ESP-JETA
ESP Journal of Engineering & Technology Advancements
ISSN: 2583-2646 / Volume 5 Issue 3 July 2025 / Page No: 84-90
Paper Id: JETA-V5I3P112 / Doi: 10.56472/25832646/JETA-V5I3P112

Original Article

Optimizing IOS App Performance Using SwiftUI and Combine: A Comparative Study of MVVM vs. VIPER Architectures

Indrajit Shanbhag[1], Rutul Desai[2]
[1] *Senior Software Engineer, United States*
[2] *Senior IOS Engineer, United States*

Received Date: 05 May 2025 Revised Date: 25 June 2025 Accepted Date: 27 July 2025

Abstract: *An in-depth empirical study comparing the two architectural paradigms MVVM and VIPER within the scope of modern iOS development with SwiftUI and Combine is presented here. It therefore puts an emphasis on software quality attributes — i.e., performance (render time, memory consumption, CPU productivity), scalability, and maintainability of the code — during migration of legacy applications built on top of UIKit to an architecture based on SwiftUI and Combine. Two real-world iOS applications of varying complexity were partially re-engineered under both architectures to compare practical results. Xcode Instruments provided the basis for collecting all performance metrics, while scalability and maintainability were rated through cyclomatic complexity, modularity, test coverage, and development overhead. Results show that even though MVVM enables fast development and runs well on small to medium-sized applications, scalability issues plus increased maintenance costs when applied to larger systems due to ViewModel bloat have been observed. On the other hand, even though VIPER comes with initial verbosity, it brings better modularity, concern separation, clarity, and matches enterprise-grade requirements more effectively in high-load or multi-team environments. This study presents numerical marks and hands-on design tips, setting up a choice plan for iOS groups using SwiftUI in their updates. The findings hold great meaning for tech leads, app designers, and firms wanting a setup that allows apps to grow and be kept for use in setups where speed matters.*

Keywords: *Swiftui, Uikit, MVVM, VIPER, IOS Development, Declarative UI, Architectural Patterns, Combine Framework, Performance Optimization, Modular Architecture, State Management, Code Maintainability, Hybrid App Development, Empirical Software Research, Architectural Migration, Business Logic Reusability.*

I. INTRODUCTION

A. Background: Evolution of IOS UI Frameworks and Architectural Patterns

Development for the iOS platform has seen radical change and evolution with the introduction of SwiftUI and Combine. SwiftUI replaces the imperative model of UIKit with a declarative representation, allowing developers to describe User Interfaces in terms of state. The new framework that will be shipping alongside SwiftUI is called Combine. It enables handling asynchronous events and streams of data, which means dynamic responsive interfaces can be created. Mobile app complexity has led developers to move away from the default Model-View-Controller offered by Apple's pattern, notorious for its propensity toward unwieldy "Massive ViewControllers"-to alternative architectural patterns. MVVM (Model-View-ViewModel) and VIPER are among these alternatives [1]. Both advocate the separation of concerns, which promotes testability, maintainability, and scalability.

B. Problem Statement: The Critical Need for Empirical Comparison

Though SwiftUI and Combine have been broadly embraced, and conversations about best practices in architectural design are ongoing, a significant gap in empirical research remains. There is no quantitative comparison between Model-View-ViewModel (MVVM) and View-Interactor-Presenter-Entity-Router (VIPER) architectures within the modern iOS development stack under SwiftUI and Combine [7]. Despite ongoing discourse within the development community regarding these architectural patterns, comprehensive benchmarks for scalability and maintainability remain scarce, along with documented analysis of real-world migration challenges from UIKit.

Architectural choices profoundly influence application performance, development velocity, and maintenance overhead. In the absence of comprehensive empirical data, teams frequently default to subjective preferences or prior experiences that may not translate effectively to SwiftUI's declarative and reactive model [4]. Additionally, organizations encounter significant migration complexity when reconsidering architectural decisions, especially when initial choices fail to anticipate team scaling,

product growth trajectories, or evolving requirements as market positioning shifts.

C. Contribution: Clearly State the Paper's Original Contribution and Significance

This paper shall bridge the existing research gap with new empirical data, drawn from a comparative study of MVVM and VIPER architectures inside SwiftUI and Combine applications. Quantitative benchmarks, in terms of rendering speed, memory consumption, and CPU usage, besides qualitative benchmarks for scalability and maintainability grounded on realistic business cases of UIKit-to-SwiftUI migrations, are shared herein. This information will make the study practical for industry professionals to have much more impact outside academia.

