Received: Added at production | Revised: Added at production | Accepted: Added at production

DOI: xxx/xxxx

ARTICLE TYPE

# Understanding the Quality and Evolution of Android App Build Systems

Pei Liu[1] | Li Li[1] | Kui Liu[2] | Shane McIntosh[3] | John Grundy[1]

[1]Faculty of Information Technology, Monash University, Victoria, Australia
[2]Huawei Software Engineering Application Technology Lab, Hangzhou, China
[3]David R. Cheriton School of Computer Science, University of Waterloo, Ontario, Canada

Correspondence
Pei Liu. Email: Pei.Liu@monash.edu
Li Li. Email: lilicoding@ieee.org
Kui Liu. Email: brucekuiliu@gmail.com
Shane MacIntosh. Email: shane.mcintosh@uwaterloo.ca
John Grundy. Email: John.Grundy@monash.edu

Build systems are used to transform static source code into executable software. They play a crucial role in modern software development and maintenance. As such, much research effort has been invested in understanding the quality and evolution of build systems, including Apache ANT, Apache Maven, and Make-based ones. However, the quality and evolution of build systems for mobile apps, such as on the Android platform, have not as yet been investigated in detail. Mobile app development, and the Android development context in particular, impose unique constrains, such as different device conditions and capabilities. It presents unique challenges, such as frequently upgraded Android frameworks, which those who implement and maintain build systems must tackle. In this paper, we present an exploratory empirical study of the build systems of 5,222 Android projects to better understand their quality and evolution. We (a) study the build technology choices that Android developers make (Gradle being recommended and the most popular choice); (b) explore the sustainability of the official Gradle build system (parts of build files are updated more frequent that others and the update of the special Gradle plugin would induce unrecommended configurations); and (c) analyze the quality of Gradle scripts for Android apps – more than a half of the open-source Android apps cannot be successfully built due to 5 common root causes.

KEYWORDS:
Build Systems, Android, Open-Source,Apache ANT, Apache Maven, Gradle

## 1 | INTRODUCTION

Build systems—the systems responsible for transforming of source code into executable software artifacts—are one of the most important technical artifacts that support the development of modern software. A typical software application is composed of many modules, containing multiple source files, which rely upon a complex layer of third-party libraries. These artifacts must be assembled carefully in order to produce a valid deliverable. It is not uncommon for build systems to orchestrate the invocation of hundreds of order-dependent commands.

Since their introduction in the 1970s [1], build systems have become a key part in the process of software development [2,3]. Build systems are at the heart of modern development approaches, such as continuous integration. In these rapid integration processes, an internal or third-party service (e.g., Circle CI [4]) verifies that the build process can still cleanly apply to changes to the codebase as they are produced by the development team. Modern rapid release strategies extend the continuous integration process to delivery/deployment, (semi-)automatically producing new releases when changes to the codebase pass a set of quality gates (e.g., code compiles, tests pass). As argued by McIntosh et al. [5], a fast and correct build system is critical. Without a robust and fast build system, continuous integration and delivery/deployment (CI/CD) processes would not produce

integration reports or new releases in a timely manner, hindering development progress[6,7,8,9]. Without a correct build system, CI/CD processes would not be reliable enough to foster trust from the development team, leading to miscommunication and/or unacceptable releases.

Due to their importance, build systems have attracted the attention of the research community. Adams et al.[10], Zadok et al.[11] and Nadi et al.[2] set out to understand how make-based build systems (co-)evolve with C and C++ codebases. McIntosh et al.[5] studied similar phenomena in Apache ANT and Maven build systems concerning the automation of Java source code. These studies have highlighted that a better understanding build systems will (1) allow project managers (especially for aging projects) to allocate appropriate personnel and resources to perform system maintenance tasks effectively, and (2) reduce build maintenance overhead on regular development activities.

While much has been discovered about the build systems of traditional applications[12,13], little is known about the evolution of build systems for mobile applications. Nowadays, Android is the most popular mobile platform with over 4,000 versions of different devices, over 2.5 billion monthly active users, and almost three million distinct Android apps published on the official Google Play store. While Android applications can be written in a Java-like language, their event-driven development model is quite different from a typical Java application.

In this work, we set out to better understand how Android build systems evolve. To do so, we perform a comprehensive empirical study of 5,222 Android applications from the AndroZooOpen corpus[14]. Our empirical study mainly answers three research questions: (1) What are the build technology choices that Android app developers make? (2) How do app developers work on the current Gradle build system to fulfill their development? (3) How many open-source Android projects provide valid Gradle scripts to successfully build deliverables?

The key contributions of this work include:

**Build Technology Choices:** We carried out an extensive study on 5,222 Android open-source projects and found that there are *four* common build technologies used – Apache ANT, Apache Maven, Eclipse ADT and Gradle. We found that of the many existing build systems, Android developers predominantly adopt Gradle, i.e., the default build technology of the Android SDK ($95.12\% = \frac{4,967}{5,222}$). We found that developers who adopted other build systems earlier have then often migrated their legacy build systems to Gradle. Of the 130 repositories we detected build system migration, 118 of them have changed their build system to Gradle. Our manual investigation further reveals that, in many cases, it is mandatory to update the fundamental build techniques as historical build techniques may not be supported anymore (it could yield library not found errors when performing the build).

**Sustainability of Gradle Build System:** We analyzed the evolution of build systems and found that after the build files are well configured, parts of the build scripts are changed more frequently than others, especially Android-related ones. Furthermore, since the Gradle plugin by itself may evolve (e.g., certain configuration keywords such as *compile*, *apk*, or *provided* may be deprecated), the underlying Gradle build scripts need to be aligned with such updates. Otherwise, the build scripts cannot be executed to generate installable apks.

**Quality of Gradle scripts in open source Android apps:** We manually summarized root causes of build failures of open-source Android projects. Among 4,774 projects containing a Gradle wrapper script, only 1,495 (or 31.32%) of them can be automatically built. From 100 randomly sampled repositories that failed to build, we manually checked their build failure reasons and summarized five key root causes, including (1) Source Code Error, (2) Configuration Error, (3) Resource File Missing, (4) Library Not Available, (5) NDK Error. The root causes of the build failures conform to a previously established set of categories by Hassan et al.[12] and will be helpful for developers to avoid such unnecessary problems.

This is the first large-scale analysis on the evolution of build systems in Android apps development so far. We detail the updates of Gradle build system scripts and investigate the rationale behind the update in Android APKs' auto-build, which would benefit both Android app developers and build system maintainers.

The rest of this paper is organized as follows. In section 2, we present the common build systems and our research questions. In section 3, we introduce our dataset selection. Section 4, 5, 6 focus on three different research questions described in section 2, respectively. Section 7 implies threats to validity. Then we discuss the potential usefulness for potential practitioners and researchers at Section 8 and related work at Section 9. At last, we conclude our work at section 10.

Our source code and datasets are all made publicly available in our artifact package[15].

## 2 | ANDROID BUILD SYSTEMS

Android applications are complicated Java-based systems with many files, directories and inter-relationships. Building these apps requires sophisticated build tool support for developers. We outline some of the most commonly used build tools for Android applications below:

- Apache **ANT**, an acronym for Another Neat Tool, was created by James Duncan Davidson in 1999[16,5]. It originated from the Apache Tomcat project in early 2000 and was designed for automating software build processes due to a number of limitations with Unix's make system, which was the *de facto* standard automatic build tool among system programming languages, such as C/C++. Apache ANT is written in Java and provides a number of built-in tasks to compile, assemble, test, and run Java applications[17,18].

- Apache **Maven**, primarily designed for Java projects, was created by Jason van Zyl in 2002 as a subproject of Apache Turbine [19]. It is also a build automation tool and can be used to manage the build process, report, and document projects. Unlike Apache ANT, it uses conventions to provide default behavior for the build procedures and also offers the ability to automatically manage third-party libraries for projects [20]. The project `botbrew-gui` of entry `jyio/botbrew-gui` introduces the Maven installation commands to build the repository.

- **Eclipse ADT** was the initial officially-supported IDE for Android app development. ADT stands for Android Development Tools. It extends Eclipse with the capacity to create Android projects, add and manage packages used by the project based on the Android Framework API, debug the applications using the Android SDK (Software Development Kit) and export signed or unsigned APKs to distribute the final artifacts [21]. The plugin was developed by Google and designed to provide developers with a powerful integrated environment to develop Android applications. For example, the Github project `DroidShows` of entry `ltGuillaume/DroidShows` uses Eclipse as its build system, and the project is still under maintenance. However, at the end of 2015, Google announced that ADT was deprecated and the IDE would be replaced by Android Studio.

- **Gradle** is an open-source build automation system, built upon the concepts of Apache ANT and Apache Maven, and also provides a Groovy-based [22] and Kotlin-based [23] Domain Specific Language (DSL). Based on Apache ANT and Apache Maven, Gradle uses tasks to represent atomic build activities, such as compiling the source code, packing a JAR file, generating Javadoc, etc. The dependencies between these tasks form a directed acyclic graph (DAG), which the Gradle tool uses to safely execute tasks in parallel to speed up build execution [24]. For example, the entry of the project `Maxr1998/home-assistant-Android` instructs users to generate the installable by executing the Gradle command.

## 2.1 | Gradle

Apache ANT and Apache Maven have been significantly investigated by our fellow researchers [5] and Eclipse ADT has been deprecated. Thus here we focus on the newly introduced Gradle. A typical Android project, created by Android Studio with the default Gradle build system, contains several different configuration files as shown in Figure 1 [1 25].

