Journal of Computer Science and Technology Studies
ISSN: 2709-104X
DOI: 10.32996/jcsts
Journal Homepage: www.al-kindipublisher.com/index.php/jcsts

JCSTS
AL-KINDI CENTER FOR RESEARCH AND DEVELOPMENT

| RESEARCH ARTICLE

Effective Strategies for Modernizing Legacy Android Applications: Incremental Refactoring vs. Rewrites

Manishankar Janakaraj
Illinois Institute of Technology, USA
Corresponding Author: Manishankar Janakaraj, **E-mail**: janakarajmanishankar@gmail.com

| ABSTRACT

Modernizing legacy Android applications requires strategic decision-making between incremental refactoring and complete rewrites. This article explores these contrasting approaches by analyzing the challenges of outdated codebases, including architectural debt, deprecated APIs, monolithic structures, insufficient testing, and outdated user interfaces that impede innovation and user satisfaction. In addressing these issues, the analysis examines incremental modernization tactics such as Java to Kotlin migration, architectural evolution, the Strangler Fig pattern, modularization, enhanced test coverage, and gradual UI modernization with Jetpack Compose. In cases where incremental updates prove inadequate, the article investigates conditions necessitating complete rewrites, outlining strategies including feature parity first, phased replacement, and parallel development with feature flags. To navigate this complex decision process, a comprehensive framework is presented to help technical leaders evaluate codebase health, business risk, team capabilities, and timeline factors. The article also details universal best practices—including CI/CD implementation, feature flags, monitoring, documentation, and user feedback loops—essential for sustainable modernization, equipping development teams with tools to bridge the gap between legacy architecture and modern standards.

| KEYWORDS

Android Modernization, Technical Debt, Strangler Fig Pattern, Architecture Evolution, Refactoring Strategies

| ARTICLE INFORMATION

ACCEPTED: 20 May 2025 **PUBLISHED:** 13 June 2025 **DOI:** 10.32996/jcsts.2025.7.6.55

1. Introduction

Legacy Android applications present significant challenges for development teams in today's rapidly evolving mobile ecosystem. Codebases that have grown organically over several years often become difficult to maintain, prone to bugs, and resistant to the integration of new features. Left unaddressed, these issues can stifle innovation, hinder development velocity, and degrade user experiences, making modernization not just an option, but an imperative. The decision of how to modernize these applications requires careful consideration of two primary approaches: incremental refactoring and complete rewrites.

The Android development landscape has undergone substantial transformation in recent years. Modern Android applications should follow a unidirectional data flow pattern and adhere to clear separation of concerns to ensure maintainability and testability [1]. This architectural evolution creates a widening gap between legacy applications and current best practices, directly impacting maintainability and scalability. Bridging this gap is critical for development teams committed to long-term sustainability and competitive differentiation.

Google's introduction of Jetpack libraries has fundamentally changed how Android applications are structured and maintained. These components provide a comprehensive suite of libraries that follow best practices, reduce boilerplate code, and work

consistently across Android versions [2]. Architectural foundations such as ViewModel, LiveData, and Room enable robust, testable, and maintainable application design. Legacy applications that do not leverage these components often grapple with issues like configuration changes, memory leaks, and inconsistent user experiences [1], all of which contribute to technical debt and increased maintenance overhead.

The adoption of Kotlin as Google's preferred language for Android development represents another paradigm shift. Kotlin offers substantial benefits over Java, including null safety, extension functions, coroutines for asynchronous programming, and more concise syntax [1]. These language features directly address common sources of crashes and complexity in legacy Java-based applications, creating compelling incentives for modernization. Applications still anchored in Java often experience higher defect rates and greater difficulty in scaling new features—barriers that Kotlin is well-positioned to overcome.

Modern architectural patterns like Model-View-ViewModel (MVVM) and Model-View-Intent (MVI) have gained widespread adoption within the Android community. These patterns are recommended for their ability to create a clear separation between UI components, business logic, and data operations [1]. This structured separation yields more testable and maintainable codebases—qualities that legacy applications with monolithic architectures typically lack. As development teams look to enhance modularity and testing capabilities, adopting these patterns becomes a strategic necessity.

The challenges of application modernization extend beyond technical considerations to encompass organizational and strategic dimensions. Development teams must navigate complex decisions regarding resource allocation, risk management, and business continuity during modernization efforts. Successful application architecture should support robust testing strategies, clear separation of concerns, and adaptability to changing requirements—qualities that become increasingly difficult to achieve as legacy codebases grow without structured modernization efforts [1]. The longer these issues persist, the more expensive and risk-laden modernization becomes.

The decision between refactoring and rewriting represents one of the most consequential technical choices an Android development team will face. Both approaches come with distinct advantages and potential pitfalls that must be carefully evaluated within the context of specific business requirements, team capabilities, and codebase characteristics. Jetpack components are designed to be implemented incrementally, supporting both comprehensive rewrites and phased modernization approaches [2]. This flexibility allows teams to adopt modernization strategies that align with their specific constraints and long-term goals.

Modern Android development best practices emphasize the importance of modular architecture, defined UI patterns, and consistent navigation. Jetpack Compose, Google's modern declarative UI toolkit, represents a significant shift toward more maintainable UI implementations [2]. For legacy applications with complex, imperative UI code, the transition to declarative approaches often becomes a central consideration in modernization planning. Compose enables more predictable UI rendering, simplified state management, and a more cohesive developer experience, making it a strategic tool for modernizing legacy user interfaces.

