Bachelor Degree Project

UNIVERSITY
OF SKÖVDE

A Comparative Analysis of Jetpack Compose and XML Views

Bachelor Degree Project in Information Technology
Basic level 30 ECTS
Spring 2024

Leo Wahlandt, Anton Brännholm

Supervisor: Simon Butler
Examiner: Yacine Atif

Abstract

This bachelor thesis presents a comparative analysis between *XML Views* (XML+Kotlin) and *Jetpack Compose* (Kotlin), focusing on picture-based applications across varying demand levels: low, medium, and high. Through the development and evaluation of three distinct applications for each toolkit, we used *Macrobenchmark* and *Android Profiler* to collect their performance in terms of startup time, frame timing metrics, and hardware resource utilization, including CPU, battery, and memory usage.

Our findings reveal that *XML Views* maintain certain advantages over *Jetpack Compose*, particularly in terms of startup time and frame timing metrics. However, our analysis also highlights notable benefits of *Jetpack Compose*, particularly in memory usage. Despite the overall advantage of *XML Views, Jetpack Compose* demonstrates the potential for optimization in specific resource-intensive scenarios.

This research contributes to the ongoing discourse surrounding the selection of a UI toolkit for Android application development, providing insights into the performance characteristics of *XML Views* and *Jetpack Compose* in picture-based applications across varying demand levels. Our findings offer practical guidance for developers in making informed decisions regarding UI toolkit selection, considering both performance and resource utilization considerations.

keywords: Android, Kotlin, XML, Jetpack Compose, Macrobenchmark, Android Profiler

Contents

1 | Introduction

When developing an Android application, the developer has to make a crucial choice of which User Interface (UI) framework to use. The UI framework plays an important role in the efficiency, maintainability, and overall user experience of mobile applications. XML-based layouts have been the traditional way of Android UI design while offering developers an imperative approach to defining the UI components and their properties. However, with the arrival of *Jetpack Compose,* a modern UI toolkit introduced by Google, developers are presented with a paradigm shift towards a more concise and composable UI development model.

Jetpack Compose is promoted by Google as the recommended modern toolkit for building native UI with less code, powerful tools, and intuitive Kotlin APIs. Although, when considering a UI framework, the performance has a crucial impact on the mobile application. Due to *Jetpack Compose* being a relatively new UI framework (Google 2024*b*) means that there is a limited amount of data on the performance of *Jetpack Compose* compared to *XML Views* in regards to testing. The main test this thesis will do is the render time and startup time of the UI, as well as CPU usage, memory usage, and battery usage. Hence, the purpose of this thesis is to provide enough data in this area by conducting benchmark tests and comparing the performance between *Jetpack Compose* and *XML Views*.

This comparative analysis aims to explore and evaluate the key differences, advantages, and limitations of *Jetpack Compose* and *XML Views* in Android app development, particularly focusing on the visual aspect. The evaluation will consider performance factors such as UI rendering time, startup time, and CPU usage for these two UI approaches when developing native Android applications. Additionally, the thesis aims to identify the advantages and disadvantages that *Jetpack Compose* and *XML Views* may affect the application during the development of a native Android application. This is done by using three different demand applications in *Jetpack Compose* and three different demand applications in XML Views. This is to give a greater depth of data to measure which UI toolkit has better performance depending on the demand of the application.

2 | Background

As of the fourth quarter of 2023, the Android market share is up to 70.1% of the world which is exceeding 3.5 billion users worldwide (Twinr 2024). The mobile industry is constantly expanding users, propelled by the increasing sales of mobile devices and smartphones (Twinr 2024). Due to the vast number of users worldwide, even a minor delay in an application can lead to significant overall downtime. If each of the 3.5 billion users has to wait 5 seconds for the application to start, the collective downtime would be enormous, resulting in a significant loss of productivity and user satisfaction. For example, if a single user opens the application 10 times a day, they would spend 50 seconds waiting. Therefore, optimizing performance is essential to ensure a positive user experience and maintain efficiency on a large scale.

When a user experiences an application with issues when the application suffers from slow UI rendering, skipping frames and the user perceives a recurring flicker on the screen, also known as *jank* (Google 2023)[1]. When animation *jank*, frozen frames, and high memory usage occur, it negatively impacts the user experience which could lead to lower rating or even app deletion (Yang 2022). Therefore it is very important to have a smooth and well-performing application so the user will keep using your application.

The performance of an app depends on many factors during the development. One of the factors depends on which toolkit being used during the development of the application. Before *Jetpack Compose* was pushed by Google to be the new way of development, *XML Views* was a traditional, widely recommended approach for UI design in native Android. However, when Google released the *Jetpack Compose* in 2021, which is the new toolkit for UI design in native Android (Google 2024*j*)[2], the work of UI design has undergone a significant change. Google's objective for developing a new toolkit for designing new UI design methods, is to simplify and accelerate UI development for Android. This is done by reducing code complexity with declarative API and powerful tools Google (2024*j*). *Jetpack Compose* is presented as an alternative to optimize the UI performance by leveraging the capabilities of the hardware, while also enhancing the consistency across multiple platforms through adaptive UI design that adapts to different devices and screen dimensions.

[1] UI jank detection
[2] Why compose

2.1 Android

Android is a comprehensive open-source platform designed for mobile devices (Google 2024*c*). Android is an open-source platform which means that the entire stack, from low-level Linux modules to native libraries, and from the application framework to complete applications, is open to the public.

2.2 Java

The most popular program languages to develop Android applications are Java and Kotlin (Putranto et al. 2020). Java was first developed in 1991 and is an object-oriented programming language. Java tries to apply to objects that exist in the real world and has good flexibility as one of the programming languages because it is multiplatform. This means that it can run on various platforms, such as Linux, Windows, and various mobile devices. Java differs from other platforms due to the Java software platform running on top of a hardware-based platform. Other platforms are usually a combination of hardware and operating systems. Java's platform has two components and they are the following:

1. Java Virtual Machine (JVM)
2. Java Application Programming Interface (Java API)

Java programs run on the JVM which isolates Java programs from any underlying hardware.

2.3 Kotlin

Kotlin is a statically typed programming language that targets Java Virtual Machine (JVM), Android, JavaScript, and native code. The Kotlin project was developed by JetBrains and started in 2010 (Putranto et al. 2020). As of February 2016, Kotlin 1.0 was released and became an open-source programming language. Kotlin utilizes all of the advantages of a modern language to the Android platform (Putranto et al. 2020). The objective of Kotlin is to be a modern language without introducing any new restrictions like compatibility, performance, interoperability, footprint, compilation time, and learning curve. Applications written in Kotlin are still runnable in JVM due to Kotlin being compiled to Java bytecode and Kotlin is fully interoperable with Java.

2.4 XML

Extensible Markup Language (XML) is used to define and store data in a shareable manner. XML supports information exchanges between computer systems such as websites, databases, and third-party applications like Android. By using Android's XML schemas, the developer can quickly design UI layouts with the corresponding screen elements. XML works the same way as creating web pages in HyperText-Markup-Language (HTML) with a series of nested elements. (Android 2011)

2.4.1 Imperative Paradigm

XML Views uses a procedural model that explicitly specifies a process to be executed which implies that *XML Views* uses an imperative UI paradigm. The imperative process models define explicitly all activities that have to be executed, their execution order, and the data flow between these activities (Endres et al. 2017). This means that *XML Views* components inside are subsequently rendered for the user. *XML Views* can also work with code bases that are either Java or Kotlin, while *Jetpack Compose* is only for Kotlin. The imperative model explicitly defines all activities that have to be executed, their execution order, and the data flow between these activities. Thus imperative process models are executed in an automated manner by the process engine. The way XML works with Android is that the developer writes the UI in an imperative manner. This means that the developer is describing the application by writing the UI from the top to bottom of the application. For instance, if the developer wants a top bar, the developer has to create a top bar with items inside it from left to right, so a title, then a menu. To update the values in the application the developer has to walk the tree to find where to update the current value of the applications. Manipulating the value manually increases the likelihood of errors (Endres et al. 2017). In an imperative object-oriented UI toolkit in Android, the developer has to instantiate a tree of widgets. This is done by inflating an *XML Views* layout file, where each widget maintains its own internal state, and exposes getter and setter methods that allow the app logic to interact with the widget.

2.5 Jetpack Compose

Jetpack Compose is Android's modern toolkit for building native UI. *Jetpack Compose* simplifies and accelerates UI development on Android with powerful tools and intuitive Kotlin APIs. *Jetpack Compose* uses a declarative API which means that the developer only needs to describe the UI and Compose takes care of the rest. With Compose, you build small, stateless components that are not tied to a specific activity or fragment. Therefore it is easy to reuse and test the components. In the declarative approach when developing Android applications, widgets are relatively stateless and do not expose setter or getter functions. Widgets are not exposed as objects. (Google 2024*j*)

2.5.1 Declarative paradigm

The main difference between *Jetpack Compose* and *XML Views* lies in the UI design paradigm (Javatpoint 2024). *Jetpack Compose* is a declarative UI paradigm for creating user interfaces. The declarative approach uses structural models that describe the desired application structure and state. Declarative programming does not offer instructions on how the problem should be solved. Instead offers a constant to check and ensure the problem is solved correctly. In simpler words, the declarative approach gives the address of how to arrive at a given destination without a thought about how it was found. A declarative approach may automate repetitive flow. Thus effective code is one major advantage of declarative programming that can be applied with the help of using a high level of abstraction, easy extension, ways, and more.

2.5.2 Recomposition

In the traditional way to update/change a widget in *XML Views* Views, the developer has to call a setter on the widget to change its internal state. With *Jetpack Compose*, to update/change the composable function, the function is called again with the new data. The major disadvantages of recomposing the entire UI can be computationally expensive if it is done incorrectly, which uses computing power which leads to more usage of the battery. Conversely, Compose solves this problem with its intelligent recomposition (Google 2024*i*). Intelligent recomposition means that *Jetpack Compose* recompositions the targeted composable function or lambdas that might have updated/changed with the new input and skips the rest. Therefore *Jetpack Compose* does not need to recomposite every function and saves computing power and battery life.

2.6 Android Performance Testing

Android performance testing's main goal is testing the functionality, usability, and compatibility of apps running on Android devices. As Kong et al. (2019) writes, an error-prone application can significantly impact the user experience which may lead to a downgrade of the application rating. An application with a low rating can lead to fewer downloads and harm the reputation of the developer and their organization. Therefore it is important to decrease the number of errors and misbehavior of the application. Thus becoming increasingly important to ensure that Android applications are sufficiently tested before release on the market. Google (2024*d*) writes that testing user interactions helps ensure users do not encounter any unexpected results, weak performance, or a poor experience when interacting with the application, therefore Google advises the developer to test their UI. One approach to testing the application is to use a human tester to perform a set of user operations on the application and verify that it returns the expected results. The disadvantage with human testers is that the approach can be time-consuming and error-prone. Therefore a more efficient approach is to write UI tests such that user actions are performed automatically. This approach allows the developer to run the tests quickly and reliably in a repeatable manner. (Google 2024*d*)

2.6.1 Testing Framework

Testing an application and measuring the performance of the application, the application needs to execute actions reliably, to ensure there is no variance between the tests. According to Jha et al. (2019), some major challenges faced by Android developers in testing their apps are time constraints, compatibility issues, lack of exposure, complexity of the tools, and lack of experience. It is ambiguous to design and write effective test cases. A test case is a specification of the inputs, execution conditions, and expected results. Therefore it takes a substantial amount of time and effort to design and write effective test cases that can detect defects in software under test. As stated by Jha et al. (2019) most app developers want to release their apps as soon as possible before someone else develops a similar application. Therefore, app developers may not have enough time to invest in designing and writing effective test cases.

As stated in Kong et al. (2019) study review, it is important to take into consideration that it is challenging to generate a perfect coverage of test cases, in order to find faults and errors in an Android application. Their study review also confirms that GUI (Graphical User Interface) testing is of importance in modern software development for guaranteeing a satisfactory user experience. Therefore most test approaches target the GUI or event mechanism.

Testing an Android application can be done using different tools to measure animation *jank*, frozen frames, and high memory usage when running on a device. Kong et al. (2019) confirms the importance of testing the GUI in modern software development to guarantee a pleasant user experience. Macrobenchmark is used to make simulations repeatable. Macrobenchmark is a library for testing larger use cases of the application, including application startup and complex UI manipulations, such as user interactions, like scrolling or running animations on the targeted device.

