Guidelines for Architecting Android Apps: A Mixed-Method Empirical Study

Roberto Verdecchia*†, Ivano Malavolta†, Patricia Lago†
*Gran Sasso Science Institute, L'Aquila, Italy
†Vrije Universiteit Amsterdam, Amsterdam, The Netherlands
roberto.verdecchia@gssi.it, {i.malavolta | p.lago}@vu.nl

***Abstract*—For surviving in the highly competitive market of Android apps, it is fundamental for app developers to deliver apps of high quality and with short release times. A well-architected Android app is beneficial for developers, e.g. in terms of maintainability, testability, performance, and avoidance of resource leaks. However, how to properly architect Android apps is still debated and subject to conflicting opinions usually influenced by technological hypes rather than objective evidence.**

In this paper we present an empirical study on how developers architect Android apps, what architectural patterns and practices Android apps are based on, and their potential impact on quality. We apply a mixed-method empirical research design that combines (i) semi-structured interviews with Android practitioners in the field and (ii) a systematic analysis of both the grey (*i.e.,* websites, on-line blogs) and white literature (*i.e.,* academic studies) on the architecture of Android apps. Based on the analysis of the state of the art and practice about architecting Android apps, we systematically extract a set of 42 evidence-based guidelines supporting developers when architecting their Android apps.

I. Introduction

Android is accounting for more than 85.9% of global smartphone sales worldwide [1], leading thousands of developers to choose Android as their first go-to development platform [2]. In the last quarter of 2018 more than 2.6 million Android apps were available in the Google Play, the official Android app store [3].

For surviving in such a highly competitive market, it is fundamental for app developers to deliver apps yielding high quality in terms of *e.g.,* performance, energy consumption, user experience. Developers are investing great efforts to deliver apps of high quality and with short release times. In this context, a well-architected Android app is beneficial for developers in terms of maintainability, evolvability, bug fixing (*e.g.,* resource leaks), testability, performance, etc. The most recent releases of the Android platform are putting more and more emphasis on the architecture of the apps, with a special focus on architecturally-relevant components[1], such as those belonging to Android Jetpack[2], the recently introduced collection of Android software components. However, *how to properly architect Android apps is still highly debated and subject to conflicting opinions*, usually influenced by technological hypes rather than objective evidence.

[1]https://developer.android.com/topic/libraries/architecture
[2]https://developer.android.com/jetpack

The goal of this paper is twofold: (i) to characterize the state of the practice on architecting Android apps and (ii) to provide a set of evidence-based guidelines for supporting developers while architecting Android apps. Given the relatively low maturity of the subject and its tight connection with industry, we apply a *mixed-method empirical research design* that combines (i) semi-structured interviews with Android practitioners in the field, and (ii) a systematic analysis of both the grey (*e.g.,* websites, on-line blogs, etc.) and white literature (*i.e.,* academic studies) on the architecture of Android apps. Specifically, starting from 5 interview transcripts and an initial set of 306 potentially-relevant primary studies, through a rigorously-defined and replicable process, we select 44 data points, *i.e.,* either interview transcripts or primary studies belonging to the grey/white literature. We analyze each data point in order to characterize how developers architect Android apps, what architectural patterns and practices Android apps are based on, and their potential impact on quality attributes such as maintainability. Finally, a set of 42 guidelines for architecting Android apps is systematically synthesized from the obtained practices. The emerging guidelines are organized around 4 themes including the most adopted architectural patterns and principles when developing Android apps (*e.g.,* Model-View-ViewModel[3]). The main **contributions** of this study are:

- interviews of 5 practitioners that provide qualitative information about architecting Android apps;
- a systematic analysis of the grey and white literature about architecting Android apps;
- a set of 42 evidence-based guidelines for architecting Android apps;
- the replication package of the study.

The **target audience** of this paper includes both Android developers and researchers. Specifically, this study benefits (i) developers by providing evidence-based guidelines for taking action towards improving the architecture of their Android apps, and (ii) researchers by objectively characterizing the state of the art and practice about architecting Android apps.

The remainder of the paper is organized as follows. The design of this study is presented in Section II, followed by the reporting and discussion of the main results in Section III. Threats to validity and related work are described in Sections IV and V, respectively. Section VI closes the paper.

[3]https://developer.android.com/topic/libraries/architecture/viewmodel

II. Study Design

In this section we report the research questions (Section II-A) and the steps of our mixed-method study (Section II-B).

A. Research questions

RQ_1: *Which are the* ***general characteristics of the architecture of Android apps****?* This question can be refined into:

$RQ_{1.1}$: *Which* ***architectural patterns*** *are considered for architecting Android apps?* This research question aims at understanding which architectural styles/patterns[4] are considered for architecting Android apps. This provides us with a better understanding of current Android architecture practices, and sets the context for the next research questions.

$RQ_{1.2}$: *Which* ***libraries*** *are referenced while considering the architecture of Android apps?* With this question we aim at identifying the programming libraries regarded as most influential while architecting Android apps. This provides further data on the technologies considered in order to support Android architecting processes.

RQ_2: *How to* ***guide developers*** *when architecting Android apps?* This research question constitutes the core of the study. By answering it, we aim at synthesizing a set of architectural guidelines for architecting Android apps. This provides practitioners with actionable guidance when architecting their Android apps, and supports researchers in future investigations on Android architecture.

RQ_3: *Which* ***quality requirements*** *are considered when developing and reasoning about the architecture of Android apps?* Today developing apps of high quality is fundamental for surviving in the Android market. This research question aims to understand which quality requirements (QRs [4], *e.g.,* performance, usability, maintainability) are taken into account when dealing with the architecture of Android apps. This provides a good understanding of which QRs are potentially impacted the most by architectural decisions.

B. Research Method

In order to answer the research questions, we adopt a *mixed-method* approach consisting of a *multivocal literature review* [5] integrated with the results of semi-structured interviews with Android practitioners. An overview of the entire process is shown in Figure 1, while Figure 2 shows the number of selected data points in each step. The remainder of this section describes the key individual steps, Step 2a through 4.

1) Multivocal Literature Review (MLR): The literature review is performed by rigorously following well established guidelines for conducting software engineering literature reviews [6], [7]. The guidelines are complemented by additional ones specifically targeted for the inclusion of grey literature in multivocal studies [8]. To have full control over the number and quality of the literature considered, the literature review is designed as a multi-stage process, reported below.

Initial Search. In order to identify the potentially relevant white literature (WL) studies, a research query is executed on Google Scholar. We opt for such digital library as (i) its adoption constitutes a sound choice to determine the initial set of literature for snowballing processes [9], (ii) from a preliminary execution of the search query it results to be more inclusive w.r.t. Scopus and IEEE Explore, and (iii) the results of the query can be processed automatically via tool-support.

