JOURNAL
COMPUTER SCIENCES INSTITUTE

JCSI 37 (2025) 391–398
Received: 14 June 2025
Accepted: 1 July 2025

Performance analysis of Jetpack Compose components in mobile applications

Analiza wydajności komponentów Jetpack Compose w aplikacjach mobilnych

Adrian Kwiatkowski*, Jakub Smołka

Department of Computer Science, Lublin University of Technology, Nadbystrzycka 36B, 20-618 Lublin, Poland

Abstract
This article presents a performance analysis of the Jetpack Compose toolkit components in mobile applications executing typical user tasks. A performance comparison was conducted between specialized and less specialized components. The Macrobenchmark, JUnit, and UIAutomator libraries were used to evaluate the performance of scrollable lists, animations, and dispatchers on three different mobile devices, with each implementation in a given scenario appearing identical. The results from the conducted tests indicate that specialized components do not always have the same or better performance than less specialized components.

Keywords: Android; Jetpack Compose; performance

Streszczenie
Artykuł przedstawia analizę wydajności komponentów zestawu narzędzi Jetpack Compose w aplikacjach mobilnych realizujących typowe zadania użytkownika. Przeprowadzone zostało porównanie wydajności pomiędzy komponentami wyspecjalizowanymi i mniej wyspecjalizowanymi. Wykorzystane zostały biblioteki Macrobenchmark, JUnit i UIAutomator za pomocą, których zbadana została wydajność listy przewijanej, animacji oraz dyspozytorów na trzech różnych urządzeniach mobilnych, przy czym każda implementacja w danym scenariuszu wyglądała tak samo. Wyniki przeprowadzonych badań pozwalają stwierdzić, że komponenty wyspecjalizowane nie zawsze cechują się taką samą, bądź lepszą wydajnością od komponentów mniej wyspecjalizowanych.

Słowa kluczowe: Android; Jetpack Compose; wydajność

*Corresponding author
Email address: s95471@pollub.edu.pl (A. Kwiatkowski)

1. Introduction

The technological advancement of mobile devices has led to a significant increase in the number of users, a trend that continues to grow each year [1]. Currently, the most widely used operating system on mobile devices is Android, which is broadly accessible to users [2]. As a result, a large number of applications are being developed for this platform, often offering similar functionalities but differing in non-functional requirements such as performance, which play a crucial role in determining an application's success and in attracting and retaining the largest possible user base [3-6]. Ensuring high performance is particularly important, as Android users operate a wide variety of mobile devices with differing hardware capabilities, including less powerful ones. Given that user ratings in app stores significantly influence application choice [6] optimizing performance is essential to prevent negative reviews arising from inadequate application responsiveness.

Many popular Android applications available on the Google Play Store use the Jetpack Compose toolkit for user interface development [7]. Jetpack Compose allows developers to build native, declarative user interfaces that update automatically upon detecting changes in the observed state.

The objective of this study is to evaluate the performance of different implementation variants of components in Jetpack Compose by conducting a series of experiments that measure and compare their efficiency on representative mobile devices. The findings aim to support mobile application developers in selecting high-performance components for building user interfaces that facilitate the execution of typical user tasks. The performance analysis will focus on components used for rendering lists, animations, and dispatchers in input and output operations.

2. Literature review

This literature review discusses previous research on the performance and quality-related factors of mobile applications. As there is still a limited number of studies specifically addressing Jetpack Compose, the review also includes research investigating the performance of other technologies. The literature review has been organized into the following areas:

1. Comparison of user interface performance.
2. Comparison of programming language performance.
3. Factors influencing application performance.
4. Factors influencing application quality.
5. Summary of the literature review.

2.1. Comparison of user interface performance

Study [8] compared the performance of applications built with traditional views, Jetpack Compose, and a combination of both, using Macrobenchmark, JUnit, and UIAutomator. In startup tests across cold, warm, and hot modes, the view-based app performed best, followed by Jetpack Compose, with the combination of both approach performing worst. Additional tests on frame rendering during list scrolling and animation showed minor differences. The combination of both performed best, while Jetpack Compose had the most unstable results, possibly due to experimental animation features.

Study [9] conducted a comparative analysis of Jetpack Compose and Flutter in an Android application. The results showed that Flutter had lower CPU usage, while consuming more memory. When comparing application size, Flutter performed better for a calculator app, whereas for a movie list app Jetpack Compose performed better. In duration tests, where execution time of tests was measured, Jetpack Compose performed better for calculator, however, for the movie list app, Flutter outperformed Jetpack Compose. Based on these results, the authors concluded that both Jetpack Compose and Flutter have their strengths and weaknesses, and it is not possible to definitely state which solution is superior.

