Kotlin Assimilating the Android Ecosystem: An Appraisal of Diffusion and Impact on Maintainability

The Journal of Systems and Software 222 (2025) 112346

Contents lists available at ScienceDirect

The Journal of Systems & Software

journal homepage: www.elsevier.com/locate/jss

Kotlin assimilating the Android ecosystem: An appraisal of diffusion and impact on maintainability☆

Riccardo Coppola *, Tommaso Fulcini, Luca Ardito, Marco Torchiano

Department of Control and Computer Engineering, Polytechnic University of Turin, Italy

ARTICLE INFO

Keywords:
Software maintainability
Android development
Kotlin

ABSTRACT

Kotlin was introduced in 2011 as an alternative to the Java programming language, promising to address many of its predecessor's limitations and positioning itself as a better option for application maintainability. In 2017, Kotlin became a first-class language for Android application development, complete with extensive tool support.

This paper aims to empirically assess the diffusion of Kotlin in developing Android applications and to investigate the impact of Kotlin adoption on application maintainability.

We mined 2708 open-source Android applications from F-Droid, focusing on the extent of Kotlin code presence, their popularity, and maintainability. This analysis adopted a set of six code metrics proxies.

The proportion of applications developed with Kotlin, either in conjunction with Java or exclusively, has continuously increased over the past five years. Currently, Kotlin is used in approximately 40% of the projects. The adoption of Kotlin in application development appears to be linked to greater popularity among end-users and developers when compared to the applications written in Java. Notably, the exclusive use of Kotlin in projects significantly enhances all the considered code maintainability metrics.

We conclude that Kotlin is rapidly gaining ground in the Android ecosystem. This trend is likely due to Kotlin's fulfilment of its promise as a superior alternative to Java, particularly in terms of maintainability.

1. Introduction

The Kotlin language was introduced in 2011, primarily as a direct alternative to Java, with which it can seamlessly coexist. Notably, Kotlin has been associated with Android development since Google made it a first-class language for writing applications in this domain (Akhin and Belyaev, 2021).

Kotlin was developed to address several challenges inherent to the Java language: handling *null* values, which can lead to NullPointerExceptions (Bose et al., 2018); enhancing maintainability by making code easier to manage throughout an application's evolution (Andrä et al., 2020); improving understandability and conciseness, thereby directly impacting developers' productivity by making code more compact and easier to comprehend (Hellbrück, 2019); and avoiding common Java coding pitfalls, such as mandatory casts, extensive use of argument lists, and data classes (Bose et al., 2018).

The novelty of the language, coupled with its characteristics mentioned above, has led to its widespread adoption among Android application developers (Oliveira et al., 2020). It is commonly reported that developers adopt Kotlin to enhance the quality of their code. However, the actual impact of Kotlin on the maintainability of codebases, compared to traditional Java-based Android applications, remains unclear.

The objectives of this paper are threefold: (i) we extend a preliminary analysis of the state of popular Android application repositories in terms of Kotlin adoption, building upon a dataset originally mined in 2019 (Coppola et al., 2019); (ii) we compute a set of state-of-the-art software quality and maintainability metrics for Kotlin; (iii) we empirically analyse the impact of Kotlin adoption on both the popularity and ratings of the applications, as well as the maintainability of software projects.

The remainder of the paper is organized as follows: Section 2 provides background information on the Kotlin programming language and maintainability metrics and reviews related work in analysing the non-functional properties of projects written in Kotlin; Section 3 describes the research method used in this study; Section 4 presents the results, which are further discussed in Section 5, including potential threats to

☆ Editor: Lingxiao Jiang.

* Corresponding author.
E-mail addresses: riccardo.coppola@polito.it (R. Coppola), tommaso.fulcini@polito.it (T. Fulcini), luca.ardito@polito.it (L. Ardito), marco.torchiano@polito.it (M. Torchiano).

https://doi.org/10.1016/j.jss.2025.112346
Received 23 December 2023; Received in revised form 12 November 2024; Accepted 12 January 2025
Available online 21 January 2025

the validity of this study; Section 6 concludes the paper and suggests directions for future research.

2. Background

In this section we report background information about the Kotlin Language, the definition of Maintainability Metrics and available tooling, and existing related work in the field of maintainability measurement for mobile applications.

2.1. The Kotlin language

The Kotlin language was developed by JetBrains in 2011 to enhance programming on the Java Virtual Machine (JVM). Although the first stable release of the language was in 2016, Kotlin became a first-class language for Android development in May 2017, with Android Studio IDE supporting it since version 3.0 in October 2017. Google self-reports that more than 60% of Android professional developers use Kotlin as their primary language.[1] Academic articles have highlighted this trend of transitioning towards Kotlin. For instance, Martinez and Mateus found that already in 2020 23% applications of a dataset of 374 were fully migrated to Kotlin (Martinez and Mateus, 2020), citing the exclusive features of the language as the main reasons for such migration (Martinez and Mateus, 2021). Hecht and Bergel quantified that in 2021 60% of the top 1000 Android apps already contained Kotlin code (Hecht and Bergel, 2021). Furthermore, the Kotlin language website highlights that developers interviewed feel more productive using Kotlin compared to Java (Anon., 2023). A significant increase in Kotlin adoption was also driven by the Android programming guidelines and learning materials provided by Google, which are now primarily in Kotlin. Major corporations, including Facebook, have begun transitioning their Java codebases to Kotlin (Anon., 2022).

A key factor in Kotlin's rapid adoption among developers is its full interoperability with the Java language, as both languages run on the JVM and can be freely mixed (Martinez and Mateus, 2021). The developers of Kotlin adopted a pragmatic approach, opting not to reimplement the entire Java collection framework and libraries but rather to ensure compatibility with the JDK interfaces. Thus, the transition to Kotlin is particularly advantageous when developing additional modules or new features in established projects without requiring complete re-translation — as the same authors highlight, in fact, the complete interoperability with Java is among the top five reasons for the migration to the Kotlin language.

Although a detailed description of Kotlin's features is beyond the scope of this work, we highlight some of its key peculiarities, either claimed or verified by empirical research, compared to Java:

- Kotlin is perceived as having a relatively flat learning curve for developers, being a more modern language than Java;
- Kotlin's design aims for safer code compared to Java, potentially leading to fewer system failures and application crashes (Flauzino et al., 2018). This is mainly achieved by supporting non-nullable types, which can reduce the likelihood of null pointer exceptions;
- The syntax of Kotlin is focused on reducing verbosity, with rough estimates suggesting a 40% reduction in Lines of Code (LOCs) for equivalent code (Schwermer, 2018). This substantial decrease in verbosity may significantly enhance the maintainability and understandability of codebases developed in Kotlin.

[1] https://developer.android.com/kotlin

2.2. Maintainability metrics

Maintainability, as defined in the literature, is the ease with which a software system or codebase can be modified to correct faults, enhance performance, or adapt to changes in its environment (IEEE, 1990). Maintainability is a critical factor in the economic success of software products. Numerous studies over the years have proposed and organized software metrics to predict or assess the maintainability of software projects. A recent systematic literature review identified 174 different software maintainability metrics in the software engineering literature (Ardito et al., 2020b).

Various tools have been presented in the literature for automatically computing maintainability metrics for software code written in different languages (Ardito et al., 2020b). However, due to Kotlin's relative novelty, there are limited tools available for computing traditional maintainability metrics for Kotlin code.

2.3. Related work

Recent literature has explored the non-functional qualities of software developed with Kotlin, comparing these to traditional Android applications written in Java.

Góis Mateus and Martinez (2019) analysed three different datasets of Android open-source applications (F-Droid, Android-TimeMachine, and Androzoo) in 2019. They found Kotlin code in just over 11% of the projects, with 33.61% of these applications being entirely written in Kotlin. The study noted an increase in the amount of Kotlin code during the evolution of the applications for 63% of the cases. It also investigated the prevalence of code smells, revealing that the studied object-oriented code smells were more common in Kotlin than in Java applications. However, the quantity of entities affected by smells was higher for Java. The introduction of Kotlin positively impacted code smell reduction in at least 50% of the apps.

Andrä et al. (2020) analysed tools for computing maintainability metrics for Kotlin-based Android Applications. In their 2020 study, they concluded that most tools available at the time offered limited support for Kotlin. This lack of support was seen as a hindrance to comparisons against Java codebases and underscored the need for the development of appropriate tools to compute such metrics for Kotlin.

Peters et al. (2021) evaluated the impact of transitioning to Kotlin on the runtime efficiency of Android apps. They reported a significant positive impact on CPU usage, memory usage, and render duration of frames, though the effect size was negligible. The study also found that most Android applications either fully migrated to Kotlin (with over 90% Kotlin code) or contained low portions of Kotlin code (less than 10%).