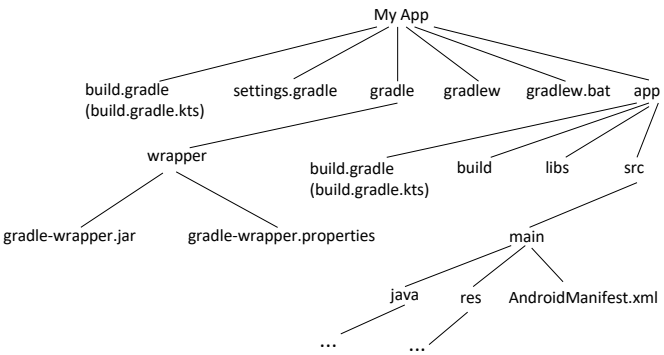



**Figure 1** Typical directory structure of Android Application adopting Gradle build system.

The structure of a Gradle configuration contains project-level and module-level build files. In the root directory of an Android app project, it contains two project-level configuration files: build.gradle and settings.gradle. The settings.gradle file is used to determine the required modules for building the app, while the file build.gradle defines build configurations [26]. A configuration represents a group of artifacts and their dependencies. In the build.gradle file, the `buildscript` block configures the repositories and dependencies. The repositories can be pre-defined remote repositories, common places for finding/sharing popular packages used by build systems (e.g., JCenter, Maven Central, and Ivy), any local repositories, or self-defined remote repositories. The dependencies declared in this block are searched and downloaded from the specified list of repositories. Unlike other build systems, these dependencies are consumed by Gradle itself, such as the dependency of the Android Gradle plugin that provides additional instructions for Gradle to build Applications. The `allprojects` block in the build.gradle defines the third-party repositories and dependencies, which can be consumed by all of the modules of the project. Different from the `buildscript` block configurations, this block section is for the modules and sub-modules in projects being built by Gradle.

[1] https://developer.android.com/studio/build

As well as the project-level build file, module-level build files configure build settings for the specific module where it resides. The build settings include the basic configurations, like dependencies and build tasks for the module-specific build.

The Gradle plain text configuration files build.gradle are generally written in the Groovy language. Groovy is a dynamic Domain Specific Language (DSL) for the Java Virtual Machine. The corresponding build file fulfilling the same function is build.gradle.kts, written in the Kotlin language [27]. Kotlin is the preferable programming language for Android Application development, announced by Google on 7 May 2019. Thus a Kotlin-based configuration makes the development language the same as configuration, which reduces the learning time for Android Application development [28].

In addition, the Gradle wrapper (i.e., `MyAppp/gradlew`) is also located in the root directory. The Gradle Wrapper is the recommended way to execute any Gradle build. It is a script that invokes a declared version of Gradle and downloads Gradle beforehand if need be. The wrapper file also relies on two files, the gradle-wrapper.jar and gradle-wrapper.properties. The gradle-wrapper.jar file is responsible for downloading the Gradle distribution, while the gradle-wrapper.properties file configures the wrapper runtime behavior.

Gradle provides different wrappers for different operating systems, such as a shell script for unix-like systems and a batch script for Windows. It allows developers to invoke Gradle without first (manually) installing it. Moreover, the script ensures build consistency by ensuing that the same version of Gradle (the one specified in the properties file) is used for each build.

The core implementation of the application is located in the directory main. It typically contains the directory java, res, and Android configuration file AndroidManifest.xml. The file with the exact name AndroidManifest.xml is a mandatory file in which the components of the app, the permissions information, and the hardware and software requirements are all listed. The directory java contains the detailed implemented Java and/or Kotlin files while the directory res holds the necessary resource files, such as the pre-loaded pictures, the localization languages support etc. [29].

## 2.2 | Research Questions

In this work we want to answer the following three key research questions:

**RQ1: Which build technologies are most prevalent in the Android ecosystem [Technology Choice]?** By answering this research question, we expect to present to the community with a clear picture of the predominant build technology choices in the Android ecosystem. Novice Android developers can, therefore, adopt the advanced and popular build system to facilitate their development. Developers using legacy build systems can still gain benefit from the popular one with the active development community and plentiful resources of the popular build system if they are willing to migrate their legacy build system.

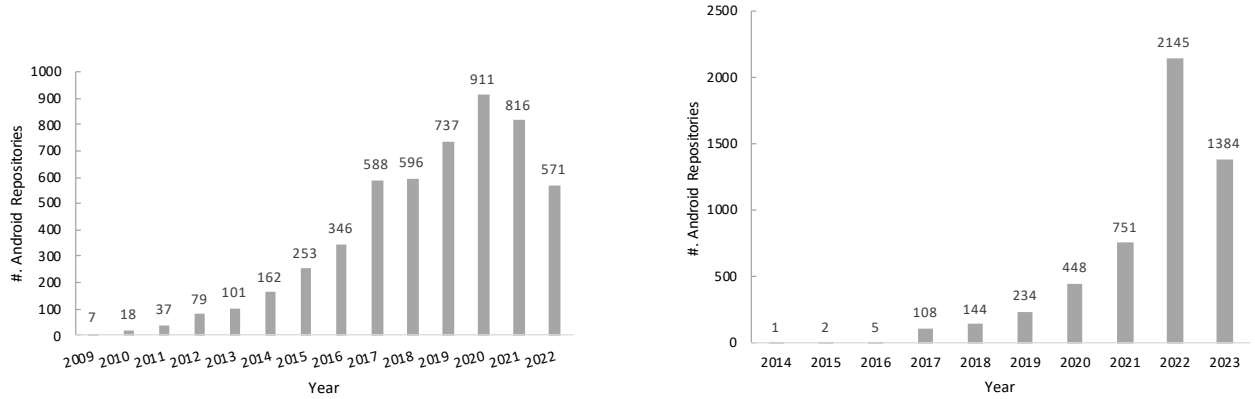
**RQ2: How do app developers use the current Gradle build system in their app development [Suitability]?** Our preliminary investigation reveals that Gradle, the current official build system recommended by Google, is the most frequently adopted build system in the Android ecosystem. Focusing on the evolution of the Gradle build system and answering this research question, we could understand the impact of the build system, Gradle, on the Android apps development and enlighten app developers on the maintenance of the build system.

**RQ3: Can open-source Android app projects be successfully built with their provided build scripts [Quality]?** Open-source Android developers publish the core app source code alongside with the build system on the public hosting site. Other developers should be able to build and test the app with the help of the provided build scripts so as to collaborate on new features and bug fixes etc. In this research question, we want to determine if the Gradle build scripts provided in open source app repositories can be directly used to build the apps. By answering this research question, we can determine how many open source Android app build scripts are valid, and the common root causes for open source apps not being able to be built.

# 3 | EXPERIMENTAL DATASET

We describe our dataset – a set of open-source Android app repositories – that we prepared to empirically address the research questions enumerated above. We used the curated AndroZooOpen dataset to select suitable open-source Android repositories. AndroZooOpen is a growing and publicly-available dataset containing over 85,269 open-source Android app repositories that are collected from various open-source project hosting sites, such as Github and Bitbucket. Since this paper focuses on understanding the build approaches and practices of Android apps, we built our dataset by considering the Android projects that have published their source code on popular code hosting sites and their installable APKs (built by the published source code) have been released on Google Play Store or F-Droid. We believe that the apps deployed on Google Play or F-Droid are more likely to involve building automation techniques than other apps, and such building techniques related configurations are also more likely to be correct. However, the majority of apps of AndroZooOpen cannot be identified on Google Play or F-Droid. That is because some of them are toy projects, which were developed just for practice by developers, while some others have been removed from Google Play Store (Google Play Store

always removes apps roughly once a quarter [2]) as Google believes that they are of low quality. We inspected every project from AndroZooOpen and identified if the projects have been built and published on Google Play Store or F-Droid. Eventually, 5,222 apps from AndroZooOpen, not only hosted on public hosting sites but also published on Google Play Store or F-Droid, are selected as the dataset for this study.




(a) The number of projects introduced per year.

(b) The number of projects in latest update per year.

**Figure 2** The number of projects introduced and updated each year.

For our identified 5,222 open source Android app repositories, we used the official provided REST API version 3 [30] to acquire their basic repository information on Github, such as the create timestamp, the last update date, the number of contributors, stars given by fellow developers, forks, and the creation of issue and pull requests etc. Figure 2(a) summarizes the distribution of the number of projects in our dataset with respect to their creation years in their corresponding online hosting sites.

In our dataset, the creation year of the projects spans from as early as 2009 when Android was first introduced to as late as 2022 when we collected the data for this study. The number of newly created open-source Android app projects continuously increases and reaches a peak in 2020 [3].

Figure 2(b) presents the distribution of the number of projects based on their latest update. The fact that the majority of apps are updated in recent years (at least 90.54% $(4,728/5,222)$ are updated within 3 years) shows that most of our selected projects are still under active development, and hence should provide a reasonable lens through which we can understand the status-quo and the evolution of Android build systems. The representativeness of our dataset is further demonstrated by the popularity of the selected projects. In Figure 3(a), over half of the projects receive 3.5 stars and 1 forks, 3 issues, and 8 pull requests. For example, repository [31] received 60 stars and was forked 12 times at the time of our visit.

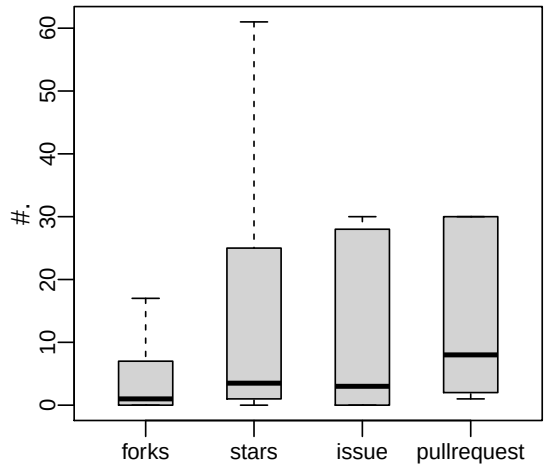
With respect to the contributors for the open-source projects, we first count the total number of contributors and then compute the number of active contributors who contribute the most of the projects periodically. Figure 3(b) shows that the selected projects are almost all implemented by a single contributor. We also compute the time interval for developers to close the created issues and pull requests. Figure 3(c) represents the results, indicating that developers take more then 19 and 3 hours to close the issues and pull requests, respectively. Liu et al. [32] first did an extensive study with respect to CI/CD utilizations among open-source Android repositories. Among our selected 5,222 Android repositories, we could only detect 1,139 (21.81%, (1,139/5,222)) projects containing CI/CD setup, which is still a low percentage. Developers should take more efforts to work on the CI/CD setup among open-source Android projects.

