Evolution of Kotlin Apps in terms of Energy Consumption: An Exploratory Study

Hesham Ahmed
Vrije University
Amsterdam, Netherlands
h.a.m.s.ahmed@student.vu.nl

Alina Boshchenko
Vrije University
Amsterdam, Netherlands
a.boshchenko@student.vu.nl

Niaz Ali Khan
Vrije University
Amsterdam, Netherlands
n.a.khan@student.vu.nl

Dmitriy Knyajev
Vrije University
Amsterdam, Netherlands
d.knyajev@student.vu.nl

Dinara Garifollina
Vrije University
Amsterdam, Netherlands
d.garifollina@student.vu.nl

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

Matias Martinez
Universitat Politècnica de Catalunya
Barcelona, Spain
matias.martinez@upc.edu

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

***Abstract*— *Context*. Java and Kotlin are the two main programming languages used to create Android applications. Kotlin almost completely replicates the capabilities offered by Java and offers extra features, making it a popular choice among developers. From a sustainability perspective, it is crucial to assess the energy usage of Kotlin-based Android applications.**

***Goal*. The goal of this study is to explore how the energy consumption of Kotlin applications evolves over time. The study also aims at identifying the key factors that influence energy consumption, to inform developers on how the changes they make affect the energy consumed by their applications.**

***Method*. To investigate how Kotlin apps' energy consumption changes through releases, we study three open-source apps Kotlin apps that are also present in the Google Play store. We conduct a measurement-based experiment during which we assess the energy consumed by several releases of each studied application, for a total of 171 executions. Afterwards, we statistically analyse the collected data to identify relevant energy fluctuations (*i.e.,*, spikes, drops). Finally, we manually inspect the source code changes in the apps to identify possible causes of the identified energy fluctuations.**

***Results*. All three studied applications exhibit a growing trend for energy consumption over the course of their releases. Moreover, abnormal energy spikes are found for all applications. There are different causes behind these variations, including OS upgrades, new features, poorly chosen design patterns and libraries, UI issues, and unstable app versions.**

***Conclusions*. Our study provides evidence that a number of not fully understood factors can affect the energy consumption of a mobile application. Further work is needed to study their impact.**

I. Introduction

Sustainable software development is an approach that has emerged in recent years to software design and development that emphasizes energy consumption, energy efficiency, and environmental sustainability. The goal of sustainable software development is to minimize the impact of applications and their infrastructure on the planet [1]. In the context of mobile applications, power consumption is becoming an increasingly significant factor that developers consider when designing and developing mobile applications. Indeed, an application with a power-hungry nature (i.e., it consumes more energy than it should) can leave users with a negative impression, which can lead to negative ratings and uninstalls [2], [3].

Currently, the market for mobile operating systems is led by the Android operating system from Google, which has more than 70% of the market as of August 2022 [4]. Historically, applications that are executed on Android have been written using Java programming language. However, in 2019, Google chose Kotlin as the main language for developing Android apps [5]. Kotlin is an open-source statically typed programming language that targets JVM (Java Virtual Machine), Android, JavaScript, and native apps [6]. There are several new programming features introduced with Kotlin that are not available in Java applications, *e.g.,* extension functions, data classes, and lambda expressions [7], [8]. Moreover, Kotlin has a functional approach and its code is generally considered more readable [9], [10].

The factors mentioned above, among many others, encourage developers to migrate their Android applications to Kotlin [9]. However, while studies have investigated how developers perform a migration to Kotlin [11], [12] and how its benefits are perceived [9], [13], [14], *the long-term impact of Kotlin adoption on energy consumption has not been investigated yet*.

We fill this research gap by investigating how the energy consumption of Kotlin applications evolves over time. To do so, we first select three high-quality open-source Kotlin applications employing well-defined inclusion and exclusion criteria and a quality assessment procedure. Then, we conduct a measurement-based experiment during which we assess 3 times the energy consumed by 57 combined releases (for a total of 171 executions), selected from the three applications. Afterward, we perform a statistical analysis of collected data to identify energy spikes and drops. Finally, we manually inspect the source code changes in the apps to identify possible causes for energy fluctuations.

We found that all the studied applications exhibit a growing trend for energy consumption over the course of their releases. Moreover, abnormal energy spikes are found for all appli-

cations. Investigating the causes behind such variations, we found a variety of factors that can affect energy consumption, including OS upgrades, the introduction of new features, poorly chosen design patterns and libraries, UI issues, and unstable app versions. Our study aims to inform developers, providing them with preliminary evidence on how changes in their application can affect the energy consumption of their applications.

II. Related Work

This paper focuses on two main areas: 1) migration from Java to Kotlin Android applications, and 2) energy consumption of Android applications.

A. Migration from Java to Kotlin in Android

Android developers can choose to write applications entirely or partially in Kotlin thanks to the interoperability between Java and Kotlin. Previous work has studied the transition from Java to Kotlin, with a special focus on Android applications, and studied the proportion of Java and Kotlin code in various versions of apps [12], [11], [13], [15]. For example, Mateus et al. [11] conducted a study on 244 Android applications that have at least one line of code written in Kotlin. Most Kotlin applications (59.43%) have at least 80% of their code written in Kotlin in their most recent version of each application.

Regarding support during migration, Mateus et al. [16] define a machine learning-based approach to support the migration of applications from Java to Kotlin-based code features and metrics. However, that approach does not consider the energy consumption of the app under migration.

Other works have focused on studying the evolution of Kotlin applications. For example, Mateus et al. [8] investigated the adoption and usage of code features provided by Kotlin, such as Coroutines, in 387 Android applications. The authors extracted 26 features after exploring the source code of the applications selected and found out that 15 of them are used in almost all the applications selected. Among those features, lambda expression, type interference, and safe calls are the most used features. Although this study talks about the evolution of Kotlin programming language features, there was no specific focus on the energy consumed by the apps and how the use of those features can impact the energy consumption.

B. Energy consumption of Android applications

Li et al. [17] studied more than 400 apps from the Google Play store and measured energy at various granularities, from the level of the entire application to the level of the source line. One of the trends that they discovered is that across all different kinds of bytecodes, the data manipulation activities (such as moving data between registers and loading operands to registers) utilize the greatest amount of energy. Differently from our work, their study targets only Android Java-based applications.

Cruz et al. introduced an automatic tool "Leafactor" that improves the energy consumption of Android applications [18]. It accomplishes this by reworking the source code to adhere to a set of recognized energy-efficient patterns. 140 open-source apps with 222 refactorings each were used to validate the tool set. Pull requests were made to the official projects in order to contribute changes to the original apps. This paper targeted applications that were written in Java.