Mohsen et al. (2021) introduced KotlinDetector, a tool for investigating the security and privacy implications of Android application packages (APKs) that include Kotlin code. This tool performs heuristic pattern scanning and invocation tracing.

3. Research method

This section details the design, goal, research questions, and procedure adopted for the study.

3.1. Research questions

Our study is structured to answer four distinct research questions. RQ1 to RQ3 aim to update a previous study from 2019 (Coppola et al., 2019); RQ4 seeks to provide insights about maintainability metrics computed on Android applications written in Kotlin, a gap identified in current literature.

- **RQ1 - Diffusion**: How widespread is the adoption of Kotlin in Android applications within open-source repositories?

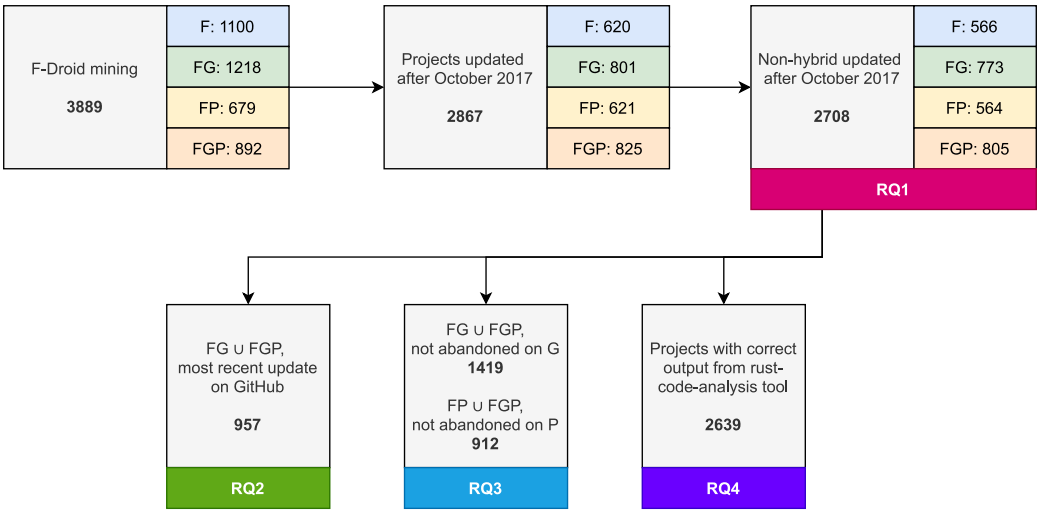

Fig. 1. Filtering steps and project sets used to answer the RQs of the study. The considered project groups are the following: F = hosted on F-Droid only; FG = hosted on F-Droid and GitHub only; FP = hosted on F-Droid and PlayStore only; FGP = hosted on F-Droid, GitHub, and PlayStore.

- **RQ2 - Evolution**: At what rate are open-source Android applications transitioning from Java to Kotlin?
- **RQ3 - Popularity**: How does the use of Kotlin affect the popularity of Android open-source applications among users and within the developer community?
- **RQ4 - Maintainability**: What impact does Kotlin have on the code maintainability of Android open-source applications?

3.2. Selection of software objects

This section outlines the methodology employed to answer the aforementioned research questions and describes the creation of our experimental dataset.

The initial step involved mining the complete corpus of projects and associated information from the F-Droid repository of open-source applications. F-Droid was chosen for several reasons: (i) its extensive use in the literature for mining Android applications (e.g., studies by Zeng et al., 2019, Grano et al., 2017, Fu et al., 2018); (ii) the availability of source code for all hosted applications; (iii) its long-standing history, featuring applications dating back to the original release of the Android framework.

To mine Android projects from F-Droid, we utilized a Selenium Chromedriver script to scrape information from the project webpages and collect the .tar.gz files containing the source code and .apk distributable files. The scraping was conducted as of April 30, 2023, resulting in the collection of 3889 Android projects.

For all packages obtained from F-Droid, we conducted a matching process with corresponding projects on the PlayStore — to gather user popularity metrics and dissemination information — and on GitHub — to collect developer popularity metrics and evolution data.

To determine if an Android project on F-Droid was also available on the PlayStore, we used the unique format of the PlayStore's URI, which ends with *details?id=package_name*. A Selenium Chromedriver script was developed to search for the packages from F-Droid and scrape information from the PlayStore when available. Manual checks were performed for each package due to a common practice of using package names from other projects when releasing applications on the PlayStore, leading to inconsistencies across repositories.

To identify if an F-Droid project had a corresponding repository on GitHub, we developed a Selenium script to locate projects declaring package names in the Android manifest file that matched those from F-Droid. Manual inspections were necessary to correctly identify the appropriate GitHub repository in cases of duplicates or forked projects sharing the same package name.

Different sets of projects were thus defined based on their presence in the considered stores (Fig. 1). As of the end of April 2023, when our final measurements were collected, we had mined 3889 projects from the F-Droid platform. Among these projects open-source (OS), 1571 (40.4%, FP ∪ FGP in figure) were also released on the Google Play Store, and 2110 (54.2%, FG ∪ FGP in figure) were published as GitHub repositories. A total of 892 (22.9%, FGP in figure) apps were available on all three repositories. Focusing on the diffusion of Kotlin, we restricted our analysis to projects updated on F-Droid or GitHub after October 2017, i.e., since Kotlin's official support by the Android Studio IDE. Of the projects, 2869 (73.7%) have been updated since October 2017, marking Kotlin's adoption as a first-class programming language for Android applications. Our analysis revealed that many projects featured negligible amounts of Java or Kotlin code. Upon manual inspection, we identified these as hybrid Android applications developed using tools for creating web-optimized mobile applications (e.g., Flutter). Consequently, we established a threshold of at least 20 lines of code in Java or Kotlin for a project to be considered native, resulting in 2708 non-hybrid projects in total. We defined a threshold of 20 lines because no such threshold has been previously defined in the literature. The threshold was slightly higher than the typical number of lines of code in default classes appearing in hybrid projects, to reduce the possible number of false positives excluded from the sample. We verified manually that all the projects with a number of LOCs below the threshold were hybrid or empty. By sampling the distribution of projects over the threshold, we were not able to identify any false negatives.

By comparing the last updates across the three repositories, we identified projects that were not concurrently updated across the platforms. We set a 45 day threshold to categorize a project as abandoned on a given repository. This concept of abandonment helps avoid inconsistent results from comparative analyses of metrics computed on unaligned versions of the same project on different repositories. No exact threshold has been provided in the literature to define a project as abandoned on a versioning platform like GitHub. We based our threshold on the works by Ait et al. who empirically verified that GitHub projects have on average a typical life of 3 months without commits before being considerable as dead projects (Ait et al., 2022). The risk of mining abandoned projects is also underlined by Kalliamvakou et al. who found that only 13% of the projects on GitHub have an update in the last 30 days (Kalliamvakou et al., 2014). Therefore, to cope with the possible high risk of considering abandoned projects and to reduce the possible number of false negatives, we halved the threshold proposed by Ait et al. for our purposes.

All information about the collected projects, along with the computed metrics, is available in an online repository.[2]

3.3. Analysis procedure

In Fig. 1, we outline the characteristics of the project sets used in each step of our study and correlate them with the research questions they help to address. Details about each filtering step and all performed analyses are described below.

3.3.1. Diffusion analysis (RQ1)

To address RQ1, we focused on non-hybrid projects updated after October 2017 (2708 projects) to compute diffusion metrics. We used the number of Kotlin Relative Lines of Code (KRL) and Kotlin Relative Files (KRF) as diffusion metrics. These ratios were computed over the total number of implementation lines and files, excluding documentation and configuration files. The most recently updated app package was considered, whether it was the tarball released on F-Droid or the last tagged release on the GitHub repository (if available). These two code metrics were calculated using the cloc tool.[3]

To facilitate understanding, we introduce a derived measure, Kotlin *Presence*, an ordinal variable with four levels:

- *No Kotlin*: Projects containing only Java code.
- *Kotlin* < 50%: Projects with a minority of Kotlin code, i.e., less than 50% in terms of LOCs.
- *Kotlin* > 50%: Projects with a majority of Kotlin code, i.e., more than 50% but less than 100%.
- *Only Kotlin*: Projects featuring only Kotlin code.

We then computed the ratio of projects in each category relative to the starting set of projects.