We compute the duration years of the selected repositories represented in Figure 3(d) during which developers still maintain the repositories on hosting sites. Figure 3(d) shows the median value of the maintenance period among the selected Android repositories is 3. Therefore, we must take these no longer maintained projects into consideration as they represent the utilization of the advanced build systems at that time.

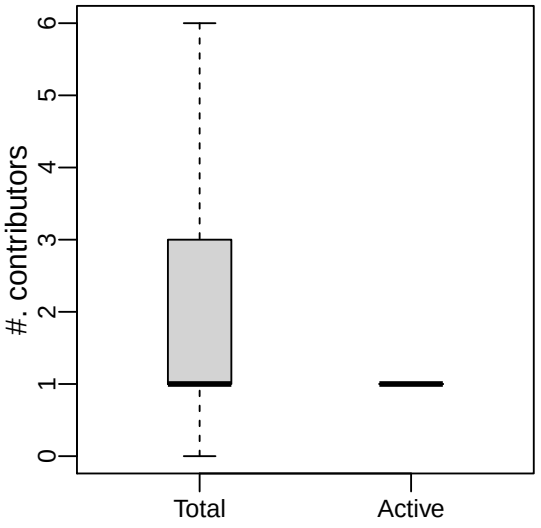
We requested the metadata of the selected apps on Google Play store with the help of Google Play Scraper [33]. Figure 4 represents the number of installs and rating score of the selected apps. Figure 4(a) shows how many installations for the selected apps. It is worth mentioning that Google Play store shows the number of installations in a range rather than a specific number [34]. For example, installs *10+* represents the corresponding

[2] https://www.appbrain.com/stats/number-of-android-apps

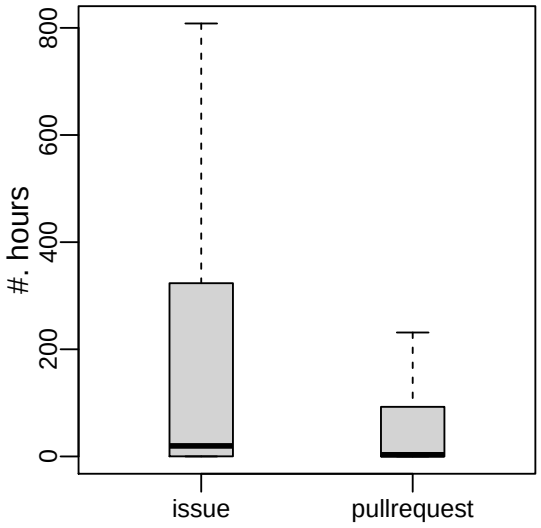
[3] It's hard to clearly pinpoint reasons why the decline of the number of open-source Android projects in 2021 and 2022. That is may because some projects were created as private and changed to public later. The outbreak of the pandemic might also have a negative effect on the open-source projects development (not limited to but included Android project



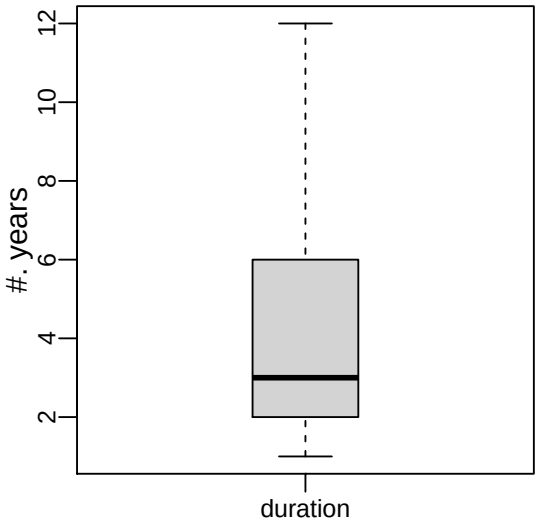

(a) Distribution of the number of forks, stars, issues, and pull-requests.




(b) Total and active number of contributors for the selected projects.




(c) Time interval to close issues and pull request.



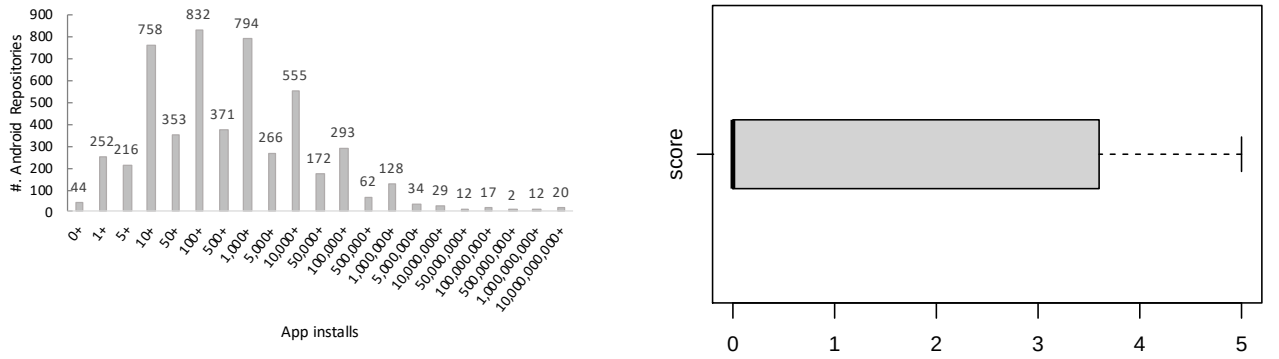

(d) Maintenance duration of the selected repositories.

**Figure 3** The basic attributes of the selected Android projects on hosting sites.

apps are installed spanning from 11 to 50 times by smartphone users. In our dataset, there are 758 apps with installs 10+. Figure 4(b) indicates the score given by app users and shown on play store. It shows the median value of score is 0 and the maximum value is 5. It is notable that not all apps in play store have a rating [4]. Some of the selected apps were uploaded recently but have gained much popularity among app users. With the installations and scores acquired from Google Play store, we could have a high-quality dataset including the newly introduced apps and fully-fledged ones to conduct our study.

The availability of the built APKs and the maintenance period of the repositories demonstrate the necessity of an extensive analysis on a large-scale dataset including projects out of maintenance in the last 3 years.

[4] https://support.google.com/googleplay/answer/6209544?visit_id=638113344792347159-3619086670&p=appgame_ratings&rd=1




(a) The number of installations of selected apps.

(b) The score of selected apps given by app users.

**Figure 4** The basic attributes of the selected Android projects on app store.

## 4 | BUILD TECHNOLOGY CHOICES (RQ1)

In this section, we answer the research question RQ1 – which build technologies are most prevalent in the Android ecosystem? The Android app build system plays a key role to compile source code and package them into APKs for further testing, deploying, signing, and distribution, so as to allow CI/CD. We conduct a preliminary analysis aiming at understanding the status of usage of different build systems used by Android developers in the historical and current Android apps?

The following sub-research questions are answered:

- **RQ1.1: What are the build techniques recurrently adopted by Android apps?**

  Android apps integrate their app code together with third-party libraries and resources and may need to be built into different variants for different types of devices. Building apps manually is arduous and prone to errors. With the help of the build system, it is easier to reuse code, resources, and configurations. In addition, extending and customizing the build process specified in a build system is a more traceable and manageable task than manual app assembly. To begin our investigation of build systems for Android Apps, we first set out to understand what build techniques have been used by developers in the Android ecosystem.

- **RQ1.2: How often do Android App developers update their build practices?**

  There are plenty of build systems that can be used to specify build processes. Newer build technologies often offer more features, but there is a cost to migrate [35]. In our second subquestion, we are interested in understanding whether and how often Android App developers migrate between technologies. This knowledge will not only help new developers choose between different build technologies that facilitate their app build processes, but also encourage maintainers to fully consider the value of migrating away from legacy build systems to state-of-the-art ones.

### 4.1 | RQ1.1: What build techniques are recurrently adopted by Android apps?

By sampling and manually analyzing projects of the dataset, we are able to identify the following build technologies, including Apache ANT, Apache Maven, Eclipse ADT, and Gradle. We now briefly describe them respectively.

**Table 1** Key Build System Resources

| | None | Eclipse ADT | Apache ANT | Apache Maven | Gradle | Multiple |
|---|---|---|---|---|---|---|
| Build files | None | .classpath<br>.project | ant.properties<br>build.xml | pom.xml | build.gradle<br>build.gradle.kts | more than one spec |

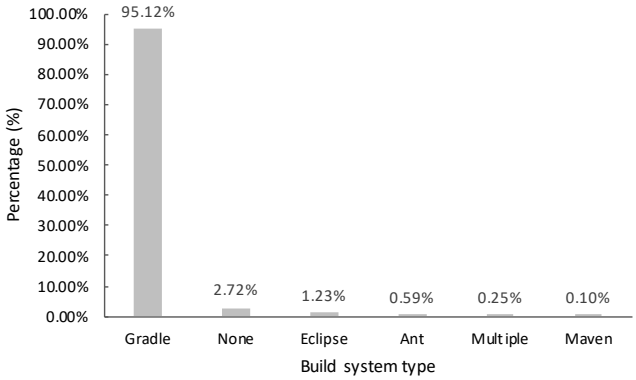



**Figure 5** The build system of Android repositories.

Given an open-source Android app project, we need to automatically identify which build technology is adopted by the project. In this work, we take a straightforward approach to achieve that. As shown in Table 1, each build technology requires specific files, which could then be leveraged to form the classification criteria. For example, if a given project contains a *build.gradle* or a *build.gradle.kts* file, it follows that the build system leveraged by this project is Gradle. Additionally, for projects that do not contain any of the recognized files for the four build techniques or contain recognized files for more than one build technology, we label the project as "None" or "Multiple", respectively. It is worth noting that projects labeled "None" do not mean they were built without any build systems. It is the developers of these projects who are not willing to upload the build scripts to source code hosting sites that prevents us from determining the build systems utilized in these projects.

