Compose Beyond the UI: Architecting Reactive State Machines at Scale

Patterns for predictable, testable, scalable UI state

Adit Lal
GDE Android

Identify

What breaks at scale

Patterns

Why it breaks? Build mental models

Solutions

How to fix it

Identify

- Common bugs that seem random

- Symptoms vs root causes

- The "it works on my machine" problem

Patterns

- State taxonomy (ephemeral → screen → domain)

- Boolean explosion → state machines

- Tangled transitions → pure functions

Solutions

- Sealed interfaces for impossible states

- TransitionResult for effects as outputs

- Testing without mocks

Three Bugs Everyone Has

Three Bugs Everyone Has

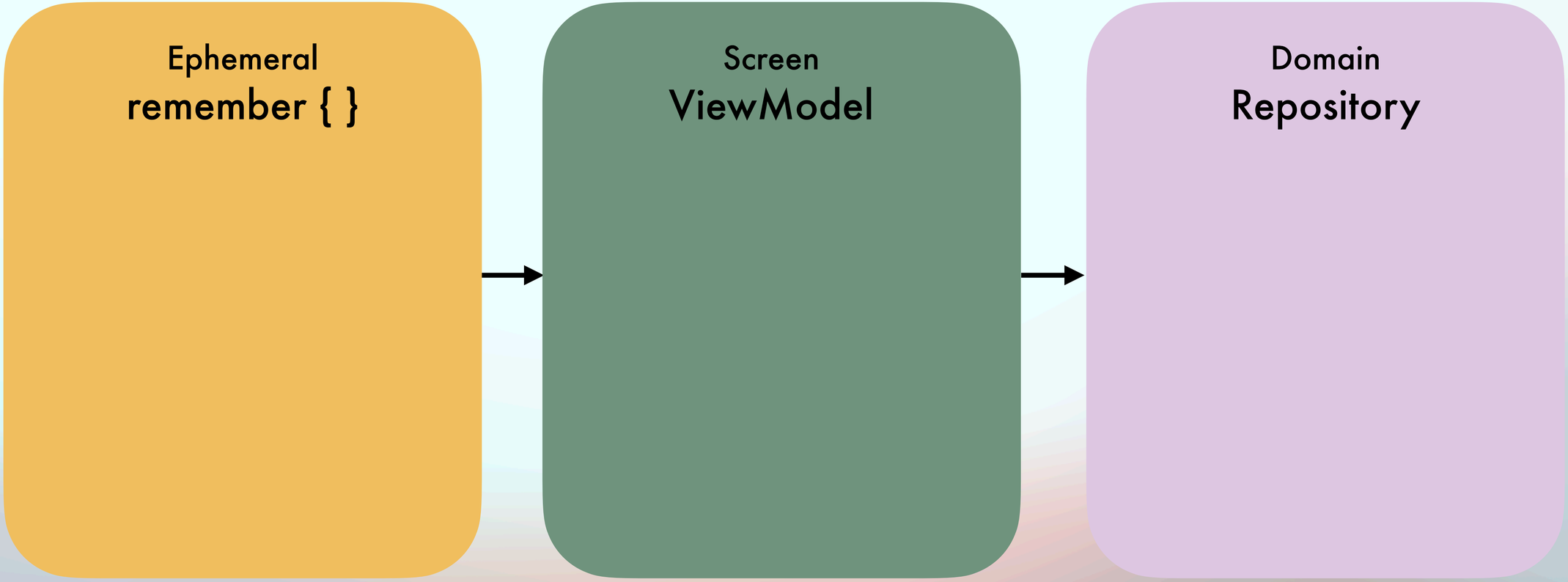
- Snackbar shows again after rotation
- Button Fires twice on fast tap
- Screen shows stale data after back navigation