Di Nucci et al. [19] performed the software-based energy profiling of Android apps. According to the authors, there are three main categories of tools used to measure the energy consumption of mobile applications, which are i) hardware-based, ii) software-based, and iii) model-based. Besides the advantages of these approaches, however, pose several limitations. In hardware-based tools, there is an extra overhead which is the sophisticated and expensive hardware components, and the goal of the research was to investigate the accurate energy profiling of software-based models that can be a substitute for more sophisticated hardware-based approaches. The software-based tool they proposed, PETrA, estimates the energy consumed by an app at the method and performs similarly to a hardware device that measures the energy consumed. Unlike our work, that work focuses on measuring the energy of a single version of each application that composes their evaluation dataset. On the contrary, in this work, we focus on studying the energy consumed by several versions of the same application.

Malavolta et al. investigated the impact of the run-time efficiency of the migration to Kotlin for Android applications. To achieve the research goal, an empirical study was performed to find out the run-time efficiency impact of migration from Java to Kotlin. In order to perform the experiment, 7,972 open-source Github repositories were examined and among them, 451 Kotlin apps were explored. They then selected 10 applications that were migrated from Java to Kotlin in a single version (i.e., no version shares both Java and Kotlin code). The study found that the migration to Kotlin resulted in a significant impact on CPU usage, render duration of frames, and memory usage. However, migration does not have significant impacts on the number of calls to the garbage collector, energy consumption, and app size [20]. In this paper, we study the energy consumption along the evolution of apps that are gradually migrated [11], those that are not migrated in a single commit.

III. Study Design

A. Goal and Research Questions

Table I shows the formulation of the goal of this study according to the Goal-Question-Metric framework [21].

The above-mentioned goal can be read as follows: *Analyze releases of Android apps for the purpose of evaluation with respect to their energy consumption from the point of view of software developers in the context of Android mobile apps with Kotlin being a dominant language (onward - Kotlin applications).*

In order to be considered the dominant language, we establish that the percentage of Kotlin code in the latest version of an application has to be greater than 70%. This metric

Analyze	Releases of Android apps
for the purpose of	evaluation
with respect to their	energy consumption
from the point of view of	software developers
in the context of	Android mobile apps with Kotlin being a dominant language (onward - Kotlin applications)

TABLE I: Goal of this study

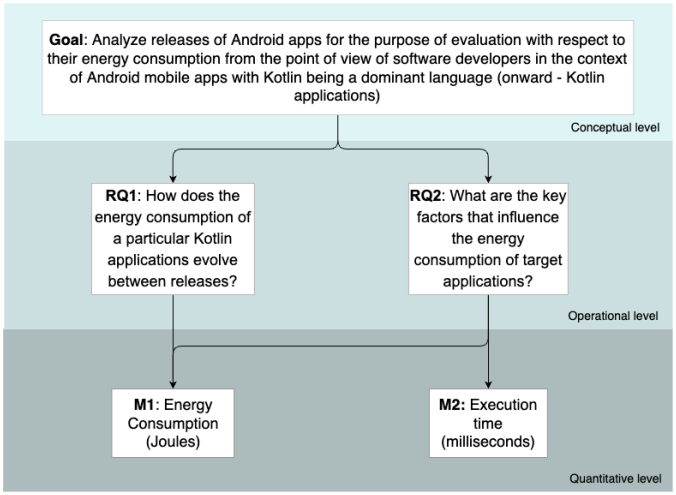


Fig. 1: Overview of the experiment

was chosen to create a definition of the "Kotlin application" and a factor, based on which we select the applications to be studied. In particular, an app A with an amount of Kotlin code greater than this percentage is considered as having Kotlin as a dominant language, thus A qualifies for inclusion in our dataset. An amount lower than this percentage is not enough to consider Kotlin a dominant language in A, so the app does not qualify for inclusion in our dataset.

A visual representation of the experiment we designed to answer the above-mentioned goal is shown in Figure 1. Specifically, we identified two main questions:

RQ1: How does the energy consumption of a particular Kotlin application evolve between releases?

The aim of this question is to investigate the presence of energy consumption improvements or degradation in new releases of Kotlin apps and to identify trends in energy consumption across releases.

RQ2: What are the key factors that influence the energy consumption of target applications?

With this question, we want to identify which specific aspects can lead to major changes in energy consumption along the evolution of an application (*i.e.,* in the sequence of release versions) and what is the rationale behind them.

The metrics we use in the experiment are the following:

- Energy Consumption (Joules) - characterizes the energy consumption of the application;
- Execution time (milliseconds) - characterizes the execution time of the single experiment.

B. Subjects selection

To select a sample of subjects for our experiment that allows for an investigation into the reasons behind certain fluctuations in terms of energy consumption in Kotlin-based mobile apps, we considered a dataset of open-source Android projects by Pecorelli et al. [22]. The dataset is composed of 1,693 open-source mobile apps, of which 139 contain Kotlin code. This dataset is chosen as our starting point because:

- This dataset is a contribution of a recent and large-scale empirical study in the context of Android applications, which means that it contains recent applications, increasing the chance of finding applications written in Kotlin, a programming language recently adopted for Android programming;
- It includes access to the GitHub repository of each application, which provides access to the source code and project history for all applications;
- It contains applications that are available on the Google Play Store, *i.e.,* real-world applications rather than a demo or unfinished applications.

To select the experimental subjects from this dataset, we first define inclusion and exclusion criteria that will allow us to select the first subset of candidate apps. Then, we apply a quality assessment procedure that produces the final sample of apps to be studied.

Table II provides a summary of the apps surviving the inclusion and exclusion criteria and the quality assessment. We now explain in detail the rationale behind each inclusion and exclusion criteria and the checks performed during the quality assessment.

1) Inclusion and exclusion criteria: We present below the inclusion and exclusion criteria applied before the quality assessment was performed, to select the subjects for our study, as shown in Table II.

- *Percentage of Kotlin code* $\geq 70\%$. As previously explained, our study focuses on applications in which Kotlin is the dominant language.
- *Active development.* The application was last updated no longer than six months before the mining was performed at the beginning of the experiment.[1] These inclusion criteria ensure that only applications with ongoing active development are considered.
- *Availability of releases.* GitHub provides a specific page to list and describe the project releases. However, its use is not mandatory and most projects do not explicitly list releases. In our study, we only consider those applications for which releases are listed on the specific page, to ensure the correct identification of releases for each experimental subject.

[1]Check of activity on the repositories under study on September 15, 2022.

Criteria	No of Apps
Initial dataset	139
After inclusion and exclusion criteria	10
After quality assessment	3

TABLE II: Inclusion and exclusion criteria for subjects selection

- *Availability of APKs.* We include only applications for which APK files are available on the releases page of the GitHub repository. Having those APKs is necessary for us to measure the energy consumed by the app during execution. Moreover, by retrieving pre-built APKs for each release, we avoid the need of building specific app versions, which may be unsuccessful due to, for instance, the existence of deprecated or missing dependencies.
- *Presence of test cases.* Availability of test cases is necessary to automatically run and exercise the application, and to measure the energy consumed by the app.
- *Removal of toy applications.* We discard applications that are not published on the Google Play store, to ensure that only real-world applications are included in the study.

