How does Migrating to Kotlin Impact the Run-time Efficiency of Android Apps?

Michael Peters
Vrije Universiteit Amsterdam
Amsterdam, The Netherlands
m.peters0811@gmail.com

Gian Luca Scoccia
DISIM, University of L'Aquila
L'Aquila, Italy
gianluca.scoccia@univaq.it

Ivano Malavolta
Vrije Universiteit Amsterdam
Amsterdam, The Netherlands
i.malavolta@vu.nl

***Abstract*—*Context*. Android developers that developed Android apps using Java 6 for a long time got introduced to Kotlin as a new programming language in 2017. Kotlin contains many features that make it a popular alternative to Java in Android development, and together with the full support of Google and its creator, Jetbrains, it is becoming an essential part of Android development. *Goal*. This study aims to empirically assess the impact of the migration from Java to Kotlin on the run-time efficiency of Android apps. *Methodology*. To achieve this goal, we mine 7,972 GitHub repositories of Android apps and identified 451 apps containing Kotlin code. Then, by applying a cross-language clone detection technique, we detect 62 commits that represent a full migration to Kotlin, while keeping the app functionally equivalent. We sample 10 apps that fully migrated to Kotlin and conducted a measurement-based experiment to compare their Java and Kotlin versions with respect to seven run-time efficiency metrics. *Results*. Our study shows that migrating to Kotlin has a statistically significant impact on CPU usage, memory usage, and render duration of frames (though with a negligible effect size), whereas it does not impact significantly the number of calls to the garbage collector, the number of delayed frames, app size, and energy consumption. *Conclusions*. This study provides evidence that developers can migrate their Android apps to Kotlin and expect comparable efficiency at runtime. As a side product, this study also confirms that most open-source Android apps either fully migrated to Kotlin (>90% Kotlin code) or contain low portions of Kotlin code (<10%).**

***Index Terms*—Empirical study; Android; Kotlin; Performance; Energy consumption**

I. Introduction

The Android operating system is the current leader in the mobile operating systems market (74% market share at the end of 2019 [1]), while also supporting a variety of different platforms such as television systems, smartwatches, multimedia car systems, and IoT devices [2]. Originally restricted to Java 6, since 2017 developers can adopt Kotlin, a modern statically-typed programming language [3], to program their applications. Kotlin introduces several features not available in Java 6 (e.g., null safety, lambda expressions) and is currently considered by Google as the main language for Android application development [4].

Given the above, developers might be interested in performing a *migration to Kotlin*, i.e., rewrite parts or the entirety of their Java app in Kotlin, in order to take advantage of the new features. However, while the level of Kotlin adoption [5], [6] and perceived benefits [7], [8] have been investigated by researchers, to date, there is no evidence on the impact that a Kotlin migration has on the application run-time efficiency.

We fill this research gap by conducting an **empirical study on the impact of the migration from Java to Kotlin on the run-time efficiency of Android apps**. In order to achieve this goal, we mine 7,972 GitHub repositories of open-source Android apps and identified among them 451 apps containing Kotlin code. Then, by applying a cross-language clone detection technique, we detect 62 commits that are responsible for a full migration to Kotlin, while keeping the app functionally equivalent. We conducted a measurement-based experiment on a sample of ten apps that fully migrated to Kotlin, to compare their Java and Kotlin versions with respect to seven run-time efficiency metrics. In addition, we provide **up-to-date statistics on the level of adoption of Kotlin in open-source Android applications distributed in the Google Play store**.

The results of our experiment highlight that migrating to Kotlin has a statistically significant impact on CPU usage, memory usage, and render duration of frames (albeit with a negligible effect size), whereas it does not impact significantly the number of calls to the garbage collector, the number of delayed frames, app size, and energy consumption.

The main contributions of this paper are:

- A quantitative analysis of the *level of adoption of Kotlin* over the lifetime of open-source Android projects.
- An empirical assessment of the Java and Kotlin versions of 10 real-world Android apps according to seven metrics related to *performance*, *app size*, and *energy efficiency*.
- A *replication package* with all raw data and scripts to replicate the experiments[1].

The study aims to support Android developers, maintainers of the Android platform and its Kotlin runtime, and researchers. The former are provided with evidence on the level of Kotlin adoption in Android development and on the run-time impact of Kotlin, which forms an objective basis for deciding about the adoption of the Kotlin language. Maintainers are given evidence on the potential run-time efficiency impact occurring after migrating to Kotlin, which can be used as a basis to further investigate root causes, to improve the Android platform and the Kotlin language itself. We inform fellow

[1]https://zenodo.org/record/5166703

researchers on the state of Kotlin usage, Kotlin migration activities, and the run-time efficiency impact of a Kotlin migration; these results can be used for further research into migration activities and run-time efficiency impact.

To allow independent verification and replication of the performed study, we make publicly available a full replication package[1], containing: (i) the Python scripts to perform all mining and data extraction steps; (ii) intermediate results; (iii) data visualizations; (iv) the Python scripts for performing the statistical analysis.

II. Background

This section provides context and discusses preliminary concepts required in subsequent sections. We provide a brief description of the Kotlin language, its usage in Android development, and we define the meaning of performing a "*migration to Kotlin*".

A. Kotlin in Android development

Kotlin is a cross-platform, statically typed, general-purpose programming language with type inference. Kotlin is designed to interoperate fully with Java and introduces several features missing in the latter such as, e.g., null safety, data classes, extension functions, lambda expressions [3]. Kotlin was originally introduced by JetBrains that, together with Google, created the Kotlin foundation to promote and advance the development of the Kotlin programming language [9]. On the 7th May 2019, Google announced that the Kotlin programming language is now its preferred language for Android app developers [4], and new projects should be developed with it. After the introduction of Kotlin as a first-class citizen by Google, Kotlin became a fully supported alternative to the previous standard language Java 6. By introducing Kotlin, Google follows the footsteps of Apple that introduced Swift in 2014 [10] as an alternative programming language for developing iOS apps. Both languages share that they replaced their more verbose predecessor, Java 6 for Android and Objective-C for iOS, with a more modern and less verbose language that is interoperable with the old language.

For developing Android applications in Kotlin, Jetbrains, and Google offer various tools. Since the release of Android Studio 3.0, Kotlin gained full IDE support, introducing features already available for Java such as, e.g., code completion, code inspection, debugging, refactoring. Additionally, a feature to convert complete Java classes to Kotlin is also made available to support developers with migrating their applications to Kotlin.

B. Kotlin Migration

A migration towards Kotlin can take many shapes and forms thanks to the many interoperability features Kotlin contains (e.g., Java types mapped to Kotlin types or annotations that help the compiler translate a concept found solely to the other). Because of these features, developers do not have to rewrite their entire codebase and can migrate using various approaches. In order to still recognize migrations across these many variations, we define a Kotlin migration as Java code being replaced with Kotlin code that is logically equivalent. This definition applies to extensive replacements such as multiple files, and as well for small replacements such as single methods.

III. Study design

This section describes all the methods used in this study. It starts with a detailed description of our research questions and continues with our dataset creation process. Finally, we describe the design of our experiment and the methods used for analyzing and interpreting the results for each research question sequentially.

A. Goal and research questions

Our main **goal** is to assess the impact of migration from Java to Kotlin on the performance and energy efficiency of Android apps. As a preliminary step, in order to understand the context of the migration act itself, we also investigate how much Kotlin is currently used in Android applications. To the best of our knowledge, no empirical studies exist that evaluate the performance or energy efficiency impact of migrations to Kotlin in real-world open-source Android applications. For structuring our research, we split our goal up into two separate research questions which are independent of each other:

RQ1: What is the level of usage of Kotlin in open-source Android apps?

