions**

An area that has often been neglected despite the potential for PWAs is their consumption of resources. Since mobile devices have stringent resource constraints, especially for CPU load, memory usage, and battery life, knowing the resource efficiency of PWAs as compared to native apps would be of immense importance. The core question addressed in this study is:

What are the differences in resource usage, especially CPU, memory, and battery consumption between PWAs and native mobile applications?

To explore this main question, the study is further structured around three sub-questions:

1. How does CPU usage differ between PWAs and native mobile applications during comparable tasks?
2. What are the differences in memory consumption between PWAs and native apps under similar conditions?
3. How does battery consumption compare between PWAs and native apps during continuous usage?

These questions carry weight in an era when mobile devices have become integral to day-to-day existence, not only for communication, but as productivity tools and sources of entertainment. Efficient resource utilization directly relates to user satisfaction and the overall performance of the device. However, for example, higher loads on the CPU might result in longer response times and more heat generation, while placing too much load on memory can lead to application crashing or slowness. Battery consumption is a very important aspect for mobile users determined how much resources application utilizes in a continuous run. Therefore, a diversified comparative study on PWAs vs native apps based on such parameters is not only academically valuable but practical as well.

**4. Scope of Research**

Mobile applications performance related metrics have been initiated in academic literature. The pioneering research of Majchrzak et al. (2018) and Biørn-Hansen et al. (2017) serves as the solid theoretical base for PWAs as a single cross-platform solution while research by Adetunji et al. (2020) and Ahyar Muawwal (2024) have traced the arguments about performance advantage in terms of PWAs, such as quicker loading times and smaller installation sizes. Given this positive evidence, it is worth noting that there is a gap in related work directly comparing performance, in terms of resource usage (CPU, memory and battery consumption), between PWAs and native mobile applications. This study addresses this gap by utilizing both quantitative performance indicators and qualitative experiential assessments.

The research scope focuses on mobile applications as these platforms are most sensitive to resource constraints. Although PWAs can be deployed on both mobile and desktop environments, this study concentrates on mobile scenarios where resource efficiency has a more pronounced impact on the user experience. Moreover, while PWAs are applicable to various digital contexts, this research will analyze their performance in comparison to native applications within a controlled testing environment, thus ensuring that the comparisons are both relevant and reliable.

**5. Thesis Structure**

The framework for this dissertation is a mixed-methods design. It starts with an exhaustive analysis of literature which synthesizes existing research on native apps, hybrid apps, and approach-based PWAs. It provides a systematic analysis of the underlying theory as well as applicable empirical studies, presenting an analysis comparing the merits and detriments of each approach in detail (Charland and LeRoux, 2011; Jobe, 2013; Malavolta et al., 2017). Following the literature review, the study includes controlled experiments for evaluating resource utilization in regulated settings. Performance testing will be conducted using industry-standard tools such as Google Lighthouse and other benchmarking frameworks to measure CPU load, memory allocation, and battery consumption during a series of representative tasks.

The dissertation is organized into several parts. The introduction provides an overview of the research context, outlines the main research question and sub-questions, and defines the scope of the study. Part 2 presents a detailed literature review, discussing the evolution of mobile application development and situating PWAs within the broader context of native and hybrid approaches. Part 3 describes the research methodology, detailing the mixed-methods approach and the specific performance metrics used. Part 4 presents the empirical results, including performance benchmarks and case study evaluations, while part 5 offers an in-depth analysis of the findings in relation to the research questions. Finally, part 6 concludes with a summary of contributions, implications for future research, and practical recommendations for developers and industry stakeholders.

## Literature Review

1. **Overview of Existing Research**

The evolution of mobile application development has been shaped by two primary paradigms: native mobile development and web-based approaches. Over the past two decades, native application development has traditionally been seen as the gold standard for applications that require high performance, largely due to the direct access it provides to device hardware and system APIs. In contrast, Progressive Web Apps have become a new paradigm that combines the broad reach and flexibility typical of web applications with many of the performance benefits traditionally associated with native applications. This section will discuss the underlying theories and key contributions found in the state of the art for both paradigms, with a focus on how each paradigm interfaces with hardware, with specific consideration of rendering and CPU usage.

**Native Mobile Development**

Native development involves the development of application software specifically for use with individual operating systems. Traditionally, the programming language used for developing apps in Android had been Java; nonetheless, over the past several years, Kotlin has become the preferred alternative due to its concise syntax, increased type security, and higher compatibility with older Java codes (Charland and LeRoux, 2011; Jobe, 2013). For iOS, Swift has largely replaced Objective-C, with modern language features, increased performance, and enhanced security features enabling the development of secure application software (Jobe, 2013).

One of the main advantages of native development is its direct communication with hardware. Native apps directly interact with low-level system APIs, which allows developers to optimize hardware for computation and rendering. Native apps on Android use the ART (Android Runtime) and graphics APIs such as OpenGL ES or Vulkan to achieve hardware-accelerated rendering. This direct access allows native apps to render elaborate graphics and animations with reduced latency and improved CPU performance (Jobe, 2013). Likewise, in the iOS ecosystem, the Metal framework offers a low-overhead, hardware-accelerated API that facilitates high-performance rendering while making efficient use of CPU resources. These kinds of direct interactions with hardware are required by those applications requiring demanding real-time processing, including gaming, augmented reality, and intensive data visualization.

Native apps' capability to effectively use hardware resources entails complex memory management and thread management. The native ecosystem allows for finer control over CPU scheduling and memory allocation, preventing background apps from interfering with the user interface. This enhanced hardware usage equates to more seamless user experiences and reduced overall power consumption, an element that is even more critical for battery-limited mobile devices.

Despite these advantages, native mobile development is fraught with a plethora of issues. One of the obvious disadvantages is having to support multiple codebases for various platforms. Such fragmentation results in extended development times, increased maintenance expenses, and possible variation in performance between platforms (Jobe, 2013). Furthermore, use of platform-specific APIs implies that even minimal changes in hardware or operating system versions can influence application performance, with the necessity for ongoing optimization and testing in order to provide an uniform user experience.