2) Quality assessment: After applying inclusion and exclusion criteria, ten apps survived and represent potential subjects for our study. These applications have been further scrutinized during a quality assessment process. The quality assessment has been performed by the authors, and the artifacts related to the quality assessment are available in our replication package.

During the quality assessment, we mainly analyzed two metrics, to select apps of higher quality. As a first metric, we considered the number of *stars* in the application GitHub repository, which is a way for users to manifest interest or satisfaction for a repository [23]. As a second metric, we considered *test coverage* as having a higher test coverage is a sign that the application could be well tested and hence of higher quality. We selected those apps with test coverage greater than 70%. Moreover, we executed the latest version of the applications on our experimental environment (described in Section III-D) to verify their proper operation, in order to know if those are usable for our experiment. Finally, as output for the quality assessment, we select the top three apps with the highest number of stars, test coverage, and that worked on the execution environment as subjects for our experiment.

Table III shows the three apps selected for the experiment (Uhabits, Anki-Android, and WiFi Analyzer), their Google Play and GitHub identifiers, the test coverage and GitHub stars count for each of them. These apps have been chosen after applying the inclusion/exclusion criteria and the quality assessment procedure. All the chosen apps contain a significant proportion of Kotlin code, have high test coverage, are updated regularly, and provide releases on GitHub complete with APKs.

From these applications, we selected releases with the most profound changes, leading to a final number of 57 releases considered in our experiment (11 for WiFi Analyzer, 15 for UHabit, and 31 for Anki-Android). This selection was performed manually by the authors, by surveying the notes of each release with prominent changes. This reduction to fifty-seven releases across the three experimental subjects is necessary, as our experiment necessitates the execution of all releases of included applications. Assuming an average of 20 releases for each of the 10 applications that respect the inclusion and exclusion criteria, assuming three measurement trials for each release, and assuming that the execution of each trial requires 180 seconds, a total of $20 \times 10 \times 180 \times 3 \approx 30$ hours of execution time would be required, which is unfeasible given the available resources.

C. Experimental variables

To answer the research questions defined in Section III we consider the release versions of the target applications as our independent variable. By target applications, we assume the sample applications from the set selected in Section III-B. As a dependent variable, we select the energy consumption (expressed in Joules) per test set run of the corresponding release. As there are 3 applications to be studied, therefore, each of them has separate independent and dependent variables, releases set, and tests set.

D. Experiment design and execution

We conduct a measurement-based experiment during which each application release is executed and exercised, and its energy consumption is recorded. For this purpose, we automate the execution of the experiment for each subject by employing an ad-hoc script that automatically clicks through the application and covers its main features with the necessary waiting operations in between steps. Each script is designed to execute one user scenario for each of the main features that the app contains (*e.g.,* creating a new card in Anki-Android). These scripts were created by manually exercising each app on the test device while recording the performed operations. When executed, the scripts replay the recorded operations and hence reproduce the same behavior in the test subject. Our created test scripts that interact with the subject application are run using MonkeyRunner[2], a utility for automated testing of Android apps. While running the scripts, the energy consumption is measured in parallel. The scripts are available in the online replication package.

Measurements for energy usage might be affected by fluctuations; hence we cannot draw reliable conclusions based on a single experiment run. To mitigate unreliable measures, the experiment is repeated three times for each app release, averaging the results. The three selected apps combined have 57 releases in total and running the test execution associated with each selected release 3 times result in 171 test trials. Assuming that each test will take 120 seconds on average and an idle time of 60 seconds between each run, this results in 171 x 180 = 9 hours of sheer execution time.

Before running the experiment, we manually prepare our device by fully charging the battery, removing the SIM and

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

App Name	Google Play ID	GitHub Repo	Stars #	Kotlin %	Releases #
Uhabits	org.isoron.uhabits	https://github.com/iSoron/uhabits/	5,221	83%	39
Anki-Android	com.ichi2.anki	https://github.com/ankidroid/Anki-Android.git	2,087	98%	1,024
WiFi Analyzer	com.vrem.wifianalyzer	https://github.com/VREMSoftwareDevelopment/WifiAnalyzer.git	1,804	97%	25

TABLE III: Inclusion and Exclusion Criteria for Study Selection

SD card, setting the brightness and sound to a minimum, enabling the stay awake developer option, and finally, disabling network data, Bluetooth, and notifications of other apps. As per Cruz [24], an idle time of 60 seconds is observed between each iteration to prevent tail energy consumption from affecting our measures. Furthermore, the cache will be cleared between every two runs. After executing all three experiment runs for a release, another one is downloaded and installed in its place. The sequence of execution of the releases will be randomized to mitigate the need to have an uncontrolled factor affecting a specific set of test subjects. The scripts and the code to collect the measurements and to perform the setup and reset phases are implemented using Android Runner, an open-source Python framework for automating experiments on Android devices [25].

To obtain accurate measurements, we run all experiment runs using the same hardware and software. The test subjects are executed on a Nokia 6.2 smartphone, running Android 10 and equipped with a Qualcomm SDM636 processor and 4GB RAM. Android Runner is executed on Raspberry Pi 3 equipped with an ARM Cortex-A53 processor and 1 GB RAM and running Debian 11. Two personal computers are used to connect to the Raspberry PI via SSH connection: an Intel Core i7-10750H and an Intel Core i5-10300H, with 16GB and 8 GB of RAM, respectively. The Android device is connected to one of the Raspberry PI USB ports with a USB to USB-C cable and has USB debugging enabled, in order to be able to execute tests with the Android Runner. To reduce the potential impact on the results of the experiment, USB charging is programmatically disabled on the Android device using the developer tools.

E. Data analysis

1) Plotting the energy consumption per application: We assume that we have applications A_k, where $k = 1..K, K = 3$. To answer RQ1 we first look at how for each A_k the level of energy consumption changes between release R_i and releases R_{i+1} assuming that the total number of applications releases A_k is $N_k, N > 1$. For each app A_k, we present the collected data in the form of a plot where the X axis is the releases of A_k, and the Y axis is the energy consumed during the experiment. The plot shows the levels of energy consumption of $R_{1k}..R_{Nk}$ and allows one to notice trends and anomalies in energy consumption. As we execute each release three times, we consider the mean value of consumption across all runs for that release.

The plots also show a smoothing line, which is fitted to the data and used to help explore the relationships between releases and energy consumption. The smoothing line is computed using LOESS (Locally Weighted Scatterplot Smoothing), a nonparametric method for smoothing a series of data in which no assumptions are made about its underlying structure [26].

2) Recognizing Spikes and Drops in the energy consumption: Based on the plot we can notice a trend in the releases time series data and visually recognize the spikes and drops. However, to formally prove the observed behavior, we need to define a criterion of the spikes/drops definition. As a criterion, we decided to pick the following: we consider a spike or drop for every pair of releases R_n-R_{n+1} in which the difference between the energy consumption is higher than the double standard deviation ($2 * SD$) of the entire series of releases of the corresponding application A_k. The reason for choosing double standard deviation instead of just the standard deviation is because the former allows us to identify larger energy fluctuation than using the latter, and thus to exclusively focus on highly significant spikes or drops that reflect anomaly behavior in energy consumption.