These Aren't Compose Bugs

These Aren't Compose Bugs

These are State bugs

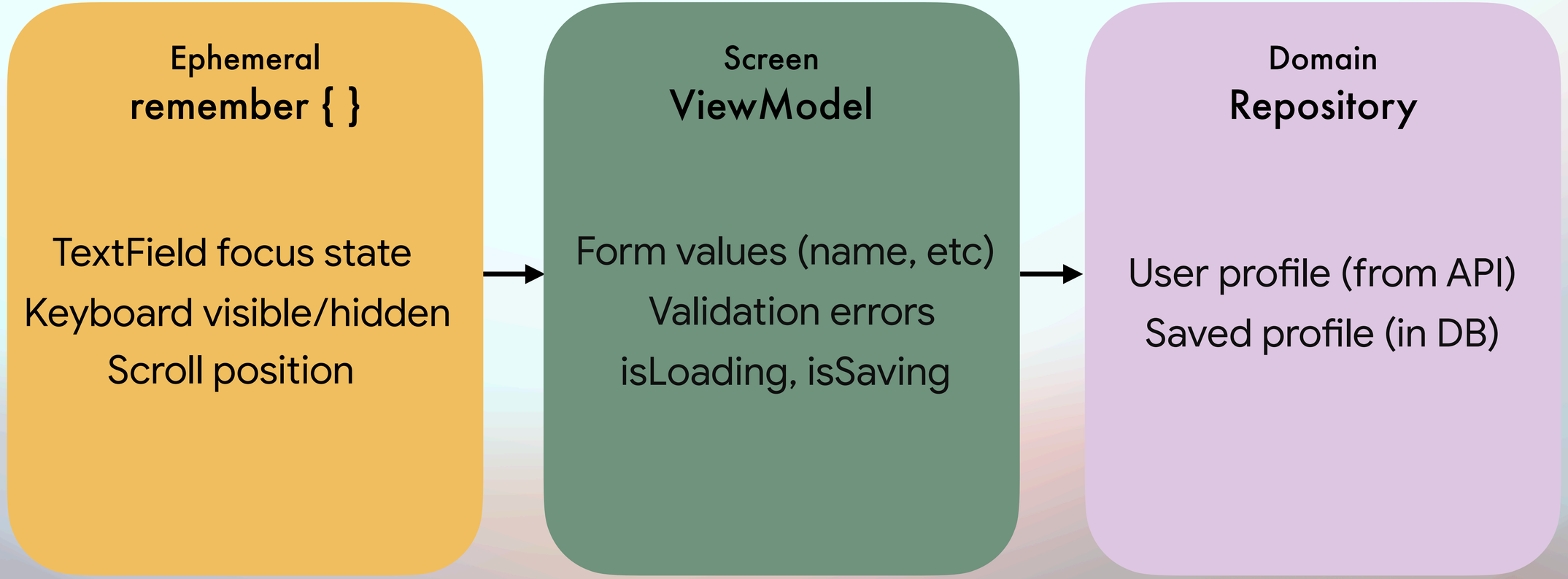
The Three Layers of State



The Three Layers of State

Form Screen

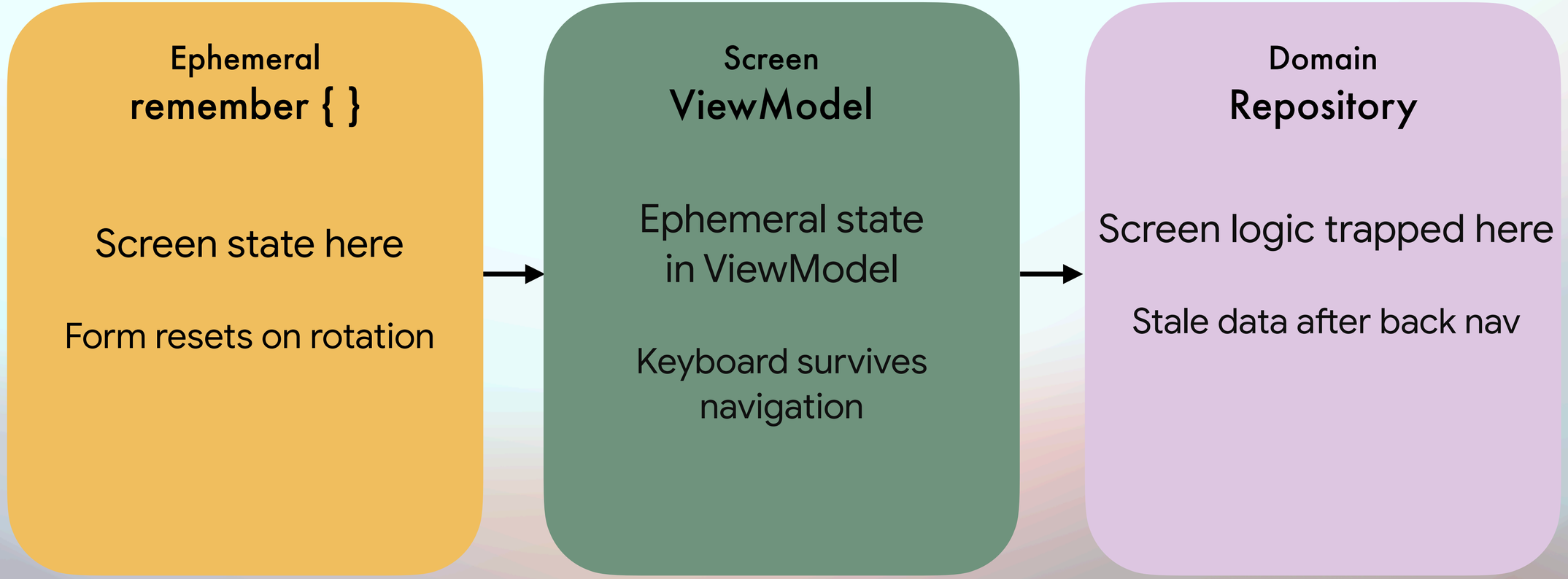
Where does each state live?