RQ2: How does a migration to Kotlin impact the run-time efficiency of Android apps?

By answering RQ1, we study how largely adopted Kotlin is within Android open-source apps and inform both Android developers and fellow researchers on the adoption level of Kotlin in Android development. Results will serve as an essential gauge for Android developers to use for deciding on the adoption of Kotlin (e.g., low usage levels might make it harder to find Android developers skilled in Kotlin). To answer this question, we will replicate part of the research done by Mateus [5] using the AndroidTimeMachine open-source Android apps dataset created by Geiger et al. [11]. Our replication will verify the results of Mateus et al. using a different dataset containing solely real-world applications and will serve as a foundation for answering our second research questions.

Answering RQ2 leads to results on whether a statistically significant difference exists in the run-time efficiency of apps that migrated to Kotlin. Results from this research question will be useful for Android developers that are debating on migrating to Kotlin. RQ2 is answered by designing and conducting an empirical assessment of the run-time efficiency of a set of Android apps that have been fully migrated to Kotlin. Specifically, we consider fully migrated apps from our dataset and for each app we consider its Java and Kotlin versions.

B. Data collection and extraction

To answer our research questions, a dataset that meets the two following criteria is necessary: (i) it must have a

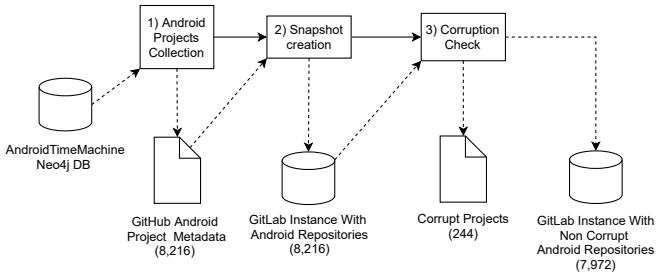


Fig. 1: Dataset collection process

clear distinction between regular Android applications and applications containing Kotlin; (ii) source code and project history must be fully accessible for all entries. Hence, to build a valid dataset, we adopt the process summarized below and shown in Figure 1.

Applications collection - As a starting point for the collection of our dataset, we use *AndroidTimeMachine*, an independently-built dataset of open-source Android applications [11]. We choose this dataset as our starting point as (i) it provides a large number of open-source applications, thus increasing the likelihood of having a representative sample of applications even after subsequent filtering steps, and (ii) it includes pointers to the GitHub repository of each entry, thus meeting the requirement of having full access to the source code and project history. Although AndroidTimeMachine is made available in ready-to-use components, it is not usable in an out-of-the-box fashion for the purpose of this study as project histories included in it have been collected in 2018, thus using them would have meant excluding more than a year's worth of application versions. Therefore, we create an updated snapshot of source code and project histories for applications included in AndroidTimeMachine.

Listing 1: Kotlin projects identification procedure

```
1  function filterRepositoriesOnKotlin(repos: List){
2     kotlinRepos = set();
3
4     for repository in repos {
5        for commit in repository.commits {
6           for file in commit {
7              if filePath.endswith(".kt"){
8                 kotlinRepos.add(repository);
9              }
10          }
11       }
12    }
13    return kotlinRepos;
14 }
```

The snapshot creation starts by extracting from AndroidTimeMachine the list of all entries with an attached GitHub repository. This query results in 8,216 GitHub repositories. We continue by importing these repositories into a GitLab Docker image[2] instance. For 244 repositories, insertion resulted in a corrupt repository containing zero commits. After inspection, we found that the GitHub repository no longer exists for 243 of these projects while for the sole project that was leftover we found that the repository was empty. These corruptions were most likely introduced in the time between our study and the AndroidTimeMachine study. Thus, our dataset of Android applications consists of a snapshot of 7,972 git repositories, hosted in a local GitLab instance for easy access.

Identifying Kotlin Applications - For finalizing the dataset, identification of projects that contain Kotlin code is required. This process is achieved by following the algorithm described in Listing 1. First, we clone each repository in our local GitLab instance (line 5 in Listing 1). Cloning the repository will automatically check out the main branch from the original GitHub repository. The process is then continued by listing all commits in the main branch by using the git `log` command (line 5). All commits are then iterated on, and files changed in each are checked for the presence of a Kotlin file extension (lines 6-7). If a Kotlin file extension is found, we tag the Android repository as a Kotlin application (line 8). The full process results in a filtered dataset of 451 applications that ever contained any Kotlin code, and thus classified as Kotlin applications. Our filtering process, does not check for the presence of Kotlin code in applications' dependencies, due to the increased complexity it introduces and due to the fact that choice of libraries is not always fully controllable by Android developers themselves.

Listing 2: Kotlin projects identification procedure

```
1  function countSlocForRepositories(repos: List){
2     results = map();
3
4     for repository in repos {
5        repository = repository.cloneRepository();
6        for commit in repository.commits {
7           commit = commit.checkout();
8           sloc = Cloc.countSloc(commit);
9           results.put(repository, sloc.java, sloc.kotlin)
                ;
10       }
11    }
12    return results;
13 }
```

Measuring Java and Kotlin SLOC - Having access to the entire git history for each application in our dataset enables us to measure the lines of code for each language directly on the source code. We do so by using the tool *CLOC*[3]. CLOC is a tool that counts the source lines of code of an application, grouped for each of the multiple programming languages it recognizes. It is able to detect blank lines and comments so, in our study, we only measure lines of actual code. The process of counting the SLOC is described in Listing 2. It starts with cloning the repository (line 5 in Listing 2) and iterating on every commit (lines 6). For each, CLOC is run on the checked-out source code (lines 7-9). It results in the count of Kotlin and Java SLOC for each version of the application for all 451 Kotlin applications in our dataset.

C. Data analysis

In the following, we describe the steps undertaken to analyze the collected data towards answering our research questions.

[2]https://docs.gitlab.com/omnibus/docker/

[3]https://github.com/AlDanial/cloc

Measuring the degree of Kotlin adoption - For answering RQ1, we categorize the 7,972 Android applications in our dataset using the three categories defined by Mateus et al. [5]: (i) *Entirely written in Java*, (ii) *Entirely written in Kotlin*, (iii) *Written in both Java and Kotlin*. Every application that is part of our Android applications dataset but not part of the Kotlin applications are assigned to the first category. Kotlin applications that never contained any lines of Java code are assigned to the second category. All remaining Kotlin applications are assigned to the third category. We present these results visually in the bar chart of Figure 3.

Measuring the proportion of Kotlin code - Furthermore, to provide a more in-depth answer for RQ1, we take the counts of SLOC for the most recent version of every Kotlin application and calculate the proportion of Kotlin code compared to Java code, disregarding any other programming language present in the codebase. We plot these in the histogram visualization of Figure 4, which shows the Kotlin proportion distribution for Kotlin applications in our dataset.

Measuring the run-time efficiency impact of a Kotlin migration – We answer RQ2 quantitatively. In the following, we describe the experiment design by first covering the selection of the subjects and then defining the independent and dependent variables, together with how the latter are measured. We continue with a formulation of hypotheses that ultimately answer RQ2 and the design of our experimental setup. Finally, we describe the statistical methods for analyzing the data and accepting or rejecting the hypotheses.