Listing 1 shows the search string we use. The query is purposely designed to be generic, in order to be as encompassing as possible while selecting a significant set of potentially relevant studies. The execution of the query for the WL returns 206 hits. Regarding the initial search of grey literature (GL), the query reported in Listing 1 is executed on the regular Google Search Engine by omitting Google Scholar specific syntax (*i.e.,* the "intitle" keywords). The search engine is selected in accordance to the recommendations for including GL in software engineering multivocal reviews [8]. Due to the high volume of returned results, we limit the search to the top 100 results as stopping rule. This number proves also to be the theoretical saturation point [8] of the returned results.

```
(intitle:architecture OR intitle:architectural
 OR intitle:architect OR intitle:architecting)
 AND (intitle:android OR intitle:"mobile app")
```

Listing 1. Search query used for automated search of white literature

Application of Selection Criteria. The search results are then filtered in order to obtain an initial set of primary studies by applying a set of well-defined inclusion and exclusion criteria. In order to systematically control the quality of the primary studies, three sets of criteria are defined: one general (*i.e.,* applying for both WL and GL), one specific to WL, and one specific to GL. A paper is included only if it satisfies all inclusion criteria and none of the exclusion ones.

TABLE I
Selection criteria for the multivocal literature review

Type	Description
General-Inclusion	Studies focusing / software architecture
General-Inclusion	Studies focusing on design or development of Android apps
General-Exclusion	Studies not published in English
General-Exclusion	Duplicate or extensions of already included studies
General-Exclusion	Studies which are not available
General-Exclusion	Studies not focusing on native Android applications, *e.g.,* Unity-based videogames, web-based apps
WL-Exclusion	Secondary or tertiary studies
WL-Exclusion	Studies in the form of editorials, tutorials, books, etc.
WL-Exclusion	Studies which have not been peer reviewed
GL-Exclusion	Studies reporting exclusively the basic principles about the Android platform and its architecture
GL-Exclusion	Studies reporting exclusively abstract best practices
GL-Exclusion	Studies reporting only trivial Android implementations
GL-Exclusion	Studies reporting an implementation without a discussion of its benefits and/or drawbacks
GL-Exclusion	Studies written for promotional purposes
GL-Exclusion	White literature
GL-Exclusion	Videos, webinars, etc.
GL-Quality	Is the publishing organization or the author reputable?
GL-Quality	Has the author published other studies in the field?
GL-Quality	Does the study add value to the research?
GL-Quality	Is the presentation of the study of high quality?
GL-Quality	Is the study supported by evidence, *e.g.,* examples/data?

In order to further ensure the quality of GL primary studies, a subset of 5 quality-evaluation criteria presented by Garousi

[4]For the sake of space, from this point onwards, "architectural style" and "architectural pattern" will be jointly referred to as "architectural pattern".

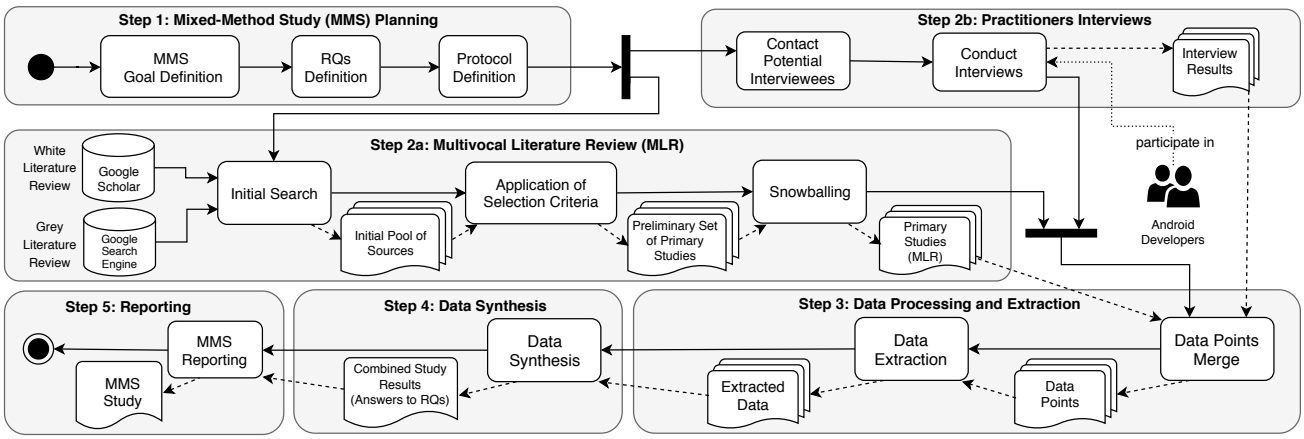

Fig. 1. Mixed-Method Study (MMS): process overview

et al. is adopted [8]. A 3-point Likert scale (yes=1, partly=0.5, and no=0) is used to assign the quality scores. A GL study is considered of sufficient quality if it scores at least 2 out of 5 total points. Table I reports the considered selection criteria.

As recommended in [10], two researchers inspect a random sample of the studies. For assessing the objectivity of this phase the inter-researcher agreement is measured, achieving a substantial agreement Cohen Kappa value of 0.79, falling slightly below the one recommended in [11], equal to 0.80.

The application of the criteria terminates with the inclusion of 6 WL primary studies and 16 GL primary studies.

Snowballing Process. Once the preliminary set of primary studies is defined, we conduct a snowballing process. Regarding WL, we adopt a standard iterative backward and forward snowballing process [9]. During this process, 629 potential relevant studies are analyzed, leading to the inclusion of 1 additional primary study. For the GL, due to the high volume of primary studies to be considered, we limit the snowballing to the links referenced in the GL primary studies (*i.e.,* we do not consider backward links), and stop the process after the first iteration. A totality of 16 new primary studies is included through this snowballing process.

2) Practitioners Interviews: In order to complement the data extracted from the MLR, we conduct semi-structured interviews with Android practitioners. This step consists in designing a survey based on our research questions, contacting potential interviewees, conducting the semi-structured interviews, and post-processing the interview results. Interviewees are selected via convenience sampling by exploiting our collaboration network. Developers are required to be affiliated to different companies, belong to distinct business domains, and possess at least 5 years of Android development experience. Out of 18 contacted practitioners, 5 result to be available for the interview. Interviews range between 30 and 60 minutes.

