

Member-only story

SwiftUI in 2025: Forget MVVM

Let me tell you why



Thomas Ricouard

Follow

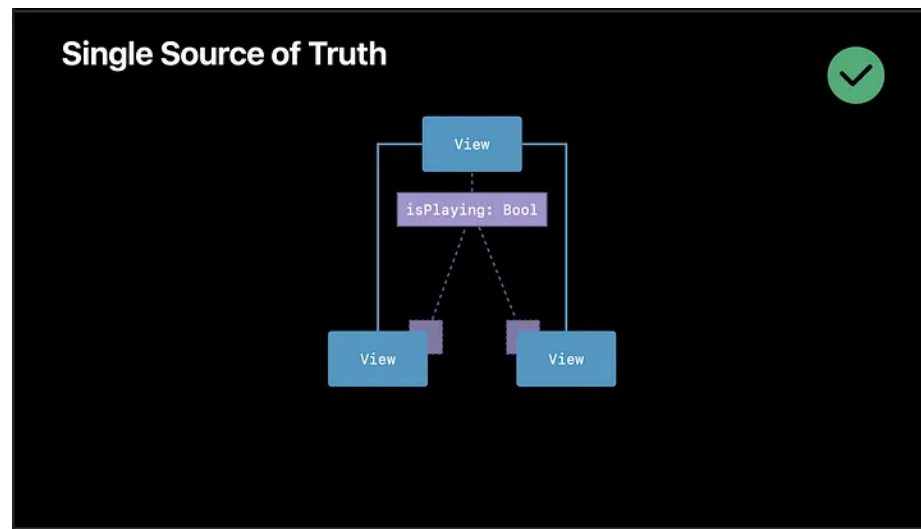
10 min read · Jun 2, 2025



979



29



Apple WWDC19, SwiftUI launch: [Data Flow Through SwiftUI](#)

It's 2025, and I'm still getting asked the same question: "Where are your ViewModels?" Every time I share this opinion or code from my open-source projects like my BlueSky client, [IcySky](#), or even the Medium iOS app, developers are surprised to see clean, simple views without a single ViewModel in sight.

Let me be clear: you don't need ViewModels in SwiftUI.

You never did.

You never will.

The MVVM Trap

When SwiftUI launched in 2019, many developers brought their UIKit baggage with them. We were so used to the Massive View Controller (MVC) problem that we immediately reached for MVVM as our savior. But here's the

thing, SwiftUI isn't UIKit. It was designed from the ground up with a different philosophy.

And this philosophy is highlighted in various WWDC videos from Apple:

Data Flow Through SwiftUI - WWDC19 - Videos - Apple Developer

SwiftUI was built from the ground up to let you write beautiful and correct user interfaces free of inconsistencies...

developer.apple.com

Discover Observation in SwiftUI - WWDC23 - Videos - Apple Developer

Simplify your SwiftUI data models with Observation. We'll share how the Observable macro can help you simplify models...

developer.apple.com

Data Essentials in SwiftUI - WWDC20 - Videos - Apple Developer

Data is a complex part of any app, but SwiftUI makes it easy to ensure a smooth, data-driven experience from...

developer.apple.com

Those are all very good sessions, and they barely mention any ViewModel.

Why? Because it's simply almost alien to the built with the data flow of SwiftUI views.

Actually, it's not even my first story on this very topic, I have another contentious article below 🙄

Removing the M from MVVM with SwiftUI

I get this question so often that I finally want to write about it. It won't be a long post on iOS app architecture...

blog.stackademic.com

SwiftUI views are structs, not classes. They're designed to be lightweight, disposable, and recreated frequently. When adding a ViewModel, you fight against this fundamental design principle.

