Concurrency in Android development – Kotlin Coroutines and RxJava

Master's degree in Computer Engineering - Mobile Computing

Guilherme Abreu Almeida

Leiria, October of 2020

Concurrency in Android development – Kotlin Coroutines and RxJava

Master's degree in Computer Engineering - Mobile Computing

Guilherme Abreu Almeida

Dissertation under the supervision of Professor Catarina Isabel Ferreira Viveiros Tavares Dos Reis and Professor Ricardo Filipe Gonçalves Martinho.

Leiria, October of 2020

Originality and Copyright

This dissertation/project report is original, made only for this purpose, and all authors whose studies and publications were used to complete it are duly acknowledged.

Resumo

Um sistema de concorrência defeituoso pode ter um impacto na experiência do utilizador de um software e, consequentemente, na empresa proprietária desse software. O principal objetivo desta pesquisa é compreender o impacto de sistemas concorrentes no desenvolvimento de aplicações Android e ajudar os programadores/empresas a discretizar as melhores abordagens para a concorrência.

A pesquisa centra-se inicialmente na importância de concorrência em aplicações Android, bem como nas principais abordagens para a concorrência/*threading* no desenvolvimento de aplicações Android. Ilustra ainda porque é que algumas abordagens de programação assíncronas não se enquadram nos padrões atuais de desenvolvimento para Android. Isto permitiu que a investigação se concentrasse nas abordagens mais relevantes para a implementação de sistemas concorrentes e, consequentemente, produzir resultados mais pertinentes para o estado atual do desenvolvimento para Android.

Depois de identificar Kotlin Coroutines e RxJava como as abordagens mais relevantes para executar trabalho concorrente em Android (no momento da escrita deste documento), esta pesquisa prosseguiu com o desenvolvimento de uma aplicação de caso de estudo. Esta aplicação foi implementada com Kotlin Coroutines e RxJava com o objetivo de reutilizar o máximo de código possível para os componentes da aplicação que não necessitam de concorrência. Existe um único módulo dedicado à interface gráfica principal da aplicação e dois módulos (um para Kotlin Coroutines e outro para RxJava) dedicados a executar, com concorrência, os passos necessários para executar cada funcionalidade e a propagar os dados necessários para a interface do utilizador. Isto permitiu uma separação clara do código específico para executar as mesmas funcionalidades com Kotlin Coroutines e RxJava, facilitando a sua comparação posterior. O design desta aplicação e das suas funcionalidades requereram uma avaliação prévia de casos de uso comuns para concorrência em Android.

Com a intenção de avaliar o impacto da utilização de Kotlin Coroutines e RxJava em aplicações Android, apurámos os principais atributos de qualidade do software a considerar para o desenvolvimento de aplicações Android. Assim, fomos capazes de nos focar principalmente no Desempenho e Manutenção de uma aplicação Android e entender como o uso de Kotlin Coroutines e RxJava afeta estes atributos.

O impacto de cada biblioteca de concorrência no desempenho e manutenção de uma aplicação Android foi medido através de métricas de software que foram fornecidas por uma combinação de testes de análise estática, benchmarks e profiling. O planeamento do conjunto de testes, a criação das ferramentas necessárias e o desenvolvimento do ambiente de teste para esta investigação também são explorados neste documento.

Os resultados de Kotlin Coroutines e RxJava foram então ilustrados, comparados e interpretados para cumprir o nosso objetivo de entender se, no momento da escrita deste documento, existe uma abordagem mais sensata para concorrência no desenvolvimento de aplicações Android de acordo com o nosso conjunto de testes.

Os resultados do nosso conjunto de testes para a nossa aplicação de estudo de caso revelaram que RxJava e Kotlin Coroutines não comprometem de forma diferente o desempenho e a manutenção de uma aplicação Android, pelo que os programadores e empresas não devem estar limitados na escolha entre estas duas bibliotecas.

Abstract

A faulty concurrency system may have an impact in the user experience of the software product and consequently to the company that owns that product. The main goal of this research is to understand the impact of concurrency in Android development and further help developers/companies to discretise the best approaches for concurrency.

The research initially centres on the importance of concurrency in Android applications as well as the main approaches for concurrency/threading in Android development. It further illustrates why some asynchronous programming approaches do not fit modern Android development. This allowed the research to concentrate on the most relevant approaches to concurrency and consequently produce more pertinent results for the current state of Android development.

After acknowledging Kotlin Coroutines and RxJava as the most relevant approaches to concurrency for Android (at the time of writing this document), this research moved on with the development of a case study application. This application was implemented using both Kotlin Coroutines and RxJava while reusing as much code as possible. There is a single module dedicated to the main user interface of the application and two modules (one for Kotlin Coroutines and one for RxJava) dedicated to concurrently run the necessary steps for each feature and further propagating the necessary data to the user interface. This allowed a clear separation of the specific code needed to perform the same features with Kotlin Coroutines and RxJava, facilitating its later comparison. The design of this application and its features required prior assessment of common use cases for concurrency in Android to form a fitting case study.

With the intent of assessing the impact of using Kotlin Coroutines and RxJava in Android applications, we discretised the main software quality attributes to consider for Android development. By taking this step, we were able to focus mainly on the Performance and Maintainability of an Android application and understand how the usage of both Kotlin Coroutines and RxJava affects these attributes.

The impact of each library in the performance and maintainability of an Android application was measured using software metrics that were provided by a combination of static analysis, benchmarks, and profiling tests. The process of designing the set of tests, setting up the required tools and the overall development of the test environment for this research is also explored in this document.

The results for Kotlin Coroutines and RxJava were then illustrated, compared, and interpreted to fulfil our objective of understanding if, at the time of writing this document, there is a more sensible approach to concurrency for Android development according to our set of tests.

The results for our set of tests and case study application revealed that RxJava and Kotlin Coroutines do not differently compromise the performance and maintainability of an Android application, for what developers and companies should not be limited when choosing between these libraries.

Keywords: Kotlin, Coroutines, RxJava, Concurrency, Android, Metrics

Contents

Originality and Copyright ..... **iii**

Abstract ..... **v**

List of Figures ..... **viii**

List of Charts ..... **ix**

List of Abbreviations and Acronyms ..... **x**

1. Introduction **1**

1.1. Motivation **2**

1.2. Objectives and Questions **3**

1.3. Document structure **4**

2. Thread management in Android **5**

2.1. Concurrency issues in Android **5**

2.2. Approaches to concurrency **6**

2.2.1. AsyncTask, IntentService, HandlerThread and CompletableFuture 7

2.2.2. RxJava 11

2.2.3. Kotlin Coroutines 12

2.3. Metrics and Tools **15**

2.3.1. Maintainability 17

2.3.2. Performance 18

2.4. Summary **19**

3. Methodology **21**

3.1. Research and Case study **22**

3.2. Experimentation **24**

3.3. Analysis and Conclusions **25**

4. Case study application **26**

4.1. Functional requirements **26**

4.2. Non-functional requirements **27**

4.3. Architecture **29**

4.4. Class Structure 33

4.5. Summary 35

5. Test Cases Design and Environment Setup 36

5.1. Test Cases Design 36

5.1.1. Static Analysis 36

5.1.2. Performance Analysis 36

Network tests 37

Database tests 37

Integration tests 38

5.2. Environment setup 39

5.2.1. Static Analysis 39

5.2.2. Performance Analysis 40

Test device 40

Mocking the network layer 41

Benchmarking 44

Profiling 47

6. Results 51

6.1. Static analysis 51

6.2. Performance Analysis 54

6.2.1. Network 54

6.2.2. Database 56

6.2.3. Integration 57

6.3. Summary 61

7. Conclusion 62

References 65

List of Figures

Figure 1. ISO/IEC 25010 attributes for software quality evaluation. 16

Figure 2. Overview of the research process. 21

Figure 3. Research and Case study phase steps. 22

Figure 4. Experimentation phase steps. 24

Figure 5. Analysis and Conclusions phase steps. 25

Figure 6. Case-study application features. 27

Figure 7. The MVVM pattern. 27

Figure 8. Overall case-study architecture. 29

Figure 9. Location request close-up. 30

Figure 10. Address request close-up. 31

Figure 11. Movie data request close-up. 32

Figure 12. Class diagram for main components. 34

Figure 13. SonarQube Measures dashboard. 39

Figure 14. Mocked network flow overview. 43

Figure 15. Enabling the garbage collector item. 48

List of Charts

Chart 1. Lines of code metric results. 52

Chart 2. Cognitive Complexity (left) and Cyclomatic Complexity (right) metric results. 52

Chart 3. Network benchmark results. 55

Chart 4. Database benchmark results. 56

Chart 5. Integration benchmark results. 57

Chart 6. Trending movies profiling results. 59

Chart 7. Local movies profiling results. 60

List of Abbreviations and Acronyms

ANR	Application Not Responding
API	Application Programming Interface
CPU	Central Processing Unit
DAO	Data Access Object
DTX	Database Test number “X”
DX	Developer Experience
ESTG	School of Technology and Management
GUI	Graphical User Interface
HTTP	Hypertext Transfer Protocol
IBTX	Integration Benchmark Test number “X”
IDE	Integrated Development Environment
IO	Input/Output
IPTX	Integration Profiler Test number “X”
JVM	Java Virtual Machine
LOC	Lines of Code
MVVM	Model-View-ViewModel
NTX	Network Test number “X”
ORM	Object Relation Mapping
OX	Objective number “X”
RQX	Research Question number “X”
UI	User Interface

1. Introduction

Developers are aware that implementing high resourceful tasks such as accessing remote services is often mandatory to comply to their applications' functionality but performing these tasks while also maintaining a pleasant user experience can prove to be a challenge.

Commonly known as processor, the Central Processing Unit (CPU) is the hardware component used by devices to execute compiled code. To instruct the CPU to execute code, developers must make use of *threads* in their software. Threads work as a mediator between the developer and the processor, consider them envelopes which transport code to the CPU to be executed. They are inevitable and play quite a significant role in software development. Developers can execute their software synchronously using only one thread, but most modern applications will most likely take advantage of multiple threads to efficiently complete their applications' functionality. This is especially true when applications feature a Graphical User Interface (GUI), where there is often a thread solely dedicated to render visual components and intercept user input.

The process of segregating work to multiple threads is commonly referred to as *concurrency* [1]. Dealing with concurrency also implies handling communications between the different threads, managing their lifecycle and anticipating all the problems that might arise from simultaneous accesses to the same memory space by different threads, like inconsistent reads or lost updates.

Despite being useful resources, threads should not be created at will. They are heavy resources which will, when exceedingly instantiated, take a hit on the allocated memory to run the software in hand. Hitting any memory-related limit will most likely crash the process in which the software is running on. These kinds of limits are most certainly important to avoid in any software product.

In spite of some programming languages having a transparent approach to concurrency where the developers are exposed to very low-level concurrency components, manually handling threads will most likely increase the complexity of the source code and the chance for unexpected errors and degraded performance.

To implement concurrency systems, developers often delegate thread management to certain third-party libraries. Such libraries allow the developer to focus on *what* and *where* to execute by isolating the *how* to execute. For instance, if developers want to run two tasks simultaneously, they should be able to state that through simple code instructions. Libraries can then take those tasks and delegate them to separate threads that are later executed by the processor, acting as mediators between developers and low-level systems.

For each software application type (e.g. server applications, mobile applications), there are standard thread management libraries that help to achieve concurrency in the best way possible. Choosing a thread management library for a project will influence the required skillset to maintain this project and might have consequences beyond the development team, since a poor concurrency system is easily noticed by end-users as well.

Android applications launch by default with a single thread, the main thread. When the user interacts with the application, it is the main thread the one responsible for updating the user interface and handling user events like clicks and swipes. If an Android developer clutters the main thread with lengthy tasks like accessing a remote service or making calls to a database, the main thread will block waiting for these requests and visually, the application will stall. This happens because the main thread alone cannot execute multiple tasks simultaneously and will most likely hurt the user experience.

To tackle this problem, developers can take advantage of concurrency techniques and deviate these lengthy tasks to be executed in worker threads. After a computation is done in a worker thread, developers can then show the result in the main thread, if needed. Following this practice, the main thread will be free of lengthy tasks and will be able to draw user interface components at a higher framerate, maintaining a pleasant user experience [2].

This research aims to explore the libraries that are available for Android developers to successfully implement concurrency onto their applications. It intends to objectively evaluate which of those libraries best fit their projects, considering the current trends in modern Android development. To understand how the usage of these thread management libraries influence the quality attributes of an Android application, this research will evaluate specific software quality metrics for a case study application, custom-made for this purpose, that uses different thread management libraries to enable concurrency.

After comprehending how using these different libraries influence the quality attributes of an Android application, it is possible to gauge if there is a more reasonable thread management approach, when compared to alternatives.

This chapter will further provide the motivation for this research as well as the formulated questions and objectives that it aims to achieve.

1.1. Motivation

Nowadays, mobile applications, as multifaceted systems, are expected not only to serve a graphical user interface (GUI) but also to do a lot of lengthy work such as consuming remote services, persisting data in a local database or reading values from sensors on the device. Since lengthy work should not be processed in the main thread, it is expected to run in worker threads, concurrently, and then get back to the main thread with the processed result.

Heavy work on the main thread has been stated as one of the main Android performance bugs and UI unresponsiveness has been highlighted as a top complaint by mobile application users [3,4].

There has been significant effort put into designing libraries meant to help developers with concurrency, some of them exclusive to Android [5–11]. Goes to show that concurrency is a relevant theme for developers and consequently to the resources behind the development of mobile applications.

Not delegating concurrency to a well-tested and proven library can lead to serious issues, but those can also occur with unproper use of these libraries. Choosing the right library to perform concurrent

work is certainly a decision that should not be lightly taken as later changes to a concurrency system will require a lot of attention and resources.

The initially chosen concurrent system can also influence the evolution of a project since it will impact the required skillset. For instance, if the concurrency system used by a project is not aided by well-known libraries it may be harder to allocate human resources that can efficiently understand and implement upon this system.

Driven by professional experience, we understand that different approaches to concurrency may have further impact beyond the codebase, such as negatively impacting the user experience and possibly limiting the application's modifiability for upcoming features or older functionalities that must be revised.

Concluding on how the use of the different libraries influences software quality and the overall environment surrounding mobile software will not only help developers but also other related stakeholders to prioritise the construction of a reliable concurrency system.

1.2. Objectives and Questions

In the context of concurrency and thread management that developers must address in their application development this research will answer to the following research questions (RQ):

RQ1. What makes a thread-management library preferred over its alternatives in Android modern development?
RQ2. How can the usage of thread management libraries affect the software quality attributes of an Android application?
RQ3. What kind of impact can the use of thread management libraries have outside the codebase?
RQ4. What is the most pertinent way to compare different thread-management libraries, contemplating the current standards in Android development?
RQ5. Is there, at the time of writing of this document, an objectively better thread-management library for Android development considering its impact on the overall software quality?

Thus, considering the work that needs to be conducted to provide an adequate answer to the above-mentioned questions, this research has the following objectives (O):

O1. Understand why some asynchronous programming approaches do not fit modern Android development.
O2. Gather common use cases for concurrency in Android.
O3. Build a sample Android application that uses standard thread management libraries to implement the previously gathered use cases.
O4. Run tests against non-functional requirements by applying quantifiable metrics to the sample application.
O5. Find a proper way to provide the analysis result to the end reader.
O6. Interpret and conclude on the results achieved.

O7. Make an analysis on what kind of impact the results may have to the people involved in software production and end-users.

1.3. Document structure

This document follows with an exploration of previous academic work and industry practices regarding the topics of concurrency in Android and software quality metrics in Chapter 2. In Chapter 3, the main phases of the methodology are illustrated alongside an explanation of the steps that compose them. Chapter 4 is dedicated to the case study Android application that was used in this research. It mentions the main requirements for a valid case study and further details the application design, architecture, and implementation. Chapter 5 illustrates and explains the test design for both the static analysis and performance tests and the main compromises that had to be made to the case study application to extract reliable results out of the performed tests. Chapter 6 presents the results of this research and analyses the several metrics that were tested, on a technical level. Finally, in Chapter 7 the main conclusions for this research are presented and future work is explored.

2. Thread management in Android

This chapter explores how concurrency can influence the user experience on Android applications and which tools are available for Android developers to implement a proper concurrency system onto their applications. After presenting these tools, the chapter reflects on how their usage can affect different quality attributes of the final software product.

As the most relevant quality attributes (that can be influenced by the used threading library) for mobile software are established, this chapter shows how they can be measured objectively by using software metrics. Some tools that help achieving quantitative data regarding those metrics are also mentioned.

This research will then propose a comparison between the standard threading libraries for Android development by analysing how they influence the most relevant quality attributes of an Android application.

2.1. Concurrency issues in Android

Android applications run, by default, in a single thread (commonly referred to as *the main thread*). This thread is used to dispatch UI (User Interface) events to the interface components and render the views of those components; this is why it can also be called the UI thread [2]. Clearly, this thread has enough work to do on its own and any blocking work should be done away from it. When this thread blocks, it will be visible to the user and it can end up with the infamous Application Not Responding (ANR) alert that will encourage the user to close the application.

To prevent this, Android developers should consider a non-blocking architecture based on ensuring that all the blocking work related code never runs in the main thread. This includes network operations, database accesses and bitmap processing, among others. Ideally, blocking work should be done in worker threads, asynchronously, and only if needed, present the result in the main thread. Thus, the main thread remains clean and always ready to respond to user events.

A study about what users complain the most on mobile applications [4] shows that out of the 12 most common complaints there are at least 2 non-functional ones that could be related to poor concurrency systems:

- Unresponsive App
- Resource Heavy

Unresponsive App is the exact problem mentioned above. Applications are doing too much work on the main thread and when this thread blocks, the application stalls and cannot process any user event. If this stall lasts too long, the user will be prompted by the operating system to close the application through an ANR alert. Despite the possibility of running blocking work on worker threads, many developers are severely impacting the user experience by not doing so.

An application being resource heavy might be due to poor management of asynchronous work. Even when an application properly delegates work across multiple threads, mindful management of such threads is still essential. If resource heavy work is not stopped when no longer needed, such work will be unnecessarily consuming resources, which can lead to slow performance and, in the worst-case scenario, application crashes. For mobile devices, applications that leak resource heavy work may also drain the device battery faster which affects the user outside the application scope.

According to an analysis on performance in Android applications [3], there were 3 types of bugs that defined roughly 94% of the 70 performance issues found, with some of them fitting in more than one type (e.g. one bug can cause both energy leaks and memory bloat):

- GUI lagging – 53/70 = 75.7 %
- Energy leaks – 10/70 = 14,3 %
- Memory bloat – 8/70 = 11.4%

The bugs found in this study very much corroborate the previous study conclusions. Graphical User Interface lagging is a possible cause for the “Unresponsive App” complaint mentioned above, and energy leaks and memory bloat will most likely result in what users can comprehend as “Resource Heavy”.

The architectural design of a software product, ensuring a well-structured and scalable concurrency system, may have a very positive impact for the end-user and, eventually, to the business surrounding the application.

2.2. Approaches to concurrency

To solve common issues regarding concurrency, there has been significant efforts put into developing libraries that abstract thread management from the software developer. Thread management can be very error-prone and, therefore, being able to delegate this work to tested libraries is something that development teams should consider. Nevertheless, there is still the need to learn how and when to use such libraries. This is where developers must commit to explore deeply whatever solution they choose for concurrency, otherwise, its misuse can lead to issues they initially meant to fix.

This section will cover the evolution that those concurrency libraries have suffered until the present time. It will start by mentioning the initial effort by the Android team to create exclusive concurrency solutions inside the Android framework and explain why they did not survive to developers’ needs, as time passed by and simpler alternatives emerged.

It will explore how CompletableFuture [9], a component provided by the Java concurrency library could improve some of the issues regarding precedent alternatives, like simpler task sequencing and threading abstraction, but could not, however, live to fulfil the capacity of implementing reactive systems.

This is where RxJava [10] came in and took the leadership on asynchronous programming in Android development. RxJava provides better task sequencing, threading abstraction and has a reactive nature that effortlessly made it the best tool to build reactive systems. It is an open-source solution which

allowed it to grow organically with the developers' needs and adapt better to new paradigms when compared to any closed-source alternatives.

Recently, Kotlin [12] was announced as an official language for Android development, and with it came a set of new features that were not available with Java. As Kotlin evolved, there was the need to build concurrency primitives for the language, giving birth to the Kotlin Coroutines library [11]. Like RxJava, it can help developers with both asynchronous programming and reactive systems. Since Kotlin Coroutines brings a paradigm shift that is not based on callbacks, like the previously mentioned tools, it will have its own section to be explored.

This chapter illustrates the differences between four main approaches to concurrency in Android development, by showing the same example of a concurrent task written with AsyncTask, CompletableFuture, RxJava and Kotlin Coroutines. This will hopefully uncover some of the reasons why developers started to have preferences amongst all these available options.

2.2.1. AsyncTask, IntentService, HandlerThread and CompletableFuture

The Android team proposed their own asynchronous programming approaches that include: AsyncTask [5], IntentService [7] and HandlerThread [8].

Out of these, AsyncTask was the best solution for most concurrency problems as it wraps all the process of delegating work to a worker thread and then, once the work is over, getting back the result to the main thread [13].

IntentService provides a good separation of concerns since any child of this class is supposed to do only one type of work and this work is automatically delegated to a worker thread. However, with IntentService, the means of communication with the main thread require some boilerplate code and a deeper knowledge of the Android framework that makes it overall a more complex solution when compared with AsyncTask, for instance. To get back to the main thread with the result from an IntentService, this service will broadcast an Intent [14] with the operation result. The target UI component must register a BroadcastReceiver [15] to catch this broadcast, further process it and finally display it (if needed). AsyncTask can conceptually do the same in a more contained form and without the need for multiple components, making it slightly easier to grasp on.