3) Studying Spikes and Drops in the energy consumption: To answer RQ2, once we identify one spike or drop, we search for the release on which the trend (spike or drop) starts and that on which it ends. Then, to determine the root cause of the drop or spike in energy consumption, we focus on the releases associated with each spike and drops previously detected. To find the potential reasons behind the energy consumption anomaly represented by a selected spike or drop, we manually check the source code of a release R_i of a target app A_n that has a spike or drop, the release notes related from R_i, and all the commits (which introduce code changes) that were made between R_i and the previous release. Additionally, during this step, we debug these pair release versions for overlapping layouts and views using the Debug GPU Overdraw tool[3], to detect energy spikes due to user interface rendering issues. Based on our findings concerning all the applications $A_1..A_K$ we will present a summary of the most likely reasons for the candidates and related discussions.

F. Study Replicability

For more details about the research such as the research methods, data used for research, and code scripts, the reader can consider the replication package of this study[4]. This replication package is made available to the researchers and practitioners to support independent verification and replication of the study.

[3]https://developer.android.com/topic/performance/rendering/inspect-gpu-rendering

[4]https://github.com/S2-group/ict4s-2023-evolution-kotlin-apps-energy-rep-pkg

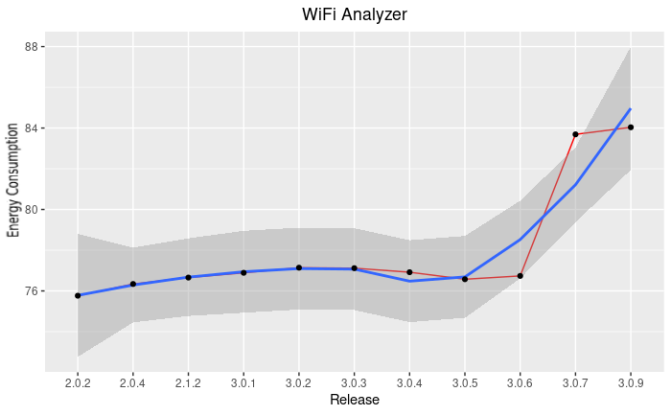


Fig. 2: Energy consumed by WiFi Analyzer across its releases, expressed in Joules (Red line = raw energy consumption; Blue line = LOESS smoothed consumption; Grey area = confidence interval 0.95).

IV. Results

In the following, we describe the results of our experiment, organized by research question.

A. How does the energy consumption of a particular Kotlin application evolve between releases?

In order to detect spikes and drops in energy consumption of the three apps under study, we first calculated the double standard deviation (in Joules) for each application and obtained the following results:

$$2 * SD(WiFiAnalyzer) = 5.8622,$$

$$2 * SD(Uhabit) = 1.635868,$$

$$2 * SD(Ankidroid) = 4.853248.$$

Then, for each application, we visualize these values in a plot, together with (*i*) the raw energy consumed by each release (represented by a red line), (*ii*) a smoothing line using a LOESS method (represented by a blue line), and (*iii*) the confidence interval around the smooth line represented by the dark grey area.

The resulting plots are presented in Figures 2, 3, and 4, which show the energy consumption throughout the release of WiFi Analyzer, Uhabits, and Anki-Android apps, respectively.

We now discuss in detail what can be observed from the plots. Regarding the WiFi Analyzer app (whose energy consumption is shown in Figure 2) it can be noticed that the energy consumption is following the upward trend, with almost all values within the confidence interval area. The only significant spike in energy consumption is the spike between releases 3.0.6 and 3.0.7. Therefore, this spike will be considered in further investigation.

The energy consumption of the Uhabits app is displayed in Figure 3. It can be noticed that the energy consumption does not follow a visible upward or downward trend, but instead has a fluctuation across releases. Almost all the energy consumption values are within the confidence interval area, with two exceptions: a spike in energy consumption between releases 1.8.9 and 2.0.0, and a drop between releases 2.0.0 and 2.0.1. Therefore, these two anomalies will be considered in further investigation in Section IV-B. Interestingly, this latter drop comes in the release immediately after the former spike, so considering this in the qualitative investigation we aim to understand more in detail the root causes behind the spike and how this was addressed in the subsequent release.

The energy consumption of Anki-Android can be observed in Figure 4. It can be observed that the energy consumption is following a slight upward trend, with a lower number of values within the confidence interval area compared to the other two applications. Based on the mean values of energy consumption, in the series of releases, there is a valid spike between releases 2.16.25 and 2.16.34, followed by a drop in energy consumption between 2.16.37 and 2.16.43. The spike/drop is considered valid when the difference between two consecutive releases is more than double the standard deviation, according to the metric described above. We calculated this difference for each spike/drop, and other spikes/drops which are visible on the plot did not match these criteria.

Therefore, these anomalies will be considered in further investigation.

B. What are the key factors that influence the energy consumption of target applications?

1) WiFi Analyzer: There is an increase in the energy consumption of WiFi Analyzer from v3.0.6 to v3.0.7. Between these versions, there were 14 commits and the number of lines of code decreased slightly from 4,592 to 4,586, as can be observed in Figure 5. However, we did not find any major change between these two revisions, such as the introduction of a new feature or a migration from one language to another. The mentioned commits introduced minor changes, such as

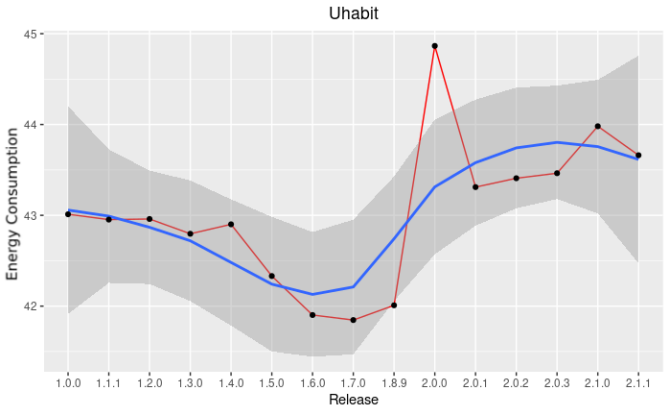


Fig. 3: Energy consumed by Uhabits across its releases, expressed in Joules. (Red line = raw energy consumption; Blue line = LOESS smoothed consumption; Grey area = confidence interval 0.95).