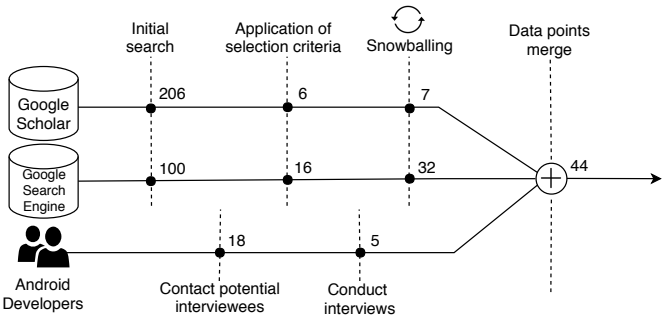

Fig. 2. Steps of the Mixed-Method Study (MMS)

3) Data Processing and Extraction: After the collection of the WL/GL primary studies and survey results, referred jointly as *data points*, the resulting 44 data points are uniformed and merged into a single pool. Subsequently, we extract from the data points the information necessary in order to answer our research questions. The data necessary to answer RQ_1 is extracted by inspecting the data points, and subsequently identifying which architectural patterns and libraries are considered. In order to extract architectural practices (RQ_2) we conduct iterative content analysis sessions [12] involving two authors of the paper. Finally, to answer RQ_3, the data points are inspected to extract which QRs are deemed to be impacted by architectural decisions. The entirety of the data extracted is mapped to the originating data points for the sake of backward traceability and replication purposes.

4) Data Synthesis: Finally, the extracted data is processed and synthesized in order to answer our research questions. In order to answer RQ_1 and RQ_3 the extracted data can be processed quantitatively. Differently, in order to answer RQ_2, we carry out a keywording process [13]. This process consists of grouping the architectural practices extracted from the data points according to their semantic similarity. This is achieved by labelling the single practices with representative keywords. The process is iterated by refining the keywords, till the grouped practices can be merged into a single guideline. Practices mapped to a guideline in this fashion are referred to as *supporting practices*. The first round of keywording leads to the identification of the general themes considered, while the subsequent ones to the formulation of the guidelines. Figure 3 summarizes the relationships between the elements of the resulting catalogue of guidelines.

For more details on the research method, research execution, and extracted data, we refer the reader to the replication

package of this study[5]. The package is made available with the aim of supporting independent verification and replication. It contains (i) the rigorous research protocol defined *a priori* which we follow, (ii) the entirety of the search and selection execution data, (iii) the raw data extracted from the data points, and (iv) the documentation of data analysis processes accompanied by the relative results.

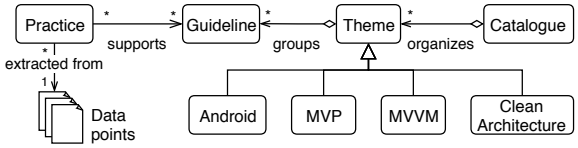

Fig. 3. Relationship between the elements of the catalogue of guidelines

III. Results

A. RQ_1: Characteristics of Architecting Android Apps

1) Android architectural patterns ($RQ_{1.1}$): The inspection of the extracted data lead to the identification of 7 architectural patterns considered when developing Android apps.

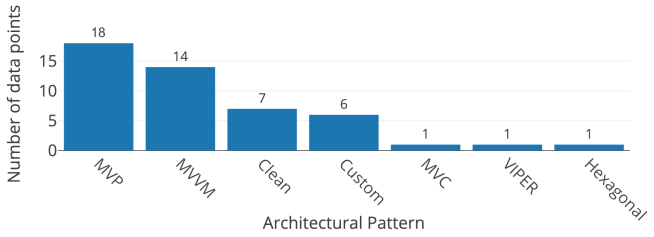

Fig. 4. Overview of architectural pattern recurrence

As shown in Figure 4, the most recurrent pattern results to be Model-View-Presenter (MVP), which is reported in 18 data points. Model-View-ViewModel (MVVM) results the second most frequent pattern. This could be associated to the recent introduction by Google of the `ViewModel` architectural component[6]. Due to its potentially drastic impact on Android architecture development, we expect the MVVM pattern to experience a fast growing trend of adoption in the coming years. Clean architecture principles [14] appear also to be frequently considered in the context of architecting Android apps. Some data points also report *ad-hoc* custom solutions, such as MVP extended through manager classes, message-driven architectures, and architectures heavily relying on RxJava. Nevertheless, such architectural solutions appear to be fragmented and slightly less popular. Other architectural patterns, *i.e.,*, Model-View-Controller (MVC), View-Interactor-Presenter-Entity-Router (VIPER) [15], and hexagonal architecture [16], result to be only scarcely considered.

> **Findings for $RQ_{1.1}$ (Android Architecture patterns):** MVP is the most considered pattern, followed by MVVM. Clean architecture principles applied to Android apps are also frequently discussed. Heterogeneous *ad-hoc* solutions are also considered.

[5]https://github.com/AndroidGuidelines/ReplicationPackage
[6]https://android-developers.googleblog.com/2017/05/android-and-architecture.html

2) Libraries considered while architecting apps ($RQ_{1.2}$): An overview of the most recurrent libraries referenced when discussing Android architecture are reported in Figure 5.

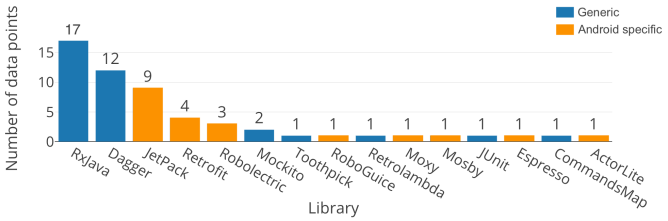

Fig. 5. Overview of library recurrence

Not surprisingly, RxJava[7] is the most mentioned library. RxJava enables a crucial programming paradigm for mobile apps, namely reactive programming. By adopting reactive programming, it is possible to efficiently deal with concurrency and asynchronous tasks, which are inherent to the mobile context. The second most recurrent library is Dagger[8], a framework maintained by Google which implements the dependency injection pattern. This library constitutes a popular choice in order to manage dependencies, and potentially avoids unnecessary boilerplate code. JetPack, a recently released official Android library focusing on architectural components, is less popular. While the architectural relevance of such library in the Android ecosystem is clear, the lower occurrence of such library can be attributed to its recent release, and hence to the time required for its adoption. The other referenced libraries, like Retrofit, Robolectric, Mockito, are less recurrent, potentially due to their lower architectural relevance, hence pointing to a well scoped selection of data points. Interestingly, only 8 out of the 15 libraries reported are explicitly conceived for Android (see Figure 5). This shows that the Android architecture is "open", *i.e.,* influenced by many generic libraries. In addition, of the Android specific libraries, only a few focus on architecture (JetPack, Moxy[9], and Mosby[10]).