HandlerThread is a subclass of Thread [16], with an implementation similar to a normal thread that keeps the thread awake by using an event loop. The use of a HandlerThread only requires to instantiate it and post work under the form of Runnable [17] to its Handler [18]. The work will run in a new thread and not on the main thread, as intended. Developers might appreciate its simplicity when compared with the IntentService approach, however, the problem remains when the result is needed in the main thread. When developers intend to propagate the result from an HandlerThread to the main thread, there is still the need to get a Handler to run in the main thread and run some sort of callback in this handler from the worker thread. This approach requires a deeper knowledge of concurrency primitives on the Android framework, making it not as high-level as AsyncTask or even IntentService.

All these approaches can abstract thread management at some level but AsyncTask took the lead as it was the most used approach to asynchronous programming in Android development until RxJava emerged.

By default, AsyncTask and IntentService both implement a queue of work and they process each task one at a time. This means that they cannot do parallel work, and that if one task takes too long, the ones waiting for it to finish may have a severe delay to reach the UI. AsyncTask can be further customised to use a ThreadPoolExecutor [19] [5] and this would allow for parallelism since the work would now be dispatched to more than one thread. In case developers choose to create their own thread pool and manage it, this will strive away the abstraction they were initially looking for.

Regarding the use of thread management libraries there are two studies [20, 21] that analysed a group of Android applications and concluded on what kind of asynchronous constructs they were using in order to run work outside the main thread and how they could help developers doing it properly.

AsyncTask was the most common construct followed by Java's Thread [22] standard construct. In the first study [20] the authors managed to create a refactor tool that would help developers transform synchronous code into asynchronous code by placing the boilerplate code of an AsyncTask around the block of code the developer intended to run in a worker thread. In the second one [21] they managed to build yet another refactoring tool, but aiming to transform AsyncTasks into IntentServices or AsyncTaskLoaders [6], since many developers held UI references in AsyncTasks and did not cancel them properly, leading to memory leaks and application crashes.

AsyncTaskLoader does the same as AsyncTask but it handles configuration changes properly by not duplicating the same AsyncTask when it is recalled inside the same context. AsyncTaskLoader uses AsyncTask internally and works better inside UI components like activities or fragments, however, AsyncTask can be used for several contexts and that versatility is why it remained the most used solution for so long.

At the moment of writing of this document, AsyncTask and AsyncTaskLoader are already deprecated [6, 23], and the IntentService seems to be on its way to the same end [24].

All the resources previously mentioned require too much boilerplate to simply dispatch one task off the main thread, but they show their worst form when the developer needs to sequence tasks, e.g. having task A and task B, the developer wants to start task B when task A finishes because task B relies on the result from task A.

Using the resources mentioned above to sequence multiple tasks will result in what is commonly called the "callback hell", which is when tasks are nested inside other tasks, using callbacks. This usually results in multiple indentation levels which harm the readability and understandability of the code.

From here on, the main libraries being discussed will feature the same code example to help draw a timeline of the evolution these tools went through. The example is a simple network call to fetch a list of users from a remote endpoint. The network request must be done in a background thread whereas the handling of the result must be done in the main thread. This is being done in a view model class, the standard place to fetch data from in Android modern development, according to the available best practices' guidelines for Android Development [25].

```
class AsyncTaskViewModel : CommonViewModel() {

    private val asyncTasksBag: MutableList<AsyncTask<*, *, *>> = mutableListOf()

    override fun fetchUsers() {

        val asyncTask = FetchUsersAsyncTask(networkClient) { userContainer ->
            val list = userContainer.list
            if (list != null) {
                onUsersFetchSuccessful(list)
            } else {
                onUsersFetchError(userContainer.throwable!!)
            }
        }

        asyncTask.execute()
        asyncTasksBag.add(asyncTask)
    }

    override fun onCleared() {
        asyncTasksBag.forEach { it.cancel(true) }
        super.onCleared()
    }

    class FetchUsersAsyncTask(private val networkClient: NetworkClient, val callback:
(UserContainer) -> Unit) : AsyncTask<Unit, Unit, UserContainer>() {

        override fun doInBackground(vararg args: Unit): UserContainer {
            val result = runCatching { networkClient.fetchUsers() }
            return UserContainer(result.getOrNull(), result.exceptionOrNull())
        }

        override fun onPostExecute(result: UserContainer) {
            super.onPostExecute(result)
            callback(result)
        }
    }

    data class UserContainer(val list: List<String>?, val throwable: Throwable? = null)
}
```

Listing 1. AsyncTask example.

The example featured in Listing 1 represents the network request using an AsyncTask. In this case there is no sequencing of tasks, there is only one callback that runs when the AsyncTask is done. It is, however, perceptible that having to execute AsyncTasks inside the callback of another AsyncTask will lead to very complex and intricate code - the “callback hell” as mentioned above.

The “callback hell” problem was later solved by CompletableFuture [9], an evolution over the early Future [26] which had a lot of limitations when it comes to threading. For instance, the very simple interface Future exposes a *get*() method that will block the caller thread if the work it is intended to do is not yet done, making Future unusable on the main thread of an Android application for example.

CompletableFuture fixed both the threading and the sequencing problem. With CompletableFuture, developers could chain tasks together using callbacks, but this time, with minimal indentation levels, preferably one only. More importantly, CompletableFuture could sequence tasks that depended on each other and individually dispatch them to different threads by using the *async* methods its interface provides.

```
class CompletableFutureViewModel(application: Application) :
CommonAndroidViewModel(application) {

    private val futureBag = mutableListOf<CompletableFuture<*>>()

    override fun fetchUsers() {

        val mainThreadExecutor = ContextCompat.getMainExecutor(getApplication())

        CompletableFuture
            .supplyAsync { networkClient.fetchUsers() }
            .thenApply { Result.success(it) }
            .exceptionally { Result.failure(it) }
            .thenAcceptAsync(Consumer { result ->
                if (result.isSuccess) {
                    onUsersFetchSuccessful(result.getOrThrow())
                } else {
                    onUsersFetchError(result.exceptionOrNull()!!)
                }
            }, mainThreadExecutor)
            .apply { futureBag += this }
    }

    override fun onCleared() {
        futureBag.forEach { it.cancel(true) }
        super.onCleared()
    }
}
```

Listing 2. CompletableFuture example.

By looking at the code needed when using CompletableFuture (Listing 2), it is possible to see a significant improvement over the AsyncTask code. However, since AsyncTask was tailored within the Android framework, the *onPostExecute*() method in its interface always runs in the main thread of the Android application, whereas the CompletableFuture version requires a deeper understanding of the Android framework in order to switch execution to the main thread.

With CompletableFuture, using any *async* method will switch the execution to a certain executor. If the developer does not pass any executor, the execution will be done in a background thread from a common thread pool (default behaviour). Therefore, to switch to the main thread, the developer must be explicit and there is no “Android way” to do it. For CompletableFuture, the main thread is just an executor like any other. Developers must either have a deeper knowledge on what an executor is and how the Android main thread can be reached or use libraries from third parties that might ease the integration of CompletableFuture with the Android framework.

CompletableFuture was introduced with Java 8 (2014), at about the same time the second edition of Reactive Manifesto came out [27]. This manifesto started to promote the idea of reactive systems, which CompletableFuture could not comply with.

2.2.2. RxJava

As the reactive paradigm started to impact a lot of areas around software development, Android was no exception. Until this paradigm shift, most of the use cases for concurrency in Android were single-shot operations. As the reactive paradigm appeared, it brought the need to modify and act upon streaming values, to be later referred as data stream operations.

A single shot is usually referred as a fire-and-forget operation. This could be a simple network request where the developer just wants to know the response of a remote service and be done with it. A data stream operation is relevant for when there is the need to react to incoming data for an undetermined period. This could be subscribing to the user's location and receiving a location every minute, so that the user interface is coherent with the user current location. Data stream operations have the concept of subscription and instead of pulling data from a data source, like a single-shot operation would, the data source will push new data to its subscribers, making the code reactive to new data.

As Android developers started to lean towards reactive systems, CompletableFuture and AsyncTask were not enough as they did not provide any easy way to integrate data stream operations, a vital part of the message-driven architecture the reactive manifesto strives for. Android developers needed a tool that could easily compose task sequences, with a good threading abstraction, and with support for not only single-shot operations but also data stream operations, fulfilling the capacity to implement a reactive system. The Netflix engineering team released RxJava [10] as an open-source library to fill in this gap.

The first version of RxJava was released back in November 2014. Being released in the same year as CompletableFuture, its reactive nature, better threading abstraction and early support for Android (through the RxAndroid library [28]) probably made this the preferred tool over any precedent for Android development. Also, being an open-source library, RxJava could grow to what developers need and had a better chance of surviving than closed-source alternatives.

RxJava implements the observer pattern [29] to support data streams and uses exclusive operators to compose task sequences [30]. To abstract low-level threading and sequencing concepts from developers, RxJava uses high-level classes like Observable, Flowable, Single and Maybe [31]. Even though these classes can be used interchangeably with each other, they serve different purposes.

The Observable/Flowable classes are used for data stream operations where the developer can generate and subscribe a stream as well as modify the values coming through that stream until it reaches its final consumer. This subscription is disposed at the developers will, while it is active, it will react to any incoming event, multiple times if intended, unlike single-shot operations that run only once. For single-shot operations, it is common to use the Single class, as it works like an Observable/Flowable, but it will only emit one value or one error to the stream.

Developers can generate chains whose operators modify the streamed values and dispatch stream events to certain thread pools; this is where the thread management abstraction comes in. With a call to the *subscribeOn* and *observeOn* operators, a developer can define which thread pool the RxJava chain should start running on and define later context switching, respectively.

At the time of writing this document, there are 454 operators available to shape RxJava chains [30]. One can see that having multiple operators is very helpful, however, having these many will also

increase the learning curve of this library as some of them do not resemble language primitives and are fully unique of this library. The same code listing example illustrated before is now shown below in Listing 3, using RxJava.

```
class RxJavaViewModel : CommonViewModel() {

    private val disposableBag = CompositeDisposable()

    override fun fetchUsers() {
        Single.fromCallable { networkClient.fetchUsers() }
            .subscribeOn(Schedulers.io())
            .observeOn(AndroidSchedulers.mainThread())
            .subscribe(
                { list -> onUsersFetchSuccessful(list) },
                { throwable -> onUsersFetchError(throwable) }
            ).apply { disposableBag.add(this) }
    }

    override fun onCleared() {
        disposableBag.dispose()
        super.onCleared()
    }
}
```

Listing 3. RxJava example.

The example above uses the previously mentioned *Single* class to run the network request (a single-shot operation). It also illustrates how developers can switch the context to the main thread by using *AndroidSchedulers.mainThread()*. This abstraction of the main thread is part of RxAndroid, a library that integrates RxJava with the Android framework.

RxJava seems to be the natural evolution of the previously mentioned libraries as it produces simpler code for concurrency and has better integration with the Android framework.

With Kotlin as an official language for Android development, developers have a new set of tools to work with. Kotlin Coroutines [11] is the library that encapsulates the concurrency primitives for Kotlin. Kotlin Coroutines can be used for Android development and can conceptually do the same as RxJava. In the next section we provide a deeper exploration on how Kotlin Coroutines work, the paradigm shift it brings and what kind of integrations it has with the Android framework.

2.2.3. Kotlin Coroutines

In 2017, the Android team announced Kotlin as an official language to write Android applications [32]. Kotlin is interoperable with Java, which eases the transition to Kotlin and allows for developers to take advantage of Kotlin features even on older projects that were fully written in Java.

Kotlin has come up with an alternative to asynchronous programming called Kotlin Coroutines. Kotlin Coroutines version 1.0.0 was released on the 29th October 2018 [33].

Kotlin Coroutines are based on coroutines [34], a former technology brought up by Melvin Conway back in July 1963.

Even though they are an alternative to regular threads, coroutines still use threads to run work on a lower level. The key feature of coroutines is that one thread can run multiple coroutines and these coroutines can suspend whenever they are waiting for a lengthy operation to finish, such as, for instance, a network request. Suspending coroutines will not block the thread they are running on (unless explicitly requested to do so). When a coroutine suspends, it offers the thread it is running on back to the thread pool to execute other coroutines, not wasting any thread time. When the coroutine is ready to continue, after suspension, it can be picked up by any thread available, not necessarily the one that picked it up initially.

Regarding the use of high-level classes like *Continuation* to implement coroutines, there is a good explanation on a study that goes back to 1984 titled "Continuations and coroutines" [35]. Kotlin's own implementation of coroutines has followed some of the principles described in this study. When compiling to JVM (Java Virtual Machine) bytecode, the coroutine code produced by Kotlin Coroutines is translated to a *Continuation* object that possesses the state needed to run the entire coroutine and also uses labels to understand which statement to run [36, 37].

In Kotlin, coroutines can suspend if they are running *suspending* functions [38]. Suspending functions look just like regular functions, but they are marked with a *suspend* modifier. This modifier was introduced into the language specifically to support coroutines.

Both regular functions and suspending functions will only return when the function is done, or an exception is thrown. The main difference between both has to do with resource management.

Regular functions are not optimised for lengthy operations. When a thread is running a regular function that does not return immediately, it blocks. A blocked thread cannot execute other functions, it is objectively a wasted resource.

Suspending functions cannot be invoked like regular functions. Suspending functions can only run inside other suspending functions or inside a coroutine. Kotlin Coroutines follow the concept of *structured concurrency* [39, 40], therefore, coroutines *require a CoroutineScope* to be launched. This CoroutineScope helps preventing memory leaks and provides developers an overall better way to manage asynchronous work.

Listing 4 represents the previously used example written with Kotlin Coroutines.

```
class CoroutinesViewModel : CommonViewModel() {

    override fun fetchUsers() {
        viewModelScope.launch {
            try {
                val list = withContext(Dispatchers.IO) { networkClient.fetchUsers() }
                onUsersFetchSuccessful(list)
            } catch (throwable: Throwable) {
                onUsersFetchError(throwable)
            }
        }
    }
}
```

Listing 4. Kotlin Coroutines example.

The example shows an integration with the Android framework where every view model class contains a *viewModelScope* ready to use and cancellation happens under the hood. The *viewModelScope* is a *CoroutineScope* that launches coroutines on the main thread by default. This scope is cancelled (it cancels every coroutine started inside it) when the view model is cleared.

Kotlin Coroutines supports both single-shot operations and data stream operations. Single-shot operations are easily achieved with simply launching a coroutine and dispatching the work to a specific thread/ thread pool. If the operation is meant to be lengthy it can probably benefit on the use of suspending functions (it is not mandatory since coroutines can run both regular functions and suspending functions interchangeably).

The provided example in Listing 4 represents a single-shot operation where *fetchUsers()* runs in a context called *Dispatchers.IO*. This way, *fetchUsers()* will be dispatched to a thread pool dedicated to Input/Output (IO) operations. The rest of the code outside the *withContext(Dispatchers.IO)* scope will run in the main thread (*Dispatchers.Main*), the default thread for the coroutines started with *viewModelScope*. In this example, *fetchUsers()* should be a suspending function since it is likely to be a lengthy operation.

Data stream operations are supported by Flow [41, 42], a subsection of Kotlin Coroutines that is dedicated specifically to this kind of operations. Just like with the RxJava Observable/Flowable, Flow allows developers to generate and subscribe to a stream, shape the data incoming from that stream, and finally consume it. Flow can be put in the same category as RxJava as it was also built under the Reactive Streams specification [43]. Flow was created with coroutines in mind and its implementation integrates with suspending functions and with the overall Kotlin Coroutines ecosystem. To start consuming a Flow, developers must call *collect()* at the end of every Flow chain. This function is a suspending function, which means that Flows can only be consumed inside a coroutine or other suspend functions, tying it all back to the structured concurrency that Kotlin Coroutines strives for.

The integration of Kotlin Coroutines within the Android framework has been growing regularly with contributions from the Android team. Regarding such integrations, there is documentation [44, 45], talks and code labs. Code labs is a platform maintained by the Google team that provides small

courses created by their own engineers. There are multiple courses on Android development, where some address the use of Kotlin Coroutines [46].

Another important contribution from the Android team was the possibility to integrate Room Database [47] with Kotlin Coroutines. This integration makes sense as database accesses can often be lengthy operations that surely can benefit with the use of coroutines and suspending functions. Retrofit [48] has also added support for Kotlin Coroutines. Being a library mainly used for Hypertext Transfer Protocol (HTTP) requests it also benefits on the use of coroutines and suspend functions.

Kotlin Coroutines are exclusive to the Kotlin language. A lot of the process that makes this library work happens at compile-time, making it impossible to use for other languages. This does restrict the use of this library in comparison with RxJava, which can be used in other JVM languages. Since this research is towards Android applications, and those can be written in both Java and Kotlin, it puts both Kotlin Coroutines and RxJava at equal stakes for adoption (unless the idea is to add Kotlin Coroutines to an Android project currently written in Java, which would require a second step of transforming the Java code into Kotlin code before adding Kotlin Coroutines).

The Kotlin team implementation follows the idea behind the original coroutines, but it adds in a few tweaks that make this concept more suitable for modern development and allow for finer control over coroutines. As an extra point, Kotlin Coroutines has significant support from the Android team which makes it a proper choice not only for Kotlin applications, but specifically for Android applications. From here on, Kotlin Coroutines will interchangeably be referred to as Coroutines.

2.3. Metrics and Tools

The previous section illustrates the evolution that threading techniques had over the years from the former AsyncTask to RxJava and Coroutines. Historically, developers seemed to elect threading libraries that produce simpler code. This section will explore how threading libraries can affect the quality of a software system and what metrics can be used to objectively define that impact.

The ISO/IEC 25010 norm [49, 50] (a revised version of the former ISO/IEC 9126 [51]) elects eight major attributes to evaluate the quality of software systems. These can be found alongside their respective sub-branches in Figure 1.

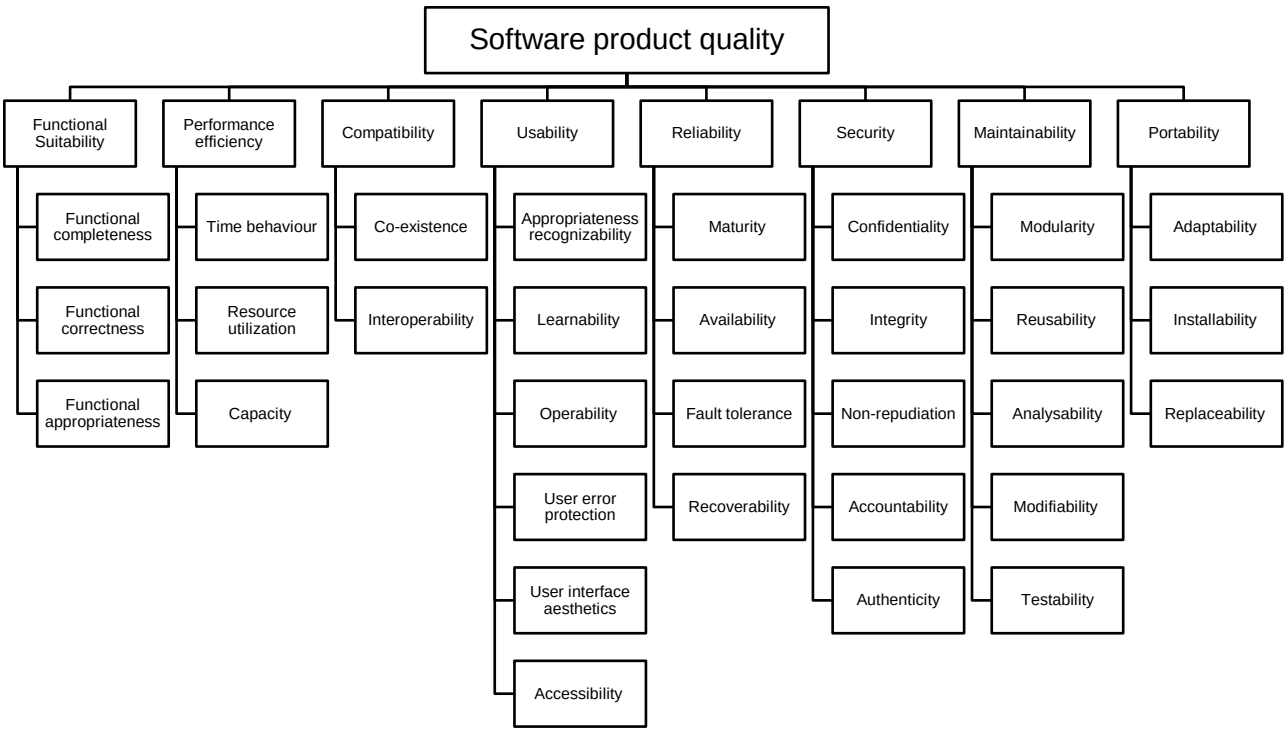

Figure 1. ISO/IEC 25010 attributes for software quality evaluation.

In a research about the importance of software quality attributes [52], the authors were able to assess which of the attributes mentioned above were the most concerning for business sustainability. The results came from an analysis of several conversations with stakeholders about which attributes they found to be the most important. The top three most concerning attributes mentioned were *Modifiability*, *Performance*, and *Usability*. The authors did recognize that all the attributes should be considered when designing and developing software, but the premise was always that certain quality attributes may have more impact than others.

From these three attributes, *Usability* is not influenced by the thread management library used for concurrency. It reflects the usability of the final product and should be agnostic to whatever library is used for threading, thereby it was discarded for this research.

The remaining two attributes, *Modifiability* and *Performance*, are influenced by the used threading library. *Modifiability* is a branch of the *Maintainability* attribute and has to do with how easy it is to change or add new features to a software system. As demonstrated in the previous sections, different threading libraries produce different code, if the code produced is too intricate, its understandability will be harmed and it will become, as development grows, less modifiable. Increased modifiability helps developers by easing the cognitive task of understanding what needs to be done to modify or add a new software feature. Indirectly, if old features are easily revised and new features are easily implemented, the company should have less technical limits to provide the best experience they can to end-users.

Performance is a main attribute itself. It branches out to *Time Behaviour*, *Resources Utilization* and *Capacity*. All these attributes can be affected by how a threading library manages the resources on a device. *Performance* is more evident to the end-user but, in the end, poor performance is an issue that developers must solve.