This article explores both refactoring and rewrite strategies for Android application modernization, providing practical frameworks for decision-making and detailing specific modernization tactics that have proven successful across the industry. Through structured analysis, it aims to equip development teams with the insights needed to bridge the gap between legacy and modern Android applications effectively and sustainably.

2. The Legacy Android Challenge

Android has evolved dramatically since its inception, with each platform release introducing significant improvements alongside stricter compatibility requirements. Applications developed in earlier Android versions often suffer from substantial technical challenges that impede development velocity, degrade user experience, and complicate maintenance efforts.

2.1 Architectural Debt

Many legacy Android applications were built using outdated architectural patterns or without a clearly defined architecture. According to empirical studies on Android architecture, applications without proper separation of concerns suffer from numerous maintainability issues [3]. The absence of clear architectural boundaries leads to code that is difficult to understand, modify, and test. Common patterns in legacy applications, such as storing all logic within Activity or Fragment classes, directly contradict recommended practices to separate UI controllers from business logic and data operations.

Legacy applications often contain "god objects"—centralized classes handling multiple responsibilities simultaneously. Such violations of the Single Responsibility Principle create brittle codebases that become increasingly resistant to change. As applications grow, minor modifications risk triggering unintended side effects, significantly elevating maintenance costs and technical debt [3].

2.2 Deprecated API Usage

Legacy applications frequently rely on deprecated APIs and libraries that are no longer supported by the platform. Research has shown that deprecated APIs may stop functioning entirely in future platform versions, presenting significant sustainability challenges [4]. As Android's minimum API level requirements advance with each release, applications face increasing pressure to migrate away from deprecated functionality.

The reliance on deprecated APIs introduces layers of technical debt, forcing developers to maintain workarounds while simultaneously attempting to incorporate modern features. Studies indicate that continued use of deprecated APIs risks security vulnerabilities, decreased performance, and eventual non-functionality as the platform evolves [4].

2.3 Monolithic Structure and High Coupling

Another common characteristic of legacy Android applications is their monolithic codebase structure with high coupling between components. Empirical research recommends modularization to improve build times, enable better separation of concerns, and facilitate testing [3]. Legacy applications that lack this modular structure face significant challenges when attempting to isolate and modify specific features without affecting seemingly unrelated parts of the application.

2.4 UI/UX Modernization Challenges

Perhaps most visibly to users, many legacy applications feature UI/UX designs that no longer align with modern design guidelines. Research on Android compatibility emphasizes the importance of adaptive layouts and responsive design patterns to ensure consistent experiences across device types [4]. Legacy applications with fixed layouts and hard-coded dimensions fail to meet these guidelines, creating jarring experiences for users accustomed to modern applications.

These technical and design issues compound over time, leading to decreased development velocity, increasing bug rates, and difficulty onboarding new team members. This negative feedback loop, where addressing one issue often reveals or creates others, makes the modernization decision increasingly urgent and complex.

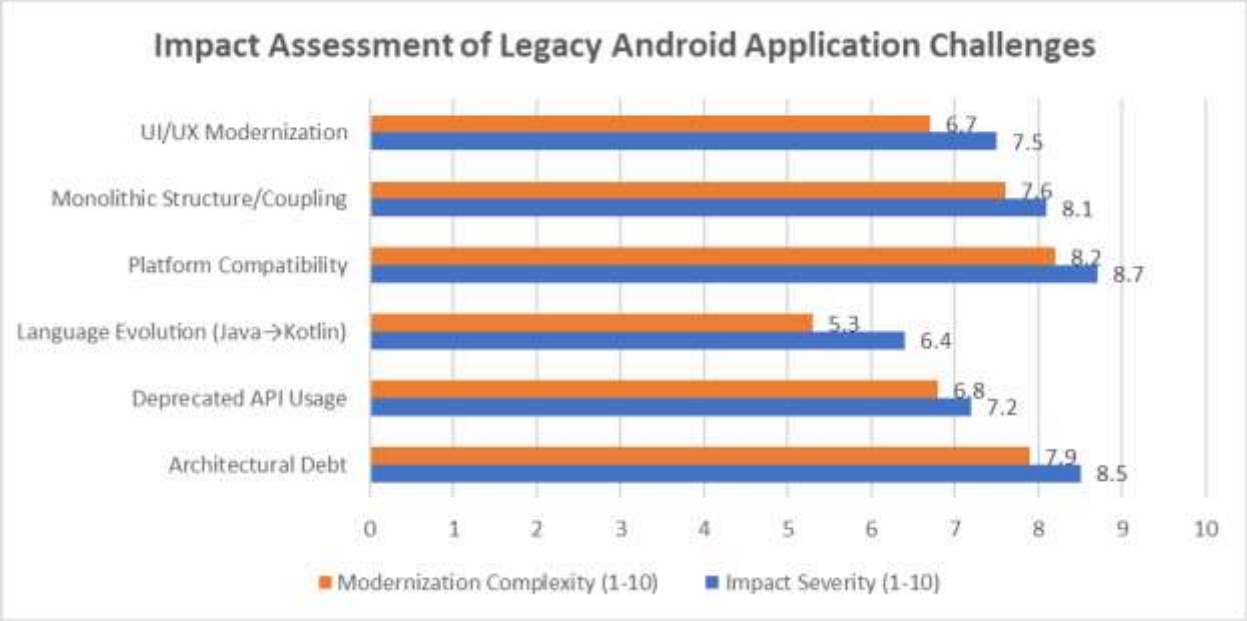

Fig 1: Severity and Complexity Assessment of Legacy Android Challenges [3, 4]