Based on the above approach for determining build technologies, we classify the latest version of all of the projects in our dataset and summarize the build technology choices in Figure 5. Unsurprisingly, the most popular build technology is Gradle (95.12% of projects), the default build system generated by Android studio (the official Android IDE).

**RQ1.1** Finding

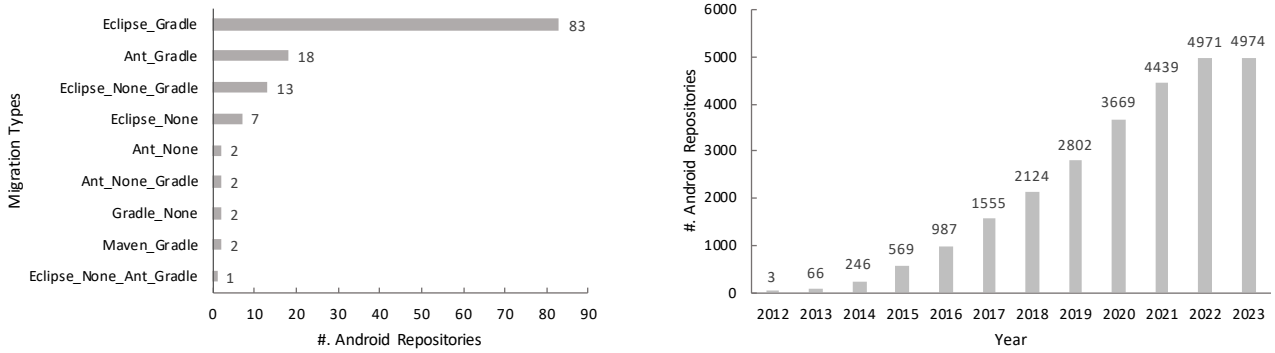
The Apache ANT, Apache Maven, Eclipse ADT, and Gradle build technologies all could be adopted for building Android apps. Among the four techniques, by far the dominant technology is Gradle - the officially supported technology of the Android development tooling.

## 4.2 | RQ1.2: How often do Android App developers update their build practices?

The majority of our selected projects adopt Gradle for their build system. However, this is only determined from the last update of every project in our dataset, which reflects their latest status. What is still unclear is how quickly the Android community has converged on its current status. To investigate this, we inspected every history commit state via Git to get the information about the build system's evolution. With the help of Git, we analyze the entire commit history via the *reflog* command, and we reuse the classification criteria to identify possible migrations of the build technology.

Figure 6(a) summarizes our experimental results concerning the changes made in migrating from one build technology to another. We tag these PriorTechnology_NextTechnology. For example, Eclipse_Gradle represents a the migration from Eclipse ADT to Gradle. During the lifecycle of an open-source project, migration among build technologies may happen more than once. For example, the developers of *hgdev-ch/toposuite-android* have migrated its build systems twice, at first from Eclipse ADT to Apache ANT, and then from Apache ANT to Gradle (i.e., Eclipse_None_Ant_Gradle).

The most popular migration pattern is from the former officially-supported build system Eclipse ADT to the current officially-supported system Gradle. We suspect that this is common because Google announced that the applications can be implemented using Kotlin, and also can be implemented in native code, such as C/C++ with the help of the Android NDK. It is not easy to implement Apps using a combination of Java, Kotlin, and native code, especially without support from the build technology. Since Apache Maven and Apache ANT lack first-class support for Kotlin and C/C++, while Gradle provides greater flexibility and better performance, adding support for building apps that combine Java, Kotlin and native code was likely easier with Gradle. In addition, Gradle, as the latest build system with the build features including build caching, compile avoidance, and an improved incremental Java compiler, is now 2-10x faster than maven [36] for most Java projects.




(a) Build system migration.

(b) Projects adopting Gradle each year.

**Figure 6** The migration of build systems

Migrating to Android Studio with the Gradle build system from Eclipse ADT is also an actively supported path [37]. Eclipse ADT projects can be imported directly into Android Studio, and developers can complete a migration by responding to a series of prompts. An import summary text file will be generated once the migration is complete. Developers could have a better understanding of what has been changed in the project by reading the generated summary file and manually correcting what has been updated incorrectly.

To migrate from Apache Maven to Gradle, some documentation [38] provides guidelines for developers. This documentation lists the mapping of build phases of Maven to tasks of Gradle and provides instructions to incorporate plugins, dependencies, and default library repositories. It is not difficult to migrate to Gradle from Apache Maven because they share conventions, such as project structure and dependency management. Our previous analysis reveals two projects that migrated from Apache Maven to Gradle directly. We manually analyzed these migrations and found that all migrated by following the instructions in the documentation.

Similar to migrating from Apache Maven to Gradle, some projects migrated to Gradle from Apache ANT. However, it is difficult to migrate from Apache ANT to Gradle, as there does not exist any standard Apache ANT build. Developers need to migrate to Gradle according to their specific Apache ANT build conventions. Fortunately, Gradle provides integration features with Apache ANT that can ease such migration. In general, there are two approaches to complete the migration. The first approach uses the existing Apache ANT build as much as possible via hook methods provided by Gradle. The other approach migrates to an idiomatic Gradle build as much as possible, even though the core and custom Apache ANT tasks can still be used directly with Gradle hook methods. In our analysis, we find that all of the migrations from Apache ANT to Gradle tend to have a purely idiomatic Gradle build. That is because the Android projects do not rely on indispensable Apache ANT tasks via Gradle hook methods.

During our investigation, we also observed that 13 (or 0.25%) Android projects appear to contain multiple build systems. For example, project *MPieter/Notification-Analyser* adopted Gradle first, and later on, it includes Maven to the project. Our manual investigation reveals that the Maven build system is included because of an externally developed library that has been recently added to the project. Nonetheless, it is non-trivial to automatically distinguish between cases of migration from one technology to another and cases of coexisting build technologies. Furthermore, our investigation also reveals that some Android app projects have adopted build systems repeatedly. For example, as revealed in the commit history of project *Ifsttar/NoiseCapture*, it first adopts Gradle, and then migrated to Maven, only to later migrate back to Gradle.

**RQ1.2**

The build system adopted by app developers to automate the build process is not immutable from the beginning of the project. It is uncommon to replace one build technology with another except for the migration recommended by the Android official.

## 5 | EVOLUTION STUDY (RQ2)

In this section, we answer research question RQ2 – how do app developers work with the current Gradle build system? The build system of Android projects changes over time. We shift our focus on Gradle as many selected projects update their build system to Gradle, the officially support system. Based on the build technologies used (cf. Section 4), and the structure of the Gradle build system (c.f. Section 2), we need to answer two research sub-questions to analyze the evolution of the Android Gradle build system:

- **RQ2.1: What has been changed in Gradle build files during the evolution of open-source Android apps?**

The Gradle build system offers the ability to perform custom build configurations without modifying the source code in the project. This flexibility empowers the developers to customize and automate multiple build configurations. Build types (providing certain properties for Gradle to build and pack to the final artifact), product flavor (determining the product versions such as free or paid), and dependencies (managing remote repositories needed by developers' own local project development), along with some other aspects are totally configurable. This research question investigates what has been changed in Gradle build files during the project development.

- **RQ2.2: Why are build files changed?**

  It is important for developers to know the necessity behind these changes since it can help both the project owner and other developers to have a fuller understanding on the updates of to app projects. Thus, this research question focuses on investigating the rationale behind observed changes to build files over time.

To answer these research questions we implemented several Python scripts to inspect every git commit in the commit history for each selected project.

## 5.1 | RQ2.1: What has been changed in Gradle build files during the evolution of open-source Android apps?

A build system is used to automate and speed-up the whole Android build process. Once the build file is finished, the developers can leverage the convenience of the build system to focus and speed-up the development of the functionalities. Any change of the functionalities or unit tests can be readily and easily tested by just running some simple command lines provided by the build system.

```
1    buildscript {
2      dependencies {
3 -  classpath 'com.getkeepsafe.dexcount:dexcount-gradle-plugin:0.4.4'
4 +  classpath 'com.getkeepsafe.dexcount:dexcount-gradle-plugin:0.6.1'
5 }
```

**Figure 7** Example of updating the classpath taken from entry cgeo/cgeo

```
1   dependencies {
2 -     implementation 'com.google.firebase:firebase-core:17.0.1'
3 +     implementation 'com.google.firebase:firebase-core:17.2.0'
4 }
```

**Figure 8** Example of updating the dependency taken from entry liaoheng/BingWallpaper.

```
1   android {
2 -         versionCode 2231
3 +         versionCode 2234
4 }
```

**Figure 9** Example of updating the android taken from entry HabitRPG/habitica-android.

[5]We tried to contact the developers of the project to figure out the rationale behind the utilization of the build system BUCK. However, we do not have any response from them.

```
-// Run this once to be able to run the application with BUCK
-// puts all compile dependencies into folder libs for BUCK to use
-task copyDownloadableDepsToLibs(type: Copy) {
-    from configurations.compile
-    into 'libs'
-}
```

**Figure 10** Example of updating the task taken from entry mrf345/online-wallpapers.[5]

### 5.1.1 | Gradle Script Changes

We investigate what kinds of build files are changed in the evolution of Android Apps. To do this, two different authors, according to the Android build specification[25] (c.f. 2.1), manually analyzed 100[6] randomly selected app projects and classified the main and frequent kinds of changes made to the typical build files. With cross-validation between them, the changes in the typical build files are subsequently grouped into four categories – classpath, dependency, android and task – changes.

Figure 7[39] shows an example of updating the classpath in the declared dependencies of the buildscript for Gradle, such an example plugin is used to count the number of methods in an Android APK or AAR file when build. In general, configuration classpath locates in buildscript section in project-level build files used to specify transitive dependencies for Gradle itself, such as Gradle plugin. Figure 8 shows an example of modifying the dependency information in the build.gradle file that declares third-party libraries used by the source code to generate the final artifacts.