The first section of this chapter illustrated how a significant number of complaints on non-functional attributes could be related to performance issues (on Android applications), and later explained how most of these issues can be solved with a proper concurrency solution. By studying how threading libraries work, developers should be able to implement the most optimised solution and consequently improve the measured software quality - for performance and modifiability.

A limitation of the study [52] is the fact that results come from stakeholders from several different businesses, that own different types of software systems. This research strives towards mobile software applications so a further step must be taken to investigate what quality attributes have more impact on those.

Another study [53] that aimed to understand which quality attributes were the most influential, determined that *Usability, Performance, and Maintainability & Support* were the most impactful for mobile software applications, making the results very in line with the previously mentioned study. Again, *Usability* was discarded for the same reason mentioned above, and also, the authors of this study refer to *Support* as the support users may receive from technical problems (which we will not correlate with the threading library used for software development).

Maintainability and *Performance* seem to prevail amongst studies as some of the most important quality attributes, even for mobile applications. This research will take advantage of this information to provide developers with a relevant study on thread management libraries.

The sections below will explore metrics and tools that help this research by objectively measuring both *Maintainability* and *Performance*.

2.3.1. Maintainability

One of the most influential metrics for non-functional attributes in software is McCabe's cyclomatic complexity metric [54], published in 1976. This metric measure software complexity by calculating how much different paths could a flow of execution follow until it completes. The metric has been the base for a significant amount of research, as it inspired other new metrics and it is still valid and used, despite being published a long time ago.

The Lines of Code (LOC) metric, also a popular one, is measured through counting how many lines of code there are in the source code. This metric may seem quite volatile because it is naturally influenced by the way developers write code, but a study on its relationship with the original McCabe's cyclomatic complexity metric found a linear relation between both. The authors could predict roughly 90% of the cyclomatic complexity variance through LOC alone [55]. On this subject there is also a study [56] that explores how code complexity metrics can determine the amount of maintenance resources needed for different software units. The authors studied how the former McCabe's cyclomatic complexity metric performed against some other metrics that derived from it and included a metric of their own that results from the ratio of the original McCabe's metric by the

number of statements. They concluded that both theirs and the original McCabe's metrics are reliable predictors of software maintenance productivity.

SonarSource [57] has proposed a new metric to analyse code that defies the McCabe's Cyclomatic Complexity metric by taking a further step into measuring how *understandable* the source code is. This new metric is called Cognitive Complexity [58], and is meant to solve some shortcomings from the original McCabe's metric. The latter lacks the understanding of modern languages' structures, and even some of most fundamental principles on the original McCabe's metric are questioned and improved by this new metric, according to the author.

SonarSource is a company dedicated to help developers to have a robust source code by providing several tools for code analysis. This company is the creator of SonarQube [59] and SonarLint [60]. These tools are known for providing helpful feedback regarding source code health and are currently used by many professionals across the globe. SonarLint is an Integrated Development Environment (IDE) extension that works at development time and warns developers about possible improvements to the code source in development-time.

SonarQube provides a deeper analysis when compared to SonarLint and features a full-fledged web application where the health of the source code is described in detail through software metrics measurements. The metrics SonarQube provides are solely based on static analysis, but it can go as far as finding bugs in the code without running it, showing how well modelled its algorithms are. SonarQube provides most of the metrics explored above, including McCabe's Cyclomatic Complexity, LOC, and their own Cognitive Complexity. This makes SonarQube a leader for code analysis and a great fit for this research.

2.3.2. Performance

ISO/IEC 25010 breaks down *Performance* in three different sub-attributes, *Time Behaviour*, *Resources Utilization* and *Capacity*. Execution time, memory usage and Central Processing Unit (CPU) usage are known metrics that can measure the first two attributes, respectively. *Capacity* has to do with testing how much load the system can take, and it can be testable through understanding how the execution time and memory/CPU usage varies as tests demands more from the software.

The Android team seems to be gaining interest in helping developers to build performant applications. Their latest work on Profiling and Benchmarking shows exactly that. Android Studio has been the IDE of choice for Android development for almost a decade. It is built by JetBrains in collaboration with the Android team, sharing significant traits with other JetBrains IDEs and also including exclusive tools that are tailored specifically for Android development, such as the Android Profiler [61].

Android Profiler provides developers a detailed amount of metrics for Android development, in real-time. These include CPU usage, memory usage, network speeds, energy spent and more. These are useful to a developer to understand how its code is affecting the application performance. The Android Profiler has been getting regular updates and it seems to reflect what the Android team feels like is important to tackle from a user experience standpoint.

According to some of the studies presented on the beginning of this chapter, it is evident that users evaluate an app based on its performance. By stating that an app is unresponsive, slow, or that it drains too much battery out of a device, the user is noticing issues related to poor performance. By analysing these metrics while triggering specific functions on the app, developers can have a more granular understanding on how the app performs in each feature, making it a great profiling tool and a fit for this research as well.

Another significant contribution to modern Android development is the Android Benchmark [62] library, also integrated with the Android Studio. Here, Android Studio can create a scaffold module dedicated to use this library with a few clicks. It was built to run in continuous delivery environments if needed and it works like the instrumented tests already known to Android testing/development. Benchmarks built with this library can help this research measuring *Capacity* since it allows for developers to introduce quantitative factors to tests, such as, for instance, making twenty simultaneous network requests, or fifty, or one hundred. This kind of tests demonstrate how the software reacts as the load increases.

Building benchmarks for applications that run in the JVM (Java Virtual Machine) is a complex task. Some studies towards that topic [63–65] show how much of the compilation process and how the JVM itself can influence performance measurements. Most factors that influence such measurements are quite low-level and it is quite hard for every developer to grasp on them as the very evolution of languages and compilers led to these concepts being more and more abstracted from the regular software development process.

The Android team has taken responsibility in building this Benchmark library that takes care of most of these factors so they can continue to be abstracted from the developer. Having such a library to delegate all these tasks to is a helpful and ambitious resource from the Android team. This research should be able to take advantage of it. As of the time of this research, this library only measures the execution time of a portion of code, but according to feature requests [66, 67], it should be able to provide more metrics in the near future, including memory-related ones.

2.4. Summary

This chapter describes the issues related to concurrency in Android Development and the perils for user experience that can arise with a faulty concurrency strategy. The Android team recognized that and came up with their own contributions so that developers could easily offload lengthy work from the main thread on Android applications.

During the last decade, these tools started not to live to the expectations of modern development and were deprecated. As the paradigms in Android development changed, Android developers started to use well-known and tested thread-management libraries. The Android team and the Android community crafted tools that help integrate these threading libraries within the framework and as these libraries grow, so do their integrations with the Android framework.

The Android team ran a survey [68] and their main focus on concurrency were RxJava and Kotlin Coroutines, the standard threading libraries for Android development at the time of writing this document.

RxJava has been serving Android developers for years now and as an open-source solution, it has been growing up to the developers' needs. Kotlin has however, come up with an exclusive solution to the Kotlin language that might defy the leadership of RxJava on Android development: Kotlin Coroutines. Since Android applications can be written in both Java and Kotlin, this makes both these libraries eligible for Android development. Technically, both libraries can coexist in the same project (and it makes sense if the project is transitioning from one library to the other), but conceptually, these libraries can do the same and, in most cases, developers should choose one of them to keep consistency in their codebase.

This chapter has demonstrated how threading abstraction, task sequencing, the possibility to build reactive systems and decreased code complexity were traits which gradually evolved throughout the years for thread management libraries and developers promoted for Android development (RQ1). During this chronological review of the main approaches to concurrency in Android development we could also understand that by falling back on some of these characteristics, some of the approaches started not meet developer's expectations and eventually became deprecated or fell into disfavour for more reasonable alternatives (O1).

This chapter also explored how Performance and Maintainability are two of the most relevant software quality attributes for Android applications and can be influenced by the usage of different thread management libraries (RQ2).

During this initial investigation, it was possible to understand how a poor concurrency strategy negatively impacts the user experience of an Android application. It was determined that some of the user top complaints and performance issues found on Android applications could be related to poor concurrency systems. The use of thread management libraries can also affect the software product on a functional level. Not only because some of these libraries do not expose the same functionality but also because they may impact the modifiability of the software. Decreased modifiability limits the speed and stability of developing new features and revising/improving older behaviour. Overall, the usage of thread management libraries may affect not only the experience of the final consumer but also influence the company's possibility to maintain/improve the software product (RQ3).

This research proposes a comparison between Coroutines and RxJava against the two most relevant quality attributes for Android application, *Maintainability* and *Performance*. This will require further exploration on what use cases are relevant for this comparison and an Android application that uses both thread management libraries for those use cases.

According to previous research on this topic there is not, to the best of our knowledge, any research that compares Kotlin Coroutines to RxJava against non-functional requirements for any platform. This research should help developers choose between both libraries by providing meaningful and grounded interpretations of the quantitative data obtained during this comparison. The results can also evidence why a library might be preferred over another and what traits lead to that preference. This may help not only the involved libraries to evolve but also help the development of new thread management libraries.

3. Methodology

This chapter will portrait how this research was conducted and what was the rationale behind each step of our methodology.

The research method shares the overall structure of an Action Research [69] and a case study-based research [70]. The case study for this work was designed specifically to serve the purpose of benchmarking two approaches for the concurrency problem solving that developers for the Android platform seldom must solve, namely RxJava and Kotlin Coroutines.

The work for this research was conducted iteratively and incrementally into two-week sprints. Every two weeks, the team met to plan the work for the next two weeks and review the work done in the previous ones. The writing of this document followed the overall process since the beginning of the research. The document was written using a tool that allowed reviewers to actively provide feedback through the ability of commenting or make changes directly to it.

This research was an exploratory research where we aimed to objectively measure the impact in the quality of the software when using Kotlin Coroutines versus RxJava, for Android development. It had three distinctive phases: *Research and Case study*, *Experimentation*, and *Analysis and Conclusions*. These major phases are presented in Figure 2 and are later detailed in separate sections.

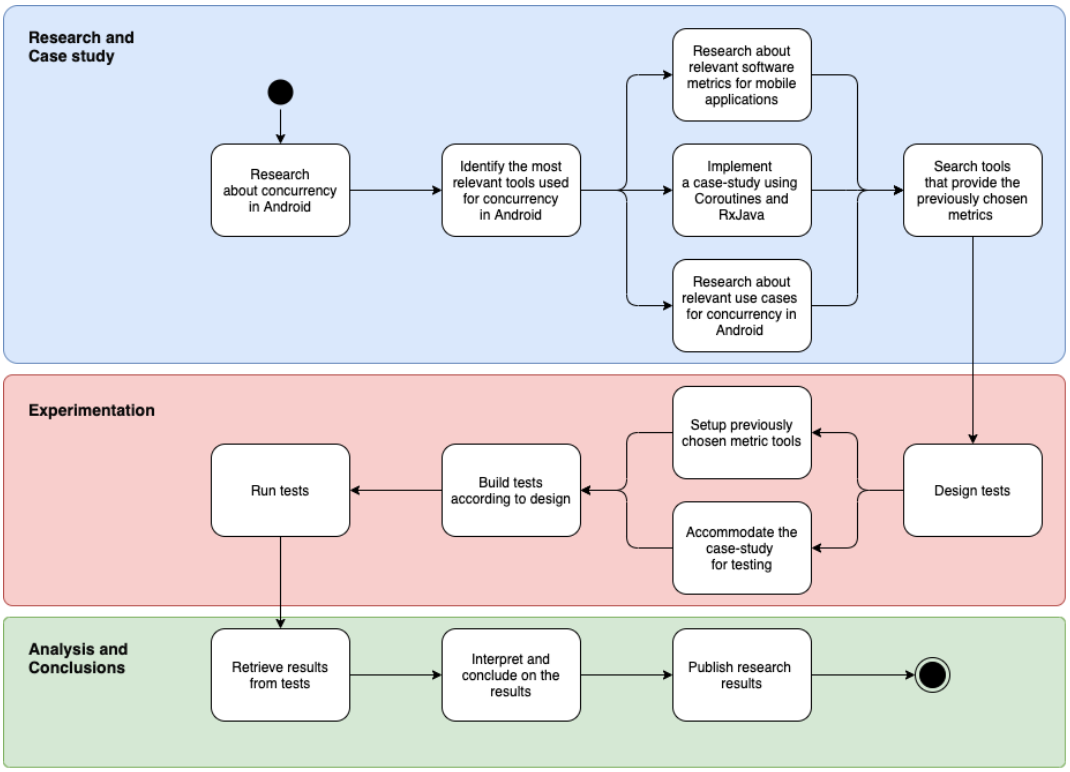

Figure 2. Overview of the research process.

3.1.Research and Case study

The *Research and Case study* phase was composed by the exploration of secondary data regarding concurrency in Android and the implementation of a case study application. An overview of this phase can be found in Figure 3.

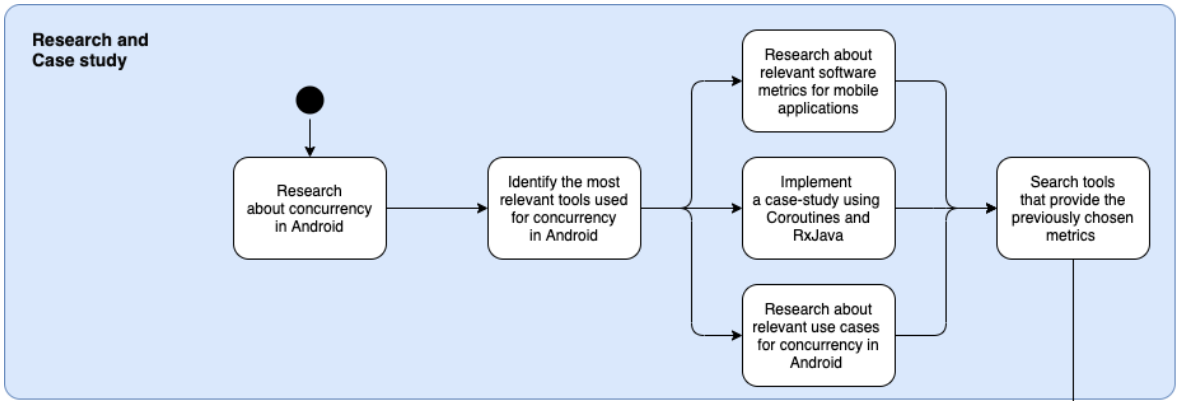

Figure 3. Research and Case study phase steps.

The initial stimulus for this research came from our professional experience (as explained previously in section 1.1). By developing Android applications with a heavy usage of concurrency, it is possible to experience how non-functional software requirements such as performance and modifiability can be affected by applying different techniques for concurrency.

We started with an investigation of previous studies regarding the topic of concurrency in Android. The first step was to understand why concurrency is important for Android development and what kind of consequences could arise from having a faulty concurrency strategy in Android applications. Google Scholar [71] was the main search engine to find academic work regarding this matter as it provided most of the studies that were explored during this step. Some of the studies had references to other relevant studies in their body that were later explored as well. A summary of the results of this stage can be read in Chapter 2.

The next step was to understand how Android developers could properly implement a concurrency system onto their applications. This allowed the research to focus on the most relevant concurrency techniques and build a more pertinent study.

The exploration of these techniques later revealed that Android developers often prefer to use tested thread-management libraries over implementing a custom solution. This led us to investigate which libraries developers were using and what makes a library preferred over another. We performed an analysis on the usage of each library in a chronological order and provided an example of the same hypothetical concurrent task for each library. This example was developed by us and it helped to illustrate the evolution that these concurrency libraries have suffered throughout the years. Most of the information that composed this section came from the official documentation of these concurrency libraries. To further understand the concepts behind each library and provide a deeper explanation on the matter we also relied on educational talks and informal articles published by

reliable sources. The analysis of previous studies regarding the topic of concurrency in Android and the subsequent research on threading libraries is reflected in sections 2.1 and 2.2.

Being focused on threading libraries, this research required an Android application to serve as a case study. The case study application ended up being developed by us, specifically for this research. To develop the case study application, three distinct phases occurred in parallel. We analysed relevant software metrics for mobile development and researched about pertinent features while we were implementing the application. The analysis of relevant software metrics came from reviewing previous academic studies on the topic, most of them found on Google Scholar and in references mentioned in those studies. This allowed the results to be more focused on relevant metrics and has cleared the path to what was important to feature in the final case-study application regarding both functional and non-functional requirements. The study on the most relevant metrics for mobile software application development is detailed in the section 2.3.

The research about pertinent features for this application was then influenced by the metrics that were interesting to evaluate and by professional experience. As we found a metric that we wanted to further test, we could start to shape the features on the application to emphasize that metric during the testing phase, providing us with more relevant results. Having professional experience helped us to identify some of the most common contexts to use concurrency in, and later to specify the features on the application to be used for those contexts. Aside from academic and professional work on the matter, the choice of the features to include on the case-study application was also corroborated by the official guides from the Android team. The Android team commonly includes libraries which have integrations with RxJava or Kotlin Coroutines for concurrency work. This allowed us to understand which libraries the Android team recommends and in which contexts does Android development mostly need concurrency.

As we knew which kind of features the application should have to meet the relevant metrics and features, we would then implement them gradually, making it an iterative process. Both professional experience and the official guides from the Android team aligned to provide us with a relevant architecture and non-concurrency libraries that are considered standard for Android development. The Android team also tends to reference some libraries and implementation details on their Codelabs [72] platform. This allowed us to have a verified and objective implementation for this case study application from a technical standpoint. The details of the case study application will be further explored in Chapter 4.

Moving forward, we searched for tools that could provide quantitative data regarding those metrics and were compatible with the case-study specifications. This search mostly relied on both search engines and professional experience. Some tools came from empirical use or peer recommendations from a professional environment. We relied on official documentation to understand technical details of each tool and even experimented with some of them to confirm their suitability to this research.

The main goal of this step was to find tools that were reliable since they would have a huge impact on the research results. The professional experience helped us not only achieving the right functionality (measuring metrics properly) but also contributed to using tools that are credited by professionals. This search is also documented in the section 2.3.

3.2. Experimentation

The *Experimentation* phase consisted in designing, building, and running the tests on the case-study application to achieve objective results regarding the performance of each library against previously chosen software quality metrics. The steps that composed this phase are illustrated in Figure 4.

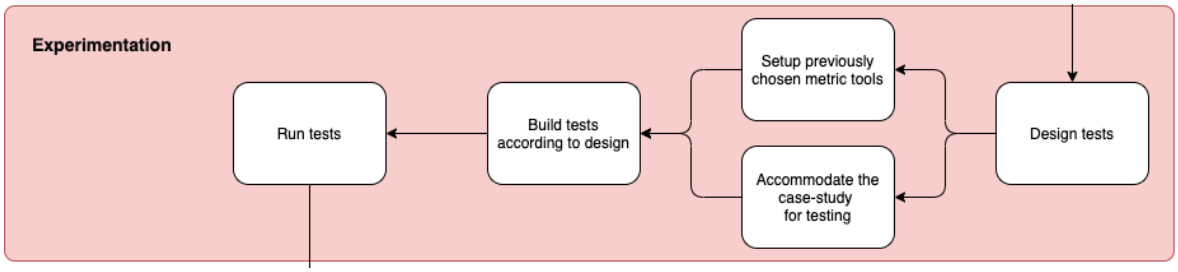

Figure 4. Experimentation phase steps.

With the overall knowledge about the tools that were available for this research and the type of behaviour we wanted to test; we designed the tests that would allow us to compare both libraries against the previously chosen attributes. The tests were designed to emphasize the metrics we chose and ensure that both libraries, RxJava and Kotlin Coroutines, were tested under the same scenarios using their own implementation for the different contexts. The test design is further explained in Chapter 5.

Following the tests design, we setup the previously studied metric measurement tools and created a test environment within the case study application code that helped with the tests' reliability. Before running the tests, we wanted to mitigate the bias of any external variable on the results. Some of the components planned to be tested relied on systems outside the application control and would indeed influence the research results in a non-deterministic way.

To improve the results reliability, we created software components in the case-study application dedicated to stabilizing this variation. Building these software components was mostly driven by documentation and professional experience. In parallel to building this test environment, we setup the tools that we had previously studied to help us objectively measure the chosen metrics. This setup was mostly oriented by the information we had from previous steps and official software providers documentation.

With everything setup, we moved on to implement the tests according to the design. Recognizing which components from each library (RxJava and Kotlin Coroutines) could be directly compared when implementing the tests came from professional experience with both libraries and from the previously studied documentation of each library. The setup of the metric measurement tools, accommodation of the case-study for testing and the test implementation are further described in Chapter 5.

To mark the end of the Experimentation phase, we executed the tests. Running the tests required some experience with Android Studio, a good understanding of the case study application and of the tools used to measure its attributes.

3.3. Analysis and Conclusions

Analysis and Conclusions consisted in analysing the results of the second phase, extracting relevant conclusions from them and finally publish them. These steps are illustrated in Figure 5.

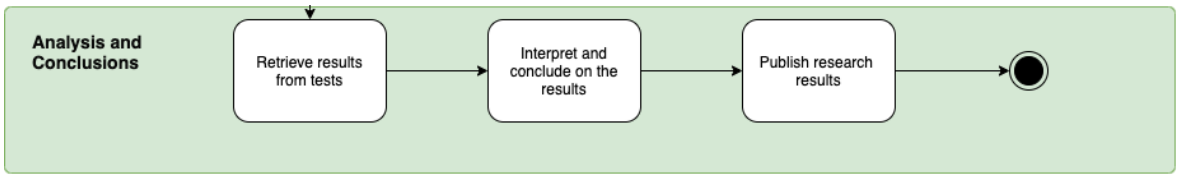

Figure 5. Analysis and Conclusions phase steps.

After running the tests, we moved on to retrieving the results provided by the metric measurement tools. Some problems emerged at the time of retrieving these results. The initial conditions we had and the output of some of the tools were not compatible for the automated creation of visualisers that would later ease the interpretation of the results. To mitigate this incompatibility, we implemented a software component and a spreadsheet that allowed us to automatically create tables and charts out of the results extracted.