2.6.2 Startup States

Testing an Android application, Macrobenchmark has three types of startup modes [/launch-time]. The startup types are the following, *Cold*, *Warm*, and *Hot*. The purpose of the different startup modes is how the tests perform during the testing.

Cold means that the application starts from scratch and the application process is not alive and must be started in addition to activity creation (Google 2024*a*). To start an application *Cold*, the device system has terminated the application process or launched the application for the first time since the device booted. The stages of beginning a *Cold* start, the system has to first load and launch the application, display a blank starting video for the application, and then create the app process. The app process then creates the application by calling the onCreate() method. The onCreate() method often has the greatest impact on load time, due to it performing the work with the highest overhead: loading and inflating views and initializing the objects that are needed for the activity to run. *Cold* starts to present the greatest challenge to minimizing startup time, due to the system and app having more work to do than the other launch states.

Therefore when starting a test running with *Warm*, the Android system creates and displays a new Activity in a currently running app process. This means that the *warm* startup skips the first 3 steps of starting the test as in *Cold* startup mode. This simulates the user back out of the application and then re-launches it and the process might continue to run, but the app must recreate the activity from scratch by calling the onCreate() function.

Hot means that the Android system brings existing Activity to the foreground, process, and Activity still exists from the previous launch as if the application is already running. Hence, the application activities are still resident in the memory, which makes it possible for the application to avoid repeating initialization, layout inflating, and rendering.

2.7 Previous work

When analyzing previous work in the area of UI performance testing in *Jetpack Compose*, it is important to take into account that *Jetpack Compose* is a relatively new UI toolkit (Google 2024*b*). Hence the result may be very different from the latest *Jetpack Compose* to the older versions. The result of this is a reason why the subject of performance testing has relatively little previous work to source data from.

Since *Jetpack Compose* was released there have been a few research papers comparing *Jetpack Compose* and *XML Views*. We analyzed three previous studies to get a better understanding of *Jetpack Compose* performance compared to *XML Views*.

Noori & Eriksson (2023) compared the performance of one application developed in *Jetpack Compose* and another one developed with *XML Views*. The logic of both applications are the same and also similar in visuals and feels but one had UI written in *XML Views* and the other was written in *Jetpack Compose*. These applications were tested on 2 different devices using different presets of the Macrobenchmark testing framework which emulates different states that the smartphone could be in during real-world usage. The applications consist of a drink catalog app, in which the user will be able to view a list of different alcoholic beverages. The application covers many essential UI elements that the user typically interacts with, including scrolling lists, buttons, text fields, drop-down lists, and navigation. The median startup time for *Jetpack Compose* while the device was COLD had 34.9% faster startup than *XML Views*. The version of *Jetpack Compose* also showed that it was 93% slower while scrolling compared to *XML Views*. As Noori & Eriksson (2023) write, the *Jetpack Compose* was faster on startup than *XML Views* on both devices, and during the scroll test, *XML Views* rendered faster than *Jetpack Compose* but *jankines* was being similar.

Fjodorovs & Kodors (2021) studied the UI rendering performance of the new Google Android user interface "*Jetpack Compose*" compared to *XML Views*. The author developed two applications with identical interfaces, one used the classic approach of Kotlin+XML layout file, and the other application was developed with *Jetpack Compose*. The UI development with Kotlin+XML represents the coupling paradigm. The coupling paradigm means that the code changes in one module require making changes to another file. The *Jetpack Compose* application represents cohesion because the application is only written in one programming language, Kotlin. The approach of the experiment to measure the rendering performance is done by starting a timer before the application renders the UI and when the application is rendered, the timer stops. The result of both XML+Kotlin and *Jetpack Compose*, which is shown in the paper, *Jetpack Compose* approximately 46% rendering time increase compared to XML+Kotlin. Fjodorovs & Kodors (2021) also mentions in the paper that *Jetpack Compose* should be investigated further when *Jetpack Compose* has a more stable release.

Milla & Radonjić (2023) investigated the development of native Android applications comparing XML+Kotlin and *Jetpack Compose*. The aim of the research was to analyze the benefits and limitations of *XML Views* and *Jetpack Compose*. To compare between *XML Views* and *Jetpack Compose*, the authors developed the same applications for *XML Views* and *Jetpack Compose*. The analysis of the code used in each technology provides an insight into how to uniquely solve the same problem. The applications used the same architecture, the MVVM (Model, View, and ViewModel). The comparison between *XML Views* and *Jetpack Compose* is that *XML Views* can be more complex and time-consuming compared to *Jetpack Compose*. When this paper was written (2023), *XML Views* was the recommended approach to create and develop an application in Android. In 2024, Google recommends *Jetpack Compose* over *XML Views*.

The largest gap in research lies in how most comparisons for the toolkits have been done. There are only a few papers on comparison between *XML Views* and *Jetpack Compose*, furthermore, most of these papers provide insufficient data, where both are compared on single application usage. We plan to instead give both toolkits the opportunity to provide greater depth of their respective strengths for varying demands of visual aspects.

2.8 Factors Affecting UI Performance

Deconstructing these factors there are multiple measurable aspects. These include rendering time, memory, CPU, and battery usage. All of these are important both for developers and for user experience down the line.

2.8.1 Rendering

UI rendering is the act of an application to generate a new frame of the application and display it on the mobile device. To keep an application smooth when a user interacts with the application, the application has to render the frames in under 16ms to achieve 60 frames per second (fps) (Google 2024*h*). If the application overruns the 16ms window, then it does not mean that the application is displayed 1ms late, it means that the *Choreographer* drops the frame entirely. As a result, the application if it overruns the 16ms window, the application will suffer slow UI rendering, and the application is forced to skip frames. When the application is skipping the frames, the user perceives stuttering, also known as *Jank*.

Another factor that decreases the performance of the application is frozen frames. Frozen frames occur when the application takes longer than 700ms to render (Google 2024*h*)[1]. Frozen frames often happen when the application appears to be unresponsive to user inputs for almost one second while the frame is rendering. For a smooth usage of the applications, the rendering should be in a 16ms window to ensure a smooth UI. However when starting the application or while transitioning to a different screen, it is common for the initial frame to take longer than 16ms to render. This is due to the application needing to inflate views, lay out the screen, and perform the initial draw all from scratch. Therefore frozen frames are not the same as *jank* and do not count as slow rendering. Despite this, the application should not take longer than 700ms to render.

When the user notices slow frames and frozen frames take place differently. Users commonly notice slow frames when scrolling in a RecyclerView and during complex animations not animating properly. Frozen frames on the other hand, often occur during app startup and during movement from one screen to another, which happens when the user navigates between different activities. (Google 2024*h*)

2.8.2 Memory

Memory usage affects a phone's longevity due to the nature of write and erase cycles, however, other hardware components often decay before that. The largest impact on application performance is excessive memory usage, such as memory leaks. By avoiding high memory usage an application will yield better performance. Applications can also suffer from out-of-memory conditions, which can lead to application pauses where the pause continues until memory can be allocated (Santhanakrishnan et al. 2016).

[1] Slow Render

2.8.3 CPU

CPU usage affects application performance where using too much may affect performance where more processing power may be needed. By using less CPU, performance may be enhanced since less data has to be processed, yielding better performance. CPU also has a trade-off relationship with its longevity and performance, where better performance increases its power usage (Kumakura et al. 2022).

2.8.4 Battery

Battery usage may not have a huge impact on application performance directly, however, more battery usage may impact a phone's longevity. Being able to have a functional phone for a longer time is good both economically and ecologically. *"Poorly written apps can sap 30 to 40 percent of a phone's juice"* Jha (2011), which supports the conclusion that better performance in applications can save battery longevity.

3 | Aim/Problem

Since the introduction of *Jetpack Compose*, it has been marketed or viewed as a new approach to Android application development. Opinions are subjective for the workflow approach Milla & Radonjić (2023), however, the actual performance aspect remains somewhat unexplored. The purpose of this thesis is to gather broader knowledge about the quantitative aspects. Getting more perspective of the new approach can be important for developers to get more depth about the advantages and shortcomings of *Jetpack Compose* and *XML Views*. The aim is not to describe the development process but to describe the actual output that this approach can provide.

3.1 Research Questions

- RQ1: To what extent does the choice of UI toolkit affect application performance?
- Hypothesis: UI toolkit has no significant effect on application performance.
  - $H1_0$: Toolkit has no significant effect on startup time.
  - $H2_0$: Toolkit has no significant effect on rendering.
- RQ2: To what extent does the choice of UI toolkit affect hardware demand?
- Hypothesis: UI toolkit has no significant effect on hardware demand.
  - $H3_0$: Toolkit has no significant effect on CPU usage.
  - $H4_0$: Toolkit has no significant effect on Memory usage.
  - $H5_0$: Toolkit has no significant effect on Battery usage.

3.2 Research Gap

As *Jetpack Compose* 1.0 was released on 28 July 2021, the research of this new and evolving UI toolkit is limited (Google 2024*b*). *Jetpack Compose* is relatively new and not much research has been done on its quantitative properties, this opened up a window to gather more knowledge and investigate further.

3.3 Developer Impact

When developing Android applications an emulator is often used to test the application and see their performance in real-time. When developing these emulators include limitations since they are not real hardware. By determining whether a toolkit runs faster during these stages, we may provide means to improve productivity for developers and companies.

3.4 User Experience Impact

When a user interacts with an application, a poorly performing application affects the user's opinions and may even end up in deletion. By developing a greater depth into understanding either toolkits performance, it can yield a better experience for the end user.

4 | Methodology

The method used for this project is a quasi-experiment, where we as researchers try our best to control variables to ensure a fair test for both toolkits. When using a quasi-experiment we have to ensure an even playing field for both toolkits to be able to determine strengths and weaknesses. This method is chosen due to the issues of randomness that are out of our control. These issues need to be avoided to ensure validity and that we as researchers have done our work to our best effort.

4.1 Dependent variables

To measure performance we have decided to use rendering time, battery usage, CPU usage, and memory usage. From these factors, we can create an overview of both performance and hardware demand in a generalized manner. These factors in combination may be simplified for how to determine performance and demand, however, these are the factors of choice so that we as researchers could spend more of our limited time determining results.

4.2 Independent variables

When creating applications and tests our goal has been to be as efficient as possible given our domain knowledge. This includes minimizing code, files, and how we generate and visualize everything. Both authors have been working in Android Studio to develop the applications. Both authors have used the same version of Android Studio and the same emulator during development.

4.3 Sampling Strategy

Data has been chosen to consider multiple factors that may affect the performance of an application, which is all aimed towards creating the best experience for end users.
Our sample size is based on iterations for each test, the amount of iterations has been discussed during the entire project and revolves around statistical factors and time constraints. By using a specific number of iterations we can decrease the time taken while also minimizing the risk of inconclusive results.

4.4 Experimental Setup

To test each application, a mobile device is preferable due to running the test on Android Studio's inbuilt emulators may differentiate from the proper result on physical devices. When running a test on an emulator, Macrobenchmark gives a warning that the performance is not representative of a real device as the emulator shares resources with the host operative system. To test the applications a mobile device is being used and the hardware specifications of the testing device are as follows:

Table 4.1: Hardware specification

Device	OnePlus Nord
OS	OxygenOS (based on Android 10)
CPU	Qualcomm® Snapdragon ™ 765G 5G
GPU	Adreno 620
RAM	8GB/12GB LPDDR4X
Android version	11

4.5 Procedures

The procedure to test the difference in performance between *Jetpack Compose* and *XML Views* is to create each application concurrently for each toolkit. This started with low demand for both *XML Views* and *Jetpack Compose*. The low-demand application is a static application with two posts. These posts are a picture with a title and a small text for the date of publication. These applications are both written with *XML* (Appendix D, Figure D.1b) and *Jetpack Compose* (Appendix D, Figure D.1a) UI toolkit.

The low-demand applications were then modified for the next step which was medium demand concurrently for both toolkits. The medium-demand applications use a horizontal scrollable list as a row list of 10 items and a vertical scrollable list of 10 posts. The posts are almost the same as on the low demand but now have two icons between the post image. The icons are a heart as a like button and a share button. The medium applications are also written with *XML* (Appendix D, Figure D.2b) and *Jetpack Compose* (Appendix D, Figure D.2a) as UI toolkit.