The Three Layers of State

Form Screen

Wrong Layer == Bugs



The Three Layers of State

The Rule

When Should the state die?

- Dies with recomposition → remember { }
- Dies with navigation → ViewModel
- Never dies → Repository

State Machines

The Boolean Explosion Problem

```
data class ScreenState(
        val isLoading: Boolean = false,
        val isSaving: Boolean = false,
        val isError: Boolean = false,
        val isSuccess: Boolean = false,
    )
```

2^4 = 16 possible combinations

Only 4 are valid

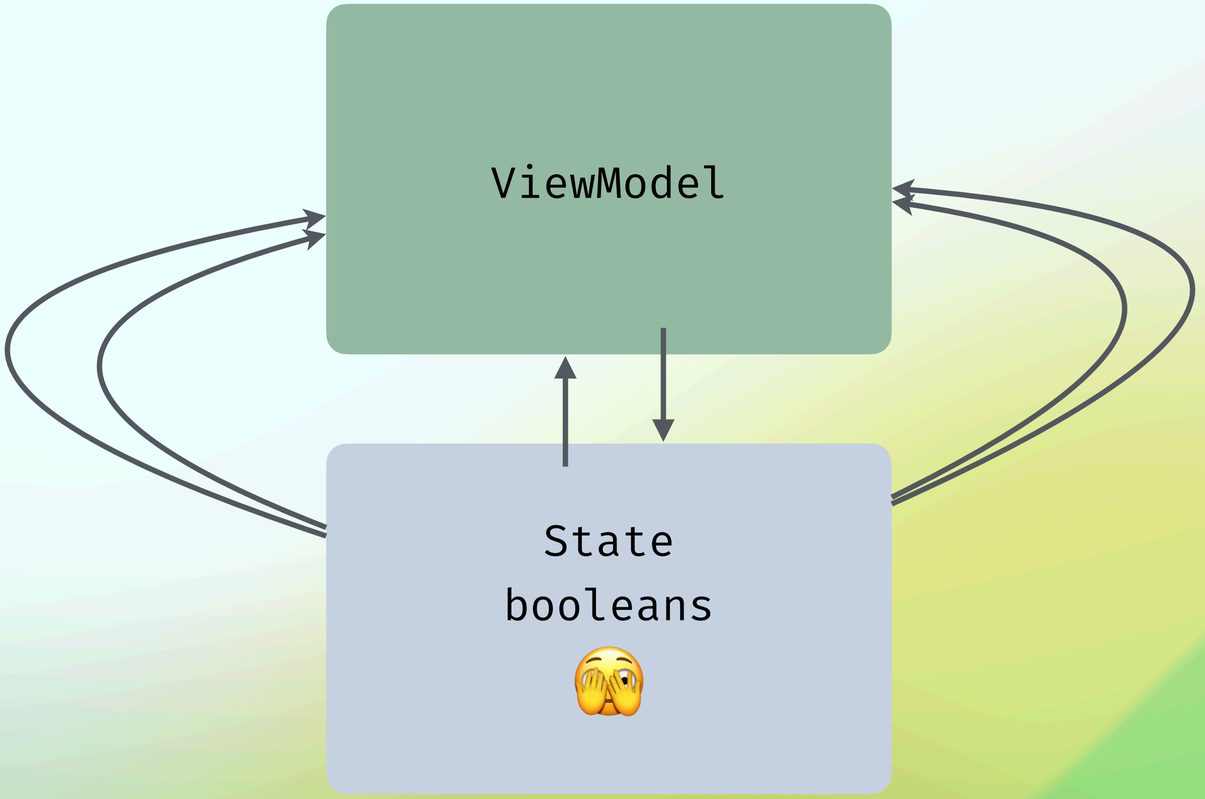
State Machines

The Boolean Explosion Problem

```
data class ScreenState(
        val isLoading: Boolean = false,
        val isSaving: Boolean = false,
        val isError: Boolean = false,
        val isSuccess: Boolean = false,
    )
```