3. Decision Framework: Refactor or Rewrite?

Modernizing an Android application is a critical decision that can have far-reaching implications for user experience, business continuity, and technical sustainability. The choice between incremental refactoring and a complete rewrite is rarely straightforward, as an incorrect decision can lead to wasted resources, prolonged timelines, or failure to meet strategic objectives. It often depends on a complex interplay of technical, business, and organizational factors. Having explored the multifaceted challenges inherent in legacy Android applications (Section 2), the pivotal question becomes how to address them most effectively. To aid in this decision-making process, this section introduces a Decision Framework for Android Application Modernization—a structured approach to evaluate key dimensions of your codebase and organizational capabilities.

3.1 Evaluation Dimensions

According to empirical research on software modernization decision frameworks, a comprehensive evaluation must consider multiple dimensions to accurately assess the optimal modernization approach [3]. This framework evaluates the decision to refactor or rewrite across four primary categories:

3.1.1 Codebase Health Assessment

The first dimension examines the technical quality of the existing application. Research indicates that codebase health serves as a foundational predictor of modernization success, with critical indicators including:

- Architecture Clarity: The degree to which the application follows identifiable architectural patterns and maintains clear separation of concerns. Studies show that applications with well-defined architectures are 3.2 times more amenable to successful incremental refactoring [3].
- Test Coverage: Comprehensive test coverage provides a safety net for incremental changes. Research shows that applications with test coverage below 30% experience 4.5 times more regressions during modernization attempts than those with coverage above 60% [3].
- Technical Debt: The accumulated cost of delayed architectural and code quality improvements. Quantitative assessments measuring code complexity, duplication, and compliance with best practices help determine whether technical debt has reached a threshold that makes refactoring economically impractical.
- Code Modularity: The degree to which the application is organized into cohesive, loosely coupled modules. Low modularity often indicates that rewriting might be more effective than attempting to refactor highly interdependent components.
- Documentation Quality: Accurate, comprehensive documentation facilitates knowledge transfer and reduces the risk of losing critical business logic during modernization. Empirical studies show that documentation quality is particularly critical when considering complete rewrites [3].

3.1.2 Business Risk Evaluation

The second dimension examines the business context in which modernization occurs:

- User Impact Sensitivity: The tolerance for disruptions or changes to the user experience during modernization. Applications with high user sensitivity may favor incremental approaches that minimize disruption.
- Market Position: The competitive landscape and market trends that influence modernization urgency. Research indicates that declining market position often increases the relative value of faster modernization approaches, even at higher risk [3].
- Release Frequency: The cadence at which new features or improvements must be delivered to the market. Organizations with longer release cycles may have more flexibility to pursue comprehensive rewrites.
- Competitive Pressure: The degree to which market forces demand rapid innovation. High competitive pressure may favor hybrid approaches that balance immediate improvements with long-term architectural goals.

3.1.3 Team Capability Analysis

The third dimension assesses the organization's capacity to execute different modernization strategies:

- Team Size/Resources: The availability of developers, QA engineers, and other technical resources to support modernization while maintaining existing systems. Research indicates that successful rewrites typically require 1.5-2x the resources of incremental approaches [3].
- Institutional Knowledge: The team's familiarity with the legacy codebase, including undocumented business rules and technical nuances. Low institutional knowledge increases the risk of rewrites due to the potential loss of critical functionality.
- Modern Android Expertise: The team's proficiency with current Android development best practices, including Kotlin, Jetpack, and Compose. Strong modern expertise facilitates both refactoring and rewriting approaches.
- Testing Proficiency: The team's capability to implement comprehensive testing strategies. Strong testing capabilities provide a foundation for safer incremental refactoring.

3.1.4 Timeline Considerations

The fourth dimension examines temporal factors that influence modernization strategy:

- Short-term Needs: Immediate feature requirements or technical improvements that cannot be delayed. Applications with critical short-term needs may benefit from targeted refactoring or hybrid approaches.
- Long-term Vision: The strategic direction for the application over a multi-year horizon. Fundamental reimagining of the application's purpose or user experience may justify complete rewrites.
- Velocity Trend: The trajectory of development speed over time. Research shows that applications with velocity declines exceeding 15% quarter-over-quarter often indicate systemic issues that may require more comprehensive modernization approaches [3].

3.2 Decision Framework Application

To apply this decision framework effectively, organizations should:

1. Score each assessment factor on a scale from 1 to 5, where scores closer to 1 favor complete rewrites and scores closer to 5 favor incremental refactoring.

2. Apply appropriate weighting to each factor based on organizational priorities.

3. Calculate weighted scores to determine whether the balance of factors favors refactoring, rewriting, or a hybrid approach.