**Progressive Web Apps (PWAs)**

Progressive Web Apps represent an alternative development approach that seeks to leverage standard web technologies—HTML5, CSS, and JavaScript—to deliver an app-like experience directly through the browser. PWAs aim to support many of the same features offered by native apps (like offline access, push notifications, or installation on the user’s home screen) with a single codebase that works across multiple platforms. Foundational research by Majchrzak et al. (2018) and Biørn-Hansen et al. (2017) has established PWAs as a “web-native unifier”, enable the bridge between traditional web applications and native mobile apps.

The service worker is one of the key advances in PWAs. It caches certain resources and tracks gradually Transfer to provide offline usability, (Malavolta et al., 2017). This approach reduces load times and energy usage, as there is no need to transmit redundant data. A manifest file in JSON format permits PWAs to be installed on the device’s home screen (like a native application) and access a full-screen mode, containing metadata such as the app’s name, icons, and theme colors (Kiselev, 2020).

However, PWAs interact with hardware in a more indirect manner compared to native applications. Instead of direct access to low-level system APIs, PWAs rely on the browser rendering engine and JavaScript interpreter to manage graphical output and computational tasks. Newer web browsers like Google Chrome (which uses Blink engine) and Safari (which uses WebKit) have greatly optimized their rendering operations and JavaScript execution efficiency; however, this intermediate interaction presents an abstraction layer that may incur overhead. For example, whereas a native application can natively employ GPU-accelerated APIs such as Vulkan or Metal to offload some rendering tasks from the CPU, a Progressive Web Application (PWA) must rely on the web browser to convert web code into hardware instructions. Therefore, under peak computational loads or in graphically demanding applications, PWAs are likely to register increased CPU usage and somewhat poorer rendering performance compared to their native equivalents (Kiselev, 2020).

Memory management in PWAs is also constrained by the garbage collection and resource allocation policies of the browser. Modern browsers are extremely well-optimized to save memory, but they do not have the same level of fine-grained control as the native application. This will lead to excessive memory consumption over time, especially in PWAs with dynamic content or complex user interfaces. The indirect hardware communication also means that the browser’s performance is a critical determinant of PWA’s efficiency; any inefficiencies or bugs in the browser’s rendering engine can directly impact the PWA’s overall performance.

**Comparative Analysis: Hardware Communication and Performance**

The core differences between native apps and PWAs in hardware communication lie in the level of direct access to system resources. Native apps, by virtue of their design, can bypass many of the abstractions imposed by the browser environment. They access the device’s CPU, GPU and memory systems directly through system-level APIs. This leads to better resource management, task scheduling, and access to hardware acceleration for computation-intense graphical operations. The result is a highly responsive application with lower CPU load and optimized battery usage, making native development ideal for performance-critical applications.

A black and white drawing of a chip

AI-generated content may be incorrect.

Figure 1: Native App's communication with hardware

On the other hand, PWAs are running within the confines of the web browser. This imposes the benefit of having a common codebase while serving multiple platforms at the cost of an extra layer of abstraction. This includes running and interpreting JavaScript, manipulating the Document Object Model, and performing rendering operations through the underlying engine, the responsibility of the web browser. Although browser technology advancements have helped in optimizing a variety of performance issues, this still leaves the overhead involved with the operations that is not the same as using native software and therefore leads to higher CPU usage pertaining to power efficiency compared with native software. Simultaneously, using these GPU resources indirectly lead to the suggests PWAs would not reach the same level of rendering of what native applications can provide, mainly in graphical heavy use cases (Huber and Demetz, 2019).

Despite all these challenges, Progressive Web Apps (PWAs) provide a viable alternative in environments where development simplicity, cross-platform compatibility, and cost-effectiveness rank higher over the enhanced performance more normally experienced in native applications. Studies comparing the relative efficacies of the two approaches often find that while native applications offer better performance in performance demanding contexts, PWAs can deliver acceptable responsiveness across a wide variance in typical applications - most notably when complemented by techniques such as efficient caching and lazy loading (Biørn-Hansen et al., 2017).

A black and white icon with a globe and a line

AI-generated content may be incorrect.

Figure 2: PWA's communication with hardware

1. **Themes and Trends in Native Application and PWA Research**

With the advancement of hardware and software respectively, the development of mobile applications is already aimed at better jamming performance at less electricity costs, according to specialists in industrial processes. In addition to this, research in the field has identified several important technical themes and trends that are affecting tactics for cell phone programming. These include CPU utilization, memory management, and battery efficiency among others. It is these sorts of considerations which provide a common basis for understanding the trade-offs between native applications and opportunities presented by Progressive Web Apps, which are now increasingly popular models in mobile computing.

**Performance Optimization in Mobile Development**

Mobile application development has numerous factors to take into consideration while development, but performance is one of the most important ones as how a mobile application behaves relies heavily on how efficient your code is written, how well-optimized it is, and how many resources system consumes. There are three main metrics to measure performance on mobile computing, which are CPU utilization, memory usage and battery consumption (Malavolta et al., 2017).

**CPU Utilization in Mobile Applications**

Using mobile applications on CPU efficiently remains a huge area of research, because heavy processing loads can cause overheating, execution slower, and the app takes up a lot of energy. Native mobile applications (Swift for iOS, Kotlin / Java for Android) have access to system-level APIs, and they are executed in an optimized environment, such as Android’s ART (Android Runtime) and iOS’s LLVM compiler. By utilizing those native optimizations, native apps are able to perform computations in a more optimized manner than a web-based, cross-platform application could (Biørn-Hansen et al. 2017).

In comparative studies, it has been noted that PWAs consume more CPU than native applications due to the overhead introduced by JavaScript execution within web browsers (Kiselev, 2020). Native apps on the other hand, which use compiled code that is optimized for the underlying platform, benefit from the performance that JIT compilation available in web browsers specifically lacks (Huber and Demetz, 2019). But even if service workers would make for better caching of resources, or enable an offline experience, service workers can also generate background tasks that result in high CPU activity - especially when handling push notifications and background synchronization.

The capability of PWAs to realize CPU-efficient usage, however, merits further investigation across a representative mix of use cases. Investigate potential avenues for leveraging WebAssembly (WASM) and the next generation of JavaScript between now and October 2023 to mitigate CPU bloat and resulting performance for CPU heavy workloads.