> **Findings for $RQ_{1.2}$ (Architecturally relevant libraries):** RxJava is the most referenced library, followed by Dagger, JetPack, and Retrofit. Approximately half of the libraries are Android-specific, while only few focus specifically on Android architecture.

B. RQ_2: Android Architecture Guidelines

In order to answer RQ_2, a total of 212 architectural practices are extracted from the selected data points. The first round of keywording leads to the classification of the practices into four emergent themes: *general Android architecture*, *MVP*, *MVVM*, and *Clean Architecture*. By applying recursively the keywording process, the practices are further clustered, till the synthesis of 42 architectural guidelines, reported in Table II. As proxy for maturity of the guidelines, we utilize the number

[7]https://github.com/ReactiveX/RxJava
[8]https://google.github.io/dagger/
[9]https://github.com/Arello-Mobile/Moxy
[10]https://github.com/sockeqwe/mosby

TABLE II
GUIDELINES FOR ARCHITECTING ANDROID APPS (ID = GUIDELINE IDENTIFIER, #SP = NUMBER OF SUPPORTING PRACTICES)

ID	#SP	Architectural guideline
		Generic for Android
A-1	18	Decouple components and explicitly inject/manage the dependencies among them.
A-2	17	Design components to be as independent as possible, build them around the features of the app and make them Android-independent.
A-3	16	Counter the tendency of Activities to grow too big in size due to functionality/responsibility bloat.
A-4	14	Strive towards separation of concerns in your architecture, where each component has well defined responsibility boundaries, a purpose, (set of) functionality, and configuration.
A-5	10	When starting a new project, carefully select a fitting architectural pattern to adhere to.
A-6	8	Organize your Java/Kotlin packages and files either by layer or by app feature.
A-7	7	Take full advantage of libraries. Do not try to reinvent the wheel and loose time by implementing boilerplate code. Focus on what makes your app stand out from the rest and delegate what is left to libraries.
A-8	7	Locally cache data for supporting offline-first experience.
A-9	6	Use exclusively interfaces to let app modules communicate. This protects the architectural structure and helps defining a clear responsibility of modules.
A-10	5	Avoid nested callbacks, as they could lead to a "callback hell". Approximatively, more than 2 levels of callbacks are considered to reduce maintainability and understandability. This problem is commonly fixed by taking advantage of the RxJava library.
A-11	4	Employ well-defined and accepted coding standards, as they improve both code understandability and maintainability.
A-12	3	Use a dedicated module to persist as much relevant data as possible. This data source should be the single source of truth driving the UI.
A-13	3	Take into consideration the lifecycle of Android components (*e.g.,* Activities and Services) – also with respect to other components – and design them as short-lived entities.
A-14	1	Have special care in designing background tasks, especially by considering the apps' lifecycle.
A-15	1	Use permissions consistently. Every component of an app that has a permission must be declared also at the app level.
		MVP-specific
MVP-1	9	Provide Views with data which is ready to be displayed.
MVP-2	5	Presenters should be Android- and framework-independent.
MVP-3	5	Access (and cache) the data provided by Models via app-scoped dedicated components.
MVP-4	4	Clearly define contracts between Views and Presenters.
MVP-5	4	The lifecycle of Presenters should follow the lifecycle of the Views, but not by replicating the complexity of the lifecycles of Android components.
MVP-6	3	Avoid to delegate too many responsibilities to Presenters, as they have the tendency to become bloat classes.
MVP-7	2	Make Presenters dependent on Views, and not Activities.
MVP-8	2	Views are passive and should always manage and expose only their state.[11]
MVP-9	2	Strive towards putting as much of the app's business logic as possible in Presenters.
MVP-10	2	Inject dependencies to Presenters into the Views when instantiating the Presenters, as this reduces coupling issues and null checks.
MVP-11	1	If an app has multiple Presenters, do not let them communicate with each other.
MVP-12	1	If necessary, retain fragments for avoiding memory leaks due to configuration changes in the activities.
		MVVM-specific
MVVM-1	5	Models, Views, and ViewModels should exclusively expose their state instead of state and data separately.
MVVM-2	4	The app should possess a single source of truth of data.
MVVM-3	3	Models should be evolvable/testable independently from the rest of the app.
MVVM-4	3	ViewModels should not refer to View-specific components.
MVVM-5	2	Views should always know about changes after ViewModels, no matter how trivial an operation may be.
MVVM-6	2	Adopt one Model for each feature of the app.
MVVM-7	2	Keep ViewModels as simple as possible. When needed, transfer responsibility to other layers, *e.g.,* Models or other components such as data transformers, components factories, etc.
MVVM-8	1	The state of the app should be defined in the Models only, whereas Views and ViewModels should be stateless.
MVVM-9	1	The data produced by the Models should be reliable and of high quality.
MVVM-10	1	Networking or data access functionalities should be performed exclusively by Models.
		Clean Architecture
CLEAN-1	13	Business logic should be completely decoupled from the Android framework.
CLEAN-2	5	The outer architectural layer should contain the entirety of the app's UI components.
CLEAN-3	4	The framework and devices layer should include the entirety of the app components which depend on Android.
CLEAN-4	4	Each architectural layer should possess its own data model.
CLEAN-5	2	Keep the UI thread as lightweight and isolated as possible.

of supporting practices (SP) of each guideline. Due to space limitations in the remainder of this section we document only the top-5 guidelines for each theme.

1) General Android Architecture Guidelines: The first and most recurrent theme regards generic architectural practices for Android, *i.e.,* not specific to any particular architectural pattern. In total, 120 practices are collected and synthesized into the following 15 general Android architecture guidelines.

A-1: "Decouple components and explicitly inject/manage the dependencies among them". While not strictly necessary, utilizing a dependency injection framework can drastically simplify the management of dependencies between Android architectural components. This supports a clean decoupling of architectural components and avoids unnecessary boilerplate for connecting them. Doing so not only improves the maintainability of the app, but also improves its testability by providing the possibility to inject mock implementations. The Dagger framework is commonly recommended to inject dependencies and solve problems afflicting reflection-based solutions.

A-2: "Design components as independent entities as possible, build them around the features of the app and make them Android-independent". As also remarked by two interviewees, a recurrent problem arises when common functionalities are not provided in base classes. This often leads to duplicated code, reducing the maintainability and testability of the app. Ideally, components should be independent from each other and their business logic should be clear and explicitly

[11]This guideline applies also for MVVM with the support of 3 practices.

separated. By quoting one of the data points "your architecture should scream the purpose of the app". Decoupled components make it easier to focus on app functionalities and their issues, without dealing with bloatware. Additionally, this enables a higher testability of the core logic of the app by making components unit-testable (ideally without requiring an emulator). Finally, by decoupling the business logic from frameworks, more emphasis is put on the business logic, making an app more testable, maintainable, and of low technical debt.