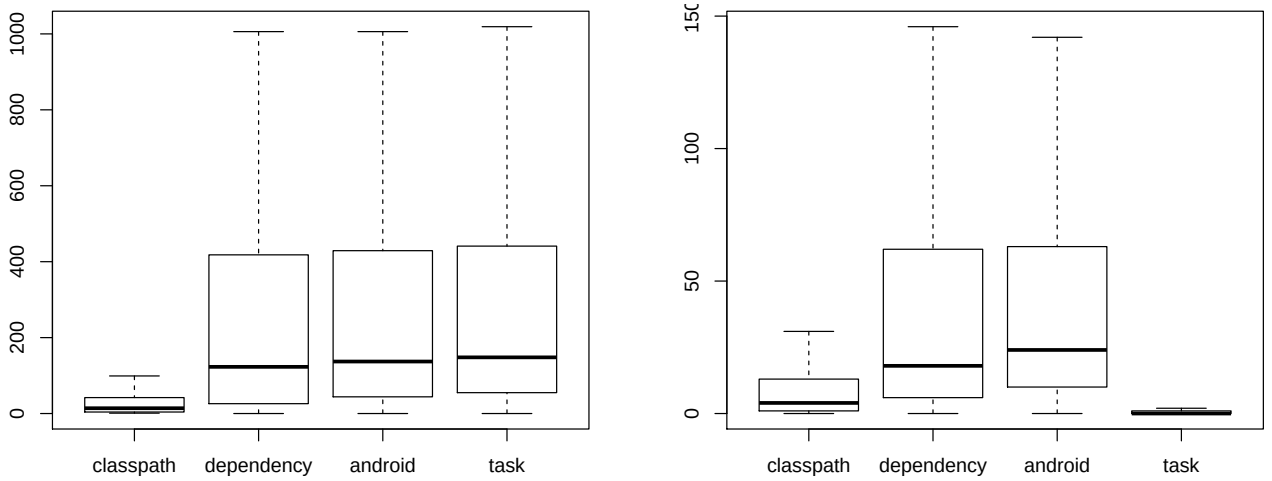
By convention, configuration dependencies can be specified both in project level build files which could be used by any sub-modules in the project and in module level build files which could only be utilized by the sub-module of the project. Figure 9 illustrates an example of revising the version code in android that is specific to Android projects in the sub-module build file to define the basic build settings such as build types and product flavors etc. The type of android is actually a configuration block in module-level build files.

In addition to version code, the android block (or android type we classified) contains a wide range of configurations, including compileSdkVersion, buildToolsVersion, applicationId, minSdkVersion, targetSdkVersion, versionName, build types (release, debug), product flavors (free, paid), signing (configurations of keyAlias, keyPassword etc.), code and resource shrinking (configurations of minifyEnabled, shrinkResources, proguardFiles), and multidex enabled when the total number of methods exceeds 65,536. We intentionally excluded the sub-block of dependencies from android block and classifies it as a separate type because it is updated more frequent than other types. Figure 10 presents an example of removing a task that was added by the developers to achieve their own built task. In general, developers do not need to build their specific tasks to finish their Android development. However, specific requirements can be satisfied with the project oriented tasks.

To have a deeper understanding of relationships between the changed code files and the triggering of the build files changes we again selected 100 projects in order to have a better representativeness. For each project, we sort the commit chronologically and then acquire the diff result between two consecutive commits. To go one step further, we also randomly select one diff result per project with both java source code and Gradle build files changed. Among the 100 diff results in 100 Android projects, only 39 of them contain direct clues indicating that new dependencies added and/or old dependencies removed would result in java source code update in order to adapt different dependencies. The remaining 61 projects do have build files and Java source code update but we cannot directly conclude there is a relationship between build files and Java source code. Moreover, 28 among the total 61 projects do not even contain updates of dependencies. The update of build files only lies in Android build configurations, such as update of SDK version, version name and version code etc.

We further investigate the updates made to Gradle script files with respect to the four categories of changes during the evolution of the projects. Figures 11(a) and 11(b) respectively illustrate the distribution on the commits and the updated statements about the four categories. From the aspect of the commit times, classpath-related updates are not committed as frequently as the updates of other three categories. From the aspect of the updated statements in each commit, the updates of classpath and task always involve a few statements, while developers always change multiple statements to adjust the related dependencies and android block. Although the task is as frequently updated as the dependencies and android block, the updates related to task always involve revising a few statements. Developers need to take special attention to the update of dependencies and android block of build files during and after the development of Android application to give customers a better satisfaction.

[6]We utilize the well-known Sample Size Calculator (https://www.surveysystem.com/sscalc.htm) with a confidence level of 95% and margin of error of 10% to compute the number of sample Android repositories. It gives out sample size 95. However, we decided to manually check 100 to ensure our representative results.




(a) The number of commits about the four update types.

(b) The number of updated statements about the four update types.

**Figure 11** Updates with respect to four update types.

**RQ2.1** Finding

The updates of build files can be grouped into four categories based on their modified positions. The android block and dependencies are modified more frequently with more updates than the classpath and task. Developers should pay special attention to the frequent update blocks, especially the Android specific one, android block, to keep the app build making use of newly released features of build system and satisfying the requirements of developers, app users, and app stores, etc.

## 5.2 | RQ2.2: Why are build files changed?

Android studio provides the automatic build system Gradle, but Gradle is not like any other build system. It only provides little authentic automation by itself. All of the useful and detailed implementation of the automation is provided by *plugins*. Intentionally, plugins add the necessary component tasks (e.g., JavaCompile), domain object (e.g. SourceSet), and conventions (e.g., the directory structure of the source code) as well as extending other helpful objects from other plugins.

### 5.2.1 | Android plugins

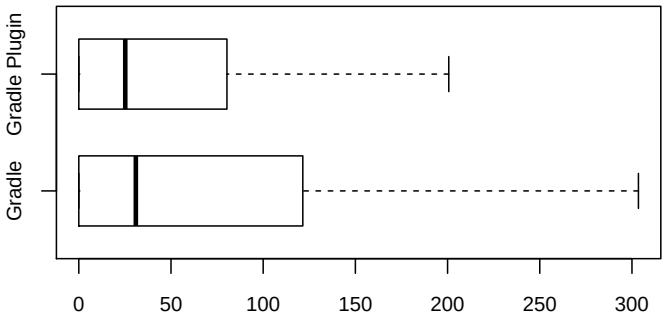


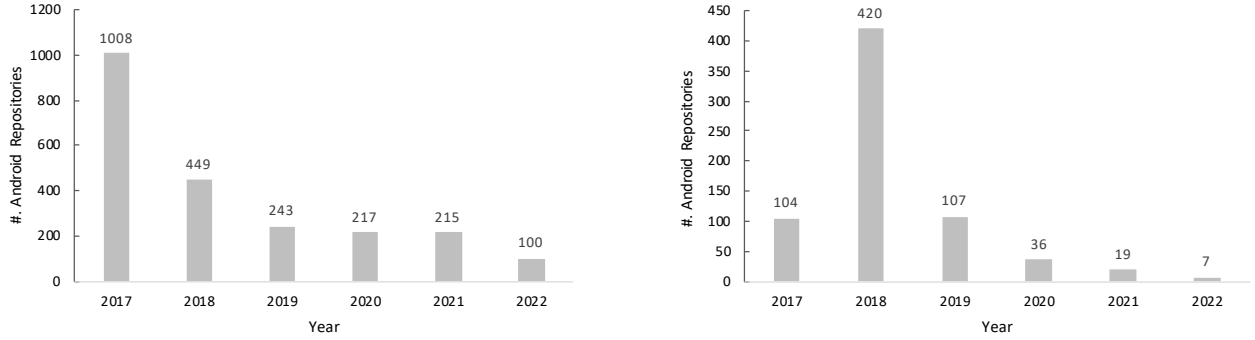

**Figure 12** The update interval (in days) of Gradle and Gradle Plugin.

When it comes to the Android App building, the Android Gradle plugin provides the specific build configurations and tasks, such as, `buildTypes`, `productFlavors` and `flavorDimensions` etc. In practice, Gradle and Android Gradle plugin can actually run independently in Android Studio, which provides the possibility to build Android Apps out of the IDE with the command line without Studio installed. In each Android repository, the build files provide the version of Gradle and Android Gradle plugin. The version of the Gradle plugin resides in the root level build file while the version of the Gradle locates in the `gradle-wrapper.properties`.

Via the Git, we examine every commit state by shifting to the specific commit with the Git command "git reset", extract the version of Gradle and Android Gradle plugin, and compute the average update interval of the version of Gradle and Android Gradle plugin. To compute the average update interval, we compute the total update time interval between two consecutive state-updated commits and calculate the average update interval for each Android app project. The distribution of the average update interval for the collected Android projects is shown in Figure 12, which shows that the version changes of Android Gradle plugin are slightly more frequent than the update of Gradle. This confirms that the Android Gradle plugin is more crucial for Android App build than Gradle, since the Gradle plugin involves the core development steps for Android Apps.

### 5.2.2 | Android configuration

The upgrade of the Android plugin could impact the build files of Android projects. One of the major changes on the build files build.gradle is the *configuration*, such as updating the classpath presented in Figure 7. The configuration in the context of Gradle refers to a set of artifacts and their dependencies. During the process of building software artifacts, their related dependencies have a specific intent in each build phase. For example, some dependencies are used in the phase of compiling source code while others would only be used at runtime. Gradle uses the term "configuration" to express the scope of a dependency with an unique name. After the Android Gradle plugin was upgraded to version 3.0.0 on October 2017, the configurations of *compile*, *apk* and *provided*[7] are deprecated. Therefore, we investigate to what extent the deprecated configurations exist in Android app projects.




(a) Number of projects containing deprecated configurations on each year

(b) Number of projects updated deprecated configurations at the end of each year.

**Figure 13** Number of repositories related to deprecated configurations.

As presented in Figure 13(a), 1,008 projects are involved with deprecated configurations in 2017. We infer that this might be caused by the projects that were migrated to Gradle in 2017, since we observe that the deprecated configurations of the projects (cf. the first column in Figure 13(b)) were changed to the recommended alternatives. From the end of 2018 to the end of 2022, the number of projects containing the deprecated configuration decreases sharply year by year, and the number of projects whose deprecated configurations are updated is reduced year by year as well. These results show that developers are prone to fix the deprecated configurations in Android app projects in recent years, but there still is a number of unaddressed issues in many projects.