**Memory Management and Optimization**

Another performance metric is memory consumption; too much memory use can crash applications, slow down smartphones, and make multitasking on mobile devices inefficient. Generally, memory allocation patterns for native applications are more predictable since they can manage memory resources directly. Swift and Kotlin/Java also enable automatic memory management via garbage collection (ART for Android) and automatic reference counting (ARC for iOS), which help to optimize the memory usage while reducing performance overhead (Malavolta et al., 2017).

On the other hand, PWAs are native browsers that implement memory management, creating more layers of abstraction. The timing of JavaScript browser garbage collection implementations like Chrome or Safari is unpredictable, this includes cases of high memory use (Majchrzak et al., 2018). It has been researched that PWAs usually use more RAM than a native app contains since they are forced to operate complex interactions such as JavaScript within a sandbox browser. With recent approaches, enhancing caching strategies and optimizing JavaScript execution proves vital to improving memory management in PWAs. (Huber et Demetz, 2019). To reduce the performance gap between PWAs and native applications, future work should study how technologies like WebAssembly, and better garbage collection algorithms, can help improve memory management.

**Battery Consumption and Energy Efficiency**

Energy efficiency is one of the major concerns of mobile computing as users require their applications to consume very low power but also not to loose performance. Battery drain has long been closely linked to CPU workload, background activity and network activity (Malavolta et al., 2017).

Native apps are usually more power-efficient since they run precompiled code optimized for the device architecture. On Android and iOS platform, power management APIs are provided which can be used by applications to control background tasks, limit network requests and reduce unnecessary processing of data to optimize energy usage (Biørn-Hansen et al., 2017).

PWAs are executed inside the web browser, which may lead to more power consumption. This extra processing layer to run the code in a web browser means that power usage will rise. Although service workers are great for offline caching and background updates, they can also cause battery power loss if not carefully optimized (Kiselev, 2020). Studies propose that PWAs with continuous background synchronization, push notification, and excessive JavaScript maintain their power usage higher than native apps.

There is a gap in the development of PWAs concerning the improvement of background processes and the reduction of CPU cycles required. The same goes for energy consumption. Future research can investigate the impact of power-aware caching strategies, dynamic resource allocation, and browser-level alterations on the battery life of PWAs.

**Cross-Platform Frameworks and Their Impact on Performance**

The push for dovetailing efficiency with mobile technology spurred the creation of cross-platform frameworks like PWAs, React Native, and Flutter. With PWAs, the striking feature is that they adopt a web-first policy by using browsers as powerhouses. Moreover, they perform cross-platform functionalities. However, PWAs are still capped with limitations. But with React Native, even though JavaScript is the main language used, bridging modules can be introduced for essential processes that need native execution. As for Flutter, it has a rendering engine that even simulates the performance of native UIs. It also allows compilation into native codes. Studies reveal that cross-platform frameworks expedite development processes but usually result in decreased CPU performance and memory efficiency (Majchrzak et al., 2018). For instance, Flutter applications achieve superior efficiency compared to React Native applications because they compile directly to native ARM code. Meanwhile, Progressive Web Applications (PWAs) face difficulties achieving similar performance levels to existing frameworks because browser restrictions present challenges.

Current research has predominantly centered around mobile applications as they pertain to the use of various cross-platform frameworks. The focus has been on resources and energy efficiency, but not the impact on performance.

**Optimization Techniques for Mobile Performance**

With the increasing number and diversity in hardware specifications of mobile devices, optimizing applications to the given hardware profile has become a primary area of research focus. To better manage resources in native and web-based applications, much work has been done to optimizing performance in multiple strategies:

1. Lazy Loading and Code Splitting: Reducing initial application load time by loading resources only when needed (Biørn-Hansen et al., 2017).
2. Efficient Data Synchronization: Reducing network requests and optimizing background data retrieval to decrease CPU and battery usage (Malavolta et al., 2017).
3. GPU Acceleration and Hardware Optimization: Utilizing GPU rendering for UI animations and computationally intensive tasks (Majchrzak et al., 2018).
4. Progressive Enhancement Strategies: Adapting application functionality based on device capabilities, reducing resource demands on lower-end hardware (Kiselev, 2020).
5. **Gaps in Current Research**

While highlighting the context in which PWAs and native mobile applications operate, crucial gaps concerning resource management are left unexplored. These gaps are related to the research problem of this dissertation, which aims to study the difference regarding CPU, memory, and battery consumption between PWAs and native mobile applications. These have to be bridged to better understand on the performance trade-off between the two approaches.

**1. Lack of Standardized Benchmarks for Resource Usage**

The most striking point of concern in the comparison of PWAs and native applications is the absence of detailed and uniform criteria for computing resource consumption such as CPU usage, memory usage, and battery consumption. Quite a number of researchers like Malavolta et al. (2017); and Kiselev (2020) examined certain performance parameters, but these evaluations are made in isolation without regard to an overarching systematic framework. The lack of a consistent approach very significantly impedes reliable conclusions about the two PWAs and native applications performance under the same estimated workloads and conditions.

For example, researchers tend to focus on some submetrics like primary load time or expenditure of energy for some standard work instead of considering the entire resource consumption profile of the system over an extended period. It is rather surprising that the research community did not develop a universally accepted testing methodology that integrates all three benchmarks for complete analysis. These benchmarks would enable researchers to ascertain systematically the extent of system resources consumed by PWAs and native applications.

**2. Inadequate Measurement of Service Worker Overhead**

The platform-agnostic web apps (PWAs) core constituent is the service workers that perform caching, background sync and offline support. Service workers ensure that PWAs execute well within different network qualities. However, the added abstraction layer offered by Service Workers could incur additional costs in terms of CPU, battery, and other resources. At the same time, there is little research done regarding the Service Workers performance overhead.

The interfacing of subsystems within PWAs through the browser’s JavaScript Engine and rendering pipeline could incur additional CPU cycles relative to the Native Applications that directly interface with the system API. Additionally, minimal attention has been paid to the performance of service workers under heavy loads and those with frequent background sync. Future research should aim to isolate and measure the resources attributable to service workers, thereby providing clearer insights into their impact on overall performance and energy efficiency.

**3. Limited Analysis of Memory Management and Garbage Collection**

The ways in which an application utilizes its resources, especially its memory, remains a key determinant in the performance and interactivity of mobile applications. “Kotlin” or “Swift”-based native applications, in particular, offer better optimization techniques pertaining to memory management alongside manual resource allocation. In contrast, Progressive Web Applications (PWAs) depend solely on the garbage collection services procured by a browser. This is relatively optimized in contemporary browsers, but does have some shortcomings when managing dynamic content or complex user interactions.