2.2. Comparison of programming language performance

Programming language performance is an important factor in selecting the appropriate technology for developing a sufficiently efficient application. Previous studies [10-12] have compared the performance of programming languages in terms of both runtime performance and the compilation time. While Java is superior in compiling time and APK size, Kotlin offers a more concise syntax, which can contribute to faster application development, fewer lines of code, and a reduced number of errors [10].

Article [11] compared Java, Flutter, and Kotlin/Native. In terms of build time, size of the installation file, startup time, and RAM usage, Java achieved better results, while Flutter performed the worst. In performance related tests involving operations on collections, REST requests, database, files, serialization, and deserialization, Flutter achieved the best execution times in the majority of tests. However, in database operations, other applications were up to ten times faster. Flutter had the highest RAM usage, and lowest CPU usage, while Java had the lowest RAM usage. Java demonstrated the most stable performance overall. The authors noted that both Flutter and Kotlin/Native were in the early stages of development at the time of testing, and their performance could be improved in the future.

Study [12] conducted an empirical analysis on open-source repositories to evaluate the impact of migrating from Java to Kotlin on application performance. The evaluation covered CPU usage, memory usage, garbage collector invocations, frame times, application size, and energy consumption. Although statistically significant differences were observed in CPU and memory usage, as well as frame time, the magnitude of these differences was negligible. The authors concluded that, due to the lack of evidence indicating a substantial negative impact on performance, there is no major reason for avoiding migration from Java to Kotlin.

2.3. Factors influencing application performance

Previous studies have addressed the factors influencing application performance, including hardware analysis, resource utilization, and the implementation of system and software [13-16]. A survey of existing literature noted that mobile devices are resource-constrained and that there are interdependencies among performance characteristics. For example, CPU frequency can have both positive and negative effects on battery lifetime, while inversely impacting responsiveness [13]. However, this is not always the case [13], [16], as the use of offloading has helped reduce energy consumption while simultaneously improving responsiveness. For these and other performance characteristics, ways to profile and optimize them have been explored. Redundant user interface rendering can cause slow rendering and frozen frames which can be solved by reducing complexity of the UI hierarchy and background, and using better hardware, but better hardware does not mitigate problems with ANR (Application Not Responding) and SNR (System Not Responding) [15].

An analysis of 20 popular mobile application repositories showed that 12 improved at least one nonfunctional parameter: most commonly execution time and memory consumption, with bandwidth usage and frame rate less frequently addressed [16]. Performance gains were often achieved through modifications of multiple files, and trade-offs between metrics, especially between execution time and memory consumption, were common

In [14] the authors investigated the responsiveness of Android software by measuring response time to screen taps simulated by ADB (Android Debug Bridge). They concluded that the most impact on responsiveness is caused by competition for CPU time among concurrent threads with equal or higher priorities. Network I/O and memory utilization do not significantly affect responsiveness unless memory usage approaches 100%. Concurrent disk I/O operations or high disk usage can negatively affect responsiveness. Therefore, it is recommended to avoid performing such operations while the user is actively interacting with the mobile device.

2.4. Factors influencing application quality

An important factor in the success of a mobile application is understanding what influences its quality. Research has shown [3-6] that, both functional and non-functional requirements play a significant role in the success of mobile applications. Low user ratings can negatively affect the popularity and revenue of an app [4-6]. When installing an application, user reviews are considered more important than application size, review content, recommended apps by store, last update date, number of downloads, required permissions, or screenshots [6], while when deciding to uninstall and app, the main reasons are lack of need for further use, and also app crashes or high

resource utilization. There are many reasons why an application is of low quality such as poor performance, crashes, poorly designed user interface or navigation, failure to understand the target audience, lack of communication with users or insufficient marketing as presented in the article [3].

There are differences between countries in user behavior regarding the adoption and abandonment of mobile applications, but despite these differences, users around the world are highly likely to abandon low quality applications, such as an app that runs slowly, crashes or is difficult to use [4]. Since many apps offer similar functionalities, users switch to better alternatives, indicating that, in some cases, non-functional requirements are more important than functional requirements.

User complaints include both functional and nonfunctional requirement problems, with the most common complaints involving functional problems, feature requests, application crashes, and connectivity issues. The most negative complaints were related to privacy concerns, hidden costs, removal of features in updates, and application crashes [5]. The authors emphasize that application update can lead to negative complaints, therefore, it is recommended to conduct, for example, regression tests for features that are planned for removal or monetization, as well as when introducing changes to the user interface.

2.5. Summary of the literature review

The literature review shows that existing research has already addressed the Jetpack Compose toolkit, including studies comparing its performance. However, no prior research has specifically examined how this toolkit behaves when using different components, nor which components are more efficient in executing typical user tasks within an application. Research on factors influencing application performance and quality have shown that both functional and non-functional requirements, such as application performance, are important to users and, consequently, to the success of an application. The study presented in this work aims to systematize existing knowledge and support mobile application developers in understanding which components and modifiers are more performant. Based on literature review, the following research thesis has been formulated: “Specialized components in Jetpack Compose offer better performance compared to more general purpose, non-specialized components”. In addition, a set of hypotheses has been defined to help accept or reject this thesis:

H1. Dedicated LazyColumn component is more efficient than the Column component regardless of dataset size.
H2. Dedicated animations are more efficient than non-specialized animations.
H3. Dedicated dispatcher is more efficient than other dispatchers.