Views as Pure State Expressions

In my latest IcySky app, every view follows the same pattern I've advocated

for years. Let me show you what I mean with a real example:

```
struct FeedView: View {
    @Environment(BlueSkyClient.self) private var client
    @Environment(AppTheme.self) private var theme

    enum ViewState {
        case loading
        case error(String)
        case loaded([Post])
    }

    @State private var viewState: ViewState = .loading
    @State private var isRefreshing = false

    var body: some View {
        NavigationStack {
            List {
                switch viewState {
                    case .loading:
                        ProgressView("Loading feed...")
                            .frame(maxWidth: .infinity)
                            .listRowSeparator(.hidden)

                    case .error(let message):
                        ErrorStateView(
                            message: message,
                            retryAction: { await loadFeed() }
                        )
                            .listRowSeparator(.hidden)

                    case .loaded(let posts):
                        ForEach(posts) { post in
                            PostRowView(post: post)
                                .listRowInsets(.init())
                        }
                }
            }
            .listStyle(.plain)
            .refreshable { await refreshFeed() }
            .task { await loadFeed() }
        }
    }
}
```

Notice what's happening here. The view state is defined right within the view using an enum. No external ViewModel needed. The view is literally just a representation of its state — nothing more, nothing less.

The Magic of Environment

This is where SwiftUI shines. Instead of manually injecting dependencies through ViewModels, I use Environment:

```
@Environment(BlueSkyClient.self) private var client

private func loadFeed() async {
    do {
        let posts = try await client.getFeed()
        viewState = .loaded(posts)
    } catch {
```

```
        viewState = .error(error.localizedDescription)
    }

    private func refreshFeed() async {
        defer { isRefreshing = false }
        isRefreshing = true
        await loadFeed()
    }
```

My `BlueSkyClient` is injected through the Environment, tested independently, and handles all the networking complexity. The view just orchestrates the user interface flow.

Real-World Complexity

“But Thomas,” you might say, “this only works for simple apps!”

Oh my god, I get that all the time.

Not true. IcySky handles authentication, complex feed algorithms, user interactions, navigation, and various degrees of complexity without view models. Sure, it’s not finished yet, but the project is starting to be sizable. Ice Cubes has been downloaded hundreds of thousands of times, and while it’s still on my “old” architecture, I’ve been slowly moving it away from ViewModel. It has more views without them than with them.

Millions use the medium iOS app, mostly SwiftUI nowadays, with very few view models (those are legacy SwiftUI we did in 2019 before my company was acquired and joined Medium). Whenever we build a new feature, we inject our services into the SwiftUI environment and build lightweight views around them. Those views hold various local states related to user interaction.

Don't hesitate to abuse `.tasks(id)` and `.onChange()` modifiers, those are great and act as a small state reducer to trigger side effects when a value is updated.

For example, here is a `headerView` from `IcySky`:

```
@Environment(BSkyClient.self) var client
@Environment(CurrentUser.self) var currentUser

private var headerView: some View {
    FeedsListTitleView(
        filter: $filter,
        searchText: $searchText,
        isInSearch: $isInSearch,
        isSearchFocused: $isSearchFocused
    )
    .task(id: searchText) {
        guard !searchText.isEmpty else { return }
        await searchFeed(query: searchText)
    }
    .onChange(of: isInSearch, initial: false) {
        guard !isInSearch else { return }
        Task { await fetchSuggestedFeed() }
    }
    .onChange(of: currentUser.savedFeeds.count) {
        switch filter {
        case .suggested:
            feeds = feeds.filter { feed in
                !currentUser.savedFeeds.contains { $0.value == feed.uri }
            }
        case .myFeeds:
            Task { await fetchMyFeeds() }
        }
    }
    .listRowSeparator(.hidden)
}
```

It's small and readable. It composes a title view, passing various states of the current view as binding to `FeedsListTitleView`.

`.task(id:)` is used to trigger a new search when the search query is updated from user input. Then we have two `.onChange(of:)` one to reset to a search placeholder when the search query is empty, and one to load and set user feeds when an environment value property is updated.

