



30-Day Android Challenge

From Basics to Advanced

(Expert-Level Deconstruction 📱)

🚀 Day 22

A Deep Dive into MVVM & MVI

Your App's Architecture is its Future! A Deep Dive into MVVM & MVI – From Basics to Scalable Challenges ⚡



Esmaeeil Moradi

@esmaeeil-moradi700



Day 22



Your App's Architecture is its Future! A Deep Dive into MVVM & MVI – From Basics to Scalable Challenges ⚡

Hey Android Architects, Developers, and Visionary Designers!

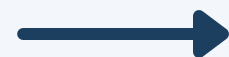
In the world of Android development, choosing the right architecture isn't just a technical decision; it's a long-term investment in your project's stability, maintainability, and scalability. Today, on Day 22 of the #30DayAndroidChallenge, we go beyond superficial definitions to explore the deepest layers of two popular and powerful architectural patterns: MVVM (Model-View-ViewModel) and MVI (Model-View-Intent). This article is a comprehensive roadmap for anyone, from Junior to Architect, aspiring to master the creation of next-generation Android applications.

As I've discussed in my previous LinkedIn articles on *Clean Architecture, MVVM, and MVI individually*, today we complete the puzzle. We'll delve into advanced considerations regarding their combination and selection. This article will help you not only grasp the concepts but also identify large-scale implementation challenges and find practical solutions, understanding how these patterns integrate with *Clean Architecture* to build seamless systems.



Esmaeel Moradi

@esmaeel-moradi700



Introduction:

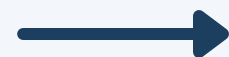
Architecture – The Backbone of Large Android Applications

As applications grow in complexity and development teams expand, the importance of architecture as the project's backbone becomes increasingly evident. Poor architecture can quickly lead to Spaghetti Code, testing nightmares, and exorbitant maintenance costs. Conversely, a strong architecture paves the way for growth, change, and innovation. MVVM and MVI are currently two of the most widely used and powerful patterns for managing the Presentation Layer in Android.

But do we truly understand these patterns? Do we know how to leverage their full potential in large-scale projects and avoid common Anti-Patterns? This article will help you go beyond superficial definitions and dive into their nuances and challenges.



Esmaeel Moradi
@esmaeeil-moradi700



MVVM (Model-View-ViewModel): Evolution and Scalability Challenges

The MVVM pattern quickly gained popularity due to its simplicity and deep integration with Jetpack Components. It divides responsibilities into three main parts: Model (responsible for data management and business logic), View (responsible for rendering UI and observing the ViewModel), and ViewModel (the intermediary between View and Model, exposing data via LiveData or StateFlow).

Why did MVVM become so popular?

- **Jetpack Integration:** MVVM naturally aligns with LiveData, Data Binding, and ViewModel from Jetpack, simplifying UI updates with reactive programming.
- **Lifecycle Awareness:** ViewModels can survive configuration changes (like screen rotations), preventing memory leaks.
- **High Testability:** By separating UI logic from the View, ViewModels are easily testable.



Esmaeel Moradi
@esmaeel-moradi700



Challenges of MVVM in Large Projects

(The Dark Side of MVVM):

As projects grow, MVVM also faces challenges that, if not managed correctly, can lead to Anti-Patterns:

- **Managing One-shot Events:** LiveData is excellent for managing UI state that needs continuous observation. However, for one-shot events like displaying a Toast, Navigation, or showing a Snackbar, using LiveData can lead to the SingleLiveEvent Anti-Pattern (events are re-emitted after configuration changes).
- **Modern Solution:** Using SharedFlow for One-shot Events and StateFlow for UI State is the best approach. SharedFlow allows event emission to multiple Collectors without needing an initial value or distinctness checks. Furthermore, emphasizing the use of repeatOnLifecycle in the View to collect Flows only in active Lifecycle states prevents unnecessary collections and re-emission issues.



Esmaeel Moradi

@esmaeel-moradi700



Complexity of Large ViewModels (God ViewModels):

In large projects, there's a temptation to put all business logic into ViewModels, leading to very large and difficult-to-maintain and test ViewModels.

