A Qualitative Comparison of iOS Presentation Layer Architectures Across UIKit and SwiftUI

Tamara Milovanović
Departman of Information and Communication Technologies
The Academy of Applied Technical and Preschool Studies
Niš, Serbia
tamara.milovanovic@akademijanis.edu.rs

Bratislav Predić
Departman of Computer Science
University of Niš, Faculty of Electronic Engeneering
Niš, Serbia
bratislav.predic@elfak.ni.ac.rs

Abstract— **Modern mobile applications evolve quickly in scope and complexity, making architectural choices critical for maintainability, testability, and long-term scalability. This paper compares widely used presentation layer architectures for iOS mobile applications, such as MVC, MVVM, MVVM-C and discusses MV as a lightweight, recently revisited baseline in state-driven UI development. We define evaluation criteria covering separation concerns, testability, navigation and discuss how these criteria behave across UIKit and SwiftUI based user interfaces. This analysis highlights the practical trade-offs between minimal boilerplate and clear responsibility boundaries and show how navigation and flow orchestration become a decisive factor as applications grow. Based on the comparison, we derive decision-oriented guidelines that to help practitioners select an architecture aligned with application size, team structure and technology stack.**

Keywords— iOS applications architecture, presentation layer, state management, navigation and flow coordination, UIKit vs SwiftUI

I. INTRODUCTION

In software engineering, architecture captures the key structural decisions that shape how an application is organized [1]. It defines how responsibilities are divided, how components communicate, and where important logic is placed, including state handling, side effects, and flow coordination [1]. In mobile development these decisions are especially consequential because the user interface is tightly connected to user interaction, device constraints, and frequent product iteration. When architecture is chosen and applied consistently, it creates clear boundaries, supports incremental evolution, and helps teams reason about the system as it grows [1].

Mobile applications are expected to evolve rapidly in both functionality and scale while maintaining reliability and a consistent user experience. As products grow from a small set of screens into feature rich systems, teams face recurring engineering challenges. They must keep presentation logic maintainable, enable effective testing, manage state changes predictably, and avoid tight coupling between the UI and application logic. In this setting, architecture is not merely a stylistic choice. It directly shapes long term quality attributes such as modularity, maintainability, and scalability [1]. In iOS development, architectural decisions are closely tied to the UI technology stack. UIKit follows an imperative programming model and has historically been the dominant foundation for iOS applications [2], while SwiftUI promotes a declarative, state driven approach in which the UI is a function of state [3]. Many real-world codebases combine both approaches, either to support incremental migration or to leverage UIKit maturity in specific areas [2], [3]. The shift toward declarative UI increases the importance of explicit state propagation and reactive updates, which in turn affects how responsibilities should be distributed between views and the layers that provide presentation logic [3], [4]. As an iOS application expands beyond a small number of screens, weaknesses in structure become increasingly visible. Presentation logic and state handling may spread across UI elements, and dependencies between features can emerge gradually as a byproduct of ongoing changes. Over time, this increases coupling, reduces cohesion, and makes testing harder because key behavior becomes difficult to validate without running the full application [1]. In state driven interfaces, frequent updates can further amplify these effects, making unintended side effects harder to track and reproduce [3], [4]. Poor architectural boundaries also raise the cost of change. Feature delivery slows as code paths become more fragile, regressions become more frequent, and refactoring requires more effort because responsibilities are not clearly separated [1]. Navigation and user flows may become entangled with view rendering, which makes complex journeys harder to extend and difficult to test in isolation. In practice, these symptoms often lead to accidental architecture, where the system functions but its structure is shaped by incremental additions rather than an intentional design [1].

This paper focuses on selecting an architecture for the iOS presentation layer under modern UI development. The goal is to provide a structured qualitative comparison of representative architectural approaches using practical criteria such as separation of concerns, testability, state propagation, navigation and flow complexity, implementation overhead, and suitability for UIKit and SwiftUI [1]-[4]. Based on this comparison, we derive decision-oriented recommendations to support architecture selection based on application scale, expected growth, and the nature of user flows rather than intentional design [1].

