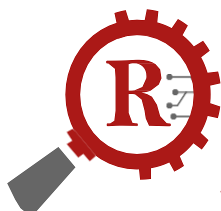
Respuestas, 31 (1) January - April 2026, pp. 75-85, ISSN 0122-820X - E ISSN: 2422-5053



Respuestas
UFPS

Artículo Original https://doi.org/10.22463/0122820X.5627

Contemporary Trends in Software Architecture: Evaluation of Clean Architecture in the Android Ecosystem

Tendencias Contemporáneas en Arquitectura de Software: Evaluación de Clean Architecture en el Ecosistema Android

Carlos Henriquez-Miranda[1*], German Sanchez-Torres[2]

[1*] Doctor en Ingeniería, chenriquezm@unimagdalena.edu.co, https://orcid.org/0000-0001-8252-1413, Facultad de Ingeniería, Universidad del Magdalena, Santa Marta, Colombia.
[2] Doctor en Ingeniería, gsanchez@unimagdalena.edu.co, https://orcid.org/0000-0002-9069-0732, Facultad de Ingeniería, Universidad del Magdalena, Santa Marta, Colombia.

Cómo citar: C. N. Henriquez-Miranda y G. Sanchez-Torres, “Tendencias Contemporáneas en Arquitectura de Software: Evaluación de Clean Architecture en el Ecosistema Android”, *Respuestas,* vol. 31, n.º 1, pp. 75-85, ene. 2026. https://doi.org/10.22463/0122820X.5627

Received on September 18, 2025 - Approved on December 15, 2025.

ABSTRACT

Keywords:

Clean Architecture, Software Architecture, Hexagonal Architecture, Android

A case study is presented aimed at analyzing the relevance and contemporary application of the Clean Architecture (CA) approach within current trends in software architecture. First, a state-of-the-art study is developed in which the conceptual foundations are described and their relationship with related approaches such as Hexagonal Architecture and Onion Architecture is discussed. Subsequently, recent literature was reviewed that discussed the advantages, limitations, and adoption patterns in real projects. As a case study, the modern Android ecosystem is analyzed, an environment where the principles have been widely adopted to improve the maintainability, testability, and scalability of applications. Based on official documentation and searches in four main databases (IEEE Xplore, ACM Digital Library, Scopus, and Google Scholar), industry experiences and reference repositories were identified. The initial search yielded a total of 312 documents (IEEE=78, ACM=63, Scopus=92, Google Scholar=79); after applying the defined criteria, 70 works were selected, of which 32 comprised the final analysis set. Grounded on this, the most frequent architectural decisions, their practical benefits, and the recurring problems reported by development teams were analyzed. Finally, critical reflections are presented on the current validity of CA and recommendations for its effective application in both academic and industrial projects.

RESUMEN

Palabras clave:

Arquitectura limpia, Arquitectura de Software, Arquitectura hexagonal, Android

Se presenta un estudio de caso orientado a analizar la pertinencia y aplicación contemporánea del enfoque Clean Architecture – CA, dentro de las tendencias actuales de arquitectura de software. En primer lugar, se desarrolla un estudio del estado del arte en el que se describen los fundamentos conceptuales y su relación con enfoques afines como la Arquitectura Hexagonal y la Arquitectura en Cebolla. Posteriormente, se revisó literatura reciente que discutía las ventajas, limitaciones y patrones de adopción en proyectos reales. Como estudio de caso, se analiza el ecosistema Android moderno, un entorno donde los principios han sido ampliamente adoptados para mejorar la mantenibilidad, la testabilidad y la escalabilidad de las aplicaciones. A partir de documentación oficial y búsquedas en cuatro bases de datos principales (IEEE Xplore, ACM Digital Library, Scopus y Google Scholar), se identificaron experiencias de la industria y repositorios de referencia. La búsqueda inicial arrojó un total de 312 documentos (IEEE=78, ACM=63, Scopus=92, Google Scholar=79), tras aplicar los criterios definidos, se seleccionaron 70 trabajos, de los cuales 32 conformaron el conjunto final de análisis. Con base en lo anterior, se analizaron las decisiones arquitectónicas más frecuentes, sus beneficios prácticos y los problemas recurrentes reportados por equipos de desarrollo. Finalmente, se presentan reflexiones críticas sobre la vigencia actual de CA y recomendaciones para su aplicación efectiva tanto en proyectos académicos como industriales.