Subjects selection – As subjects for the experiment, we need a sample of applications for which both a version entirely written Java and a version entirely written in Kotlin exists. An important aspect to consider in the selection of these applications is that the transition from Java to Kotlin must be a *pure migration*. A migration from Java to Kotlin is pure if it does not introduce any new functionality in the app. By considering pure migrations, we are reasonably confident about the functional equivalence of the Java and Kotlin versions of an app. We start our sampling by identifying in our dataset those projects for which the Kotlin code replaces all Java code between two consecutive commits in the project history. To do so, first, we identify all logically equivalent code chunks in Kotlin and Java for all projects in our dataset, by applying the cross-language clone (CLC) detection method proposed by Cheng et al. [12]. Employing this methodology, all commits in revision histories are analyzed to identify Kotlin migration commits. Whenever a Java deletion and a Kotlin addition resulting in a CLC is detected, the commit is classified as a Kotlin migration, since it meets the definition of containing deleted Java code that is replaced by logically equivalent Kotlin code. This led to the identification of 3,674 Kotlin migration commits. Among these, we found 62 projects for which Kotlin code replaces all Java code between two consecutive versions.

Adopting these projects as the base for our subject selection increases the likelihood that the migration is a pure migration, as it was completely performed in a single commit. From the initial 62 projects, we pick a sample of 10 applications using stratified random sampling [13]. We use two characteristics for the stratified random sampling: (i) the application's category as listed in the Google Play store, (ii) the total Kotlin and Java SLOC being either lower than a threshold t or greater or equal to it. We stratify by app category in order to have a balanced set of apps with respect to their provided functionalities. We chose $t = 5000$ SLOC since we observed that apps with lower SLOCs tend to be either single-purpose or extremely basic. This sampling procedure increases the likelihood that a varied set of categories, as well as both small and large projects, are well represented in our sample. To be certain that the sample consists of only pure migrations, we manually inspect the sampled migration commit via three heuristics: (i) verify that the paths of the deleted Java files and added Kotlin files are matching; (ii) verify that the deleted Java code and the added Kotlin code are functionally similar; (iii) we install both the Java and Kotlin versions of the app and manually check that they provide exactly the same functionalities, i.e., all buttons and screens are functionally equal in the two versions. If the migration passes all of these manual checks, we consider it pure. Our initial sample of 10 migrations successfully passed all three heuristics, and therefore, we did not have to introduce measures to deal with impure migrations. The 10 applications are presented in Table I.

TABLE I: Experiment subjects

Id	Name	Category	Java SLOC	Kotlin SLOC
a_1	Fortune-Android	Entertainment	580	542
a_2	whitakers-words-android	Books & Reference	433	414
a_3	slounik	Books & Reference	1,443	1,380
a_4	Glyph	Trivia	1,891	2,012
a_5	TaskGame	Adventure	7,348	6,053
a_6	SimpleHTMLTesterAndroid	Productivity	393	372
a_7	drag-select-recyclerview	Libraries & Demo	624	505
a_8	DFReminder	Tools	542	421
a_9	R.tools	Tools	793	728
a_{10}	home-button	Tools	203	168

Independent and dependent variables – As an independent variable, we use the programming language used in the application. It has two treatments: a 100% usage of Java before migration to Kotlin and a 100% usage of Kotlin after the migration to Kotlin.

The dependent variables of this experiment consist of well-known metrics for both performance and energy efficiency of Android apps:

- CPU usage (*cpu*): optimizing CPU usage provides a faster and smoother experience to the user while also preserving battery life [14]. We define CPU usage as the percentage of the device's total CPU capacity used by an application at given points in time during its lifetime. It is measured using the Android Debugging Bridge (ADB) `dumpsys cpuinfo` command throughout the entire duration of the experiment at a sampling frequency of one second.
- Memory usage (*mem*): physical memory is constrained on mobile devices due to clear limitations in space, and therefore, memory is a valuable resource in Android.

Excessive memory consumption can degrade app performance and can cause application crashes [15]. We define memory usage as the amount of RAM in KB used by an application at given points in time during its lifetime. It is measured using the `dumpsys meminfo` ADB command throughout the entire duration of the experiment at a sampling frequency of one second.

- Number of calls to the garbage collector (*gc*): The system's memory is freed up automatically by the garbage collector. Poor memory management, such as the introduction of memory leaks, not only causes more garbage collector calls but also intensifies the work done by each call, thus degrading performances [16]. The number of calls is counted by reading device logs through the ADB `logcat` utility. Additionally, we checked the source code of every application on potential explicit invocations of the garbage collector. None of the 10 apps were making explicit calls to the garbage collector.
- Frame times (*ns*) and the number of delayed frames (df): When Android renders a frame, it takes a certain amount of time to do so. This frame time is, therefore, an essential factor in perceived performances when using an Android application. The ideal frame rate is 60 frames per second (FPS). To achieve this rate, frames must be rendered in under 16ms; otherwise, the system is forced to skip frames. As the human eye is very keen on noticing drops in FPS, the user will perceive such events as stuttering in the app [15]. Therefore, the amount of such delayed frames directly affects the user's experienced performance as well. We measure these metrics by running the `dumpsys gfxinfo framestats` ADB command throughout the entire duration of the experiment. We count frames that took more than 16ms to be rendered as delayed frames.
- App size (*as*): App size is defined as the size of the application when packaged into an APK. App size can influence how users perceive an app since devices have limited storage available, and in some circumstances, costs may be involved when downloading large files from the Internet. It is measured by taking the size of the APK binary file of each app in bytes.
- Energy consumption (*en*): energy consumption stands for the number of Joules consumed by the device in a period of time. Low power consumption is a critical non-functional requirement when building an Android app, as mobile devices have limited battery and, when neglected, it seriously impacts the users' perceived app quality [17]. In our experimentation, energy consumption is represented in Joules and it is measured by means of a software-based technique based on the ADB `batterystats` tool. In the literature, the accuracy of software based approaches has been reported to be reasonably close to hardware-based ones [6], [18].

Hypotheses – To answer RQ2, we formulate a null hypothesis for each dependent variable, specifically:

- *cpu*: being μ^{cpu} the mean CPU usage per run for a given application version, we define the null and alternative hypotheses as:

$$H_0^{cpu} : \mu_{java}^{cpu} = \mu_{kotlin}^{cpu} \quad H_1^{cpu} : \mu_{java}^{cpu} \neq \mu_{kotlin}^{cpu}$$

- *mem*: being μ^{mem} the mean of memory usage per run for a given application version, we define the null and alternative hypotheses as:

$$H_0^{mem} : \mu_{java}^{mem} = \mu_{kotlin}^{mem} \quad H_1^{mem} : \mu_{java}^{mem} \neq \mu_{kotlin}^{mem}$$

- *gc*: being μ_v^{gc} the mean of the number of GC calls per run for a given application version, we define the null and alternative hypotheses as:

$$H_0^{gc} : \mu_{java}^{gc} = \mu_{kotlin}^{gc} \quad H_1^{gc} : \mu_{java}^{gc} \neq \mu_{kotlin}^{gc}$$

- *ft*: being μ_v^{ft} the mean of frame time values per run for a given application version, we define the null and alternative hypotheses as:

$$H_0^{ft} : \mu_{java}^{ft} = \mu_{kotlin}^{ft} \quad H_1^{ft} : \mu_{java}^{ft} \neq \mu_{kotlin}^{ft}$$

- *df*: being μ_v^{df} the mean of the number of delayed frames per run for a given application version, we define the null and alternative hypotheses as:

$$H_0^{df} : \mu_{java}^{df} = \mu_{kotlin}^{df} \quad H_1^{df} : \mu_{java}^{df} \neq \mu_{kotlin}^{df}$$

- *as*: being μ_v^{as} the size of a given application version, we define the null and alternative hypotheses as:

$$H_0^{as} : \mu_{java}^{as} = \mu_{kotlin}^{as} \quad H_1^{as} : \mu_{java}^{as} \neq \mu_{kotlin}^{as}$$

- *en*: being μ_v^{ec} the mean energy consumption per run for a given application version, we define the null and alternative hypotheses as:

$$H_0^{en} : \mu_{java}^{ec} = \mu_{kotlin}^{ec} \quad H_1^{en} : \mu_{java}^{ec} \neq \mu_{kotlin}^{ec}$$

Experiment design and execution – We automate the execution of the tests for each subject employing an ad-hoc script that automatically clicks through the application and covers all of its features with the necessary waiting operations in-between steps. The scripts are available in the online replication package[1]. Each script is designed to execute one user scenario for each of the features that the app contains (e.g., editing settings, visiting screens, scrolling through content, and creating entries). We create these scenarios based on a manual exploration of the experiment subjects, during which we explored the running app, its Google Play Store page, and its source code. While running the scripts, all metrics are measured in parallel. Before and after the experiment, a thorough setup and reset phase is included. The scripts and the code for the measurements are implemented using Android Runner, an open-source Python framework for automating experiments on Android devices [6].