3. Research method

In order to evaluate the performance of specialized and non-specialized components in Jetpack Compose, three research scenarios were designed. Each scenario was implemented within a separate activity using both specialized and non-specialized components. All implementations were designed to maintain identical functionalities and visual appearance. Performance of components and Jetpack Compose itself may vary depending on the mobile device and its specifications. Therefore, the experiments were conducted on three distinct smartphones from different manufacturers, each with varying specifications (Table 1). The devices are further referred to using abbreviated labels: S1, S2, and S3, respectively from left to right.

The testing environment is unstable due to the presence of background processes and the potential for device overheating, both of which may affect performance during testing and contribute to variability between tests. To minimize environmental instability, it is essential to perform a preparatory procedure prior to benchmarking. This includes fully charging the device’s battery, closing all background applications, disabling WI-FI and Bluetooth connectivity, closing applications that can display over other applications, and deactivating screen timeout to prevent the device from entering sleep mode during performance testing. Additionally, the device should be connected to a computer via USB to use ADB, and its screen orientation should be set to portrait. The device must remain idle and unused for the duration of each test. Despite the preparatory procedure, certain uncontrollable factors may still contribute to environmental instability and affect measurement consistency. To mitigate the impact of such variability, each test was repeated ten times, and the final results were obtained by calculating the arithmetic mean of the collected data.

The Macrobenchmark library was used to implement automated performance tests simulating typical user interactions within a mobile application. In addition, the JUnit and UIAutomator libraries were used to construct test cases and simulate user gestures. Macrobenchmark runs in a separate process, which allows it to restart and precompile the target application.

Table 1: Smartphone specifications

Smartphone (ID)	Samsung Galaxy A52s (S1)	Xiaomi Redmi Note 10 5G (S2)	Motorola Edge 40 (S3)
Model	SM-A528B/DS	M2103K19G	XT2303-2
Operating system	Android 14	Android 13	Android 14
CPU	Snapdragon 778G 5G	Mediatek Dimensity 700	Mediatek Dimensity 8020
GPU	Adreno 642L	Mali-G57 MC2	Mali-G77 MC9
RAM	6 GB	6 GB	8 GB
Display	1080 x 2400 px, 405 ppi, 120 Hz	1080 x 2400 px, 405 ppi, 90 Hz	1080 x 2400 px, 402 ppi, 144 Hz

The tested application was build in release mode with R8, reflecting the configuration typically encountered by end users in production environments, instead of debug mode that imposes a performance cost [17]. Furthermore,

baseline profiles were used to improve the performance of the application's initial launch and user interactions, such as navigation and scrolling, by reducing the code interpretation in critical parts of the application.

The obtained results were processed using a script, after which the data were imported into a spreadsheet to generate the corresponding tables and charts. The distribution of the measurements was examined to determine whether they followed a normal distribution. Based on this assessment, either an ANOVA statistical test was performed, or, in cases where normality was not observed, the Kruskal-Wallis test was applied to compare the mean values obtained.

The first research scenario focused on rendering a list of elements, which is a standard method for evaluating user interface performance [8-9]. The dataset used to generate the list was constructed by creating 40 paragraphs of Lorem Ipsum text, which were then repeated circularly. This approach allowed the dynamic creation of datasets of varying sizes. Two approaches were utilized: a dedicated component, LazyColumn, and a less specialized component, Column. The LazyColumn creates a vertically scrollable list that can contain large number of elements, as it only composes the elements needed at a given moment. On the other hand, the Column creates a vertical list and composes and lays out all elements regardless of whether they are visible. In addition, Column requires the verticalScroll modifier to enable vertical scrolling. Performance was evaluated by measuring the startup time to full display, as well as the maximum memory usage. Both metrics were averaged, with lower startup time and lower memory usage indicating better component performance. The components were examined under lower and higher workloads, specifically for list containing 10, 100, and 1000 elements. Each configuration was tested across three application startup modes:

1. Cold – the application process is not alive, and must be started in addition to Activity creation.
2. Warm – create and display a new Activity in a currently running application process.
3. Hot – bring existing Activity to the foreground.