3.3.2. Evolution analysis (RQ2)

To address RQ2, we focused on the set of projects that were active on GitHub with the last updates after October 2017, encompassing 967 projects. For analysing the evolution of these projects, we took monthly snapshots from October 2017 to April 2023, considering the last commit of each month. Each snapshot was analysed using the cloc tool, as was done for RQ1. We then categorized the projects into different Kotlin adoption groups (as defined for the previous research question) and reported the ratio of projects in each group relative to the total number of projects at each monthly snapshot.

3.3.3. Collection of popularity metrics (RQ3)

To address RQ3, we analysed two distinct sets of projects: (i) projects present on the PlayStore (regardless of their GitHub presence) that were not abandoned on the PlayStore (totalling 912 projects). For these, we used the average rating (i.e., *Stars*) as a popularity metric; (ii) projects available on GitHub (regardless of their PlayStore presence) that were active on GitHub. For these, the number of stars and the number of watching accounts were considered as popularity metrics. Both metrics for GitHub projects are absolute figures and thus greatly influenced by the project's lifespan. Consequently, they were normalized by dividing the absolute numbers by the project's lifespan in months.

3.3.4. Computation of maintainability metrics (RQ4)

To answer RQ4, we focused on the set of projects updated after October 2017, considering the most recent repository update between the tarball hosted on F-Droid and the last commit on GitHub (if available).

For this study, we utilized rust-code-analysis,[4] a Rust library that analyses and extracts information from source code. This library is based on the TreeSitter parsing library (Ardito et al., 2020a). Rust-code-analysis was chosen for its reliability, having been used by Mozilla to assess the maintainability of their codebases and the existence of a branch of the tool capable of analysing Kotlin code.[5] The rca tool provides a single value for each metric per project; the metrics are computed for each *space*[6] in the project, and then summed over all the spaces. A space is defined in rca as the smallest source code structure that includes at least a function or a closure, and is a language-agnostic feature that can be extracted for all languages supported by the tool.

From the metrics computed by rca, we selected six that are most closely related to code maintainability and understandability. Definitions of these metrics are provided below.

- **CC:** McCabe's definition of cyclomatic complexity: it calculates the code complexity by examining the control flow of a program. It is measured as the number of linearly independent paths through a piece of code (Ebert et al., 2016).
  The cyclomatic number of a graph G with n vertices, e edges, and p connected components is computed with the following formula and is equal (for a strongly connected graph) to the maximum number of linearly independent circuits (McCabe, 1976):

  $$CC = e - n + p$$

  McCabe's CC for a program is lower-bounded by 0 and is not upper-bounded. Higher values indicate a higher complexity of the code. Average values for CC are reportedly between 1 and 2 (Vasa and Schneider, 2003).
- **Halstead Difficulty:** The Halstead suite is a set of seven statically computed metrics, all based on the number of distinct operators (n1) and operands (n2) and the total number of operators (N1) and operands (N2). The suite provides a series of information, such as the effort required to maintain the analysed code, the size in bits to store the program, the difficulty of understanding the code, an estimate of the number of bugs present in the codebase, and an estimate of the time needed to implement the software (Dorofeev and Wenger, 2019; Hariprasad et al., 2017).
  Halstead Difficulty (or error-proneness) represents the difficulty in developing a specific errorless piece of code, and is computed with the following formula:

  $$D = \frac{n1}{2} * \frac{n2}{N2}$$

  Halstead Difficulty is not upper-bounded and values are typically in the 10–100 range even for small-sized software artefacts (Govil, 2020).
- **Halstead Effort:** The effort required to implement or understand a program. The measure is directly proportional to the difficulty and the volume (measured as $V = N \log 2(n)$) of the program, as computed with the following formula:

  $$E = D * V$$

  In the Volume formula, n is defined as the program vocabulary (sum of the number of distinct operators and the number of

[2] http://softeng.polito.it/coppola/kotlin_2023_subject_data.csv
[3] https://github.com/AlDanial/cloc
[4] Please note that the name *Rust Code Analysis* indicates that the tool is written in Rust and is not exclusively for measuring Rust code.
[5] https://github.com/mozilla/rust-code-analysis
[6] Additional documentation about spaces is available on the GitHub repository of rca: https://github.com/mozilla/rust-code-analysis/blob/master/src/spaces.rs#L27-L50.

distinct operands in the program) and N is defined as the program length (sum of the total number of operators and the total number of operands).
Halstead Effort is not upper-bounded and values are typically above 10e4 even for small-sized software artefacts (Govil, 2020). Given the large absolute size and variation of the Halstead Effort metric, in this study, we will consider its log10 for our analyses.

- **MI:** Maintainability Index, a derived metric to measure the easiness of maintaining a code base. Several variations are available for MI. In the context of this empirical experiment we have used the MI formula implemented in the Visual Studio IDE:

$$MI = \max(0, 171 - 5.2\ln(\overline{V}) - 0.23\overline{V}(g') - 16.2\ln(\overline{LOC}) * 100/171)$$

where $\overline{V}$ is the average Halstead Volume per module, $\overline{V}(g')$ is the average cyclomatic complexity per module, and $\overline{LOC}$ is the average lines of code per module. In the case of Java and Kotlin code, a function or method is considered an individual module.
The Visual Studio implementation of the Maintainability Index is lower-bounded by 0, and higher values signal higher code maintainability (low maintainability for $MI \leq 10$, medium maintainability for $MI < 10 < 20$, high maintainability for $MI \geq 20$).

- **WMC:** The Weighted Method per Class (WMC) metric is defined as the sum of the complexity of a class' local methods. The WMC metric is intended to measure the combined complexity of a class' local methods (Li, 1998). In rca, McCabe's Cyclomatic Complexity (CC) is used as the base complexity metric for WMC.

$$\text{WMC} = \sum_{i=1}^{n} c_i$$

where $c_i, \ldots, c_n$ are the complexities of the methods (Chidamber and Kemerer, 1994).

- **Cognitive Complexity:** It is a measure of how difficult a unit of code is to intuitively understand, by examining the cognitive weights of basic software control structures (Shao and Wang, 2003). The rust-code-analysis tool implements the definition of Cognitive complexity provided by SonarSource.[7] Typical values of the Cognitive metric are lower than 10, with higher values indicating a lower comprehension of the source code by developers and maintainers.

3.3.5. Statistical analysis (RQ3, RQ4)

As far as RQ3 and RQ4, we formulate a generic null hypothesis concerning the effect of Kotlin's presence in a project on the output — popularity and maintainability — variables (Y).

H_0 : there is no difference in the mean level of the variable Y between the different levels of Kotlin presence.

To test the hypothesis we perform a linear regression and test the significance of the coefficients using an ANOVA test. The regression equation uses the *No Kotlin* level as a reference and has three dummy or indicator variables for the three remaining Kotlin levels:

$$Y = c_0 + c_{minK} \cdot x_{minK} + c_{majK} \cdot x_{majK} + c_{onlyK} \cdot x_{onlyK}$$

Where:

- c_0 is the so-called intercept, represents the mean value for projects with Presence = *No Kotlin*
- x_{lk} are the indicator variables for the distinct levels of Kotlin presence (lk),

$$x_{lk} = \begin{cases} 1 & \text{if } Presence = lk \\ 0 & \text{otherwise} \end{cases}$$

- c_{lk} are the coefficients of the indicator variables.

The ANOVA test checks the null hypothesis that the coefficients are 0. Thus if the p-value for a given coefficient is $< \alpha$ we reject the null hypothesis and assume that it is non-zero, i.e. the effect of the indicator variables is statistically significant.

The decision on whether to reject the null hypotheses will be taken using the usual $\alpha = 0.05$ level, i.e. we accept a 5% risk of type I error. Since we perform multiple tests ($n = 9$, one for each of the three metrics to answer RQ3 and one for each of the six metrics to answer RQ4) on the same independent variable — Presence — to compensate for the family-wise error rate, we apply the Bonferroni correction, i.e. we will take our decisions comparing then p-values to $\alpha_B = \alpha/n = 5.5 \cdot 10^{-3}$.

4. Results

This section reports the results of our experiment, divided by Research Question. The repository for the analysis is provided as a replication package on GitHub.[8]

4.1. Diffusion (RQ1)

In Table 1, we present the results of measuring the diffusion metrics defined to address RQ1. We divided the projects into two categories: based on the marketplaces where they were released and based on the platform (either F-Droid or GitHub) where the source code was most recently updated. The table enumerates the total number of applications per group, projects that feature only Java code (no Kotlin), projects with a minority (less than 50%) of Kotlin in terms of LOCs, projects with a majority (more than 50%, less than 100%) of Kotlin code, and projects exclusively using Kotlin. We also provide the average LOCs of Kotlin in the projects, but this average is calculated only for projects that include Kotlin code. The distribution of projects based on the level of Kotlin presence is visualized in bar plots in Fig. 2 divided by the hosting repository (F: hosted on F-Droid only; FG: hosted on F-Droid and GitHub; FP: hosted on F-Droid and PlayStore; FPG: hosted on all three repositories) and Fig. 3 (divided by the most recently updated source code).