We perform our experiment on two different devices. The two devices consist of an older generation device (Google Nexus 5) and a more recent device (Google Pixel 3a). We

consider the type of Android device as a blocking factor since the two devices have different hardware specifications.

Before running the experiment, we manually prepare our factory reset device by fully charging the battery, removing the SIM and SD card, setting brightness and sound to a minimum, enabling the stay awake developer option, and finally, disabling network data, WiFi, Bluetooth, and notifications of other apps. Continuing, before starting a single test run, we implemented a setup step that clears the device log files, and performs a fresh install of the app. Finally, after every single test run, we include a timeout step that waits 2 minutes in order to prevent tail energy consumption from influencing our measures.

Figure 2 gives an overview of the experiment execution process. Android Runner hosts the plugins that measure our defined metrics as well as the scripts that execute the experiment. Our created test scripts that interact with the subject application are ran using MonkeyRunner[4], a utility for automated testing of Android apps. In parallel, all plugins are running and collecting the necessary data for each metric. For each device, the experiment is run 20 times per subject in randomized order.

Analysis of experimental data – We start our data analysis by verifying our assumption that the data gathered for each hypothesis is not normally distributed. It consists of visual analysis of each metric's Q-Q plots [19] and the application of the Shapiro-Wilk test [20]. We make use of the Shapiro-Wilks test because it is proven to be one of the most powerful tests for testing normal distribution [21]. The result of this verification indicates which statistical test we can use for testing our hypotheses.

For testing our hypotheses, we need to prove that a difference between the Java and Kotlin versions of the same app exists. In order to find out whether the two independent distributions measured for each hypothesis are different, we perform statistical tests. Since we obtained evidence that verifies our assumption on the non-normal distribution of our data (further elaborated upon in the results Section IV), we make use of the non-parametric Mann-Whitney U test [22]. It assesses whether the Kotlin and Java measurements come from a different distribution and thus rejects the null hypothesis. For interpreting the resulting p-values, we use the Benjamini Hochberg [23] $p-$value correction procedure. It reduces the chance of Type-I errors that may occur due to the application of multiple statistical tests.

For assessing the magnitude of potential differences found for each of our hypotheses, we calculate the Cliff's δ effect size. Cliff's δ effect size is a non-parametric test that requires no assumptions on the data's distribution. It quantifies whether the results of the treatments are either larger or smaller than the other. We interpret the effect sizes as follows: large $\delta \geq 0.474$, medium $\delta \geq 0.33$ and small $\delta \geq 0.147$ [24].

[4]https://developer.android.com/studio/test/monkeyrunner

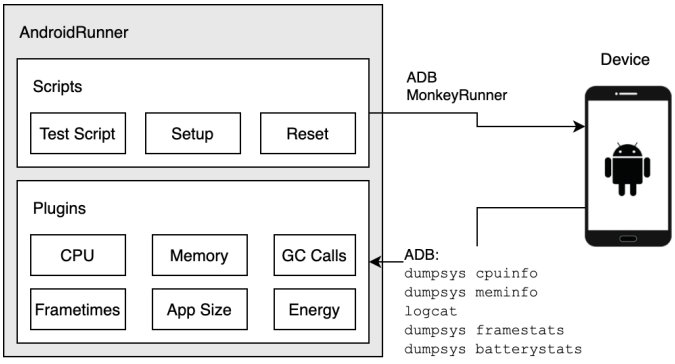


Fig. 2: Experiment execution overview

IV. Results

A. What is the level of usage of Kotlin in open-source Android apps? (RQ1)

Figure 3 summarizes our results on the frequency of Java and Kotlin code in the 7,972 apps in our dataset, using the three categories defined in Section III-C. It can be observed that in our dataset a total of 7,521 apps are entirely written in Java, while 432 apps are written in both Java and Kotlin, and only 19 applications are entirely written in Kotlin.

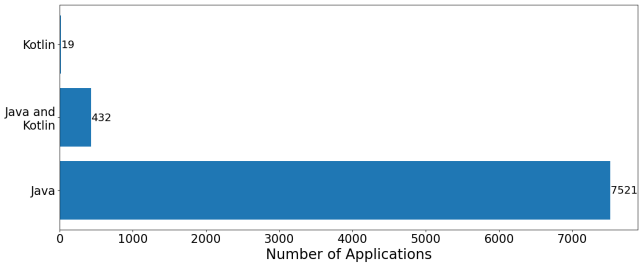


Fig. 3: Frequency of Kotlin and Java for the analyzed apps

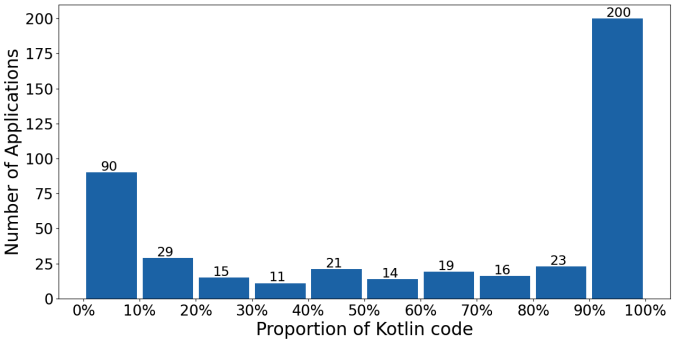


Fig. 4: Proportion of Kotlin code in analyzed apps

Results on the observed proportion of Kotlin code are plotted in Figure 4. It shows the amount of Kotlin applications per proportion bin (i.e., ranges of the percentage of Kotlin code). As seen, 200 applications have a proportion of Kotlin code that is higher than 90% and they represent 44.34% of our datasets Kotlin applications. On the other side, we see 90

applications that have a proportion of Kotlin code that is lower than 10%. The remaining 148 applications have a variable proportion of Kotlin code, ranging from less than 20% to up to 80%. Combined, 64.30% of the Kotlin applications have either less than 10% Kotlin code or more than 90% Kotlin code in their latest version.

B. How does a migration to Kotlin impact the run-time efficiency of Android apps? (RQ2)

In this section, we present the results of the experiments we conducted to answer RQ2. We explore the data, verify our data distribution assumption, continue with hypotheses testing, and finally assess the effect sizes.