The second research scenario involved rendering three types of animations. The selection of dedicated animation components was based on the official Jetpack Compose guidelines [18]. In both implementations, dedicated and custom, the animations were visually identical and executed over the same duration, and performed in both forward and reverse directions. In the first animation type, the parent container gradually changed its size to adapt to the expanding or shrinking child content. The child content initially consisted of two vertically arranged text components that expanded to six and then shrink back to two. The dedicated component for this animation was animateContentSize, while the custom implementation used layout modifier and a low-level Animatable object. The second animation type used the same graphical interface as in the first and involved toggling the visibility of UI elements. This was achieved using the AnimatedVisibility component in the dedicated implementation, while the custom implementation used an Animatable object and dynamically controlled the element's opacity via the graphicsLayer modifier. Once the element became fully transparent, it was removed from the composition. The third and final animation type implemented a crossfade transition between two screens, using the Crossfade component in the dedicated implementation. Similar to the previous custom implementations, Animatable objects were used to adjust the opacity of overlapping screen contents. The initial screen retained the layout from the first animation type, while the second screen displayed two images vertically. For all animation types, the render thread execution time was measured for each frame, and the results were averaged. As the metric is subject to minimization, lower thread execution times (rendering times) indicate better component performance. For reference throughout this study, the animations are abbreviated according to the sequence of types as follows: A1, A2, A3.

The third research scenario involved executing tasks related to input and output operations on a text file and a database, based on which the screen content was rendered. The dataset [19] used in this scenario was the first result returned by Kaggle when searching for the term "Books". It contains information about books stored in a CSV file. The following tasks were executed:

- Importing data from the CSV file into the database.
- Retrieving all records.
- Retrieving all records and sorting by title.
- Retrieving all records and sorting by ranking.
- Retrieving the first record.
- Deleting data from the database.

Only the first task was performed on the file, while all subsequent tasks were executed on the database. The sorting operations were carried out within a Kotlin function. Each task was considered complete once the corresponding action had been performed and the screen rendered based on the returned data. A dedicated IO dispatcher was utilized for the input and output operations, along with the Default dispatcher and a custom dispatcher configured with the highest thread priority, which may influence the software's responsiveness [14]. The execution time of each task was measured and averaged. As the measured metric is subject to minimization, a shorter execution time indicates better performance of the dispatcher.

During the testing phase of the experiment, issues were observed on devices S2 and S3 in obtaining valid results for each iteration, despite correct operation on device S1 and the emulator. Specifically, in the scenario comparing list rendering performance, certain iterations on S2 and S3 failed to record startup time in the warm and hot startup modes. To address this issue, the number of iterations for these modes was increased on the affected devices, and the first ten iterations containing valid measurement data were selected for analysis. A similar issue occurred on the same devices during the measurement of animation frame rendering time. Upon inspection of the system trace, it was found that the RenderThread was not consistently registered, resulting in the absence of valid results for some iterations. As with the previous

issue related to list rendering, the number of iterations was increased on these devices, and the first ten iterations containing valid measurement data were selected for analysis.

4. Results

This chapter presents the results obtained from the experiments conducted on three mobile devices for each of the defined research scenarios.

For the first research scenario, measurements were collected for startup time and memory usage (Tables 2-4). On devices S2 and S3, some iterations lacked valid measurements for the warm and hot startup modes. As a result, additional iterations were performed, and only the first ten iterations containing valid data were included in the analysis.

For the smallest tested list size of 10 elements, the average startup time and memory usage were comparable between the LazyColumn and Column implementations across all startup modes and devices. Under these conditions, each device demonstrated slightly different behavior, with either implementation yielding better performance depending on the device. The maximum observed differences in startup time and memory usage were 5.6% and 27.0%, respectively. The high 27.0% memory difference was due to two outliers. Despite this, statistical testing showed no significant differences in most cases. The only statistically significant differences were found for cold startup memory usage on devices S2 and S3, where the differences of 1.5% and 1.4% were observed. On S2, LazyColumn performed better, whereas on S3, Column performed better. A key advantage of LazyColumn is its stability, despite the increasing size of the list, both startup time and memory usage remained relatively consistent. In contrast, the performance of Column showed a significant decrease as the list size increased, resulting in higher values for both metrics compared to LazyColumn. This trend was observed consistently across all devices for the larger list sizes of 100 and 1000 elements and was statistically confirmed by significant differences in group means for both metrics.

It is also worth noting the increase in memory usage observed in the warm startup mode. In some cases, memory consumption in this mode was higher than in both cold and hot startups, while memory usage in hot startup was consistently the lowest. This pattern was observed across all tested devices. A possible explanation is that, in warm startup, the application retained a partially loaded state from the previous launch, which had not yet been fully released when a new state was created during the activity restart.