The next step was to analyse the results and to further understand if there were any advantages in using one library over another. This step mainly relied on going back to previous studies about each metric we measured to understand the impact RxJava and Kotlin Coroutines in quality attributes. From there, we could further compare them and conclude on whether there is a library that makes more sense to opt for. The results of this research will be later featured in Chapter 6 and their interpretation will follow in Chapter 7.

The last step was to publish the results of this research, write a scientific article and provide the case study application codebase under the form of an open-source project which any interested reader can consult/use.

4. Case study application

This section will illustrate the process behind developing a case-study application to this research as well as explore its architecture and implementation.

The case study is an Android application that provides the user with a list of trending movies and local movies. The user can then choose a movie from the list to see an overview of that movie and eventually be redirected to YouTube to watch the trailer. This set of features emphasize the previously chosen software attributes and require the use of concurrency techniques to be properly implemented.

The case study application was developed using the Kotlin programming language. All the concurrency work was managed by RxJava or Kotlin Coroutines, the chosen concurrency libraries to be compared for this research.

In the next sections, we will explore the set of functional and non-functional requirements of the case study application and provide technical details related to its implementation.

The full implementation of the case study application can be found on GitHub as an open project [73].

4.1. Functional requirements

The case study is an Android application built solely for this research. It features a simple list that includes details about movies and the possibility to be redirected to YouTube to watch their trailers.

The list of movies can either be a list of trending movies or a list of movies that are currently playing in the user's country, e.g. "movies now playing in Portugal", if the user is in Portugal. To detect the user's location the application will prompt for the location data permission and in case the user allows it, the app will fetch the country code of the user location and show the movies currently playing for that country. For demonstration purposes, when the user activates this feature of fetching the local movies, the app will periodically retrieve the user location and the list of movies for the current country.

Figure 6 represents three screenshots from the case-study application that illustrate the main features. The first two represent the trending movies and the local movies lists, respectively. The third represents the interaction that can be done with each movie in the list to read an overview and eventually play its trailer by redirecting the user to YouTube.

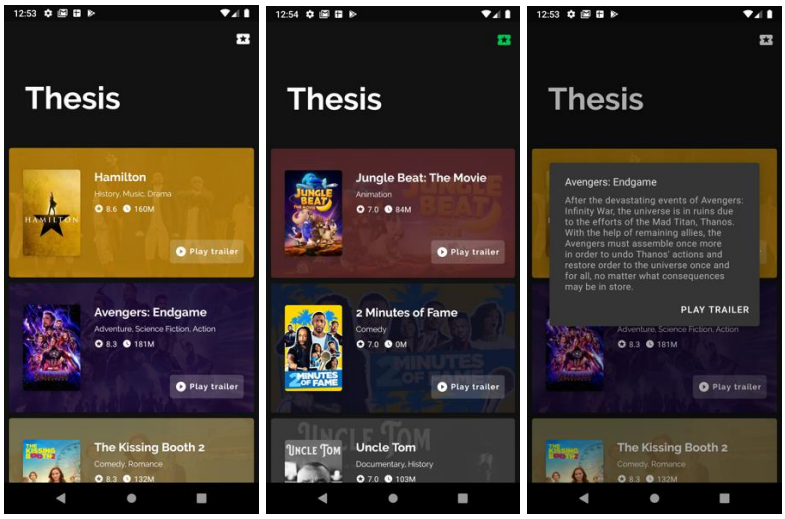

Figure 6. Case-study application features.

With this set of features, we could analyse and compare both single-shot operations and data stream operations using RxJava and Kotlin Coroutines.

4.2. Non-functional requirements

The subject to this research needed to fulfil some pre-conditions to form a relevant case study. This section will explore those conditions. The subject for this research must be an Android application. The application should be written in Kotlin as Google has stated that Android development is increasingly becoming Kotlin-first and advises developers starting new projects to use Kotlin [74].

Developers do not have to follow any specific architecture to build Android applications. However, the Android team features an official guide to architecture, available in the Android documentation [25]. Despite this guide not carrying any form of obligation, the subject to this research should follow most of the principles stated by this guide so the featured concurrency cases come from a modern and recommended architecture.

This guide introduces MVVM (Model-View-ViewModel) [69, 70] as the main architectural pattern to follow in Android development. MVVM is a notification-based pattern and mainly relies on reactive components to be implemented. As previously mentioned, both Kotlin Coroutines and RxJava support data-stream operations which inherently help developers looking forward to implementing software using this pattern. Figure 7 depicts the main interactions between the different layers when using the MVVM architectural pattern.

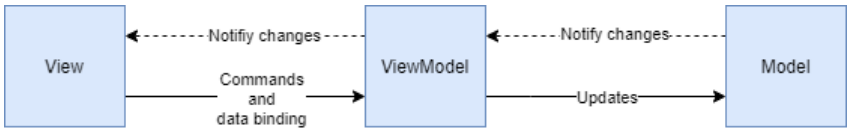

Figure 7. The MVVM pattern.

The Android team defends that the user interface should reflect the data available in a local data source, in this case, an SQLite database [77]. This kind of link between the interface and the database is rarely direct. It is very likely that persisted data needs treatment before reaching the user interface. This treatment can be a lengthy operation that developers should avoid doing in the application main thread. This became evident when the Android team allowed Room [47], their own implementation of an SQLite Object Relational Mapping (ORM), to integrate with standard thread management and reactive libraries such as RxJava and Kotlin Coroutines. These libraries can successfully strive this lengthy work from the main thread and push the notifications from the model layer to upper layers. Therefore, our case-study application should implement this sort of link between the user interface and the local database to study this case of concurrency in Android. The application should also use the Room persistence library to do so, as it is recommended and designed by the Android team.

The guide provided by the Android team also accounts for the use of remote services to enrich the user experience. One common case for concurrency would be a network request as its length is undetermined, reason enough to do it off the main thread. This guide makes use of Retrofit [48], a library that facilitates the integration with HTTP based services by providing type-safety and threading out of the box. Retrofit is a very popular and tested library amongst Android developers, thus its use in the official guide. For these reasons, the subject should consume data from a remote service using Retrofit as it also integrates with standard thread management libraries [77, 78].

Another advantage of mobile applications is their ability in using the available sensors in the device to understand the current user context. Like the database case, this one also represents a possible stream of data that needs to be manipulated to be relevant to the end-user and should also be considered for our case-study application.

The official guide uses the Repository pattern [80] to abstract the data layer access from the presentation layer. This abstraction is advantageous for complex domain systems and should also be featured in the case study.

Once the case-study application implements these requirements, it will be possible to study both single-shot and data stream operations, commonly used across Android development.

The case-study application must also use both Coroutines and RxJava to test their integrations with the Android framework and libraries that are considered standard for Android development.

4.3. Architecture

This section will cover the architecture for the case-study application by explaining the communication between the different components and further explain their implementation.

The case-study application implementation follows most of the principles recommended by the Android official guide to architecture [25]. As previsouly mentioned, the application follows the MVVM architectural pattern alongside the Repository pattern for the data access layer. Figure 8 represents the main components in the application architecture and the communication between them.

The components marked with the orange and purple circles have a different implementation for RxJava and Kotlin Coroutines. The dotted lines represent data stream connections between the components, whereas the non-dotted ones represent single-shot connections.

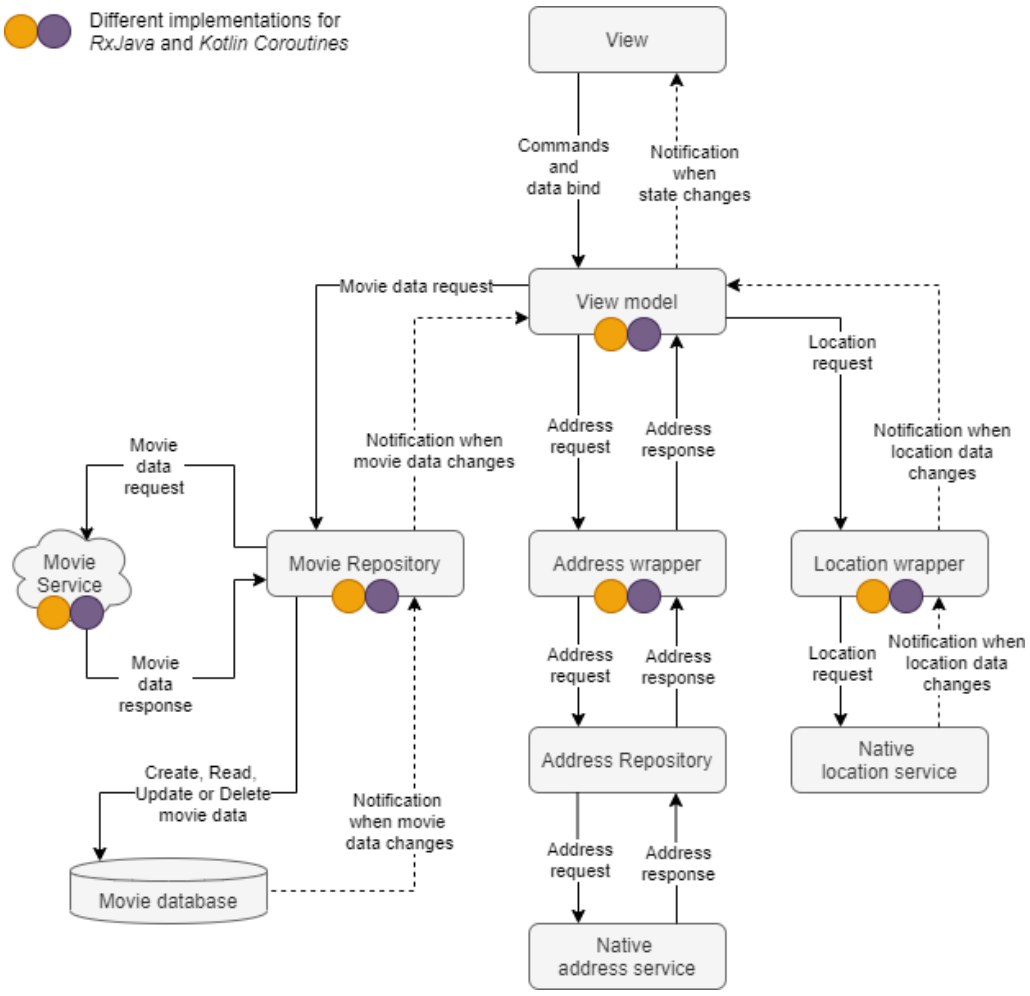

Figure 8. Overall case-study architecture.

The View component represents the upper layer of the application. This is where the UI components sit and where the interaction with the user happens and is further propagated to lower layers. The View component solely depends on its single source of truth, the ViewModel.

Following the MVVM pattern, the View must reflect whatever state the ViewModel holds in a reactive manner. Every time there is a change in the ViewModel state, the View automatically draws itself to reflect that new state. The communication between these two components is not unidirectional however, the ViewModel also receives events from the View to further decide how to alter its state.

As illustrated in Figure 8, the ViewModel component has different implementations for Kotlin Coroutines and RxJava. This is because most of the asynchronous work starts in the ViewModel, and both RxJava and Kotlin Coroutines use different techniques to manage it. For instance, whereas RxJava will mostly use *Single* and *Flowable* for single-shot operations and data stream operations, respectively, Kotlin Coroutines will rather use *suspend functions* and *Flow*. Although the ViewModel classes for each library have different implementations, they both extend the same abstract class to ensure they expose the same methods and properties to the common View component.

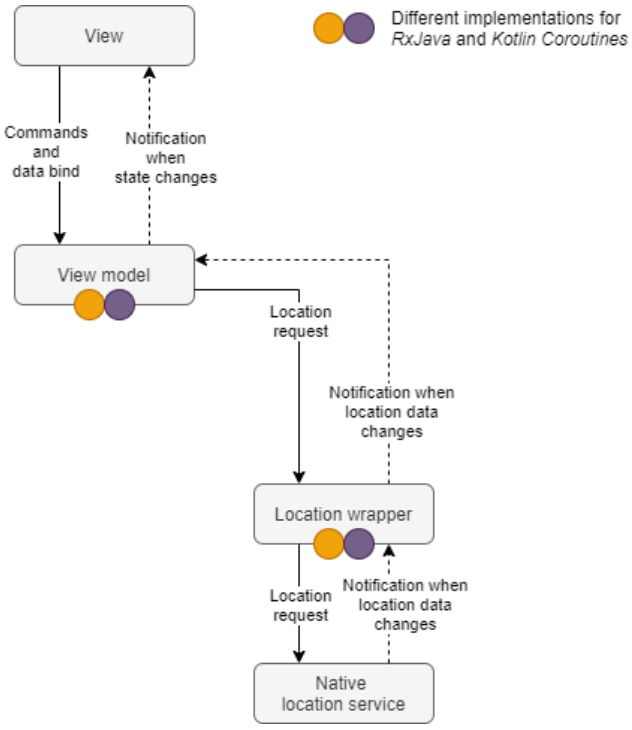

Figure 9. Location request close-up.

Highlighting the user location request, we have two main components: 1) the location wrapper, and 2) the location service (Figure 9). The ViewModel interacts solely with the location wrapper, not directly with the location service.

The location service comes from the Android framework and it is accessed through a *FusedLocationProviderClient* [81] instance. This class allows developers to subscribe to new location updates and further consume the new locations under the form of regular callbacks.

We tend to keep this reactive approach of receiving a new location, but instead of using regular callbacks, we wish to propagate the new location using RxJava or Kotlin Coroutines. Both these libraries have dedicated ways to transform callback-based code into their own paradigms. The RxJava and Kotlin Coroutines location wrapper implementations uses *Flowable* and *Flow (callbackFlow)*, respectively, to wrap the location service request and then propagate the new location in a reactive manner. This allows the ViewModel class to be completely unaware of any call-back-based code and solely interact with fully featured components from the RxJava and Kotlin Coroutines libraries for this data stream operation.

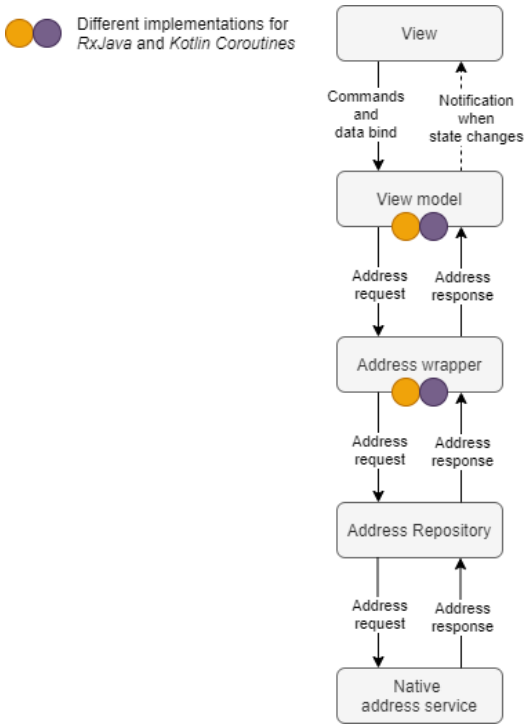

Figure 10. Address request close-up.

The previously explained location request exposed location coordinates, but the feature for the local movies required the country code of the user's location. To obtain further meta-data about the user's location, we used an address request that enriched and followed the location request.

The set of components used for this address request (Figure 10) is conceptually similar to that of the location request (Figure 9). We have the ViewModel interacting with the Address wrapper and not the Address repository directly. The Address repository exposes a single method that fetches the address meta-data using the previously fetched coordinates and further propagates the results under the form of a regular call-back. To avoid working directly with callbacks on the ViewModel and test

the callback transformation to the RxJava and Kotlin Coroutines paradigms, we have created a wrapper component, like we did for the location request.

The main structural difference between the address request and the location request is the that the first is a single-shot operation whereas the second is a data stream operation. The way both RxJava and Kotlin Coroutines propagate the address request result is different from that of the location request. The AddressRepository single method returns the results under the form of a callback, this was purposely done to demonstrate the way both RxJava and Kotlin Coroutines can wrap single shot callbacks. The Address wrapper implementation for RxJava and Kotlin Coroutines uses the *Single* and the *suspendCancellableCoroutine* constructs, respectively, to wrap the callback-based code and propagate the address result.

This will abstract the callback-based code from the ViewModel and more importantly allow the direct integration of this address request with the location request. This integration is possible because both requests expose library (RxJava and Kotlin Coroutines) exclusive components that are compatible with each other, e.g. *Flowable* from the location request is compatible with the *Single* from the address request, the same happens with *Flow* and *suspend functions* (the *suspendCancellableCoroutine* construct exposes a *suspend function*) in the Kotlin Coroutines scenario.

Both the location and the address request integrate to help implementing the local movies feature, but there is a common set of components (Figure 11) to both features which compose the core section of this architecture.

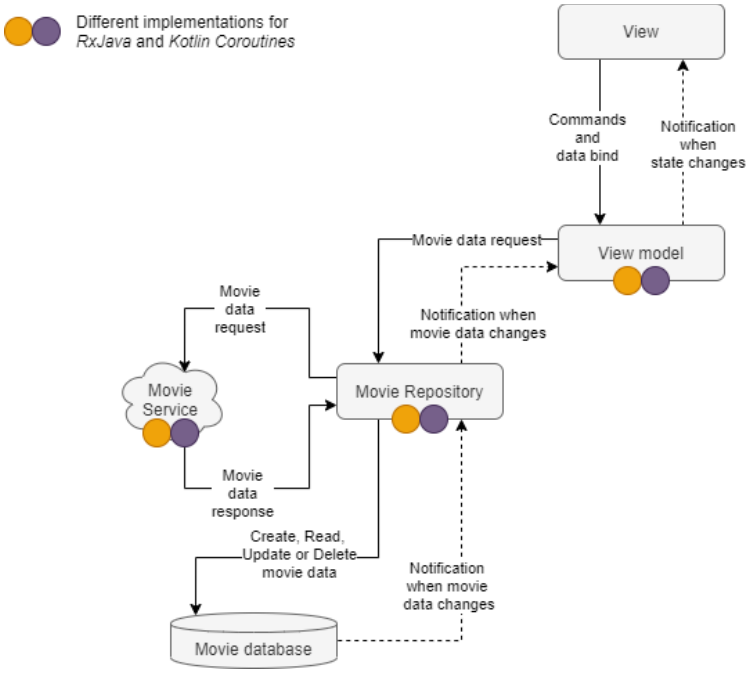

Figure 11. Movie data request close-up.

This set of components is responsible for obtaining the movie data from an external movie service and then persisting that data in a local database. The information stored in the local database is then exposed reactively to upper layers of the application.

The Movie repository class abstracts the interaction with the external movie service and the local database from the ViewModel. It consumes data from the external service using Retrofit and manages the local database access using Room database. Both Retrofit and Room have integrations with RxJava and Kotlin Coroutines.

The movie service interface for both libraries is inherently different. The RxJava version exposes *Single* instances for the external service accesses whereas the Kotlin Coroutines exposes *suspend functions*. The RxJava version required a custom call adapter factory [78, 81] to successfully integrate Retrofit with RxJava objects. The integration with Kotlin Coroutines is built-in with Retrofit and did not require any extra dependency.

The local database is common to both libraries, but it exposes the data in different forms that are natural to each library. Single shot database operations like insertion/deletion of new records are done through *suspending functions* and *Completable* objects by Kotlin Coroutines and RxJava, respectively.

As illustrated in Figure 11, querying the movie data is done in a reactive way, meaning every time there is new information for that query, Room pushes it via a data stream to the layer that subscribed it. The RxJava and Kotlin Coroutines repository version exposes *Flowable* and *Flow* instances for that effect, respectively. The data being propagated in this data stream connections is later formatted in the ViewModel component and the stream is finally subscribed in the View component that reflects the new information.

4.4. Class Structure

This section will illustrate how the case study's class structure helped the research testing Kotlin Coroutines and RxJava against the chosen software quality attributes. The way the classes are segregated is mainly interesting for static-analysis tests, as it does not directly influence the performance of the application.

When observing the code produced by Kotlin Coroutines and RxJava, we ensured both versions of the code could be directly compared. This meant we needed to have Kotlin Coroutines and RxJava performing the same tasks using the same file/class structure.

The case-study application includes a module dedicated to each library, the *CoroutinesKit* for Kotlin Coroutines and the *RxJavaKit* for RxJava. The CoroutinesKit and the RxJavaKit contain all the code specific to Kotlin Coroutines and RxJava, respectively. Both modules have a similar file structure, and each file/class is designed to expose the same exact behaviour despite using different technology.

The architecture section illustrated in Figure 11 represents some core components for the case study application. The main classes for that section are represented below in Figure 12 using a class diagram.

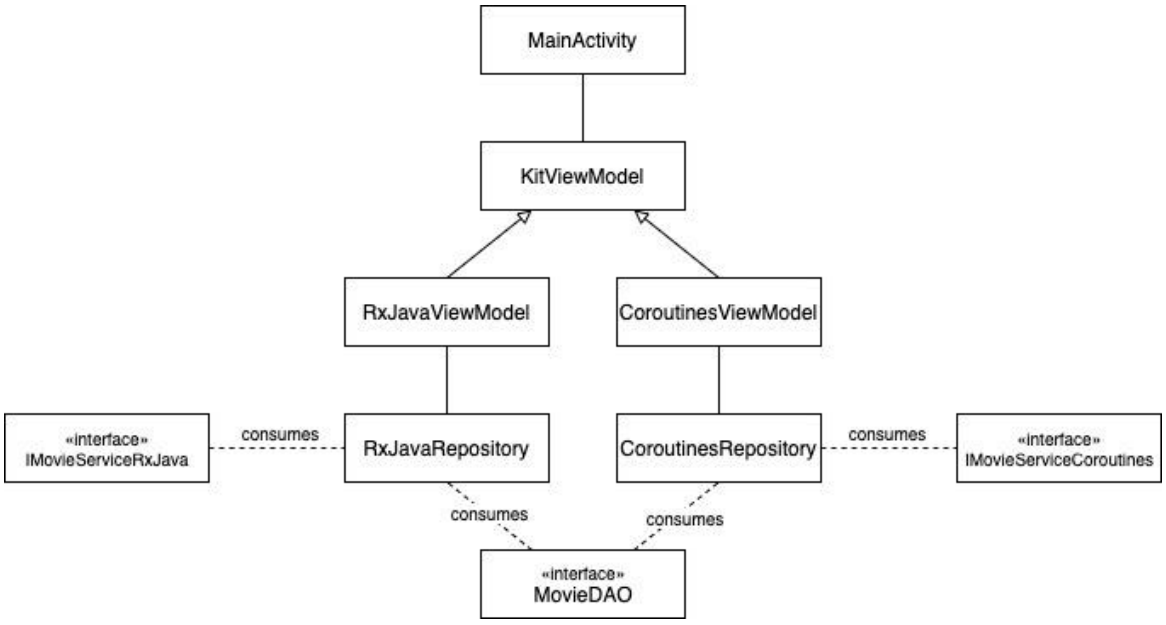

Figure 12. Class diagram for main components.