Upon analysing the total number of projects, it is evident that 1023 out of 2708 (37.8%) featured Kotlin code. Of these, 551 out of 2708 (20.3%) were entirely written in Kotlin. The proportion of projects with Kotlin presence increases if the most recent update was on F-Droid (38.9%) rather than on GitHub (35.8%).

In the rightmost two columns of Table 1, we present the percentage of Kotlin LOCs and Kotlin files in projects that include Kotlin. These data suggest that when Kotlin is used, it typically comprises the majority of the code (75.90% LOCs, 77.80% files), indicating a preference among developers for Kotlin over Java when both languages are used in the same project.

In Table 2, we report statistics on project abandonment for the non-hybrid projects updated after October 2017. A comparison of the most recent update dates across multiple platforms reveals that many projects on F-Droid and the PlayStore are not kept synchronized with their GitHub counterparts (nearly half of the projects on F-Droid and GitHub, and on all three repositories). Conversely, fewer projects are abandoned on GitHub while being updated on other platforms. This trend could be anticipated, as it is unlikely that updates on the stores follow every tagged release on GitHub. Notably, nearly 16% of the projects available on all three platforms were not updated on GitHub, possibly due to the projects becoming closed-source or migrating to other source code hosting platforms.

[7] https://www.sonarsource.com/docs/CognitiveComplexity.pdf

[8] https://github.com/riccardocoppola/kotlin_rmd_study

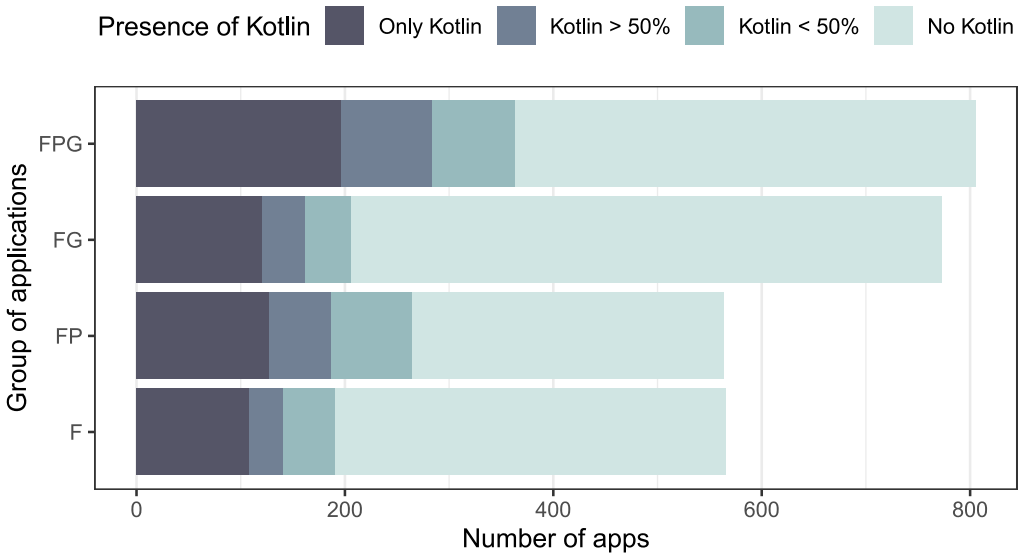

Fig. 2. Statistics about the diffusion of Kotlin per platform where apps are released (F = F-Droid; P = PlayStore; G = GitHub).

Table 1
Diffusion (RQ1) metrics on non-hybrid projects updated after October 2017 (F: F-Droid; P: PlayStore; G: GitHub)

						Avg. on Projects w/Kotlin	
	Apps	No Kotlin	Kotlin min.	Kotlin maj.	Only Kotlin	Kotlin LOCs	Kotlin files
F	566	376 (66.4%)	50 (8.83%)	32 (5.65%)	108 (19.1%)	75.10%	77.00%
F, P	564	300 (53.2%)	78 (13.8%)	59 (10.5%)	127 (22.5%)	72.10%	74.20%
F, G	773	567 (73.3%)	44 (5.69%)	42 (5.43%)	120 (15.5%)	79.20%	80.90%
F, P, G	805	442 (54.9%)	80 (9.94%)	87 (10.8%)	196 (24.3%)	77.00%	79.10%
Last update: F	1751	1070 (61.1%)	185 (10.6%)	140 (7.99%)	356 (20.3%)	74.10%	76.10%
Last update: G	957	615 (64.3%)	67 (7.00%)	80 (8.36%)	195 (20.4%)	79.40%	81.20%
Whole set	2708	1685 (62.2%)	252 (9.30%)	220 (8.12%)	551 (20.3%)	75.90%	77.80%

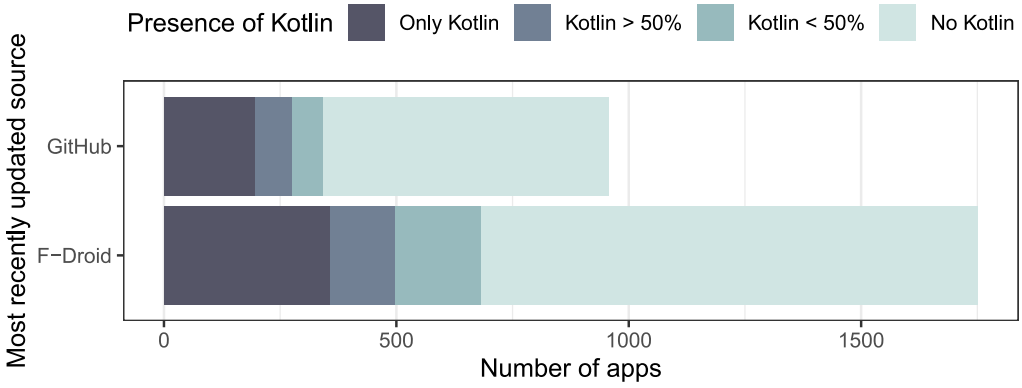

Fig. 3. Statistics about the diffusion of Kotlin per platform where the source code was lastly updated.

Table 2
Statistics about project abandonment (update on the store less recent than 45 days before the overall last update) for non-hybrid projects updated after October 2017. (F: F-Droid; P: PlayStore; G: GitHub)

	Abandoned on		
Present on ↓	F-Droid	PlayStore	GitHub
F, P	122 (21.6%)	122 (21.6%)	–
F, G	375 (48.5%)	–	76 (9.8%)
F, P, G	380 (47.2%)	390 (48.4%)	127 (15.8%)

4.2. Evolution (RQ2)

We evaluated the history of Kotlin adoption in projects that had their last update after October 2017 and were most recently updated on the GitHub platform, encompassing 957 projects. These projects were analysed through monthly snapshots to identify the trend of Kotlin code adoption.

Fig. 4 illustrates the trend of Kotlin adoption by month since October 2017, within the set of 957 considered projects. For each month, projects that were no longer updated in the previous year were excluded from the analysis. It is observed that the number of projects incorporating Kotlin has steadily increased over the surveyed period: from 6.8% (39 out of 574) to 48% (241 out of 501). Concurrently, there

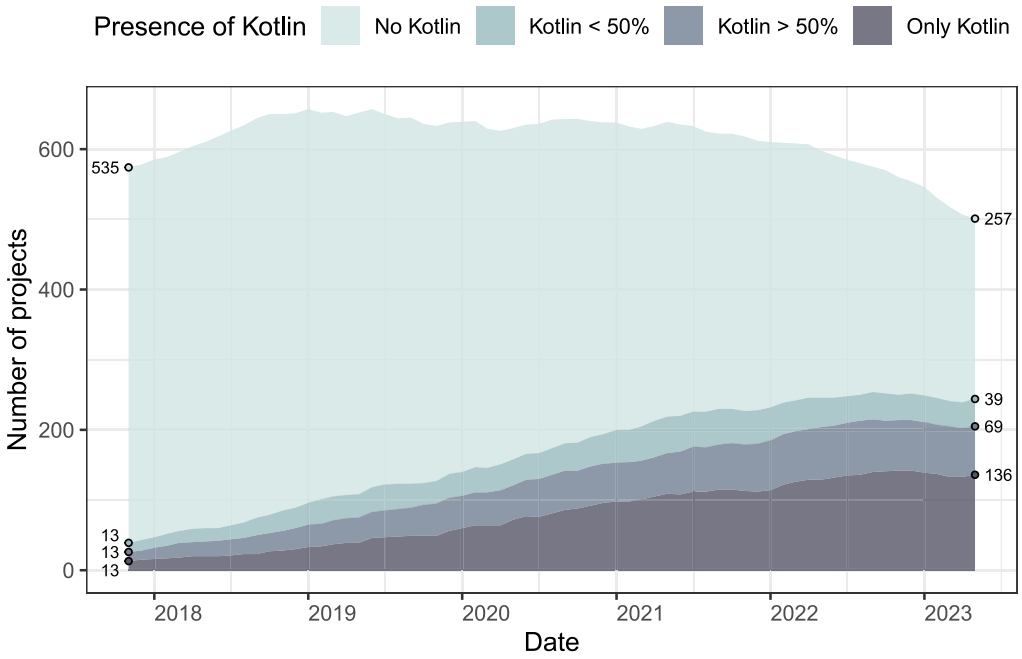

Fig. 4. Projects that were updated in the last 12 months, with only Kotlin, with Kotlin minority, with Kotlin majority, and with only Java, per month from October 2017 to May 2023.