Table 2: Startup time and memory usage results during list application startup on smartphone S1

Implementation	Startup mode	Element count	Startup time ± std (ms)	Memory usage ± std (MB)
LazyColumn	Cold	10	557.7 ± 27.4	7.5 ± 0.1
LazyColumn	Cold	100	541.3 ± 20.3	7.4 ± 0.0
LazyColumn	Cold	1000	546.0 ± 14.6	7.5 ± 0.0
Column	Cold	10	555.5 ± 33.9	7.5 ± 0.1
Column	Cold	100	679.4 ± 20.4	22.9 ± 0.0
Column	Cold	1000	1,715.8 ± 38.0	242.8 ± 5.0
LazyColumn	Warm	10	153.2 ± 14.3	11.3 ± 2.0
LazyColumn	Warm	100	156.3 ± 17.8	11.3 ± 2.0
LazyColumn	Warm	1000	151.2 ± 11.9	11.2 ± 2.3
Column	Warm	10	146.2 ± 10.3	11.2 ± 2.1
Column	Warm	100	292.6 ± 16.9	37.0 ± 9.8
Column	Warm	1000	1,396.3 ± 16.3	135.0 ± 2.2
LazyColumn	Hot	10	60.2 ± 7.4	5.4 ± 1.4
LazyColumn	Hot	100	60.3 ± 7.6	5.6 ± 1.7
LazyColumn	Hot	1000	61.2 ± 8.6	5.7 ± 1.7
Column	Hot	10	62.0 ± 7.9	5.5 ± 1.8
Column	Hot	100	85.4 ± 5.8	11.5 ± 6.4
Column	Hot	1000	340.2 ± 7.5	47.1 ± 2.5

Table 3: Startup time and memory usage results during list application startup on smartphone S2

Implementation	Startup mode	Element count	Startup time ± std (ms)	Memory usage ± std (MB)
LazyColumn	Cold	10	453.7 ± 16.7	6.3 ± 0.1
LazyColumn	Cold	100	479.9 ± 21.5	6.3 ± 0.0
LazyColumn	Cold	1000	486.0 ± 20.0	6.4 ± 0.0
Column	Cold	10	480.9 ± 44.6	6.4 ± 0.0
Column	Cold	100	641.8 ± 86.4	18.8 ± 0.0
Column	Cold	1000	1,978.6 ± 76.3	191.3 ± 10.0
LazyColumn	Warm	10	190.6 ± 21.1	8.8 ± 2.0
LazyColumn	Warm	100	190.1 ± 10.1	7.5 ± 1.7
LazyColumn	Warm	1000	199.1 ± 12.3	7.5 ± 1.5
Column	Warm	10	187.1 ± 11.4	7.5 ± 1.0
Column	Warm	100	497.4 ± 54.0	27.9 ± 6.7
Column	Warm	1000	2,292.0 ± 283.9	204.7 ± 1.2
LazyColumn	Hot	10	63.7 ± 4.6	5.2 ± 0.8
LazyColumn	Hot	100	63.1 ± 5.4	4.6 ± 0.7
LazyColumn	Hot	1000	65.3 ± 5.9	5.0 ± 0.9
Column	Hot	10	64.8 ± 8.1	5.2 ± 1.2
Column	Hot	100	106.5 ± 3.3	10.8 ± 2.9
Column	Hot	1000	471.0 ± 7.2	94.1 ± 6.8

Table 4: Startup time and memory usage results during list application startup on smartphone S3

Implementation	Startup mode	Element count	Startup time ± std (ms)	Memory usage ± std (MB)
LazyColumn	Cold	10	332.8 ± 22.2	10.9 ± 0.0
LazyColumn	Cold	100	331.5 ± 8.5	10.9 ± 0.0
LazyColumn	Cold	1000	333.9 ± 15.6	11.0 ± 0.0
Column	Cold	10	319.7 ± 9.5	10.8 ± 0.0
Column	Cold	100	415.7 ± 15.3	26.0 ± 0.0
Column	Cold	1000	1,181.0 ± 14.6	214.8 ± 0.1
LazyColumn	Warm	10	114.5 ± 5.9	19.2 ± 4.4
LazyColumn	Warm	100	124.5 ± 14.5	20.3 ± 8.8
LazyColumn	Warm	1000	130.5 ± 6.7	22.6 ± 6.1
Column	Warm	10	119.3 ± 8.3	21.7 ± 7.4
Column	Warm	100	210.6 ± 12.4	63.2 ± 21.3
Column	Warm	1000	1,064.8 ± 57.4	260.7 ± 41.2
LazyColumn	Hot	10	61.9 ± 7.9	6.5 ± 1.7
LazyColumn	Hot	100	67.7 ± 7.2	7.7 ± 4.9
LazyColumn	Hot	1000	63.4 ± 9.7	8.6 ± 3.6
Column	Hot	10	65.2 ± 13.5	8.9 ± 4.3
Column	Hot	100	85.8 ± 8.0	14.8 ± 9.2
Column	Hot	1000	295.6 ± 5.0	51.1 ± 4.8

The results from the second research scenario (Tables 5-8), which compared the performance of three types of animations, reveal notable differences in frame rendering times between the dedicated and custom implementations. On devices S2 and S3, some iterations did not produce valid results due to the render thread not being properly registered in the system trace. As a result, animations were executed more than ten times on these

devices, and the first ten iterations that contained valid frame rendering times were selected for further analysis.