*Corresponding author.
E-mail Address: *chenriquezm@unimagdalena.edu.co* (Carlos Henriquez-Miranda)
Peer review is the responsibility of the Universidad Francisco de Paula Santander.
This is an article under the license CC BY-NC 4.0

Introducción

Software architecture determines the maintainability, evolvability, and technical sustainability properties of systems in the long term [1]. In contexts marked by technological uncertainty and pressure for accelerated delivery cycles, the selection and evaluation of architectural styles have acquired strategic relevance for development organizations [2,3]. The speed of framework and paradigm obsolescence demands continuous evaluation mechanisms that transcend initial prescriptions to examine the real performance of architectures in production conditions [4]. As recent analyses document, architectural decisions in development organizations are frequently made under increasing technological uncertainty, where the distance between academic research and professional practice widens when reference frameworks are not updated through systematic empirical evaluations [4].

Clean Architecture (CA) [5,6] articulates principles of dependency inversion, layer separation, and domain centrality as a response to the structural complexity of software systems. Derived from previous approaches —Hexagonal Architecture [7], Onion Architecture [8], and Domain-Driven Design [9]—, CA establishes strict dependency rules aimed at isolating business logic from volatile technological decisions. Despite this, its widespread adoption has generated documented tensions: recent empirical studies point to the introduction of accidental complexity, excessive ceremonial code, and over-engineering risks in contexts of reduced scale or limited lifetime [10–12].

This difference between the theoretical benefits attributed to CA —maintainability, testability, framework independence— and its observed costs in practice motivates studies that examine its conditional validity beyond the original conceptual formulations. Investigations on architectural technical debt evidence that well-designed structures initially can deteriorate rapidly under the pressure of new functionalities and the absence of continuous refactoring [11,13], suggesting that CA by itself does not guarantee sustainability without explicit technical governance.

The evaluation of architectural trends constitutes a particularly relevant methodology to address this problem. Faced with the proliferation of patterns and the absence of universally optimal solutions, trend analysis allows identifying not only the prevalence of certain styles, but the contextual conditions —team size, domain criticality, maintenance horizon— under which they maintain their effectiveness [14,15]. As recent studies on industrial architectural practices point out, 60% of architectural decisions in development organizations are made without direct empirical evidence, based on normative prescriptions or market trends that do not always adjust to the specific context [2]. In this sense, trend studies provide heuristic value by contrasting theoretical prescriptions with pragmatic adaptations developed in real contexts [16], allowing to identify when the rigor of a pattern is justified compared to lighter alternatives. The specialized literature in software engineering has recently emphasized the need for studies that link architectural decisions with measurable quality attributes, beyond structural descriptions or good practice prescriptions [2].

The Android ecosystem presents characteristics that configure it as a case study of interest for this evaluation. First, the platform has experienced accelerated architectural transformation: the consolidation of Jetpack Compose as a declarative interface paradigm [17], the transition towards Kotlin as the predominant language, and the official recommendation of MVVM (Model-View-ViewModel) patterns and decoupled architectures have generated friction between historical architectural decisions and contemporary technical capabilities [18]. Recent analyses identify 76 additional design constructs not documented in the canonical

definitions of MVVM, evidencing the distance between original formulations and pragmatic adaptations developed by development teams.