Then the medium application was copied and modified for high demand. High-demand applications are almost the same as medium-demand applications. The big difference is that the vertical post list is smaller and under the list there is a three times three grid of smaller pictures. These applications are written with either *XML* (Appendix D, Figure D.3b) or *Jetpack Compose* (Appendix D, Figure D.3a) as UI toolkit.

Every application was later revised to make them as similar as possible with the same images, image sizes, padding, and margins. Following the development of the applications tests were created for benchmark. The first Macrobenchmark test is a startup test. The same code for this test can be run on either toolkit on every application. Then an interactive test was used to measure the amount of time the CPU took to render each frame while scrolling the application using *Frame Timing Metric (FTM)*.

4.6 Application

In this section, the applications of each demand will be described.

4.6.1 Low demand

The low-demand application is a static application with 2 posts. Each post in the applications is an image with a title and a publish date to visualize a social media application. The low-demand applications are used to measure the startup time and used to compare between both *XML* (Appendix D, Figure C.1) and *Jetpack Compose* (Appendix D, Figure C.1).

4.6.2 Medium demand

The medium-demand applications use a horizontal row of 10 items and each item is an image with a title under it as seen in D.2b and D.2a.
XML Views (C.1) utilizes the functionality of horizontal scroll view and scroll view to create the visual aspects of the post and row scroll windows. These separate views need to utilize linear layout constraints to be inflated with their corresponding posts and cards. Using a linear layout for both of these views is common practice and recommended by Google themselves (Google 2024*e*)[1]. Using these views applications are able to include complex hierarchies and the development process is similar to creating web services using *HTML*.
Jetpack Compose (C.1) uses a *LazyRow()* that is a composable function that provides a set of composable items and lays out the items that are only visible in the components viewport (Google 2024*f*)[2]. The visible component in the viewport is the post the user can see without scrolling the application and the other post in the *LazyRow* will only be rendered if the user scrolls to the next post. The *LazyColumn* works the same way as *LazyRow* but it is a vertical scroll instead of horizontal. Lazy functions are recommended for use when the list has to display a large number of items or a list of an unknown length (Google 2024*f*)[3].

4.6.3 High demand

High-demand applications are almost the same as medium-demand applications and the only difference is the grid layout under the post scroll. In *Jetpack Compose* (C.1) a *LazyVerticalGrid* was used to display a grid layout of 9 items. *LazyVerticalGrid* is a composable function to display the items in a grid (Google 2024*f*). The grid is in a vertical orientation. *XML Views* (C.1)instead utilizes a Recycler view (Google 2024*g*)[4] which is common practice to create grids of items. Recycler view is commonly used to be able to manipulate data but has appropriate tools to handle a grid and generate items for the grid.

[1]Horizontal scroll view
[2]List and grids
[3]List and grids
[4]Recylcer view

4.7 Data Collection Methods

Benchmark provides a JSON file with data that can be measured, compared, and visualized using Python. Collected data was then used for statistical analysis to reduce the risk of validity concerns for statistical errors. Macrobenchmark can only be used to measure startup time in milliseconds (ms) and the total time in milliseconds (ms) for the CPU to produce a frame *(FTM)*. Testing the startup time for each application, the test A.1 is used. The test to collect the post-scrolling data is in Figure A.2 and the data from row scrolling is in Figure A.3. To test the applications with row scrolling and post scrolling, the A.4 was used. *FTM* is measured by the percentile of the frame it takes to produce. The data is as follows: *P50, P90, P95, and P99* where each percentile is the percentage of the frames produced.
To collect the memory, CPU, and battery, *Android Profiler* had to be used instead, this is due to API limitations from Macrobenchmark.

4.7.1 Memory

For memory usage *Android Profiler* has been used to see total memory usage during each iteration of the tests. The data is collected from the *FTM* measurement. *Android Profiler* shows the usage for the specific iteration how much total memory has been used, and the amount of memory usage is then noted and stored for each application.

4.7.2 CPU

CPU usage has been measured with *Android Profiler* by utilizing graph usage of CPU over time. The graph visualizes the percentile of usage over the time of the test and to use a good value of the usage of the CPU, the peak value is used.

4.7.3 Battery

When measuring the amount of battery being used is collected with *Android Profiler* which measures the current usage of the battery. The current is recorded as the instantaneous current in microamperes (μA).

4.8 Data Analysis Plan

The data collected during the study was objective, meaning there is no room for ambiguity between results. The data analysis includes steps to counteract statistical faults that may occur during the data collection and analysis phase. Concerning hypothesis, using a hypothesis that has no strict boundaries, for example, “XML is better” would not yield the same analysis as “neither is better than the other” in which we can describe situations and outcomes where the toolkits perform differently. The research questions serve the same purpose, in which we have a way to present an answer which is not yes or no, but rather an elaboration on how and why our results are presented.

4.9 Ethical Considerations

Ethical considerations during the project rely on being objective when creating applications. A constant reminder of progressing procedures needs to be considered due to utilizing available

tools in a proper manner. Other ethical considerations concern our results, where we objectively need to present our results by not creating an analysis of "this is better than that" since our domain knowledge may limit these results to some degree. Being careful with how we present results will be a very impactful factor for ethical concerns. Further usage of sources has been thoroughly researched to remain ethical throughout the entirety of this paper.

5 | Results

When investigating results, Macrobenchmark has been used, which is a built-in tool in Android Studio, where a test suite can be created. These tests are general for every application, except for scrolling tests which are not used for the low demand applications. Macrobenchmark outputs JSON data which has been used to calculate and influence statistical faults where they may occur. The JSON data has also been used to create graphs to represent the outputs of each test and compare the applications to each other. To determine hardware demand, *Android Profiler* has been used as well as the Sensor Logger application. Data has been extracted using the built-in *Android Profiler* graphs to determine hardware usage.

5.1 Program startup

For startup, the application starts and closes. Here a time interval can be determined for the time from start to being loaded. The results gathered are presented in milliseconds. The test runs *StartupMode.COLD* so that there is no background task for the application running, meaning it's the first time to start on the phone each time(no multi-task/ background process). The iteration started at 10, using a confidence interval of 95% for these results, an expected amount of iterations could be determined. Using sample size calculation 5.1 a table 5.1was created for an expected amount of iterations.

$$n = \frac{z^2 \cdot \sigma^2}{E^2} \tag{5.1}$$

Table 5.1: Expected iterations

Application	**Expected iterations**
Jetpack low	79
XML low	43
Jetpack medium	246
XML medium	417
Jetpack high	671
XML high	1769

1769 iterations is unreasonable for our time constraint so other measures had to be taken into consideration. Instead, we determined the results are sufficient by using the same confidence interval and presenting upper and lower bounds for each test to determine how close to reality each startup test is. Using the formula 5.2 we can determine a margin of error. By instead running 30 iterations a new table 5.2 can be presented.

$$E = Z \cdot \left(\frac{\sigma}{\sqrt{n}} \right) \tag{5.2}$$

Table 5.2: Expected mean value (a = 0.05)

Application	**Mean including bounds**
Jetpack low	1316.516 ± 3.61
XML low	1327.894 ± 3.75
Jetpack medium	1373.256 ± 0.68
XML medium	1352.212 ± 1.58
Jetpack high	1377.562 ± 27.19
XML high	1341.087 ± 9.32

The tests for each application are fairly basic, the test uses the metric *StartupTimingMetrtic()* to collect the startup time. When the test runs, it presses the home button on the device and then starts the activity, and waits for it to fully load on the device. *StartupTimingMetrtic()* test measures the time from pressing the home button to when the application has fully loaded on the device as seen in Figure A.1.
Using these we determine that the data is reasonable except for *Jetpack Compose*- and *XML Views* high. This can clearly be visualized as why the margin of error is significant compared to the others, where the first three tests for *Jetpack Compose* run as outliers compared to the rest: 1819.794 ms, 1783.691 ms, 1804.562 ms.
Where the mean value excluding outliers for *Jetpack Compose* high is about 1375.268 ms. The last time for *XML Views* high is 1458.208749 ms compared to the mean excluding outlier which is about 1353.465 ms. A margin of error ranging from 0.68 ms to 3.75 ms is a reasonable outcome when weighing the amount of iterations to a reasonable result. For excluded outliers we can determine new ranges for these applications by simply removing outliers, just to be able to determine a more reasonable range. This can be argued to threaten validity, however, so can including the outliers. The observed behaviour may be due to computer hardware, which can have an impact on why this behavior is present.

Table 5.3: Mean excluding outliers

Application	Mean excluding outliers
Jetpack high	1373.331 ± 5.26
XML high	1352.611 ± 5.89

These values are acceptable to the tests we are conducting, which use a better mean value that does not favor any of the used toolkits. To present these differences we have decided to include both results since we want to minimize other factors impact on application performance. This can also be seen in Figure E.3 and Figure E.4 Now using all data available, we can calculate performance differences between each stage, marked as lower, mean, and upper bounds for each application.

Table 5.4: All results startup

Boundary	Best performance	Percentage better performance
low lower	Jetpack	0.855
low mean	Jetpack	0.864
low upper	Jetpack	0.873
medium lower	XML	1.625
medium mean	XML	1.556
medium upper	XML	1.487
high lower	XML	1.404
high mean	XML	2.721
high upper	XML	4.027
modified high lower	XML	1.584
modified high mean	XML	1.531
modified high upper	XML	1.479

For the results of the startup test, *XML Views* has the majority in faster start-up times, whereas *Jetpack Compose* outperforms it slightly at the low-demand applications. A large discrepancy can be shown at the high-demand upper boundary, whereas the modified calculations show similar results to previous tests. Usage of multiple measurements for each application comparison is used to counteract the fact that the volume of iterations is kept lower due to time constraints and other limiting factors that may affect a larger amount of iterations. Furthermore, performing a t-test determines the significance of the startup tests. Every demand proves to have significant results compared to our alpha value of 0.05, as seen below.

Table 5.5: T-test Startup

Demand	P-value	Difference
Low	0.01	Significant
Medium	7.25e-7	Significant
High	0.01	Significant

5.2 Post scroll test

To maintain consistency with previous tests, 30 iterations of the *FTM* tests have been used for each application to determine the time each frame takes to produce for the CPU. The *FTM* test for each application is very similar to keep consistency between each application so the test data will be as similar between each application. As seen in Figure A.2, the test will do 30 iterations and for each iteration, the test will first press the home button on the device, then start the activity and wait. When the activity has started the scroll test begins. The scroll test first finds the scrollable object on the device with the name `"post_list"`, waits for idle, and then sets where on the device the test should scroll by setting the gesture margin. When the test knows where to scroll, the test will scroll in the down direction by 300%. The reason for 300% is that the scroll action has to cycle for more than one post. The last wait for idle is only so the scroll action is at a complete stop. A t-test cannot be performed because each run of *XML* produces 786 rows of data points, whereas each run of *Jetpack Compose* produces only 10 rows of data points.

Table 5.6: Post scroll

Application	**P50 mean**	**P90 mean**	**P95 mean**	**P99 mean**
XML Medium Demand	8	11	12	20
Jetpack Compose Medium Demand	15	1235	1240	1249
XML High Demand	8	11	14	134
Jetpack Compose High Demand	9	1237	1243	1251

For these results P50 mean is almost negligible between the applications. *XML Views* medium- and high demand has equal results, while *Jetpack Compose* medium only has 7ms more, and high only has 1ms more.
For P90 and up we can see a significant benefit for *XML Views*, this can conclude a result of *XML Views* takes less time to produce the frame as seen in E.5 for *XML Views* and E.6 for *Jetpack Compose*.

5.3 Row scroll test

The Row scroll test is practically the same as the post scroll test, the only difference is the objective name for the row list and the direction of the scroll is right instead of down. See Figure A.3

Table 5.7: Row scroll

Application	**P50 mean**	**P90 mean**	**P95 mean**	**P99 mean**
XML Medium Demand	8	12	14	23
Jetpack Compose Medium Demand	15	1235	1242	1249
XML High Demand	8	12	13	134
Jetpack Compose High Demand	12	1236	1245	1257

Row scroll tests can conclude similar results as post scroll tests, where P50 differences are almost negligible while P90 and up present a large discrepancy between the applications. A t-test cannot be performed because *XML* and *Jetpack Compose* runs have different numbers of data points.