Depending on the tested device and animation type, both the average frame rendering times and standard deviations varied. The only case in which the custom implementation achieved a lower mean frame rendering time than the dedicated implementation was for animation A1 on device S1, where the difference was 0.03 ms (0.6%).

Table 5: Animation frame rendering time results

Animation	Smartphone	Dedicated implementation ± std (ms)	Custom implementation ± std (ms)
A1	S1	5.53 ± 0.11	5.49 ± 0.07
	S2	4.99 ± 0.04	5.03 ± 0.08
	S3	2.89 ± 0.09	4.10 ± 0.71
A2	S1	5.29 ± 0.12	5.39 ± 0.08
	S2	4.68 ± 0.06	4.76 ± 0.05
	S3	3.05 ± 0.40	3.67 ± 0.64
A3	S1	5.86 ± 1.17	6.98 ± 1.84
	S2	5.62 ± 2.07	5.75 ± 0.94
	S3	3.56 ± 0.86	3.73 ± 0.87

Table 6: Detailed frame rendering time results for animation A1

	Dedicated implementation (ms)	Custom implementation (ms)
P50	4.79	4.99
P90	5.83	6.12
P95	6.07	8.97
P99	7.01	14.61
Average	4.33	4.82
Std	1.58	2.39
Min	1.38	1.44
Max	20.49	24.58

Table 7: Detailed frame rendering time results for animation A2

	Dedicated implementation (ms)	Custom implementation (ms)
P50	4.63	4.80
P90	5.52	5.91
P95	5.92	7.99
P99	9.32	11.36
Average	4.23	4.54
Std	1.63	1.97
Min	1.44	1.48
Max	19.71	20.03

Table 8: Detailed frame rendering time results for animation A3

	Dedicated implementation (ms)	Custom implementation (ms)
P50	4.77	4.88
P90	6.82	10.77
P95	10.25	13.42
P99	15.08	15.60
Average	4.80	5.33
Std	3.02	3.98
Min	1.42	1.55
Max	72.98	166.75

However, statistical analysis indicated that this difference was not significant. A similar situation occurred also with animation A1 on device S2, where the difference was also 0.03 ms (0.6%) and likewise not statistically significant. In all other animations and across all tested devices, the statistical tests confirmed significant differences in average frame times in favor of the dedicated implementations, with differences ranging from 1.9% to 41.6%. In the aggregated results for each animation type, half of the animation frames were rendered faster using the dedicated components. Additionally, the dedicated implementations achieved lower average frame times, lower standard deviations, as well as lower minimum and maximum values. The highest standard deviation was observed for animation A3, which involved transitioning between two different screens using a crossfade effect.

The results obtained from the final research scenario (Table 9) present the average execution times of individual tasks along with their corresponding standard deviations. Analysis of the results indicates that the performance of all three tested dispatchers was comparable. The shortest average total task execution time on each of the tested mobile devices was achieved by the implementation using the custom dispatcher configured with the highest thread priority. However, statistical tests did not reveal any significant differences between the dispatchers in the average total task execution time. It is noteworthy that the custom dispatcher exhibited lower standard deviations compared to the other two dispatchers. Despite achieving the shortest total execution time, the custom dispatcher did not consistently outperform the others in every task. Among the six analyzed tasks, it achieved the fastest execution time in three cases on device S1, in four on device S2, and in two on device S3.

Table 9: Task execution time results using different dispatchers

Task	Smartphone	IO Average ± std (ms)	Default Average ± std (ms)	Custom Average ± std (ms)
Importing data	S1	1,376 ± 157	1,378 ± 115	1,354 ± 52
	S2	1,542 ± 62	1,557 ± 86	1,508 ± 45
	S3	1,377 ± 76	1,355 ± 53	1,356 ± 43
Retrieving all records	S1	219 ± 16	211 ± 13	206 ± 16
	S2	226 ± 10	258 ± 119	214 ± 16
	S3	184 ± 22	177 ± 10	176 ± 16
Retrieving all records and sorting by title	S1	199 ± 6	193 ± 5	195 ± 7
	S2	222 ± 9	221 ± 11	214 ± 17
	S3	182 ± 10	181 ± 12	170 ± 9
Retrieving all records and sorting by ranking	S1	196 ± 6	193 ± 7	192 ± 5
	S2	229 ± 13	232 ± 18	220 ± 10
	S3	174 ± 12	172 ± 8	174 ± 7
Retrieving the first record	S1	72 ± 9	68 ± 8	69 ± 8
	S2	78 ± 4	91 ± 46	87 ± 33
	S3	52 ± 7	51 ± 8	55 ± 10
Deleting data	S1	112 ± 55	80 ± 9	87 ± 34
	S2	125 ± 72	160 ± 74	136 ± 75
	S3	71 ± 49	112 ± 51	77 ± 53
Total time	S1	2,173 ± 148	2,124 ± 115	2,104 ± 49
	S2	2,422 ± 131	2,518 ± 171	2,379 ± 102
	S3	2,039 ± 90	2,048 ± 73	2,008 ± 75

At the same time, there were no cases in which the custom dispatcher was the slowest on devices S1 and S2. However, on device S3, there were two tasks in which it recorded the longest execution time among the compared dispatchers.