The MainActivity class represents the View component of the architecture. It solely relies on a view model class that extends KitViewModel, which can be either CoroutinesViewModel or RxJavaViewModel. These are obliged to expose the same methods because they extend KitViewModel. This ensures we can compare both classes directly in static analysis.

Each ViewModel class uses a repository class to fetch movie data. There is the CoroutinesRepository which is used by CoroutinesViewModel and the RxJavaRepository which is used by RxJavaViewModel. Both these repository classes could not extend a common class/interface because they inherently expose different methods. As CoroutinesRepository exposes the movie data under the form of *suspending* functions and *Flow* whereas the RxJavaRepository uses exclusive RxJava objects like *Completable* and *Flowable* for the same behaviour. Despite not being forced to expose the same methods as the ViewModel classes do, we built both repository classes with the same structure. Both repositories contain the exact same method/property structure and expose the same behaviour. This compliance in structure is what allowed us to compare both repository classes directly using code static analysis.

Each repository then uses remote and local sources of information. The remote source is the movie web service we use to fetch movie data and the local source is the local database we use to persist the movie information. Each library has a dedicated web service class, IMovieServiceCoroutines and IMovieServiceRxJava, for Kotlin Coroutines and RxJava, respectively. The local database is accessed through a single Data Access Object (DAO) called MovieDAO. MovieDAO is an interface and there is only one instance of its implementation (generated by Room Database) which is shared between the repository classes of both libraries. The structure of the classes used to access the web service and the local database are mainly dictated by Retrofit and Room, respectively. These classes look essentially the same for Kotlin Coroutines and RxJava, and therefore do not represent interest for static analysis.

For the represented section of the architecture, the focus for static analysis comparison are the ViewModel and Repository specific classes for each library.

The process to segregate classes that was demonstrated here was extended to the other sections of the architecture. Following this process, we ensured every comparison we did between the code produced with Kotlin Coroutines and RxJava was reliable.

4.5. Summary

This chapter explored the case study application and all the requirements we considered deemed to compose a valid and relevant subject to this research. It started by tracking some non-functional requirements from an Android official guide for architecture. By following this guide, we ensured our case study application met the current trends for modern Android development and by doing so, we could also retrieve some common use cases for concurrency in a modern architecture (O2). It not only provided us with relevant use cases but also referred standards libraries that are used in conjunction with thread management libraries to comply with functional requirements (e.g. Retrofit is a popular library that help developers integrate with HTTP based services and it allows integration with RxJava or Kotlin Coroutines to provide network results under a more familiar and controlled form to the developer).

After acknowledging common use cases for concurrency in Android development, we developed a case study application with functionalities that allowed us to explore those concurrency use cases with both RxJava and Kotlin Coroutines (O3). This chapter provided a detailed description of the main architectural components of the case study application and how they communicate with each other. It also validated the possibility of directly comparing both RxJava and Kotlin Coroutines by analysing the class structure on the case study application.

5. Test Cases Design and Environment Setup

This chapter will describe the thought process behind the tests that were done to compare Kotlin Coroutines and RxJava. These include static analysis and performance analysis tests. It will also portrait the process of setting up the environment to test the hypotheses of this research as well as provide validation for the results obtained.

5.1. Test Cases Design

This section will be primarily focused on explaining the tests which measured the *Maintainability* and *Performance* of our case study application. It will cover static analysis tests and performance analysis tests.

5.1.1. Static Analysis

The case study separates the code written with both libraries by having the *CoroutinesKit* module and the *RxJavaKit* module independent from each other. These modules expose the exact same behaviour, but their code is written using Coroutines and RxJava, respectively. These modules contain a very similar file structure since they both follow the application architecture; this means they can be directly compared when using static analysis.

Each library module contains at least the view model class, the movie repository class, the location request wrapper class, and the address request wrapper class. These classes represent the code that can differ from having an implementation using Kotlin Coroutines and an implementation using RxJava, this is exactly the code which this research aims to compare between the two libraries.

The metrics to be measured will be Lines of Code, Cognitive Complexity and Cyclomatic Complexity. These metrics and their impact have been previously described in the Section 2.3.1.

5.1.2. Performance Analysis

This section will explore what kind of tests were done to measure the performance for both Kotlin Coroutines and RxJava, when executing the same features. These tests measure not only the individual performance of certain components of the application like networking or database access, but also of the integration of those components.

Most of the testing done in this regard does not depend directly on the features implemented by the case study but rather the components that form those features. For instance, when measuring performance there is no need to create scenarios that make sense within the case study functionality, but rather understand how each component evolves both individually and when chained with other components. For this section of the testing phase, we made sure to test both single-shot operations and data stream operations for Kotlin Coroutines and RxJava as well as synchronous and parallel requests.

By testing all these variants, it is possible to have a deeper understanding of how each library performs against the previously mentioned performance attributes: *Time Behaviour*, *Resources Utilization* and *Capacity*.

This section describes the three main test categories used to measure the performance between both Kotlin Coroutines and RxJava. Those are network tests, database tests and integration tests.

All the categories feature Benchmarking tests where the single metric to be extracted is time of execution. The integration tests also feature profiling tests that provide the memory usage, maximum CPU usage and time of execution metrics.

All the categories include tests that are built on top of hypothetical scenarios which do not make sense within the case study functionality (e.g. there is no feature in the case study application that requires fifty network requests simultaneously, but there is a performance benchmark test that measures that hypothetical scenario), the integration category will also include tests that measure the performance of both libraries when executing the case study features: fetching trending movies and fetching movies playing in local theatres.

Network tests

Network benchmark tests are related to the integration of Retrofit with both Kotlin Coroutines and RxJava. The tests for this category, to be labelled as Network Tests (NT):

NT1. Do a single network request.
NT2. Do three sequential network requests.
NT3. Do two parallel network requests.
NT4. Do ten parallel network requests.
NT5. Do twenty parallel network requests.
NT6. Do fifty parallel network requests.

This set of tests will provide granular results for single-shot network accesses with both sequential and parallel requests. The set of parallel requests test will allow the research to explore how both libraries handle the load of requests by gradually increasing the number of parallel requests.

Database tests

Database benchmark tests refer to testing Room integration with both Kotlin Coroutines and RxJava. The tests for this category, to be labelled as Database Tests (DT):

DT1. Insert two objects into the local database.
DT2. Insert ten objects into the local database.
DT3. Insert twenty objects into the local database.
DT4. Insert fifty objects into the local database.
DT5. Insert one hundred objects into the local database.
DT6. Clear the local database and inserts twenty objects into the local database.
DT7. Query twenty objects from the local database.
DT8. Query fifty objects from the local database.

DT9. Query one hundred objects from the local database.
DT10. Query twenty movies twenty times in parallel.
DT11. Query twenty movies using a reactive pattern.

These tests are mostly focused on querying and inserting data in the local database as those were common use cases for concurrency with database accesses. This set of tests will provide the research with values for both single-shot and data stream requests for database. Both tests for inserting and querying will gradually increase the number of movies to insert/query to understand how both libraries manage that load. The set of tests also features tests for parallel request where the database will be queried simultaneously by multiple clients and a request for cleaning and inserting in the database sequentially.

Integration tests

Integration tests are tests that rely on both network and database requests. This categories feature both hypothetical scenarios and real scenarios using the case study application functionality.

The benchmark tests for this category, to be labelled as Integration Benchmark Tests (IBT):

IBT1. 1 routine - 1 x fetch movie list, 1 x fetch movies details, 1 x save to database
IBT2. 2 routines - 2 x fetch movie list, 2 x fetch movies details, 2 x save to database
IBT3. 3 routines - 3 x fetch movie list, 3 x fetch movies details, 3 x save to database
IBT4. 4 routines - 4 x fetch movie list, 4 x fetch movies details, 4 x save to database
IBT5. Throw a signal after fetching one list of movies and fetch another list of movies when this signal is received using reactive components.

For the benchmark tests, most only differ in the number of times they call a certain method. For instance, the test called *1 routine* fetches a list of movies, fetches the details for each movie and saves them to the database, it runs this routine where it calls each method only once. The test called *4 routines* will call the same exact methods but will run each method four times. Apart from these tests there is also one test that integrates two methods in a reactive fashion using a data stream operation.

The integrations tests were complemented with tests to the case study original functionality using the Android Profiler. The case study is composed by two main features, fetching a trending list of movies, and fetching the local movies considering the user location. The tests to these features will be designated as Integration Profiler Tests (IPT1 and IPT2, respectively).

Both features differentiate on the components they use and on the structure of the requests to those components. To assess how Kotlin Coroutines and RxJava behave on a real Android application, we profiled the case study application and ran each feature 3 times from beginning to end. The evaluation is then assessed by the average values of the provided metrics for each feature.

5.2. Environment setup

This section is dedicated to clarifying the setup of the test environment. It will cover how the previously described tests were setup, what tools were used and how the extraction of the results proceeded. This section is composed of two sub-sections: Static Analysis and Performance Analysis.

The Performance Analysis section will also explain how we mitigated some shortcomings that could negatively impact the reliability of the results as well as describe the software components we developed to complement the main tools used for performance analysis: the Android Jetpack Benchmark library [62] and the Android Profiler [61].

5.2.1. Static Analysis

Static analysis-based metrics were measured using a local SonarQube [59] server. The setup of this local server is best described in SonarQube's documentation [83]. The local SonarQube application runs by default on port 9000, and by accessing it through a browser, developers may consult the many measures and statistics this service offers in a user-friendly interface.

On SonarQube, after selecting an analysed project, these measures can be found within the Measures tab, under the Size and Complexity sections, respectively. These sections are evidenced below in Figure 13.

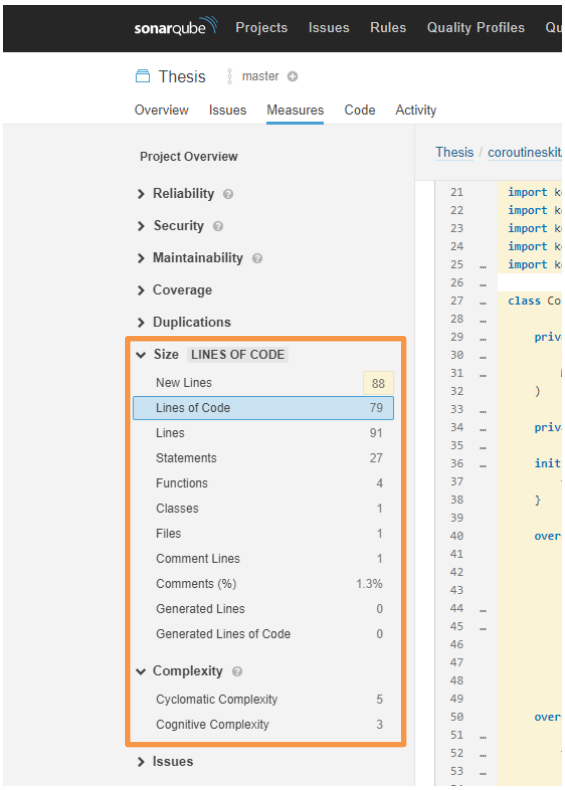

Figure 13. SonarQube Measures dashboard.

The SonarQube server identified the project as a Kotlin application and provided analysis tailored specifically for the language.

The project includes a Gradle task that covers the integration between the project and the SonarQube local server. This task is called *sonarqube* and it is in the *build.gradle* file inside the *app* module (c.f. Listing 5). Looking at the "sonar.sources" property, SonarQube will receive the source code of both the Coroutines and RxJava kits, allowing both to be analysed simultaneously. This task encapsulates all the properties needed to run the analysis and it requires the SonarQube Gradle plug-in [84] that was previously imported into the project.

```
sonarqube {
    properties {
        property "sonar.projectKey", "Thesis"
        property "sonar.projectName", "Thesis"
        property "sonar.host.url", "http://localhost:9000/"
        property "sonar.login", "xxxxxxxxxxxxxxxxxxxxxxxx"
        property "sonar.scm.disabled", true
        property "sonar.projectBaseDir", "$projectDir/.."
        property "sonar.sources", "coroutineskit/src/main/java," +
                "rxjavakit/src/main/java,"
        property "sonar.coverage.exclusions", "**/*"
    }
}
```

Listing 5. Gradle task to integrate with SonarQube.

The results obtained by static analysis rely solely on SonarQube's algorithm and rules as the process has not been modified in any way from its default configuration.

5.2.2. Performance Analysis

To measure performance and resource related metrics, the tests relied primarily on two tools: Android Profiler and Android Jetpack Benchmark.

When measuring performance there are some variables that need to be stabilized so the results are reliable, namely having a consistent network response time and making sure memory is optimized when analysing its usage.

This section will not only cover the extra behaviour that took to stabilize these variables but also some other components developed specifically to accompany both testing tools, Android Benchmark and Android profiler.

Test device

All the tests conducted, from both benchmark and profiler, ran in the exact same device: a Huawei P9 Lite running Android 7, API 24.

Initially, the tests were meant to run in an emulator through Android Studio, but the Android Benchmark library has a set of configurations that must be met to run properly. These include not using a virtual device. Benchmark tests will throw a warning and will not start until these

configurations are either met or suppressed. The library highly recommends not suppressing these, since not having the recommended configurations can harm the reliability of the test results [85]. The testing was also done in a real device for the Android Profiler tests, for consistency.

All the tests ran using the device's Airplane mode, so they are not as easily affected by external applications' behaviour. Both Benchmark and Profiler results came from a series of iterations to avoid any outliers.

Mocking the network layer

There are tests related to the use of Retrofit and its own integration with both Kotlin Coroutines and RxJava. Originally, the application will use Retrofit to consume an external service, through HTTP. When dealing with network calls, there is no way to control the response time from the client application since it highly depends on the network infrastructure and the external service's performance.

Having an unpredictable network response time would make any time measuring highly unreliable. To mitigate this issue, the tests environment utilized a mocked system that would return pre-made responses in a fixed response time.

This mock system should not alter Retrofit's behaviour in any way, and this was possible to achieve by overriding the Retrofit's default HTTP client. Internally, Retrofit uses OkHttp [86] to execute network calls. When building a Retrofit instance, developers can provide an OkHttpClient instance which is the HTTP client that Retrofit will use for network calls. This can be seen in Listing 6, if no client is passed to the builder, Retrofit will create one at the time of its build.

```
val service: IMovieServiceCoroutines by lazy {
    builder
        .client(MovieHttpClient.get())
        .build()
        .create(IMovieServiceCoroutines::class.java)
}
```

Listing 6. Building Retrofit instance with client override.

Overriding the OkHttpClient in a Retrofit instance can be very useful to have an HTTP client that is more suited to the project needs. The case study implementation uses it to provide the maximum number of requests in simultaneous. In this case, the test environment uses the OkHttpClient to leverage on the use of Interceptors [87].

Interceptor is an interface with a single method *intercept* that will run for every network request. The *intercept* method is expected to return a *Response* object which is OkHttp's model for a network response. It is a simple abstraction, the OkHttpClient expects a response from these interceptors and propagates it down to Retrofit, independently from where that response comes from (real network or not).

Every OkHttpClient instance has a list of interceptors where requests will go through. This list has always at least one interceptor, the CallServerInterceptor. This is the interceptor that will make the

actual call to the network and provide the network response. This is always the last interceptor in the list, meaning it is always the last one to run, if needed.

When building an OkHttpClient instance, developers can provide custom interceptors. This is a common practice in professional Android development to deal with authentication or logging requests/responses from the network. Most custom interceptors will most likely call *chain.proceed(request)* inside the *intercept* method, passing the request to the next Interceptor on the list, until it reaches CallServerInterceptor where the network call is executed.

In this case, the test environment will avoid using the CallServerInterceptor, by early returning a response instead of calling the network.

MockInterceptor is an implementation of Interceptor whose *intercept* method simply sleeps the current thread for 50 milliseconds and then returns a pre-selected response that matches what the application is expecting, ignoring the CallServerInterceptor. The specific interval of 50 milliseconds was an approximation of the real response time we were getting from the network but it is not relevant, the main goal is to keep the response time consistent across all requests done through Retrofit. The implementation of this class is represented in Listing 7.

```
class MockInterceptor : Interceptor {
    override fun intercept(chain: Interceptor.Chain): Response {
        Thread.sleep(50)
        return chain.request().getMockedResponse()
    }
}
```

Listing 7. MockInterceptor implementation.

Therefore, and for the purpose of testing, Retrofit will use an OkHttpClient that uses a MockInterceptor to intercept its requests.

By using this mocked OkHttpClient instance, every response will take 50 milliseconds to return, creating a consistent response time that will ensure reliability to the tests results. This logic is completely unknown to Retrofit. Retrofit will behave as if the call came from the network, which is exactly the scenario this research intends to test.

The role of MockInterceptor in the mocked network flow can be seen below in Figure 14, where it is visible how it does not delegate the request to the upper layer CallServerInterceptor. Instead, it directly provides a pre-set response downstream, acting as the network layer.

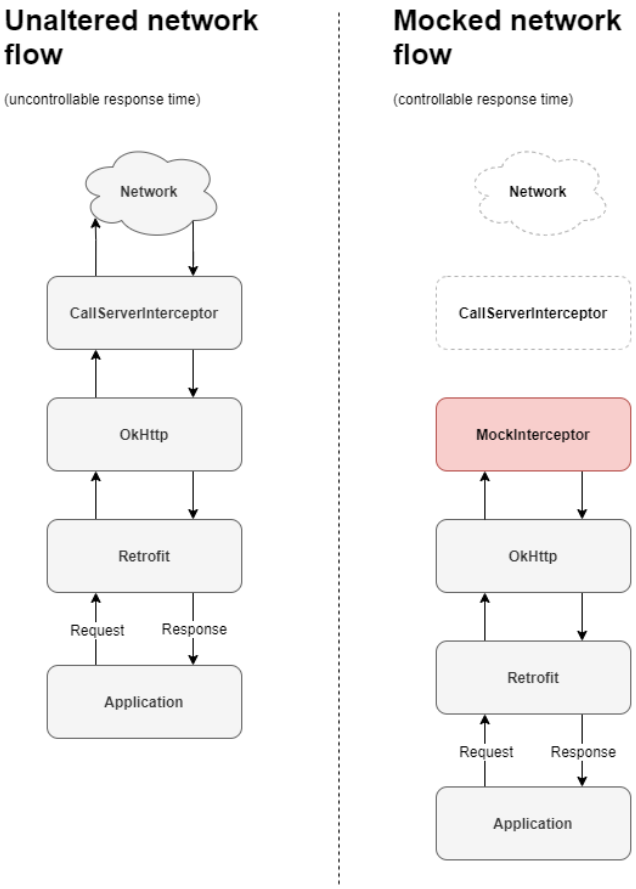

Figure 14. Mocked network flow overview.

The original instantiation of the mocked OkHttpClient can be seen in Listing 8, as well as its usage within the Retrofit build.

```
private val builder: OkHttpClient.Builder = OkHttpClient.Builder()
    .cache(null)
    .dispatcher(Dispatcher().apply { maxRequestsPerHost = 20 })

fun mock(): OkHttpClient = builder.addInterceptor(MockInterceptor())
    .build()
```

```
fun mock(): IMovieServiceCoroutines = builder
    .client(MovieHttpClient.mock())
    .build()
    .create(IMovieServiceCoroutines::class.java)
```

Listing 8. Building a mocked Retrofit client.

When fetching the user location, the original implementation uses Geocoder [88] to fetch address data from the coordinates returned by the GPS sensor. Since this request is not done through Retrofit, the MockInterceptor layer explained above will not have any effect.

The wrapping class surrounding the Geocoder request receives a parameter *mock* that will decide whether the service should access the network to fetch the addresses for a certain set of coordinates. If this parameter is true, the function will not fetch the addresses from the network, instead it returns immediately a list of mocked addresses that follows what the application is expecting. This behaviour can be seen in the AddressRepository class, under the KitProtocol module.

Having a mocked network layer helped the research by providing the possibility to test both Kotlin Coroutines and RxJava against the same conditions, preventing any instability related to response times from external services.

Benchmarking

The guide provided in [62] by the Android team to adopt the Jetpack Benchmark tool is very explicit and easy to follow up. This library has a very convenient integration with Android Studio by allowing developers to create a new module entirely dedicated to the benchmark with only a few clicks.

The generated module includes all the imports needed to use this library as well as a pre-setup of the configurations that are mandatory to run the benchmark correctly, out-of-the-box (e.g. the Activity that runs the benchmark tests must never be debuggable, and this can be done by setting the application *debuggable* flag to false on the benchmark module manifest) [85].

Having a dedicated module allowed the benchmark code to be contained from the application code. This makes sense because ideally, whereas the application module is tied to a specific context, the benchmark module is not. The benchmark module can test scenarios that are not implemented in the main application, allowing for a finer research through more granular tests.

Benchmark tests are still instrumented tests, meaning they still run in an Android application. This application is simply not the main application, it is sort of a placeholder application with a stall interface.

For this research, every single benchmark test will run 50 times and will include warmup runs (the number of warmup runs is variable and depends on the Android benchmark library algorithm) to keep a consistent processor performance throughout the measurement. This is the default behaviour provided by the benchmark library.

```
@Test
override fun insert_ten_movies() = benchmarkRule.measureRepeated {
    runWithTimingDisabled {
        runBlocking { localSource.suspendNuke() }
    }
    runBlocking {
        val movieEntities = List(10) { getMockEntity(it) }
        localSource.suspendInsert(movieEntities)
    }
}
```

Listing 9. Benchmark test example.