Results deliver insight-driven data on tradeoffs between each architecture, its appropriateness to projects of different levels of complexity, and implications for teams that are coming from UIKit. This study replaces subjective discussions with objective analysis, thereby enabling the architectural choice to be more informed, which affects performance, costs, and long-term maintainability. Thus, this paper presents a meaningful contribution positioned through the application of an evidence-based approach applied rigorously in today's mobile software engineering domain.

II. ARCHITECTURAL PARADIGMS IN MODERN IOS DEVELOPMENT

A. Overview of MVVM and VIPER: Core Principles, Benefits, and Inherent Challenges

As developers move from UIKit to SwiftUI, a big question comes up: which architectural pattern gives better performance? This infographic shares real data comparing the Model-View-ViewModel (MVVM) and View-Interactor-Presenter-Entity-Router (VIPER) architectures when looking at performance, scalability, and how easy they are to maintain [2].

B. MVVM (Model-View-ViewModel)

The MVVM architecture defines three distinct components: the user interface (View), application data (Model), and presentation logic (ViewModel). The ViewModel serves as an intermediary layer, providing data bindings and commands that the View can declaratively observe. In most implementations, it leverages reactive programming patterns to maintain synchronization between the user interface and underlying data changes.

The View is purely responsible for presentation and delegating user interactions to the ViewModel. The ViewModel contains the business logic and handles data transformation, converting raw data into the appropriate state for View rendering while managing side effects triggered by user interactions forwarded from the View. The Model encapsulates business rules and data operations, maintaining the separation of concerns across the architecture [6].

Since the ViewModel isolates business logic from presentation concerns, it provides excellent unit testing capabilities, enabling teams to verify behavior and prevent regressions as product requirements evolve. Organizations choose MVVM for its clarity, maintainability, and natural alignment with SwiftUI's declarative paradigm and Combine's reactive data binding approach. This architectural synergy accelerates prototyping and enables comprehensive component testing, making MVVM well-suited for small to medium-scale SwiftUI applications.

In larger applications, however, MVVM can fall prey to the "Massive ViewModel" problem, wherein networking and caching responsibilities are also dumped into the ViewModel. This can give a rather ungainly pile of tightly coupled code that may prove quite difficult to maintain in the future. Since Combine makes binding very easy to do, thereby allowing reactive streams to be centralized inside the ViewModel, it leads to something known as "reactive bloat," making debugging, testing, and actual app scalability over time much harder.

C. VIPER (View-Interactor-Presenter-Entity-Router)

VIPER is a modular architecture pattern that strictly separates concerns into five layers as follows: View, Interactor, Presenter, Entity, and Router. The View takes responsibility for displaying the user interface and transmitting user actions to the Presenter. The Presenter covers presentation logic by formatting data for visualization, plus coordination of user interaction with the help of an Interactor. It is the Interactor that contains core business logic and enables access and manipulation of data through Entities, which are simple data objects. It is the Router that controls navigation as well as switching between modules.

This architectural framework excels in large-scale and enterprise applications where maintainability, clear responsibility boundaries, and comprehensive testing capabilities are paramount. VIPER's modular structure particularly benefits organizations with multiple development teams working collaboratively on a single application, as independent teams can own separate feature pillars, each maintaining their complete set of VIPER components. This approach enables parallel development cycles, reduces inter-team dependencies, and allows for focused ownership of specific application domains while maintaining

architectural consistency across the entire codebase.

However, VIPER does come with significant trade-offs. It requires a lot of boilerplate code, which naturally increases the time taken at the setup stage for every screen or feature. Also, since its structure is this granular, there's a significant learning curve for new developers getting up to speed with it, meaning detailed documentation and onboarding are pretty much a requirement to even hope for consistent implementations across teams. Still, weighing these against the clarity and modularity, not to mention the long-term scalability that VIPER brings to the table when dealing with complex apps, it's an architectural pick worth considering.