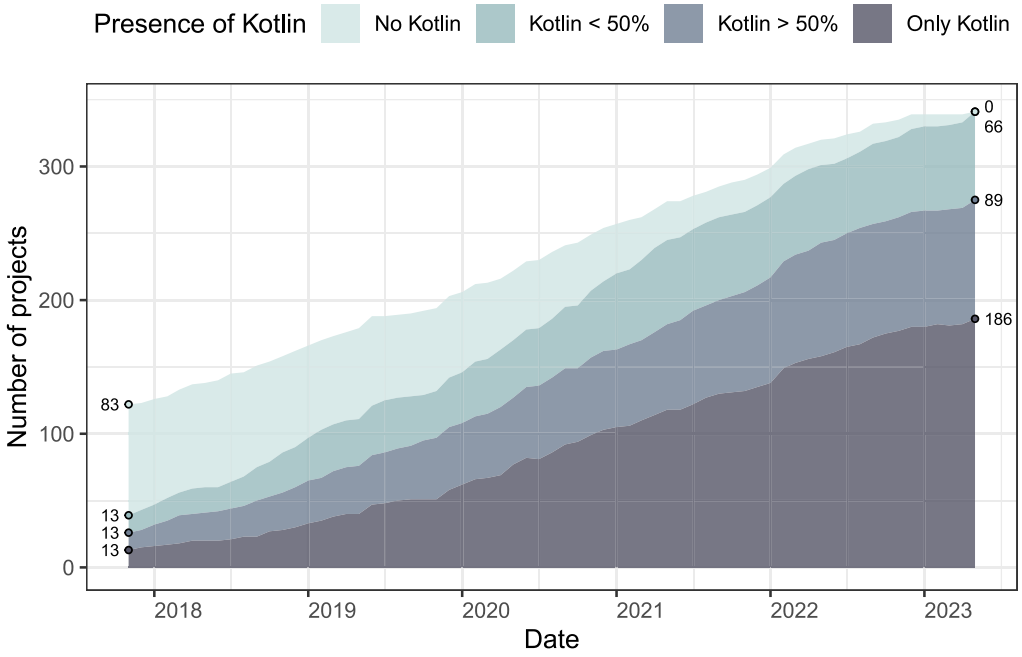

Fig. 5. Evolution of Kotlin relative LOCs on the lifespan of projects featuring Kotlin and with a most recent update on GitHub.

is a noticeable decrease in the number of projects featuring only Java, with a reduction of 52.2% since October 2017, and 13.4% in the first four months of 2023 alone.

Fig. 5 narrows the focus to the 342 projects that included Kotlin and were most recently updated on GitHub. In this analysis, we also included projects that had not been updated in the last 12 months. The figure reveals a clear trend of increasing proportions of Kotlin code within Android projects.

4.3. Popularity (RQ3)

To determine whether the adoption of Kotlin had any impact on how end-users and other developers perceive the projects, we investigated the correlations between the presence of Kotlin code and popularity metrics on both the PlayStore market and the GitHub platform.

We gathered the ratings from the PlayStore for all projects available on this market and not abandoned, totalling 912 projects. Additionally,

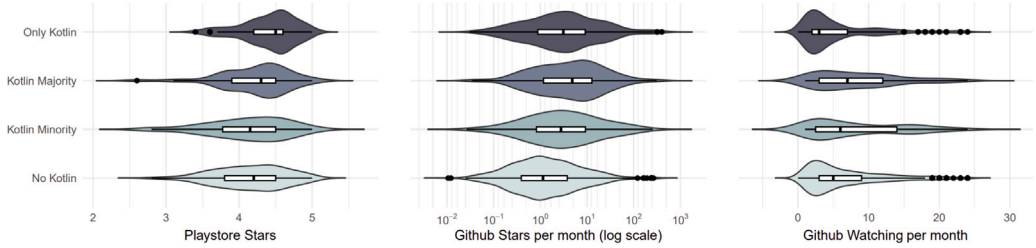

Fig. 6. Violin plots of distribution of popularity metrics (RQ3), higher values mean better popularity.

Table 3
Mean and standard deviation for RQ3 metrics.

	No Kotlin		Kotlin minority		Kotlin majority		Only Kotlin	
	Mean	SD	Mean	SD	mean	SD	Mean	SD
PlayStore stars	4.14	0.49	4.11	0.55	4.22	0.47	4.40	0.37
GitHub stars per month	6.63	24.34	13.90	33.05	14.17	35.91	15.47	46.35
GitHub watching per month	0.49	1.20	0.62	1.01	0.48	0.59	0.65	1.91

Table 4
Results (estimates and p-values) of the statistical analysis for RQ3 metrics (p-values in bold if smaller than $\alpha_B = \alpha/n = 5.5 \cdot 10^{-3}$)

	PS Stars		GH Stars		GH Watching	
	est.	p-value	est.	p-value	est.	p-value
Kotlin minority	−0.03	0.719	7.27	0.026	0.13	0.356
Kotlin majority	0.08	0.300	7.54	0.012	−0.01	0.907
Only Kotlin	0.26	$\mathbf{1.35 \cdot 10^{-4}}$	8.84	$\mathbf{3.51 \cdot 10^{-5}}$	0.15	0.081

we collected data on the number of stars and watching users on GitHub for all projects present on this platform and not abandoned, totalling 1419 projects. These metrics were then normalized with respect to the project's age in months. Fig. 6 displays the distribution of these three metrics across the respective project sets. Table 3 shows the mean and standard deviation for the metrics, while Table 4 presents the results of the ANOVA test on the linear models used to assess the effect of Kotlin presence on the popularity metrics.

We observed a significant positive effect of having a project fully developed in Kotlin on the average number of stars on the PlayStore platform (average 4.4 mean rating, compared to a 4.14 mean rating, with a p-value of 1.35e−04). There was no statistically significant difference between the groups "Kotlin minority" and "Kotlin majority" compared to the reference group "Only Java".

In terms of GitHub stars — normalized by the project's lifespan in months — per project, the estimate for the group "Only Kotlin" is 8.84, with a p-value of 3.35e−05. This indicates a statistically significant positive difference in normalized GitHub stars for projects fully developed in Kotlin. A positive difference was also observed for projects with a Kotlin minority and majority, but these differences were not statistically significant.

Regarding GitHub Watching — normalized by the project's lifespan in months —per project, there was no statistically significant difference between any group and the reference "Only Java" group. However, we noticed a slight increase in the GitHub Watching metric for projects featuring Kotlin compared to those with full Java implementations.

4.4. Maintainability (RQ4)

The metrics to address RQ4 were collected for all projects updated after October 2017. Following the application of the rust-code-analysis tool, we excluded projects from the set that returned null results for any of the 6 computed metrics. This exclusion was due to either computation errors in the tool or the inability to analyse the complete code tree in very large-sized projects. After this additional filtering phase, 2639 projects were considered for analysis. Fig. 7 depicts the distribution of the six metrics, Table 5 reports the mean and standard deviation, and Table 6 presents the results of the ANOVA analysis of linear models used to evaluate the impact of Kotlin presence on maintainability metrics.

For the Weighted Methods per Class (WMC) metric, the "Kotlin Majority" and "Only Kotlin" groups showed a statistically significant difference compared to the reference group "No Kotlin". This suggests that applications using Kotlin tend to have fewer weighted methods per class, with an average decrease from 17.03 to 7.47 when only Kotlin is utilized.