- **Solution:** Employ Use Cases or Interactors to encapsulate specific business logic. This keeps ViewModels smaller, making them merely intermediaries between the View and the Use Cases. This approach also benefits from Dependency Injection[You can read these articles ["Dagger-Hilt, Koin, or Manual DI?"](#) ["Dependency Injection Basics"](#) ["Advanced Hilt & Dagger Deep Dive!"](#)] (as explained in my ["7 Proven Practices for Scalable Android Architecture"](#) article) to easily inject Use Cases into ViewModels.



Esmaeel Moradi

@esmaeel-moradi700



Testability:

Challenges of LiveData in Unit Tests:

- Testing LiveData traditionally required specific rules like `InstantTaskExecutorRule` to immediately execute operations on the Main Thread.
 - **Solution:** Migrating to `StateFlow` and `SharedFlow` (which we covered on "Day 21") naturally improves testability. Using `runTest` from the `kotlinx-coroutines-test` library (also covered on Day 21) allows precise control over time and Coroutine execution in tests, making testing ViewModels that use Flow much simpler and more reliable.

◦

Performance Optimization with MVVM:

- Correct use of `StateFlow` and `distinctUntilChanged()` to prevent unnecessary UI recompositions by emitting only distinct values.



Esmaeel Moradi

@esmaeel-moradi700

MVI (Model-View-Intent): Clarity and Determinism in State Management



The MVI pattern brings unparalleled clarity by focusing on deterministic state management and a Unidirectional Data Flow. In MVI, the Model represents a single, immutable UI State. The View renders this state and sends user Intents (e.g., button clicks). The ViewModel (which acts as a Processor or Store in MVI) processes Intents, updates the state, and publishes the new state.

Why is MVI powerful for complex state management?

- **Unidirectional Data Flow:** All state changes follow a clear path (Intent -> Model -> View), making Debugging much simpler.
- **Immutable State:** By creating new instances of State instead of modifying existing ones, unintended Side Effects are prevented, and Debugging is simplified due to a single source of truth.
- **High Predictability:** Due to the clear data flow and immutable state, the application's behavior can be easily predicted, which is ideal for apps with complex state logic (like forms or real-time dashboards).



Esmaeel Moradi

@esmaeeil-moradi700



The Concept of Intent in MVI:

Intent in MVI is completely different from an Android Intent. In MVI, Intent refers to the user's intention or an action that causes a state change.

- ***User Intent:*** *An action originating from the View (e.g., ClickIntent, TextChangeIntent).*
- ***System Intent:*** *An action originating from the Model or Use Case (e.g., DataLoadedIntent, ErrorOccurredIntent).*

Single Source of Truth and Reducer in MVI:

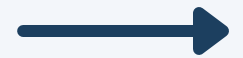
At the heart of MVI is the concept of a Single Source of Truth for the UI state, managed through a Reducer function. A Reducer is a pure function that takes the current State and an Intent as input and returns a New State. It has no Side Effects and always produces the same output for the same input.



Esmaeeil Moradi

@esmaeeil-moradi700

Challenges of MVI in Large Projects:



- **Boilerplate Code:** MVI implementations can involve a lot of Boilerplate for defining Intents, States, and Reducers.
 - **Solution:** Utilizing Code Generation tools, Custom DSLs, or small helper libraries can help reduce this Boilerplate.
- **Managing Side Effects:** (e.g., navigation, displaying Toast, showing a Dialog) Since the State must be immutable and only hold UI State, Side Effects must be managed separately.
 - **Solution:** Employing the concept of an Effect (sometimes called a Single Event or Command) which is a one-shot event consumed by the View. These Effects can also be implemented with SharedFlow.
- **Complexity for Juniors:** The learning curve for MVI can be steep for newer developers.
 - **Solution:** Strong architecture documentation, conducting training workshops, and starting with simpler MVI implementations before moving to more complex frameworks.

Best Practices for MVI:

- Keep Intents and States Small: Avoid defining overly large Intents and States that encompass multiple responsibilities.
- Easy Testability of Reducers and Intents: Due to the pure nature of Reducers, testing them is straightforward.