And this basically allows for a straightforward app setup, if we take the whole `IcySky` setup and simplify it a bit:

```
@main
struct IcySkyApp: App {
    @Environment(\.scenePhase) var scenePhase

    @State var client: BSkyClient?
    @State var auth: Auth = .init()
    @State var currentUser: CurrentUser?
    @State var router: AppRouter = .init(initialTab: .feed)
```

```

@State var isLoadingInitialSession: Bool = true
@State var postDataControllerProvider: PostContextProvider = .init()

var body: some Scene {
    WindowGroup {
        TabView(selection: $router.selectedTab) {
            if client != nil && currentUser != nil {
                ForEach(AppTab.allCases) { tab in
                    AppTabRootView(tab: tab)
                        .tag(tab)
                        .toolbarVisibility(.hidden, for: .tabBar)
                }
            } else {
                ProgressView()
                    .containerRelativeFrame([.horizontal, .vertical])
            }
        }
        .environment(client)
        .environment(currentUser)
        .environment(auth)
        .environment(router)
        .environment(postDataControllerProvider)
        .modelContainer(for: RecentFeedItem.self)
        .sheet(
            item: $router.presentedSheet,
            content: { presentedSheet in
                switch presentedSheet {
                    case .auth:
                        AuthView()
                            .environment(auth)
                    case let .fullScreenMedia(images, preloadedImage, namespace):
                        FullScreenMediaView(
                            images: images,
                            preloadedImage: preloadedImage,
                            namespace: namespace
                        )
                }
            }
        )
        .task(id: auth.sessionLastRefreshed) {
            if let newConfiguration = auth.configuration {
                await refreshEnvWith(configuration: newConfiguration)
                if router.presentedSheet == .auth {
                    router.presentedSheet = nil
                }
            } else if auth.configuration == nil && !isLoadingInitialSession {
                router.presentedSheet = .auth
            }
            isLoadingInitialSession = false
        }
        .task(id: scenePhase) {
            if scenePhase == .active {
                await auth.refresh()
            }
        }
    }
}

```

All of the environments are initialized at app launch and injected in the view hierarchy. So they can be retrieved by any view. If you look at the `task(id:)` modifiers, you'll see that I'm handling `Auth` there. The app authentication state is derived from the `Auth` environment and expressed at the app level to have a side effect in the router, which will then present or dismiss the login screen.

The key is proper separation of concerns:

- **Models:** Your data structures and business logic

- **Services:** Network clients, databases, utilities (injected via Environment)
- **Views:** Pure state representations that orchestrate user interactions

SwiftData: The Perfect Example

If you need more evidence that Apple designed SwiftUI to work without ViewModels, look no further than SwiftData. Apple's own data persistence framework is the perfect case study for why fighting the framework's design leads to unnecessary complexity.

SwiftData was built from the ground up to work directly within SwiftUI views using property wrappers like `@Query`, `@Model`, and the `@Environment(\.modelContext)`. Here's how natural it feels when you embrace the framework:

```
struct BookListView: View {
    @Query private var books: [Book]
    @Environment(\.modelContext) private var modelContext
    @State private var showingAddBook = false

    var body: some View {
        NavigationStack {
            List {
                ForEach(books) { book in
                    BookRowView(book: book)
                        .swipeActions {
                            Button("Delete", role: .destructive) {
                                deleteBook(book)
                            }
                        }
                }
            }
            .navigationTitle("Books")
            .toolbar {
                Button("Add Book") {
                    showingAddBook = true
                }
            }
            .sheet(isPresented: $showingAddBook) {
                AddBookView()
            }
        }
    }

    private func deleteBook(_ book: Book) {
        modelContext.delete(book)
        try? modelContext.save()
    }
}
```

Clean, simple, and direct. The `@Query` property wrapper automatically handles data fetching, change observation, and UI updates. The model context is available right where you need it.

Now imagine trying to force this through a ViewModel:

```
// DON'T DO THIS - Fighting the framework
@Observable
class BookListViewModel {
    private var modelContext: ModelContext
    var books: [Book] = []

    init(modelContext: ModelContext) {
        self.modelContext = modelContext
        fetchBooks() // Manual data fetching
    }

    private func fetchBooks() {
        // Manual descriptor creation
        let descriptor = FetchDescriptor<Book>()
        do {
            books = try modelContext.fetch(descriptor)
        } catch {
            // Error handling
        }
    }

    func deleteBook(_ book: Book) {
        modelContext.delete(book)
        try? modelContext.save()
        fetchBooks() // Manual refresh - yuck!
    }

    func refreshBooks() {
        fetchBooks()
    }
}

struct BookListView: View {
    @State private var viewModel: BookListViewModel

    init(modelContext: ModelContext) {
        _viewModel = State(initialValue: BookListViewModel(modelContext: modelContext))
    }

    var body: some View {
        // Same UI code, but now we need manual refresh calls
        // and lose automatic change observation
    }
}
```

Look at all that boilerplate! You're manually fetching data, manually refreshing after changes, and losing SwiftData's automatic change observation. You've turned a few lines of declarative code into a complex imperative mess.