In terms of the Cyclomatic Complexity (CC) metric, the difference between the "Kotlin Majority" and "Only Kotlin" groups, compared to the reference group "No Kotlin", was statistically significant, with an average decrease from 1.88 to 1.74 and 1.70, respectively.

Concerning the log10 of the Halstead Effort, a statistically significant difference was observed between the "Kotlin Majority" and "Only Kotlin" groups against the reference group "Only Java", with an average decrease from 5.43 to 4.79 and 4.42, respectively.

A similar significant difference was noted for the Halstead Difficulty, with a decrease from 29.33 to 11.49 moving from "No Kotlin" to "Only Kotlin".

Regarding the Maintainability Index (MI) metric, all groups exhibited a statistically significant positive effect compared to the "No Kotlin" reference group. For projects exclusively using Kotlin, an increase from 30.64 to 38.14 was observed.

Finally, for the Cognitive Complexity metric, a statistically significant decrease was noted only for the "Kotlin Majority" (reduction of 0.44) and "Only Kotlin" (reduction of 0.56) groups, compared to the average value of 1.42 for the "No Kotlin" reference group.

5. Discussion

On the basis of the finding reported above, we summarizes the main findings of our study and frame them in the context of related work. We also discuss potential threats to the validity of this study.

5.1. Summary of findings

Our primary objective was to evaluate the extent of migration from Java to Kotlin in the Android ecosystem and to quantify the impact of such migration on code maintainability.

Over ten years since Kotlin's initial release and more than five years since Google adopted it as a first-class language for Android applications, Kotlin has established itself as a mature and reliable language.

Answer to RQ1: 38% of analysed apps contain Kotlin code.

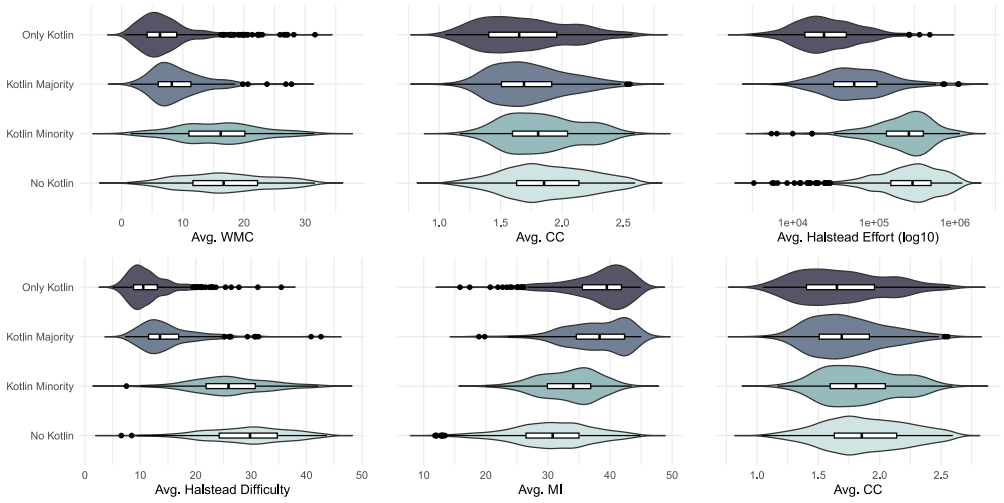

Fig. 7. Violin plots of distribution of maintainability metrics (RQ4), lower value indicate better maintainability except for MI.

Table 5
Mean and standard deviation for RQ4 metrics.

	No Kotlin		Kotlin minority		Kotlin majority		Only Kotlin	
	Mean	SD	Mean	SD	Mean	SD	Mean	SD
WMC	17.03	7.1	15.81	6.66	9.19	4.36	7.47	4.98
CC	1.88	0.34	1.84	0.31	1.74	0.30	1.70	0.36
$\log_{10}$(Halstead E)	5.43	0.39	5.35	0.38	4.79	0.42	4.42	0.38
Halstead D	29.33	7.24	26.3	6.74	14.82	**5.26**	11.49	3.94
MI	30.64	6.47	33.33	4.69	37.88	5.03	38.14	5.23
Cognitive	1.42	0.60	1.35	0.56	0.98	0.48	0.86	0.54

Table 6
Results (estimates and p-values) of the statistical analysis for RQ4 metrics (p-values in bold if smaller than $\alpha_B = \alpha/n = 5.5 \cdot 10^{-3}$)

	WMC		CC		Halstead Effort	
	est.	p-value	est.	p-value	est.	p-value
Kotlin Minority	−1.22	0.011	−0.04	0.131	−0.08	0.006
Kotlin Majority	−7.84	$\mathbf{1.43 \cdot 10^{-63}}$	−0.15	$\mathbf{2.77 \cdot 10^{-9}}$	−0.64	$\mathbf{2.79 \cdot 10^{-110}}$
Only Kotlin	−9.57	$\mathbf{3.48 \cdot 10^{-157}}$	−0.18	$\mathbf{5.74 \cdot 10^{-24}}$	−1.01	$\mathbf{< 1.0 \cdot 10^{-200}}$

	Halstead Difficulty		MI		Cognitive	
	est.	p-value	est.	p-value	est.	p-value
Kotlin Minority	−3.02	$\mathbf{8.65 \cdot 10^{-11}}$	2.69	$\mathbf{1.66 \cdot 10^{-9}}$	−0.07	0.129
Kotlin Majority	−14.51	$\mathbf{3.09 \cdot 10^{-190}}$	7.24	$\mathbf{1.36 \cdot 10^{-57}}$	−0.44	$\mathbf{2.51 \cdot 10^{-27}}$
Only Kotlin	−17.84	$\mathbf{< 1.0 \cdot 10^{-200}}$	7.5	$\mathbf{2.54 \cdot 10^{-97}}$	−0.56	$\mathbf{14.67 \cdot 10^{-74}}$

A consistent increase was observed in the number of Android projects featuring at least partial Kotlin implementation. Many project repositories that now include Kotlin code started with Java only and were migrated during their lifespan, either through complete codebase rewrite or by using Kotlin to integrate new features. This trend may reflect a widespread perception among developers of the benefits of a language transition. Other influencing factors could include Android development guidelines, which now favour Kotlin over Java, and Kotlin's full interoperability with Java, facilitating the rapid integration of Kotlin-written features into existing Java codebases.

Answer to RQ2: Since 2017, the proportion of active projects using Kotlin increased from 8% to 48%.

Regarding software quality perception, we used three quantitative measures as proxies: the average rating (number of stars) on the PlayStore and the average number of stars and watching users on GitHub. These metrics represent different stakeholder perspectives for Android projects: PlayStore stars reflect the final users' quality perception, while GitHub measures indicate interest from open-source developers and contributors. We observed a significant positive difference for projects developed entirely in Kotlin compared to pure Java projects in terms of PlayStore stars and GitHub stars. Although potentially biased by the size and variability of the applications considered, the PlayStore stars result suggests that preferring Kotlin over Java positively impacts the quality as perceived by end-users. This finding can guide developers, evidencing the beneficial effects of transitioning an existing Java project to Kotlin or adopting Kotlin from the start of a new project. The GitHub stars result was anticipated, as it is reasonable that projects using newer languages attract more interest from fellow developers.

Answer to RQ3: End-users on the PlayStore rate projects using Kotlin a quarter of a star higher; on GitHub, pure Kotlin projects receive 8 additional stars per month w.r.t. pure Java ones.

Maintainability is a highly touted aspect of the Kotlin language, with its primary appeal being enhanced code understandability. The results for the six maintainability metrics considered consistently demonstrate significantly higher maintainability for Android applications that include Kotlin code (either exclusively or mixed) compared to pure Java applications. The differences were significant for all metrics when Kotlin predominated over Java, with more pronounced effects observed when Kotlin was the sole language used. The reduction in code complexity when transitioning from Java to Kotlin is highlighted by the decrease in average McCabe's Cyclomatic Complexity (−9.6%) and particularly in Weighted Methods per Class (−56%), indicating

a substantial simplification in code structure and organization with Kotlin.

The size of the code, measured by the Halstead Effort, and the relative difficulty in maintaining an error-free version, indicated by the Halstead Difficulty, both decreased significantly by −18.6% and −60.8%, respectively. However, caution should be exercised when interpreting results involving Halstead metrics, given their known limitations in adapting to newer programming languages and software paradigms. The Halstead metrics we considered are derived from the number of operators and operands in the program, so their reduction can be seen as evidence of reduced verbosity and, thus, increased understandability (Shepperd, 1992).