5. Discussion

The experimental study was successfully conducted in its entirety, and all test scenarios were implemented as planned. A complete set of performance data was

collected across three different mobile devices, allowing for comprehensive analysis.

In the first research scenario, for the smallest tested dataset of 10 elements, the average values were comparable between the LazyColumn and Column implementations, regardless of the startup mode. Statistical tests revealed no notable differences, except for memory usage in the cold startup mode on devices S2 and S3, where LazyColumn used less memory on S2, and Column was more efficient on S3. These experimental results do not allow for a definitive conclusion about which implementation is more efficient for small lists. As the number of list elements increased, a significant decline in performance was observed for the Column implementation in terms of both startup time and memory usage, while the performance for LazyColumn remained considerably stable. This observation supports the assumption that LazyColumn composes and lays out only the currently needed elements, in contrast to Column, which composes and lays out all components regardless of their visibility. According to the statistical test results, the differences in mean values between these implementations for larger datasets were statistically significant, demonstrating that LazyColumn is a more efficient and scalable solution than Column. Based on these findings, hypothesis H1, that the dedicated LazyColumn component is more efficient than the Column component regardless of data size, was rejected. Although significant differences were observed for larger datasets, both implementations demonstrated similar performance for small datasets containing 10 elements. Therefore, the use of LazyColumn is recommended for large or unknown dataset sizes, while for small datasets, both implementations exhibit similar performance.

In the second research scenario, the dedicated animation implementations achieved shorter average frame rendering times in all tested cases, except for animation A1 on device S1, where the difference between implementations was not statistically significant. Similarly, no statistically significant difference in favor of the dedicated implementation was found for animation A1 on device S2. In all other combinations of devices and animation types, the average frame rendering times were significantly lower for the dedicated implementations. Additionally, in half of the measured frames, rendering was faster with dedicated animations than with custom ones. Based on these observations, hypothesis H2, stating that dedicated animations are more efficient than non-specialized animations, was accepted, as dedicated animations proved to be equally or more efficient than their custom counterparts. Given their higher performance and availability as built-in, tested components, it is recommended to use dedicated animations. This not only enhances performance but also reduces developer workload and minimizes the risk of introducing bugs into the code.

In the third research scenario, it was observed that the configuration using a custom dispatcher with the highest thread priority achieved the shortest average total task execution time across all tested devices. However, statistical tests did not indicate any significant differences in the average execution times among the analyzed dispatchers. The absence of a clear performance advantage for the custom dispatcher may be attributed to the characteristics of the tested application, which did not perform a high number of concurrent operations across multiple threads. During task execution, the main thread remained idle, awaiting the results from the dispatcher. Furthermore, the application operated in the foreground, which led the operating system to schedule approximately 95% of CPU time [20] to this application, regardless of thread priority. As a result, the dispatcher did not need to compete with other threads for CPU time, which limited the potential performance benefit of using a custom dispatcher with highest thread priority. Given the lack of evidence supporting a performance advantage of the dedicated IO dispatcher, hypothesis H3, stating that a dedicated dispatcher is more efficient than other dispatchers, must be rejected.

6. Conclusions

In summary, the results obtained across all research scenarios indicate that dedicated components in Jetpack Compose do not always provide better performance than their less specialized counterparts. While dedicated components were more performant in the majority of performance tests, there were cases in which non-specialized components delivered comparable or even better results. Consequently, the research thesis stating that specialized components in Jetpack Compose offer better performance compared to more general purpose, non-specialized components must be partially rejected.

Nevertheless, the use of specialized components is recommended, particularly in applications where performance is not a critical concern and the characteristics of the data, such as size, are unknown. Dedicated components demonstrated greater stability and robustness with respect to varying data inputs, and in some scenarios, significant performance advantages were observed. In contrast, the performance gains achieved through custom implementations were not consistently observed across all devices and did not differ significantly from those achieved using dedicated components.

Further research should consider expanding the experimental scope to include scenarios involving lists with image containing elements, the use of keys versus keyless elements, and interactive operations such as scrolling, item insertion, deletion, and sorting. In the context of animations, it would be valuable to explore additional animation types, evaluate whether varying frame rates impact component performance, and measure both the number of frames that exceeded their assigned rendering time and the extent of those overruns, as these factors directly influence animation smoothness. For dispatchers, testing under higher concurrency conditions involving multiple threads and parallel tasks would help assess whether increased thread priority affects performance under heavier workloads. Such investigations would contribute to a more comprehensive understanding of the behavior of Jetpack Compose components in real world mobile application use cases.