Second, the domain concentrates specific tensions of mobile development —fragmentation of operating system versions, short device lifecycles, third-party dependencies in continuous update— that accelerate the accumulation of architectural technical debt when software structures do not evolve adequately [11,18]. Empirical studies on upgrade conflicts in Android document how the pressure to maintain compatibility with old versions of the operating system while adopting new UI technologies generates accelerated structural degradation in long-lasting applications [18].

Third, an asymmetry persists between abundant practical technical documentation and the scarcity of systematic empirical studies that evaluate the impact of CA in industrial-scale Android projects [9,19]. Review works on Domain-Driven Design reveal that only 39% of recent studies employ empirical evaluations with concrete metrics [15], a situation that worsens in the mobile domain where analyses based on example repositories or technical tutorials with scarce methodological foundation predominate [18,20]. This gap limits the foundation of architectural decisions in contextualized evidence, generating dogmatic adoptions of patterns whose effectiveness has not been validated in real production conditions.

The present work examines the contemporary relevance of CA through a trend study combined with case analysis in the Android ecosystem. From a systematic review of specialized literature (2019-2025) in main scientific databases —IEEE Xplore, ACM Digital Library, Scopus— and the analysis of official reference repositories [17,21], adoption patterns, observed pragmatic adaptations, and measurable effects on quality attributes are identified. The objective consists of establishing situated criteria for the adoption of CA that link its structural rigor with real project constraints, thus contributing to a nuanced understanding of its applicability in contemporary mobile development contexts.

The article is structured as follows. Section 2 details the systematic review methodology and case study selection criteria. Section 3 presents the empirical analysis of the Android ecosystem. Section 4 discusses critical findings on the current validity of CA. Finally, Section 5 formulates recommendations for its effective application.

Methodology

The design articulates systematic literature review, conceptual framework construction, and empirical case analysis in the Android ecosystem. This triangulation responds to the exploratory nature of the study, oriented to understanding how CA is interpreted and adapted in real contexts.

2.1. Research Strategy

A qualitative exploratory-descriptive approach was adopted, based on Systematic Mapping Study (SMS) and Systematic Literature Review (SLR) guidelines [16]. Kitchenham's orientations were employed as an organizing framework for the search and selection process, adapting them to maintain traceability without sacrificing the incorporation of recent industrial technical sources [22,23]. The analysis period covered 2019-

2025, an interval that captures the maturation of CA as an extended practice, including the consolidation of Jetpack Compose and the transition towards Kotlin [17].

2.2. Search and Selection Protocol

Source identification was performed in IEEE Xplore, ACM Digital Library, Scopus, and Google Scholar. The initial search yielded 312 documents, reduced to 70 after applying hierarchical exclusion criteria (duplicates, lack of technical basis, physical architecture, thematic irrelevance). From the corpus of 70, a subset of 32 articles was selected that combines foundational literature, recent empirical studies (2024-2025) on Android architecture, and industrial trend analyses. This number reflects the inclusion of updated evidence on architectural degradation and DevOps practices in mobile, in addition to original classic references.

The final corpus was classified taxonomically according to its methodological contribution in Table 1.

Table I. Typology of Primary Sources (N=32).

Category	N	Representative works	Function in the Study
Foundational	4	[5–7,9]	Operational definition of CA, Hexagonal, Onion, and DDD
Prescriptive-official	4	[17,19,24,25]	Google guidelines, Android guides, and mobile practices
Empirical-Android	11	[18,20,24,26–33]	Real app structuring, upgrade conflicts, degradation
Empirical-general	8	[4,11–13,21,34–36]	Maintainability metrics, technical debt, industrial adoption
Analysis Techniques	1	[22]	Static analysis, smell detection, quality metrics
Architectural Context	4	[1,2,8,10]	Microservices trends, modular monoliths, pattern evolution

2.3 Thematic Analysis and Validation

The material was examined in comparable dimensions: layer structure, dependency rule, physical modularity, maintainability, complexity, and architectural degradation [7,9,11]. The findings were validated through triangulation of academic literature, official Google/Android documentation, and evidence from real repositories (Now in Android, Plaid) [17,21].