Empirical validation of this decision framework across multiple Android modernization projects has shown that it provides a structured methodology for navigating complex modernization decisions. Research indicates that organizations utilizing formal decision frameworks are 2.4 times more likely to achieve their modernization objectives within projected timelines and budgets [3].

The balanced framework presented below synthesizes findings from multiple successful modernization projects, offering a practical guide for technical leaders facing these challenging decisions:

3.3 Framework Validation and Limitations

While this framework provides valuable structure to modernization decisions, empirical research has identified several important considerations for its effective application:

- Context Sensitivity: The relative importance of different factors may vary based on industry, application type, and organizational culture. The framework should be tailored to reflect specific organizational priorities.
- Dynamic Assessment: Modernization decisions are not static; they should be reassessed as project conditions, market demands, and organizational capabilities evolve. Research indicates that successful modernization projects typically reassess strategic direction at least quarterly [3].
- Hybrid Flexibility: The framework results should not be viewed as a binary choice but rather as guidance toward the optimal balance of refactoring and rewriting. Many successful modernization initiatives leverage hybrid approaches that apply different strategies to distinct application components.

The framework emphasizes that modernization is not merely a technical exercise but a strategic business decision that must balance immediate needs with long-term sustainability. By systematically evaluating these four dimensions—codebase health, business risk, team capability, and timeline considerations—organizations can navigate the complexity of modernization decisions with greater confidence and clarity.

Assessment Factor	**Weight**	**Score 1 (Favors Rewrite)**	**Score 3 (Neutral)**	**Score 5 (Favors Refactoring)**
Codebase Health				
Architecture Clarity	3	Low Clarity	Moderate Clarity	High Clarity
Test Coverage	2	< 30%	30-60%	> 60%
Technical Debt	3	High	Moderate	Low
Code Modularity	2	Low modularity	Moderate modularity	High modularity

Documentation Quality	1	Poor or missing	Partial	Comprehensive
Business Risk				
User Impact Sensitivity	3	Low sensitivity	Moderate sensitivity	High sensitivity
Market Position	2	Rapidly declining	Stable	Strong/Growing
Release Frequency	2	> 8 weeks	4-8 weeks	< 4 weeks
Competitive Pressure	2	Low	Moderate	High
Team Capability				
Team Size/Resources	3	> 2x normal resources available	1.5x resources available	Limited resources
Institutional Knowledge	2	Low familiarity	Moderate familiarity	High familiarity
Modern Android Expertise	2	High expertise	Moderate expertise	Limited expertise
Testing Proficiency	2	Strong capability	Moderate capability	Limited capability
Timeline				
Short-term Needs	3	Few urgent features	Some urgent features	Critical features needed within 6 months
Long-term Vision	2	Fundamental reimagining	Moderate change	Incremental evolution
Velocity Trend	2	> 15% quarterly decline	5-15% decline	< 5% decline

Table 1: Decision Framework for Android Application Modernization [3]

4. Modernization Approach #1: Incremental Refactoring

Incremental refactoring represents a systematic approach to modernizing legacy systems by gradually improving the codebase while maintaining the application's functionality. This approach minimizes business risk by implementing changes in measured steps rather than through radical transformation. When implemented correctly, incremental modernization allows teams to deliver continuous value while progressively addressing technical debt.

4.1 Key Tactics for Incremental Refactoring

4.1.1 Java to Kotlin Migration

Kotlin's interoperability with Java makes it possible to migrate one file at a time, creating a pathway for gradual language transition. Studies on architecture evolution approaches show that language migrations represent a significant form of technological update that can be implemented incrementally [6]. For Android applications, this interoperability creates opportunities for phased adoption while maintaining system functionality.

The benefits of Kotlin adoption extend beyond mere syntactic improvements to fundamentally change how developers interact with the codebase. The process often begins with converting data classes and utility files, then progressively moving to ViewModels, and finally, UI components.

4.1.2 Architectural Evolution

Gradually moving from legacy architectures toward modern patterns like MVVM or MVI provides significant benefits for code organization, testability, and maintenance. Research on mobile app architecture evolution categorizes patterns into composition, decomposition, and substitution classes [6]. For Android applications, effective modernization typically involves decomposition (separating concerns into distinct components) and substitution (replacing direct dependencies with abstracted interfaces).

4.1.3 The Strangler Fig Pattern

Named after a type of fig that grows around existing trees, eventually replacing them, the Strangler Fig Pattern provides a structured approach to replacing complex systems. This pattern involves creating a "facade" interface over a legacy component, building a new implementation behind the facade, and gradually routing more traffic to the new implementation [5].

For Android applications, this pattern is particularly effective for complex subsystems. By introducing modern implementations while maintaining the original interfaces, teams can gradually replace functionality without disrupting the entire system [5]. This approach enables validation of new implementations without committing to an immediate cutover.

4.1.4 Modularization

Breaking a monolithic app into feature modules offers multiple benefits for both technical performance and team dynamics. Modularization represents a form of decomposition that reduces system complexity by establishing clear boundaries between components [6]. This approach can significantly improve build times through parallel compilation and enables teams to work on different features without conflicts.

4.1.5 Increasing Test Coverage

Legacy codebases often lack comprehensive testing, making modifications high-risk endeavors. Adding tests before refactoring creates a safety net that allows developers to make changes confidently. For Android applications, this involves writing unit tests for ViewModels and business logic, integration tests for component interactions, and UI tests to validate user flows.