The `@Query` property wrapper isn't just convenient - it's smart. It automatically:

- Fetches your data
- Observes changes in the underlying store
- Updates the UI when data changes
- Handles sorting and filtering efficiently
- Manages memory and performance optimizations

When you move this logic into a ViewModel, you lose all these benefits and have to implement them manually. You're literally fighting against the framework's design.

Even more complex scenarios work beautifully with SwiftData's view-centric approach:

```
struct AuthorBooksView: View {
    let author: Author

    @Query private var books: [Book]
    @Environment(\.modelContext) private var modelContext

    init(author: Author) {
        self.author = author
        // Dynamic queries work seamlessly
        let predicate = #Predicate<Book> { book in
            book.author?.id == author.id
        }
        _books = Query(filter: predicate, sort: \.title)
    }

    var body: some View {
        List(books) { book in
            BookRowView(book: book)
        }
        .navigationTitle(author.name)
    }
}
```

This is the beauty of SwiftUI and SwiftData working together as designed. No ViewModels, no manual data management, just clean, declarative code that's easy to understand and maintain.

Apple didn't create SwiftData to work with ViewModels, they created it to work directly with SwiftUI views because that's the most natural and efficient pattern for the framework.

The Testing Reality

"How do you test this?" is the next question I always get.

Here's the truth: testing SwiftUI views provides minimal value. Your views should be so simple that bugs are immediately visible. What you should test are your building blocks — your network clients, your data models, your business logic.

If you want to test your views, do UI automation and E2E tests.

With my approach:

- Test your `BlueSkyClient` thoroughly with unit tests
- Test your data models and transformations
- Use SwiftUI previews for visual regression testing
- Let your views be simple state expressions

Sometimes a view grows beyond what's comfortable in a single file. That's your signal to split it, not to add a `ViewModel`. In `IcySky`, I have compound views like:

```
struct PostDetailView: View {
    let post: Post
    @State private var isExpanded: Bool = false

    var body: some View {
        ScrollView {
            LazyVStack(spacing: 0) {
                PostHeaderView(post: post)
                PostContentView(post: post)
                PostActionsView(post: post, isExpanded: $isExpanded)
                PostRepliesView(postId: post.id)
            }
        }
    }
}
```

Q 1

Each subview handles its own state and interactions. No `ViewModel` coordination needed. Use `State` and `Binding` to manage the data flow between your views.

But if you want to test your view with introspection, I've been using a great library for a while, `ViewInspector`:

GitHub - nalexn/ViewInspector: Runtime introspection and unit testing of SwiftUI views

Runtime introspection and unit testing of SwiftUI views - nalexn/ViewInspector

github.com

If you want to test your view representation other than with a snapshot test, you can do so using `ViewInspector`. For example, here is how we test our notifications Text renderer in the Medium iOS app.

```
@MainActor
final class ActivityTextBuilderTestsTests: XCTestCase {
    func test_follow_activity() async {
        let activity = GraphQL.Activity(notificationType: "users_following_you",
```

```

        notificationName: UUID().uuidString,
        isUnread: false,
        occurredAt: 0,
        actor: .init(id: "test", name: "Test user"))
    let build = ActivityTextBuilder(activity: activity)
    let text = try! build.buildActorText().inspect().text().string()
    XCTAssertEqual(text, "Test user started following you")
}

func test_catalog_recommend() async {
    let activity = GraphQL.Activity(notificationType: "catalog_recommended",
        notificationName: UUID().uuidString,
        isUnread: false,
        occurredAt: 0,
        actor: .init(id: "test", name: "Test user"),
        catalog: .init(id: "test", name: "Test catalog"))
    let build = ActivityTextBuilder(activity: activity)
    let actorText = try! build.buildActorText().inspect().text().string()
    let contentText = try! build.buildContentText().inspect().text().string()
    XCTAssertEqual(actorText, "Test user clapped for")
    XCTAssertEqual(contentText, " Test catalog")
}
}

```

This is useful to confirm that building your view with one input gives you the expected and consistent output.