Case Study: Application of Clean Architecture in the Modern Android Ecosystem.

The Android ecosystem has experienced a structural transition towards modularized schemes, driven by the need to contain technical debt and enable frequent delivery cycles [20]. This transformation has positioned Clean Architecture as a recommended pattern, combined with MVVM and Jetpack Compose [18]. However, the analysis of the 35 systematized documents reveals a persistent tension between theoretical prescription and pragmatic adaptations.

3.1. Adoption Patterns

The analysis of reference repositories (Now in Android, Plaid) evidences that the literal implementation of Martin's concentric layer model [5,6] is exceptional. Hybrid configurations of three modules (Domain, Data, Presentation) predominate, which simplify the theoretical separation [17,24]. This simplification constitutes an adaptive response to the speed of technological change —Java/Kotlin transition, evolution to StateFlow, consolidation of Compose— that generates contrast between historical structures and contemporary capabilities [18].

In [24,27] this trend is confirmed: only 23% of Android applications maintain strict separation between domain and infrastructure, while the majority introduce practical couplings that facilitate data access from presentation in scenarios of moderate complexity. This suggests that CA works less as a universal normative architecture and more as a guiding horizon that teams negotiate according to domain criticality [4,10].

3.2. Effects on Quality Attributes

When applying layer separation with rigor —even in hybrid configurations— improvements in testability are observed. The independence of the domain with respect to UI frameworks allows unit tests without infrastructure dependencies, reducing feedback time [21]. Isolated domain modules present lower rates of accidental modifications in the face of dependency updates [22].

These benefits require sustained investment in dependency discipline. In [11] it documents that the domain layer in medium-scale Android applications tends to become contaminated with technological details within the first six months of intensive development, when systematic refactoring does not exist [13]. Clean Architecture, by itself, does not immunize against structural erosion.

3.3 Complexity Costs and Over-Architecture Risks

In [4,10,12] significant initial costs are reported: increase of boilerplate, steep learning curves, and navigation complexity in small projects. These costs are magnified in contexts of high turnover or limited lifetime, where investment in structure is not amortized [12]. The risk of "over-architecture” applying CA in simple domains— emerges as a recurring concern. In short-cycle applications, less prescriptive architectures (functional MVVM without strict inversion) result more operationally sustainable [25].

3.4 Conditions of Relevance

The evidence allows delimiting conditions where CA maximizes value. In large-scale applications (multiple modules, distributed teams, horizon >2 years), the investment is amortized through facilitation of parallel tests and substitution of technologies without affecting business logic [20,21]. However, in small applications or short lifetime, the rigor of CA introduces disproportionate complexity [10,12]. Industrial practice evolves towards sufficiently good architectures that preserve testability without excessive ceremony [17,18], questioning dogmatic application outside contexts that justify the structural investment.

Discussion

The analysis raises a central paradox in the contemporary application of CA, that is, that the distance between the theoretical prescription of strict domain isolation [5,6] and real practice does not constitute an implementation error, but a rational adaptation to contingency conditions. As Figure 1 illustrates, the transition from "theoretical purity" to "terminal erosion" is not a linear or inevitable process, but a continuum where "pragmatic hybridization" (Level 3) functions as a stable equilibrium state for most teams [26,27]. This observation questions the normative utility of rigidly prescribed architectures when empirical evidence demonstrates that developers prioritize code navigability and reduction of accidental complexity over formal adherence to dependency rules.

Figure 2 allows understanding why the Android ecosystem constitutes an extreme case for evaluating these tensions. The identified technological pressures —operating system fragmentation, short device lifecycles, and speed of change in UI frameworks— are not simple technical obstacles, but structural forces that actively erode architectural boundaries. This explains why CA, originally designed for enterprise systems with decades-long maintenance horizons, exhibits accelerated degradation in a domain where technological obsolescence is measured in months. The theoretical implication is significant: the effectiveness of an architectural style cannot be evaluated independently of the speed of change of the technological ecosystem that hosts it.