5.4 All test

For all tests, we run 100 iterations with previous measurements to determine application performance for medium and high-demand applications. The reason for 100 iterations on *All Test* is because the data extraction is very simple from this point. The test is both using the *ScrollPostList()* from the post scroll test and *ScrollRowList()* function from the row scroll test combined as one test with 100 iterations (A.4).

Table 5.8: All Test

Application	**P50 mean**	**P90 mean**	**P95 mean**	**P99 mean**
XML Medium Demand	9	12	15	29
Jetpack Compose Medium Demand	67	298	1241	1254
XML High Demand	10	15	20	29
Jetpack Compose High Demand	11	721	1246	1258

These results show similarities to the individual tests from post and row scroll. *Jetpack Compose* medium demand for P50 has a slower time than its corresponding high demand. The assumption for this result is that the bottom of *Jetpack Compose* high includes an image gallery that covers the post-scroll view. For FTM, a t-test is not possible due to the output of the test, where the runs do not inform you of which measurement is used p50, p90, p95, or p99. Both output files are dissimilar in output and lines. So to the benefit of the test, the apparent results show a benefit to *XML Views* except for p50.

5.5 Hardware test

To determine hardware usage all tests have been used as a script to run, using this we can collect data from each iteration for CPU, memory, and battery usage using *Android Profiler*.

For low-demand applications, a start-up test has been used since it does not include any scroll functionality. Using this we can determine differences between each demand application. To run these tests 30 iterations were picked because of the complexity of data collection using *Android Profiler*. Every data point has to be extracted manually from each iteration. Extracted data from hardware tests can be found in C.2 as *Android profiler* in PDF or Excel formats.

5.5.1 CPU

CPU has been measured using peak usage for each *XML Views* and *Jetpack Compose* application. From the 30 iterations we can present a table of sample means for a more general view of the results. 30 iterations have been used to calculate a mean and a true mean boundary with 95% confidence. This can be shown in the table below and in Figure E.8.

Table 5.9: CPU peak percentage usage

Demand	**XML**	**Jetpack Compose**
Low	18.614	19.463
Medium	19.893	36.980
High	21.297	27.820

CPU performance shows consistent results for *XML Views* across every single application, to be noted again, is that Low-Demand uses startup to measure, and it still yields consistent CPU performance across the board. *Jetpack Compose* instead presents the most CPU usage for the medium-demand application. Overall we can see that *XML Views* offers an advantage to CPU usage overall. Using the alpha value of 0.05 we can create the following table to compare each demand stage to each other including lower, mean, and upper boundary.

Table 5.10: CPU peak percentage usage

Demand	**Best performance application**	**Percentage improvement**
Low lower	XML	5.18
Low mean	XML	4.46
Low upper	XML	3.79
Medium lower	XML	61.62
Medium mean	XML	60.07
Medium upper	XML	58.77
High lower	XML	20.54
High mean	XML	26.55
High upper	XML	31.99

For these values, we can clearly see *XML Views* outperforms *Jetpack Compose* in CPU peak usage. The difference between low demand is very subtle, however extremely apparent at medium- and high demand. Furthermore, a t-test has been used to determine the significance of the differences, which points to medium- and high-demand having significant differences between the toolkits.

Table 5.11: T-test CPU

Demand	**P-value**	**Difference**
Low	0.08	Not Significant
Medium	1.28e-25	Significant
High	9.54e-8	Significant

5.5.2 Memory

Memory has been measured using total memory usage for each application. 30 iterations have been used collected by observation the same way as CPU. There are some inconsistencies in the results, these are presented as "ex" in the table below, to exclude outliers for a more consistent comparison. This could not be done for *Jetpack Compose* Medium and High arbitrarily, and instead will be explained further below, hence these get an asterisk after their value. Can also be visualized in Figure E.9 including all iterations and Figure E.10 excluding semi-arbitrary outliers.

Table 5.12: Memory Usage (MB)

Demand	**XML**	**Jetpack Compose**
Low	88.620	84.660
Medium	327.679	379.867
High	274.243	267.52
Low ex	56.809	57.377
Medium ex	337.321	379.867*
High ex	337.435	267.52*

For memory usage, extracted data concludes that *XML Views* offers more consistent results, however, important to note (as seen by an asterisk) that *Jetpack Compose* values could go as low as 57MB usage consistently for some iterations, while other iterations could fluctuate between 400-900MB usage as shown in the datasheet C.2. This is important to note for the results. The difference is almost unnoticeable for low and medium demand, however for excluding outliers result *Jetpack Compose* outperforms *XML Views* at high demand excluding outliers. Both a true mean with 95% confidence including and excluding outliers. To be noted, the outliers have been picked semi-arbitrarily and can be shown in the datasheet. Also to be noted is the inconsistency of *Jetpack Compose* memory usage, which has the same value compared to both *XML Views* included and excluded outlier values.

Table 5.13: Total Memory Usage(MB) Efficiency

Demand	**Best performance application**	**Percentage improvement**
Low lower	Jetpack Compose	2.75
Low mean	XML	4.57
Low upper	Jetpack Compose	5.53
Medium lower	Jetpack Compose	14.89
Medium mean	XML	14.74
Medium upper	XML	35.04
High lower	Jetpack Compose	19.03
High mean	Jetpack Compose	2.48
High upper	XML	8.23
Low lower ex	Jetpack Compose	28.05
Low mean ex	XML	0.995
Low upper ex	XML	23.25
Medium lower ex	Jetpack Compose	19.27
Medium mean ex	XML	11.86
Medium upper ex	XML	33.54
High lower ex	Jetpack Compose	54.23
High mean ex	Jetpack Compose	23.11
High upper ex	XML	0.98

Furthermore, running t-test, no significant differences can be determined using an alpha value of 0.05.

Table 5.14: T-test Memory

Demand	**P-value**	**Difference**
Low	0.84	Not Significant
Medium	0.39	Not Significant
High	0.88	Not Significant

5.5.3 Battery

The battery has been measured by current for each application. The main objective of collecting data was to use an application in the background called *Sensor Logger* and the reason why the data from *Sensor Logger* was never used is due to the test running in different time frames. For instance, *All Test* for *Jetpack Compose* High Demand took only 15 minutes while *All Test* for *XML Views* medium Demand took 45 minutes. The collected data from *Sensor Logger* could not represent this study. Conversely, *Android Profiler* collects the instantaneous current in microampere (μA) during the test. Battery consumption has been measured by observation from the *Android Profiler*. A current can be collected for current usage in microampere (μA), these results show apparent inconsistencies so outlier exclusion has been applied semi-arbitrarily as shown in data chart C.2 and Figure E.11 with all iterations and Figure E.12 excluding semi-arbitrary outliers.

Table 5.15: Battery Usage

Application	**Current (μA)**
XML Low Demand	89069
Jetpack Compose Low Demand	68600
XML Medium Demand	91241
Jetpack Compose Medium Demand	102167
XML High Demand	120300
Jetpack Compose High Demand	318367

Battery usage has proved difficult to measure during the project, we would argue that these results could be somewhat inconclusive using the extracted data. *XML Views* shows clear advantages for the high-demand application, while *Jetpack Compose* shows more benefits for low- and medium-demand (excluding semi-arbitrary outliers).

Table 5.16: Battery Current Usage Using outlier data

Demand	**Best performance application**	**Percentage improvement**
Low lower	Jetpack Compose	17.32
Low mean	Jetpack Compose	25.97
Low upper	Jetpack Compose	29.89
Medium lower	XML	63.49
Medium mean	XML	11.32
Medium upper	Jetpack Compose	15.65
High lower	XML	95.13
High mean	XML	90.33
High upper	XML	86.93

Due to the nature of the tests, some iterations result in negative battery current, these are included in the above table. The battery testing would benefit from a higher API level to extract data more consistently, however, this data still shows differences between the toolkit, especially for the high-demand applications where *XML Views* performs much better. We believe there may be incorrect measurements performed from the profiler, which compensates for current usage between iterations, therefore exclusion method results are included in the below table.

Table 5.17: Battery Current Usage Using outlier data

Demand	**Best performance application**	**Percentage improvement**
Low lower ex	XML	3.32
Low mean ex	XML	1.68
Low upper ex	XML	0.38
Medium lower ex	XML	2.61
Medium mean ex	Jetpack Compose	2.47
Medium upper ex	Jetpack Compose	6.33
High lower ex	XML	107.90
High mean ex	XML	99.47
High upper ex	XML	92.54

T-test has also been used to determine the significance of battery usage differences.

Table 5.18: T-test Battery

Demand	**P-value**	**Difference**
Low	0.34	Not Significant
Medium	0.66	Not Significant
High	8.71e-11	Significant

6 | Discussion

Application development during this study has proved to be simple. Application development has only been a minor part of the research. The problematic parts instead are related to data extraction and analysis. Using the tools of choice, output has been limited during the study and is something we would like to see improved in future research.
Subjectively we see no major differences in development, we had some ideas that needed to be changed or compromised, such as circular images where *XML Views* had no built-in tools to do so. Gallery views had to be slightly different between the applications as well. Choosing a toolkit for development time could be considered irrelevant since it mostly depends on workflow and domain knowledge for either toolkit. Milla & Radonjić (2023) discussed that over time *XML Views*-based applications increases in size and complexity. This is due to one component being defined and used in a layout file, the component must be connected to a fragment class and be inflated. Where the screen layouts are defined in *XML* files and the logic handles the user interaction are defined in activity files. Therefore it is problematic that the application will be written in two languages, *XML* for the layout screen and Java/Kotlin for the activity when developed. Meanwhile *Jetpack Compose* is only written in Kotlin. *Jetpack Compose* is a declarative approach, which means the developer describes what the program should do, not how to do it. This makes *Jetpack Compose* increase the ease of use and simplicity, mostly because of only one programming language. The conclusion of Milla & Radonjić (2023) is that *XML Views*-based approach of UI design can be quite complex and time-consuming compared to *Jetpack Compose*.
Emulator performance differences are considered negligible due to poor emulator performance overall. No noticeable differences can be extracted using our pc/mac hardware. Therefore only an Android mobile device was used to conduct tests and remained the focus for gathering results. The tests were written to measure different aspects of an Android application which is to measure the startup time, the time the CPU takes to produce a frame as if a user is interacting with the application, CPU usage, memory usage, and battery usage. The applications the test ran on were either using *XML Views* or *Jetpack Compose* as UI toolkits and had different types of demand.
The demand for the applications was in three stages, The first stage was the low demand as a static application with two posts, each post consisting of an image, title, and publish date. Medium demand was more demanding as it had 1 horizontal row scroll of 10 items where each item was 1 image and 1 title. Under the row, there is a vertical scroll of 10 posts. Each post of those 10 is the same as in low demand with one image, title and publish date. High demand is similar to medium demand but has a grid layout under the vertical post scroll. Then all the applications were tested with the same tests and the data was collected plotted and compared between the different demands to draw a conclusion of which UI toolkit to use.

6.1 Tests

In this chapter, the tests will be discussed based on the result from the previous chapter between *XML Views* and *Jetpack Compose* low, medium, and high demand.

6.1.1 Startup test

The startup test was the first test to be created. For data extraction, this proved to be a remarkable start to gathering results. There were multiple data points that could be used from the JSON data which allowed for thorough analysis. As seen in graph E.2 and E.1, the comparison between the startup time for *XML Views* and *Jetpack Compose* is not as large as previous work. As Fjodorovs & Kodors (2021) tested his *XML Views* and *Jetpack Compose* applications which were one image and some Lorem Ipsum text, *Jetpack Compose* was 46% slower than *XML Views* in startup time. Due to the application, Fjodorovs and Kodors is only one image and some text, we can assume the applications are smaller than our low-demand applications. Fjodorovs & Kodors (2021) tested their applications in 2021 and as of 2024, our low demand *Jetpack Compose* is performing ≈1% faster than our low demand *XML Views*.

The medium demand applications result shows that *XML Views* performs better on average by ≈1.6% over *Jetpack Compose*. For high demand, *XML Views* performs better than *Jetpack Compose* on average by ≈2.7%. The result therefore shows that *XML Views* medium and high startup time outperform by a small margin over *Jetpack Composer* medium and high demand. Conversely *Jetpack Compose* performs better with low demand than *XML Views* low demand by ≈1%.