$2^4 = 16$ possible combinations

Only 4 are valid



Every action can touch every flag

State Machines

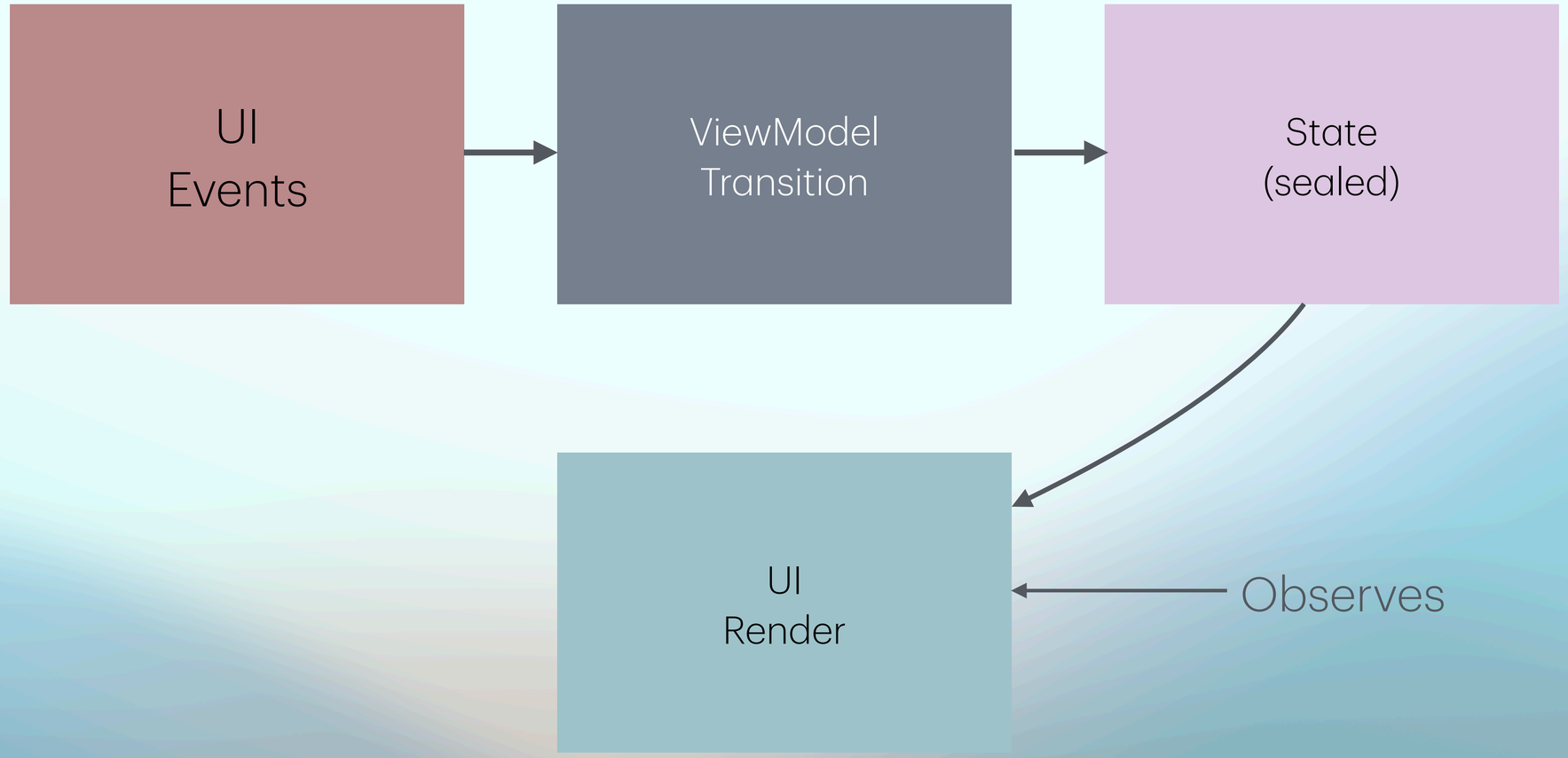
Sealed Interface Solution

Make IMPOSSIBLE STATES impossible

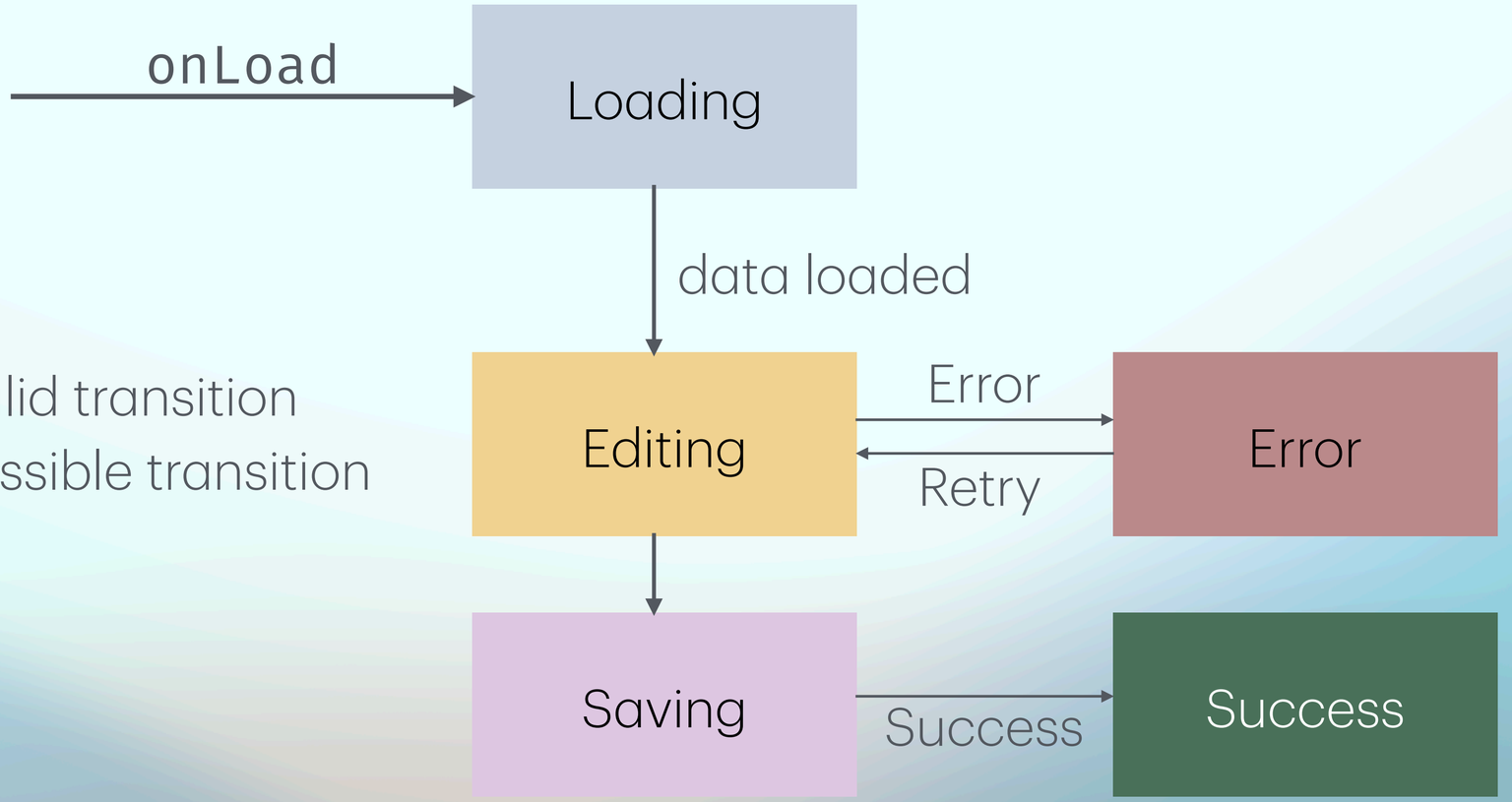
```
sealed interface ScreenState {
  data object Loading : ScreenState
  data class Editing(val form: Form) : ScreenState
  data class Saving(val form: Form) : ScreenState
  data class Success(val msg: String) : ScreenState
  data class Error(val reason: String) : ScreenState
}
```

5 states. Only 5. No combinations.