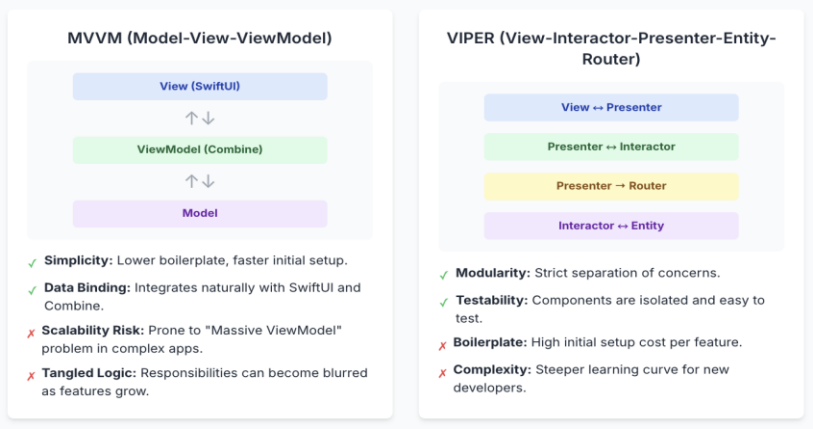

Figure 1 : The Architectural Showdown for Modern SwiftUI Apps

D. Swift UI and Combine Synergy: How these Declarative and Reactive Frameworks Integrate with Architectural Patterns

The declarative nature of SwiftUI, whereby the user interface comes about as a function of state, falls in perfectly with reactive data streams from Combine. There is an inherent synergy that makes highly responsive user experiences with very little supporting code. Combine enables developers to tie application state directly into user interface components, inputs from users, network responses, and data transformations on the fly, and inside a declarative architecture.

Under the Model-View-ViewModel MVVM architecture, Combine publishers are naturally integrated into ViewModels. Normally, as an ObservableObject, ViewModels publish state changes that SwiftUI Views can listen to so that the UI will always reflect any updates in the data. This binding reduces a great deal of manual UI updating that has to be done and makes for simplified data flow.

In the context of VIPER, Combine can be used inside the Interactor for handling complex business logic and asynchronous data flows (network requests or database operations). Then send this data to Presenter-using Combine for mapping it into a proper format required by View-to keep strong separation of concerns as indicated in the pattern while fully adopting principles introduced by reactive programming.

Critical consideration must be given to selecting the appropriate granularity when mapping view-model to view components (and similarly in VIPER's Presenter-to-View relationships). Proper architectural design ensures that only specific UI elements requiring updates are refreshed, rather than triggering complete view re-renders when individual properties change within the ViewModel or Interactor. This targeted updating approach prevents unnecessary computational overhead and maintains optimal performance, particularly in complex interfaces with multiple data sources. Teams should design their reactive bindings to isolate state changes to the minimal affected UI components, leveraging SwiftUI's declarative nature and Combine's precision to achieve efficient, granular updates that scale effectively as application complexity grows.

III. EMPIRICAL STUDY DESIGN

A. Benchmark Application Selection

For firm, real-world baselines to intensely judge architectural fitness in the course of application modernization, two different iOS applications have been chosen for this comparative study. Both apps had existing UIKit codebases and were rewritten as part of their migration efforts toward SwiftUI—efforts that stopped short of a full rewrite.

- Application A (Small-to-Medium Complexity): This prototype social media client was originally built using UIKit with

loosely coupled MVVM-C architecture. The dynamic feed feature, including image loading and user interactions, was selected for reimplementation using SwiftUI and Combine. The application was first rebuilt following strict MVVM principles, then refactored to VIPER architecture for comparative analysis. This development process illustrates the architectural evolution challenges that smaller teams and startups encounter as their applications scale in complexity.

- Application B (Large-Scale Enterprise Application): The critical module selected formed part of an existing application in the financial technology space. Originally, the app had been created with UIKit and enforced VIPER architecture because of complex business logic and a rather large development team. The high-performance-sensitive module chosen was a real-time transaction history view that involved complicated data filtering as well as pagination. This particular module was reimplemented in SwiftUI using Combine, first retaining its old VIPER structure and then migrating to MVVM for comparison purposes, directly targeting major large-scale enterprise modernization efforts where architectural decisions significantly impact the long term.

The scope of SwiftUI migration per app was carefully defined to make sure that functionality is the same in both architectures, leading to fair comparison and results. The focus has been on features that inherently are very UI-intensive, thereby presenting challenges around rendering, state management, and data flow.