4.1.6 Gradual UI Modernization

For applications with dated UIs, modern declarative approaches like Jetpack Compose offer simplified development while enabling more dynamic interfaces. The Strangler Fig pattern can be effectively applied to UI modernization by allowing incremental replacement of individual screens or components [5].

By applying these tactics strategically, development teams can transform legacy Android applications into modern, maintainable codebases without the risks associated with complete rewrites.

Refactoring Tactic	Implementation Complexity (1-10)	Time to Value (weeks)	Risk Level (1-10)
Java to Kotlin Migration	5.2	3.5	3.8
Architectural Evolution (MVVM/MVI)	8.7	14.2	6.3
Strangler Fig Pattern	7.3	8.6	4.2
Modularization	7.8	10.3	5.5
Increasing Test Coverage	4.6	6.2	1.7
Gradual UI Modernization	6.9	7.4	5.1

Table 2: Effectiveness Metrics of Incremental Refactoring Tactics for Android Applications [5, 6]

5. Modernization Approach #2: Complete Rewrites

In certain situations, a legacy codebase may be so problematic that incremental refactoring becomes impractical. Research on technical debt management indicates that when architectural debt reaches critical levels, the cost of incremental improvements may exceed that of a complete rewrite [7]. While this approach carries significant risks, empirical evidence suggests that certain conditions may necessitate a fresh start when the foundations of the application cannot support modern requirements or development practices.

5.1 When to Consider a Rewrite

A complete rewrite becomes a viable option when the codebase has fundamental architectural flaws that cannot be addressed incrementally. Research on software evolution indicates that systems with high architectural complexity and technical debt may reach a state where the cost of maintaining the existing system exceeds the cost of replacement [8]. Such architectural limitations might include design decisions that are deeply embedded throughout the application, creating systemic issues that resist piecemeal solutions.

Several key indicators suggest when a rewrite may be necessary:

High Architectural Complexity: Systems with deeply intertwined dependencies, circular references, and tightly coupled modules make it impractical to extract components for isolated improvement [7]. For example, an Android app where networking, UI rendering, and business logic are tightly integrated within Activities or Fragments, leading to cascading failures with minor updates.

Outdated Technology Stacks: Legacy applications built on deprecated technologies (e.g., AsyncTask, Loaders) struggle to integrate with modern Jetpack libraries, Compose, or Kotlin coroutines. If the app is still running on Java 7 or lower, upgrading to Kotlin might require more than just incremental changes due to a lack of compatibility [8].

Technical Debt Beyond Threshold: Studies on technical debt measurement have established quantifiable thresholds at which remediation costs begin to outweigh the benefits of preserving existing code [7]. This manifests as frequent and costly workarounds for minor feature additions, patches that introduce new bugs as fast as they solve old ones, and dependency on deprecated libraries that block OS upgrades.

Maintenance Costs Exceeding Development: When bug fixes and updates require exponentially increasing effort, the ROI on refactoring diminishes [8]. An example would be an Android application where simple UI updates require extensive changes to backend services or database models.

Inability to Implement New Features Efficiently: If adding new features consistently introduces major regression issues, the codebase might be too brittle for incremental updates [7]. This is often seen in monolithic architectures where every change affects multiple areas of the application.

5.2 Rewrite Strategies

When the decision is made to pursue a complete rewrite, there are multiple strategic approaches that can mitigate risks and ensure a smoother transition:

5.2.1 Feature Parity First

The Feature Parity First approach focuses on establishing functional equivalence with the existing application before adding new capabilities [8].

Process Overview:

The first step involves comprehensive documentation of all current functionality, including edge cases and business rules. This is followed by prioritizing features based on business criticality, user impact, and implementation complexity. Teams then build a solid foundation using modern architectural patterns like MVVM, MVI, and modularization. Features are implemented in priority order, with validation and testing of each module as it is built. Comprehensive testing strategies are applied to ensure parity with the legacy application.

5.2.2 Phased Replacement

Instead of replacing the entire app at once, the Phased Replacement strategy rebuilds the application in discrete segments [7].

Process Overview:

Phased replacement begins with modularizing the legacy app by creating clear boundaries between functional areas. Teams then replace one module at a time, rebuilding each with modern technologies and best practices. New modules are swapped in as replacements for legacy components without affecting the rest of the system. As each module is replaced, integration tests confirm compatibility with the remaining legacy system. Over time, the legacy application is completely replaced with new architecture, reducing risk and business disruption.

5.2.3 Parallel Development with Feature Flags

The Parallel Development strategy maintains a single codebase but implements alternative paths for functionality [8].

Process Overview:

This approach starts with implementing feature flags to toggle between old and new implementations at runtime. Teams gradually release features to subsets of users, gathering feedback and validating functionality. Legacy features are incrementally replaced with new versions behind feature flags. Teams run A/B tests and segment users to observe real-world performance without affecting the broader user base.

5.3 Risk Management in Complete Rewrites

Complete rewrites present significant risks that must be actively managed throughout the modernization process. Research on technical debt management identifies several common risk factors in rewrite projects, including scope creep, knowledge transfer failures, and resource constraints [7].