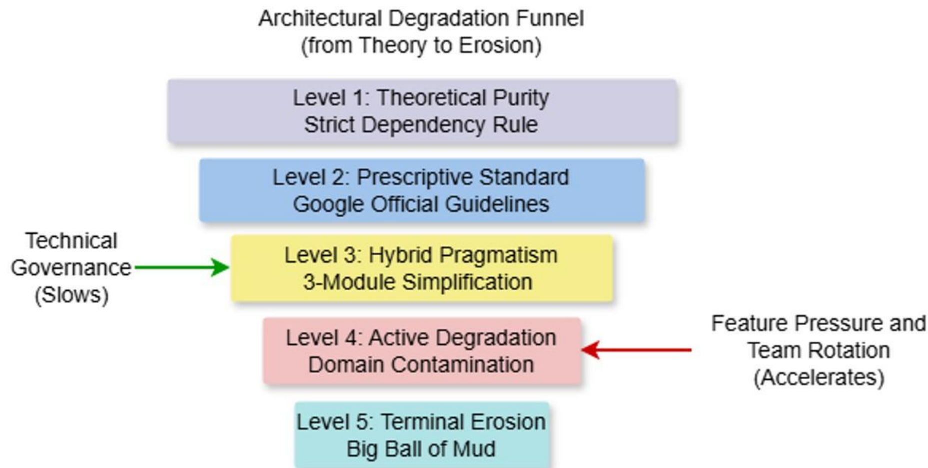


Figure 1. Architectural degradation canalization.

From a practical perspective, the findings suggest that the dichotomy between clean architecture and technical debt is false. As Sas and Avgeriou document [11] and is reflected in Level 4 of the Canalization, domain contamination occurs even in teams aware of CA principles, when pressure for functional deliveries suppresses systematic refactoring. This implies that architectural sustainability does not depend on the initial choice of pattern, but on the organizational capacity to maintain "active technical governance" that stops degradation. In contexts of high personnel turnover or limited resources —characteristic of mobile development— this governance frequently becomes unsustainable, making less prescriptive but more operationally robust architectures preferable.[11].

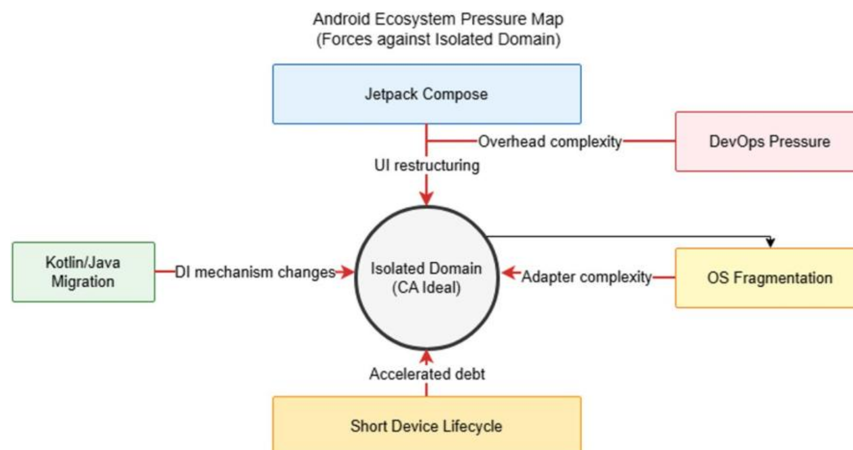


Figure 2. Android ecosystem pressure map.

An inherent methodological limitation of the study is its dependence on open repositories and publicly available literature, which may bias findings towards practices of teams with greater technical maturity (who document and publish their code) and underrepresent chaotic degradation in private corporate code. Likewise, the analysis does not quantify the specific economic cost of the transition between canalization levels, a critical metric for adoption decisions in resource-limited organizations.