II. CONTEXT AND SCOPE

This study examines architectural choices specifically within the iOS presentation layer. The presentation layer is treated as the part of the application that turns state into UI, processes user interactional, and mediates UI driven behavior through presentation logic [1]. In addition to view rendering, this scope includes state ownership and transformation, validation and formatting for display, and the initiation of UI triggered side effects that impact what the user sees [3], [4]. The goal is to evaluate how architectural approaches organize these responsibilities and how clearly, they separate them from each other [1].

To keep the comparison focused and applicable across different codebases, business rules and data persistence are

not analyzed as independent layers. Instead, they are considered only insofar as they influence presentation concerns, for example through asynchronous operations, error handling and the need to keep UI state consistent with remote local data [1]. Similarly, dependency injection and module boundaries are treated as supporting practices rather than primary comparison targets, unless they directly affect testability at the presentation level [1]. A central motivation for this scope is the role of state driven updates and navigation flows in modern iOS application. In practice, state changes originate from user input, network responses and local storage updates, and they must be propagated in a predictable way to avoid inconsistent UI and hard to reproduce behavior [3], [4]. At the same time, navigation increasingly represents complex user journeys rather than simple screen transitions, often requiring shared state and coordination across multiple views [7]. For this reason, the comparison emphasizes how different approaches handle state propagation, side effects, and flow coordination without leaking responsibilities into UI code [1], [7]. These following sections use this scope to compare representative iOS presentation layer architectures and to derive practical recommendations for architecture selection under UIKit, SwiftUI and hybrid codebases [2], [3].

III. Technical Context and Compared Architecture

A. *UIKit, SwiftUI and state driven UI*

UIKit is based on an imperative UI model where UI updates are typically triggered explicitly in response to events and data changes [2]. SwiftUI adopts a declarative model in which the interface is described as a function of state and is automatically re-rendered when that state changes [3]. The conceptual difference between explicit imperative updates and declarative state-driven rendering is illustrated in Figure 1. In many production codebases for the two frameworks coexist, enabling gradual adoption of SwiftUI while retailing UIKit components remain practical [2], [3]. State driven UI development makes state ownership and propagation central architectural concerns. UI behavior is influenced by state changes originating from user input, asynchronous network responses, and local persistence updates [3], [4]. When state flow is not explicit, side effects become harder to trace, and UI inconsistencies become more likely [3], [4]. In SwiftUI based code, reactive updates are commonly supported through *ObservableObject* and property wrappers such as *@Publiched [3]*, often paired with Combine style data streams for asynchronous events [4]. Swift language documentation provides additional background on these mechanisms and related language features [5]

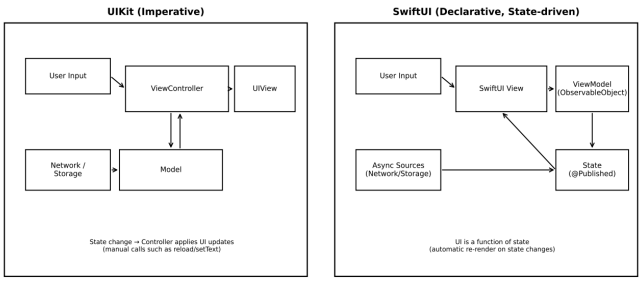

Figure 1. iOS UI paradigms and state propagation: UIKit imperative updates vs SwiftUI declarative rendering.

B. *Model-View-Controller (MVC)*

Model-View-Controller separates responsibilities into Model, View, and Controller. The model represents application data and operations, the view renders the interface, and the controller mediates user interactions by interpreting events, updating the model, and triggering UI updates [2] as shown on Figure 2. In iOS practice, MVC maps naturally to UIKit because screen logic is typically centered in *UIViewController* [2]. As applications grow, controllers often become the place where diverse responsibilities accumulate, which motivates the use of clearer responsibility boundaries in more structured approaches [1].