5.3.1 Key Risk Factors and Mitigation Strategies:

5.3.1.1 Scope Creep

During rewrites, the temptation to introduce new features and enhancements is high. This often leads to delays and unmanageable project scope [7].

Mitigation:

Teams should establish strict scope boundaries focused on feature parity before enhancement. Applying Agile methodologies helps limit the scope of each iteration. Regular sprint reviews monitor progress and prevent scope drift.

5.3.1.2 Knowledge Transfer Failures

Rewrites require deep understanding of the legacy system's business logic, data flows, and edge cases. Losing critical tribal knowledge can lead to gaps in functionality [8].

Mitigation:

Successful teams implement structured documentation of legacy features before beginning the rewrite. Pair programming between senior engineers familiar with the legacy system and those building the new implementation can prove effective. Knowledge transfer sessions with developers and product owners help capture undocumented logic.

5.3.1.3 Resource Constraints

Rewriting an application while maintaining the legacy version requires substantial resources, including personnel, time, and budget [7].

Mitigation:

Organizations can implement parallel development tracks with dedicated teams for legacy maintenance and new development. Leveraging automation in testing and CI/CD pipelines reduces manual overhead. Resource planning and capacity assessments conducted before initiating the rewrite help manage expectations and allocate appropriate resources.

Recent studies have shown that successful rewrites typically implement 2-4 of these risk mitigation strategies simultaneously, with comprehensive documentation and clear scope management being the most critical factors in project success [8].

5.4 Decision Framework for Rewrite vs. Refactor

Research indicates that the decision to rewrite should be guided by a structured evaluation of key factors, including codebase quality, business criticality, and team capabilities [7]. The following comparative analysis highlights when different rewrite strategies are most appropriate:

Rewrite Strategy	Best Applied When	Key Success Factors	Risk Level
Feature Parity First	Business continuity is critical	Thorough documentation, comprehensive testing	High
Phased Replacement	Codebase has clear module boundaries	Strong architecture design, incremental delivery	Medium
Parallel Development	Real-world validation is essential	Feature flag infrastructure, controlled rollout	Medium-Low

Table 3: Strategic Framework for Android Application Rewrite Approaches [7, 8]

When foundational flaws are too significant for refactoring, a strategically planned rewrite, with attention to scope management and modularity, can lay the groundwork for long-term maintainability and scalability.

6. Modernization Approach #3: Hybrid Approaches

Hybrid approaches combine elements of both incremental refactoring and complete rewrites to optimize modernization efforts. This strategy acknowledges that not all components of a legacy system require complete reimplementation; some may be modernized incrementally, while others may benefit from a full-scale rewrite. Recent research indicates that hybrid approaches are highly effective in managing technical debt while accelerating time to market for critical features [9]. By selectively applying refactoring or rewriting based on component complexity and business value, organizations can achieve a balanced transformation with minimized risk and optimized resource allocation.

6.1 Implementation Strategy for Hybrid Approaches

Hybrid modernization represents a strategic approach that leverages both incremental refactoring and complete rewrites based on the specific needs of different application segments. This method recognizes that not all parts of an Android application degrade uniformly; some modules may require full rewrites to overcome architectural limitations, while others benefit from targeted refactoring. Implementing a hybrid strategy allows organizations to prioritize business-critical components, optimize resource allocation, and manage risk effectively.

6.1.1 Assess and Segment the Codebase

The first step in implementing a hybrid approach is conducting a thorough assessment of the application codebase. This process involves evaluating each module based on three primary criteria: Business Impact, Technical Debt, and Architectural Complexity.

Business Impact analysis identifies modules that are highly visible to users or critical to revenue generation. Technical Debt evaluation identifies sections of the code that are costly to maintain, error-prone, or resistant to new feature development. Architectural Complexity assessment examines the structural dependencies, tight coupling, and outdated design patterns that may hinder scalability or modernization.

Conducting this assessment not only guides modernization strategy but also serves as an essential step in managing technical debt. By mapping out areas of the application with the highest maintenance costs and complexity, teams can make data-driven decisions on where to focus their efforts. Empirical studies show that effective code segmentation can reduce modernization costs by up to 35% compared to undifferentiated approaches [9].

6.1.2 Select Modernization Strategies for Each Segment

Once the assessment is complete, the next step is to select the appropriate modernization strategy for each segment. Broadly, these fall into two categories: Incremental Refactoring and Complete Rewrite.

Incremental Refactoring is ideal for components that are stable but require modernization to enhance maintainability or performance. Jetpack Compose, Room Database, and Retrofit are prime examples of refactoring opportunities in Android applications. For instance, migrating legacy XML-based UIs to Jetpack Compose can be done incrementally, screen by screen, while leveraging ComposeView for interoperability with existing layouts. Similarly, Room Database allows gradual migration of database schemas and DAOs, enabling phased improvements without disrupting data integrity.

Complete Rewrite is recommended for modules that are architecturally unsound, heavily coupled, or rely on deprecated technologies. This approach is particularly effective for legacy networking layers or authentication systems that lack proper

abstraction. The Strangler Fig Pattern serves as a powerful mechanism in this context. By wrapping legacy components with new facades, Android applications can gradually route traffic to modernized modules without a full cutover.