The 2025 Reality

As we head deeper into 2025, SwiftUI has matured. We have `@Observable`, improved Environment handling, and better async support, task lifecycle, etc... The tools are there to build complex apps without the overhead of ViewModels.

Almost all the view lifecycle essentials are at the view level, and it's tough to build around the view because it means forwarding many things outside of the view itself.

I'll reconsider ViewModel in SwiftUI when/if Apple provides a way to access the environment outside the view. I know you can do it with a library like TCA, but to me, the only canon is vanilla SwiftUI.

So, if you could do something like the code below, where with a macro, you could associate a class with a view and access its injected environment object and values, it would open many possibilities for even cleaner code.

```

@Observable
@ObservableWithEnvironment<UserListView>
class UserListDataSource {
    @Environment(\.databaseService) var database

    var users: [String] = []

    func fetch() async - {
        self.users = await database.fetch()
    }
}

```

```
    }  
  }  
  
  struct UserListView: View {  
    @State private var dataSource = UserListDataSource()  
  
    var body: some View {  
      List(dataSource.users, id: \.self) { user in  
        Text(user)  
      }  
      .task {  
        await dataSource.fetch()  
      }  
    }  
  }  
}
```

Why This Matters

Every ViewModel you add is:

- More complexity to maintain
- More objects to keep in sync
- More indirection between intent and action
- More cognitive overhead for your team

ViewModels are also an easy way to bloat. You'll have a great time if you force yourself to split your views into the smallest possible unit. ViewModel, like Controller in MVC, encourages you to take all your logic code and shove it outside so it can look “clean”

SwiftUI gives you powerful primitives: `@State`, `@Environment`, `@Observable`, `Bindable/Binding`. Use them. Trust the framework. Your future self will thank you.

The Bottom Line

In 2025, there's no excuse for cluttering your SwiftUI apps with unnecessary ViewModels. Embrace the framework's design. Let your views be simple, pure expressions of state. Focus your testing and complexity on the parts that matter — your business logic and services.

Your SwiftUI apps will be cleaner, more maintainable, and more enjoyable to work on.

Goodbye MVVM 🙋

Long live The View 🙌

Happy coding 🚀

Swift

Swiftui

Mvvm

Programming

Design Patterns



Written by Thomas Ricouard

6.6K followers · 984 following

Follow

📱 🚀 🇫🇷 [Entrepreneur, iOS/Mac & Web dev] | Now @Medium, @Glose 📖 |
Past @google 🔍 | Co-founded few companies before, a movies 🎬 app and
smart browser one.

Responses (29)



Till Gartner

What are your thoughts?



Angel Rodriguez

Jun 4



This makes sense and can work if the views just handle their own view states. But wouldn't you say the "model" object you describe is technically your VM anyway?

For me the VM's main purpose is to abstract away the UI/business logic so I can unit... [more](#)



41



1 reply

[Reply](#)



Muhammed YILDIRIM

Jun 2



Thanks! You raised a big question mark in my mind.

I've been learning SwiftUI and publishing apps for the last two years. While I was working on my 5th app for the App Store, I was really struggling with code readability and writing clean ViewModels.

I... [more](#)



42



[Reply](#)



Jan Rabe

6 days ago



So ... you're just putting everything into global variables? Sounds like lots of debugging sessions over the course of longer project duration to find that side effect that's causing issues or testing more than necessary when changing anything.

It's cheap to build but hard to maintain imho.



28



[Reply](#)

[See all responses](#)

More from the list: "Swift"

Curated by Till Gartner



Thomas Ricouard

**Building iOS apps with
Cursor and Claude Code**



· Jun 4



In Stacka... by Priyanka ...