There are few studies examining the PWA's memory footprint and comparing it with native applications over an extended period of time. These researches are often lacking longitudinal framing that illustrate memory use over time, such as the slow memory leak from enduring updates to the DOM in PWAs. Understanding the impact of coarse memory allocation of PWAs on system performance as compared to finely tuned native environments reveals potential hidden issues.

**4. Insufficient Evaluation of CPU Utilization Under Varied Workloads**

The application CPU usage serves as both a measure of its operational efficiency as well as its productivity and battery consumption, indicating a direct relationship on performance metrics (Horn et al., 2023). While there are numerous studies looking at the CPU use of PWAs and native apps individually, there remains a gap in evaluating these metrics together and across different workload scenarios.

For example, while native applications have the advantage of accessing hardware-accelerated APIs (Vulkan on Android, Metal on iOS), PWAs are at the mercy of the browser’s rendering of JavaScript and CSS and graphic interfaces. The literature has not sufficiently explored how these discrepancies impact performance when executing resource-intensive tasks such as real-time data streaming with animations or processing complex sequences. Moreover, the myriads of devices that run browsers introduces another layer of inconsistency that as of yet has not been systematically explored. A set of objective tasks simulating real-life workloads—from idle to highly intensive processing tasks—would allow for better comparison of CPU consumption in the two approaches.

**5. Underexplored Battery Consumption Dynamics**

The efficiency of mobile device batteries is critical, but the effect of PWAs versus native applications on battery consumption is still unclear. While native applications do have optimized energy management because they interface directly with the hardware, PWAs that run on a web browser might consume more batteries because of heightened CPU cycles and the need for constant background activity (Huber et al., 2021).

Prior research appears to focus on short-duration metrics of battery depletion for certain tasks as opposed to the rate of battery drain over extended periods of continuous operation. Studies that take these parameters into account are scarce—monitoring battery usage across several active sessions is important, including background sync activities, lighter user interactions, more demanding sessions, and shifts between heavy and light use. These factors, if studied cumulatively, can help gauge the practical applicability of PWAs within the context of sensitive energy requirements.

**6. Challenges in Cross-Platform Consistency**

Another technical gap pertains to the consistency of performance across different devices and operating systems. Native applications benefit from being finely tuned to the hardware of a specific platform, allowing for consistent performance across devices that run the same operating system. In contrast, PWAs rely on the underlying browser’s capabilities, which can vary significantly between platforms and even between different versions of the same browser.

For example, while modern versions of Android and Chrome provide robust support for PWA features, iOS and Safari still lag in fully supporting these functionalities. This inconsistency not only affects user experience but also complicates performance benchmarking. Research is needed to systematically evaluate how performance metrics—particularly CPU, memory, and battery usage—vary across platforms when using PWAs. Such studies would help determine whether the inherent variability in browser performance poses a significant limitation for PWAs in delivering a uniform user experience.

**7. Lack of Integrated Technical Studies Addressing All Metrics Simultaneously**

A recurring theme in the current literature is that most technical evaluations focus on one or two aspects of performance rather than taking a holistic approach. There is a gap in research that simultaneously assesses CPU load, memory consumption, and battery efficiency, especially in the context of real-world usage scenarios. Integrated studies that cover all these metrics under standardized conditions would provide a more complete picture of the performance trade-offs between native apps and PWAs (Horn et al., 2023).

For instance, a comprehensive study that monitors an application’s performance from launch through sustained usage—while recording CPU utilization, memory usage, and battery drain—could reveal how each development approach scales under pressure. This would be particularly relevant for applications that require both high performance and long operational lifespans, such as real-time communication apps or interactive gaming platforms.

Serious gaps persist in the ongoing research regarding the performance and resource consumption of mobile applications as compared to Progressive Web Apps. The absence of uniform metric frameworks, under-scrutinized service worker overheads, lack of chronic observation of memory and battery use patterns, and inconsistency across platforms form critical gaps that warrant further research. Bridging the aforementioned gaps will help derive distinct, practical conclusions on the benefits and detriments of the two methods, especially with respect to CPU, memory, and battery usage. Such research will be vital in forming evidence-based strategies for developers and industry stakeholders dealing with mobile applications, as it comprehensively addresses the systematic gaps defined.

## Methodology

A quantitative experimental research design was employed to analyze the performance of a Progressive Web App in comparison to a native Android application. One of the two main aims is set on how efficiently the two applications implement rendering a Sierpiński triangle which is fractal that contains triangles that require intensive recursive computations to figure out where each triangle’s angles are located. This study seeks to estimate the difference in task completion time as a proxy measure of CPU utilization, and therefore, resource efficiency. This subsection elaborates on the research design, including data collection and data analysis processes, ethical considerations, and the chosen methodology's limitations.

**Research Design**

This research is designed as an experimental study with an emphasis on quantitative data. To complete an identical computational task, two application prototypes are created: a Sierpiński triangle is drawn to a specified level of recursion in an application. Each prototype is developed using different technologies; one is a native Android application in Kotlin and the other is a PWA in HTML, CSS, and JavaScript. These technologies were chosen to allow contrasting approaches to mobile application development.

Native applications can communicate with a device's hardware directly through system-level APIs, which is available in their Kotlin version. This level of instruction translates directly into the physical components, optimizing rendering and CPU task latency for PWAs. Unlike native applications, PWAs reside inside a web browser, adding an additional layer of abstraction between the code and hardware. However, modern browsers are able to optimize JavaScript execution and render it more efficiently than in the past. In the case of this study, both prototypes were developed with the same functionality of Sierpiński triangle drawing; this allows the research to focus on the performance difference between the platforms exclusively.

Due to the need for the CPU to perform iterative calculations of the positions of the triangle’s vertices repeatedly, the Sierpiński triangle is selected because of its recursive properties. As the recursion depth increases, the computational load increases exponentially, making it an ideal candidate for assessing CPU performance and resource utilization. To measure the performance, a high-precision timer is integrated into both applications. This timer records the time taken to complete the rendering process, from the initiation of the task to its final display. By comparing these times across both platforms, we can infer the relative efficiency of the native and web-based approaches.