Acknowledging the debated limitations of Halstead metrics, we included the more contemporary Maintainability Index (MI) and Cognitive Complexity metrics in our study. The Maintainability Index showed a notable increase when comparing Java-only versus Kotlin-only projects (+24.5%). It is important to note that, based on the Visual Studio implementation of the Maintainability Index we used, all project groups averaged high maintainability (values above 20). A −39.4% decrease in the Cognitive Complexity metric aligns with the differences measured for other complexity measures (WMC and CC) and further suggests greater readability of Kotlin code. These findings offer valuable insights to the Android developer community, highlighting tangible benefits in terms of code understandability, conciseness, and readability compared to traditional Java code.

It is worth underlining that five out of six metrics that were considered in this study were not influenced by the grammar and syntax rules of the involved languages, given that the CC and Cognitive metrics are only influenced by the control flow graph that can be defined for the analysed source code, WMC is influenced by the number of methods, and Halstead Effort and Difficulty are based on the number of operators and operands. The specific language features of Kotlin have no direct impact on these items, therefore all the changes in this metric are assumed to reflect changes in the way developers utilize the different languages and to the enforcement of Kotlin best practices. The Maintainability Index (MI) metric instead does depend on the number of LOCs of the analysed code artefact, therefore the syntax and grammar aspects of Kotlin may have a significant influence on the number of lines of code that are produced. We however deem a comparison of MI as fair between the two languages, since our final objective is an evaluation of the maintainability of codebases in Java and Kotlin, and the number of LOCs to implement an application is an aspect that directly influences its maintainability.

Answer to RQ4: Consistent improvement of all maintainability related metrics: McCabe CC ↓ 9.6%, WMC ↓ 56%, Halstead Effort ↓ 18.6%, Halstead Difficulty ↓ 60.6%, Cognitive Complexity ↓ 39.4%, Maintainability Index ↑ 24.5%.

5.2. Comparison with related work

Góis Mateus and Martinez (2019) in 2019 assessed the adoption of the Kotlin language and the evolution of applications utilizing it. Compared to their findings, our study identified both a higher ratio of applications containing Kotlin (1023 out of 2708 vs 244 out of 2167) and a greater number of applications entirely written in Kotlin (551 vs 82), effectively demonstrating how the Android developer community has transitioned to Kotlin over the last four years. In terms of code quality, Mateus and Martinez evaluated the presence of 10 Code Smells as a proxy for app quality, whereas we employed a set of six theoretical metrics. Their findings indicate that while Kotlin is not free from flaws, applications initially written in Java showed improved quality after introducing Kotlin.

Peters et al. (2021), in a more recent study, confirmed the positive trend of adopting Kotlin for Android app development. Aligning with Mateus and Martinez, they observed that the highest concentration of apps occurs when the percentage of Kotlin LOCs is between 90 and 100% or 0 and 10%. From their sample of 7972 applications, they noted an increase in apps with more than 90% Kotlin LOCs from 127 in 2019 to 659 in 2023, and those with less than 10% Kotlin rose from 44 in 2019 to 123 in 2023. This indicates that while the number of apps using Kotlin increased, their distribution based on the percentage of LOCs remained consistent. analysing our trends for the ratio of Kotlin LOCs, we also observed that when Kotlin is adopted, most projects are nearly entirely written in Kotlin (in our set of 1053 projects with Kotlin, 659–64.4% —feature more than 90% Kotlin LOCs). Our results thus align with Peters et al.'s findings, suggesting that developers transitioning to Kotlin eventually prefer it over Java.

Finally, the results of RQ4 address the open question posed by Ardito et al. (2020c) in their 2020 study, which stated, "*there is no evidence that the use of Kotlin, as a substitute for Java, either enhances or lessens software maintainability*". Our study provides statistically significant evidence that using Kotlin as a native programming language leads to better maintainability than Java.

5.3. Threats to validity

We discuss the threats to the validity of this study as per the taxonomy presented by Wohlin et al. (2012).

Threats to external validity concern the generalizability of our study's results to broader populations beyond the sampled applications. The dataset's size may not encompass the full diversity of Android applications. Enhancing the study by considering other repositories of open-source Android applications cited in related literature (e.g., AndroidTimemachine or AndroZoo) or by accounting for the inherent variability of Android applications (e.g., distribution across different application categories in various stores) could improve this aspect. Extending the study to include hybrid applications and other domains beyond Android would also increase external validity, providing a more comprehensive understanding of Kotlin vs. Java maintainability. A final threat to external validity is that our sample is limited to open-source applications. Thus, we cannot assert that the results extend to closed-source, proprietary apps, although a similar trend might be present, albeit potentially slower.

Threats to construct validity relate to the accuracy and suitability of the measurements used to evaluate the applications' characteristics. The project size and diffusion were based on LOCs and files without considering the code's internal organization or the presence of languages other than Java or Kotlin, potentially introducing bias when assessing the relative importance of Java vs. Kotlin in an Android open-source project. The selected popularity metrics were those immediately available on the hosting platforms, but other measures might offer more precise insights into user-perceived application quality (e.g., user feedback, review comments, or developer discussions). For stronger construct validity, comparing the same set of applications in both Kotlin and Java would be beneficial, reducing confounding variables. However, this extension would require a complete rewrite of experimental subjects, as there are no known open-source Android applications written in both languages covering identical features. Future work might evaluate files translated from Java to Kotlin in projects that transitioned during their lifespan.

Threats to internal validity pertain to potential biases and errors that could affect the study's outcomes. A first threat to the internal validity is related to the selection and classification of the projects in the sample to hybrid projects (by using a 20 LOCs threshold) and to abandoned projects (by using a threshold of 45 days since last code update). These thresholds have been validated in the present study for the absence of false positives but are not systematically validated and can lead to false negatives in the final set of papers used for our measures (i.e., hybrid projects with more than LOCs and/or abandoned projects with a commit more recent than 45 days). Future research may take into consideration the utilization of more refined ML-based approaches,

such as the one proposed by Coelho et al. (2020), for a more dependable classification of abandoned projects. A critical aspect is the tool used for measuring maintainability; if the tool has inherent biases or flaws favouring one language, it could compromise internal validity. Nevertheless, the tool is considered reliable, having been utilized in large-scale open-source projects and previous empirical studies (Ardito et al., 2021). Other internal validity threats include possible errors in manual checks during project mining (e.g., identifying the correct manifest file in GitHub projects with multiple manifests). A final threat to the internal validity of the study is the fact that the maintainability of a software project is not only influenced by static properties (i.e., lines of code and size and number of methods) but also by the developers' skill level and expertise, and by the functionalities implemented by the source code. The influence of these aspects cannot be captured by the metrics that are computed in this study.

Threats to conclusion validity are about the accuracy of conclusions drawn from statistical tests. Variance analysis on a linear model was used to check the statistical significance of the coefficients. We did not use non-parametric tests, considering the sample size — 2700 data points — sufficient for the central limit theorem to apply, allowing for result interpretation despite minor deviations from normality. Our decision was based on the conventional 5% significance level, and we applied the Bonferroni correction to mitigate the family-wise error rate.

6. Conclusions

In this paper, we conducted an empirical analysis of the diffusion, evolution, perceived quality, and maintainability of Android applications developed with Kotlin compared to pure Java software projects. The results indicate no negative effects of adopting Kotlin for Android projects. Furthermore, we found that projects predominantly or entirely using Kotlin significantly enhance the maintainability of software artefacts.

The findings have several implications for various stakeholders in the Android ecosystem:

- For **open-source developers** interested in Kotlin, our research offers evidence of a large and growing body of Kotlin-based open-source Android projects. This software corpus has received positive developer feedback on the platform, affirming Kotlin's initial purpose to provide a *better Java for developers*.
- For **companies** releasing Android applications, our study provides empirical evidence that adopting Kotlin in Android projects is cost-effective compared to Java programming, without compromising the quality perceived by end-users.
- For **software engineering researchers**, this work serves as a foundation for comparative analyses of Kotlin and Java, offering a dataset of Android projects with corresponding code-level metrics for future studies.

Compared to our initial 2019 study (Coppola et al., 2019), which examined Kotlin's diffusion among Android projects on F-Droid, we observed an increase from nearly 1/5th to 1/3rd of projects containing Kotlin, among those updated since October 2017. These figures confirm an ongoing transition from Java to Kotlin in Android development. This study not only corroborates previous findings on Kotlin's spread but also presents new evidence of Kotlin's superior maintainability and readability compared to Java. All Android systems will be assimilated: resistance is futile (Frakes, 1996).