Literature

[1] Key statistics of smartphone users worldwide, https://prioridata.com/data/smartphone-stats/, [13.06.2025].

[2] Number of Android users worldwide, https://www.bankmycell.com/blog/how-many-android-users-are-there, [07.05.2025].

[3] V. N. Inukollu, D. D. Keshamoni, T. Kang, M. Inukollu, Factors Influencing Quality of Mobile Apps: Role of Mobile App Development Life Cycle, International Journal of Software Engineering & Applications 5(5) (2014) 15–34, https://doi.org/10.5121/ijsea.2014.5502.

[4] S. L. Lim, P. J. Bentley, N. Kanakam, F. Ishikawa, S. Honiden, Investigating Country Differences in Mobile App User Behavior and Challenges for Software Engineering, IEEE Transactions on Software Engineering 41(1) (2015) 40–64, https://doi.org/10.1109/TSE.2014.2360674.

[5] H. Khalid, E. Shihab, M. Nagappan, A. E. Hassan, What Do Mobile App Users Complain About?, IEEE Software 32(3) (2015) 70–77, https://doi.org/10.1109/MS.2014.50.

[6] S. Ickin, K. Petersen, J. Gonzalez-Huerta, Why Do Users Install and Delete Apps? A Survey Study, Proceedings of the 8th International Conference on Software Business (2017) 186–191, https://doi.org/10.1007/978-3-319-69191-6_13.

[7] What developers are saying about Jetpack Compose, https://developer.android.com/develop/ui/compose/adopt#what-developers-are-saying, [19.05.2025].

[8] J. Szczukin, Performance analysis of user interface implementation methods in mobile applications, Journal of Computer Sciences Institute 26 (2023) 13–17, https://doi.org/10.35784/jcsi.3070.

[9] M. Kusuma, A. H. Rifani, B. Sugiantoro, Comparison analysis of Jetpack Compose and Flutter in Android-based application development using Technical Domain, In 2023 Eighth International Conference on Informatics and Computing (ICIC) (2023) 1–5, https://doi.org/10.1109/icic60109.2023.10381987.

[10] B. P. D. Putranto, R. Saptoto, O. C. Jakaria, W. Andriyani, A Comparative Study of Java and Kotlin for Android Mobile Application Development, In 2020 3rd International Seminar on Research of Information Technology and Intelligent Systems (ISRITI) (2020) 383–388, https://doi.org/10.1109/isriti51436.2020.9315483.

[11] K. Wasilewski, W. Zabierowski, A Comparison of Java, Flutter and Kotlin/Native Technologies for Sensor Data-Driven Applications, Sensors 21(10) (2021) 3324, https://doi.org/10.3390/s21103324.

[12] M. Peters, G. L. Scoccia, I. Malavolta, How does Migrating to Kotlin Impact the Run-time Efficiency of Android Apps?, In 2021 In IEEE 21st International Working Conference on Source Code Analysis and Manipulation (SCAM) (2021) 36–46, https://doi.org/10.1109/SCAM52516.2021.00014.

[13] M. Hort, M. Kechagia, F. Sarro, M. Harman, A Survey of Performance Optimization for Mobile Applications, IEEE Transactions on Software Engineering 48(8) (2022) 2879–2904, https://doi.org/10.1109/TSE.2021.3071193.

[14] J. Fu, Y. Wang, Y. Zhou, X. Wang, How resource utilization influences UI responsiveness of Android software, Information and Software Technology 141 (2022) 106728, https://doi.org/10.1016/j.infsof.2021.106728.

[15] H. Lin, C. Liu, Z. Li, F. Qian, M. Li, P. Xiong, Y. Liu, Aging or Glitching? What Leads to Poor Android Responsiveness and What Can We Do About It?, IEEE Transactions on Mobile Computing 23(2) (2024) 1521–1533, https://doi.org/10.1109/TMC.2023.3237716.

[16] J. Callan, O. Krauss, J. Petke, F. Sarro, How do Android developers improve non-functional properties of software?, Empirical Software Engineering 27(5) (2022) 113, https://doi.org/10.1007/s10664-022-10137-2.

[17] Jetpack Compose performance overview, https://developer.android.com/develop/ui/compose/performance, [17.04.2025].

[18] Quick Jetpack Compose animation guide, https://developer.android.com/develop/ui/compose/animation/quick-guide, [17.04.2025].

[19] Goodreads books dataset from Kaggle, https://www.kaggle.com/datasets/jealousleopard/goodreadsbooks, [22.04.2025].

[20] Guide to better performance through threading, https://developer.android.com/topic/performance/threads#priority, [22.04.2025].