State Machines
UniDirectional Pipe

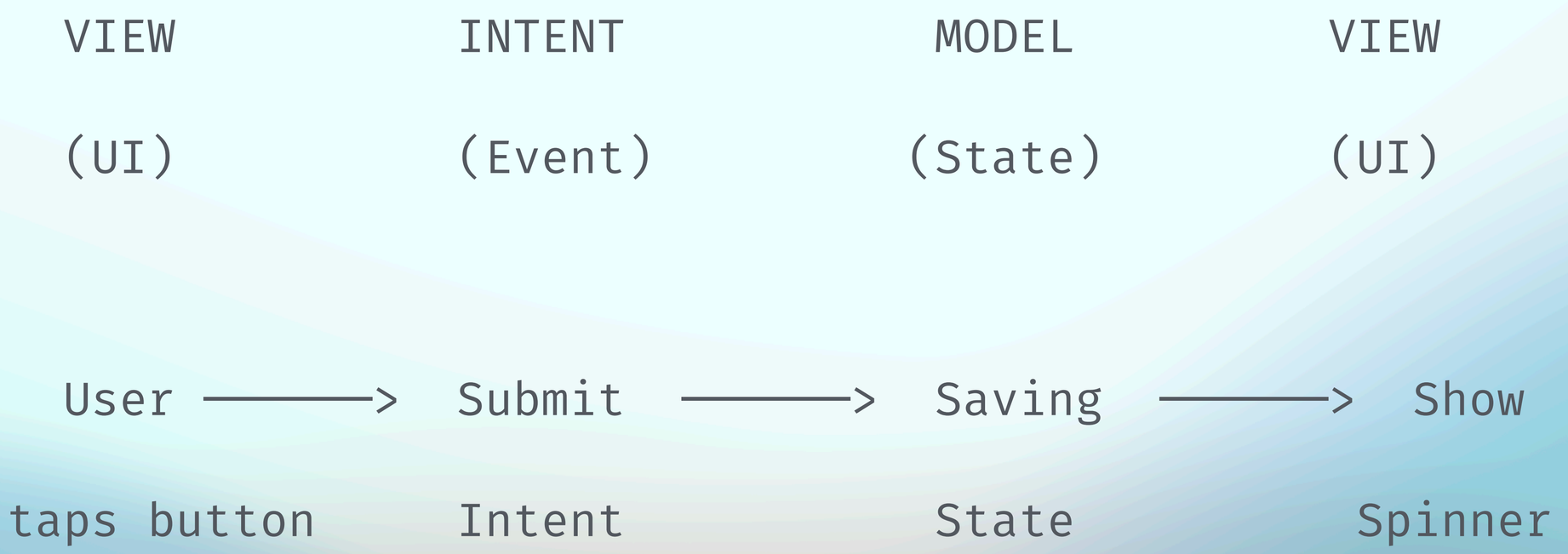


State Machines
UniDirectional Pipe



Each arrow is a valid transition
Missing arrow = impossible transition

State Machines
UniDirectional Pipe



State Anti-Patterns

State Anti-Patterns

ANTI-PATTERN #1: LaunchedEffect Self-Cancellation Trap

```
var shouldProcess by remember { mutableStateOf(false) }

LaunchedEffect(shouldProcess) {
    if (shouldProcess) {
        doSomething()
        shouldProcess = false
        delay(200)
        doImportantWork()
    }
}
```

State Anti-Patterns

ANTI-PATTERN #1: LaunchedEffect Self-Cancellation Trap

```
var shouldProcess by remember { mutableStateOf(false) }

LaunchedEffect(shouldProcess) {
    if (shouldProcess) {
        doSomething()  // ✅ Runs
        shouldProcess = false     // 💀 Triggers cancellation
        delay(200)  // ⏸️ Suspension point
        doImportantWork() // ❌ NEVER RUNS
    }
}
```

State Anti-Patterns

ANTI-PATTERN #1: LaunchedEffect Self-Cancellation Trap

Timeline

shouldProcess = true —> LanchedEffect starts

doSomething() —> completes

shouldProcess = false —> Key Changed, cancellation scheduled

delay(200) —> Coroutine suspends

Recomposition runs (~16ms)

- Key is changed
- Cancels old LaunchedEffect

doImportantWork() —> never reached

```
var shouldProcess by remember { mutableStateOf(false) }

LaunchedEffect(shouldProcess) {
    if (shouldProcess) {
        doSomething()
        shouldProcess = false
        delay(200)
        doImportantWork()
    }
}
```

State Anti-Patterns

ANTI-PATTERN #1: LaunchedEffect Self-Cancellation Trap

Fix

```
var shouldProcess by remember { mutableStateOf(false) }

LaunchedEffect(Unit) {  // Key never changes
    snapshotFlow { shouldProcess }
        .filter { it }
        .collect {
            doSomething()
            shouldProcess = false  // Safe now
            delay(200)
            doImportantWork()      // ✅ Runs
        }
}
```

Key is stable. State change doesn't kill the coroutine.

State Anti-Patterns

Rule

LAUNCHEDEFFECT RULE

If you change the key inside the effect...

State Anti-Patterns

Rule

LAUNCHEDEFFECT RULE

If you change the key inside the effect...
...and you have any suspension point after...

State Anti-Patterns

Rule

LAUNCHEDEFFECT RULE

If you change the key inside the effect...
...and you have any suspension point after...
...your code after that point won't run.

Key = lifecycle

Change key = restart lifecycle

Restart = cancel previous

Transitions Are Functions

Transitions Are Functions

```
data class TransitionResult(
      val newState: ScreenState,
      val effects: List<Effect> = emptyList()
  )
```

Transitions Are Functions

```
data class TransitionResult(
        val newState: ScreenState,
        val effects: List<Effect> = emptyList()
    )

fun ScreenState.onSubmit(): TransitionResult = when (this) {
    is Loading -> TransitionResult(this)
    is Editing -> TransitionResult(
        newState = Saving(form),
        effects = listOf(Effect.SaveProfile(form))
    )
    is Saving -> TransitionResult(this)
    is Success -> TransitionResult(this)
    is Error -> TransitionResult(
        newState = Saving(form),
        effects = listOf(Effect.SaveProfile(form))
    )
}
```

```
    val newState: ScreenState,
    val effects: List<Effect> = emptyList()
)
```

Transitions Are Functions

```
fun ScreenState.onSubmit(): TransitionResult = when (this) {
    is Loading -> TransitionResult(this)
    is Editing -> TransitionResult(
        newState = Saving(form),
        effects = listOf(Effect.SaveProfile(form))
    )
    is Saving -> TransitionResult(this)
    is Success -> TransitionResult(this)
    is Error -> TransitionResult(
        newState = Saving(form),
        effects = listOf(Effect.SaveProfile(form))
    )
}
```

```
    val newState: ScreenState,
    val effects: List<Effect> = emptyList()
)
```

Transitions Are Functions

```
fun ScreenState.onSubmit(): TransitionResult = when (this) {
    is Loading -> TransitionResult(this)
    is Editing -> TransitionResult(
        newState = Saving(form),
        effects = listOf(Effect.SaveProfile(form))
    )
    is Saving -> TransitionResult(this)
    is Success -> TransitionResult(this)
    is Error -> TransitionResult(
        newState = Saving(form),
        effects = listOf(Effect.SaveProfile(form))
    )
}
```

Transitions Are Functions

Pure function. No API calls inside. No side effects. Just data in, data out.

```
fun ScreenState.onSubmit(): TransitionResult = when (this) {
    is Loading -> TransitionResult(this)
    is Editing -> TransitionResult(
        newState = Saving(form),
        effects = listOf(Effect.SaveProfile(form))
    )
    is Saving -> TransitionResult(this)
    is Success -> TransitionResult(this)
    is Error -> TransitionResult(
        newState = Saving(form),
        effects = listOf(Effect.SaveProfile(form))
    )
}
```

Why Pure functions?