Future research could extend the dataset to encompass open-source application repositories from various platforms beyond Android, offering a broader view of Kotlin and Java's maintainability in different development contexts. A more detailed examination of maintainability metrics could also be undertaken, including additional metrics for Kotlin and Java. Further, a more granular, file-by-file analysis during the transition from Java to Kotlin could identify specific maintainability advantages, providing practical guidance for developers aiming to optimize their migration process for improved maintainability.

CRediT authorship contribution statement

Riccardo Coppola: Writing – review & editing, Writing – original draft, Visualization, Validation, Supervision, Software, Resources, Project administration, Methodology, Investigation, Formal analysis, Data curation, Conceptualization. **Tommaso Fulcini:** Writing – review & editing, Investigation, Data curation. **Luca Ardito:** Writing – review & editing, Writing – original draft, Validation, Supervision. **Marco Torchiano:** Writing – review & editing, Writing – original draft, Visualization, Validation, Supervision.

Declaration of competing interest

The authors declare that they have no known competing financial interests or personal relationships that could have appeared to influence the work reported in this paper.

Data availability

All data used is provided as replication package.

References

Ait, A., Izquierdo, J.L.C., Cabot, J., 2022. An empirical study on the survival rate of GitHub projects. In: Proceedings of the 19th International Conference on Mining Software Repositories. pp. 365–375.

Akhin, M., Belyaev, M., 2021. Kotlin language specification. Kotlin Lang. Specif..

Andrä, L.-M., Taufner, B., Schefer-Wenzl, S., Miladinovic, I., 2020. Maintainability metrics for Android applications in kotlin: An evaluation of tools. In: Proceedings of the 2020 European Symposium on Software Engineering. pp. 1–5.

Anon., 2022. From zero to 10 million lines of Kotlin. https://engineering.fb.com/2022/10/24/android/android-java-kotlin-migration/. (Accessed 30 September 2023).

Anon., 2023. Kotlin for Android. https://kotlinlang.org/docs/android-overview.html.

Ardito, L., Barbato, L., Castelluccio, M., Coppola, R., Denizet, C., Ledru, S., Valsesia, M., 2020a. Rust-code-analysis: A rust library to analyze and extract maintainability information from source codes. SoftwareX 12, 100635.

Ardito, L., Barbato, L., Coppola, R., Valsesia, M., 2021. Evaluation of rust code verbosity, understandability and complexity. PeerJ Comput. Sci. 7, e406.

Ardito, L., Coppola, R., Barbato, L., Verga, D., 2020b. A tool-based perspective on software code maintainability metrics: A systematic literature review. Sci. Program. 2020, 1–26.

Ardito, L., Coppola, R., Malnati, G., Torchiano, M., 2020c. Effectiveness of Kotlin vs. Java in android app development tasks. Inf. Softw. Technol. 127, 106374. http://dx.doi.org/10.1016/j.infsof.2020.106374, URL https://www.sciencedirect.com/science/article/pii/S0950584920301439.

Bose, S., Mukherjee, M., Kundu, A., Banerjee, M., 2018. A comparative study: Java vs kotlin programming in android application development. Int. J. Adv. Res. Comput. Sci. 9 (3), 41–45.

Chidamber, S.R., Kemerer, C.F., 1994. A metrics suite for object oriented design. IEEE Trans. Softw. Eng. 20 (6), 476–493. http://dx.doi.org/10.1109/32.295895.

Coelho, J., Valente, M.T., Milen, L., Silva, L.L., 2020. Is this GitHub project maintained? Measuring the level of maintenance activity of open-source projects. Inf. Softw. Technol. 122, 106274.

Coppola, R., Ardito, L., Torchiano, M., 2019. 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. pp. 8–14.

Dorofeev, K., Wenger, M., 2019. Evaluating skill-based control architecture for flexible automation systems. In: 2019 24th IEEE International Conference on Emerging Technologies and Factory Automation. ETFA, pp. 1077–1084. http://dx.doi.org/10.1109/ETFA.2019.8869050.

Ebert, C., Cain, J., Antoniol, G., Counsell, S., Laplante, P., 2016. Cyclomatic complexity. IEEE Softw. 33 (6), 27–29. http://dx.doi.org/10.1109/MS.2016.147.

Flauzino, M., Veríssimo, J., Terra, R., Cirilo, E., Durelli, V.H.S., Durelli, R.S., 2018. Are you still smelling it? A comparative study between Java and kotlin language. In: Proceedings of the VII Brazilian Symposium on Software Components, Architectures, and Reuse. pp. 23–32.

Frakes, J., 1996. First contact. Paramount Pictures.

Fu, X., Lee, D., Jung, C., 2018. nAdroid: Statically detecting ordering violations in Android applications. In: Proceedings of the 2018 International Symposium on Code Generation and Optimization. pp. 62–74.

Góis Mateus, B., Martinez, M., 2019. An empirical study on quality of Android applications written in kotlin language. Empir. Softw. Eng. 24, 3356–3393.

Govil, N., 2020. Applying halstead software science on different programming languages for analyzing software complexity. In: 2020 4th International Conference on Trends in Electronics and Informatics. ICOEI (48184), IEEE, pp. 939–943.

Grano, G., Di Sorbo, A., Mercaldo, F., Visaggio, C.A., Canfora, G., Panichella, S., 2017. Android apps and user feedback: A dataset for software evolution and quality improvement. In: Proceedings of the 2nd ACM SIGSOFT International Workshop on App Market Analytics. pp. 8–11.

Hariprasad, T., Vidhyagaran, G., Seenu, K., Thirumalai, C., 2017. Software complexity analysis using halstead metrics. In: 2017 International Conference on Trends in Electronics and Informatics. ICEI, pp. 1109–1113. http://dx.doi.org/10.1109/ICOEI.2017.8300883.

Hecht, G., Bergel, A., 2021. Quantifying the adoption of kotlin on Android stores: Insight from the bytecode. In: 2021 IEEE/ACM 8th International Conference on Mobile Software Engineering and Systems, MobileSoft. IEEE, pp. 94–98.

Hellbrück, S., 2019. A Data Mining Approach to Compare Java with Kotlin. (Bachelor's thesis). Metropolia Ammattikorkeakoulu.

IEEE, 1990. IEEE standard glossary of software engineering terminology.

Kalliamvakou, E., Gousios, G., Blincoe, K., Singer, L., German, D.M., Damian, D., 2014. The promises and perils of mining github. In: Proceedings of the 11th Working Conference on Mining Software Repositories. pp. 92–101.

Li, W., 1998. Another metric suite for object-oriented programming. J. Syst. Softw. 44 (2), 155–162. http://dx.doi.org/10.1016/S0164-1212(98)10052-3, URL https://www.sciencedirect.com/science/article/pii/S0164121298100523.

Martinez, M., Mateus, B.G., 2020. 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.

Martinez, M., Mateus, B.G., 2021. Why did developers migrate android applications from java to kotlin? IEEE Trans. Softw. Eng. 48 (11), 4521–4534.

McCabe, T.J., 1976. A complexity measure. IEEE Trans. Softw. Eng. SE-2 (4), 308–320. http://dx.doi.org/10.1109/TSE.1976.233837.

Mohsen, F., Oosterhaven, L., Turkmen, F., 2021. KotlinDetector: Towards understanding the implications of using kotlin in Android applications. In: 2021 IEEE/ACM 8th International Conference on Mobile Software Engineering and Systems, MobileSoft. IEEE, pp. 84–93.

Oliveira, V., Teixeira, L., Ebert, F., 2020. 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, pp. 206–216.

Peters, M., Scoccia, G.L., Malavolta, I., 2021. 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, IEEE, pp. 36–46.

Schwermer, P., 2018. Performance evaluation of kotlin and java on android runtime.

Shao, J., Wang, Y., 2003. A new measure of software complexity based on cognitive weights. Can. J. Electr. Comput. Eng. 28 (2), 69–74.

Shepperd, M., 1992. Products, processes and metrics. Inf. Softw. Technol. 34 (10), 674–680.

Vasa, R., Schneider, J.-G., 2003. Evolution of cyclomatic complexity in object oriented software. In: Proceedings of 7th ECOOP Workshop on Quantitative Approaches in Object-Oriented Software Engineering. QAOOSE'03, Darmstadt, Germany, pp. 1–5.

Wohlin, C., Runeson, P., Höst, M., Ohlsson, M.C., Regnell, B., Wesslén, A., 2012. Experimentation in Software Engineering. Springer.

Zeng, Y., Chen, J., Shang, W., Chen, T.-H., 2019. Studying the characteristics of logging practices in mobile apps: A case study on f-droid. Empir. Softw. Eng. 24, 3394–3434.