Esmaeel Moradi

@esmaeel-moradi700

Architectural Coexistence: Combining MVVM/MVI with Clean Architecture



A crucial point in understanding Android architectures is that Clean Architecture, MVVM, and MVI focus on different layers of the application and can be combined. Clean Architecture addresses the overall application structure (with Domain, Data, Presentation layers), while MVVM and MVI primarily focus on the Presentation Layer.

How to Combine Clean + MVVM/MVI?

This combination brings the benefits of both approaches:

- **Clean Architecture Layers:**

1. **Domain Layer:** Contains the core business logic and Use Cases or Interactors. **This layer is Framework Independent.**
2. **Data Layer:** Responsible for managing data sources (network APIs, local database). **This layer provides interfaces for the Domain Layer.**
3. **Presentation Layer:** Responsible for managing UI components (Activities, Fragments, Composables). **This is where MVVM or MVI come into play.**



Esmaeel Moradi

@esmaeel-moradi700



- **Role of ViewModel in this Combination:**

The ViewModel (whether in MVVM or MVI) resides in the Presentation Layer. This ViewModel interacts with Use Cases (from the Domain Layer) and Repository (from the Data Layer) to prepare data for display and process user Intents.

- **Benefits of the Combination:**

- **Framework Independence** and **High Testability:**

You benefit from the advantages of Clean Architecture.

- **Optimized UI Logic and State Management:**

You leverage the power of MVVM for simplifying UI Logic and MVI for State management.

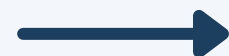
- **Scalability and Modularity:**

As I explained in my ["7 Proven Practices for Scalable Android Architecture"](#) article, this combination helps in modularizing code (dividing the project into Gradle modules based on feature or layer), which is essential for Enterprise projects.