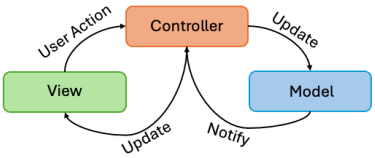

Figure 2. Model-View-Controller architecture

C. *Model-View-ViewModel (MVVM)*

Model-View-ViewModel introduces a ViewModel that owns presentation state and presentation logic, transforming model data into UI ready values and handling user actions in a testable way [6]. The view becomes a projection of state exposed by the ViewModel, which reduces direct coupling between UI rendering and underlying data [6], as shown on Figure 3. MVVM aligns especially well with SwiftUI become the framework is state driven and supports observation and binding, allowing UI to update automatically when ViewModel state changes [3]. In UIKit similar communication in commonly implemented through delegation, closures, or reactive streams [2], [4].

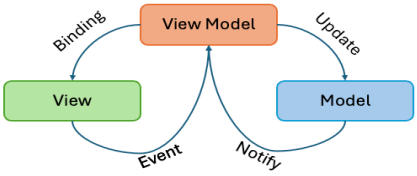

Figure 3. Model-View-ViewModel architecture

D. *Model-View-ViewModel-Coordinator (MVVM-C)*

MVVM-C extends MVVM by adding a Coordinator responsible for navigation and flow management, as shown Figure 4. The coordinator constructs screens, drives transitions, and centralizes multi-screen journeys, which helps keep navigation logic out of views and view models [7]. This is particularly useful in applications with onboarding flows, authentication, or complex multi-step user journeys, where routing decisions and shared state across screens can otherwise increase coupling. In practice, coordinators also become a natural place for assembling dependencies consistently [7].

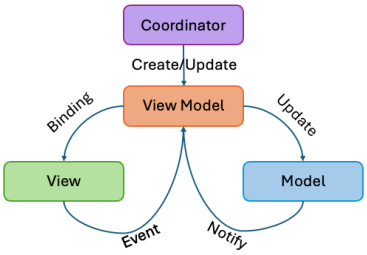

Figure 4. Model-View-ViewModel-Controller architecture

E. Model-View (MV)

Model-View is included as a lightweight baseline where the view renders model state directly with minimal structural overhead. The model acts as the source of truth, while the view reflects model changes, typically following a unidirectional flow from model to view, and process is shown on Figure 5. This style can fit small, state driven UI components well, especially in declarative environments where UI updates naturally flow state changes [3]. As application complexity grows, additional conventions are often needed to handle interaction logic and navigation consistently [1].

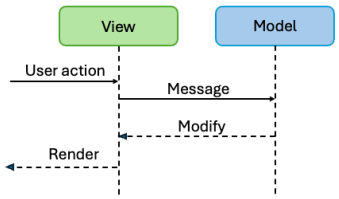

Figure 5. Model-View architecture

IV. Methodology

Separation of concerns refers to how clearly responsibilities are divided between UI rendering, presentation logic, state ownership, and flow coordination [1]. Testability captures how easily presentation behavior and state transitions can be validated with unit tests, ideally without requiring full UI execution [1]. State propagation describes how state is represented, owned, and delivered to the UI, including the clarity of data flow and control over UI driven side effects [3], [4]. Navigation and flow complexity considers how well the approach supports multi-screen journeys and conditional routing while keeping responsibilities localized [7]. Implementation overhead reflects the amount of structural and boilerplate code typically required to apply the approach consistently. Suitability for UIKit and SwiftUI considers how naturally an approach maps to UIKit's imperative model [2], SwiftUI's declarative state driven model, and hybrid codebases [2], [3].

A. Scoring

Each architecture is assessed qualitatively against the criteria using a five point scale. A score of one indicates that satisfying a criterion is typically difficult in realistic applications without additional conventions, while a score of five indicates strong support through clear responsibility boundaries and predictable behaviour [1]. Intermediate values represent trade-offs where the criterion can be achieved but usually requires extra discipline or supporting mechanisms. Scores are reported together with brief justifications to emphasize comparability rather than numerical precision [8].

B. Reference scenario

To ground the comparison, all approaches are considered under a representative iOS scenario consisting of authentication with error handling, a list and detail flow backed by asynchronous data loading, and a multi-step form requiring validation and conditional progression. The scenario includes shared state that must remain consistent across screens, such as user session status, loading and error states, and partially completed form data. This setup exercises core presentation layer concerns, particularly state propagation, UI triggered side effects, and navigation coordination across multiple screens [4], [7].