Listing 9 represents an example of a Benchmark test for Coroutines, but the behaviour is the same for RxJava. The measurements will be applied to the code inside the *benchmarkRule.measureRepeated* scope. The code must be blocking, otherwise the scope would return immediately, providing wrong test results.

Inside this scope, developers can also use *runWithTimingDisabled* to run any statement whose execution time will not be accounted in the final measurement. This is useful to eliminate any form of cache produced by previous runs. For instance, this example tests a database insertion. For consistency, the test will delete every record from the database before inserting new records, since the deletion code is running inside the *runWithTimingDisabled* scope. The result will exclude the time it took to delete these records, providing only the execution time for the insertion.

The tests are sub-divided into three categories inside the benchmark module:

- Network tests (NT1-NT6)
- Database tests (DT1-DT11)
- Integration tests (IBT1-IBT5)

Both Kotlin Coroutines and RxJava contain a class for each one of these categories and each library implementation features the same exact scenarios. All the source code for benchmarks can be found inside the benchmark module. Test classes implement a specific interface according to their category to keep naming consistency. This is helpful to automatically compare both the libraries using statistic tools as it will be mentioned further ahead.

- Network tests classes (implement INetworkBenchmark):
    - Kotlin Coroutines: CoroutinesNetworkBenchmark
    - RxJava: RxJavaNetworkBenchmark.
- Database test classes (implement IDatabaseBenchmark):
    - Kotlin Coroutines: CoroutinesDatabaseBenchmark
    - RxJava: RxJavaDatabaseBenchmark.
- Integration test classes (implement IIntegrationBenchmark):
    - Kotlin Coroutines: CoroutinesIntegrationBenchmark
    - RxJava: RxJavaIntegrationBenchmark

To run the benchmark tests, developers may simply run *connectedCheck*, a Gradle task that is a part of the Android plugin for Gradle. This task is frequently used to run instrumented tests in Android.

Unfortunately, using Android Studio 3.6.3, it was not possible to run tests individually as it would result in a timeout where the Android Studio could not detect the emulator's status. This seems to be happening for more users of this library [89], and could be avoided by using the command line instead of any user interface component inside the studio.

In order to obtain a full report of every run for each benchmark test there was the need to also add *"android.enableAdditionalTestOutput=true"* to the gradle.properties file, as stated in the documentation of the library for projects that use Android Gradle Plugin with version 3.6 or higher.

Running *connectedCheck* will run every benchmark test inside the benchmark module and provide a JSON report with details about the correspondent execution time. This JSON report contains an array

called “benchmarks”, where all the data about each benchmark comes from. This report provides the median, minimum, and maximum execution time in nanoseconds for each benchmark test as well as the exact execution time for every run within that test. In this case, each test runs 50 times, therefore, the report provided an array of 50 values.

An example of this benchmark report can be found below in Listing 10. The example is emphasizing the benchmark results, but the report also includes metadata about the device it ran on under the field “context”.

```
{
  "context" : ...,
  "benchmarks": [
    {
      "name": "insert_twenty_movies",
      "params": {},
      "className":
"com.example.benchmark.coroutines.database.CoroutinesDatabaseBenchmark",
      "totalRunTimeNs": 24521986455,
      "metrics": {
        "timeNs": {
          "minimum": 2866146,
          "maximum": 9556250,
          "median": 3316667,
          "runs": [
            4701562,
            4545834,
            ...
          ]
        }
      },
      "warmupIterations": 1269,
      "repeatIterations": 1,
      "thermalThrottleSleepSeconds": 0
    },
    {
      "name": ...
    }
  ]
}
```

Listing 10. Example of the benchmark JSON report, with emphasis to the benchmark results.

One major point of this research is also to provide a simple environment for any developer that wishes to run their own benchmarks and see the results immediately with a proper format that allows for easier conclusions. This process should not be tedious and should mostly be automatic, so one further step this research takes is to build a tool to automate the process of having the raw data from the benchmarks formatted into a simpler and more meaningful way to visualise data, using charts and tables.

This tool was developed specifically for this research and the process behind building it is described in Appendix A.

Hopefully, this approach will ease the work of any developer that wishes to build new benchmarks using this research’s environment by writing different tests specific to its needs and easily being able to see the results in tables, alongside charts that will possibly evidence more direct conclusions.

Profiling

Aside from looking at benchmark data, this research also aims to understand how Kotlin Coroutines and RxJava behave under a regular Android application condition. This section will portrait how the research went about profiling the case study. The results from the Android profiler merely served to complement the benchmark results for the integration tests.

The main tool for this job was Android Profiler, a built-in tool within Android Studio to monitor an Android application and show multiple metrics regarding a session, in real-time.

The metrics extracted from profiling the Android application were memory usage and the maximum of CPU percentage usage for the two main features of the app: fetching trending movies and fetching movies playing in the local theatres. Alongside both these metrics, there was an execution time metric that is calculated through the application code, using a timer software component that indicates how much time the feature request took to complete.

This section will explain the process of stabilizing the memory usage during manual testing and how the execution time was extracted to accompany other metrics provided by the Android profiler.

The profiler tests consisted of two sessions, one for Kotlin Coroutines and one for RxJava. Each session would then have three runs for each feature.

The allocated memory always depends on the system and there is no easy way to stabilize it for every single run, but there was an effort in starting every run with about the same base memory as the previous ones. The base memory for every run featured in this research was around 40 MB, for both features. This was achieved by forcing the JVM to garbage collect before every run, to try and stabilize the memory usage.

Forcing the garbage collection to run can be done with Android Profiler, but unfortunately it was not available for the test device used by this research. It had to be done through the application code using the System class [90] and the Runtime class [91], specifically the *gc()* method.

This method was invoked by clicking a menu item in the application's toolbar. This item is hidden by default and it becomes visible when the user long presses the top part of the screen where the title sits at. The Figure 15 represents the step to enable this item.

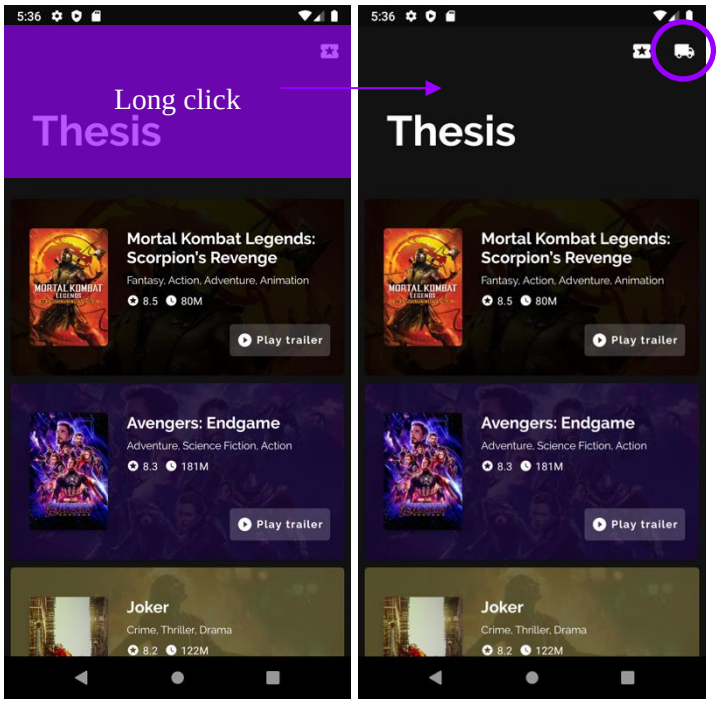

Figure 15. Enabling the garbage collector item.

The execution time was not extracted from the Android profiler. It is possible to do so since it provides the execution time for a certain thread. Nevertheless, it would be extremely unviable considering that both features run on multiple threads to complete. Through code however, it is easy to measure at what time the user made the request and at what time it received a response.

In order to keep it minimal and apply the same exact way of measuring the execution time for each run, an interface MovieTimer (can be found under the KitProtocol module) was created to act as a timer for each feature. This interface exposes four simple methods, that can be seen below in the Listing 11.

```
fun startTrendingMoviesTimer() {
    trendingRunNumber++
    startTrending = System.currentTimeMillis()
}

fun stopTrendingMoviesTimer() {
    log("Trending Run #$trendingRunNumber took ${System.currentTimeMillis() -
startTrending} ms.")
}

fun startLocalMoviesTimer() {
    localRunNumber++
    startLocal = System.currentTimeMillis()
}

fun stopLocalMoviesTimer() {
    log("Local movies Run #$localRunNumber took ${System.currentTimeMillis() -
startLocal} ms.")
}
```

Listing 11. MovieTimer exposed methods.

As the naming implies, there are both a method to start the timer and a method to stop it, for each feature.

The start method increments the run number for that feature, and it saves a timestamp of the time at which it was called in. The stop method logs the run number and the time it took to complete in milliseconds. The run number is useful to correlate the execution time and the data coming from the profiler, since it follows same order as the events seen in the profiler. Below, in the Listing 12, it is possible to see the usage of the timer for both Kotlin Coroutines and RxJava, respectively.

```
override fun fetchTrendingMovies() {
    startTrendingMoviesTimer()
    viewModelScope.launch {
        try {
            isLoading.value = true
            repository.fetchTrendingMovies()
            isLocalMovies.value = false
        } catch (t: Throwable) {
            message.value = context.getString(R.string.generic_movie_error)
            Log.e(LOG_TAG, "Could not fetch movies.", t)
        } finally {
            stopTrendingMoviesTimer()
            isLoading.value = false
        }
    }
}
```

```
override fun fetchTrendingMovies() {
    startTrendingMoviesTimer()
    disposableBag += repository.fetchTrendingMovies()
        .subscribeOn(Schedulers.io())
        .observeOn(AndroidSchedulers.mainThread())
        .doOnSubscribe { isLoading.value = true }
        .doFinally {
            stopTrendingMoviesTimer()
            isLoading.value = false
        }
        .subscribe(
            { isLocalMovies.value = false },
            { throwable ->
                message.value = context.getString(R.string.generic_movie_error)
                Log.e(LOG_TAG, "Could not fetch movies", throwable)
            }
        )
}
```

Listing 12. Examples of the timer interface usage.

As previously mentioned [65], measuring the execution time using this approach may not be as reliable as the benchmark approach. Nevertheless, it serves as an indicative value that could later emphasize any considerable difference between RxJava and Kotlin Coroutines.

6. Results

This section will present the results of this research regarding metrics for both Coroutines and RxJava libraries. Code outside the dedicated modules will not be analysed as it is common to both libraries. The results will be subdivided in two sections: *Static Analysis* and *Performance*. After presenting the results there will be a conclusion section to reflect on them.

The case study uses the following versions of software:

- Kotlin: *1.3.72*
- Kotlin Coroutines Core (with Android support): *1.3.9*
- RxJava: *2.2.16*
    - RxAndroid: *2.1.1*
    - RxJava adapter for Retrofit: *2.7.1*
- Retrofit: *2.7.1*
    - Gson converter for Retrofit: *2.7.1*
- Room: *2.2.4*
    - Room KTX (support for Kotlin Coroutines): *2.2.4*
    - Room RxJava: *2.2.4*

The results here presented might significantly vary when using different versions of the software or a different device.

6.1. Static analysis

This section will be dedicated to the results achieved through static analysis of the source code produced with Kotlin Coroutines and RxJava.

The results for the Lines of Code (LOC), Cognitive Complexity, and Cyclomatic Complexity can be found below in Chart 1 and Chart 2.

These results came from SonarQube. For the LOC metric, we had to manually subtract the lines related to imports and package since they are mostly irrelevant to evaluate the effort of building these components for both libraries.

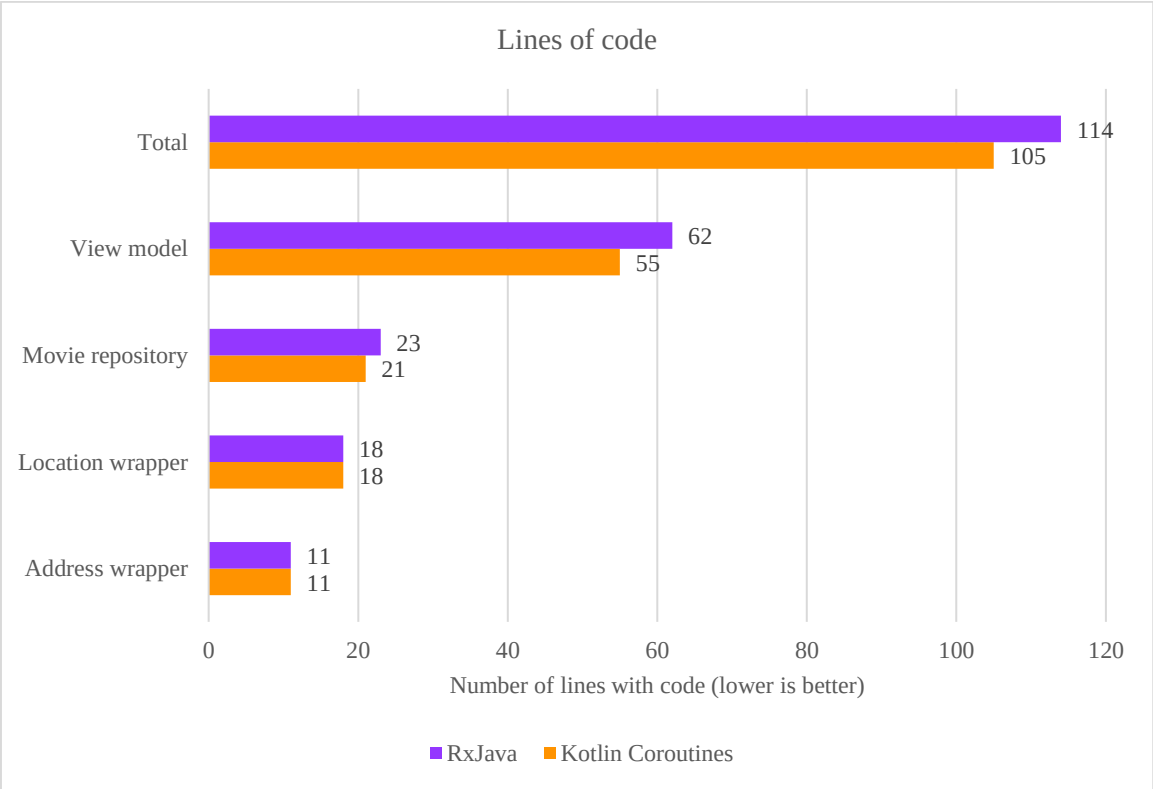

Chart 1. Lines of code metric results.

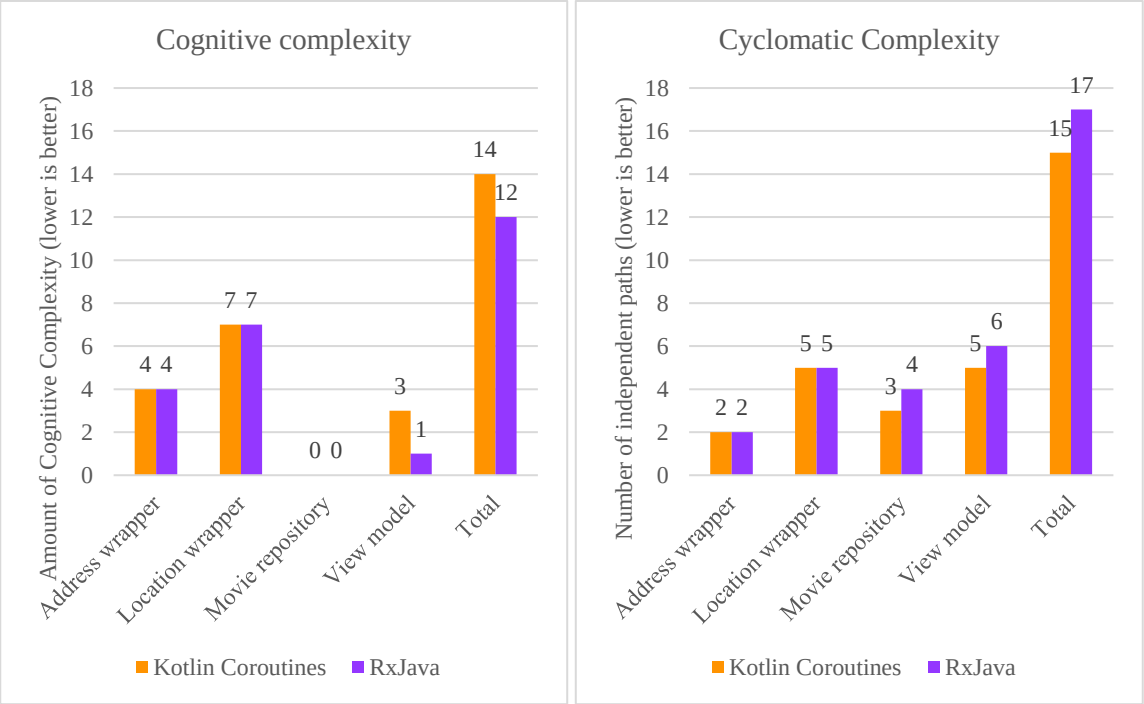

Chart 2. Cognitive Complexity (left) and Cyclomatic Complexity (right) metric results.

The wrapper components for both Kotlin Coroutines and RxJava share the exact same structure and therefore, the same values for the metrics considered. The differences between the usage of both libraries seem to reside in the ViewModel and movie Repository components.

RxJava performed better at the Cognitive Complexity metric and Kotlin Coroutines performed better at the remaining ones (LOC and Cyclomatic complexity).

The view model class for RxJava is expected to be slightly larger than the Kotlin Coroutines version due to RxJava not having an out-of-the-box integration with the Android ViewModel class. This requires developers to instantiate extra objects and override certain methods from the framework ViewModel class. For the Kotlin Coroutines version, the Android ViewModel class features, out-of-the-box, a *viewModelScope* object. This object is ready to be used for asynchronous tasks and handles cancellation under the bonnet, saving it a few lines of code. On the repository classes the difference between lines is minimal and could surely be none if the goal were to match both sums.

Despite the results stating that RxJava requires more lines to do the same tasks, this difference mostly origins from lack of direct integrations with the Android framework. The number of lines for RxJava could be easily lowered down by building a few abstractions around these classes or by using third-party libraries. These values could also mutate drastically if a different developer wrote the code. Being quite a volatile metric and considering the low difference in results, the LOC metric is not really defining any advantage in using any of the libraries over the other.

Considering the complexity metrics, the differences between the usage of both libraries are not substantial. RxJava seems to take the lead by 2 units on the Cognitive Complexity metric and Kotlin Coroutines does the same for Cyclomatic Complexity metric.

The difference in Cognitive Complexity only occurs when comparing the ViewModel component for Kotlin Coroutines and RxJava. Cognitive Complexity evaluates the complexity of the code and it is focused on nested statements/levels of indentation. Kotlin Coroutines does require, *for the most part*, more indentation levels than RxJava. For instance, when starting an asynchronous task, RxJava requires only an object (might be a *Completable*, *Flowable*, etc.) to start an RxJava chain and proceed with the asynchronous work, this keeps the indentation levels to the minimum. With Kotlin Coroutines, either a *launch* or an *async* call is required. Both *launch* and *async* constructs require a lambda as a parameter, and all the code meant to be asynchronous is required to be inside that lambda. Therefore, even for a basic asynchronous request, Kotlin Coroutines requires at least one extra level of indentation. On top of that extra level, it is very common to catch exceptions with a try/catch expression which adds in an extra indentation level. The view model component for Kotlin Coroutines does feature both the launch construct and the try/catch expression for error handling. Those could be the causes for this difference in Cognitive Complexity.

Cyclomatic Complexity, succinctly, measures the number of independent paths which a certain function can take until it returns. Each method has a minimum of one path, so we can expect the Cyclomatic Complexity of a class to be at least the number of methods it contains. This was not the case for the Kotlin Coroutines version of the movie repository. After some testing, we understood that SonarQube did not count the extension function *CoroutineScope.getMoviesDetails* as a normal function, and therefore, does not increment the Cyclomatic Complexity for the Kotlin Coroutines version. If we count it in, as we should, the number of paths for the movie repository component of

both Kotlin Coroutines and RxJava are the same. This lowers down the difference between both versions to one path. This remaining path resides in the RxJava view model component. It is expected that the RxJava version of this component results in one extra path since this class contains five methods whereas the Kotlin Coroutines version only contains four.

The static analysis results, *for the considered metrics and case study application*, do not represent any substantial advantage in using any of the libraries over the other.

6.2. Performance Analysis

This section will be dedicated to present the results related to the performance of Kotlin Coroutines and RxJava in Android development.

The results will be subdivided in three categories – network, database, and integration. Network and database categories feature solely benchmark tests. The integration category features both benchmark and profiling tests. Benchmark tests produce results for the execution time metric whereas profiling tests will feature other metrics like CPU and memory load.

6.2.1. Network

This section will explore the results for the network access performance tests. As previously mentioned, the results illustrated here come solely from benchmark tests. The setup of the test environment that ensures the benchmark tests are unbiased of network access instability is further detailed in Section 5.2.

The results for the network benchmark tests can be found below in Chart 3.