A-3: "Counter the tendency of Activities to grow too big in size due to functionality/responsibility bloat". Android Activitiesshould ideally contain exclusively logic handling the user interface (UI) and operating system interactions. Nevertheless, a common architectural issue consists of delegating too many functionalities and responsibilities to a single Activity. This leads to Activities slowly becoming god-classes. As the Android framework does not support the reuse of methods implemented in activities, code tends to be directly copied into other ones, increasing code duplication and impacting negatively the app's maintainability. Additionally, testing might become a challenging task, as complex business logic could reside in Activities, which by themselves result arduous to unit test. Finally, as activities are kept in memory at runtime, "god-activities" can lead to the deterioration of apps' performance.

A-4: "Strive towards separation of concerns in your architecture, where each component has well defined responsibility boundaries, a purpose, (set of) functionality, and configuration". Architectural components of an app should have a single, well defined, responsibility. As a component grows bigger, it should be split up. By following the single responsibility principle, the app architecture naturally supports the structure of developer teams and development stages. Additionally, monoliths are detected in the early stages and modules become testable in isolation. Finally, if the app is built using Gradle, modularization can improve the performance of the build process and ease the development of Instant apps[12]. It is important to notice that, while modularization may imply little effort if considered early in the project, it might become an extremely expensive process in later development stages.

A-5: "When starting a new project, carefully select a fitting architectural pattern to adhere to". Picking the right architectural pattern (*e.g.,* MVP or MVVM) for the context and business goals of the app is a crucial decision. By adhering to an architectural patter selected *a priori*, separating responsibilities into components becomes a more straightforward process, and the growth of architectural technical debt is hindered. It is important to note that, when a certain level of adaptability/maintainability is not required, the selection of an ill-suited architectural pattern might lead to over-engineering. Choosing the architectural pattern to adopt is hence a non-trivial decision which should be taken by considering the context of apps, and their business/organizational goals.

2) MVP-specific Android Architecture Guidelines: The second most considered theme regards practices related to the MVP architectural pattern. In total, 40 practices are collected and synthesized in the following 12 MVP-specific guidelines.

MVP-1: "Provide Views with data which is ready to be displayed". The view layer of Android apps tends to become bloated with responsibilities, and hence becomes harder to maintain. In order to alleviate such problem, Activities and Fragments can be provided with preprocessed data ready to be displayed. This can be achieved by delegating data-processing tasks to one or more dedicated components. In such manner, Activities and Fragmentsare relieved from the task of transforming and filtering domain-specific data, potentially improving the testability and usability of the app.

MVP-2: "Presenters should be Android- and framework-independent.". To abstract Presenter components from the implementation details, Presenters should ideally avoid dependencies to the Android framework. This also entails not creating a lifecycle in Presenters, as it may hinder their maintainability and evolvability. In order to access app resources and preferences, View and Model components can be used instead, respectively. Additionally, by developing Presenters as only dependent on Java, the testability of Presenters drastically improves, as now non-instrumented unit-test cases can be written for such components.

MVP-3: "Access (and cache) the data provided by Models via app-scoped dedicated components". When developing an Android app, a common issue which might emerge is related to restoring the state of Views. This issue can be solved by adapting slightly the architecture of apps. Specifically a data manager component (*e.g.,* a data store or Jetpack Repository) can be introduced. This component is responsible for data related tasks such as fetching data from the network, caching results or returning already cached data. By scoping such component at the app level and not at the one of single Activities, issues relative to restoring View states, *e.g.,* in the occurrence of a screen orientation change, are solved through an architecturally maintainable solution.

MVP-4: "Clearly define contracts between the Views and the Presenters". Before starting to develop a new app feature, a good architectural practice consists in writing a contract documenting the communication between the View and the Presenter. The contract should document for each event in the View which is the corresponding action in the Presenter. By implementing contract interface classes, the source code of apps become more understandable, as the relation between the View and the Presenter is explicitly documented.

MVP-5: "The lifecycle of Presenters should follow the lifecycle of the Views, but not by replicating the complexity of the lifecycles of Android components". By having callbacks related to the Activity lifecycle in Presenters, Presenters become tightly coupled to Activities lifecycle. This can have a negative impact in terms of maintainability. From an architectural perspective, Presenters should not be responsible for data-related tasks. It is hence advised not to retain Presenters. An alternative solution would be to use a caching mechanism to retain data, keep Presenters stateless, and destroy Presenters when their corresponding Views are destroyed.

[12]https://developer.android.com/topic/google-play-instant

3) MVVM-specific Android Architecture Guidelines: Another recurrent theme identified in the practices regards the MVVM pattern. In total, 24 practices are collected and synthesized in the following 10 MVP-specific guidelines.

MVVM-1: "Models, Views, and ViewModels should exclusively expose their state instead of events or data separately". For example, to ensure that Views display up-to-date content, it is recommended that ViewModels expose states rather than just events. This can be achieved by bundling together the data that needs to be displayed. In such way, when one of the fields to be displayed changes, a new state is emitted and the View is updated. This entails that each user interaction involves an action in the ViewModel, enabling a clean separation of concerns between MVVM components.

MVVM-2: "The app should possess a single source of truth of data". In the context of mobile applications, consistency of data can become an issue. While caching mechanisms allow to save energy and bandwidth, multiple data sources can create inconsistencies and even conflicting Views. In order to avoid such issues, it is recommended to designate a dedicated component as single source of truth for the entire app. Specifically, in the context of MVVM, the Room persistence library[13] is an official architectural component of Android which is specifically tailored for such task.

MVVM-3: "Models should be evolvable/testable independently from the rest of the app". Well-designed ViewModels should completely decouple Views from Model classes. In such way, by strictly adhering to the MVVM pattern, Models and Views can evolve independently and be tested with ease. Additionally, by applying the inversion of control principle and implementing ViewModels decoupled from the Android framework, it is possible to test ViewModels via unit tests. In contrast, if the binding between the MVVM components is too complex and intertwined, testing and debugging Android apps can become a cumbersome challenge.

MVVM-4: "ViewModels should not refer to View-specific components". Passing context to ViewModel instances can result in a dangerous practice. In fact by storing the reference to an Activity in a ViewModel, once the Activity gets destroyed (*e.g.,* due to a screen rotation), a memory leak could occur. By quoting a Google Android Developer Advocate: "*The consumer of the data should know about the producer, but the producer - the ViewModel - doesn't know, and doesn't care, who consumes the data.*"[14]. In order to adhere to this guideline, the LiveData[15] architectural class provided by the Jetpack library can be used, so that Activities can simply observe the changes of the ViewModel's data.