In addition to discussing the version update between Gradle and Gradle plugin, we also try to determine the updates of the detailed Android configurations, such as multiDex, minify, and signing related. In the beginning, we detect if the Android projects have such build-specific configurations. We set every project to the latest commit when we downloaded them and detect if such configurations are specified in the files build.gradle or build.gradle.kt. To be more specific, if the configurations of multiDexEnabled, multiDexKeepFile, or multiDexKeepProguard are designated, we

[7] https://developer.android.com/studio/build/dependencies#dependency_configurations

could conclude that multiDex related configuration is specified. Consequently, the configuration minifyEnabled and proguardFiles describe minify related and the configuration keyAlias, keyPassword, storeFile, and storePassword represent signing-related configuration. Among the total 5,222 projects, we could detect 708, 2,766, 1,402 projects supporting multiDex, minify, and signing related configurations, respectively.

Unexpectedly, only 17 projects have intentionally set the config option minifyEnabled to true, while others were set to false. This is not our expectation as we opted to believe that this config option should be set to true in order to have smaller executables. However, to have a valid configuration for build shrink (i.e., unused code and resources removal), obfuscation (i.e., class and member names length reduction), and optimization (i.e., app size reduction with more advanced strategies) so as to reduce size of the released apps, developers not only need to set minifyEnabled to true but also carefully specify the special classes, methods, and fields in the proguard rules files [40], which is time-consuming and error-prone [41,42,43]. Therefore, Android developers intentionally set minifyEnabled to false to prevent rare potential non-functional bugs.

With regard to configuration multiDexEnabled, it is easy to grasp such attributes. The maximum number of referenced methods in an Android app is 65,536. Once the number of methods exceeds the limitation, the build error message indicating that your app has reached the limit of the Android build architecture would remind developers to set the configuration to true. If the developers finish the development and build, they also need to sign the final APKs with a certificate before release and upload the APKs on an App store. However, the key used for signing is confidential for developers. They need to protect such sensitive information carefully. Therefore, they may upload these projects with sensitive information and signing related configurations excluded intentionally.

**RQ2.3** Finding

For the maintenance of Android app projects, the Gradle plugin (built on Gradle and providing substantial tasks to build) is updated more frequent than the basis Gradle. Upgrading Gradle plugin in projects would urge developers to replace deprecated configurations to gain benefits from the newly recommended ones. Even though the configurations have been deprecated several years ago, some developers are still utilizing such configurations. It is necessary to remind developers abandon the deprecated ones while updating Gradle plugins so as to make fully usage of the new released versions.

## 6 | BUILD SCRIPT QUALITY STUDY (RQ3)

In this section, we answer the research question RQ3 – Can open-source Android app projects be successfully built with their provided build scripts? Collaborators, providing new features and fixing issues for open-source projects, need to be able to successfully build projects automatically with the provided scripts. Constructing their own, often complicated build scripts, would make this very difficult. In addition, in the mobile software engineering community, many state-of-the-art approaches proposed for analyzing Android apps often focus directly on Android APKs, as most Android apps are only released to the community in bytecode (e.g., via Google Play store). To use these approaches, analysts often need first to build the open-source Android apps into APKs, and such builds often need to be done automatically. In addition,

Keeping this requirement in mind, we answer our last research question by checking the quality of the build scripts provided by open-source Android apps. If an Android application cannot be built successfully via the build system, we manually analyze it to determine what are the root causes. To build every single project of the dataset, we constitute our script to build Applications by invoking Gradle wrapper with the following simple command. Ideally, with this command, the build process should be automatically and smoothly completed, if the build environment is properly set up.

```
gradlew assembleDebug
```

By default, there are two different build variants for every Android project: one is in debug mode, and the other is in release mode. Release version intended for app users, while debug mode is better for App developers and researchers. As app developers and researchers could fix more potential issues exposed while building in debug mode compared to release mode to improve the quality of these projects. We focus on Gradle wrapper (instead of Gradle) to build open-source Android apps because Gradle wrapper helps to prepare the correct version of Gradle for building the project. Otherwise, we need to install and set up the correct Gradle version (as specified by the project) to build the project. This process is however tedious as many Gradle versions have been released in the lifetime of Gradle. For example, the Android projects in our dataset, which are developed between 2009 and 2022, require Gradle versions ranging from as early as `v1.0` to the latest `v7.6`.

Many open-source Android projects provide a Gradle wrapper to help developers ease the build process. Among the 4,974 projects considered in our dataset, 4,774 of them (or 95.97%) contain Gradle wrapper scripts. In this research question, we focus on those 4,774 open-source Android app projects to evaluate their provided build scripts' quality. To control the experimental time, we further set a 10 minutes threshold for each build. If a build cannot be done in 10 minutes, we consider it as a failed case.

## 6.1 | RQ3: Can open-source Android app projects be successfully built with their provided build scripts?

Among the 4,774 projects containing Gradle wrapper script, only 1,495 of them can be automatically built, giving a success rate of 31.32%. The fact that more than half of the considered projects cannot be successfully built reveals a serious problem to the community. It questions the quality of the app projects (poorly maintained with problematic code that cannot be compiled) or the app's build scripts, which are not properly configured and hence are invalid to produce the final APK.

Hassan et al.[12] also did a similar study to investigate the auto-build of Java projects with build systems Apache ANT, Apache Maven, and Gradle on 200 Java projects downloaded from Github. They found that 91 of them cannot be built successfully and summarized three types of root causes for these failures. What's more, they tried to auto-repair on the three types of failures and successfully fixed 65 Java projects. Similar to the existing literature, Android projects also contain dependency problems. Unlike traditional Java projects with traditional build systems, Android project build requires much more additional processes, such as NDK build, code shrinking, and signing etc., which indeed could induce build problems and cannot be fixed easily.

To understand the reason why so many of these Android projects cannot be built, we manually analyze and summarize the root causes behind the large number of failures. To do this we manually look at the build logs of such projects that cannot be successfully built. Since it would be very time consuming to go through all the failed projects, we randomly selected 100 projects for this purpose, as done previously. Table 2 summarizes the 5 main categories of build failure reasons we identified. It is worth mentioning that the total number of projects in Table 2 is 95 as the root causes of the remaining 5 projects cannot be well identified. We discuss these separately.

**Table 2** Root causes of the build failure.

| Type | Number | Example |
|---|---|---|
| Source Code | 7 | Lukieoo/DroidCenter: MainActivity.java:54 error: ';' expected |
| Configuration Error | 8 | jp9573/BVM-Alumni: The SDK directory '/home/jay/Android/Sdk' does not exist. |
| Resource File Missing | 49 | NuclearGandhi/blich-android: keystore.properties (No such file or directory) |
| Library Not Available | 26 | Futsal-Manager/android-client: Could not find com.google.firebase:firebasecore:10.0.1 |
| NDK Error | 5 | alexcohn/native-camera2: ndk/16.1.4479499/build/ core/setup-toolchain.mk:172 :*** missing seperator |

The first column presents the error type, while the second column shows the number of the projects that are categorized into that type. The last column provides a concrete example discovered during our manual observation process. In total, there are five well defined types of root causes observed: (1) Source Code Error, (2) Configuration Error, (3) Resource File Missing, (4) Library Not Available, and (5) NDK Error. We now detail these five types, respectively.

- **Source Code Error.** This type describes syntax errors in source code of the projects. For example, when building the `Lukieoo/DroidCenter` project, it yields the following error: "MainActivity.java:54 error: ';' expected".

- **Configuration Error.** This type represents errors caused by the incorrect settings of configuration files. For example, there are some projects with the SDK location hardcoded in their configuration build.gradle file. For example, when building the `jp9573/BVM-Alumni` project, it yields the following error message: "The SDK directory '/home/jay/Android/Sdk' does not exist".

- **Resource File Missing.** This type denotes errors raised by missing certain resource files, such as the keystore property files. App developers could either unintentionally or intentionally introduce missed files in their open-source app projects. Indeed, for the former case, it could simply be the results of human errors, which are known to be hard to avoid. For example, our manual investigation reveals that the `gradle.properties` file of project `Young-Gon/kidtube` is missed in the repository. This resource file has defined several variables that are referenced by the Gradle build file *build.gradle*. As a result, the build script yields a " not found error" when building the project. For the latter case, developers may deliberately ignore such files containing confidential information (e.g., via the Git `.gitignore` file). For example, the project `sfischer13/robot-openthesaurus` misses the property file `keystore.properties` as this file contains information that is specific to each builder. Hence, it should not be publicly released and distributed. As another example, project `abigyani/Ojass18` has a resource file called *google-services.json* missed in the repository. As indicated by the name, this resource file provides configuration information for

the app to access Google services. Since this configuration file may contain sensitive information as well, e.g., the developers' APK KEY allocated by Google, it could be intentionally be removed by developers.

- **Library Not Available.** This type suggests that some dependent libraries cannot be properly located by Gradle at the building time. For example, when building project `Futsal-Manager/android-client`, which requires two firebase-related libraries as indicated in the build script (cf. Figure 8). However, when launching the build script, it will yield the following error message: "Could not find com.google.firebase:10.0.0". Our manual investigation reveals that this problem is caused by the fact that these libraries have now been removed from the Android SDK where they are initially placed. Developers now need to resort to remote Maven repositories [8] to configure these two libraries.

- **NDK Error.** This type depicts errors caused by the Android Native Development Kit (NDK), which is often leveraged to include native code. Some of the projects leveraging NDK produce errors related to CMake or different NDK versions. For example, the `alexcohn/native-camera2` project requires NDK bundle, specifically, NDK revision 14 under folder ndk-bundle, which is not properly set up when building the project. This error is actually introduced by the fact that Google has changed the NDK package structure since 2019. If the latest version of NDK is installed while the build configuration still points to an old NDK version, the build will fail. Such a process seems to be trivial, but it requires developers to go through the whole process (including install the latest NDK version) before recognizing the problem.