PREDICTABLE

Same input → same output. Always.

TESTABLE

No mocks. No coroutines. Just assertions.

DEBUGGABLE

Log every transition. Replay any bug.

Effects as Outputs

Effects as Outputs

Side Effect are outputs, not inline code

```
// ❌ Effect inside transition
  fun onSubmit() {
      _state.value = Saving(form)
      repository.save(form)  // Side effect here
      navigator.goTo(Success)  // Another one
  }
```

Effects as Outputs

Side Effect are outputs, not inline code

```
// ❌ Effect inside transition
 fun onSubmit() {
     _state.value = Saving(form)
     repository.save(form)  // Side effect here
     navigator.goTo(Success)  // Another one
 }
```

```
// ✅ Effects as data
 fun ScreenState.onSubmit() = TransitionResult(
     newState = Saving(form),
     effects = listOf(
         Effect.SaveProfile(form),
         Effect.Navigate(Routes.Success)
     )
 )
```

Effect Types

Effect Types

Common Effects

```
sealed interface Effect {
      data class ShowSnackbar(
          val message: String,
          val id: UUID = UUID.randomUUID()
      ) : Effect
      data class Navigate(val route: String) : Effect
      data class SaveToDatabase(val data: Form) : Effect
      data class TrackAnalytics(val event: String) : Effect
      data object HapticFeedback : Effect
  }
```

Effect Types

ViewModel Executes

```
class ProfileViewModel : ViewModel() {
    private val _state = MutableStateFlow<ScreenState>(Loading)
    val state = _state.asStateFlow()

    fun onSubmit() {
        val result = _state.value.onSubmit()
        _state.value = result.newState
        result.effects.forEach { execute(it) }
    }

    private fun execute(effect: Effect) = when (effect) {
        is Effect.SaveProfile → viewModelScope.launch {
            repository.save(effect.form)
        }
        is Effect.ShowSnackbar → _snackbar.emit(effect)
        is Effect.Navigate → _navigation.emit(effect.route)
    }
}
```

Effect Types

ViewModel Executes

```
class ProfileViewModel : ViewModel() {
    private val _state = MutableStateFlow<ScreenState>(Loading)
    val state = _state.asStateFlow()

    fun onSubmit() {
        val result = _state.value.onSubmit()
        _state.value = result.newState
        result.effects.forEach { execute(it) }
    }

    private fun execute(effect: Effect) = when (effect) {
        is Effect.SaveProfile → viewModelScope.launch {
            repository.save(effect.form)
        }
        is Effect.ShowSnackbar → _snackbar.emit(effect)
        is Effect.Navigate → _navigation.emit(effect.route)
    }
}
```

Effect Types

ViewModel Executes

```
class ProfileViewModel : ViewModel() {
    private val _state = MutableStateFlow<ScreenState>(Loading)
    val state = _state.asStateFlow()

    fun onSubmit() {
        val result = _state.value.onSubmit()
        _state.value = result.newState
        result.effects.forEach { execute(it) }
    }

    private fun execute(effect: Effect) = when (effect) {
        is Effect.SaveProfile → viewModelScope.launch {
            repository.save(effect.form)
        }
        is Effect.ShowSnackbar → _snackbar.emit(effect)
        is Effect.Navigate → _navigation.emit(effect.route)
    }
}
```

Async State Machines

Async State Machines

ASYNC = EXPLICIT STATES

```
sealed interface Async<out T> {
      data object Idle : Async<Nothing>
      data object Loading : Async<Nothing>
      data class Success<T>(val data: T) : Async<T>
      data class Error(val error: Throwable) : Async<Nothing>
}

  //Usage:
data class ProfileScreenState(
      val profile: Async<UserProfile> = Async.Idle,
      val saveStatus: Async<Unit> = Async.Idle
)
```

Async State Machines

WHY ASYNC<T>?

Cancellation

Job cancelled → back to Idle

Not stuck in Loading forever

MULTIPLE REQUESTS

Each operation gets its own Async<T>

No boolean flags colliding

RETRY

Error holds throwable.