Median values for all collected metrics are displayed in Table II. For reasons of space only median values for the Nexus 5 device are included in the table, although values for the Pixel 3a are available in the online replication package. In Figure 5, the results for all metrics *per device* are visualized via boxplots. We can observe that the results per device differ in actual values. This is expected due to the entirely different hardware specifications of the two devices, We will not investigate the impact of differences between devices any further since we treat the device used as a blocking factor. However, the Java and Kotlin results per device are very similar. We can observe that for both devices, the median of all metrics, except for app size, is roughly the same. Collected metrics on the Nexus 5 show more similarities in the distribution of values for the Pixel 3a. The app size is clearly showing an increase, even though the same APK was installed on both devices.

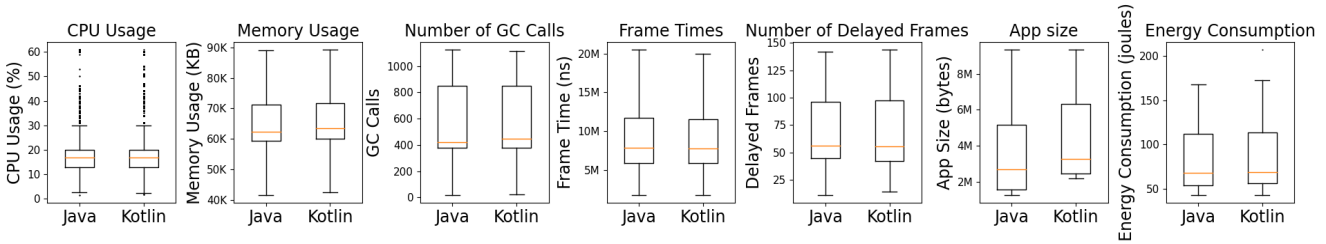
To exemplify our non-normal distribution assumption, we present the Q-Q plots for the collected measures in Figure 6; From inspecting the Q-Q plots, we are already reasonably confident that our assumption of non-normality will hold. Nonetheless, we present the Shapiro-Wilks test results in

TABLE II: Median values of collected metrics on the Nexus 5 device for each experiment subject

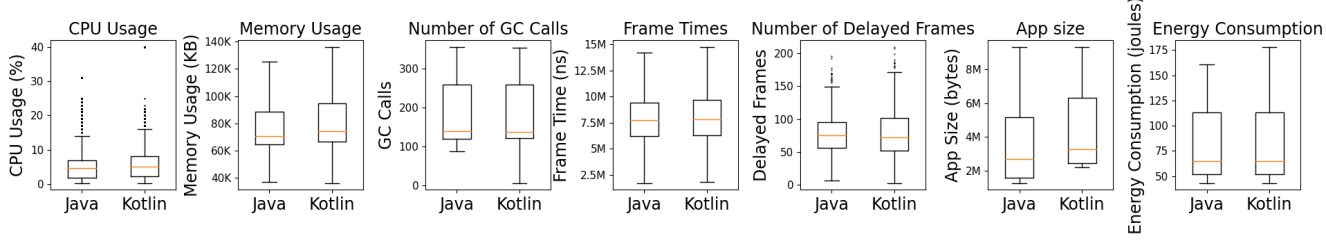
Id	Version	*cpu (%)*	*mem (kb)*	*gc (#)*	*ft (#)*	*df (#)*	*as (kb)*	*en (J)*
a_1	Java	15	66,048	517	6,665,167	44.5	9,344,435	67.38
	Kotlin	16	66,326	522.5	6,665,900	45.5	9,350,166	68.86
a_2	Java	11	59,255	314.5	14,791,386	52.5	6,219,999	44.14
	Kotlin	11	59,866	317	15,124,297	52.5	6,966,677	45.30
a_3	Java	11	60,198	372.5	6,298,024	27.5	1,535,875	51.93
	Kotlin	13	61,335.5	372	6,287,044	29	2,330,123	53.49
a_4	Java	21	61,264	1,097	10,708,849	413	3,298,964	164.48
	Kotlin	20	63,542	1,092	10,680,827	435	5,221,501	169.09
a_5	Java	19	81,854	852	7,009,759	95.5	5,675,809	117.91
	Kotlin	19	82,929	847	7,072,927.5	97.5	6,669,816	121.49
a_6	Java	17	93,277	409.5	7,641,309	124	1,472,271	81.92
	Kotlin	18	95,769.5	405	7,520,508	113	2,876,077	83.55
a_7	Java	17	59,552	1,014	5,890,543	75.5	1,700,408	113.5
	Kotlin	18	60,173.5	1,014	5,856,785	77	2,222,051	115.28
a_8	Java	13	64,157	372	7,422,840	43	1,257,911	53.80
	Kotlin	12	65,487	377	7,491,825	43	2,207,478	56.75
a_9	Java	14	60,948	412	6,919,922	68.5	2,105,176	54.89
	Kotlin	12	61,307	412.5	6,811,116.5	68.5	2,846,987	56.69
a_{10}	Java	16	93,163	472	7,083,347	50	3,671,644	68.66
	Kotlin	16	94,890.5	467.5	7,230,705	33	3,649,437	68.27

Table III for making the final conclusion on whether the assumption holds. From it, we can see that only the energy consumption metric barely accepts the normal distribution hypothesis through the Shapiro-Wilks test. However, since it barely does so and the Q-Q plot instead do exhibit a non-normal distribution of values, we proceed with the previously defined non-parametric statistical tests.

Continuing with our hypotheses testing, we present the Mann-Whitney U test p-value results for all of our metrics (previously introduced in Section III-C) in Table IV. Since we utilize a p-value correction procedure, interpretation of whether a p-value allows us to reject the null hypothesis is not straightforward. Hence, in Table IV we mark in bold the test for which we are able to reject the null hypothesis after we applied the Benjamini-Hochberg p-value correction procedure. It can be observed that H^{cpu}, H^{mem}, and H^{ft} are rejected while the remaining hypotheses are not rejected.

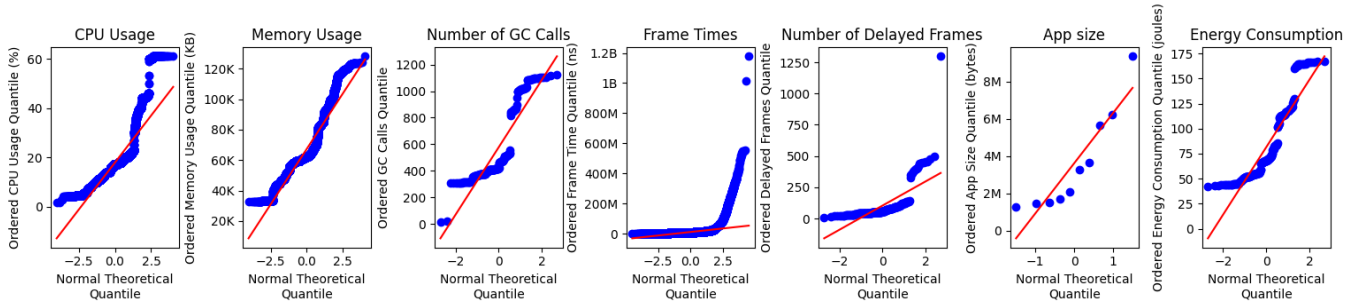


(A) Nexus 5

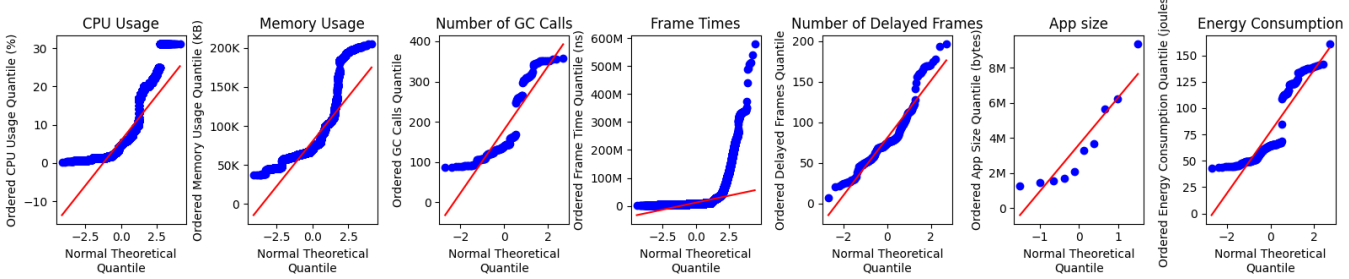


(B) Pixel 3a

Fig. 5: Boxplot visualisations for the Nexus 5 and Pixel 3a results



(A) Nexus 5



(B) Pixel 3a

Fig. 6: Q-Q plots of the measures collected from the Nexus 5 and Pixel 3a

TABLE III: Shapiro-Wilks test p-values (bold if $p \geq 0.05$)