MVVM-5: "Views should always know about changes after ViewModels, no matter how trivial an operation may be". Adhering to this guideline implies that all the logic in the Views should be moved to the ViewModels. While the purpose of ViewModels is to pre-process data to be ready to use by Views, it might be tempting to implement minor operations in Views. Nevertheless, adhering to this guideline guarantees a higher level of consistency and reliability of all the components which are based on the ViewModels.

4) Clean Architecture Android Guidelines: While rather a set of architectural practices than a pattern, clean architecture results to be a common theme in Android architecture literature. In total, 28 practices on this theme are collected and synthesized into the following 5 architectural guidelines.

CLEAN-1: "Business logic should be completely decoupled from the Android framework". By adhering to the clean architecture principles, the innermost layers of an app (*i.e.,* where all the business logic of the app resides) should be "frontend agnostic". This means that this layers are completely decoupled from the Android framework, and could be ideally implemented as pure Java packages. Additionally, as this layers represent the core of Android apps, they should be developed before all other layers. Changes to the innermost layers should be driven exclusively by business decisions.

CLEAN-2: "The outer architectural layer should contain the entirety of the app's UI components". In order to ensure a clear separation of concerns among the clean architecture layers of an app, it is paramount that everything related to Android UI is grouped in a module residing in the outer architectural layer. As the other architectural layers of the app should be "frontend agnostic" (see guideline CLEAN-1), different patterns (*e.g.,* MVP or MVVM) can be implemented in this layer. Activities and Fragments should not handle any other logic than the one necessary to render the UI. This allows a clear separation of concern among the architectural layers of an app, enhancing the understandability, modifiability and testability of its components.

CLEAN-3: "The framework and devices layer should include the entirety of the app components which depend on Android". In accordance to the clean architecture principles, all components related to the framework should be grouped in the outer architectural layer. This includes all components which contain Android specific implementations, which should not be present in the business logic layers. Examples include, in addition to the user interface model, the data persistence module (*e.g.,* LiveData, DAOs, ORMs, Shared Preferences, Retrofit, etc.) and eventual dependency injection frameworks.

"CLEAN-4: Each architectural layer should possess its own data model". By implementing a data model at every layer, a high degree of decoupling between layers can be achieved. Specifically, by following this guideline, the outer layers of apps can be implemented without any explicit knowledge of the implementation details of the inner layers. This means that the origin of the data becomes transparent to the client and hence, in a repository pattern fashion, data sources can be added, removed, or changed without much effort.

CLEAN-5: "Keep the UI thread as lightweight and isolated as possible". In accordance to guidelines CLEAN-1 and CLEAN-2, presenters residing in the outer layers of apps modeled through clean architecture principles should

[13]https://developer.android.com/jetpack/arch/room

[14]https://medium.com/upday-devs/android-architecture-patterns-part-3-model-view-viewmodel-e7eeee76b73b

[15]https://developer.android.com/topic/libraries/architecture/livedata

be kept lightweight. In fact, Presenters should be composed with interactor components, *i.e.,* use cases residing in the business logic layers, which are responsible for executing tasks outside the main UI thread of the app. Once the task are finished, the Views are updated through a callback with the processed data. Besides callback-based communication among components, other techniques used in order to keep the UI thread lightweight rely on the inversion of control principle and intent-based communication.

Throughout this study we notice that low-level technical concerns (*e.g.,* management of screen rotation, access to sensors) are often intertwined with architectural concerns (*e.g.,* how to structure the whole app, how UI events should flow within the app) without a clear separation between them. Our emerging guidelines can help Android developers in starting to (i) abstract from the low-level details of the app, and (ii) reason on its overall structure and related architectural concerns.

Finally, it must be noted that the identified guidelines should be seen as recommendations, rather than strict rules to be followed at any cost. Indeed, by quoting one of our data points: *"No architecture is perfect for every use-case and Architecture Guidelines are just recommendations. Hence [developer] feel free to use whatever suites your use-case."*. In other words, we advise developers not to consider the guidelines provided in Table II as a whole, but rather to reflect carefully on which ones should be applied in their projects, depending on the current technical, business, and organizational context.

Findings for RQ_2 (Android Architectural Guidelines): 212 architectural practices are extracted and synthesized into 42 architectural guidelines, reported in Table II. Four main overarching themes emerge from the guidelines: *generic Android architecture guidelines*, *MVP-specific*, *MVVM-specific*, and *Clean Architecture*. The 5 most mature guidelines per theme are detailed, while the remaining are documented in Table II.

C. RQ_3: Quality Requirements

Figure 6 shows the QRs considered when architecting Android apps. We observe that maintainability, testability, and performance are the highest ranking ones.

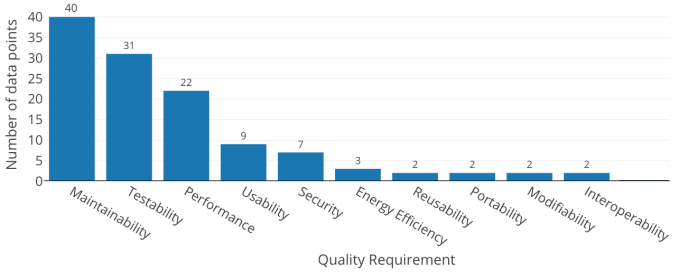

Fig. 6. Quality Requirements considered while architecting Android apps

Overall, the results gathered for RQ_3 are in line with what is intuitively evinced by inspecting the results of RQ_2. In fact, a high number of guidelines deal with modularization, separation of concerns, and deal with components size coupling, etc. The application of such principles impacts primarily maintainability and testability, as discussed in Section III-B. Interestingly, those principles are also strongly related to the maintainability issues frequently occurring over the lifetime of Android apps [17]. Additionally, guidelines such as avoiding "god-activities" (A-3) or use specific data management solutions (MVP-3) can drastically impact the performance of apps. Architectural guidelines influencing primarily other QRs are overall less frequent.

Moreover, we observe that most quality requirements regard development-time QRs (*e.g.,* maintainability, testability) rather than runtime attributes (*e.g.,* performance, energy). The focus on static QRs in Android emerges also from the guidelines reported in Section III-B. In fact, most of the synthetized guidelines consider development-time architectural views of apps (*e.g.,* A-1) rather than runtime ones (*e.g.,* A-8).

Findings for RQ_3 (Android Architecture Quality Requirements): Maintainability, testability, and performance are the most considered QRs when architecting Android apps. Most QRs regard development-time attributes.