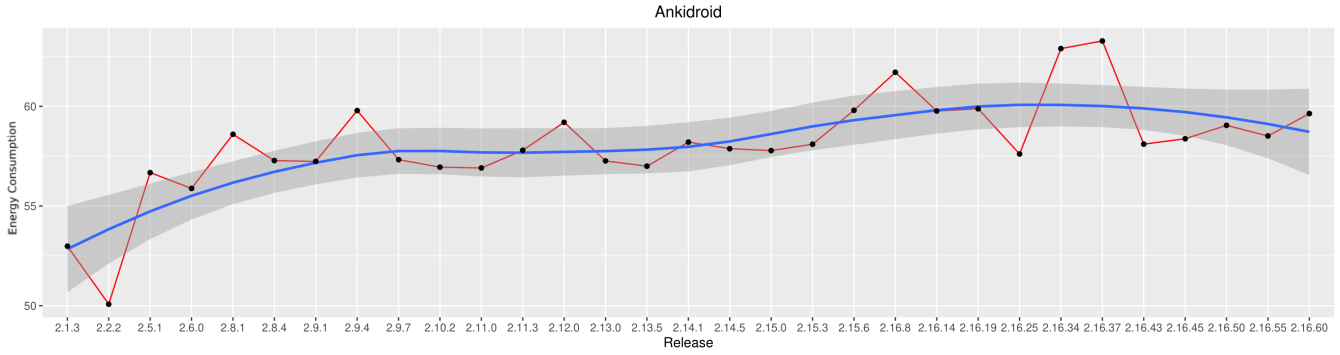


Fig. 4: Energy consumed by Anki-Android across its releases, expressed in Joules. (Red line = raw energy consumption; Blue line = LOESS smoothed consumption; Grey area = confidence interval 0.95).

bug fixes. Thus, we hypothesize that a possible reason for this spike might be the addition of support for Android 12 [27]. Indeed, this is consistent with observations from a study of Pathak et al. [28], who studied bug reports and developer discussions, and found that about 20% of energy-related defects appeared in Android apps after the introduction of an OS upgrade.

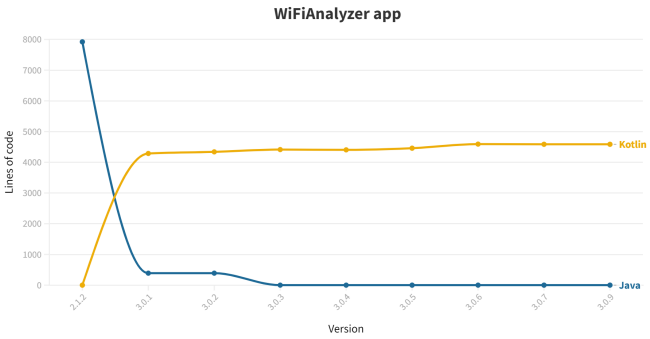


Fig. 5: Lines of code of WiFi Analyzer

We also studied the impact of energy consumption on other major changes that occurred along the WiFi Analyzer lifecycle. First, we focus on the migration from Java to Kotlin. WiFi Analyzer's code base migrated fully from Java to Kotlin in one release (3.0.1), which occurred before the release with an energy spike (3.0.7). The release prior to the migration, *i.e.,* 2.1.2, has 149 Java files and 5,848 lines of code, and the revision after the migration, 3.0.1, has 130 Kotlin files with 3,294 lines of code. The reduction in the number of lines of Kotlin code with respect to Java code is one of the main motivations to perform migrations [9], as it may improve the maintainability of the application. The application under study is an example of this statement: the migration reduced the total amount of code by more than 40%. Even though the code almost halved, it did not affect the energy trends, which stayed almost constant at about 77 Joules before and after the migration.

Secondly, after inspecting the release notes, we observed the adoption of the DataBinding library[5], which allows the integration of UI components to data sources using a declarative format rather than programmatically. WiFi Analyzer started to use DataBinding from version 2.1.2 to bind UI elements to data. This is considered an antipattern (argumented by, for instance, [29], [30]) as business logic is written in the UI, *i.e.,* directly into the XML files of the definition of UI layouts. This practice impacts code readability, thus adding complexity to code refactoring and maintainability [31]. DataBinding is supposed to improve app performance because binding occurs during compilation time by generating binding classes in advance [32]. However, Figure 3 does not show any substantial difference in energy consumption between revision 2.0.4 and revision 2.1.2, *i.e.,* the version before and after the DataBinding introduction.

2) Uhabits: Uhabits conducted its migration from Java to Kotlin gradually: it took 13 releases from version 1.8.7 (fully written in Java) to version 2.0.1 (fully written in Kotlin). The releases between them mix Java and Kotlin code. We recall that this migration is different from that one done in the WiFi Analyzer, which was fully migrated from one release to the next one (this is called 'one-step migration' [11]). A sharp rise in energy consumption can be observed in Figure 3, in release 2.0.0. Partially, this spike can be explained by the unstable nature of release 2.0.0, as it was an alpha pre-release. Moreover, one of the new functionalities introduced in this version (*e.g., "synchronization across devices"*) was labeled as buggy and removed in version 2.0.1. The same functionality was later fixed and reintroduced in version 2.1. Energy consumption also decreased by about 2 Joules in release 2.0.1. Another potential reason for this energy spike is the introduction of six new features in version 2.0.0, which introduced complexity in the application code: *synchronization across devices*, *tracking of numerical habits*, *adding notes to habits*, *skipping days without breaking streak*, *showing question marks for missing data*, and *delaying start of a new day until 3 am*.

Moreover, we found that version 2.0.1 introduced code

[5]https://developer.android.com/topic/libraries/data-binding

refactors. In particular, developers started using question marks and assertion operators to avoid Null Pointer Exception (NPE) errors. Failure to prevent NPE errors may result in no-sleep energy bugs [33]. In addition to null safety checks, other new Kotlin features were introduced, such as extension functions in utilities for Activities, Fragments, Dialogs, Views and other primitive data types, wrapper classes as well as scope functions such as apply(), let(), run(), with() which considerably reduced all the boilerplate code and increased readability.

During the GPU overdraw inspection of Uhabits, we analyzed all versions with significant spikes mentioned above. As a result, we found issues in several UI elements: the habit widget, the toolbar on the app's main page, and almost all elements on the about page. An example can be seen in Figure 7 a) where the red color identifies elements that were overdrawn 4 or more times [34]. Following this inspection, the code was analyzed to double-check the presence of UI issues. Indeed, the XML layouts have a complex view hierarchy as well as unneeded background layouts. For instance, in the About Page (presented in Figure 7 a)), a grouping of displayed elements was achieved using multiple TextViews inside a ScrollView. The same effect could have been achieved using a singular LinearLayout and setting the margins accordingly. In addition, the list that displays the developers of the application located at the bottom of the page was created using 194 TextView elements. A more efficient solution to display lists is to use RecyclerView [35]. All of that has an impact on energy consumption.

Uhabits in version 2.0.0 switched from AsyncTask threads to Kotlin coroutines for its asynchronous code. Kotlin coroutines are a lightweight concurrency design pattern [36]. In a study of Saoungoumi-Sourpele et al. [37], Kotlin coroutines were found to be more CPU- and memory-efficient when compared to conventional AsyncTask threads. However, Uhabits exhibits an energy spike where it adopted coroutines, hinting that adoption of coroutines might not always be beneficial with respect to energy consumption.