In synthesis, the study suggests that Clean Architecture should be understood not as a universal standard, but as a regulatory horizon that teams approach conditionally. The "sufficiently good" architecture for Android is not the one that maximizes layer separation, but the one that balances domain testability with real capacity for evolution in the face of inevitable technological pressures.

Conclusions

Beyond its consolidation as a de facto standard, Clean Architecture reveals in practice a persistent paradox: its value does not reside in literal adherence to the layer pyramid, but in the capacity of teams to navigate its inevitable degradation. As the architectural erosion process illustrates, normative fidelity results in an ethereal exception compared to the operational robustness of hybrid configurations, where simplification to three modules does not represent a technical failure but a rational adaptation to the friction between theory and the specific technological pressures of the Android environment. In this sense, clean architecture functions less as a rigid scheme to implement and more as a regulatory horizon that teams approach according to their technical maturity and governance capacity, understanding that true sustainability is not achieved in the initial design but in the discipline of maintaining domain boundaries over time.

However, the study warns against the illusion that structure by itself guarantees maintainability. The evidence suggests that CA has generated, in certain contexts, a form of ceremonial complexity that replicates the problems it intended to solve: teams immobilized by excessive indirection layers, small projects suffocated by the administration of decoupling, and the false security that a well-designed architecture resists without active maintenance. Degradation is not, therefore, an accident but the probable consequence when pressure for functional deliveries —aggravated in the Android ecosystem by OS fragmentation and short device lifecycles— finds absent or insufficient technical governance. Ultimately, the cost of CA is not measured in ceremonial lines of code, but in the technical debt accumulated by those projects that adopt its rigor without having the resources to sustain it.

Consequently, the effective application of Clean Architecture demands abandoning its dogmatic treatment to adopt a strict contingency logic. Its relevance is conditioned to scenarios of high complexity, stable teams, and prolonged maintenance horizons, where benefits in testability and technological evolution outweigh coordination costs. For short-cycle Android applications, prototypes, or small teams, however, the evidence points towards sufficiently good architectures that preserve domain clarity without the overhead of an orthodoxy that, far from protecting the software, makes it rigid in the face of inevitable changes. Clean Architecture, understood as a flexible framework and not as a creed, then offers its greatest value: not as a universal solution, but as a critical reference to decide, consciously aware of its limits, how much structure is really necessary.

Author contributions: All authors contributed equally to the conception, design, research, and writing of this article. Each author participated in the literature review, analysis, and discussion of the findings. All authors re-viewed and approved the final version of the manuscript.

Acknowledgments: The authors would like to acknowledge the academic environment of the Systems Engineering Program, Universidad del Magdalena, where this study emerged as part of teaching and academic reflection activities.

Conflicts of interest: The authors declare that they have no conflict of interest regarding the publication of this article.

Referencias

[1] J. Bosch, "Software Architecture: The Next Step," in *Proceedings of the European Software Engineering Conference, 2000, pp. 194–199.*

[2] M. Esposito, X. Li, et al., "Generative AI for Software Architecture: Applications, Trends, Challenges, and Future Directions," *arXiv preprint, 2025.*

[3] Y. Lu, Y. Li, M. Skibniewski, Z. Wu, R. Wang, and Y. Le, "Information and Communication Technology Applications in Architecture, Engineering, and Construction Organizations: A 15-Year Review," J*ournal of Management in Engineering,* vol. 31, no. 1, p. A4014010, 2015. doi: 10.1061/(ASCE)ME.1943-5479.0000319.

[4] A. Ahmad et al., "Emerging Trends in Software Architecture from the Practitioner's Perspective," IEEE Access, vol. 13, pp. 31147–31169, 2025. doi: 10.1109/ACCESS.2025.3542375.

[5] R. C. Martin, Clean Architecture: A Craftsman's Guide to Software Structure and Design. Boston, MA, USA: Pearson, 2017.