The root causes for the remaining 5 projects all do not belong to any of the previous types. For example, the build failure reason for project `pacien/tincapp`. When building these five projects, our script yields the error messages, such as "Json credentials cannot specify a Service Account email while PKCS12 credentials must do so", "No such property: POM_DESCRIPTION for class", "Could not find method packageNameSuffix() for arguments [.debug] on BuildType_Decorated", "Could not get unknown property 'SHOPIFY_ACCESS_TOKEN' for BuildType_Decorated", and "D8: Cannot fit requested classes in a single dex file". We do not classify these into the previous well-defined ones as we co notuld gain any useful insights about the root causes from these error messages.

**RQ3** Finding

Over half of the considered open-source Android app projects cannot be automatically and successfully built based on their provided Gradle wrapper script. There are different reasons causing such failures. In this work, through manual investigation, we have identified five well defined types of root causes that could help app developers avoid such failures in advance.

## 7 | THREATS TO VALIDITY

**External Validity:** The primary threat to the external validity of our work concerns the choice of the selected open-source Android apps. We curate our dataset from the publicly available dataset, AndroZooOpen, which collects open-source projects mostly from Github. The study result would only reflect the status of projects hosting on Github. However, the results could still valid to some extent among Android community as the large-scale dataset is adopted. In addition, we set the building threshold to 10 minutes to reduce time costs. Consequently, the number of successful build projects is influenced by this artificial setting.

**Internal Validity:** The major threat to the internal validity of our study concerns the build system detection and build failures of Android projects. App developers push their projects on Github with file .gitignore. They may intentionally filter out build system specific files, such as .classpath, .project, or accidentally forget to delete the specific files. However, the number of Android projects classified as None and Multiple is quite small. The experiment could still reflect the real status of the open-source Android projects.

**Construct Validity:** The major threat to the construct validity of our study lies in possible errors in the implementation of our experimental scripts and tools. To determine the build techniques of Android projects, we implemented Python scripts to check whether the specific files in Table 1 exist or not. However, there are a few projects that include source code of other open-source libraries, which also contain their own build files. To mitigate this threat, we have carefully reviewed the toolchains and manually validated partial experimental results against selected benchmarks. However, we cannot exclude all the third-party libraries in the projects. We plan to mitigate this in the future.

**Conclusion Validity:** The primary threat to the conclusion validity of our work lies in the size of the sampled dataset, including the number of sampled projects to analyze the relationship between Java source code and build files update, the number of sampled projects to reveal the root causes of auto-build failures. To mitigate this, we resort to the well-known Sample Size Calculator with a confidence level of 95% and a margin

[8] https://firebase.googleblog.com/2017/08/some-updates-to-apps-using-google-play.html

error of 10% to compute the sample size and select the projects randomly. In addition, the external validity also lies in our manual work, such as manually summarizing the root causes of failed Android projects, manually determining the relationship between build files and Java source code. To mitigate this, we cross-validated the manually analyzed results drawn by two different authors.

# 8 | DISCUSSION

We discuss the key potential implications of this work for both practitioners and researchers.

## 8.1 | Implications for Practitioners

**On the Need of Supporting Automated Migration of Build Techniques.** As revealed in the findings of RQ1, build techniques are continuously evolving and Android app developers also continually update their projects to align with the latest build techniques (e.g., from Apache ANT to Apache Maven and from Eclipse ADT to Gradle). Even though, to the best of our knowledge, our community has not proposed automated tools to help developers automatically achieve such a purpose. Often, developers need to manually change the build techniques. Such a process might be time-consuming as developers may not have pre-experience about the new build technique (hence may need time to resolve various issues raised in the process). Furthermore, there is not only a need to migrate build techniques but also a requirement to update the same build technique to its latest versions. For example, when manually observing the failed builds, we find that there are some legacy projects leveraging Gradle under version 1.2, which uses HTTP to download Gradle distributions. However, Gradle services will only respond to requests with HTTPS since January 2020. The requests made with HTTP will be denied. In order to build these legacy projects, we need to upgrade the Gradle version to at least 2.2.1. To free developers from handling such a time-consuming process, we argue that there is a need to invent promising approaches to automatically migrate build techniques in given open-source Android app projects.

**The Necessity of Automating Version Upgrade of the Third-party Libraries.** RQ2 unveiled the necessity of the version upgrade of third-party libraries. The evolution of third-party libraries, such as bug fixes involvement, feature enhancements, enforces app developers to upgrade to latest version in order to take fully advantage of the newly released ones. For example, newly added APIs ease app function development and bug fixes involvement improve the robustness of apps. To speedup the version upgrade, automatic approaches to complete version update are necessary.

**Automated Fix Build Errors.** The fact that only 31.32% of open-source Android apps can be automatically built into APKs shows that many open-source apps cannot directly benefit from existing state-of-the-art app vetting tools to mitigate potential quality or security issues. To mitigate this, we argue that there is a need to propose automated repairing tools to the community to help users directly fix build errors to allow successful builds of such projects that cannot be in the first place.

**On the Need of Protecting Sensitive Information While Achieving Successful Auto-build.** As revealed in the findings of RQ3, resource file missing is the dominant root cause for the auto-build failures. However, some of the missing files are intentionally filtered out by the App developers since these files contain sensitive information, such as the keystore.properties file. To release these projects on Github and guarantee the success of auto-build for other developers and researchers, we argue that new techniques are necessary for developers to protect such sensitive information when uploading to Github while achieving successful build in anywhere.

## 8.2 | Implications for Researchers

**On the Need of Advanced Code Shrinking Techniques.** To shrink the final build of Android projects, App developers need to set minifyEnabled to true. However, there are so many potential bugs induced if they set the item minifyEnabled to true, such as [41,42] etc. Therefore, some of the developers intentionally set minifyEnabled to false to avoid these potential build issues for a smooth build. To provide a better build experience and achieve smaller final installables, more advanced techniques are urgently needed for Android community.

**On the Need of Supporting Continuous Delivery.** The fact that many open-source Android projects cannot be successfully built impedes the research progress of Android researchers. Some of the projects are out of maintenance by the developers because they have changed their focus to other projects. To properly remind developers to update the projects in time, the continuous delivery feature provided by the hosting site Github could be utilised, which would build the projects periodically and inform the author once an error occurred during the build process. With the help of the continuous delivery feature, developers could fix the build error on-time and provide a consistent delivery both in binary and open-source approaches. We, therefore, argue that there is a strong need to adopt continuous integration and delivery into the development process of Android apps.

**Better Tools to Do Error Analysis.** We developed our automatic build error repair tool according to the manual analysis on the randomly selected 100 open-source projects. In order to fix different build errors as much as possible, manual summarization of the root causes of the different

build errors is necessary. However, it is time-consuming and painstaking for researchers. To relieve the burden saddled on open-source Android researchers, better static analysis tools are indispensable. The new proposed tools can be used to not only detect potential build errors lying in Java source code of the project per se but also the build system, such as the library removal from the library repositories (e.g., Maven, Lvy). Thus, we believe that an automatic tool that can be used to detect root causes of the build failure is necessary. By applying these root cause detection tools, different root causes can be considered when we perfect our error build repair artifact.

**Different Projects Investigation.** Build systems leveraged by Android projects are not unique for Android app development. They can also be used to build projects written in other languages, such as C++ and Java etc. Build failures investigation on other projects rather Android applications can also benefit for our researchers to have a better understanding of the build systems achieving a more comprehensive analysis.

## 9 | RELATED WORK

Build systems innate the ability to transform source code implementation to final executable artifacts easing the whole development process. As of the importance of build systems, plenty of researches were conducted by our fellow researchers to investigate the effectiveness and validity of build systems. We now summarize the key works related to our research topic in the area of build systems.

**Java projects.** Plenty of researches focus on the build systems utilized for Java projects as of the popularity of Java programming language. Mcintosh et al.[5] discuss the details of Java build systems, especially the build system Apache ANT and Maven, and argue that the update of the build system always requires the change of the project's source code and also the build system evolves both statically and dynamically with regard to the size and complexity. Hassan et al.[12] study how Java projects can be automatically built with three popular build systems adopted including Apache ANT, Apache Maven, and Gradle. They downloaded 200 Java projects with the highest number of stars on Github and found 91 cannot be built automatically. They also summarized and categorized the root causes of these build failures. Macho et al.[44,45] only focused on build changes from Apache Maven build files. They summarized build change types and found that version changes and dependency changes more frequently than other types and build changes are not equally distributed over the projects' timeline. Shridhar et al.[46] analyzed thirteen Eclipse projects and five Apache projects. They summarized 6 different build change categories, including adaptive, corrective, perfective, preventative, new functionality, and reflective, and concluded that the corrective, adaptive and (to some extent) new functionality changes are the most common, and induce the largest churn and invasiveness in the build system. We, however, focus on the newer build system Gradle, which can also be used to build Java and Android project but has different build semantics compared to Apache ANT and Apache Maven.

**Other projects.** Suvorov et al.[47] analyze the build system migration on K Desktop Environment (KDE) and Linux kernel and find that the build system migration follows the model of spiral. They also summarized four different challenges for software engineers to tackle while performing a build system migration. Gligoric et al.[48] focus on the migration of build systems and provide an automatic approach to migrate any of the build systems to the Microsoft cloud-based build system. They developed an automatic approach and implemented their approach in a tool named Metamorphosis, which reduces the size of the synthesized scripts up to 46%. Kumfert et al.[49] study how big the hidden labor overhead spent on the build infrastructure. In Kumfert et al. survey, the developers spend on average 11.91% of their development time on the build maintenance while the maximum is 35.71%. Robles et al.[50] focus on the relationships and differences between the project core source code and other source artifacts including interface specifications, internationalization and localization modules etc., which the developers use as input to produce the final deliverable. Zaidman et al.[51] study the co-evolution between the core source code of the project and its test and introduce the views of the change history, the growth history and the test quality evolution, which are applied on two open-source systems to figure out the co-evolution to aware with developers and manager alike practitioners. Gall et al.[52] concentrate on large systems to examine the system's building blocks such as modules to reveal the logical obvious or hidden dependencies and change patterns among these modules. Our work, however, concentrates on the upgrade during the evolution of the Gradle-based Android projects.