3) Anki-Android: Anki-Android is still in its migration process from Java to Kotlin and has not released any stable version so far. To this date, there are 89 alpha releases of the 2.16 version. The distribution of lines of code for Anki-Android is visible in Figure 8. Before the beginning of the migration, version 2.15.6 had 64,446 lines of pure Java code, and the latest 2.16.89 version was *fully* rewritten in Kotlin with 71,887 lines of Kotlin code. Indeed, even if the lines of code have slightly increased, this does not contrast with the intuition that Kotlin helps in reducing the code size, since a large number of new features have been added throughout these releases, covering a period of more than a year.

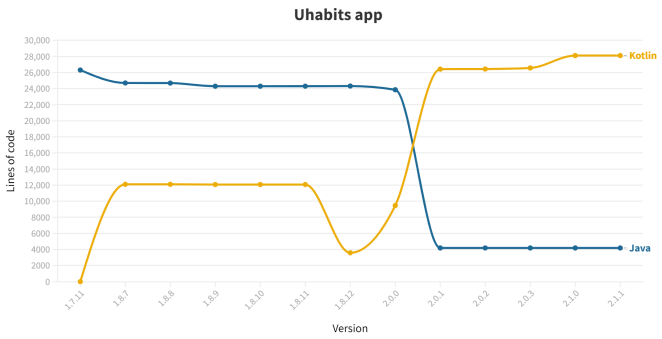
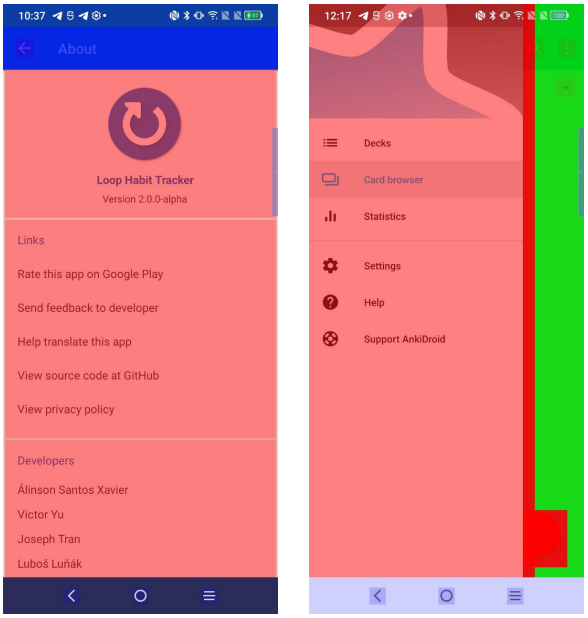


Fig. 6: Lines of code of Uhabits



a) Uhabits b) Anki-Android

Fig. 7: GPU overdraw debugging of two apps (Blue area = no overdraw; Green area = overdrawn more than two times; Red area = overdrawn four or more times)

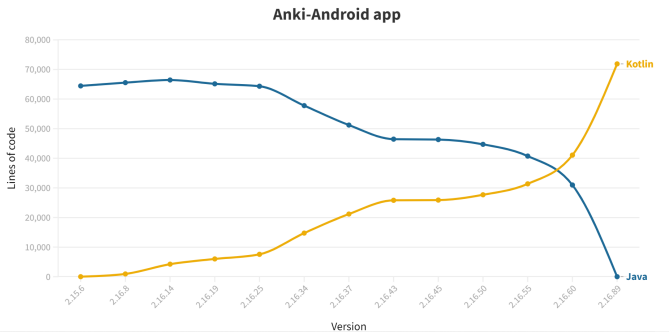


Fig. 8: Lines of code of Anki-Android

The graph shows a significant decrease of 3 Joules in energy consumption in version 2.2.2 which could be explained by a large number of fixes and improvements made in this version, such as: improving the speed of showing cards, removing animation as it was buggy, simplifying menus and setting, making volume duck on any background music, fixing whiteboard feature, removing duplication and many more. Afterward, this downward trend was followed by a sharp increase of 7 Joules

between versions 2.2.2 and 2.5.1. The reason behind such a spike could be full support for APKG export and import which was added in version 2.3 and was reported as an issue later due to its slow performance [38] as well as support Android 6 Marshmallow support [39].

Moreover, between versions 2.6.0 and 2.8.1, there was a noticeable rise of 3 Joules due to new features added, namely sending apkg to arbitrary app (*e.g.,* Google Drive), displaying AnkiWeb, and new widgets. A similar spike was noted between these releases 2.9.1 and 2.9.4 which could be explained with support for new AnkiWeb encryption changes and various patches for Anki Desktop added. Versions 2.9.4 to 2.10.2 showed a dramatic drop in energy use, which was maintained at the same level as it was in the previous 2.5.1 version or five years earlier. It is explained by useful light-weight new alternative tools such as addition of CSS style capability to heavy checkmark and down arrow in a card, support for card Javascript to reload current card programmatically, as well as over 50 minor to serious fixes (e.g. crashes during card rendering, auto-sync, audio recording, high-frequency Webview, multimedia, etc.) and around 10 improvements such as improving performance, v2 scheduler compatibility with Anki ecosystem, handling or detection of full sync, user messaging on network connection failures and many more. Regarding spikes and drops in version 2.16 could be explained by the unstable nature of the version, which has only 89 alpha releases to this date.

From GPU overdraw debugging Anki-Android (Figure 7 *b*)), we noticed that from version 2.10 the Hamburger Menu page has been red-colored, which means that this view was overdrawn four or more times for this version. This issue was less impactful in previous versions from Anki-Android, in which a lower number of overdraws were found [34].

V. Discussion

In this section, we elaborate on our findings and on their implications, grouped by research questions.

A. RQ1: How does the energy consumption of a particular Kotlin application evolve between releases?

In our work, we analyze the energy consumption across releases of three Kotlin applications. To determine significant fluctuations in energy consumption of the analyzed applications, we adopted a margin of error equal to two times the standard deviation, and in all the applications, we detected anomalies that exceed this threshold. During our analysis, we found at least one upward trend in energy consumption for all three applications considered. These spikes are usually followed by minor fluctuations (such as in the case of Anki-Android and WiFi Analyzer) and more significant ones (such as in the case of Uhabits).

These results show the need to increase the testing effort before doing a release, by carrying out more exhaustive functional and non-functional tests, which include, for example, the measurement and assertion of the energy consumption of the changes introduced by a release. As we have seen, for instance, in Uhabits, an energy bug can be introduced in a new version of the application. This finding calls for further research on the detection of energy bugs.

B. RQ2: What are the key factors that influence the energy consumption of the target applications?

We conducted a manual investigation into the root causes behind the detected energy spikes and drops, analyzing the applications' source code, their release notes, and the presence of UI issues. As a result, we highlight the following factors which, according to our investigation, have a direct impact on the appearance of spikes or drops in energy consumption.