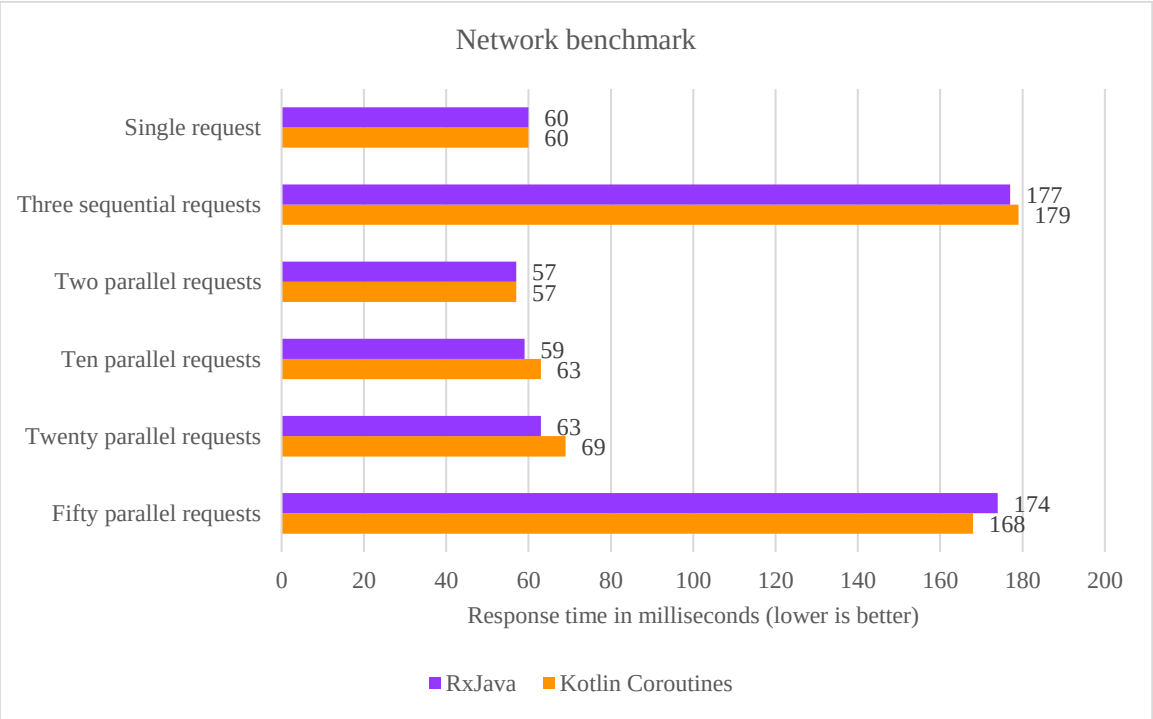

Chart 3. Network benchmark results.

Overall, RxJava performs better than Kotlin Coroutines. RxJava takes the advantage over Kotlin Coroutines on three of the performance tests, whilst the opposite only happens for one of the tests.

Analysing the results, both libraries performed equally for the single and the two parallel requests tests. The remaining tests had minor differences in response time, up to six milliseconds when executing twenty and fifty parallel requests. For the twenty parallel requests test, RxJava took six milliseconds less than the Kotlin Coroutines version whereas for the fifty parallel requests test it took six milliseconds more.

These tests are objectively measuring how much time it takes for Retrofit to access the network, parse the response, and finally deliver a Kotlin object to the scope of the network call. The test environment we setup has a static response time of 50 milliseconds to return a pre-established JSON response to Retrofit. This static response time plus the time it takes to parse the response explains why the results for parallel requests are around the sixty-millisecond mark, except for the fifty-parallel request.

The case study application limits Retrofit to twenty simultaneous network calls. When we request more than twenty parallel calls, those extra calls will join a queue and only execute when there is space in the twenty-call limited pool. This explains why the fifty parallel requests test derives in a considerably greater response time when compared to the other parallel requests.

Further exploring the parallel requests, it is possible to assess an average of how much time each request took for our set of tests by summing the times for the parallel requests and dividing it by 82, the total of requests for all the parallel requests tests. RxJava took an average of 4,30 milliseconds per request and Kotlin Coroutines took, on average, 4,35 milliseconds per request.

In conclusion, although RxJava performed better overall, Kotlin Coroutines had very similar results with the difference between them being most-likely unnoticeable to an end-user.

6.2.2. Database

This section will present the results for the database access tests. The results for the database benchmark tests can be found below in Chart 4.

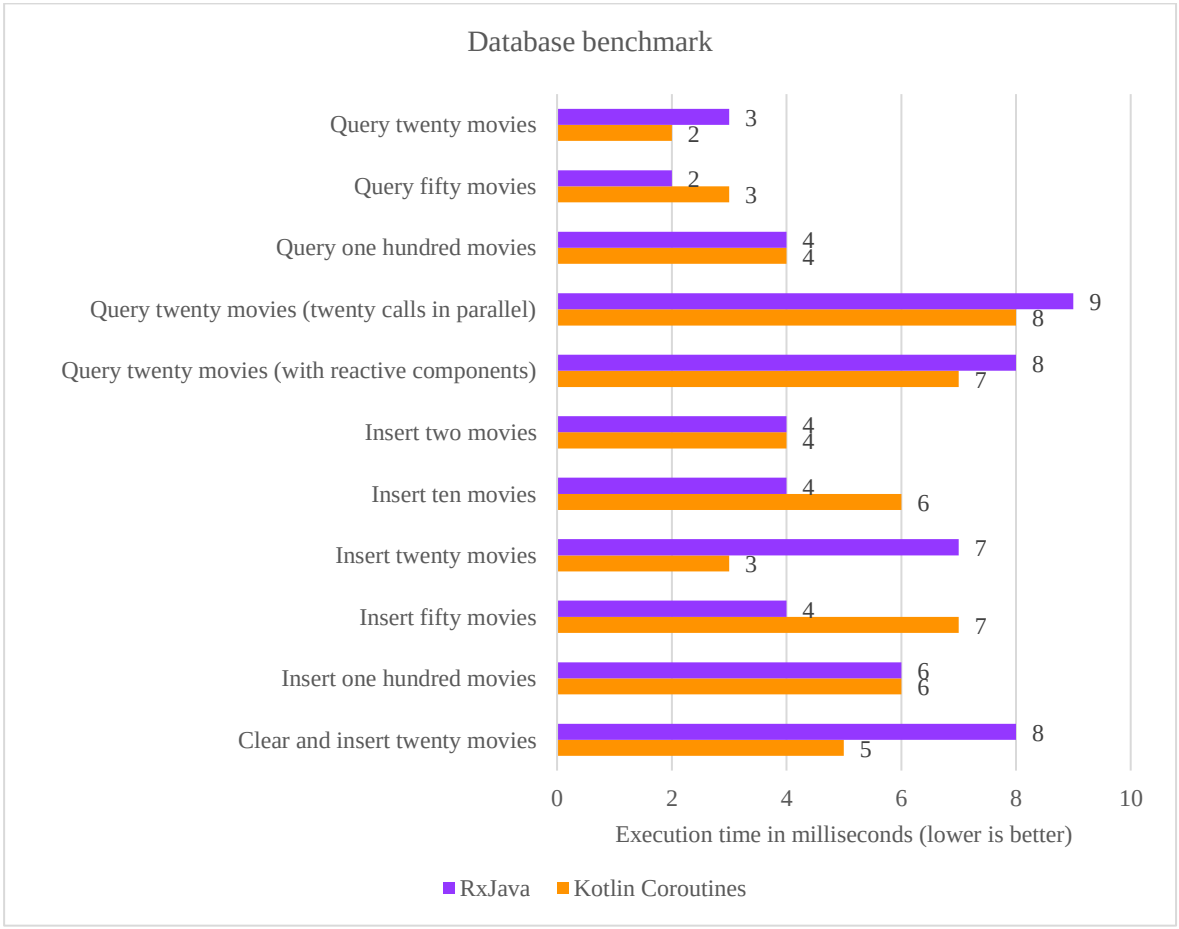

Chart 4. Database benchmark results.

Analysing these results, they are very similar for both Kotlin Coroutines and RxJava. Not only the results are close between both libraries, but it is also hard to find a consistent progression from test to test. Both Kotlin Coroutines and RxJava seem to have erratic evolutions.

For instance, for RxJava, the average execution time to insert twenty movies was greater than the average time it took to insert one hundred movies. These inconsistencies can be found for both RxJava and Kotlin Coroutines, making these results unpredictable and hard to draw conclusions from.

Summing the times for every test, Kotlin Coroutines did 55ms and RxJava did 59ms. The maximum difference between the two libraries in a test is four milliseconds when inserting twenty movies.

Kotlin Coroutines did better for this set of runs but the low difference in execution time and the disparity in the results are not assertive enough to recommend a library over the other for database access.

6.2.3. Integration

This section explores the results for the integration tests. Integration tests feature both benchmarking and profiling tests.

The results for the benchmark tests can be seen below in Chart 5.

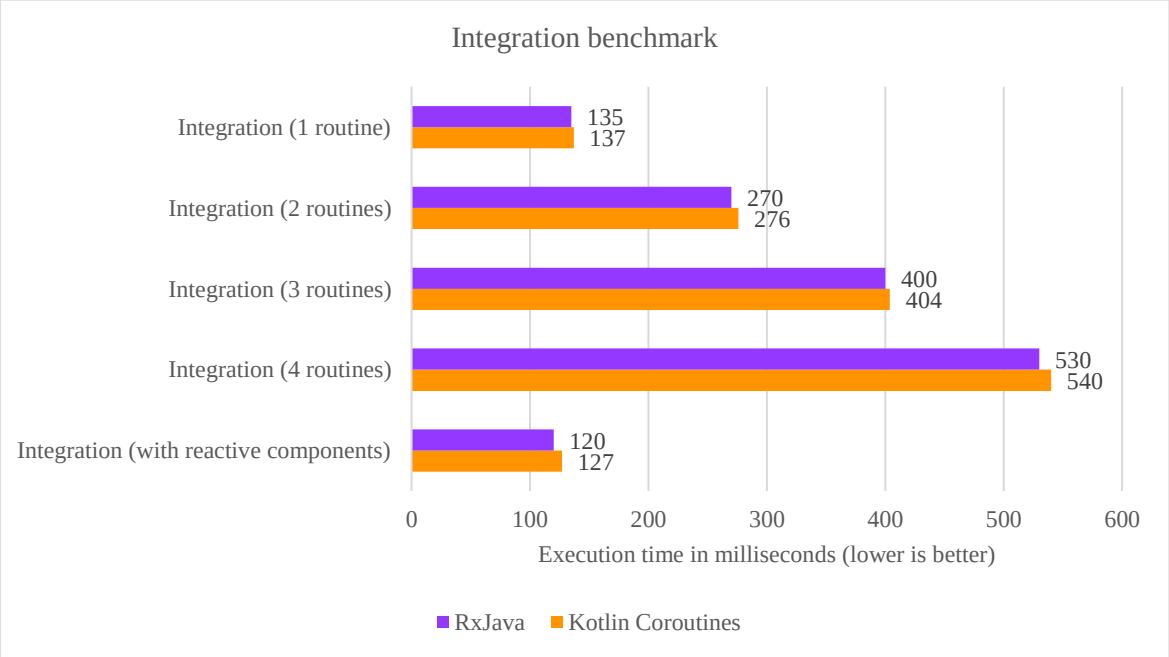

Chart 5. Integration benchmark results.

According to the benchmark results, there is a consistent evolution for both libraries and RxJava has shorter execution times than Kotlin Coroutines for every test.

To better understand the integration evolution, we must not focus solely on the execution time. If the database and network accesses are faster for a certain library, the execution time for the integration of both will most likely be faster as well.

Using a routine as the unit of measure, we can assess how much it costs, on average, to add one routine to a sequence. Taking the Kotlin Coroutines results, running one routine by itself takes 137ms. Running two integrated routines costs 276ms, therefore, adding one routine to the sequence cost 139ms, which represents an increase of 1,01% regarding the original 137ms. Continuing this procedure for the remaining tests, adding a new routine to the two routines sequence costs 128ms, which is 0,93% below the original 137ms and adding a fourth routine to the three routine sequence

cost 136ms which represents a decrease of 0,99% regarding the original 137ms. For Kotlin Coroutines, on average, adding a new routine to the sequence costs approximately 0,98% of the time for a single routine, and 0,97% for RxJava.

Although the differences are not of considerable amount, it is safe to say RxJava is slightly more optimized for integration than Kotlin Coroutines for our set of benchmark tests.

To enrich the results from benchmarking, we ran integration tests using the Android Profiler. The process to extract these results is deeply explained in Section 5.2. The profiling results are divided per feature of the application. Each run presented in Chart 6 and Chart 7 is a sequence of tasks which the case study application must perform to fully execute a feature.

The trending movies feature requires one call to the network to fetch the list of movies, followed by multiple parallel network calls to fetch the details for each movie in the list, an access to the database to persist the details and, finally, a push from the reactive component attached to the local database of the new data upwards to the view layer.

The local movies feature will do everything the trending movies feature does and more. Instead of simply making a network call to fetch a list of movies, it will first grab the user coordinates through a reactive component, then fetch the code of the country the user is located on, and only then proceed with the same steps as the trending movies feature.

Each feature had three runs, and the discussed results will be the average values from those runs. The metrics considered for the profiling tests were memory usage, maximum CPU usage and execution time. The memory usage charts do not represent the full amount of memory the application used when running these features. Instead, it represents the difference between the highest value of memory usage and the initial value of memory usage for when the feature started running.

The results for the trending movies profiling session can be found below in Chart 6.

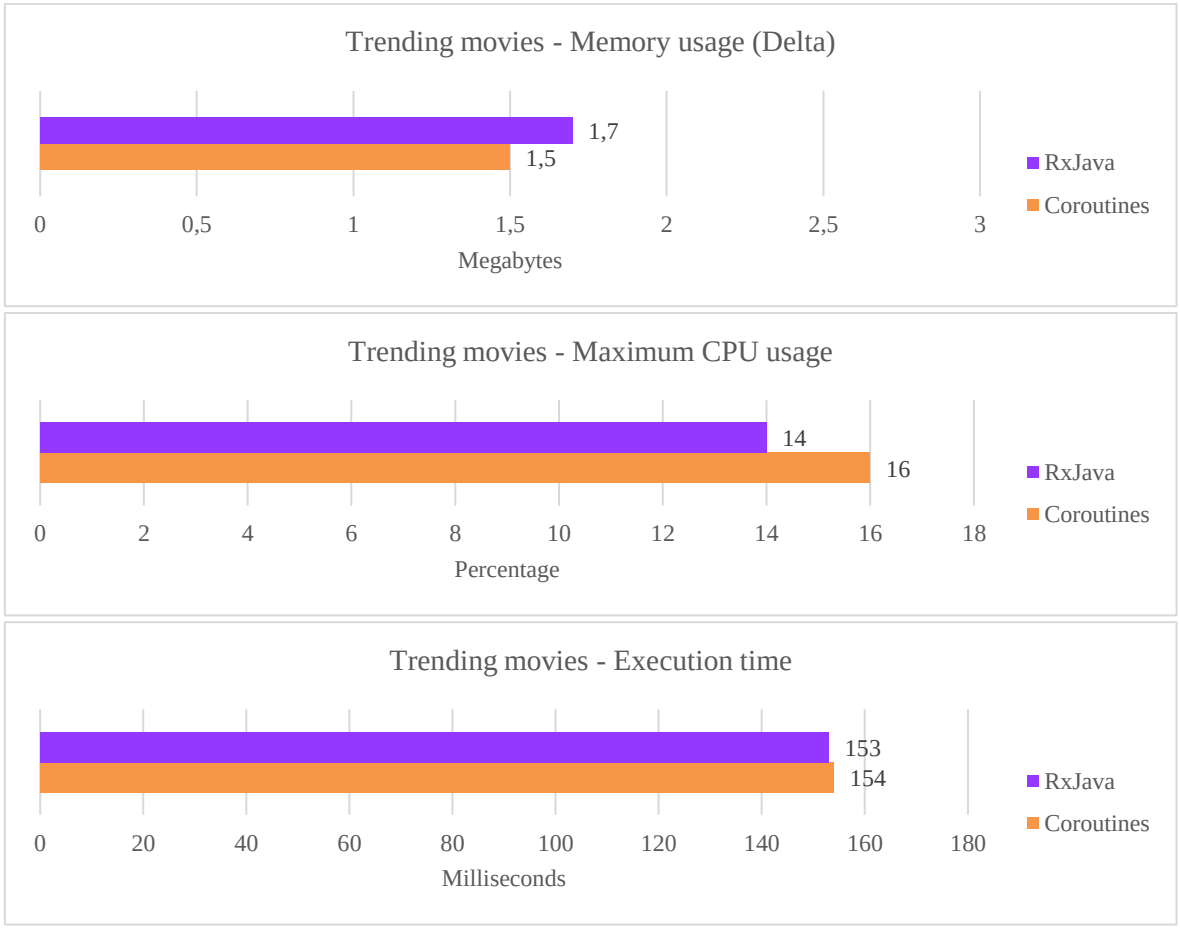

Chart 6. Trending movies profiling results.

Focusing on the trending movies feature, all the metrics considered for profiling have similar results for Kotlin Coroutines and RxJava. Kotlin Coroutines used less memory, but as a trade-off it had a taller peak of CPU usage, about 2% more. On average Kotlin Coroutines did about the same time as RxJava for this feature which does not define a clear advantage for any of the libraries.

The local movies feature results can be found below in Chart 7.

Chart 7. Local movies profiling results.

Kotlin Coroutines used slightly less memory but it also had the highest CPU usage peak at 18%. Despite the difference in resources usage, the execution time was about the same for both libraries when fetching the local movies.

Overall, the performance for both libraries in the integration tests (both benchmark and profiler) is about the same with RxJava taking a slight edge in execution time for the benchmark tests.

6.3. Summary

Referring to the static analysis tests, the results do not represent any substantial advantage in using a library over the other. Most of the difference in Lines of Code and Cyclomatic Complexity between RxJava and Kotlin Coroutines came from RxJava not having an out-of-the-box integration with the ViewModel class in the Android framework. This could easily be solved by writing some code to mimic the same integration for RxJava. For Cognitive Complexity, RxJava stated lower values than Kotlin Coroutines. As previously explained, Cognitive Complexity focus heavily on nested statements and launching a coroutine will, for the most part, take an extra level of indentation when compared to starting an RxJava chain. This is not meant to question the reliability of this metric but when looking at the code for the Kotlin Coroutines view model version, it does not look any more complex for having these extra levels of indentation when compared to RxJava.

In conclusion, *for the set of considered static analysis metrics*, the values for both Kotlin Coroutines and RxJava are not distant from each other and were fairly justified during their analysis, leading us to believe they are not assertive enough to state that one of the libraries will inherently result in higher costs of software maintainability for an Android application.

Referring to the performance tests, the results state that RxJava has slightly better performance than Kotlin Coroutines when integrating with Retrofit for network requests and when using multiple components to compose a larger sequence of tasks. When benchmarking, the maximum difference between both libraries across tests for all categories (network, database, and integration) was 10 milliseconds.

When systems have a response time of 0,1 seconds (100 milliseconds) or less, humans perceive it as being *instantaneous* [91, 92]. With this information, it is possible to add that having a maximum difference of 10 milliseconds in execution times between both libraries will hardly be noticeable to the end-user and therefore, does not define a substantial advantage in using one library over the other.

In conclusion, if an application is time sensitive and must be extremely optimized to reproduce the best times possible, RxJava seems to perform better for that case. For regular Android applications, the differences between Kotlin Coroutines and RxJava will most likely not be noticeable as they had very minimal differences for most of our tests.

Considering our case study application and our results, Kotlin Coroutines and RxJava do not seem to differently compromise the *Maintainability* or *Performance* of an Android application when compared to each other. Considering those two quality attributes, developers, and the entities responsible for developing and publishing an Android application should not worry about choosing between these thread management libraries. Neither of them should technically limit the application to provide the best possible user experience.

7. Conclusion

After studying previous work which researched the top user complaints and most common performance issues in Android applications [3, 4], it was possible to correlate some of them to having a faulty concurrency system. Because faulty concurrency systems are visible to the end-user, they may have a negative impact not only in the user experience but consequently on the business surrounding the software product. For having a clear impact outside the codebase, we realised that ensuring a proper concurrency strategy should be a concern not only for developers but also for any human resources responsible for the funding, development and publishing of an Android application.

Further validating the importance of concurrency and considering the complexity of manually handling all the aspects that come with it, there has been significant work in developing libraries that strive to help developers implementing a proper concurrency system in their applications [5–11]. Some of these libraries were developed exclusively for Android by the Android team, and others are not bound to the operating system nor the platform they run on, but can easily integrate with Android development with the use of third-party libraries (e.g. RxJava is not exclusive to Android but seamlessly integrates with the platform via the use of RxAndroid, a library that serves as a bridge between RxJava and the Android ecosystem).

Although multiple solutions have been created to serve the purpose of helping development teams with concurrency, not all withstood the test of time. During the last decade, some of these approaches fell in disuse or became deprecated. In order to compose a pertinent research, we took on a path to discard some of the available thread management libraries for Android with a valid and robust state-of-the-art research that demonstrates why some of those libraries do not provide what is expected for modern Android development and consequently, why some approaches have been elected over others (RQ1) (O1). This allowed to us to move on with a better idea of what developers find important to be featured in a thread management library and consequently, what were the most relevant thread management libraries to further investigate: RxJava and Kotlin Coroutines.

Then, we evaluated how using a thread management library could impact the software quality of an Android application [52, 53]. This further led us to the two most pertinent software quality attributes to consider for Android development (RQ2): Performance and Maintainability. The use of thread management libraries can also affect the software product on a functional level. Not only because some of these libraries do not expose the same functionality but also because they may impact the modifiability of the software. Decreased modifiability limits the speed and stability of developing new features and revising/improving older behaviour. Overall, the usage of thread management libraries may affect not only the experience of the final consumer but also influence the company's possibility to maintain/improve the software product (RQ3).

The research then proceeded with the comparison of Kotlin Coroutines and RxJava. To best assess if any of these libraries had a clear advantage over the other for concurrency in Android, we decided to compare them using a case-study application.

Before building the application, we started by tracking some non-functional requirements from an Android official guide for architecture [25]. By following this guide, we ensured our case study application met the current trends for modern Android development and by doing so, we could also retrieve some common use cases for concurrency in a modern architecture (O2). With a set of non-functional requirements which the case study Android application should meet, we then moved on to develop a set of features that require the use of concurrency using the pre-defined thread management libraries, Kotlin Coroutines and RxJava (O3).

Assessing the most relevant quality attributes, Performance and Maintainability, allowed the case-study application and the tests design to emphasize those attributes and form a more relevant set of tests to investigate potential advantages in using RxJava or Kotlin Coroutines (RQ4). This process was then followed by the actual setup of the tools we needed for testing as well as the development of the test cases and their execution (O4).