6.1.2 Scroll test

Scroll tests were the second part of the tests to be created and used. To extract data here was more complicated, despite this, mean values could be extracted to minimize the risks of inconclusive results, helping us create a better analysis for the scroll tests.

The scroll test consisted of one *Macrobenchmark* test with 100 iterations. The test was to do a horizontal scroll of a row of 10 posts and scrolling vertically of a column of 10 posts. The results between these tests are very similar for *XML Views* medium and high demand. The amount of time the CPU takes to produce *P50, P90, P95, P99* when scrolling vertically and horizontally are very close between medium and high demand as seen in Figure for *Jetpack Compose* E.6 and *XML Views* E.5.

For *XML Views* to produce 50% of the frames only takes 9 ms for medium and 10 ms for high demand. Conversely, the time for *Jetpack Compose* producing 50% of the frames takes longer for medium demand with 67 ms while high demand took 11 ms. The high demand of *Jetpack Compose* result of the test are just 1 ms above high demand *XML Views*. While medium demand *Jetpack Compose* takes 59 ms longer than *XML Views* medium demand.

Looking at the plot of *XML Views* with 90% of the frames produced, it only takes 12 ms for medium and 15 ms for high demand to be produced. The increased time for *XML Views* to produce 90% of the frames is ≈ 33% and 50%. At 90% of the frames produced, *Jetpack Compose* increases the time to produce a frame for medium demand from 67 ms to 298 ms which is an increase of ≈ 344.78%. For high demand the amount of time to produce 90% of the frame is even worse as the time increased by ≈ 6454.55% from 11 ms to 721 ms

The time for *XML Views* when producing 95% and 99% of the frames, the time does not increase as much as for *Jetpack Compose*. *Jetpack Compose* time increases greatly for both demands, from 90% to 95% of the frames, the time increases from 298 ms to 1241 ms for medium demand and 721 ms to 1246 ms. The increased amount then just increases a small amount from 95% to 99% which is from 1241 ms to 1254 ms for medium demand and from 1246 ms to 1258 ms.

The test results show that *XML Views* outperforms *Jetpack Compose* as it takes less time for the CPU to produce the frames for the applications. While observing the tests of the applications, the medium demand and high demand of *Jetpack Compose*, the test did not go as smoothly as for *XML Views*. The *Jetpack Compose* applications have stutter and *jank* on the device while scrolling which is caused by *Jetpack Compose* applications taking longer to load the frames.

6.1.3 All test

All tests proved to be the most complicated part to measure. Since our API on the hardware limited Macrobenchmark output we had to be more creative in how data was gathered. By doing 30 iterations for each we could manually observe each iteration performance as seen in Figure E.7. Previous tests have shown that early and late iterations could include compromised data due to exclusive factors. To avoid this many iterations had to be observed to be able to conclude any results for hardware performance. For medium and high demand all test was used, while for low-demand applications a start-up test was used. Before each test run the phone battery was charged to maximum capacity to increase validity and decrease outlying factors affecting results. Since the output was also in the form of graphs, we had to be creative about which data points to use. We could easily find CPU maximum usage, this may not be the absolute best approach since the application may use less mean CPU between applications, however, since the tests are very rapid, maximum CPU usage was a suitable measurement.
Memory had a more straightforward approach, where total memory usage was easily observed using the available tools. Each iteration gives results for this which was then used to measure memory usage.
Battery also had an easier approach to measure, where current is shown during the iteration, this data was used to measure battery usage.

6.2 Sustainability

How can this research benefit sustainability? For the ecological dimension, this field can decrease hardware usage by gathering information regarding phone hardware. By decreasing hardware usage longevity can be increased, since the Android market covers 70% of the global market (Twinr 2024), the ecological benefits can be significant.
For the societal dimension, this work can increase equity for developers, by creating more thorough research of available tools, with benefits and shortcomings, developers anywhere have the opportunity to utilize this information to their benefit. Extending this further by including more

developers more people can be reached with applications utilizing this information.
Economically this research can potentially help old hardware utilize new software better. By decreasing the gap between high- and low-end hardware. By offering lower-end hardware with equal performance, revenue can instead be focused on other sustainability development efforts.
Google's model may fail for what they are trying to produce for the time being as *XML Views* views have been stable for a much longer time. However,*Jetpack Compose* updates are rapid and consistent and may be beneficial in the future.

UN sustainability development goal 9 has been kept in mind when conducting this research. *"Build resilient infrastructure, promote inclusive and sustainable industrialization and foster innovation"* where this research focuses on innovation and inclusion. By encouraging innovation the mobile market's negative impact on sustainability can be decreased with better performance for both hardware and software.

6.3 Future work

For future work, we would encourage more hardware to be tested. This will allow the research to be more general. Further, we would encourage more iterations to gather more precise data for the tests. We would also suggest more refined tests for hardware specifically, which is easier for Android API versions over 30. Macrobenchmark is very limited for used hardware during this study. Future work could also utilize and improve application code if needed, this again is a part of the study that may be hard to set rules for and may influence the outcome in multiple ways. Further focus on data collection would also be beneficial, to create a new dimension for hardware measurements. We would also suggest future research to include heavier demand on the application to test our theory that it would benefit *Jetpack Compose* and be more objective in the comparison.

6.4 Limitations

The limitations of the project heavily rely on time constraints. This is very clear in how our applications are developed and their respective functionality. The application is not the main goal, but instead having an application that can objectively answer our questions about whether one toolkit yields an advantage versus the other. This leads to our next limitation of domain knowledge. Our knowledge surrounding application development in Android is limited and may differ between the toolkits and how we solve certain issues regarding the applications. Our knowledge constraint further connects to statistical analysis regarding test specifications, where our limited knowledge may not yield the perfect equations regarding statistically sound iterations to results. Data collection on hardware tests also suffered from time constraints and extraction techniques, where 6 applications needed to be observed manually 30 times for 3 data points for each iteration.

6.5 Threats to validity

During this research, there may occur multiple validity concerns. This section discusses how these threats are handled.

6.5.1 Construct validity

The goal is to measure rendering time, battery usage, and CPU usage based on a toolkit for UI. We decided to keep the applications strictly visual without functionality. We decided to use common icons and menus, again, without functionality. We deemed that the only functionality needed is scrolling when measuring performance, where we can measure whether frames are loaded incorrectly or late. The same applies to the images used where we need to know and understand how fast they can be rendered.
For tests, the measurements we used can contribute to the results of the study that we are searching for. This is mentioned in the application where our measurements are deemed valid for the type of study we are doing.
Regarding our domain knowledge, issues may arise regarding how our applications are developed, and where there might exist more optimal solutions for each application, this is something we have been very careful about to ensure that the application comparison is conclusive.

6.5.2 Internal validity

The internal validity of our research revolves around whether we are using the correct measurements for what we are trying to evaluate. Performance can be interpreted with ambiguity, whether it affects hardware/software or other aspects of an application. For the case of the study, we have done our best to explain what we interpret performance to be, which includes all of our measurements in the results. These aspects are the ones that affect performance the most in different aspects of both battery time and user experience. With these measurements, we can determine a scale for our specific application, and which toolkit performs the best.

6.5.3 Conclusion validity

To minimize conclusion validity concerns we have included statistical analysis to further improve the presented results to a reliable degree. Hardware conclusions have to be carefully considered since the instrumentation limits our capabilities. Instead of enhancing these with statistical analysis, observation had to be used to collect data for hardware usage, which have been done with great consideration.

6.5.4 External validity

For external validity, it is crucial to understand our limitations to hardware. This may affect the outcome of the research, however, by using the hardware we have available, hopefully, our results can be generalized to a decent degree. This validity concern is enhanced by our documentation, available code, and instructions to perform these tests for anyone. We heavily encourage replication of this study, for others to be able to create and read their own data to either disprove or enhance our conclusions.

7 | Conclusion

In conclusion, this bachelor thesis explored the differences in UI performance between *XML Views* and *Jetpack Compose* across three different levels of demand. A thorough literature review was conducted to gain an understanding of *XML Views* and *Jetpack Compose*, which informed qualitative choices throughout the study. Three versions of applications for *XML Views* and three versions of *Jetpack Compose* were developed and benchmarked to measure UI performance in various ways. The benchmark test represented different use cases for each application. The results of these tests indicate that *XML Views* performs better at medium and high demand in startup time and scrolling while *Jetpack Compose* performs better at startup time for low demand and better memory usage on all applications.

7.1 Research Question 1

Initially, our research focused on evaluating the performance disparities between *XML Views* and *Jetpack Compose*. Through testing, it became evident that *XML Views* holds a distinct advantage, particularly in scenarios where applications lack additional functionality. The visual differences between the two setups were apparent, with *Jetpack Compose* manifesting noticeable visual artifacts compared to the smoother rendering of *XML Views*.
Moreover, our findings suggest a relationship between application complexity and toolkit performance. While *Jetpack Compose* exhibits promising potential with increased object density, *XML Views* emerges as the preferred choice for better frame rate and overall performance, especially under higher demand.

- $H1_0$: Can be rejected for each demand.
- $H2_0$: Can not be rejected due to data not being testable to t-test.
- *XML Views* offers a better frame rate as observed during testing.
- *Jetpack Compose* offers better start-up performance for low-demand applications of a significant nature, while *XML Views* offers better performance for medium- and high-demand of a significant nature.
- UI toolkit has a significant effect on startup time on application performance, where *XML Views* proves to be the better choice.

7.2 Research Question 2

Expanding our investigation to include hardware utilization, we evaluated the impact of UI toolkit selection on key metrics such as memory usage, battery consumption, and CPU efficiency. Notably, *Jetpack Compose* demonstrates inconsistent low and high memory usage across all applications, while *XML Views* offers more consistent memory usage across every application. However, when examining battery and CPU performance, *XML Views* showcase a significant advantage. The varying strengths of each toolkit present the potential for hardware demands, proving a beneficial approach based on specific application requirements.

- $H3_0$: Can be rejected for medium and high demand, not for low demand.
- $H4_0$: Can not be rejected for any demand application.
- $H5_0$: Can be rejected for high demand. Can not be rejected for low and medium demand.
- *XML Views* offers better hardware performance for battery and CPU.
- Both methods offer different memory usage capacities in different cases.
- For the chosen metrics, *XML Views* offers the best hardware performance overall, of which the difference is significant for medium-demand and high-demand cpu, and high-demand battery.
- UI toolkit has an effect on hardware demand, depending on which part of hardware is the most important, both applications show some individual benefits.

7.3 Objective

In evaluating the broader implications of our study, we evaluated the practical implications for developers and end-users alike. Our analysis suggests that *XML Views* offer benefits in enhancing end-user satisfaction in medium-demand and high-demand, particularly in scenarios mirroring our experimental setup. This mainly concerns the smoothness of the application during usage. By assessing the performance and hardware implications of UI toolkit selection, our study aims to equip developers with insights to optimize application performance and user experience. While our conclusions highlight the advantages of *XML Views*, future considerations may account for broader development considerations, scalability concerns, and long-term viability.
In summation, our research contributes to the available data surrounding UI toolkit selection, offering additional insights into performance dynamics and hardware implications. By combining empirical findings and qualitative observations, we present a comprehensive narrative that highlights the considerations in UI toolkit selection and application performance optimization.

Bibliography

Android, D. (2011), 'Android', *Retrieved February* **24**, 2011.

Endres, C., Breitenbücher, U., Falkenthal, M., Kopp, O., Leymann, F. & Wettinger, J. (2017), Declarative vs. imperative: two modeling patterns for the automated deployment of applications, *in* 'Proceedings of the 9th International Conference on Pervasive Patterns and Applications', Xpert Publishing Services (XPS), pp. 22–27.