C. Results

Based on the criteria defined in previous section, the compared architectures are assessed qualitatively and summarized in a single comparison matrix. The goal of the matrix is to provide a compact overview of how each approach typically supports maintainability-related concerns in the iOS presentation layer, especially under state-driven UI development [3], [4] and multi-screen navigation [7]. Scores are not intended to be interpreted as precise measurements, but as a structured synthesis of responsibility placement and common implementation characteristics observed in practice [1].

Table 1 reports the qualitative scores for MVC, MVVM, MVVM-C, and MV across the selected criteria. A higher score indicates that the architecture generally supports the criterion with clearer boundaries and less reliance on additional conventions, while lower scores indicate that satisfying the criterion typically requires extra discipline, supporting mechanisms, or results in increased coupling [1].

TABLE I. QUALITATIVE COMPARISON CRITERIA AND SCORING SCALE

Criterion	MVC	MVVM	MVVM-C	MV
Seperation of concerns	2	4	5	3
Testability	2	4	4	3
State propagation	2	4	4	4
Navigation / flow handling	2	3	5	2
Implementation overhead	5	3	2	5
Suitability for UIKit	5	4	5	2
Suitability for SwiftUI	2	5	4	4

The matrix includes seven criteria that directly reflect the main sources of complexity in modern iOS presentation layers. Separation of concerns captures how clearly the approach divides UI rendering, presentation logic, state ownership, and coordination responsibilities [8]. MVC receives a lower score because, in UIKit practice, screen-level responsibilities frequently concentrate in view controllers as the application grows, which can blur boundaries between rendering, logic, and coordination [1]. MVVM scores higher because it explicitly introduces a ViewModel that owns presentation state and logic, reducing the tendency for UI components to accumulate non-UI responsibilities [1]. MVVM-C scores highest because it further separates navigation and flow orchestration into a dedicated Coordinator [7], strengthening responsibility boundaries at scale. MV is scored in the middle because it can keep view and model roles clean in simple cases, but often requires additional conventions once interaction logic and coordination become non-trivial [1].

Testability reflects whether presentation behaviour and state transitions can be validated with unit tests without running the full UI. MVC is scored lower because logic frequently lives inside view controllers, which are tightly coupled to UIKit lifecycle and UI dependencies [2]. MVVM improves testability by relocating presentation behaviour into the ViewModel, which is typically easier to isolate and test [1]. MVVM-C maintains similar testability, with the

additional benefit that navigation decisions can be tested separately at the coordinator level [7]. MV supports testing of model logic reasonably well, but can become harder to test consistently when ad-hoc glue code grows around interaction handling [1].State propagation evaluates how clearly state is represented and delivered to the UI. MVVM and SwiftUI align strongly here because UI is naturally derived from observable state, making propagation explicit. MVC is scored lower because state updates often depend on manual coordination and explicit UI refresh calls, which increases the risk of scattered updates and inconsistent UI as complexity grows [1]. MVVM-C inherits MVVM's strengths in state propagation, while MV can also score well in state-driven UI settings, provided that state ownership remains explicit and side effects are controlled.

Navigation and flow handling is separated as its own criterion because multi-screen journeys become a dominant source of complexity in larger applications. MVVM-C scores highest due to the explicit Coordinator role, which centralizes routing and flow control [7]. MVC and MV are scored lower because navigation is often embedded within screen controllers or views, increasing coupling between screens and making flows harder to evolve [1], [2]. MVVM sits in the middle; it can support navigation cleanly, but without a dedicated coordination layer navigation responsibilities may still spread across views or view models [1].Implementation overhead captures the typical structural cost of applying the approach consistently. MVC and MV score highest on low overhead because they require fewer supporting types and less boilerplate for simple applications [2]. MVVM introduces moderate overhead due to ViewModel types and explicit state interfaces [6]. MVVM-C has the highest overhead because coordinators add additional components and conventions [7], which is justified primarily when flows and feature scope demand that structure.