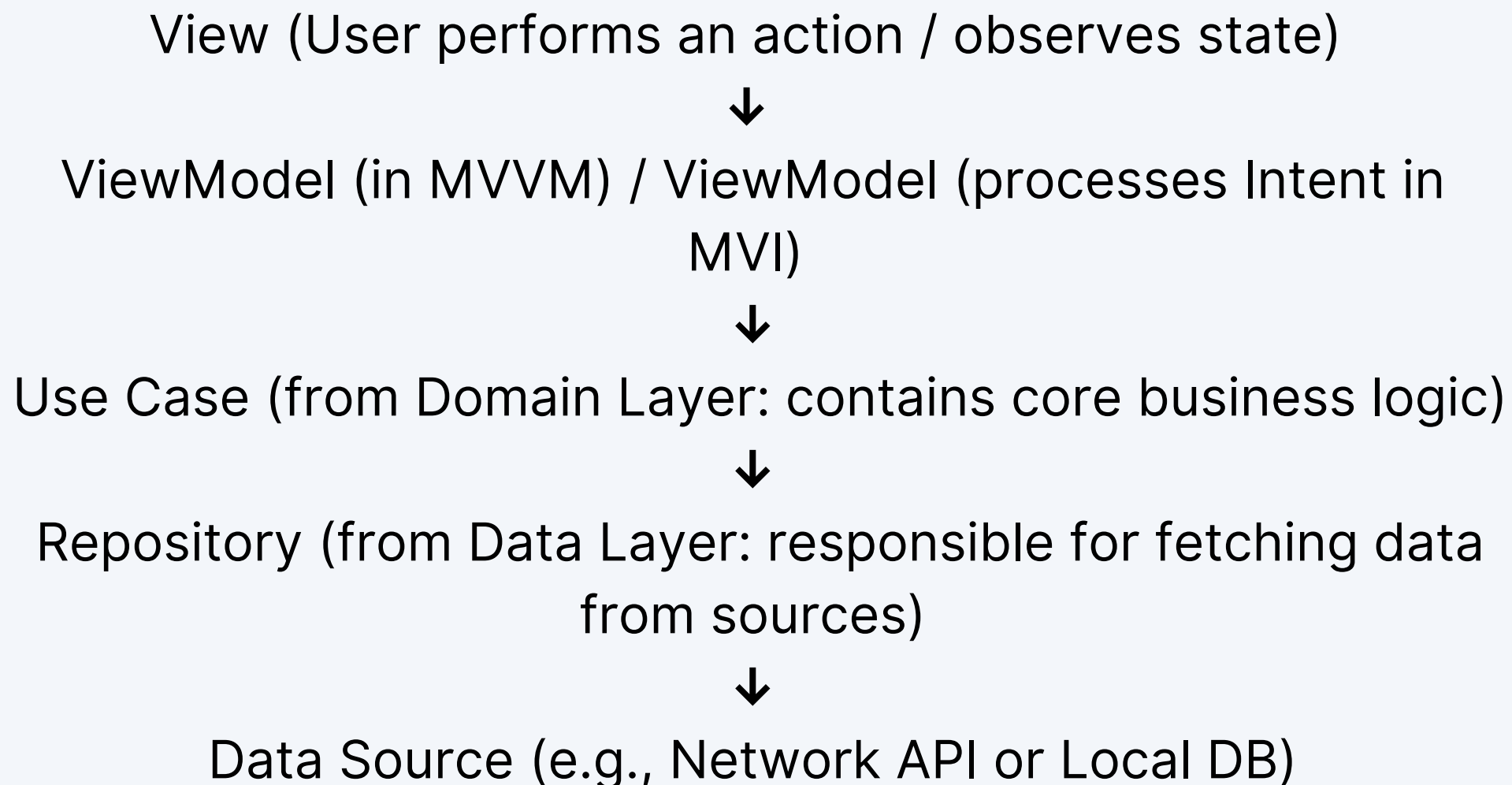


Esmaeel Moradi

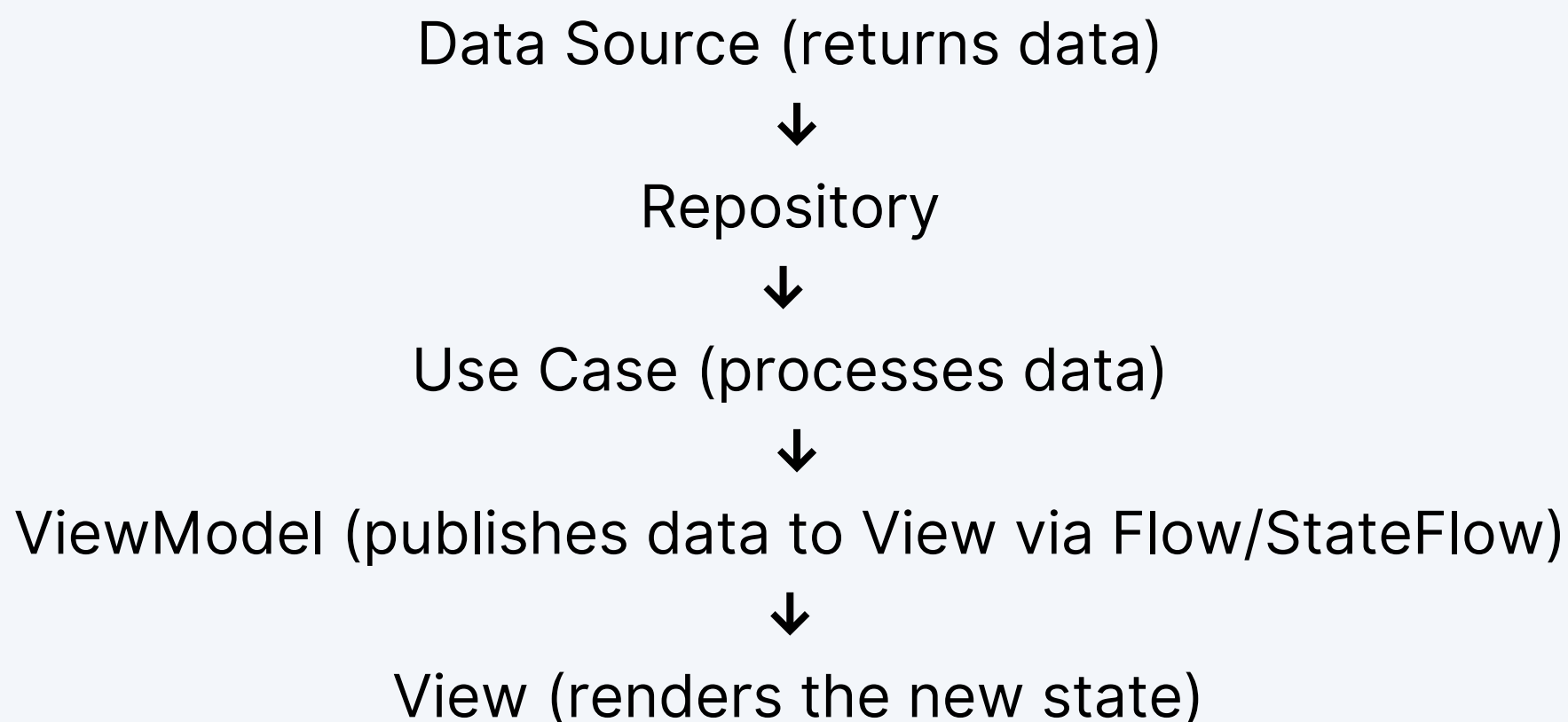
@esmaeel-moradi700



Structural Example of Data Flow in Clean + MVVM/MVI:



(And in the return direction):



Esmaeel Moradi
@esmaeel-moradi700

Intelligent Decision-Making: Which Pattern for Your Project?



There is no "one-size-fits-all" architecture. The best pattern depends on specific project needs, team size, and their experience. As discussed in my [architecture comparison article](#) and the [Single Activity Architecture](#) article, decisions must be made based on requirements.

- **Key Factors in Selection:**

- **Project Size and Complexity:**

1. **Small/Medium Projects:** MVVM might suffice due to its simplicity and reduced boilerplate.
2. **Large/Enterprise Projects:** Combining Clean Architecture with MVVM or MVI is essential.

- **Team Size and Experience:**

1. **Junior Teams:** MVVM, with its gentler learning curve, is a good starting point.
2. **Teams Experienced in Reactive Programming:** MVI can be a powerful option due to its high transparency.

- **Application Type:**

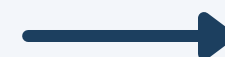
1. **UI-centric with Dynamic UIs:** MVVM is highly suitable.
2. **State-heavy with Complex State Logic (e.g., forms or real-time dashboards):** MVI, with its high predictability, excels.

- **Specific Needs:** If Debugging Ease and Predictability are your top priorities, MVI is a strong choice.



Esmaeel Moradi