IV. Threats to Validity

External Validity. The primary threat to this category is represented by the selection of the data points, which might not be representative of the state of the art and practice. To mitigate this threat, we adopt 3 different data sources (semi-structured interviews, WL, and GL). This leads to a more heterogeneous set of data points for our study. Additionally, to ensure the soundness and quality of the MLR data points, a thorough selection and quality evaluation process is conducted via a set of well-defined evaluation criteria. To identify the interviewees, convenience sampling is adopted. This constitutes a threat to external validity, mitigated by selecting interviewees that resulted heterogeneous in terms of type of apps developed, company, background and developer role.

Internal Validity. To mitigate potential threats to internal validity, we follow a rigorous research protocol defined *a priori*. To avoid biases related to data collection through semi-structured interviews, we perform such step prior to the MLR execution and by following an interview guide as part of the protocol. Internal validity threats of the MLR are mitigated by following established guidelines for conducting WL reviews [6], [7] integrated with guidelines for the inclusion of GL [8].

Construct Validity. The most prominent threat to construct validity regards the potential inappropriateness of data point selection. To mitigate this treat we use multiple data sources. As suggested by Wholin *et al.* [10], the quality of the MLR selection process is ensured by measuring inter-researcher agreement on a random subsample of potentially relevant studies. Additionally: (i) we perform the MLR by adhering to well-documented search and selection processes predefined in a rigorous protocol, and (ii) the semi-structured interviews are

conducted exclusively with developers with at least 5 years of experience. The adoption of Google Scholar and Google Search Engine might constitute a bias due to their underlying algorithms. We mitigate this threat by using well-defined selection criteria and conducting a snowballing process.

Conclusion validity. The data extraction and synthesis processes are conducted by strictly adhering to the *a priori* defined protocol, designed specifically to collect the data necessary to answer our RQs. This reduces potential biases associated to such processes and guarantees that the extracted data is appropriate for our RQs. Furthermore, the study was conducted by adhering to best practices from several sources [6]–[10]. We document each phase of our study in a publicly available research protocol, thus aiding replicability. To ensure the quality of the guidelines, the keywording process and guideline synthesis is conducted collaboratively by two researchers. Conflicts are managed with the intervention of a third researcher. Possible threats related to the interview process are mitigated by conducting internal and external pilots during the interview design phase. This is repeated several times to extensively refine the interview process.

V. Related Work

Despite the wide diffusion of Android apps and their increasing complexity [18], at the time of writing we found a surprising low number of **research studies about the architecture of mobile apps**. By mining and reverse engineering the architecture of more than 1,400 Android apps, Bagheri *et al.* studied the role of software architecture in the design and development of mobile software, extracted the architectural principles that have been applied by app developers, and identified architectural anti-patterns of the Android programming model [19]. They found that Android apps are complex, composed of tens of components, and organized according to multiple architectural styles. These findings motivated us to investigate how developers architect Android apps, eventually leading to the results in this study. Even though our study and that of Bagheri *et al.* share the same target audience, *i.e.,* Android developers and researchers, the research goals are profoundly different: Bagheri *et al.* focus on known architectural principles and how they are reflected in the Android programming model, whereas we aim at characterizing the state of the art and practice on architecting Android apps. Moreover, the methodologies applied in the two studies are completely different – Bagheri *et al.* mined the apps from app stores and statically analyzed their bytecode, whereas we contact Android practitioners complemented with a systematic analysis of the grey and white literature.

An exploratory study targeting common architectural characteristics of 12 real Android applications is reported in [20]. The study is based on the partial extraction of the architecture of the apps using the JDepend tool, followed by the manual analysis of the source code of the targeted apps. The main results of such manual analysis revealed that MVC is a recurrent pattern in Android (although with some violations). In our work we apply a totally different research methodology, where we target professional developers working on industrial projects, rather than developers working on open-source apps. Also, the work proposed in [20] is exploratory in nature and aims at observing the characteristics of the architecture of Android apps, whereas we aim at providing actionable guidelines for helping developers during their everyday activities.

A new MVC-based architectural pattern called Android Passive MVC, is proposed in [21], with the aim of producing Android apps with better maintainability, extensibility, performance, and less complexity. The proposed pattern has been applied to an example of social networking app in collaboration with a development company. Differently from [21], we do not aim at providing a new architectural pattern, rather we accept the existence of many pre-existing ones in Android apps (also confirmed in [19], [20]) and aim at supporting developers while architecting Android apps, without forcing them to learn and apply new (potentially unsupported) architectural patterns.

The authors of [22] performed a preliminary study on how to develop Android apps according to the Software Product Line (SPL) approach. Their case study shows that an adaptation of the principles for SPLs can be adopted for developing Android apps. We differ as we do not imply a change in developers' habits by means of a new development paradigm like SPL, but aim at supporting them in taking better-informed decisions about the architecture of their apps.

A study about the challenges faced by mobile app developers (not only Android) has been proposed by Joorabchi *et al.* [23]. The challenges have been extracted in a qualitative manner from a combination of 12 interviews with practitioners and an on-line questionnaire with 188 participants. Differently from us, they focus on mobile apps in general (incl. web and hybrid apps) and are orthogonal to architectural concerns, *i.e.,* they do not cover the challenges directly related to architecture, but focus on challenges related to *e.g.,* testing. Interestingly, in our study we found some confirmation of their insights, *e.g.,* the importance of testability, partially managed by following the MVP or MVVM patterns.

From a methodological perspective, **multivocal studies** (*i.e.,* systematic studies targeting both grey and white literature) are being published only recently in the field of software engineering [5], *e.g.,* investigating code smells in testing artifacts [24], the startups ecosystem [25], and microservices [26]. Researchers are also complementing multivocal studies with other research methodologies (*e.g.,* semi-structured interviews in our case), thus leading to mixed-method studies. For example, Maro *et al.* combined a tertiary literature review, a case study with a company, and a multivocal literature review to identify challenges and solutions about software traceability in the automotive domain [27]. In the literature there is no multivocal study on mobile apps. So, even though also our research combines a multivocal study with other research methodologies (*e.g.,* interviews), the subjects and therefore the outcomes of our study are novel by focussing on architecting practices for Android apps.

VI. Conclusions and Future Work

This paper presents a mixed-method empirical study on the state of the art and practice on architecting Android apps. A key result of our study is the set of 42 evidence-based guidelines for architecting Android apps. In addition to the guidelines, our study reveals that: (i) so far there are very few academic articles targeting Android architecture (this may be an emerging research gap for academic researchers), (ii) MVP and MVVM are the most recurring architectural patterns, (iii) RxJava, Dagger, and JetPack are the most mentioned libraries when dealing with the architecture of Android apps, and (iv) maintainability, testability, and performance are the most considered QRs when architecting Android apps.