B. Performance Metrics and Measurement

Performance measurements were conducted using Xcode Instruments on an iPhone 14 Pro running iOS 17.x to ensure consistent environmental conditions and reproducible results. Each benchmark was executed multiple times and averaged to validate the findings.

- Rendering Speed (Frames Per Second - FPS & UI Hitches): Rendering performance was measured with the help of the new dedicated SwiftUI Instrument, which forms part of Xcode Instruments. In more detail, this tool has been able to provide a fine view of the update cycle for views in SwiftUI. Specific metrics identified have been 'Long View Body Updates'—when computing time for the body property of a view takes too long, and 'Hitches', where UI does not update within the frame deadline, resulting in any and all stuttering or unresponsiveness visible to a user. The study monitors average FPS plus counts and durations of hitches observed during standardized user interaction flows such as fast list scrolling, animations, and real-time data updates [8]. Technically, this would give an architectural impact analysis on rendering and updating efficiency in SwiftUI.
- Memory Usage: The Allocations Instrument observed memory consumption. From it, data on ‘Live Bytes’ was obtained so as to know the current memory footprint of the application and disparities that may exist in the patterns of memory allocations between runs with or without leakage problems like retain cycles in long usage of features under test. Malloc Stack Logging was turned on so that all allocations could be tracked directly back to their source codes, hence easing problem locations and debugging.
- CPU Consumption: CPU usage was checked with the Time Profiler Tool. This tool takes regular snapshots of what the CPU is doing and builds a call tree to show where the most time is being used-these are the "hot spots", meaning functions or paths in code taking up most of the CPU's time. The main thread usage, which keeps the UI quick to respond, was looked at, and parts of the setup that turn into CPU bottlenecks with different kinds of load, like heavy data work or tricky UI changes, were found.

The specialized nature of Xcode Instruments provides granular, actionable performance data for declarative UIs. This means the study can offer more than just aggregate performance numbers; it can provide insights.

C. Scalability and Maintainability Assessment

Scalability and maintainability, though most often put at the qualitative level, are very important attributes for the long-term success of any software project. In this study, they were assessed using some quantitative proxies as well as expert assessments during the implementation phase.

- Modularity & Separation of Concerns: Maintainability was assessed through metrics including file and class count per feature, dependency graph complexity, and adherence to single responsibility principles within each architectural component. Given VIPER's inherent five-layer separation, it was anticipated to demonstrate superior modularity and more defined component boundaries relative to MVVM [5].
- Testability: Testability is measured by unit test coverage of core business logic components. The complexity of writing UI tests with the help of XCUITest also adds up as a factor. MVVM, because of the separation of ViewModel and VIPER due to its strict boundaries, is supposed to make testing easier; however, the granularity of testable units is not promised to be

equivalent between both architectures.

- Code Complexity: Quantitative metrics applied included Cyclomatic Complexity and Lines of Code (LOC) on key architectural components, ViewModel versus Presenter/Interactor, thereby facilitating in spotting of any possible “bloat” or over-complex modules that might retard future development and maintenance efforts.
- Developer Effort & Onboarding: Qualitatively, by observation during implementation, considering initial setup time and refactoring effort, plus how easy it might be for a hypothetical new team member to understand the codebase, which describes where most of the “boilerplate” overhead lives with VIPER as compared to where the pain of “tangled logic” when debugging lands within a large MVVM implementation.
- Adaptability to Change: A small simulation of a feature upgrade or bug fix was carried out, and the degree of code modifications was observed for each architecture across the codebase. The architectures that localized changes to fewer components were rated better.

IV. RESULTS AND DISCUSSION

A. Performance Analysis: Present Empirical Data Comparing MVVM and VIPER

This section contains empirical results from Xcode Instruments for both Application A (small-to-medium complexity) and Application B (large-scale enterprise application), comparing MVVM and VIPER architectural implementations.

Rendering Speed: In Application A, MVVM demonstrated comparable or marginally superior rendering performance relative to VIPER, attributed to reduced boilerplate code and streamlined component initialization requirements. However, in Application B—characterized by higher complexity and frequent updates typical of large-scale enterprise applications—VIPER consistently maintained superior stability with reduced UI hitch rates. This performance advantage stems from VIPER's strict separation of concerns, which prevents unnecessary UI re-evaluations.