	Nexus 5		Pixel 3a	
	Java	Kotlin	Java	Kotlin
cpu	0	0	0	0
mem	0	0	0	0
gc	$1.92e^{-23}$	$5.57e^{-22}$	$1.68e^{-9}$	$1.41e^{-8}$
ft	0	0	0	0
df	$4.64e^{-15}$	$7.63e^{-15}$	$5.40e^{-15}$	$4.87e^{-14}$
as	$1.52e^{-13}$	$9.62e^{-14}$	$3.31e^{-15}$	$4.48e^{-15}$
en	**0.52**	**1.02**	**0.52**	**1.02**

In other words, the distribution of values differs significantly between the Java and Kotlin treatments of the apps only for CPU usage, memory usage, and frame times.

TABLE IV: Mann-Whitney U test p-value results (bold if $p \leq 0.05$ after the Benjamini-Hochberg correction procedure) and Cliff's δ effect size results (N = Negligible, S = Small, M = Medium, and L = Large)

	Nexus 5	Pixel 3a	δ (Nexus 5)	δ (Pixel 3a)
cpu	$\mathbf{7.2e^{-32}}$	$\mathbf{5.96e^{-23}}$	-0.062 (N)	-0.086 (N)
mem	$\mathbf{1.83e^{-57}}$	$\mathbf{4.37e^{-233}}$	-0.085 (N)	-0.142 (N)
gc	0.836	0.842	-	-
ft	$\mathbf{4.29e^{-9}}$	$\mathbf{1.61e^{-29}}$	0.011 (N)	-0.024 (N)
df	0.486	0.41	-	-
as	0.241	0.241	-	-
en	0.233	0.978	-	-

We assess the magnitude of the difference in results between the Java and Kotlin treatment using Cliff's δ effect size measure. The last two columns of Table IV present the effect size for all metrics for which we are able to reject the Mann-Whitney U test null hypothesis together with an interpretation according to [24]. These results show that the difference between the two treatments is negligible for all metrics.

V. Discussions

In this section we discuss the obtained results and contextualize them with respect to the current state of the art.

Our methods for answering RQ1 partially replicate the methods used by Mateus et al. [5]. Therefore, it is interesting to compare the results and find out whether there are differences and the reasons behind them. Starting with the results on the adoption of Kotlin, we found that 451 (5.66%) applications in our dataset contain Kotlin, while 7,521 (94.34%) do not contain any Kotlin code. Similarly, Mateus et al. found that 244 (11.26%) applications contain Kotlin, and 1,923 (88.74%) do not contain any Kotlin code. The collected data show similar trends and the observed percentages only differ slightly, thus hinting that **Kotlin adoption in Android apps is still an ongoing process**. For researchers and Android tool vendors, this means that Java-based methods and techniques likely remain still valid and applicable for the majority of apps. This result is particularly important since as of today a whole ecosystem of Java-based methods and techniques exists, ranging from static analyzers [25], mostly based on Soot [26] like Flowdroid [27] and PAPRIKA [28], to input generators like IntelliDroid [29], dynamic analysis tools like DroidTrace [30], etc. Nevertheless, Kotlin can be considered as the default language when developing new Android apps and Google is promoting Kotlin as the main language for Android application development [4]. We mirror the invitation of Baresi et al.

saying that **the research community *should get out of the (Java+Android) comfort zone* and strive towards studying new technologies** [31]. In this context, the new technology is the Kotlin programming language and its related ecosystem of tools and libraries [32].

Comparing the proportion of Kotlin code in the mined apps (see Figure 4), we observe that our findings are in line with the results of Mateus et al [5]. Specifically, we can observe that most applications tend to have a proportion of Kotlin code either in the 0-10% range or in the 90-100% one. This result is interesting for researchers and tool vendors since they can reasonably assume that an **Android app to be analyzed is either fully Java- or Kotlin-based, with very few cases where the Java-Kotlin proportion is balanced**. Moreover, Mateus et al. found that 82 out of 244 (33.61%) applications do not contain any Java code and are solely written in Kotlin, while we found 19 out of 451 (4.40%) applications that are solely written in Kotlin and did not ever contain any Java code. Both results provide evidence that apps fully written in Kotlin are still a minority in the ecosystem of open-source Android apps published in the Google Play store. As future work, it would be interesting to investigate on the spread of Kotlin among closed-source/commercial apps in the Google Play store and assess if they exhibit different characteristics with respect to the ones analyzed in this study and in the one by Mateus et al.

For what concerns our results on RQ2, they complement other researches on the impact of a Kotlin migration on open-source applications. We found a negligible difference between the Java version and the Kotlin version of our analysed apps for all metrics. Although the distribution of values is significantly different for cpu, mem, and ft, the actual difference is negligible (as shown by the calculated Cliff's δ presented in Table IV). Therefore, we conclude that although the community found some overhead introduced by the Kotlin language in some cases, as mentioned in Section II, we did not find evidence that the overall impact of migrating to Kotlin is significant when considering the run-time efficiency of an Android app. In summary, we can inform app developers and the maintainers of the Android platform (and its Kotlin runtime) that, according to our experiment, the impact of migrating to Kotlin on the run-time efficiency of Android apps tends to be negligible. Therefore, within the scope of our experiment, **we found no major reasons for an Android developer to not migrate their existing Java Android app to Kotlin or start a new Android app using Kotlin as main programming language**.

VI. Threats to validity

In this section, we discuss the threats to the validity of our study, in accordance with the categorization defined by Cook and Campbell et al. [33], and describe how each of them applies to our study.

Internal validity. It refers to the causality relationship between treatment and outcome [13]. Most metrics in our study are measured using various existing tools. Therefore, we rely on these tools' correctness, and thus, potential issues of these tools could affect our study results. An essential tool we use in our study is CLOC, and therefore we rely on the correctness of its SLOC counting implementation. We also rely on the tools found on the Android devices and accessible via ADB for run-time efficiency-related metrics. We did encounter a single issue with the frame times reported by ADB, and it resulted in a few corrupted frame times (negative or extremely high values of multiple days of time). We mitigated this issue by manually removing the corrupt frame times from our dataset. We also rely on a software-based power profiler (i.e., BatteryStats) instead of a hardware-based one. While a software-based approach is not as accurate as a hardware-based approach, there is evidence that accuracy is comparable [18]. Future replication of our study using a hardware-based power profiler can further mitigate this potential threat to validity.

