ANDROID · KOTLIN · JETPACK COMPOSE

From XML to Jetpack Compose

Lessons Learned While Modernizing Android Apps

Shubham

Founder, Papaya Coders

Android Engineer

INTRODUCTION

Who Am I?

Building Android products as a founder, and shipping code as an engineer.

Founder, Papaya Coders

Building a product studio around fintech and consumer Android apps.

Android Engineer

Hands-on in production codebases — not just architecture diagrams.

Jetpack Compose

Led Compose migrations on real, revenue-critical apps.

Kotlin & Coroutines

Clean, idiomatic Kotlin — structured concurrency over callback soup.

Content Creator

26K+ subscribers learning Android engineering in public.

Community

Speaking and mentoring engineers navigating this exact transition.

A DECADE IN THE MAKING

Why Compose? The Long Road Here

Android UI evolved through pain — each step solved a problem and created a new one.

1

Activity

Single-screen apps, no fragments

2

Fragment

Modular screens, lifecycle chaos

3

XML Layouts

Declarative-ish, imperative glue

4

ViewBinding

Type-safe views, still verbose

5

RecyclerView

Flexible lists, adapter boilerplate

6

Jetpack Compose

Declarative, Kotlin-native UI

SHOW, NOT TELL

XML vs Compose: Side by Side

Same screen. Same result. A fraction of the code.

3× LESS CODE

```xml
<TextView
    android:id="@+id/title"
    android:text="Hello"
    android:textSize="24sp" />

// In Activity
title.text = "Hello"
recyclerView.adapter = MyAdapter()
// ...+ ViewHolder, DiffUtil,
// notifyDataSetChanged, null checks
```

XML + VIEWS · Imperative, Verbose

~40 lines total

```
@Composable
fun Greeting(name: String) {
  Text(
    text = "Hello $name",
    fontSize = 24.sp
  )
}

LazyColumn {
  items(list) { Item(it) }
}
```

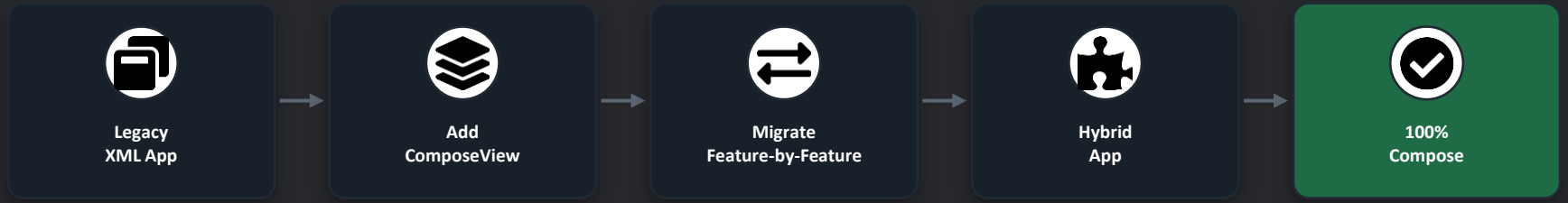
JETPACK COMPOSE · Declarative, Concise

~12 lines — same result

HOW WE ACTUALLY DID IT

Don't Rewrite Everything

Production migration is a marathon, not a sprint.



ComposeView embeds Compose inside XML layouts. **AndroidView** embeds legacy Views inside Compose. Use both liberally during transition — there's no shame in a hybrid app.

"We picked the screen with the lowest blast radius, shipped it behind a flag, and watched crash-free rate for a week before touching anything else."

THE CORE PATTERN

Thinking in State

Unidirectional data flow: UI emits events, ViewModel holds truth, Compose reflects state.

```
// ViewModel
val uiState = MutableStateFlow("")

// Composable
val state by viewModel.uiState
    .collectAsStateWithLifecycle()
```

State Hoisting: push state up to the caller. Composables become pure functions — predictable, testable, and reusable.

`remember()`

Survives recomposition.

`rememberSaveable()`

Survives configuration change.

`derivedStateOf()`

Computed only when inputs change.

WHAT WE DISCOVERED

Performance & Recomposition

Compose redraws only what changed — smart by default, but you have to help it.

`remember`

`derivedStateOf`

`Stable Data Classes`

`LazyColumn Keys`

`Skip Lambda Params`

`Layout Inspector`

What is recomposition? Compose reruns the functions whose inputs changed — nothing else. Android Studio's Layout Inspector highlights recomposing composables in real time.

Our biggest win: *adding stable keys to LazyColumn cut scroll jank by more than half.*

A NEW TEAMMATE

AI + Android Development

Use it to bootstrap screens, not to replace judgment.

ChatGPT | Claude | Gemini | Copilot

The Magic Prompt

"Convert this XML screen to Compose with Material 3, MVVM, State Hoisting, and Dark Mode."

Where It Helps

Mechanical XML → Compose conversion, boilerplate, and repetitive patterns.

Always Review

AI skips edge cases, accessibility, and architecture fit. Manual review is non-negotiable.

BATTLE-TESTED

Lessons Learned From Production

The mistakes cost us real time. The practices got adopted for a reason.

Mistakes We Made

- Huge composables — 500+ line functions are unmaintainable.
- Business logic in UI — keep composables dumb.
- Passing ViewModel everywhere — use state hoisting instead.
- Ignoring recomposition — profile early, not after launch.

Best Practices

- Small composables — single responsibility, easy to test.
- Stateless UI — push state up, events down.
- Clean Architecture — Compose is just the view layer.
- Compose Preview — design in isolation, iterate fast.

BEFORE YOU GO

Key Takeaways

01

Think in State, not Views.

02

Migrate Gradually.

03

Compose Boosts Productivity.

04

Performance Matters.

05

AI Assists, You Decide.

"Start small. Learn deeply.

Build better Android apps."

Founder, Papaya Coders

Building Android products end to end.

Android Engineer

Still hands-on in production code, every day.

Thank You!

Questions, war stories, or migration horror stories of your own — let's talk.

Founder, Papaya Coders

github.com/1902shubh

linkedin.com/in/1902shubh

Questions?