User taps retry → Loading again

Testing

Testing

TESTING: NO MOCKS NEEDED

```
@Test
    fun `submit from Editing transitions to Saving`() {
        val state = ScreenState.Editing(form = testForm)

        val result = state.onSubmit()

        assertThat(result.newState).isEqualTo(Saving(testForm))
        assertThat(result.effects).containsExactly(
            Effect.SaveProfile(testForm)
        )
    }
```

Testing

TESTING: NO MOCKS NEEDED

```
@Test
    fun `submit while Saving is ignored`() {
        val state = ScreenState.Saving(form = testForm)

        val result = state.onSubmit()

        assertThat(result.newState).isEqualTo(Saving(testForm))
        assertThat(result.effects).isEmpty()
    }
```

Avoiding AntiPatterns

Avoiding AntiPatterns

Snackbar Shows Twice

```
// Bad: ❌ State-based one-time event
 val error by viewModel.error.collectAsState()

 LaunchedEffect(error) {
     error?.let {
         snackbarHostState.showSnackbar(it)
         // Rotation → resubscribes → shows again
     }
 }
```

Avoiding AntiPatterns

Snackbar Shows Twice

```
// Good: ✅ Effect with unique ID
data class ShowSnackbar(
    val message: String,
    val id: UUID = UUID.randomUUID()
)

LaunchedEffect(effect?.id) {
    effect?.let {
        snackbarHostState.showSnackbar(it.message)
        viewModel.consumeEffect()
    }
}
```

Avoiding AntiPatterns

Forgotten Remember

```
//Bad: ❌ New instance every recomposition
@Composable
fun Counter() {
    var count = mutableStateOf(0)  // No remember!
    Button(onClick = { count.value++ }) {
        Text("Count: ${count.value}")  // Always 0
    }
}
```

Avoiding AntiPatterns

Forgotten Remember

```
// Good:✅ Survives recomposition
@Composable
fun Counter() {
    var count by remember { mutableStateOf(0) }
    Button(onClick = { count++ }) {
        Text("Count: $count")
    }
}
```

Avoiding AntiPatterns

Backwards Write = Infinite Loop

```
// Bad:❌ Write after read = infinite recomposition
@Composable
fun Broken() {
    var count by remember { mutableStateOf(0) }

    Text("$count")  // Read
    count++         // Write → triggers recomposition → repeat
}
```

Avoiding AntiPatterns

Forgotten Remember

```
//Good: ✅ Write only in callbacks
@Composable
fun Fixed() {
    var count by remember { mutableStateOf(0) }

    Text("$count")
    Button(onClick = { count++ }) {  // Write in event
        Text("Increment")
    }
}
```

Avoiding AntiPatterns

derivedStateOf Misue

```
//Useless: ❌ Output changes as often as input - no benefit
var firstName by remember { mutableStateOf("") }
var lastName by remember { mutableStateOf("") }

val fullName by remember {
    derivedStateOf { "$firstName $lastName" }  // Pointless
}
```

Avoiding AntiPatterns

derivedStateOf Misue

```
 // ✅ Just compute it
val fullName = "$firstName $lastName"

Correct use:
// ✅ Output changes LESS than input
val listState = rememberLazyListState()
val showButton by remember {
    derivedStateOf { listState.firstVisibleItemIndex > 0 }
}
// Scroll position changes every pixel
// Boolean changes rarely
```

Avoiding AntiPatterns

collectAsState vs collectAsStateWithLifecycle

Wasteful:

```
// ❌ Keeps collecting when app backgrounded
val state by viewModel.state.collectAsState()
```

collectAsState:
App backgrounded → still collecting → battery drain

Better:

```
// ✅ Lifecycle-aware - pauses when STOPPED
val state by viewModel.state.collectAsStateWithLifecycle()
```

collectAsStateWithLifecycle:
App backgrounded → paused → resumes when visible

Avoiding AntiPatterns

Unstable Lambda Captures

```
//Bad: ❌ New lambda every recomposition
 @Composable
 fun Parent() {
     var text by remember { mutableStateOf("") }

     Child(onClick = { doSomething() })  // New instance each time
     TextField(value = text, onValueChange = { text = it })
 }
 // TextField updates → Parent recomposes → new lambda → Child recomposes
```

Avoiding AntiPatterns

Unstable Lambda Captures

```
//Good: ✅ Stable lambda
@Composable
fun Parent() {
    var text by remember { mutableStateOf("") }

    val onClick = remember { { doSomething() } }
    Child(onClick = onClick)  // Same instance
    TextField(value = text, onValueChange = { text = it })
}
```

Climax

Four Rules

Separate

Model

Emit

Test

Climax

Four Rules

Separate — State lives where its lifecycle matches

Model

Emit

Test

Climax

Four Rules

Separate	State lives where its lifecycle matches
Model	Sealed interfaces, not boolean bags
Emit	
Test	

Climax

Four Rules

Separate	State lives where its lifecycle matches
Model	Sealed interfaces, not boolean bags
Emit	Effects as outputs, not inline code
Test	

Climax

Four Rules

Separate	State lives where its lifecycle matches
Model	Sealed interfaces, not boolean bags
Emit	Effects as outputs, not inline code
Test	Pure functions, no mocks

Thats all folks!

Slides

https://linktr.ee/aldefy