Having used different tools for testing purposes [59, 61, 62], the results for the comparison needed to be further formatted to be presented in a way they made sense and were easy to interpret. This was achieved using charts which presented the results relative to each thread management library on each test, side by side (O5).

Finally, we concluded on the results to understand if RxJava and Kotlin Coroutines would differently impact the performance and maintainability of an Android application (O6) (RQ5). After a technical review of the results, we also gauged their impact to the people involved in software production and end-users (O7).

Our work shows that choosing one option from Kotlin Coroutines and RxJava does not seem to differently compromise the *Maintainability* or *Performance* of an Android application as the results for our set of tests do not define any substantial advantage in using one library over the other.

Functionality wise, for *our case study application*, we did not find any limitation from these libraries either. Both were able to provide the necessary components to implement the proposed features with concurrency.

The process that was taken to retrieve the results for both Kotlin Coroutines and RxJava came from a set of tests that can easily be extended to test other thread management libraries or by adding new tests to the current set. We find this useful as any interested reader will be able to run their own tests using Kotlin Coroutines, RxJava or other libraries which it wishes to compare. Also, by having this somewhat agnostic setup, it will also be possible to compare any newer version that might be released for these libraries and further understand if the results point to new conclusions.

It could be interesting to use different case study applications and different domains to explore if the conclusions are different and why. Extending the same concept, using different software attributes and different metrics could complement this research. Attributes like community, support, or Developer Experience (DX) may be worth to evaluate and be a deciding factor for thread management libraries.

Using thread management libraries like RxJava or Kotlin Coroutines is not sufficient on its own to have good values for Performance and Maintainability on Android applications. Elaborating a

document with best practices examples for each thread management library could help developers produce code with greater quality when using these libraries.

Expanding this same study to other development platforms like iOS and Web (both back-end and front-end development) should also help developers of those areas to discretize the best approaches they can take to concurrency. Assumingly, not all these sectors may be able make use or have Kotlin Coroutines or RxJava as standard thread management libraries, therefore the evaluation for what libraries to test should be re-done specifically to each sector.

References

[1] H. Sutter and J. Larus, “Software and the Concurrency Revolution,” 2005.

[2] Android Developers, “Processes and threads overview | Android Developers.” [Online]. Available: https://developer.android.com/guide/components/processes-and-threads. [Accessed: 09-Nov-2019].

[3] Y. Liu, C. Xu, and S.-C. Cheung, “Characterizing and Detecting Performance Bugs for Smartphone Applications,” 2014.

[4] H. Khalid, E. Shihab, M. Nagappan, and A. E. Hassan, “What Do Mobile App Users Complain About?”

[5] Android Developers, “AsyncTask | Android Developers.” [Online]. Available: https://developer.android.com/reference/android/os/AsyncTask. [Accessed: 24-Nov-2019].

[6] Android Developers, “AsyncTaskLoader | Android Developers.” [Online]. Available: https://developer.android.com/reference/android/support/v4/content/AsyncTaskLoader.html. [Accessed: 24-Nov-2019].

[7] Android Developers, “IntentService | Android Developers.” [Online]. Available: https://developer.android.com/reference/android/app/IntentService. [Accessed: 24-Nov-2019].

[8] Android Developers, “HandlerThread | Android Developers.” [Online]. Available: https://developer.android.com/reference/android/os/HandlerThread. [Accessed: 24-Nov-2019].

[9] Oracle, “CompletableFuture (Java Platform SE 8).” [Online]. Available: https://docs.oracle.com/javase/8/docs/api/java/util/concurrent/CompletableFuture.html. [Accessed: 24-May-2020].

[10] ReactiveX Community, “GitHub - ReactiveX/RxJava: RxJava – Reactive Extensions for the JVM – a library for composing asynchronous and event-based programs using observable sequences for the Java VM.” [Online]. Available: https://github.com/ReactiveX/RxJava. [Accessed: 24-Nov-2019].

[11] Kotlin Team, “Coroutines Overview - Kotlin Programming Language.” [Online]. Available:

https://kotlinlang.org/docs/reference/coroutines-overview.html. [Accessed: 09-Nov-2019].

[12] Kotlin Team, “Kotlin (programming language).”

[13] Android team, “AsyncTask | Android Developers (onPostExecute).” [Online]. Available: https://developer.android.com/reference/android/os/AsyncTask#onPostExecute(Result). [Accessed: 20-Jun-2020].

[14] Android Developers, “Intent | Android Developers.” [Online]. Available: https://developer.android.com/reference/android/content/Intent. [Accessed: 24-Nov-2019].

[15] Android Developers, “BroadcastReceiver | Android Developers.” [Online]. Available: https://developer.android.com/reference/android/content/BroadcastReceiver. [Accessed: 24-Nov-2019].

[16] Android Developers, “Thread | Android Developers.” [Online]. Available: https://developer.android.com/reference/java/lang/Thread. [Accessed: 24-Nov-2019].

[17] Android Developers, “Runnable | Android Developers.” [Online]. Available: https://developer.android.com/reference/java/lang/Runnable. [Accessed: 24-Nov-2019].

[18] Android Developers, “Handler | Android Developers.” [Online]. Available: https://developer.android.com/reference/android/os/Handler. [Accessed: 24-Nov-2019].

[19] Android Developers, “ThreadPoolExecutor | Android Developers.” [Online]. Available: https://developer.android.com/reference/java/util/concurrent/ThreadPoolExecutor. [Accessed: 24-Nov-2019].

[20] Y. Lin, C. Radoi, and D. Dig, “Retrofitting Concurrency for Android Applications through Refactoring.”

[21] Y. Lin, S. Okur, and D. Dig, “Study and Refactoring of Android Asynchronous Programming (T),” in *2015 30th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, 2015, pp. 224–235.

[22] Oracle, “Thread (Java Platform SE 7),” *Java Platform SE 7 Official API*, 2014. [Online]. Available: https://docs.oracle.com/javase/7/docs/api/java/lang/Thread.html. [Accessed: 20-Jun-2020].

[23] “…/AsyncTask.java · Gerrit Code Review.” [Online]. Available: https://android-review.googlesource.com/c/platform/frameworks/base/+/1156409/6/core/java/android/os/A

syncTask.java. [Accessed: 24-Nov-2019].

[24] "Deprecate IntentService (I1a91afee) · Gerrit Code Review." [Online]. Available: https://android-review.googlesource.com/c/platform/frameworks/base/+/1165344. [Accessed: 24-Nov-2019].

[25] Android Developers, "Guide to app architecture | Android Developers," *Android Developers*, 2018. [Online]. Available: https://developer.android.com/jetpack/docs/guide. [Accessed: 16-Feb-2020].

[26] Oracle, "LocalDate (Java Platform SE 8)." [Online]. Available: https://docs.oracle.com/javase/8/docs/api/java/util/concurrent/Future.html. [Accessed: 24-May-2020].

[27] J. Bonér, D. Farley, R. Kuhn, and M. Thompson, "The Reactive Manifesto," *Reactivemanifesto.Org*, vol. 2, no. 16 September 2014, 2014.

[28] ReactiveX community, "GitHub - ReactiveX/RxAndroid: RxJava bindings for Android." [Online]. Available: https://github.com/ReactiveX/RxAndroid. [Accessed: 26-May-2020].

[29] "Observer pattern - Wikipedia." [Online]. Available: https://en.wikipedia.org/wiki/Observer_pattern. [Accessed: 24-Nov-2019].

[30] ReactiveX Community, "ReactiveX - Operators," *reactivex Documentation*. [Online]. Available: http://reactivex.io/documentation/operators.html. [Accessed: 24-Nov-2019].

[31] ReactiveX Community, "ReactiveX - Observable," 2014. [Online]. Available: http://reactivex.io/documentation/observable.html. [Accessed: 24-Nov-2019].

[32] Maxim Shafirov, "Kotlin on Android. Now official | Kotlin Blog," 2017. [Online]. Available: https://blog.jetbrains.com/kotlin/2017/05/kotlin-on-android-now-official/. [Accessed: 09-Jun-2020].

[33] Kotlin Team, "Releases · Kotlin/kotlinx.coroutines." [Online]. Available: https://github.com/Kotlin/kotlinx.coroutines/releases?after=1.2.0. [Accessed: 09-Nov-2019].

[34] "Coroutine - Wikipedia." [Online]. Available: https://en.wikipedia.org/wiki/Coroutine. [Accessed: 09-Nov-2019].

[35] C. T. Haynes and D. P. Friedman, "Continuations and Coroutines."

[36] M. Moskala, "Marcin Moskała - Understanding Kotlin Coroutines - YouTube." [Online].

Available: https://www.youtube.com/watch?v=DOoJnJJnAG4. [Accessed: 25-Nov-2019].

[37] Android Team, “Kotlin Coroutines 101 - Android Conference Talks - YouTube.” [Online]. Available: https://www.youtube.com/watch?v=ZTDXo0-SKuU. [Accessed: 24-May-2020].

[38] Android Team, “Suspend functions - Kotlin Vocabulary - YouTube.” [Online]. Available: https://www.youtube.com/watch?v=IQf-vtIC-Uc. [Accessed: 24-May-2020].

[39] Kotlin team, “Basics - Kotlin Programming Language.” [Online]. Available: https://kotlinlang.org/docs/reference/coroutines/basics.html#structured-concurrency. [Accessed: 30-May-2020].

[40] R. Elizarov, “Structured concurrency - Roman Elizarov - Medium.” [Online]. Available: https://medium.com/@elizarov/structured-concurrency-722d765aa952. [Accessed: 30-May-2020].

[41] Kotlin Team, “Asynchronous Flow - Kotlin Programming Language.” [Online]. Available: https://kotlinlang.org/docs/reference/coroutines/flow.html#asynchronous-flow. [Accessed: 25-May-2020].

[42] R. Elizarov, “KotlinConf 2019: Asynchronous Data Streams with Kotlin Flow by Roman Elizarov - YouTube.” [Online]. Available: https://www.youtube.com/watch?v=tYcqn48SMT8. [Accessed: 24-May-2020].

[43] “Reactive Streams.” [Online]. Available: https://www.reactive-streams.org/. [Accessed: 25-May-2020].

[44] Android Developers, “Improve app performance with Kotlin coroutines | Android Developers.” [Online]. Available: https://developer.android.com/kotlin/coroutines. [Accessed: 09-Nov-2019].

[45] Android Developers, “Use Kotlin coroutines with Architecture components.” [Online]. Available: https://developer.android.com/topic/libraries/architecture/coroutines. [Accessed: 09-Nov-2019].

[46] Android Developers, “Using Kotlin Coroutines in your Android App.” [Online]. Available: https://codelabs.developers.google.com/codelabs/kotlin-coroutines/#0. [Accessed: 09-Nov-2019].

[47] Android Developers, “Room Persistence Library | Android Developers,” 2018. [Online]. Available: https://developer.android.com/topic/libraries/architecture/room. [Accessed: 16-

Feb-2020].

[48] Retrofit Community, “square/retrofit: Type-safe HTTP client for Android and Java by Square, Inc.” [Online]. Available: https://github.com/square/retrofit. [Accessed: 16-Feb-2020].

[49] wikipedia, “Iso 25010,” *Iso 25000*, 2015. [Online]. Available: https://iso25000.com/index.php/en/iso-25000-standards/iso-25010. [Accessed: 31-May-2020].

[50] ISO/IEC, “ISO/IEC 25010:2011 - Systems and software engineering -- Systems and software Quality Requirements and Evaluation (SQuaRE) -- System and software quality models,” 2011. [Online]. Available: https://www.iso.org/standard/35733.html. [Accessed: 31-May-2020].

[51] “ISO/IEC 9126 – Wikipedia,” *Wikipedia*, 2018. [Online]. Available: https://en.wikipedia.org/wiki/ISO/IEC_9126. [Accessed: 30-May-2020].

[52] I. Ozkaya, L. Bass, and R. L. Nord, “focus quality requirements Making Practical Use of Quality Attribute Information.”

[53] J. M. Fernandes and A. L. Ferreira, “Quality attributes for mobile applications,” in *Modern Software Engineering Methodologies for Mobile and Cloud Environments*, 2016, pp. 141–154.

[54] “Cyclomatic complexity - Wikipedia.” [Online]. Available: https://en.wikipedia.org/wiki/Cyclomatic_complexity. [Accessed: 09-Nov-2019].

[55] G. JAY, J. E. HALE, R. K. SMITH, D. HALE, N. A. KRAFT, and C. WARD, “Cyclomatic Complexity and Lines of Code: Empirical Evidence of a Stable Linear Relationship,” *J. Softw. Eng. Appl.*, vol. 02, no. 03, pp. 137–143, Oct. 2009.

[56] G. Gill and C. Kemerer, “Cyclomatic complexity density and software maintenance productivity,” *Softw. Eng. IEEE Trans.*, vol. 17, pp. 1284–1288, 1992.

[57] SonarSource, “Code Quality and Security | Developers First | SonarSource.” [Online]. Available: https://www.sonarsource.com/. [Accessed: 26-May-2020].

[58] G. A. Campbell, “COGNITIVE COMPLEXITY,” 2018.

[59] SonarSource, “Code Quality and Security | SonarQube.” [Online]. Available:

https://www.sonarqube.org/. [Accessed: 12-Apr-2020].

[60] SonarSource, “SonarLint | Fix issues before they exist,” 2019. [Online]. Available: https://www.sonarlint.org/. [Accessed: 29-Sep-2020].

[61] Android Developers, “Measure app performance with Android Profiler | Android Developers,” no. November, pp. 1–5, 2018.

[62] Android Developers, “Benchmark app code | Android Developers.” [Online]. Available: https://developer.android.com/studio/profile/benchmark. [Accessed: 12-Apr-2020].

[63] B. Goetz, “Java theory and practice: Anatomy of a flawed microbenchmark,” *IBM Dev.*, 2005.

[64] J. Gil Keren Lenz and Y. Shimron, *A Microbenchmark Case Study and Lessons Learned*. .

[65] B. Goetz, “Java theory and practice: Dynamic compilation and performance measurement.” 2004.

[66] “With the addition of memory benchmarking, rename some API [149979716] - Visible to Public - Issue Tracker.” [Online]. Available: https://issuetracker.google.com/issues/149979716. [Accessed: 26-May-2020].

[67] “Investigate adding memory perf/memory pressure metrics [133147125] - Visible to Public - Issue Tracker.” [Online]. Available: https://issuetracker.google.com/issues/133147125. [Accessed: 26-May-2020].

[68] Android Team, “Understand Kotlin Coroutines on Android (Google I/O’19) - YouTube.” [Online]. Available: https://www.youtube.com/watch?v=BOHK_w09pVA. [Accessed: 17-Feb-2020].

[69] E. Ferrance, “THEMES IN EDUCATION ACTION RESEARCH Northeast and Islands Regional Educational Laboratory At Brown University,” 2000.

[70] D. J. A. Edwards, “Types of case study work: A conceptual framework for case-based research 1.”

[71] “Google Scholar.”

[72] Google, “Google Codelabs,” 2016. [Online]. Available: https://codelabs.developers.google.com/. [Accessed: 22-Jul-2020].

[73] G. Almeida, “6uilhermeAlmeida/Thesis,” 2020. [Online]. Available: https://github.com/6uilhermeAlmeida/Thesis. [Accessed: 09-Nov-2020].

[74] Android Developers, “Android Developers Blog: Google I/O 2019: Empowering developers to build the best experiences on Android + Play.” [Online]. Available: https://android-developers.googleblog.com/2019/05/google-io-2019-empowering-developers-to-build-experiences-on-Android-Play.html. [Accessed: 16-Feb-2020].

[75] Docs Microsoft, “The MVVM Pattern | Microsoft Docs,” *docs.microsoft.com*, 2018. [Online]. Available: https://docs.microsoft.com/en-us/previous-versions/msp-n-p/hh848246(v=pandp.10)?redirectedfrom=MSDN. [Accessed: 13-Jul-2020].

[76] Docs Microsoft, “Introduction to Model/View/ViewModel pattern for building WPF apps | Microsoft Docs.” [Online]. Available: https://docs.microsoft.com/en-us/archive/blogs/johngossman/introduction-to-modelviewviewmodel-pattern-for-building-wpf-apps. [Accessed: 13-Jul-2020].

[77] SQLite.org, “SQLite Home Page.” p. One, 2014.

[78] Retrofit Community, “Add first-party Kotlin coroutine suspend support by JakeWharton · Pull Request #2886 · square/retrofit.” [Online]. Available: https://github.com/square/retrofit/pull/2886/commits/b761518aa174c7b0512b73f2fe70e2e908f24081. [Accessed: 12-Apr-2020].

[79] Retrofit Community, “retrofit/retrofit-adapters/rxjava2 at master · square/retrofit.” [Online]. Available: https://github.com/square/retrofit/tree/master/retrofit-adapters/rxjava2. [Accessed: 12-Apr-2020].

[80] Edward Hieatt and Rob Mee, “P of EAA: Repository.” [Online]. Available: https://martinfowler.com/eaaCatalog/repository.html. [Accessed: 14-Jul-2020].

[81] Google, “FusedLocationProviderClient | Google APIs for Android.” [Online]. Available: https://developers.google.com/android/reference/com/google/android/gms/location/FusedLocationProviderClient.html. [Accessed: 28-Jul-2020].

[82] Retrofit Community, “CallAdapter.Factory (Retrofit 2.7.1 API).” [Online]. Available: https://square.github.io/retrofit/2.x/retrofit/retrofit2/CallAdapter.Factory.html. [Accessed: 28-Jul-2020].

[83] SonarSource, “Get Started in Two Minutes Guide | SonarQube Docs.” [Online]. Available: https://docs.sonarqube.org/latest/setup/get-started-2-minutes/. [Accessed: 10-May-2020].

[84] SonarSource, “SonarScanner for Gradle | SonarQube Docs.” [Online]. Available:

https://docs.sonarqube.org/latest/analysis/scan/sonarscanner-for-gradle/. [Accessed: 12-Apr-2020].

[85] Android Developers, “Benchmark app code (Configuration Errors) | Android Developers.” [Online]. Available: https://developer.android.com/studio/profile/benchmark#configuration-errors. [Accessed: 14-Apr-2020].

[86] Square, “OkHttp.” [Online]. Available: https://square.github.io/okhttp/. [Accessed: 12-Apr-2020].

[87] Square, “Interceptors - OkHttp.” [Online]. Available: https://square.github.io/okhttp/interceptors/. [Accessed: 12-Apr-2020].

[88] Android Developers, “Geocoder - Android Developers,” 2013. [Online]. Available: https://developer.android.com/reference/android/location/Geocoder. [Accessed: 26-Apr-2020].

[89] “Timed out waiting for process [150330359] - Visible to Public - Issue Tracker.” [Online]. Available: https://issuetracker.google.com/issues/150330359. [Accessed: 26-Apr-2020].

[90] Oracle Corporation, “System (Java Platform SE 7),” 2014. [Online]. Available: https://docs.oracle.com/javase/7/docs/api/java/lang/System.html. [Accessed: 26-Apr-2020].

[91] Oracle Inc, “Runtime (Java Platform SE 7),” *Java 7 documentation*, 2013. [Online]. Available: https://docs.oracle.com/javase/7/docs/api/java/lang/Runtime.html. [Accessed: 26-Apr-2020].

[92] J. Nielsen, “Response Time Limits: Article by Jakob Nielsen,” *Usability Engineering*, 1993. [Online]. Available: https://www.nngroup.com/articles/response-times-3-important-limits/. [Accessed: 06-Sep-2020].

[93] R. B. Miller, “Response time in man-computer conversational transactions INTRODUCTION AND MAJOR CONCEPTS.”

[94] A. G. Project, “The Apache Groovy programming language.” [Online]. Available: https://groovy-lang.org/. [Accessed: 26-Apr-2020].

[95] Github Community, “Arkni/json-to-csv: JSON to CSV converter.” [Online]. Available: https://github.com/Arkni/json-to-csv. [Accessed: 26-Apr-2020].

Appendix A. **JSON to Excel Adapter**

Although JSON is a very acceptable format for the benchmark report, Excel for macOS has no easy way of extracting data from a JSON file at the time of writing this document. One solution we found was to convert this JSON data into CSV (Comma Separated Values), a more standard way to represent data that is intended to be written in a table format.

To achieve this, the benchmark module relies on a Gradle task *benchmarkJsonToCsv* that will grab the JSON file generated by the *connectedCheck* task, convert its "benchmarks" array to CSV and write a CSV file under a folder called "results", inside the benchmark's module root directory. This task is written in Groovy [94], and it uses json-to-csv [95], a library that aided with both the conversion from JSON to CSV and with the writing of the CSV file.

A second Gradle task was created that integrates *connectedCheck* with *benchmarkJsonToCsv,* this task is called *benchmarkCsvReport.*

As the CSV is automatically generated after the benchmarks run, this allows for some automation on the Excel side as well. Creating the table with all the benchmark results was not a problem since Excel supports CSV as a data source and even allows a table to be connected to a source, meaning every time the CSV changes, Excel can refresh the data within the table with a simple click. This is advantageous since manually creating tables every time there is new data is not sustainable and defeats the wanted purpose of automation.

The Excel has 4 sheets, one for the source table (the one that is connected directly to the CSV file) and one for each test type (network, database, and integration). The sheets corresponding to each test type contain a pivot table each that filters out the source table in order to have only the test results related to that test type, this filtering is done through the test class name, that is common to tests from the same type. This means every time a developer adds a new test to any of the test classes, they will appear under the pivot table for that specific test class.

Pivot tables can calculate new fields based on the ones from the source table, in this case, they calculate the average execution time for the tests, in milliseconds.

Excel also allows a user to create charts out of pivot tables, this is how the charts for each test type are being generated. Since these pivot tables also refresh whenever the source table has new results, every time the a developer runs the tests and has a new CSV file with new data, it can simply refresh the source table with the new values and Excel will automatically generate the pivot tables alongside charts that will reflect those new values.

The excel file can be found alongside the case study implementation in the same GitHub repository [73].