In our run-time efficiency experiment, maturation might play a role when our test scenarios are run multiple times. We mitigate this potential threat by our extensive setup and reset phase, by performing a two-minute waiting operation between runs, and by executing different treatments in random order. Another possible threat to validity is represented by the various potential interference that can occur on a real device and potentially affect the resulting outcomes. We mitigate these by taking all of the steps listed in Section III-C. These ensure that such inferences are limited as much as possible. The random execution of the different treatments also reduces the chance that such an interference affects only a single treatment and thus biases results.

Construct validity. It deals with the relation between theory and observation [13]. We mitigated potential construct validity threats by defining all details related to the design of our study (e.g., the goal, research questions, tools, variables, statistical analysis procedures) before starting their execution.

External validity. It deals with the generalizability of obtained results [13]. Our research relies on the availability of the full source code and history of an application, so we limited the selection of our subjects to open-source applications. Hence, there is the risk that obtained results might not generalize to all Android applications, including non-open-source ones. In particular, our results might not generalize to more sizeable applications and to applications that perform a more heavy CPU usage. We mitigated this risk by selecting applications from AndroidTimeMachine [11], a dataset that only includes open-source apps published in the Google Play store and hence more likely to adhere to a minimum standard of quality and to be reasonably complex applications.

Due to time constraints, we restricted our run-time efficiency experiment to two devices and ten apps due to the manual work necessary for implementing the test scenarios executed to exercise each application. This potentially impacts the generalization of our found results on run-time efficiency to other devices and apps. We mitigated this potential bias by selecting both a new and an older generation device, that vary significantly in specifications, and selecting apps

using stratified random sampling. Moreover, a fresh install of the applications was performed prior to each experiment run. Hence, our results do not take into account potential improvements in application's performance stemming from Android runtime (ART) has profiler guided optimization [34].

Conclusion validity. It deals with issues that affect the ability to draw the correct conclusions from the outcome of an experiment [13]. We utilize various statistical tests to prove our assumptions and test our hypotheses and therefore limit the room for error when interpreting the experiment results. Additionally, we perform the Benjamini-Hochberg p-value correction procedure to account for potential type one errors. For our results that do not incorporate any statistical tests, we paid close attention to not draw conclusions too quickly and compare our results with the ones found in other studies. Finally, we provide a publicly available replication package[1] that makes it possible to verify our findings independently.

VII. Related work

This section covers related work on Kotlin in Android development and Android run-time efficiency. It provides a brief overview and explains the differences and potential overlap between the related work and our study.

A. Kotlin in Android Development

As previously mentioned, Mateus et al. [5] investigated the adoption and evolution of Kotlin code in Android applications and its impact on code quality. Part of our study replicates their own, investigating of usage of Kotlin in open-source Android apps. The same authors, in another study et al. [35] researched the adoption of various Kotlin features in Android applications and how their usage varies over the application evolution. They found that type inference, lambdas, and safe calls are the most used features and are found in nearly all applications.

Martinez et al. [36] investigate how and why developers migrate from Java to Kotlin and present statistics on migrations in Android applications. The authors aim to answer the *"why"* through interviewing developers and the *"how"* question by analyzing the progress of migration from start to finish (0% to 100% Kotlin SLOC). Oliveira et al. [7] studied how developers are dealing with the adoption of Kotlin in Android applications, their perception about its advantages and disadvantages, and the common problems they face. They do so by analyzing 9,405 Kotlin related questions on the Android Stack Overflow and by interviewing seven Android developers that regularly use Kotlin. They found that Android developers find Kotlin easy to understand and believe that it improves code quality, readability, and productivity. Coppola and colleagues [8] analyzed a set of open-source Android apps in order to research the impact a Kotlin migration has on the success of the application. They do so by statistically analyzing the correlation between the relevance of Kotlin code and the popularity of app releases. They found that projects featuring Kotlin have a higher popularity average and that a statistically significant correlation exists between the presence of Kotlin and the number of stars on the GitHub repository. Therefore, they conclude that migration to Kotlin comes without a cost on popularity among users and fellow developers. Differently from all the mentioned studies, in our work, we complemented the investigation on the level of Kotlin adoption in open-source Android applications with experimentation to investigate the impact of migration to Kotlin on the run-time efficiency of Android apps.

B. Run-time efficiency in Android Development

Hecht et al. [37] investigated the presence of code smells in Android applications and study whether fixing these improves a variety of user experience related performance metrics. They do so through an experiment with two open-source applications treated with the existing code smells fixed. Willocx et al. [38] investigated the run-time efficiency of various mobile cross-platform tools. To do so they conducted an experiment during which ten different versions of the same application, each developed with a different cross-platform framework, are exercised and run-time metrics collected. Malavolta and colleagues [39] assessed the impact of service workers on the energy efficiency of progressive web apps, by running a total of 7 progressive web apps on two devices, while measuring the energy consumption of the devices. Their results highlight that service workers do not have a significant impact on the energy consumption of progressive web apps. Carette et al. [40] developed a tool, called HOT-PEPPER, to automatically correct code smells and evaluate their impact on the energy consumption of Android applications. Using their tool, they derived multiple versions of five open-source apps by correcting each detected smell independently. Experimental validation shows a reduction in energy consumption of up to 4,83% when the code smells are corrected. Our study goals differ from the ones mentioned above, but we share some similarities in collected metrics and experimental design.

VIII. Conclusions and future work

In this study, we performed an in-depth study on the run-time efficiency of the migration to Kotlin for Android applications. To do so, we collected a dataset of open-source Android applications and investigated the current level of Kotlin adoption in Android applications.

From the collected results, we can conclude that Kotlin usage is still limited but rising, and most applications are either almost fully migrated to Kotlin (>90%) or contain very little Kotlin (<10%). With regards to run-time efficiency, we found that migration to Kotlin has a negligible impact on all of our run-time efficiency metrics.

For future work, our run-time efficiency research can be extended to other applications of the Kotlin language (e.g., server-side Kotlin or Kotlin native applications). Our research on the types of migration activities can also be extended to other languages and frameworks (e.g., migration to Swift in iOS development) in order to find results on language migration in general. Finally, we plan to replicate the research on the level of Kotlin adoption in the context of closed-source/commercial Android apps.

REFERENCES

[1] "Mobile operating system market share worldwide." [Online]. Available: https://gs.statcounter.com/os-market-share/mobile/worldwide

[2] Android, *Android Platform : Android Developers*. [Online]. Available: https://developer.android.com/about

[3] Kotlin, "Comparison to java programming language." [Online]. Available: https://kotlinlang.org/docs/reference/comparison-to-java.html

[4] F. Lardinois, "Kotlin is now google?s preferred language for android app development," 2019. [Online]. Available: https://techcrunch.com/2019/05/07/kotlin-is-now-googles-preferredlanguage-for-android-app-development/

[5] B. G. Mateus and M. Martinez, "An empirical study on quality of android applications written in kotlin language," *Empirical Software Engineering*, vol. 24, no. 6, pp. 3356–3393, 2019.

[6] I. Malavolta, E. M. Grua, C.-Y. Lam, R. De Vries, F. Tan, E. Zielinski, M. Peters, and L. Kaandorp, "A framework for the automatic execution of measurement-based experiments on android devices," in *Proceedings of the 35th IEEE/ACM International Conference on Automated Software Engineering Workshops*, 2020, pp. 61–66.

[7] V. Oliveira, L. Teixeira, and F. Ebert, "On the adoption of kotlin on android development: A triangulation study," in *2020 IEEE 27th International Conference on Software Analysis, Evolution and Reengineering (SANER)*. IEEE, 2020, pp. 206–216.

[8] R. Coppola, L. Ardito, and M. Torchiano, "Characterizing the transition to kotlin of android apps: a study on f-droid, play store, and github," in *Proceedings of the 3rd ACM SIGSOFT International Workshop on App Market Analytics*, 2019, pp. 8–14.