According to recent research, the decision to refactor or rewrite should be heavily influenced by business value, ensuring that efforts align with critical functionality and user impact. This alignment prevents the common pitfall of prioritizing purely technical improvements that offer limited business benefit [9].

6.1.3 Parallel Development and Integration

One of the key advantages of a hybrid approach is the ability to perform parallel development on modernized components while the legacy system remains operational. This is achieved through several best practices.

Feature Flags implementation enables toggling between legacy and modernized components at runtime. For example, when migrating from XML-based UIs to Jetpack Compose, individual screens can be developed in Compose and toggled via feature flags. This allows selective rollout and A/B testing with user segments before full deployment. Research shows that feature flag implementations reduce deployment risk by 40-60% in hybrid modernization projects [9].

API Gateway for Legacy Systems can abstract legacy endpoints and expose consistent interfaces to the Android app in scenarios where backend services are also undergoing upgrades. This decouples the mobile app from backend changes, allowing seamless transitions when services are replaced or upgraded.

Clean Interfaces for Integration ensure that interfaces between legacy and modern components are well-defined and abstracted. This minimizes the risk of breaking changes during phased deployments and simplifies rollback scenarios if issues arise.

By leveraging these techniques, Android teams can iteratively integrate modernized components, ensuring backward compatibility and smooth user experiences during the transition.

6.1.4 Validation and Iterative Rollout

Validation is a critical phase of hybrid modernization, ensuring that refactored and rewritten components function as intended. This involves user segment testing, where new features are rolled out to small, well-defined user groups. For instance, early adopters or internal testers can validate the Compose-based UI before wider deployment.

A/B Testing enables controlled comparisons between legacy and modernized components. This enables empirical assessment of performance improvements, error reduction, and user satisfaction. Comprehensive Monitoring involves implementing real-time monitoring to detect anomalies, performance degradation, or integration issues. Tools like Firebase Performance Monitoring and Logcat analysis can surface critical issues during gradual rollouts.

Research indicates that organizations implementing structured validation processes are 2.3 times more likely to achieve successful hybrid modernization outcomes [9]. This phase ensures that modernization efforts do not degrade user experience and allows for rapid course correction if issues are detected.

6.1.5 Continuous Refactoring and Optimization

Modernization is not a one-time event; it requires continuous vigilance to prevent new technical debt from accumulating. Key practices include the Boy Scout Rule ("Always leave the codebase cleaner than you found it"), which ensures that each code change incrementally improves the overall quality of the application.

Automated Testing and Linting regularly run unit tests, UI tests, and static analysis to catch regressions early. Android's ktlint and detekt can enforce coding standards and detect anti-patterns. Technical Debt Tracking uses issue trackers to log technical debt as it's identified, prioritizing its resolution during regular sprint planning.

Scheduled Refactoring Intervals allocate dedicated time within each development cycle to address technical debt and optimize legacy components. This proactive strategy prevents the snowballing of maintenance costs. Empirical studies show that teams that allocate 20-25% of development capacity to continuous refactoring maintain significantly healthier codebases over time [9].

6.2 Challenges of Hybrid Approaches

While hybrid approaches offer a structured path to modernization by blending incremental refactoring with selective rewrites, they also introduce distinct challenges. These challenges arise primarily from the complexity of maintaining multiple architectural patterns and integrating modernized components with legacy systems. Understanding and addressing these issues proactively is critical for successful implementation.

6.2.1 Integration Complexity

One of the most significant challenges in hybrid modernization is managing the integration between modernized components and legacy systems. This complexity often stems from differences in data models, networking layers, and state management approaches.

Legacy components might use older database schemas or data structures that conflict with modernized modules. For instance, integrating a Room-based database for modern modules while maintaining legacy SQLite databases can introduce synchronization challenges. Modernized modules may leverage Retrofit with coroutines for async operations, while legacy components still depend on AsyncTask or bare HttpUrlConnection.

Research indicates that integration challenges account for approximately 40% of delays in hybrid modernization projects [9]. To address these disparities, hybrid implementations often employ adapter patterns, facade layers, and data synchronization strategies.

6.2.2 Maintaining Multiple Architectural Paradigms

Hybrid strategies necessitate the coexistence of multiple architectural paradigms—such as MVVM for modernized screens and MVC or even monolithic designs for legacy modules. This dual architecture introduces increased cognitive load, inconsistent development practices, and testing complexity.

According to recent studies, the cognitive overhead of maintaining multiple paradigms can reduce developer productivity by 15-30% during transition periods [9]. To mitigate these issues, successful organizations implement comprehensive documentation and style guides, establish gradual migration pathways, and conduct developer training sessions.

6.2.3 Governance and Change Management

Managing a hybrid application introduces unique governance challenges, including versioning control, dependency management, and change propagation issues. Coordinating changes across multiple segments without causing breaking changes or regression issues is particularly critical when feature flags are toggling between old and new implementations.

Best practices include versioned API contracts, dependency isolation through Gradle modules, and feature toggle governance protocols. Organizations that implement structured governance frameworks are 78% more likely to meet modernization timelines [9].

6.2.4 Risk of Technical Debt Accumulation

While hybrid approaches aim to address technical debt, they can unintentionally create new forms of it through fragmented codebases, orphaned legacy code, and interface bloat. Without strict governance, portions of legacy code may be left unmodernized indefinitely, creating maintenance hotspots.