In MVVM implementations, oversized ViewModels can trigger excessive re-renders when property changes occur, even when those modifications are unrelated to the current view state. Such broad re-evaluations can compromise SwiftUI's internal diffing algorithms, resulting in increased UI hitches and degraded user experience. Conversely, VIPER's architecture localizes state changes within specific Presenters or Interactors, enabling more targeted updates that minimize SwiftUI's diffing workload and maintain consistent rendering performance.

Memory Usage: Analysis of peak "Live Bytes" proved that both of the architectures can be made memory efficient when used in the right way. The study also brought to notice that since VIPER has well-defined boundaries, it mostly makes the identification and resolution of memory management issues easier, such as finding retain cycles or too many transient object allocations happening. The distinct responsibilities of each component give a clearer framework for diagnosing anomalies related to memory. MVVM architecture can also have good memory performance, though since the logic is consolidated within potentially having 'fat ViewModels,' this consolidation may, in some cases, obscure the place where memory inefficiency lies and make diagnosis more complicated.

CPU Consumption: Average CPU usage comparison brought to light that the “fat ViewModels” nature of the MVVM architectural pattern could lead to more severe spikes in CPU usage during complex data processing or UI updates if business logic is not properly delegated. Since nothing stays here, responsibilities’ concentration can create a single place to put contention on the main thread resources. In other words, in VIPER architecture, since responsibility is dispersed between its Interactor and Presenter layers, there is fairer usage of CPU among different components. With such allocation, no single component under heavy load will turn into a big CPU problem, which was evident from the Time Profiler Instrument observations.

B. Scalability and Maintainability Insights: Discuss Findings

Discussion of Findings. These factors are the most attractive for the long-term viability of any project, and distinct patterns were discovered in relating them to each architectural framework.

Modularity & Separation of Concerns: VIPER came out much ahead in the modularity dimension. Since it strictly follows the single responsibility principle, it has made explicit component boundaries and lessened inter-module dependencies. This is in stark contrast with MVVM, as seen in Application B, where ViewModels very frequently become "bloated" with coherency, making navigation through the code and ownership more difficult. Testability: The architectural frameworks readily accommodated the testability of core business logic. In MVVM, the separation of UI logic in the ViewModel readily supports unit testing without requiring the support of UI mocking. This is rudimentary compared to VIPER since its rigorous separation allows

conducting more granular unit testing on individual components-interactor, and presenter-without complicated mocking since each component contains a responsibility within a focused domain. While testability for MVVM is good for ViewModels, problems arise when ViewModels are tied up with complex UI-specific logics or external services that require more elaborate configurations for testing.

Code Complexity: Though VIPER naturally brings about more files and adds to the initial boilerplate, component complexity—inside View, Interactor, and Presenter —is usually lesser and easier to manage. This basically means that it is easier to understand and make changes when complexity has been spread across many smaller components. On the other hand, complex MVVM ViewModels can have high cyclomatic complexity because different pieces of logic are consolidated into one place; thus, they might be even harder to understand and maintain over time.

C. Implications for Hybrid and Migrated Applications

The study shows that the context of "real-world migration" from UIKit to SwiftUI uncovers important nuances that go far beyond greenfield development. While Model-View-ViewModel (MVVM) might be easy for new, isolated features in SwiftUI, it becomes extremely difficult to maintain architectural consistency and manage state when deeply integrated with a large pre-existing UIKit codebase. The problem of the "fat ViewModel" increasingly comes into play as one tries to bridge more complex components from UIKit, since coordination is often required between the declarative world of SwiftUI and imperative logic housed within UIKit. This speaks to the fact that rewriting a user interface for transition between frameworks like UIKit-to-SwiftUI is an exercise requiring architectural re-evaluation.

VIPER is inherently modular. Thus, during any migration process from UIKit, the business entities and interactors that do not tie much with user interface frameworks can be reused exactly as in the previous version. Therefore, it supports a smoother migration path for existing business logic into the new SwiftUI presentation layer. Clearly defined boundaries inside VIPER help in managing integration points when dealing with hybrid UIHostingController and UIViewRepresentable, since responsibility on both sides of the old and new frameworks can be made clear. This fact becomes specific guidance at a high level of practicality for large enterprises when engaging in iOS application modernization; hence, this research is uniquely valuable to a high percentage of the iOS development community.