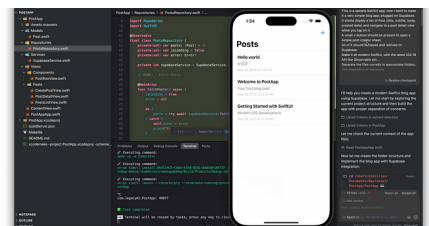
**SwiftUI Interview
Questions and Answers**



· Dec 18, 2024

[View list](#)

More from Thomas Ricouard



Thomas Ricouard

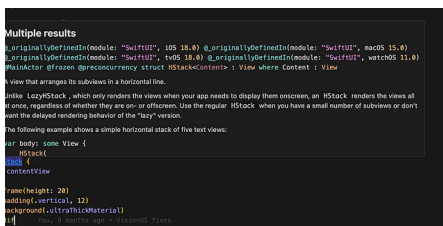
Vibe Coding iOS apps with Claude 4

It's much better at SwiftUI & iOS than previous models

May 23

364

4



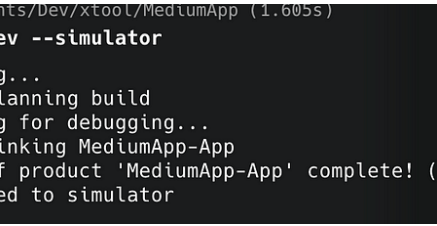
Thomas Ricouard

How to use Cursor for iOS development

Oct 22, 2024

1.7K

22



Thomas Ricouard

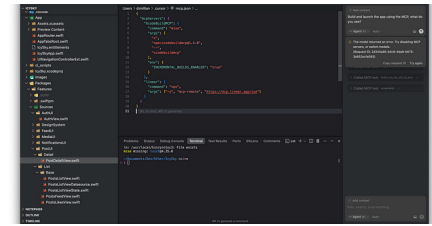
Build an iOS app faster than ever with xttool

Did you know that Xcode is not necessary anymore?

May 23

210

5



In Level Up Coding by Thomas Ricouard

Agentic iOS workflow with XcodeBuildMCP and Cursor

The Cursor Agent can now build, run, and see your app!

May 24

202

2

See all from Thomas Ricouard

Recommended from Medium

https://medium.com/@dimillian/swiftui-in-2025-forget-mvvm-262ff2bbd2ed

Page 15 of 17



Koti Avula

weak let vs weak var in Swift 6.2: Why Apple added?

Important: weak let means that a property cannot be changed after creation, but it can...

★ Jun 1 🖱️ 83 💬 2 📌 ⋮



JS In JavaScript in Plain English by GeekSociety

I Stopped Building Frontends. Now I Use MCP Servers to Let AI Run ...

It's 2025, and the way we build applications has fundamentally changed.

★ Jun 2 🖱️ 4K 💬 159 📌 ⋮



LONG In Long. Sweet. Valuable. by Ossai Chinedum

I'll Instantly Know You Used Chat Gpt If I See This

Trust me you're not as slick as you think

★ May 16 🖱️ 8.3K 💬 452 📌 ⋮



AI In Realworld AI Use Cases by Chris Dunlop

Why clients pay me 10x more than developers who are better at...

Last week I charged \$15,000 for work a better coder would do for \$1,500 and I think you...

★ Jun 2 🖱️ 3.6K 💬 57 📌 ⋮

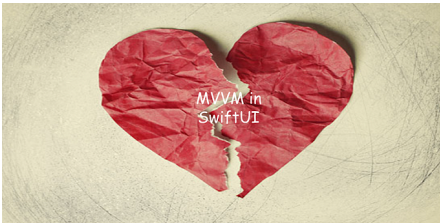


Tanishq Arora

5 Xcode Extensions That Will Save You Hours

Let's face it—Xcode is powerful, but not always the most efficient tool out of the box....

5d ago 🖱️ 80 💬 4 📌 ⋮



minal kewat

Breaking Up with MVVM in SwiftUI

In the early days of SwiftUI, MVVM (Model-View-ViewModel) was the default...

6d ago 🖱️ 17 💬 2 📌 ⋮

See more recommendations

[Help](#) [Status](#) [About](#) [Careers](#) [Press](#) [Blog](#) [Privacy](#) [Rules](#) [Terms](#) [Text to speech](#)