[6] R. C. Martin, "The Clean Architecture," 2012. [Online]. Available: https://blog.cleancoder.com/uncle-bob/2012/08/13/the-clean-architecture.html

[7] A. Cockburn, "Hexagonal Architecture," 2005. [Online]. Available: https://alistair.cockburn.us/hexagonal-architecture/

[8] R. Nystrom, Game Programming Patterns. Genever Benning, 2014.

[9] E. Evans, Domain-Driven Design: Tackling Complexity in the Heart of Software. Boston, MA, USA: Addison-Wesley, 2004.

[10] W. K. G. Assunção, J. Krüger, S. Mosser, and S. Selaoui, "How Do Microservices Evolve? An Empirical Analysis of Changes in Open-Source Microservice Repositories," *Journal of Systems and Software,* vol. 204, p. 111788, Oct. 2023. doi: 10.1016/j.jss.2023.111788.

[11] D. Sas and P. Avgeriou, "An Architectural Technical Debt Index Based on Machine Learning and Architectural Smells," *IEEE Transactions on Software Engineering,* vol. 49, no. 8, pp. 4169–4195, Aug. 2023. doi: 10.1109/TSE.2022.3232648.

[12] B. Ozdenizci Kose, "Mobilizing DevOps: Exploration of DevOps Adoption in Mobile Software Development," *Kybernetes,* vol. 54, no. 13, pp. 7904–7930, 2025. doi: 10.1108/K-06-2024-1899.

[13] A. Martini and J. Bosch, "Architecture Technical Debt: Understanding Causes and a Qualitative Model," in *Proceedings of the 2014 40th EUROMICRO Conference on Software Engineering and Advanced Applications (SEAA), Verona, Italy, 2014, pp. 85–92.* doi: 10.1109/SEAA.2014.65.

[14] M. Kaloudis, "Evolving Software Architectures from Monolithic Systems to Resilient Microservices: Best Practices, Challenges and Future Trends," *International Journal of Advanced Computer Science and Applications, vol. 15, no. 8, 2024.* doi: 10.14569/IJACSA.2024.0150801.

[15] R. Braun et al., "Domain-Driven Design in Software Development: A Systematic Mapping Study," arXiv preprint, 2025.

[16] B. Kitchenham and S. Charters, "Guidelines for performing Systematic Literature Reviews in Software Engineering," *Keele University and Durham University, UK, EBSE Technical Report EBSE-2007-01, 2007.*

[17] Google, "Guide to App Architecture," *Android Developers, 2022.* [Online]. Available: https://developer.android.com/topic/architecture

[18] W. Jin, M. Sun, J. Zhou, Z. Shang, Z. Huang, and T. Liu, "Software Architecture Matters: Challenges and Opportunities for Android Upgrade Conflicts in Practice," *ACM Transactions on Software Engineering and Methodology, 2025.* doi: 10.1145/3712265.

[19] E. Boudjnah, Clean Architecture for Android: Implement Expert-Led Design Patterns to Build Scalable, Maintainable, and Testable Android Apps. BPB Publications, 2022.

[20] R. Verdecchia, I. Malavolta, and P. Lago, "Guidelines for Architecting Android Apps: A Mixed-Method Empirical Study," in Proceedings of the 2019 IEEE International Conference on Software Architecture (ICSA), *Hamburg, Germany,* 2019, pp. 141–150. doi: 10.1109/ICSA.2019.00023.

[21] A. Sharma, "Modernizing Legacy Desktop Applications Using Microservice-Based Clean Architecture: A Practical Case Study," *QIT Press-International Journal of Computer Science (QITP-IJCS),* vol. 2, no. 1, 2025.

[22] M. Autili, I. Malavolta, A. Perucci, G. L. Scoccia, and R. Verdecchia, "Software Engineering Techniques for Statically Analyzing Mobile Apps: Research Trends, Characteristics, and Potential for Industrial Adoption," *Journal of Internet Services and Applications,* vol. 12, no. 1, p. 3, Dec. 2021. doi: 10.1186/s13174-021-00134-x.