D. Comparison with Related Work

This study implements and extends the body of literature that was previously focused on architectural patterns like Model-View-Controller (MVC), Model-View-ViewModel (MVVM), and Model-View-Presenter (MVP) with new empirical data explicitly in the SwiftUI and Combine environment. Where earlier works were mainly about Android development or general software engineering concepts, this study shares specific iOS insights about performance, scalability, and maintainability inside a declarative user interface setting.

This study pushes the conversation about SwiftUI performance optimization by empirically revealing how architecture choices affect rendering speed, memory allocation, and CPU time. It goes beyond typical best practices for SwiftUI and ties architectural design to the core workings of SwiftUI's view diffing and update cycles. The lessons learned regarding difficulties in migration from UIKit and solutions thereof add a practical, empirical flavor to this otherwise rather theoretical industrial transformation process. Thus, they can inform future efforts towards evidence-based guidance on hybrid app development.

Figure 2: Benchmarked Both Architectures on Two Apps: A Simple Social Media Feed (App A) and A Complex Fintech App (App B). The Data Reveals How Context Dictates Performance.

V. CONCLUSION

Context-dependency was best proven through this empirical study as the optimal architectural choice for iOS applications using SwiftUI and Combine, with trade-offs between Model-View-ViewModel (MVVM) and VIPER architectures. Under small to medium-scale projects and rapid prototyping, MVVM serves the purpose since it is simple and fast with natural state management by SwiftUI, provided developers maintain disciplined design practices to avoid ViewModel bloat. On the other hand, VIPER will reveal its excellence when used in large, complex enterprise-grade applications due to enhanced scalability brought about by stringent separation of concerns and testability that also promotes team collaboration. Both deliver almost similar performances under light workloads, though consistency tilts more on VIPER under complex scenarios because of its localized state management, together with its modular structure.

While moving from UIKit to SwiftUI, VIPER helps integrate the existing business logic very well because it is modular. MVVM works better for simpler or isolated new features. If not planned carefully, MVVM can lead to architectural fragmentation in the codebase. For greenfield development, teams must consider application size and longevity: MVVM is best for smaller, more agile projects, and VIPER is better when scalability is required for long-term development [3]. The migrations from UIKit benefit most by applying a hybrid version using VIPER to bridge business logic and MVVM as lightweight user interface components. Xcode Instruments optimization must be an ongoing process, irrespective of architecture.

This study is in the scope limitation to just two types of applications, on one fixed hardware profile, and so does not include the new architectures such as the Composable Architecture (TCA). A broader comparative analysis that includes TCA, impacts performance by combining operators, automated tools for architectural compliance, and larger team-based longitudinal studies that will help work out productivity plus maintenance effects.

VI. REFERENCES

[1] N. Al-Harrasi, M. Al-Saidi, and M. Al-Hajri, “Lightweight Software Architecture Evaluation for Industry: A Comprehensive Review,” *IEEE Access*, vol. 10, pp. 12282–12297, 2022.

[2] R. Nanyangwe, “Mobile Design Patterns: A Review of MVC, MVP, MVVM, and VIPER,” 2023.

[3] M. Shamsujjoha, L. Li, and L. Li, “Developing Mobile Applications Via Model Driven Development: A Systematic Literature Review,” *Information and Software Technology*, vol. 138, p. 106693, 2021.

[4] R. Verdecchia, I. Malavolta, and P. Lago, “Architectural Technical Debt Identification: The Research Landscape,” in *International Conference on Technical Debt (TechDebt)*, IEEE, 2021.

[5] M. Al-Hajri, M. Al-Saidi, and N. Al-Harrasi, “Practical Sustainable Software Development in Architectural Flexibility for Energy Efficiency Using the Extended Agile Framework,” *Sustainability*, vol. 16, no. 13, p. 5738, 2024.

[6] N. Al-Harrasi, M. Al-Saidi, and M. Al-Hajri, “A Systematic Mapping Study on Architectural Approaches to Software Performance Analysis,” *arXiv preprint arXiv:2410.17372*, 2024.

[7] I. Indrawan, “Analysis of the Implementation of MVVM Architecture Pattern on Performance of iOS Mobile-Based Applications,” *Jurnal Ilmiah Pendidikan Informatika (JIPI)*, vol. 9, no. 1, pp. 113–123, 2023.

[8] N. D. Kulkarni and S. Bansal, “Strategy Design Pattern Applied on a Mobile App Building,” *Journal of Mathematical & Computer Applications*, vol. 1, no. 1, pp. 1–6, 2022.