[9] "Google teams up with jetbrains to form kotlin foundation - 9to5google," 2018. [Online]. Available: https://9to5google.com/2018/10/04/google-jetbrains-kotlin-foundation/

[10] T. Verge, "The 22 most important things apple announced at wwdc 2014," jun 2014. [Online]. Available: https://www.theverge.com/2014/6/2/5765048/apple-wwdc-2014-os-x-yosemite-ios-8-and-all-the-news-you-need-to-\\know

[11] F.-X. Geiger, I. Malavolta, L. Pascarella, F. Palomba, D. Di Nucci, and A. Bacchelli, "A graph-based dataset of commit history of real-world android apps," in *Proceedings of the 15th International Conference on Mining Software Repositories*, 2018, pp. 30–33.

[12] X. Cheng, Z. Peng, L. Jiang, H. Zhong, H. Yu, and J. Zhao, "Mining revision histories to detect cross-language clones without intermediates," in *Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering*, 2016, pp. 696–701.

[13] C. Wohlin, P. Runeson, M. Host, M. Ohlsson, B. Regnell, and A. Wesslen, *Experimentation in Software Engineering*, ser. Computer Science. Springer, 2012.

[14] "Inspect cpu activity with cpu profiler: Android developers." [Online]. Available: https://developer.android.com/studio/profile/cpu-profiler

[15] Y. Liu, C. Xu, and S.-C. Cheung, "Characterizing and detecting performance bugs for smartphone applications," in *Proceedings of the 36th international conference on software engineering*, 2014, pp. 1013–1024.

[16] A. Asadi, "Everything you need to know about memory leaks in android." Jul. 2019. [Online]. Available: https://proandroiddev.com/everything-you-need-to-know-about-memory-leaks-in-android-\\d7a59faaf46a

[17] A. Muzaffar, "Tame your android apps power consumption - optimize for battery life," Nov. 2015. [Online]. Available: https://android.jlelse.eu/tame-your-android-apps-power-consumption-optimize-for-battery-life-\\6d19d316aa06

[18] D. Di Nucci, F. Palomba, A. Prota, A. Panichella, A. Zaidman, and A. De Lucia, "Software-based energy profiling of android apps: Simple, efficient and reliable?" in *2017 IEEE 24th international conference on software analysis, evolution and reengineering (SANER)*. IEEE, 2017, pp. 103–114.

[19] M. B. Wilk and R. Gnanadesikan, "Probability plotting methods for the analysis of data," *Biometrika*, vol. 55, no. 1, pp. 1–17, 1968.

[20] S. S. Shapiro and M. B. Wilk, "An analysis of variance test for normality (complete samples)," *Biometrika*, vol. 52, no. 3/4, pp. 591–611, 1965.

[21] N. M. Razali, Y. B. Wah *et al.*, "Power comparisons of shapiro-wilk, kolmogorov-smirnov, lilliefors and anderson-darling tests," *Journal of statistical modeling and analytics*, vol. 2, no. 1, pp. 21–33, 2011.

[22] H. B. Mann and D. R. Whitney, "On a test of whether one of two random variables is stochastically larger than the other," *The annals of mathematical statistics*, pp. 50–60, 1947.

[23] D. Thissen, L. Steinberg, and D. Kuang, "Quick and easy implementation of the benjamini-hochberg procedure for controlling the false positive rate in multiple comparisons," *Journal of educational and behavioral statistics*, vol. 27, no. 1, pp. 77–83, 2002.

[24] J. Romano, J. D. Kromrey, J. Coraggio, J. Skowronek, and L. Devine, "Exploring methods for evaluating group differences on the NSSE and other surveys: Are the t-test and Cohen's d indices the most appropriate choices," in *annual meeting of the Southern Association for Institutional Research*. Citeseer, 2006, pp. 1–51.

[25] L. Li, T. F. Bissyandé, M. Papadakis, S. Rasthofer, A. Bartel, D. Octeau, J. Klein, and L. Traon, "Static analysis of android apps: A systematic literature review," *Information and Software Technology*, vol. 88, pp. 67–95, 2017.

[26] R. Vallée-Rai, P. Co, E. Gagnon, L. Hendren, P. Lam, and V. Sundaresan, "Soot: A java bytecode optimization framework," in *CASCON First Decade High Impact Papers*, 2010, pp. 214–224.

[27] S. Arzt, S. Rasthofer, C. Fritz, E. Bodden, A. Bartel, J. Klein, Y. Le Traon, D. Octeau, and P. McDaniel, "Flowdroid: Precise context, flow, field, object-sensitive and lifecycle-aware taint analysis for android apps," *Acm Sigplan Notices*, vol. 49, no. 6, pp. 259–269, 2014.

[28] G. Hecht, N. Moha, and R. Rouvoy, "An empirical study of the performance impacts of android code smells," in *Proceedings of the international conference on mobile software engineering and systems*, 2016, pp. 59–69.

[29] M. Y. Wong and D. Lie, "Intellidroid: A targeted input generator for the dynamic analysis of android malware." in *NDSS*, vol. 16, 2016, pp. 21–24.

[30] M. Zheng, M. Sun, and J. C. Lui, "Droidtrace: A ptrace based android dynamic analysis system with forward execution capability," in *2014 international wireless communications and mobile computing conference (IWCMC)*. IEEE, 2014, pp. 128–133.

[31] L. Baresi, W. G. Griswold, G. A. Lewis, M. Autili, I. Malavolta, and C. Julien, "Trends and challenges for software engineering in the mobile domain," *IEEE Software*, vol. 38, no. 1, pp. 88–96, 2020.

[32] Makery, "Kotlin resources." [Online]. Available: https://www.kotlinresources.com

[33] T. D. Cook, D. T. Campbell, and A. Day, *Quasi-experimentation: Design & analysis issues for field settings*. Houghton Mifflin Boston, 1979, vol. 351.

[34] "Improving app performance with art." [Online]. Available: https://android-developers.googleblog.com/2019/04/improving-app-performance-with-art

[35] B. G. Mateus and M. Martinez, "On the adoption, usage and evolution of kotlin features in android development," in *Proceedings of the 14th ACM/IEEE International Symposium on Empirical Software Engineering and Measurement (ESEM)*, 2020, pp. 1–12.

[36] M. Martinez and B. G. Mateus, "How and why did developers migrate android applications from java to kotlin? a study based on code analysis and interviews with developers," *arXiv preprint arXiv:2003.12730*, 2020.

[37] G. Hecht, N. Moha, and R. Rouvoy, "An empirical study of the performance impacts of android code smells," in *Proceedings of the international conference on mobile software engineering and systems*, 2016, pp. 59–69.

[38] M. Willocx, J. Vossaert, and V. Naessens, "Comparing performance parameters of mobile app development strategies," in *Proceedings of the International Conference on Mobile Software Engineering and Systems*, 2016, pp. 38–47.

[39] I. Malavolta, G. Procaccianti, P. Noorland, and P. Vukmirovic, "Assessing the impact of service workers on the energy efficiency of progressive web apps," in *2017 IEEE/ACM 4th International Conference on Mobile Software Engineering and Systems (MOBILESoft)*. IEEE, 2017, pp. 35–45.

[40] A. Carette, M. A. A. Younes, G. Hecht, N. Moha, and R. Rouvoy, "Investigating the energy impact of android smells," in *2017 IEEE 24th International Conference on Software Analysis, Evolution and Reengineering (SANER)*. IEEE, 2017, pp. 115–126.