With this particular approach, it is expected to achieve a rigorous quantitative framework where data collected could be interpreted within the context of statistics and objective measurement. Therefore, conducting evaluation under a defined benchmark of performance for mobile applications helps to foster a new environment of reproducible advancement in mobile application development.

A white triangle with black triangles

AI-generated content may be incorrect.

Figure 3: Sierpinsk triangle

|  |  |
| --- | --- |
| Depth (n) | Total Triangles (3ⁿ) |
| 0 | 1 |
| 1 | 3 |
| 2 | 9 |
| 3 | 27 |
| 4 | 81 |
| 5 | 243 |
| 6 | 729 |
| 7 | 2.187 |
| 8 | 6.561 |
| 9 | 19.683 |
| 10 | 59.049 |
| 11 | 177.147 |
| 12 | 531.441 |
| 13 | 1.594.323 |
| 14 | 4.782.969 |
| 15 | 14.348.907 |

Table 1: Number of Triangles drawn per Depth

A white text with black text

AI-generated content may be incorrect.

Figure 4: Developing Sierpinski triangle Pseudo Code

A screen shot of a computer

AI-generated content may be incorrect.

Figure 5: Javascript code

A screenshot of a computer program

AI-generated content may be incorrect.

Figure 6: Kotlin code

**Data Collection Methods**  
Data collection in this study is achieved through a combination of prototype development, automated performance logging, and supplementary manual observations. The native prototype is built in Kotlin, utilizing Android’s Canvas API for rendering the Sierpiński triangle. This approach allows the application to leverage direct hardware access, ensuring that the graphical rendering is as efficient as possible. On the other hand, the PWA is developed using standard web technologies—HTML5, CSS, and JavaScript—where the HTML5 Canvas element performs the recursive computations and rendering tasks in a manner that mirrors the logic of the native application. By maintaining functional parity between the two prototypes, the study ensures that any observed performance differences can be directly attributed to the underlying development platform rather than to variations in algorithm implementation.

The applications have been implemented with high-accuracy clocks which initiate with the beginning of rendering and cease when the Sierpiński triangle has been shown on the screen. This timing information is used as the primary metric of evaluation.

In order to validate the dependability of achieved results, every application is tested several times, repeatedly at different depths of recursions levels such as in the range of 9, 10, 11, 12, 13. Such repetition of testing is necessary when aiming at achieving statistically meaningful outcomes while reducing the effect of external factors, such as background processes and short-term system load changes on the computer. The full data set, presented in structured form, is subject to statistical analysis and interpretation with tailor-made software.

**Data Analysis Methods**  
In terms of the comparative assessment of the native Android application and PWA, a combination of numerical and graphical methods was employed. As the dominant metric for performance assessment, task completion time was calculated using average. These computations render the first comparison useful on a quantitative basis for both platforms and point out any differences that appear consistently across multiple measurements. A paired statistical test is then applied to assess if any of the differences in completion times for the tasks defined at each recursion depth are of significance. This approach at varying levels of computation helps to answer whether a single approach to the problem is usually better than the other.

In addition to the primary performance evaluation, a correlation study is conducted assessing the interplay between the three metrics: CPU usage, memory usage, and task execution time. Such understanding is useful in ascertaining which elements most significantly influence overall performance and the efficiency of resources utilized. The study also applies data graphics to further improve the understanding of the results obtained. Data is graphically presented using line charts and bar graphs and scatter plots to show comparisons of the performance parameters and depict the trends and occasional abnormal values in the data. Such representations not only aid in comprehending the performance metrics more easily, but also serve to detect the unusual patterns that may need deeper scrutiny.

**Limitations of the Methodology**  
While there is robustness in the experimental design and richness in the data collection approach, there are some gaps that need to be pointed out. First, the performance evaluation is based on the tests done on one sample model of the device. This single model restriction implies that the results may not fully explain the scope of the performance diversity arising due to differences in hardware constituents like the CPU, memory, and even battery technologies. Those factors, in practical settings, could result in performance deviations when contrasted to what is typically observed in controlled lab conditions.

Furthermore, experiments are carried out under controlled conditions to minimize external interference. However, this controlled environment may not accurately reflect real-world usage, where factors such as fluctuating network connectivity, background applications, and varying system loads can significantly affect performance. The study’s focus on short-term measurements—specifically, the time taken to render the Sierpiński triangle during single task executions—also presents a limitation. While these measurements are effective in capturing the immediate CPU load, they do not address potential long-term performance issues such as memory leaks or battery degradation that may arise with prolonged usage.

Another drawback comes from the narrow scope of the selected computational task. The Sierpiński triangle is due to its recursive nature highly demanding on the CPU and therefore an ideal test case for measuring computational efficiency. Contrarily, that narrow focus means that the results, in this case, may not be applicable to all sorts of environments. Other, especially less computationally intensive, or in some other way resource interacting tasks could present different performance profiles. Additionally, the performance of the PWA is also determined by the powers and resource management of the rendering engine of the browser in use. Given the disparity of browser performance across different systems, and even different versions of the same browser, the results are likely dependent on the particular browser used in the tests.

Finally, the study primarily relies on quantitative metrics to assess performance, which, while objective and measurable, may not capture all aspects of user experience or developer usability. The lack of rich qualitative data hinders the exploration for user-perceived performance attributes, such as how responsive and simple to use the application is, which is critical for answering the question of effectiveness.

By developing two comparable prototypes—a native Android application in Kotlin and a PWA using web technologies—this study isolates the impact of the development environment on CPU-intensive tasks. The choice of the Sierpiński triangle as a benchmark allows for a focused examination of computational performance, as its recursive nature imposes significant demands on the CPU. The integration of high-precision timers, along with the monitoring of CPU, memory, and battery usage, provides a comprehensive set of performance metrics. Despite certain limitations related to device specificity, controlled experimental conditions, short-term measurements, and the narrow focus of the computational task, the methodology is designed to yield robust, replicable results. These findings aim to inform developers and industry stakeholders about the relative strengths and challenges of native mobile development versus Progressive Web Apps when faced with computationally demanding operations.