In the WiFi Analyzer application, we found that a possible reason for a spike might be the addition of support for Android 12, which was released in October 2021. This finding is a call for further research on the impact of adding support to the most recent Android releases, and how this impact can be minimized. Consider that every Android application will eventually need to support newer versions of Android.

At the same time, despite WiFi Analyzer's code base migrated fully from Java to Kotlin in one release and even though the size of the app almost halved in terms of lines of code, it did not affect the energy trends. This finding could mean that the code features used before and after migration (from Java and Kotlin, respectively) consume a similar amount of energy for carrying out the same tasks. That could be explained as the fact that developers, even after migrating from Java to Kotlin, continue to program in Kotlin as they used to do using Java, that is, for instance, using the same or similar code features and not adopting the new ones introduced by Kotlin (listed in [11]). Consequently, as the code in Java and Kotlin may look similar, the bytecode could also be (we recall that both Java and Kotlin are compiled to JVM bytecode). For this reason, we do not observe variations in energy consumption before and after migration. Future research shall be conducted to determine the impact on the energy consumption of each Kotlin feature.

In the Uhabits application, we observed a significant spike followed by a significant drop. This could be explained by the unstable nature of the alpha pre-release with a synchronization bug, which was removed in the next release to be fixed in version 2.1. This shows that the introduction of new application features and the adoption of new language features or new third-party libraries should be thoroughly tested in order to check if they introduce a regression (energy or functional) bug. This finding calls for further research on the detection of energy bugs.

VI. Threats To Validity

In the following, we describe the threats to the validity of our study. We discuss the threats by making use of the classification by Cook and Campbell [40].

Internal Validity. Internal validity refers to the causality relationship between treatment and outcome [41]. In our experiment, maturation might play a role when our test scenarios are run multiple times. We mitigate this potential threat through

our extensive setup and reset phase, by performing a two-minute waiting operation between runs, and by executing different releases 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 the steps listed in Section III-D, to ensure that such inferences are limited as much as possible. However, during testing of the Anki-Android application versions 2.16.34 and 2.16.37 excessive heating of the phone was detected, which could have affected the results. Moreover, we used fixed devices and application parameters throughout the entire experiment cycle, since there is a risk of obtaining invalid data when employing different instruments, environments, and devices during the test.

External Validity. External validity deals with the generalizability of obtained results [41]. Our study relies on the availability of the full source code and development 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 Kotlin applications, including non-open source ones. We mitigated this risk by selecting applications from a dataset that only includes open-source apps published in the Google Play store and by defining a set of clear inclusion and exclusion criteria that allowed us to select applications more likely to be representative of real-world Kotlin applications. Moreover, per performed a quality assessment of candidate applications that ensures that apps used in our study adhere to a minimum standard of quality and complexity.

Due to time constraints, we restricted our experiment to three applications and fifty-seven releases, due to the manual work necessary for implementing the test scenarios and the required execution time needed to exercise each application. This potentially impacts the generalizability of our results to other apps. We mitigated this threat by conducting a qualitative analysis of releases for which an energy anomaly was identified. During this analysis, we identified and discussed several root causes that are likely to be found in other Kotlin and Android applications.

Construct Validity. Construct validity deals with the relation between theory and observation [41]. We mitigated potential construct validity threats by defining all details related to the design of our study (*e.g.,* the goal, research questions, and tools) before starting its execution. However, as we used one energy profiler (*batterystats*) to collect the energy measurements, our experiment suffers from mono-method bias. To limit the potential impact of this threat, we repeated the experiment three times for each tested release and considered the mean value across all three measurements.

In addition, to ensure that interactions with the applications during the experiments are representative of real-world usage, we constructed usage scenarios that exercise the main functionalities of tested applications. These main functionalities were identified prior to beginning the experimentation, by manually inspecting the applications. To obtain reliable data from different releases the main functionalities were tested in all versions of the same application. After conducting the experiment, the experiment logs were analyzed for anomalies, to build confidence in the correct execution of the experiment.

Conclusion Validity. Conclusion validity deals with issues that affect the ability to draw the correct conclusions from the outcome of an experiment [41]. A potential threat to the validity of our conclusions comes from the limited sample size used in our experiment, due to the considerable manual effort required to design scenarios and execute the experiment. To mitigate this threat we used a conservative double standard deviation margin of error in our statistical analysis, to limit the risk of false positives in the identified energy spikes or drops. Moreover, we complemented our statistical analysis with a qualitative investigation of releases for which energy anomalies were identified. For all the manually analyzed releases, we identified antipatterns or technical reasons that are likely to be the root causes behind the identified energy anomaly. Finally, we provide a publicly available replication package to independently verify our findings.

VII. Conclusions

We conducted a study to evaluate energy consumption during the evolution of Kotlin mobile apps. An experiment has been conducted by choosing three apps from a predetermined dataset and analyzing their energy consumption across all app releases. From the results obtained, we can conclude that energy consumption is generally following an upward trend, and in all the applications we detected significant energy spikes and drops. Among the major factors that impacted those fluctuations, we can underline OS upgrades, the release of new features, poorly chosen design patterns and libraries, UI issues, and unstable app versions.

We highlight the following possible directions for future research on Kotlin-based Android applications:

- Design, development, and evaluation of new approaches for detecting energy (regression) bugs in Android apps;
- Conduct further experiments on the impact of supporting more recent Android releases on the energy consumption of Android apps;
- Conduct further experiments on the impact of the usage of Kotlin-specific features on the energy consumption of Android apps;
- Extending the dataset with more application and repeating the experiment on a wider suite of test devices to build confidence in our results;
- Further in-depth source code investigations, in order to find out more about the underlying reasons behind energy consumption fluctuations.

Acknowledgement

This paper has been funded by the "Ramon y Cajal" Fellowship (RYC2021-031523-I) and the GAISSA Spanish research project (ref. TED2021-130923B-I00; MCIN/AEI/10.13039/501100011033).

REFERENCES

[1] R. Verdecchia, P. Lago, C. Ebert, and C. De Vries, "Green it and green software," *IEEE Software*, vol. 38, no. 6, pp. 7–15, Nov. 2021, publisher Copyright: © 1984-2012 IEEE.

[2] 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.

[3] H. Khalid, E. Shihab, M. Nagappan, and A. E. Hassan, "What do mobile app users complain about?" *IEEE software*, vol. 32, no. 3, pp. 70–77, 2014.

[4] "Mobile Operating System Market Share Worldwide | Statcounter Global Stats," Feb. 2023, [Online; accessed 16. Feb. 2023]. [Online]. Available: https://gs.statcounter.com/os-market-share/mobile/worldwide

[5] "Android's Kotlin-first approach," Aug. 2022, [Online; accessed 16. Feb. 2023]. [Online]. Available: https://developer.android.com/kotlin/first

[6] "FAQ | Kotlin," Feb. 2023, [Online; accessed 16. Feb. 2023]. [Online]. Available: https://kotlinlang.org/docs/faq.html

[7] Y. T. Daniela Gotseva and P. Danov, "Comparative study java vs kotlin," in *Proc. 27-th National Conference with International Participation "TELECOM 2019"*, 2019.