Fjodorovs, I. & Kodors, S. (2021), 'Jetpack Compose and XML layout rendering performance comparison', *HUMAN. ENVIRONMENT. TECHNOLOGIES. Proceedings of the Students International Scientific and Practical Conference* (25), 49–54.
URL: *https://journals.ru.lv/index.php/HET/article/view/6779*

Google (2023), 'UI jank detection'.
URL: *https://developer.android.com/studio/profile/jank-detection*

Google (2024*a*), 'Android cold start'.
URL: *https://developer.android.com/topic/performance/vitals/launch-timecold*

Google (2024*b*), 'Android release notes'.
URL: *https://developer.android.com/jetpack/androidx/versions/all-channeljuly*$_2$8$_2$021

Google (2024*c*), 'AOSP overview'.
URL: *https://source.android.com/docs/setup/about*

Google (2024*d*), 'Automate UI tests: Android developers'.
URL: *https://developer.android.com/training/testing/instrumented-tests/ui-tests*

Google (2024*e*), 'Horizontalscrollview'.
URL: *https://developer.android.com/reference/android/widget/HorizontalScrollView*

Google (2024*f*), 'Lists and grids'.
URL: *https://developer.android.com/develop/ui/compose/lists*

Google (2024*g*), 'Recyclerview'.
URL: *https://developer.android.com/reference/androidx/recyclerview/widget/RecyclerView*

Google (2024*h*), 'Slow rendering'.
URL: *https://developer.android.com/topic/performance/vitals/render*

Google (2024*i*), 'Thinking in compose'. [Accessed 11-03-2024].
URL: *https://developer.android.com/develop/ui/compose/mental-model*

Google (2024*j*), 'Why compose: Jetpack Compose: Android developers'.
URL: *https://developer.android.com/develop/ui/compose/why-adopt*

Javatpoint (2024), 'What is imperative programming - Javatpoint'.
URL: *https://www.javatpoint.com/what-is-imperative-programming*

Jha, A. K., Kim, D. Y. & Lee, W. J. (2019), A framework for testing Android apps by reusing test cases, *in* '2019 IEEE/ACM 6th International Conference on Mobile Software Engineering and Systems (MOBILESoft)', pp. 20–24.

Jha, S. (2011), Poorly written apps can sap 30 to 40% of a phone's juice, *in* 'CEO, Motorola Mobility, Bank of America Merrill Lynch 2011 Technology Conference'.

Kong, P., Li, L., Gao, J., Liu, K., Bissyandé, T. F. & Klein, J. (2019), 'Automated testing of Android apps: A systematic literature review', *IEEE Transactions on Reliability* **68**(1), 45–66.

Kumakura, K., Oguchi, M., Kamiyama, T. & Yamaguchi, S. (2022), CPU usage trends in Android applications, *in* '2022 IEEE International Conference on Big Data (Big Data)', pp. 6730–6732.

Milla, E. & Radonjić, M. (2023), 'Analysis of developing native Android applications using XML and Jetpack [compose]', *Balkan Journal of Applied Mathematics and Informatics* **6**(2), 167–178.
URL: *https://js.ugd.edu.mk/index.php/bjami/article/view/6019*

Noori, Z. & Eriksson, C. (2023), UI Performance Comparison of Jetpack Compose and XML in Native Android Applications, PhD thesis.
URL: *https://urn.kb.se/resolve?urn=urn:nbn:se:kth:diva-328328*

Putranto, B. P. D., Saptoto, R., Jakaria, O. C. & Andriyani, W. (2020), 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)', pp. 383–388.

Santhanakrishnan, G., Cargile, C. & Olmsted, A. (2016), Memory leak detection in Android applications based on code patterns, *in* '2016 International Conference on Information Society (i-Society)', pp. 133–134.

Twinr (2024), 'Android vs. IPhone market share: Exploring dominance in Mobile OS'.
URL: *https://twinr.dev/blogs/android-vs-ios-market-dynamics/*

Yang, Y. (2022), 'Accurately measure Android app performance with profileable builds'.
URL: *https://android-developers.googleblog.com/2022/10/accurately-measure-android-app-performance-with-profileable-builds.html*

A | Code for testing applications

```
@Test
fun startup() = benchmarkRule.measureRepeated(
    packageName = "com.example.highdemand_jetpack",
    metrics = listOf(StartupTimingMetric()),
    iterations = 30,
    startupMode = StartupMode.COLD
) {
    pressHome()
    startActivityAndWait()
}
```

Figure A.1: Test code: startup()

```kotlin
@Test
fun scrollPostTest() = benchmarkRule.measureRepeated(
    packageName = "com.example.highdemand_jetpack",
    metrics = listOf(FrameTimingMetric()),
    iterations = 30,
    startupMode = StartupMode.COLD
) {
    pressHome()
    startActivityAndWait()

    //Scroll on the post lazy column
    scrollPostList()
}


private fun MacrobenchmarkScope.scrollPostList() {
    val contentList = device.findObject(By.res("post_list"))

    device.waitForIdle()

    contentList.setGestureMargin(device.displayWidth / 3)

    contentList.scroll(Direction.DOWN, 300f)

    // Wait for the scroll to finish
    device.waitForIdle()
}
```

Figure A.2: Test code: scrollPostTest()

```kotlin
@Test
fun scrollRowTest() = benchmarkRule.measureRepeated(
    packageName = "com.example.highdemand_jetpack",
    metrics = listOf(FrameTimingMetric()),
    iterations = 30,
    startupMode = StartupMode.COLD
) {
    pressHome()
    startActivityAndWait()

    //Scroll on the row list
    scrollRowList()
}
private fun MacrobenchmarkScope.scrollRowList() {
    val contentList = device.findObject(By.res("row_list"))

    device.waitForIdle()

    contentList.setGestureMargin(device.displayWidth / 4)

    contentList.scroll(Direction.RIGHT, 300f)

    device.waitForIdle()
}
```

Figure A.3: Test code: scrollRowTest()

```kotlin
@Test
fun allTest() = benchmarkRule.measureRepeated(
    packageName = "com.example.highdemand_jetpack",
    metrics = listOf(FrameTimingMetric()),
    iterations = 1,
    startupMode = StartupMode.COLD
) {
    pressHome()
    startActivityAndWait()
    scrollPostList()
    scrollRowList()
}
```

Figure A.4: Test code: allTest()

B | Instructions for testing

1. Locate build variants. B.1
2. Set module: app to active build variant: benchmark. B.1
3. Locate project folder.
4. Go to: app/src/test/java/com/example/APPLICATION/ExampleUnitTest.kt. B.2
5. Press green arrow: modify run configuration. B.3
6. Instrumentation arguments. B.4
7. Press +, androidx.benchmark.suppressErrors (enter), EMULATOR (enter), APPLY. B.5
8. Run desired test B.6

If errors occur follow this guide (https://developer.android.com/codelabs/android-macrobenchmark-inspect)