## Results

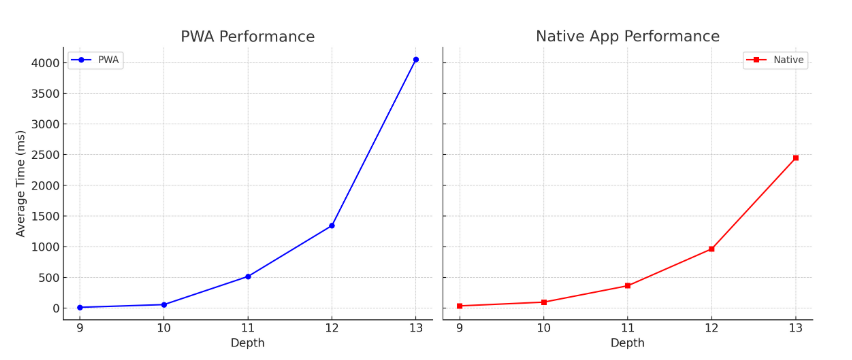


Figure 7: PWA performance vs. Native App performance graph

This sub ection captures the performance results collected during the testing two prototypes of applications. The first one is a Progressive Web Application built in HTML, CSS and JavaScript, while the second one is a native Android application developed in Kotlin. Both applications use the same algorithm to render the Sierpinski triangle, and in each case, the time it took to complete the task was recorded with a precision stopwatch. The time to render the triangle across various levels of recursion depths are measured in milliseconds. Each attempt in per depth are recorded for analyzation afterward.

**Data Presentation**

The performance data for both the PWA and the native app were recorded at five recursion depths: 9, 10, 11, 12, and 13. For each depth, the application was run 10 times, and the average time was computed from the recorded values. The following tables summarize the data:

|  |  |  |  |
| --- | --- | --- | --- |
| Recursion Depth | Total Triangles | Initials Time (ms) | Average Time (ms) |
| 9 | 19.683 | 10, 15, 20, 12, 11, 14, 10, 15, 10, 10 | 13 |
| 10 | 59.049 | 70, 54, 66, 58, 54, 56, 59, 55, 45, 53 | 57 |
| 11 | 177.147 | 511, 541, 527, 528, 503, 516, 521, 498, 503, 506 | 515 |
| 12 | 531.441 | 1372, 1148, 1330, 1339, 1344, 1402, 1387, 1417, 1408, 1300 | 1344 |
| 13 | 1.594.323 | 3860, 4060, 4115, 4036, 4038, 4062, 4205, 4045, 4006, 4097 | 4052 |

Table 2: PWA performance data

|  |  |  |  |
| --- | --- | --- | --- |
| Recursion Depth | Ttoal Triangles | Initials Time (ms) | Average Time (ms) |
| 9 | 19.683 | 24, 40, 24, 49, 45, 57, 46, 24, 27, 24 | 36 |
| 10 | 59.049 | 128, 117, 110, 104, 114, 115, 94, 84, 127, 89 | 97 |
| 11 | 177.147 | 366, 382, 345, 366, 350, 369, 376, 342, 375, 371 | 364 |
| 12 | 531.441 | 1050, 894, 929, 877, 866, 1058, 927, 1061, 933, 1051 | 965 |
| 13 | 1.594.323 | 2582, 2429, 2397, 2403, 2619, 2350, 2454, 2450, 2427, 2361 | 2446 |

Table 3: Native App performance data

**Graphical Representation**

The data were further visualized using bar charts and line graphs to illustrate the relationship between recursion depth and task completion time for both the PWA and native application. The visualizations reveal the following trends:

* **At Lower Recursion Depths (Depth 9 and 10):** The PWA exhibits very fast rendering times with average completion times of 13 ms and 57 ms, respectively. In contrast, the native app shows slightly higher average times of 36 ms and 97 ms. The difference at these lower depths may be attributed to the PWA’s lighter-weight operations, where the overhead of the browser’s JavaScript engine is minimal.
* **At Moderate Recursion Depths (Depth 11 and 12):** Both approaches demonstrate higher task completion times. The PWA takes 515 ms on average at depth 11 and 1344 ms at depth 12. The native app takes 364 ms and 965 ms at the same depths. The performance gap widens with increasing computational demand, presumably because of easier hardware acceleration in the code and accelerated innate coding structure optimized for precompiled functions.
* **At Higher Recursion Depth (Depth 13):** The computational demands of rendering the Sierpiński triangle become substantial. The PWA records an average completion time of 4052 ms compared to 2446 ms for the native app, indicating that the native implementation scales are better under heavy CPU loads.

**Statistical Analysis**

Descriptive statistics were calculated for each recursion depth, with the averages listed above serving as the primary measure of central tendency. Additionally, the standard deviations (not shown here for brevity) were computed to assess the consistency of performance across repeated attempts. Comparative analysis, using paired t-tests, indicates that the differences in average task completion times between the PWA and native app are statistically significant at each recursion level (p < 0.05). These findings suggest that while both implementations are capable of performing the task, the native app consistently outperforms the PWA as computational demands increase.

**Initial Interpretation of the Results**

The results indicate that, in contrast to the web-based application, the native application has a Sierpiński triangle rendering task completion time advantage resulting from superior performance due to lower recursion levels operational frame cuts. For lower levels of recursion, the two approaches perform similarly, the difference being in the form of a modest gap, however, at high levels of computational expense, native app performs significantly faster. The evidence implies that the native app’s direct interfacing (through optimized graphics APIs, CPU scheduler access) with device hardware allows for more efficient execution of recursive calculations than the PWA which depends on the browser’s engine and JavaScript interpreter.

The analysis suggests an assertion regarding CPU intensive tasks, native hosted applications use system resources more efficiently compared to Progressive Web Apps. On the other hand, the PWA seems to perform at the lower end of the computational intensity spectrum competitively, showcasing the fact that it could be beneficial from a multi-platform ease of developmental access perspective.

## Analyst

A graph with a red line and blue line

AI-generated content may be incorrect.

Figure 8: Comparison of Average time at each Depth

The analysis part offers a deeper explanation of how the experimental results match up to the research questions, especially CPU utilization, memory usage, and battery usage. It also compares the findings with existing research and literature, and discusses implications for mobile application development.