[8] B. G. Mateus and M. Martinez, "On the adoption, usage and evolution of kotlin features in android development," New York, NY, USA, 2020. [Online]. Available: https://doi.org/10.1145/3382494.3410676

[9] M. Martinez and B. Gois Mateus, "Why did developers migrate android applications from java to kotlin?" *IEEE Transactions on Software Engineering*, vol. 48, no. 11, pp. 4521–4534, 2022.

[10] ONLINE, "Conversational Kotlin: A Look at the Benefits of Readable Code ," 2022, https://gs.statcounter.com/os-market-share.

[11] B. Góis 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. [Online]. Available: https://doi.org/10.1007/s10664-019-09727-4

[12] M. Peters, G. L. Scoccia, and I. Malavolta, "How does migrating to kotlin impact the run-time efficiency of android apps?" in *2021 IEEE 21st International Working Conference on Source Code Analysis and Manipulation (SCAM)*, 2021, pp. 36–46.

[13] 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*, ser. WAMA 2019. New York, NY, USA: Association for Computing Machinery, 2019, p. 8–14. [Online]. Available: https://doi.org/10.1145/3340496.3342759

[14] 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.

[15] M. Kim, Y. Kim, H. Jeong, J. Heo, S. Kim, H. Chung, and E. Lee, "An empirical study of deep transfer learning-based program repair for kotlin projects," in *Proceedings of the 30th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, ser. ESEC/FSE 2022. New York, NY, USA: Association for Computing Machinery, 2022, p. 1441–1452. [Online]. Available: https://doi.org/10.1145/3540250.3558967

[16] B. G. Mateus, M. Martinez, and C. Kolski, "Learning migration models for supporting incremental language migrations of software applications," *Information and Software Technology*, vol. 153, p. 107082, 2023. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0950584922001914

[17] D. Li, S. Hao, J. Gui, and W. G. Halfond, "An empirical study of the energy consumption of android applications," in *2014 IEEE International Conference on Software Maintenance and Evolution*, 2014, pp. 121–130.

[18] L. Cruz, R. Abreu, and J.-N. Rouvignac, "Leafactor: Improving energy efficiency of android apps via automatic refactoring," p. 205–206, 2017. [Online]. Available: https://doi.org/10.1109/MOBILESoft.2017.21

[19] 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?" pp. 103–114, 2017.

[21] R. Solingen, V. Basili, G. Caldiera, and D. Rombach, *Goal Question Metric (GQM) Approach*, 01 2002.

[20] M. Peters, G. L. Scoccia, and I. Malavolta, "How does migrating to kotlin impact the run-time efficiency of android apps?" pp. 36–46, 2021.

[22] F. Pecorelli, G. Catolino, F. Ferrucci, A. De Lucia, and F. Palomba, "Software testing and android applications: A large-scale empirical study," *Empirical Softw. Engg.*, vol. 27, no. 2, mar 2022. [Online]. Available: https://doi.org/10.1007/s10664-021-10059-5

[23] H. Borges and M. T. Valente, "What's in a github star? understanding repository starring practices in a social coding platform," *Journal of Systems and Software*, vol. 146, pp. 112–129, 2018.

[24] L. Cruz, "Green software engineering done right: a scientific guide to set up energy efficiency experiments," http://luiscruz.github.io/2021/10/10/scientific-guide.html, 2021, blog post.

[25] 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*, ser. ASE '20. New York, NY, USA: Association for Computing Machinery, 2021, p. 61–66. [Online]. Available: https://doi.org/10.1145/3417113.3422184

[26] E. H. Frank, "Regression modeling strategies with applications to linear models, logistic and ordinal regression, and survival analysis," 2015.

[27] ReactiveCircus, "android-emulator-runner," Feb. 2023, [Online; accessed 16. Feb. 2023]. [Online]. Available: https://github.com/ReactiveCircus/android-emulator-runner/issues/222

[28] A. Pathak, Y. C. Hu, and M. Zhang, "Bootstrapping energy debugging on smartphones: a first look at energy bugs in mobile devices," in *Proceedings of the 10th ACM Workshop on Hot Topics in Networks*, 2011, pp. 1–6.

[29] Q. Gil, "Antipattern: Ui databinding android," 2019, https://quinngil.com/2019/02/05/antipattern-ui-databinding-android/.

[30] T. Mutton, "An argument against data binding," 2017, https://medium.com/@hellotimmutton/an-argument-against-data-binding-13e2aaf7a9b1.

[31] Q. Gil, "Antipattern: UI Databinding Android ," 2019, https://quinngil.com/2019/02/05/antipattern-ui-databinding-android/.

[32] P. Aideloje, "Using data binding to prevent slow rendering in Kotlin - LogRocket Blog," *LogRocket Blog*, Jul. 2022. [Online]. Available: https://blog.logrocket.com/data-binding-prevent-slow-rendering-kotlin

[33] A. Pathak, A. Jindal, Y. C. Hu, and S. P. Midkiff, "What is keeping my phone awake? characterizing and detecting no-sleep energy bugs in smartphone apps," in *Proceedings of the 10th international conference on Mobile systems, applications, and services*, 2012, pp. 267–280.

[34] "Inspect GPU rendering speed and overdraw," May 2022, [Online; accessed 16. Feb. 2023]. [Online]. Available: https://developer.android.com/topic/performance/rendering/inspect-gpu-rendering

[35] "RecyclerView | Android Developers," Jan. 2023, [Online; accessed 16. Feb. 2023]. [Online]. Available: https://developer.android.com/reference/androidx/recyclerview/widget/RecyclerView

[36] "Kotlin coroutines on Android," Sep. 2022, [Online; accessed 16. Feb. 2023]. [Online]. Available: https://developer.android.com/kotlin/coroutines

[37] R. Saoungoumi-Sourpele, J. M. Nlong, J.-R. K. Kamdjoug, and G. V. Yufui, "Improve image decoding in lightweight environment using a coroutines based approach," *Journal of Computer and Communications*, vol. 8, no. 10, pp. 60–74, 2020.

[38] ankidroid, "Deleting decks and importing cards from apkg is too slow," Feb. 2023, [Online; accessed 17. Feb. 2023]. [Online]. Available: https://github.com/ankidroid/Anki-Android/issues/2547

[39] ONLINE, "After marshmallow update l, Nexus 5 has become very slow ," 2015, https://forums.androidcentral.com/legacy-android-other-oss/596561-after-marshmallow-update-l-nexus-5-has-become-very-slow.html.

[40] T. Cook and D. Campbell, *Quasi-Experimentation: Design and Analysis Issues for Field Settings*. Houghton Mifflin, 1979.

[41] C. Wohlin, P. Runeson, M. Höst, M. C. Ohlsson, B. Regnell, and A. Wesslén, *Experimentation in software engineering*. Springer Science & Business Media, 2012.