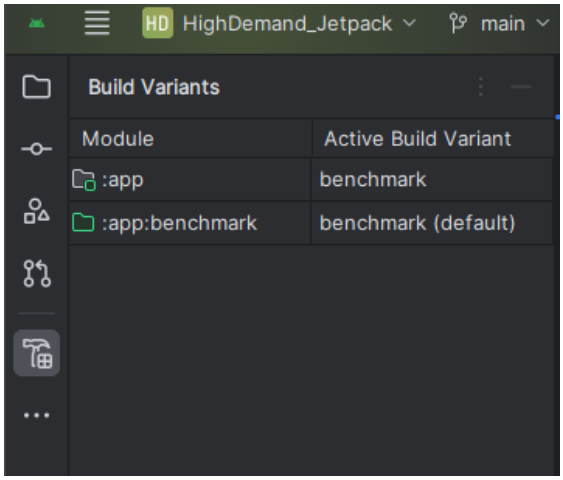


Figure B.1: Build Variants

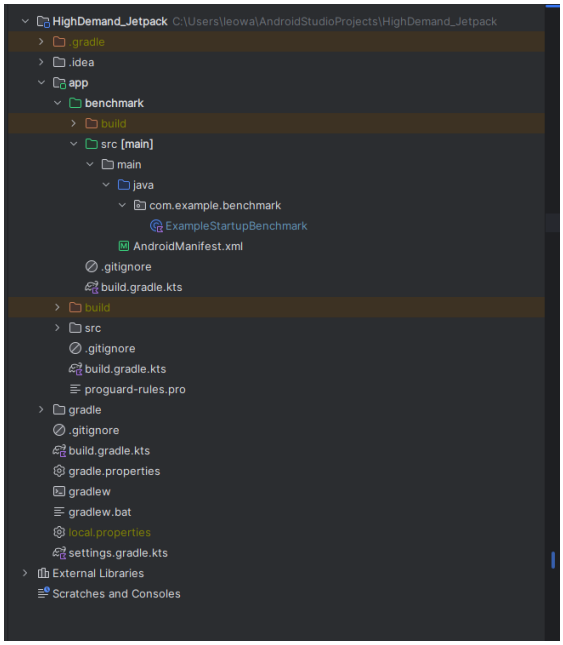


Figure B.2: File tree of Benchmark

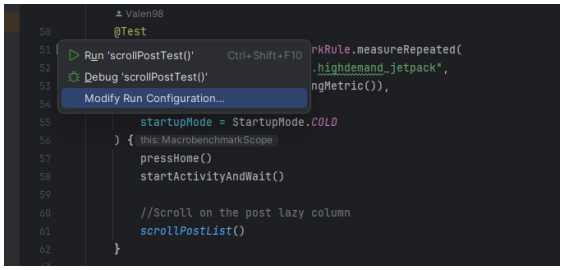


Figure B.3: Modify Run Configuration

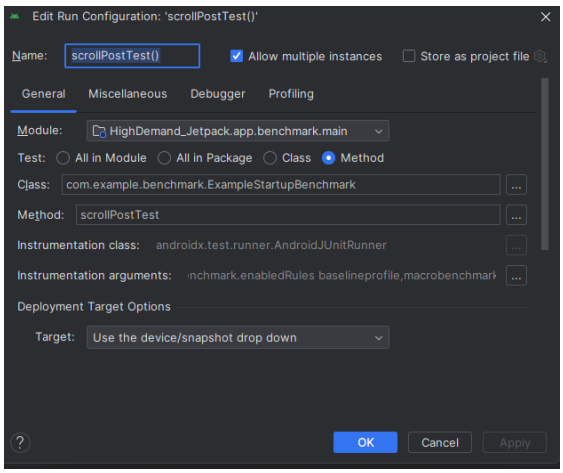


Figure B.4: Edit Instrument arguments

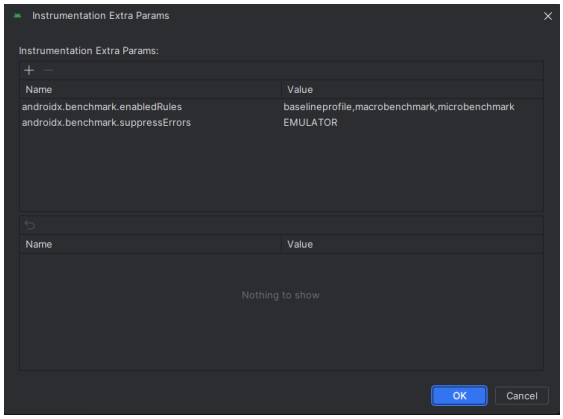


Figure B.5: Add extra instrumentation parameters

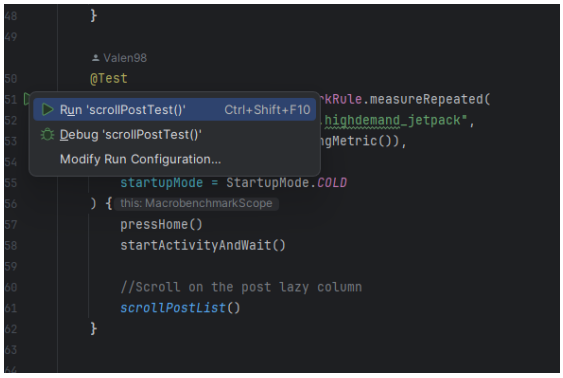


Figure B.6: Run the test

C | Data sources

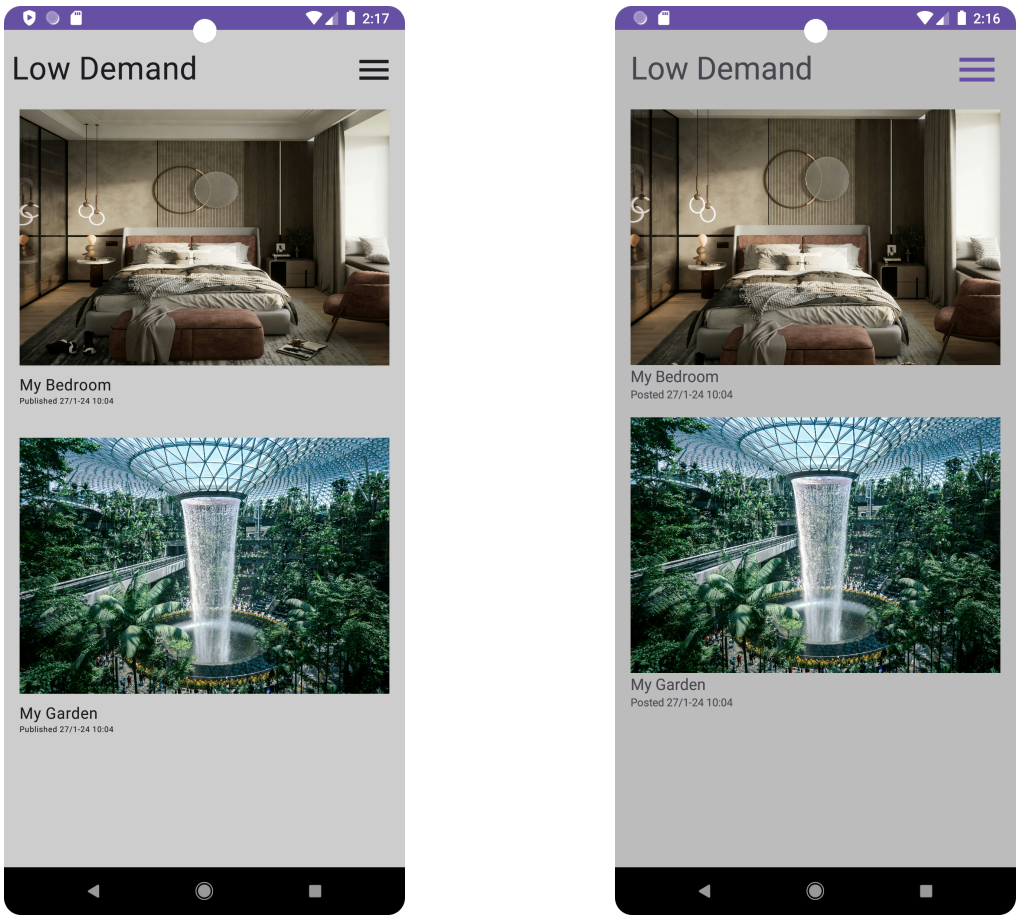
C.1 Applications

Low Demand XML: `https://github.com/AntonBjarne/LowDemand_XML`
Medium Demand XML: `https://github.com/AntonBjarne/MediumDemand_XML`
High Demand XML: `https://github.com/AntonBjarne/HighDemand_XML`
Low Demand Jetpack Compose: `https://github.com/Valen98/LowDemand_Jetpack`
Medium Demand Jetpack Compose: `https://github.com/Valen98/MediumDemand_Jetpack`
High Demand Jetpack Compose: `https://github.com/Valen98/HighDemand_Jetpack`

C.2 Data

Python plot and extracted data: `https://github.com/Valen98/ExamData`
Excel of CPU, memory, and battery data for each application:
`https://github.com/Valen98/ExamData`

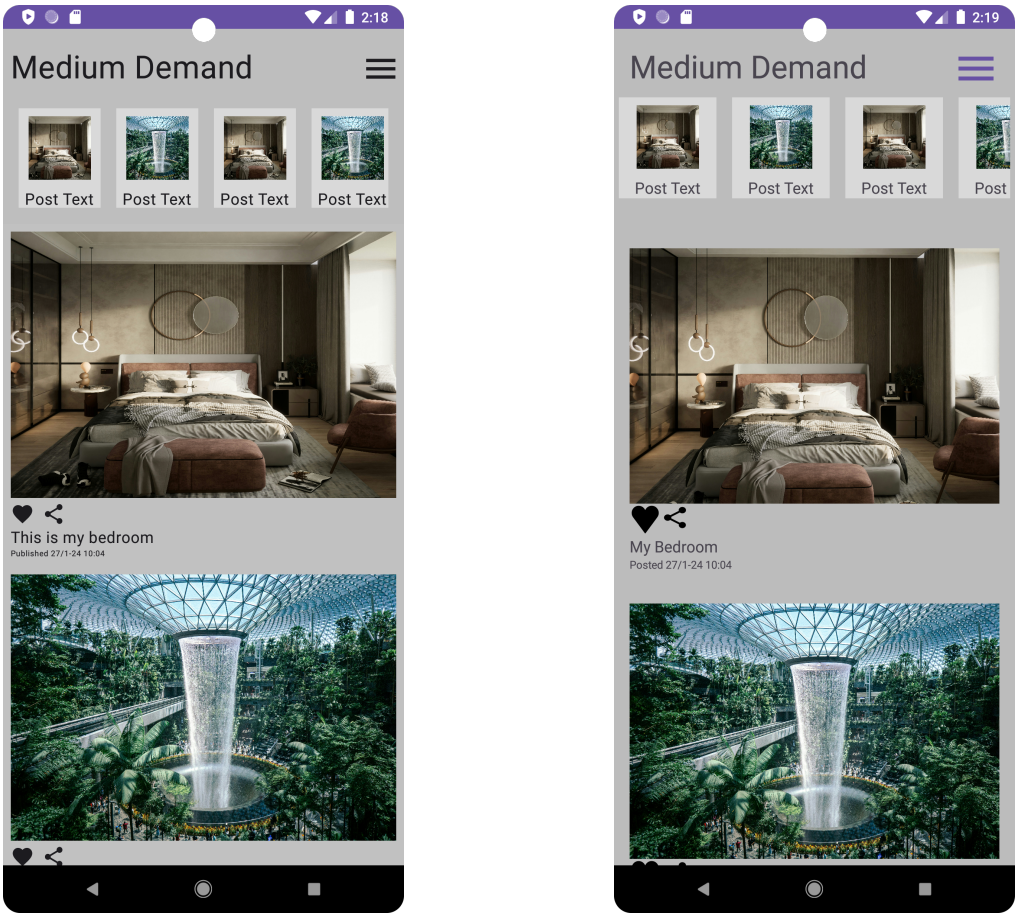
D | Applications



(a) Jetpack Compose

(b) XML Views

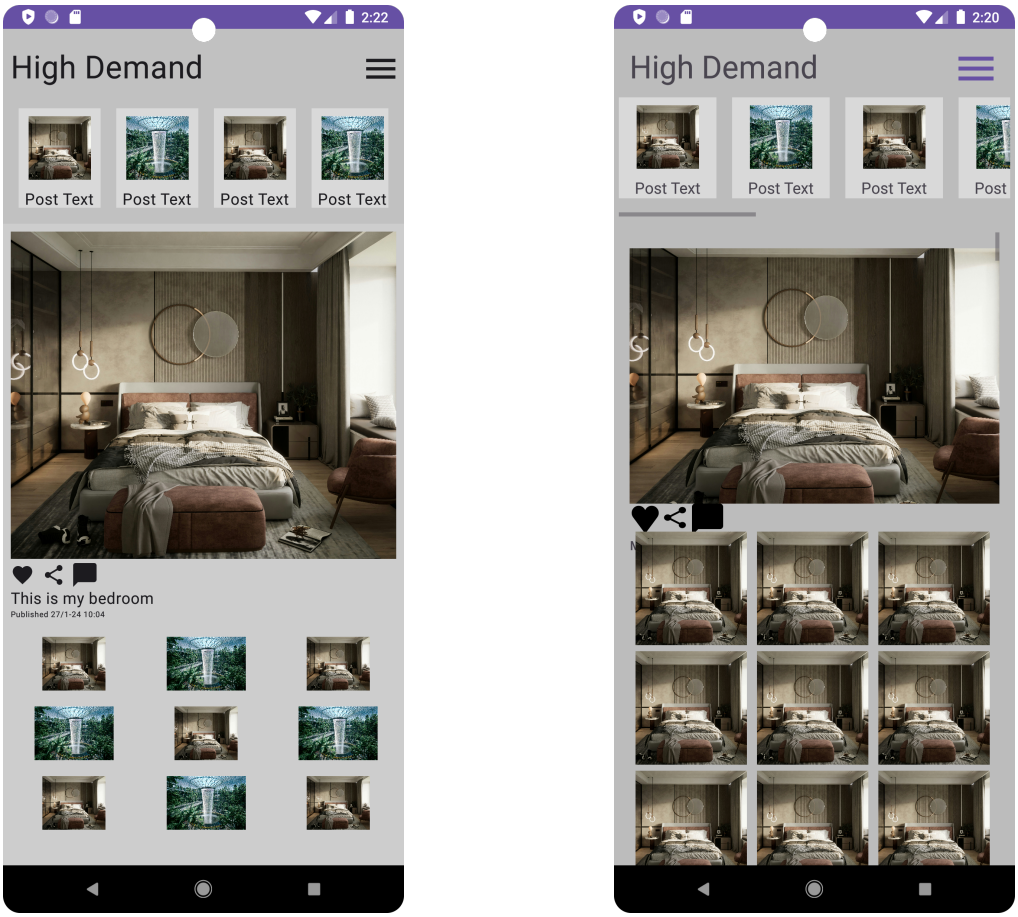
Figure D.1: Low Demand Application written in *Jetpack Compose* and *XML Views*



(a) Jetpack Compose

(b) XML Views

Figure D.2: Medium Demand Application written in *Jetpack Compose* and *XML Views*



(a) Jetpack Compose

(b) XML Views

Figure D.3: High Demand Application written in *Jetpack Compose* and *XML Views*

E | Figures

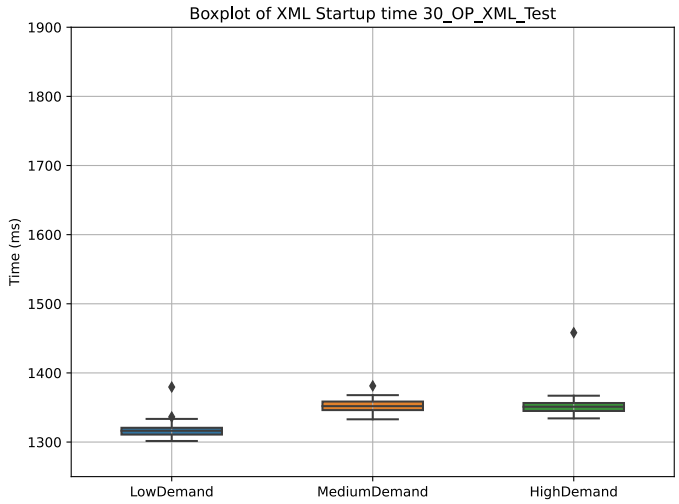


Figure E.1: 30 startup iteration of XML on the device Oneplus

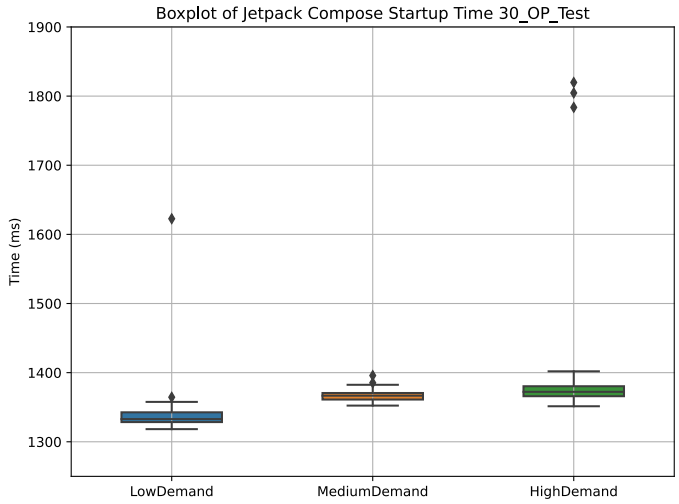


Figure E.2: 30 startup iteration of Jetpack Compose on the device Oneplus

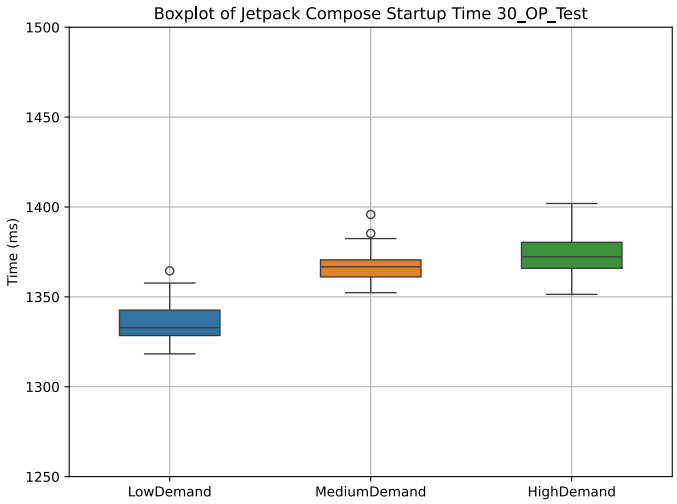


Figure E.3: 30 startup iteration of Jetpack Compose on the device Oneplus excluding outliers