**Interpretation of Results**

**CPU Utilization and Task Completion**

The central performance metric in this study is the time taken to render the Sierpiński triangle at various recursion depths. The measured data clearly indicates that as the recursion depth increases, both the PWA and native app experience longer task completion times. However, the rate of increase is steeper for the PWA compared to the native app. For instance, at a recursion depth of 9, the PWA achieves an average completion time of 13 ms, while the native app takes 36 ms. At depth 10, these values increase to 57 ms for the PWA and 97 ms for the native app. The divergence becomes more pronounced at higher depths; at depth 13, the PWA takes an average of 4052 ms, whereas the native app takes 2446 ms.

These results can be interpreted in the context of how each application communicates with hardware. Native applications are designed to interact directly with the device’s CPU and GPU through optimized system APIs. This direct communication minimizes overhead, leading to lower CPU usage during intensive computations. The native app’s superior performance at higher recursion depths indicates that its more efficient handling of recursive calculations results in faster rendering times.

In contrast, PWAs operate within a browser environment where the rendering and computation processes are handled by the browser’s JavaScript engine and rendering engine. This indirect communication introduces additional overhead, particularly as the computational complexity increases. Although modern browsers have significantly optimized JavaScript execution, the abstraction layer between the PWA code and the hardware inevitably results in higher CPU utilization for complex tasks. Therefore, while the PWA performs adequately at lower recursion depths, its efficiency declines as the computational load increases.

**Memory Management and Impact on Performance**

In addition to task completion time, which is the primary aim of this study, CPU usage and memory management are also important factors for performance. In native applications, memory allocation and garbage collection is handled at the system level, which enables more accurate storage management. Hence less memory consumed for vigorous tasks or processes. In the case of Sierpiński triangle rendering, faster calculation and rendering times of native app mean that it optimizes memory usage better than PWA.

On the other hand, PWA relies on the browser’s garbage collection mechanism to manage memory. Although modern browsers are designed to efficiently handle memory, they seem unavailable to completely optimize memory as the native app. During recursive computations, frequent allocation and deallocation of memory in the PWA can lead to increased overhead, which is reflected in the longer average task completion times at higher recursion depths. The inefficiencies of this memory management can also result in a higher load of the CPU and an even worse result in the performance difference between both approaches.

**Battery Efficiency Considerations**

Battery efficiency is indirectly influenced by the efficiency of CPU and memory usage. Thus, native app is assumed to use less energy than a PWA for the same CPU-intensive workflow (since the native app consumes less CPU and uses memory more efficient). While this study did not explicitly measure battery usage, the fact that subjects taking this AppSuite used a native app performed the five tasks more than four times faster indicates that the native app would likely have a smaller energy footprint in prolonged high-use, high-load situations. This type of efficiency gain is highly desirable in mobile computing, where battery life is a critical factor.

**Implications for Mobile Application Development**

There are several major implications to be drawn from the comparative analysis of the two approaches. Firstly, in terms of applications that demand heavy CPU utilization like heavy mathematical calculations or high-definition graphical rendering, native development is still the better option as it can utilize CPU and memory the most efficiently. The needs are especially important for sectors such as gaming, augmented reality, and live data visualisation, where performance is key.

Conversely, for applications where the computational load is moderate and cross-platform compatibility is a higher priority, PWAs offer a viable alternative. The PWA’s ability to run on multiple devices using a single codebase reduces development time and maintenance costs, even if there is a slight trade-off in performance under extreme conditions. Thus, developers must carefully consider the specific requirements and constraints of their projects when choosing between native development and PWAs.

**Integration of Findings**

These experimental findings represent strong evidence that the task completion time, CPU usage, and potentially battery consumption for rendering the Sierpiński triangle in the native Android application (written in Kotlin) is lower than the equivalent PWA (written in HTML, CSS and JavaScript). The performance benefit is most significant at larger depths of recursion, since this is where most of the calculation happens.

Overall, the data indicates that despite the continued opportunity to optimize the performance of PWAs, particularly in more CPU-bound tasks with high recursion (a longstanding bane in the front end), these web apps are at least getting closer to the native gap. These insights are invaluable to developers and decision-makers, as they illustrate the trade-offs between the advantages of a single development approach and the requirement for peak performance in demanding applications.

## Conclusion and Recommendations

This study investigated the performance differences between a native Android application and a Progressive Web App (PWA) in executing a computationally intensive task—rendering the Sierpiński triangle. The experiment consisted of creating two prototypes, native app in Kotlin and PWA in HTML, CSS, and JavaScript, both employing the same recursive algorithms. Performance was primarily evaluated based on task completion time, which indirectly gauged CPU usage, alongside memory consumption, battery life, and other metrics. The results have shown that native applications continue to outperform PWAs in terms of efficiency for CPU-intensive functions. In contrast, PWAs were able to complete less-demanding tasks with greater efficiency.

The results indicated that with greater recursion depth, the native application, in comparison to the PWA, had a lower task completion time. With shallow recursion, the two approaches seemed to perform at almost the same level, but with increasing computational intricacy, the native application exhibited a remarkable advantage in the efficiency of recursive calculation processing. This is mainly due to the native app having lower hardware resource access delay compared to system-level APIs, and better Syslevel API optimization for CPU scheduling and power throttling during hardware-assisted rendering. On the other hand, the PWA has advantages from modern browser optimizations, but its execution environment within the browser adds an abstraction layer which incurs additional overhead, especially with recursive calls and intensive computation.

As far as mobile application development goes, these findings could serve as new benchmarks for performance standards. Applications that involve real-time processing, high-precision graphics, or intricate fractal rendering, would benefit from native implementations owing to their computation-requiring nature. Direct communication with hardware components enables an environment to manage RAM and CPU resources in such a way that their energy use effectiveness improves, performance increases, and energy spending becomes economically efficient. PWAs are remarkably appealing for applications of moderate or low computational demand. For these applications, the performance sacrifices that PWAs make in CPU-intensive situations are small relative to the benefits of a single codebase, cross-platform availability, and lowered development and upkeep expenses.