@esmaeel-moradi700



Comparative Review from "Android Architectures" article:

Feature	Clean	MVVM	MVI
Focus	Full app structure	UI logic separation	State management
Complexity	High	Medium	Medium/High
Best For	Large-scale projects	Dynamic UIs	Complex state flows
Reactive Support	Optional	Built-in (Jetpack)	Built-in
Learning Curve	Steep	Moderate	Steep



Esmaeel Moradi
@esmaeel-moradi700

Conclusion:

The Power of Architecture for Android's Future

Choosing and implementing the right architecture is a vital investment in your project's long-term success. MVVM and MVI, especially when combined with Clean Architecture, are powerful tools for building next-generation Android applications that not only meet your current needs but are also prepared for future growth and changes.

A deeper understanding of these patterns helps you avoid Anti-Patterns and unnecessary Boilerplate, enabling you to write code that is maintainable, testable, and scalable. Are you ready to take your Android projects to the next level and become a true architect? This article is your roadmap!

Remember, as the [Android Developers documentation](#) states: **"Architectures are tools, not goals. Choose the one that best suits your project's needs, team expertise, and scalability requirements."**



Esmaeel Moradi

@esmaeel-moradi700

Looking forward to your
insights and experiences!
Let's elevate Android
development together.



Esmaeeil Moradi

@esmaeeil-moradi700