Our results provide developers with an organized set of guidelines for taking action towards improving the architecture of their apps (*e.g.,* when adopting MVP, strive towards making Presenters Android- and frameworks-independent – cf. MVP-2). Researchers, in turn, can benefit from the provided overview of the state of the art and practice, *e.g.,* for tailoring their research towards those QRs that concern developers the most when dealing with the architecture of their apps.

This study opens for many future research directions. Firstly, we are planning a large-scale confirmatory study involving practitioners for checking the correctness and completeness of the proposed guidelines. Secondly, properly designed analysis tools might automatically check violations of guidelines via static and/or dynamic analysis techniques, and recommend solutions for those violations. Furthermore, we strive towards the implementation of an Android reference architecture [28], in order to lay the foundations for compliance-based architectural technical debt analyses [29]. Thirdly, it would be interesting to empirically assess how applying the proposed guidelines can actually impact the quality of the mobile app, thus enabling developers to quantitatively assess the gains in having well-architected apps, potentially speeding up the industrial adoption of the proposed guidelines.

References

[1] "Global mobile OS market share in sales to end users from 1st quarter 2009 to 1st quarter 2018," 2018. [Online]. Available: https://www.statista.com/statistics/266136/global-market-share-held-by-smartphone-operating-systems/

[2] "Global developer population and demographic study 2017," 2017. [Online]. Available: https://evansdata.com/press/viewRelease.php?pressID=244

[3] "Number of available applications in the Google Play Store from December 2009 to June 2018," 2018. [Online]. Available: https://www.statista.com/statistics/266210/number-of-available-applications-in-the-google-play-store/

[4] ISO/IEC, "ISO/IEC 25010 - Systems and software engineering - Systems and software Quality Requirements and Evaluation (SQuaRE) - System and software quality models," Tech. Rep., 2010.

[5] V. Garousi, M. Felderer, and M. V. Mäntylä, "The need for multivocal literature reviews in software engineering: complementing systematic literature reviews with grey literature," in *Proceedings of the 20th International Conference on Evaluation and Assessment in Software Engineering*. ACM, 2016, p. 26.

[7] D. Budgen and P. Brereton, "Performing systematic literature reviews in software engineering," in *Proceedings of the 28th international conference on Software engineering*. ACM, 2006, pp. 1051–1052.

[6] B. Kitchenham and P. Brereton, "A systematic review of systematic review process research in software engineering," *Information and software technology*, vol. 55, no. 12, pp. 2049–2075, 2013.

[8] V. Garousi, M. Felderer, and M. V. Mäntylä, "Guidelines for including grey literature and conducting multivocal literature reviews in software engineering," *Information and Software Technology*, 2018.

[9] C. Wohlin, "Guidelines for snowballing in systematic literature studies and a replication in software engineering," in *Proceedings of the 18th international conference on evaluation and assessment in software engineering*. ACM, 2014, p. 38.

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

[11] B. Kitchenham, R. Pretorius, D. Budgen, O. P. Brereton, M. Turner, M. Niazi, and S. Linkman, "Systematic literature reviews in software engineering–a tertiary study," *Information and software technology*, vol. 52, no. 8, pp. 792–805, 2010.

[12] W. Lidwell, K. Holden, and J. Butler, *Universal principles of design*. Rockport Pub, 2010.

[13] K. Petersen, S. Vakkalanka, and L. Kuzniarz, "Guidelines for conducting systematic mapping studies in software engineering: An update," *Information and Software Technology*, vol. 64, pp. 1–18, 2015.

[14] R. C. Martin, *Clean architecture: a craftsman's guide to software structure and design*. Prentice Hall Press, 2017.

[15] J. Gilbert and C. Stoll, "Architecting iOS Apps with VIPER." [Online]. Available: https://www.objc.io/issues/13-architecture/viper/

[16] E. Wolff, *Microservices: flexible software architecture*. Addison-Wesley Professional, 2016.

[17] I. Malavolta, R. Verdecchia, B. Filipovic, M. Bruntink, and P. Lago, "How Maintainability Issues of Android Apps Evolve," in *2018 IEEE International Conference on Software Maintenance and Evolution (ICSME)*. IEEE, 2018, pp. 334–344.

[18] A. I. Wasserman, "Software engineering issues for mobile application development," in *Proceedings of the FSE/SDP workshop on Future of software engineering research*. ACM, 2010, pp. 397–400.

[19] H. Bagheri, J. Garcia, A. Sadeghi, S. Malek, and N. Medvidovic, "Software architectural principles in contemporary mobile software: from conception to practice," *Journal of Systems and Software*, vol. 119, pp. 31–44, 2016.

[20] E. Campos, U. Kulesza, R. Coelho, R. Bonifácio, and L. Mariano, "Unveiling the Architecture and Design of Android Applications," in *Proceedings of the 17th International Conference on Enterprise Information Systems-Volume 2*, 2015, pp. 201–211.

[21] K. Sokolova, M. Lemercier, L. Garcia, and L. C. Saint Luc, "Towards High Quality Mobile Applications: Android Passive MVC Architecture," *International Journal On Advances in Software*, vol. 7, no. 2, pp. 123–138, 2014.

[22] T. Dürschmid, M. Trapp, and J. Döllner, "Towards architectural styles for Android app software product lines," in *Proceedings of the 4th International Conference on Mobile Software Engineering and Systems*. IEEE Press, 2017, pp. 58–62.

[23] M. E. Joorabchi, A. Mesbah, and P. Kruchten, "Real challenges in mobile app development," in *Empirical Software Engineering and Measurement, 2013 ACM/IEEE International Symposium on*. IEEE, 2013, pp. 15–24.

[24] V. Garousi and B. Küçük, "Smells in software test code: A survey of knowledge in industry and academia," *Journal of Systems and Software*, vol. 138, pp. 52–81, 2018.

[25] N. Tripathi, P. Seppänen, G. Boominathan, M. Oivo, and K. Liukkunen, "Insights into startup ecosystems through exploration of multi-vocal literature," *Information and Software Technology*, vol. 105, pp. 56–77, 2019.

[26] J. Soldani, D. A. Tamburri, and W.-J. Van Den Heuvel, "The pains and gains of microservices: A systematic grey literature review," *Journal of Systems and Software*, vol. 146, pp. 215–232, 2018.

[27] S. Maro, J.-P. Steghöfer, and M. Staron, "Software traceability in the automotive domain: Challenges and solutions," *Journal of Systems and Software*, vol. 141, pp. 85–110, 2018.

[28] R. Verdecchia, "Identifying Architectural Technical Debt in Android Applications through Compliance Checking," in *International Conference on Mobile Software Engineering and Systems*, 2018.

[29] R. Verdecchia, I. Malavolta, and P. Lago, "Architectural Technical Debt Identification: The Research Landscape," in *International Conference on Technical Debt (TechDebt)*, 2018.