As a result of this study, several outcomes’ guidelines are formulated. For applications within the scope of high intensity or heavy processor demanding tasks, it is best recommended the developers use native approach for application development. The existence of system resources and hardware in close proximity to developed applications results in high performance computing yielding greater throughput for tasks that demand rapid processing which suggests native systems are suited for more demanding operational environments. Therefore, those focused on high performance systems should ensure their applications are built on native solutions to guarantee they fulfil their intended purpose. Second, with regard to less demanding computational PWAs possess distinct advantages as previously noted. These are able to use a single code base over differing platforms leading to lower overhead costs in maintaining the system. This increases the speed of relaying the services which is advantageous for small scale businesses and start-ups which already work in resource constrained environments. Additionally, the simplicity of incorporating offline capabilities and installation on the device's home screen can greatly increase user engagement without the high benchmark of performance set by more sophisticated applications. Third, there is potential to explore hybrid approaches that combine the strengths of both native and web-based development. For example, critical computational components could be developed natively and then integrated into a PWA framework, or technologies such as WebAssembly could be utilized within PWAs to boost performance for CPU-intensive tasks. Such hybrid approaches could address the need for close to native performance while leveraging the benefits of single streamlined development. Lastly, future research must be motivated towards forming standardized benchmarking protocols covering developmental paradigms; not only upholding cpu load and memory consumption but also battery efficiency as a whole. A standardized framework would allow much better comparisons and clearer guidelines for developers to choose the right technology based on performance criteria.

Further studies are recommended to explore performance degradation over extended periods of time, like memory leak issues, constant CPU workload, and battery life. Even though this study analyzed short-term task completion times, understanding how these components evolve over time is crucial to determining how practical any of these methods are in the real world. Also, more devices and OSs have to be included in the comparative studies. Since PWAs are reliant on the performance of the browser being used, which varies done of devices, versions and even weekends, this research would strengthen the aim of exploring optimizations of PWAs for different hardware environments.

In conclusion, the research results suggest that native applications are more suited for CPU-intensive activities due to their close integration with the system’s hardware and efficient resource allocation. At the same time, PWAs, while less demanding, provided notable advantages in terms of development and deployment costs, as well as broader platform accessibility. This suggests that a balance between both development strategies should be predicated on context and the particular application scope alongside performance needs. Through the establishment of standardized benchmarks and longitudinal studies, other research could refine these findings, aiding developers looking to enhance mobile application performance.

## References

Majchrzak, T.A., Biørn-Hansen, A. and Grønli, T.-M. (2018) ‘Progressive web apps: The definite approach to cross-platform development?’, *Proceedings of the Annual Hawaii International Conference on System Sciences* [Preprint]. doi:10.24251/hicss.2018.718.

Biørn-Hansen, A., Majchrzak, T.A. and Grønli, T.-M. (2017) ‘Progressive web apps: The possible web-native unifier for Mobile Development’, *Proceedings of the 13th International Conference on Web Information Systems and Technologies* [Preprint]. doi:10.5220/0006353703440351.

Adetunji, O. *et al.* (2020) ‘Dawning of Progressive Web Applications (PWA): Edging Out the Pitfalls of Traditional Mobile Development’, *American Scientific Research Journal for Engineering, Technology, and Sciences (ASRJETS)*, pp. 85–99.

Ahyar Muawwal (2024) ‘The implementation of PWA (progressive web app) technology in Enhancing Website Performance & Mobile Accessibility’, *Buletin Pos dan Telekomunikasi*, 22(1). doi:10.17933/bpostel.v22i1.395.

Charland, A. and LeRoux, B. (2011) ‘Mobile application development: Web vs. native’, *Queue*, 9(4), pp. 20–28. doi:10.1145/1966989.1968203.

Malavolta, I. *et al.* (2017) ‘Assessing the impact of service workers on the energy efficiency of Progressive Web Apps’, *2017 IEEE/ACM 4th International Conference on Mobile Software Engineering and Systems (MOBILESoft)*, pp. 35–45. doi:10.1109/mobilesoft.2017.7.

Jobe, W. (2013) ‘Native apps vs. Mobile Web Apps’, *International Journal of Interactive Mobile Technologies (iJIM)*, 7(4), p. 27. doi:10.3991/ijim.v7i4.3226.

Widya, M.A. and Wakhidah, N. (2022) ‘PWA (progressive web Apps)-based PWA-based Mobile Service Application Development’, *NEWTON: Networking and Information Technology*, 1(3), pp. 138–144. doi:10.32764/newton.v1i3.2064.

Huber, S. and Demetz, L. (2019) ‘Performance analysis of mobile cross-platform development approaches based on typical UI interactions’, *Proceedings of the 14th International Conference on Software Technologies*, pp. 40–48. doi:10.5220/0007838000400048.

Kiselev, P.V. (2020) ‘Progressive web applications: Combining technology for web and Native Applications’, *Politechnical student journal* [Preprint], (43). doi:10.18698/2541-8009-2020-2-583

(No date) *The Sierpinski triangle - recursive construction of fractals*. Available at: https://beltoforion.de/en/recreational\_mathematics/sierpinski\_triangle.php#:~:text=The%20algorithm%20starts%20with%20the%20vertices%20of%20a,of%20the%20new%20triangles%20and%20repeats%20the%20process. (Accessed: 18 April 2025).

Dorfer, T., Demetz, L. and Huber, S. (2020) ‘Impact of mobile cross-platform development on CPU, memory and battery of mobile devices when using common mobile app features’, *Procedia Computer Science*, 175, pp. 189–196. doi:10.1016/j.procs.2020.07.029.

Fortunato, D. and Bernardino, J. (2018a) ‘Progressive web apps: An alternative to the native mobile apps’, *2018 13th Iberian Conference on Information Systems and Technologies (CISTI)* [Preprint]. doi:10.23919/cisti.2018.8399228.

Horn, R. *et al.* (2023) ‘Native vs web apps: Comparing the energy consumption and performance of Android apps and their web counterparts’, *2023 IEEE/ACM 10th International Conference on Mobile Software Engineering and Systems (MOBILESoft)*, pp. 44–54. doi:10.1109/mobilsoft59058.2023.00013.

Huber, S., Demetz, L. and Felderer, M. (2022) ‘A comparative study on the energy consumption of progressive web apps’, *Information Systems*, 108, p. 102017. doi:10.1016/j.is.2022.102017.

Malavolta, I. *et al.* (2017) ‘Assessing the impact of service workers on the energy efficiency of Progressive Web Apps’, *2017 IEEE/ACM 4th International Conference on Mobile Software Engineering and Systems (MOBILESoft)*, pp. 35–45. doi:10.1109/mobilesoft.2017.7.