## 10 | CONCLUSION

Build systems are significant in software construction due to the complexity of modern software. Focusing on the build system of open-source mobile Android applications, we have extensively studied the evolution of the build systems leveraged by Android applications, especially the relatively newer build system Gradle. We found that there are four different auto build technologies used for Anroid open source projects – Apache ANT, Apache Maven, Eclipse ADT, and Gradle. By far the most popular one is Gradle, the default build system recommended by Google when start a new project on the official IDEA Android Studio. Furthermore, not only are developers willing to utilize Gradle as their build system, but some of the developers updated their project's build system to use Gradle. The frequently updated parts of build files can be classified into four different code blocks – classpath, dependency, android, and task. Among the four different categories, the dependencies and android are modified more frequently than other two groups. The Gradle plugin, built on Gradle and responsible for detailed substantial auto tasks, is updated more frequently

than the update of the Gradle *per se*. With the newer Gradle plugin, developers also replace the deprecated configurations so as to make fully usage of the alternative recommended ones. When trying to build all of our open source benchmark apps, we found that over half of the projects cannot be built successfully via their provided Gradle build script. There are five main root causes that we identified to help app developers to avoid such build failures in their released open source projects.

## 11 | ACKNOWLEDGMENTS

This work was supported by the Australian Research Council (ARC) under a Laureate Fellowship project FL190100035, a Discovery Early Career Researcher Award (DECRA) project DE200100016, and a Discovery project DP200100020.

## References

1. Feldman SI. Make—A program for maintaining computer programs. *Software: Practice and experience* 1979; 9(4): 255–265.
2. Nadi S, Holt R. The Linux kernel: A case study of build system variability. *Journal of Software: Evolution and Process* 2014; 26(8): 730–746.
3. Gelle L, Saidi H, Gehani A. Wholly!: a build system for the modern software stack. In: Springer. ; 2018: 242–257.
4. Circle CI. https://circleci.com/; 2022.
5. Mcintosh S, Adams B, Hassan AE. The evolution of Java build systems. *Empirical Software Engineering* 2012; 17(4-5): 578–608.
6. Ståhl D, Bosch J. Modeling continuous integration practice differences in industry software development. *Journal of Systems and Software* 2014; 87: 48–59.
7. Hilton M, Tunnell T, Huang K, Marinov D, Dig D. Usage, costs, and benefits of continuous integration in open-source projects. In: IEEE. ; 2016: 426–437.
8. Shahin M, Babar MA, Zhu L. Continuous integration, delivery and deployment: a systematic review on approaches, tools, challenges and practices. *IEEE Access* 2017; 5: 3909–3943.
9. Humble J, Molesky J. Why enterprises must adopt devops to enable continuous delivery. *Cutter IT Journal* 2011; 24(8): 6.
10. Adams B, De Schutter K, Tromp H, De Meuter W. The evolution of the linux build system. *Electronic Communications of the EASST* 2008; 8.
11. Zadok E. Overhauling Amd for the'00s: A Case Study of GNU Autotools.. In: ; 2002: 287–297.
12. Hassan F, Mostafa S, Lam ES, Wang X. Automatic building of java projects in software repositories: A study on feasibility and challenges. In: IEEE. ; 2017: 38–47.
13. Celik A, Knaust A, Milicevic A, Gligoric M. Build system with lazy retrieval for Java projects. In: ; 2016: 643–654.
14. Liu P, Li L, Zhao Y, Sun X, Grundy J. AndroZooOpen: Collecting Large-scale Open Source Android Apps for the Research Community. *Star* 2018; 1(800): 1300.
15. Pei Liu . Open-Source. https://zenodo.org/record/6975868; 2022. Last viewed: 09-08-2022.
16. Apache Foundation . Apache ANT. https://ant.apache.org/; 2020. Last viewed: 23-Jun-2020.
17. Oliveira Barros dM, Almeida Farzat dF, Travassos GH. Learning from optimization: A case study with Apache Ant. *Information and Software Technology* 2015; 57: 684–704.
18. Oliva GA, Santana FW, Oliveira dKC, Souza dCR, Gerosa MA. Characterizing key developers: a case study with apache ant. In: Springer. ; 2012: 97–112.
19. Miller FP, Vandome AF, McBrewster J. *Apache Maven*. Alpha Press . 2010.

20. Apache Foundation . Apache Maven. https://maven.apache.org/; 2020. Last viewed: 23-Jun-2020.
21. Eclipse Packaging Project . ADT Plugin for Eclipse. http://www.dre.vanderbilt.edu/~schmidt/android/android-4.0/out/target/common/docs/doc-comment-check/sdk/eclipse-adt.html; 2020. Last viewed: 23-Jun-2020.
22. Apache Groovy project . Apache Groovy. https://groovy-lang.org/; 2020. Last viewed: 23-Jun-2020.
23. Kotlin Foundation . Kotlin. https://kotlinlang.org/; 2020. Last viewed: 23-Jun-2020.
24. Gradle Inc. . Gradle. https://gradle.org/; 2020. Last viewed: 23-Jun-2020.
25. AndroidDev . Build. https://developer.android.com/studio/build; 2020. Last viewed: 26-Aug-2020.
26. Gradle Configuration . Gradle Configuration. https://docs.gradle.org/current/dsl/org.gradle.api.artifacts.Configuration.html; 2020. Last viewed: 4-Jun-2022.
27. Mateus BG, Martinez M. An empirical study on quality of Android applications written in Kotlin language. *Empirical Software Engineering* 2019; 24(6): 3356–3393.
28. Flauzino M, Veríssimo J, Terra R, Cirilo E, Durelli VH, Durelli RS. Are you still smelling it? A comparative study between Java and Kotlin language. In: ; 2018: 23–32.
29. Liu P, Xia Q, Liu K, et al. Towards automated Android app internationalisation: An exploratory study. *Journal of Systems and Software* 2023; 197: 111559.
30. Github . GitHub Developer https://docs.github.com/en/rest?apiVersion=2022-11-28; 2023.
31. BennyKok/PxerStudio . https://github.com/BennyKok/PxerStudio; 2023. Last viewed: 3-Feb-2023.
32. Liu P, Sun X, Zhao Y, Liu Y, Grundy J, Li L. A First Look at CI/CD Adoptions in Open-Source Android Apps. In: ; 2022: 1–6.
33. npm package . Google Play Scraper https://www.npmjs.com/package/google-play-scraper; 2023.
34. Google Play Installs . App installs https://cpidroid.com/blog/93/how-google-displays-the-app-installs-count-on-google-play-listing; 2023.
35. McIntosh S, Nagappan M, Adams B, Mockus A, Hassan AE. A large-scale empirical study of the relationship between build technology and build maintenance. *Empirical Software Engineering* 2015; 20(6): 1587–1633.
36. Apache Groovy project . Migration documentation. https://docs.gradle.org/current/userguide/migrating_from_maven.html/; 2022. Last viewed: 22-Mar-2022.
37. Migrate to Android Studio from Eclipse ADT . Migrate to Android Studio. https://developer.android.com/studio/intro/migrate#migrate-eclipse; 2020. Last edited: 9-Jul-2020.
38. Migrate to Android Studio from Apache Maven . Migrate to Gradle. https://docs.gradle.org/current/userguide/migrating_from_maven.html; 2020. Last edited: 9-Jul-2020.
39. A Gradle plugin. https://plugins.gradle.org/plugin/com.getkeepsafe.dexcount; 2021. Last viewed: 24-Dec-2021.
40. specification dA. Shrink, obfuscate, and optimize your app.; 22-May-2020. https://developer.android.com/studio/build/shrink-code#groovy.
41. Exceptions in Android minify . Problem of setting minifyEnabled to true. https://stackoverflow.com/questions/55396131/taskexecutionexception-with-minifyenabled-true-when-building-app; 2020. Last updated: 22-May-2020.
42. minify related bugs in Github . minifyEnabled issues. https://github.com/ionic-team/capacitor/issues/739; 2020. Last updated: 22-May-2020.
43. MinifyEnabled set to True . Problem of setting minifyEnabled to true. https://stackoverflow.com/questions/57630670/using-minifyenabled-true-is-not-ok-in-development; 2020. Last updated: 22-May-2020.
44. Macho C, McIntosh S, Pinzger M. Extracting build changes with builddiff. In: IEEE. ; 2017: 368–378.
45. Macho C, McIntosh S, Pinzger M. Automatically repairing dependency-related build breakage. In: IEEE. ; 2018: 106–117.

46. Shridhar M, Adams B, Khomh F. A qualitative analysis of software build system changes and build ownership styles. In: ; 2014: 1–10.

47. Suvorov R, Nagappan M, Hassan AE, Zou Y, Adams B. An empirical study of build system migrations in practice: Case studies on KDE and the Linux kernel. In: IEEE. ; 2012: 160–169.

48. Gligoric M, Schulte W, Prasad C, Van Velzen D, Narasamdya I, Livshits B. Automated migration of build scripts using dynamic analysis and search-based refactoring. *ACM SIGPLAN Notices* 2014; 49(10): 599–616.

49. Kumfert G, Epperly T. Software in the DOE: The Hidden Overhead of"The Build". tech. rep., Lawrence Livermore National Lab., CA (US); The address: 2002.

50. Robles G, Gonzalez-Barahona JM, Merelo JJ. Beyond source code: the importance of other artifacts in software development (a case study). *Journal of Systems and Software* 2006; 79(9): 1233–1248.

51. Zaidman A, Van Rompaey B, Demeyer S, Van Deursen A. Mining software repositories to study co-evolution of production & test code. In: IEEE. ; 2008: 220–229.

52. Gall H, Hajek K, Jazayeri M. Detection of logical coupling based on product release history. In: IEEE. ; 1998: 190–198.