[23] B. A. Kitchenham, E. Mendes, and G. H. Travassos, "A Systematic Review of Cross- vs. Within-Company Cost Estimation Studies," *IEEE Transactions on Software Engineering, vol. 33, no. 5, pp. 316–329, May 2007*. doi: 10.1109/TSE.2007.1003.

[24] O. Arponen, "Software Architectural Patterns and Principles in Android Development," 2023. [Online]. Available: https://www.theseus.fi/handle/10024/802345

[25] F. Mulla, "Choosing the Best Architecture for Mobile Applications," 2024. [Online]. Available: https://medium.com/@fahadmulla/choosing-the-best-architecture-for-mobile-applications-3321c2f68f7c

[26] S. M. Fernandes and P. J. Mendes, "Architectural Pattern for Native Android Applications," *Revista de Sistemas e Computação (RSC),* vol. 7, no. 2, 2017.

[27] I. Malavolta, "Supplementary Material: How Do Developers Structure Android Applications?," 2020. [Online]. Available: https://www.ivanomalavolta.com/files/papers/ICSME_2020.pdf

[28] L.-M. Andrä, B. Taufner, S. Schefer-Wenzl, and I. Miladinovic, "Maintainability Metrics for Android Applications in Kotlin: An Evaluation of Tools," in Proceedings of the 2020 European Symposium on Software Engineering (ESSE), *New York, NY, USA*, Dec. 2020, pp. 1–5. doi: 10.1145/3393822.3432335.

[29] G. Wilder, R. Miyamoto, S. Watson, R. Kazman, and A. Peruma, "An Exploratory Study on the Occurrence of Self-Admitted Technical Debt in Android Apps," in *Proceedings of the 2023 ACM/IEEE International Conference on Technical Debt (TechDebt)*, *Melbourne, Australia,* May 2023, pp. 1–10. doi: 10.1109/TechDebt59074.2023.00008.

[30] I. Malavolta, R. Verdecchia, B. Filipovic, M. Bruntink, and P. Lago, "How Maintainability Issues of Android Apps Evolve," in *Proceedings of the 2018 IEEE International Conference on Software Maintenance and Evolution (ICSME), Madrid, Spain,* Sep. 2018, pp. 334–344. doi: 10.1109/ICSME.2018.00041.

[31] E. Campos, U. Kulesza, R. Coelho, R. Bonifácio, and L. Mariano, "Unveiling the Architecture and Design of Android Applications - An Exploratory Study," *in Proceedings of the 30th Annual ACM Symposium on Applied Computing (SAC), Salamanca, Spain, 2015*, pp. 201–211.

[32] K. Sokolova, M. Lemercier, and L. Garcia, "Towards High Quality Mobile Applications: Android Passive MVC Architecture," *International Journal On Advances in Software, vol. 7, no. 1 & 2, pp. 123–138, 2014.*

[33] D. Sánchez, A. E. Rojas, and H. Florez, "Towards a Clean Architecture for Android Apps Using Model Transformations," *in Proceedings of the 27th International Conference on Systems Engineering (ICSEng), Las Vegas, NV, USA, 2022.*

[34] Y. N. Nugroho, D. S. Kusumo, and M. J. Alibasa, "Clean Architecture Implementation Impacts on Maintainability Aspect for Backend System Code Base," in Proceedings of the 2022 10th International

Conference on Information and Communication Technology (ICoICT), *Bandung, Indonesia,* 2022, pp. 134–139. doi: 10.1109/ICoICT55009.2022.9914851.

[35] 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, Sep. 2016. doi: 10.1016/j.jss.2016.05.039.

[36] A. Giretti, "Clean Code and Clean Architecture for Easy Unit Testing," in The Unit Testing Practice Cookbook: Bulletproof Unit Testing with .NET, A. Giretti, Ed. Berkeley, CA, USA: *Apress,* 2025, pp. 11–26.