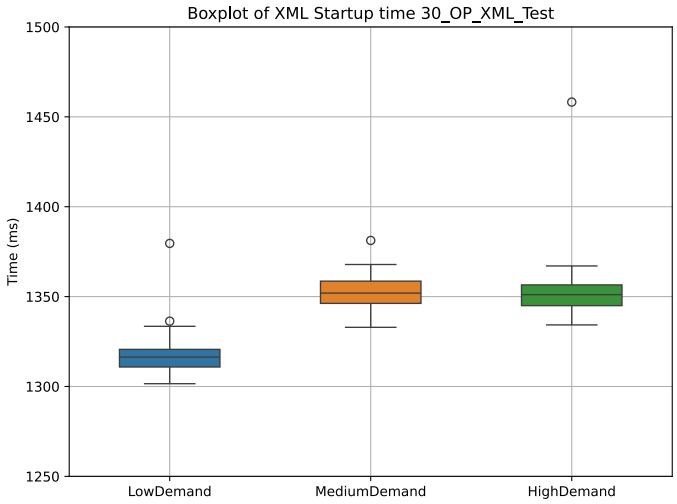


Figure E.4: 30 startup iterations of XML on the device Oneplus excluding outliers

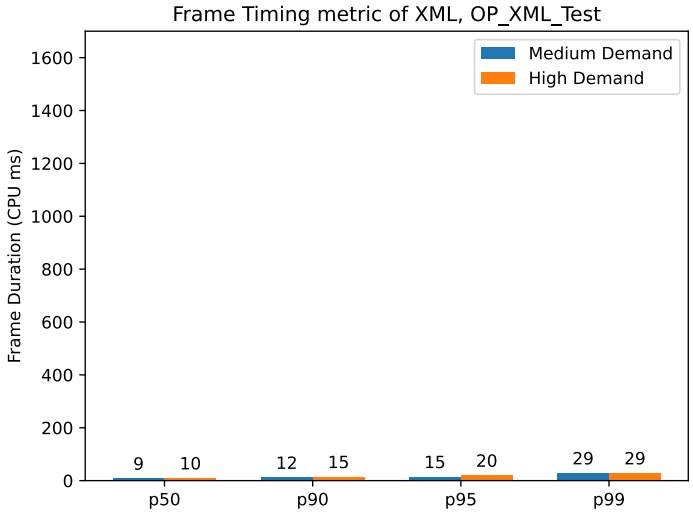


Figure E.5: 100 FTM iteration of XML on the device Oneplus

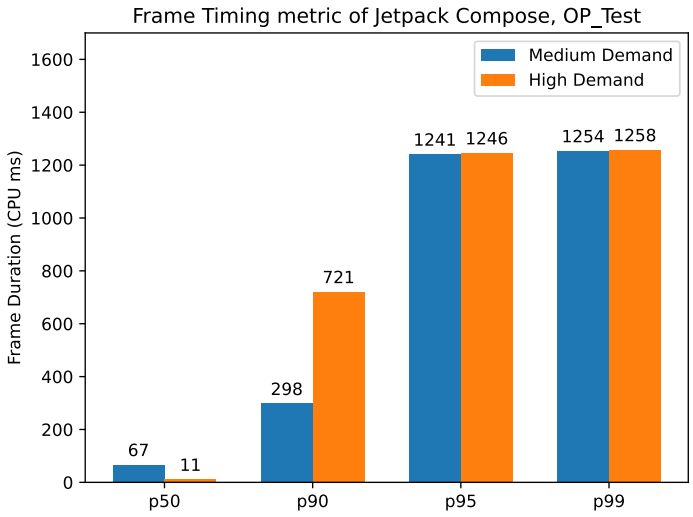


Figure E.6: 100 FTM iteration of Jetpack on the device Oneplus

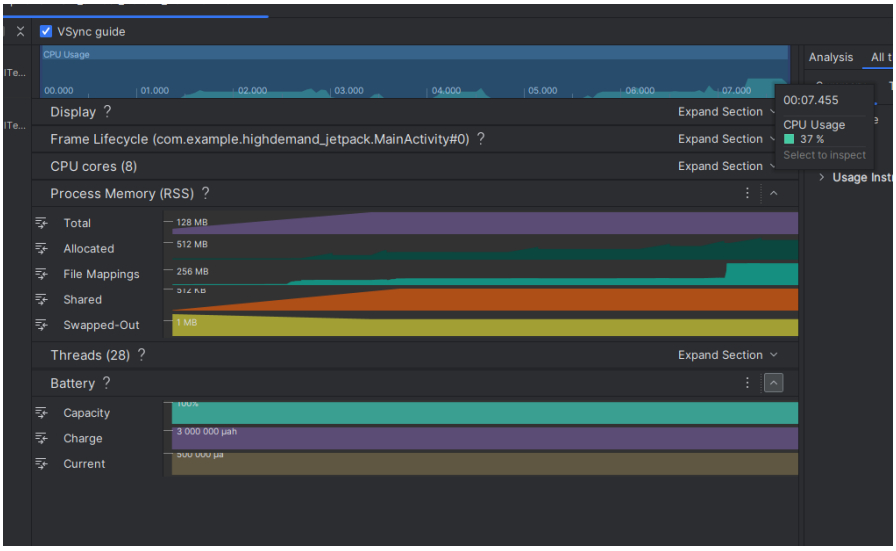


Figure E.7: Example output of Android profiler

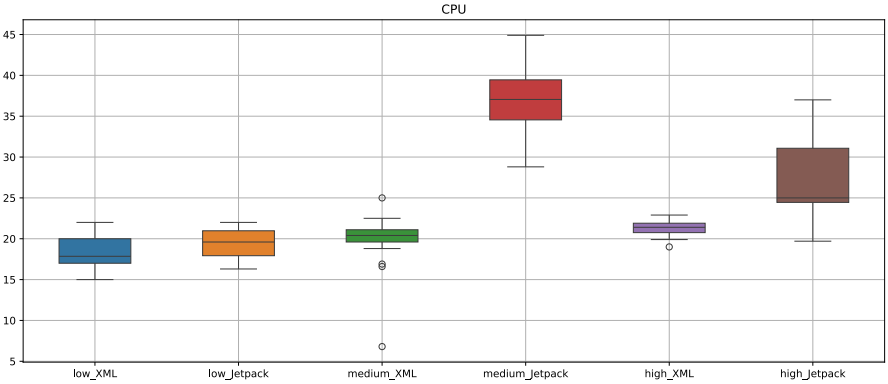


Figure E.8: 30 Iteration between each demand collecting CPU data

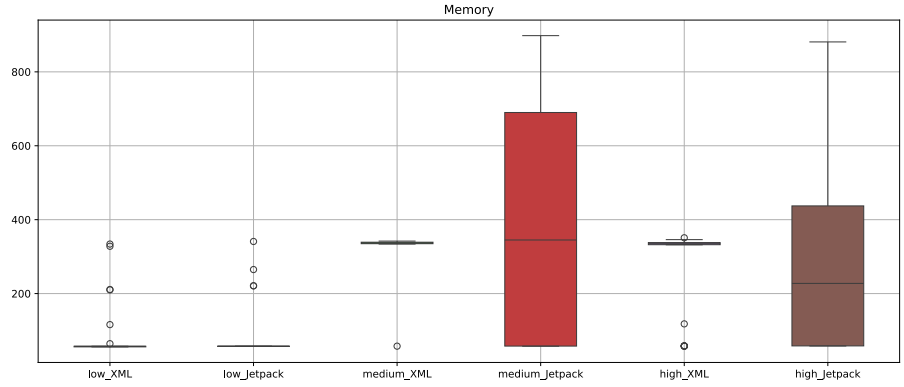


Figure E.9: 30 Iteration between each demand collecting memory data

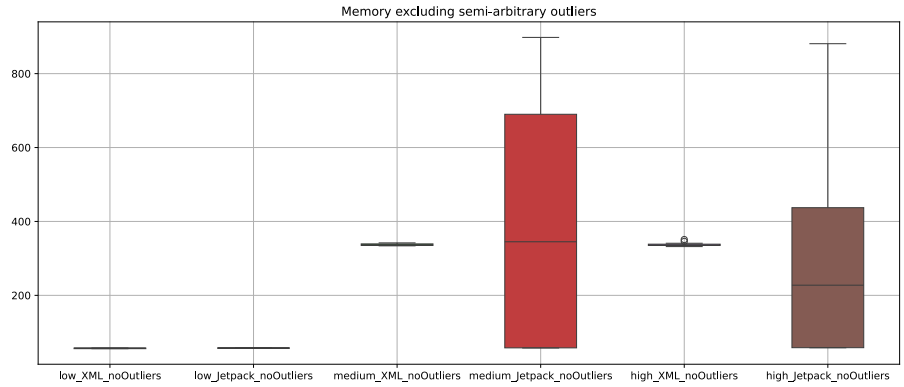


Figure E.10: 30 Iteration between each demand collecting memory excluding semi-arbitrary outliers data

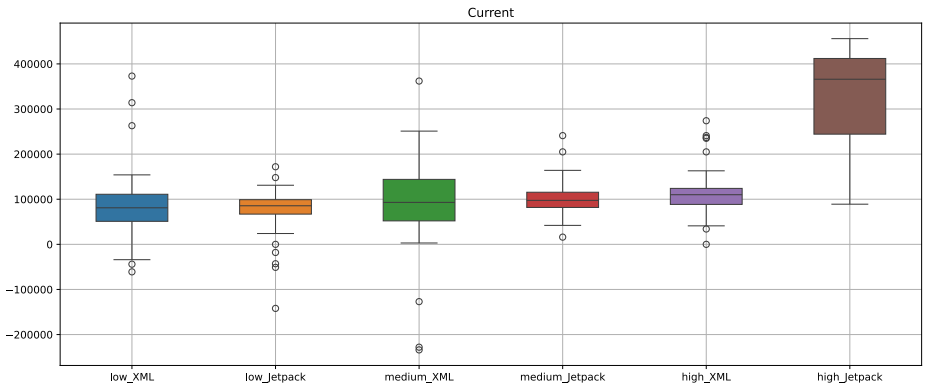


Figure E.11: 30 Iteration between each demand collecting battery current data

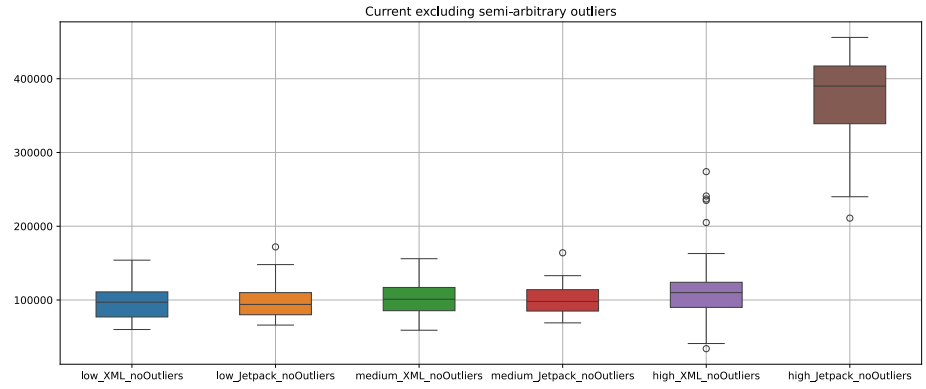


Figure E.12: 30 Iteration between each demand collecting battery current data excluding semi-arbitrary outliers data