Finally, suitability for UIKit and suitability for SwiftUI reflect how naturally each architecture maps to the underlying UI paradigm. MVC remains a strong fit for UIKit due to its alignment with view controllers and UIKit's imperative update model [2]. MVVM also fits UIKit well, but typically requires explicit state-to-UI wiring (delegation, closures, or reactive streams) [2], [4]. SwiftUI's declarative rendering strongly favors approaches centered on observable state [3], which is why MVVM scores highest for SwiftUI [3], [6]. MVVM-C remains highly suitable when navigation and flow complexity is a primary concern in SwiftUI or hybrid systems [3], [7]. MV scores lower for UIKit because UIKit screens still require view controllers [2], while it scores relatively higher for SwiftUI because unidirectional, state-driven rendering can be expressed naturally [3].

V. Conclusion

Architectural choices in the iOS presentation layer have a direct impact on maintainability, testability, and the ability to evolve an application as its scope and navigation structure grow [1]. The qualitative comparison presented in this work highlights that there is no universally optimal architecture. Instead, suitability depends on application complexity, the dominant UI paradigm [2], [3], and the expected growth of navigation flows and shared state [1].For small UIKit applications with limited navigation and relatively simple state, MVC remains a pragmatic choice due to its low implementation overhead and its natural alignment with UIKit conventions [2]. It is particularly appropriate for prototypes, proof of concepts, and small products where fast delivery is the primary constraint and the risk of complex flows is low. However, as features accumulate, additional discipline is required to prevent excessive responsibility concentration in view controllers [2].

For SwiftUI-first applications and screens where UI is largely a projection of observable state, MVVM is generally the most suitable default [3], [6]. It supports a clear separation between rendering and presentation state, encourages predictable state propagation, and improves unit test feasibility by isolating presentation behavior in ViewModels [6]. MVVM is recommended for medium-sized applications where state and asynchronous data loading are common, but navigation remains manageable without extensive flow orchestration [1].

When applications involve complex multi-screen journeys such as onboarding, authentication, deep navigation hierarchies, or multi-step forms with conditional routing, MVVM-C becomes advantageous [7]. Introducing a Coordinator centralizes navigation decisions and flow control [7], reduces coupling between screens, and supports modular evolution of navigation graphs [1]. Although MVVM-C increases structural overhead, it is justified in scenarios where navigation and flow management are a dominant source of complexity and where long-term scalability is a key requirement [1]. Finally, MV can be used as a lightweight baseline for simple, state-driven UI components, particularly in SwiftUI contexts where unidirectional state-to-view rendering is natural [3]. It can be effective for small features or isolated screens with minimal interaction logic. As complexity increases, however, MV typically requires additional conventions to handle interaction logic and navigation consistently [1], which often leads teams toward MVVM or MVVM-C for larger systems [1].

Acknowledgment

This work has been supported by the Ministry of Education, Science and Technological Development of the Republic of Serbia.

References

[1] R. C. Martin, *Clean Architecture: A Craftsman's Guide to Software Structure and Design*. Prentice Hall, 2017.

[2] Apple Inc., "UIKit," *Apple Developer Documentation*, accessed Jan. 26, 2026. Available: https://developer.apple.com/documentation/uikit.

[3] Apple Inc., "SwiftUI," *Apple Developer Documentation*, accessed Jan. 26, 2026. Available: https://developer.apple.com/documentation/swiftui

[4] Apple Inc., "Combine," *Apple Developer Documentation*, accessed Jan. 26, 2026. Available: https://developer.apple.com/documentation/combine

[5] Apple Inc., "The Swift Programming Language (6.2.3)," *Swift.org Documentation*, accessed Jan. 26, 2026. Available: https://docs.swift.org/swift-book/documentation/the-swift-programming-language

[6] YM. Fowler, "Presentation Model," 2004, accessed Jan. 26, 2026. Available: https://martinfowler.com/eaaDev/PresentationModel.html

[7] S. Khanlou, "Coordinators Redux," 2015, accessed Jan. 26, 2026. Available: https://khanlou.com/2015/10/coordinators-redux

[8] E. Gamma, R. Helm, R. Johnson, and J. Vlissides, *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1994.