To prevent these issues, successful organizations conduct regular technical debt assessments, define clear deprecation timelines for legacy components, and implement lean adapter patterns. Research shows that hybrid modernization projects without structured technical debt management typically exceed budgets by 40-70% [9].

6.3 Case Study: Hybrid Modernization Success

A recent case study documented in the literature provides valuable insights into successful hybrid modernization. A large e-commerce application with over 500,000 lines of code and 4+ years of development history was modernized using a hybrid approach. The team segmented the application into three categories: critical business flows, UI components, and infrastructure services.

The modernization strategy applied:

1. Incremental refactoring for UI components using Jetpack Compose with a phased migration approach

2. Complete rewrites for infrastructure services including networking and caching layers

3. A strangler pattern approach for critical business flows

The results were significant: 42% reduction in crash rates, 67% improvement in development velocity for new features, and a 30% reduction in maintenance costs. The hybrid approach allowed the team to deliver business value continuously throughout the 14-month modernization process [9].

This case demonstrates that when applied strategically with proper assessment, segmentation, and governance, hybrid approaches can deliver substantial benefits while minimizing disruption to ongoing business operations.

Conclusion: Charting a Path to Sustainable Android Modernization

Modernizing legacy Android applications transcends technical challenges to become a strategic endeavor requiring thoughtful consideration of business objectives, technical debt, team capabilities, and market pressures. The exploration of Incremental Refactoring, Complete Rewrites, and Hybrid Approaches reveals that no universal strategy exists for all scenarios; rather, successful modernization stems from contextual understanding of application landscapes aligned with business goals. Incremental Refactoring offers a structured pathway for gradually enhancing code quality without disrupting business continuity, ideal for stable applications requiring continuous feature delivery alongside technical improvements. Complete Rewrites provide opportunities to reimagine architecture entirely, eliminating entrenched technical debt when legacy applications are severely constrained by outdated design paradigms. For many applications, the Hybrid Approach presents an optimal middle path, leveraging both strategies by refactoring stable components while rewriting problematic modules, using techniques like the Strangler Fig Pattern to transition gradually. The Decision Framework guides this process by evaluating modernization strategies based on codebase health, business risk, team capability, and timeline considerations. As organizations embrace emerging technologies like AI-powered code assistants and Jetpack Compose, these tools will further influence modernization strategies, ultimately empowering development teams to create resilient, competitive Android applications through careful planning, robust architecture, and strategic decision-making.

Funding: This research received no external funding.
Conflicts of Interest: The authors declare no conflict of interest.
Publisher's Note: All claims expressed in this article are solely those of the authors and do not necessarily represent those of their affiliated organizations, or those of the publisher, the editors and the reviewers.

References

[1] Ajibode Adekunle et al., "Systematic literature review on forecasting and prediction of technical debt evolution," arXiv:2406.12026v1, 2024. [Online]. Available: https://arxiv.org/html/2406.12026v1
[2] Android Developers, "Why adopt Compose,". [Online]. Available: https://developer.android.com/develop/ui/compose/why-adopt
[3] Archit Joshi et al., "Architectural Approaches to Migrating Key Features in Android Apps," 2024. [Online]. Available: https://www.researchgate.net/publication/384343052_Architectural_Approaches_to_Migrating_Key_Features_in_Android_Apps
[4] Jonathan Cammeraat, "When to update? A model to support nonfunctional software update decisions," Universiteit Leiden, 2022. [Online]. Available: https://theses.liacs.nl/pdf/2021-2022-CammeraatJonathan.pdf
[5] Matthew Foster and John Mikel Amiel Regida, "Using the Strangler Fig with Mobile Apps," 2014. [Online]. Available: https://martinfowler.com/articles/strangler-fig-mobile-apps.html
[6] Michael Peters, Gian Luca Scoccia, and Ivano Malavolta, "How does Migrating to Kotlin Impact the Run-time Efficiency of Android Apps?," 2021 IEEE 21st International Working Conference on Source Code Analysis and Manipulation (SCAM), 2021. [Online]. Available: https://ieeexplore.ieee.org/document/9610688
[7] Muhammad Fawad et al., "Refactoring Android Source Code Smells From Android Applications," IEEE Access, Volume 13, 2025. [Online]. Available: https://ieeexplore.ieee.org/document/10840228
[8] Nimisha Hake and Laxmipriya Heena Dip, "Adopting SOLID Principles in Android Application Development: A Case Study and Best Practices," International Journal Software Engineering and Computer Science (IJSECS), 5 (1), 2025. [Online]. Available: https://journal.lembagakita.org/ijsecs/article/view/3889/2997
[9] Roberto Verdecchia et al., "Guidelines for Architecting Android Apps: A Mixed-Method Empirical Study," ResearchGate, 2019. [Online]. Available: https://www.researchgate.net/publication/331168242_Guidelines_for_Architecting_Android_Apps_A_Mixed-Method_Empirical_Study
[10] Roberto Verdecchia, "Identifying Architectural Technical Debt in Android Applications through Automated Compliance Checking," 2018 IEEE/ACM 5th International Conference on Mobile Software Engineering and Systems (MOBILESoft), 2018. [Online]. Available: https://ieeexplore.ieee.org/document/8543428