



Mobius 2026

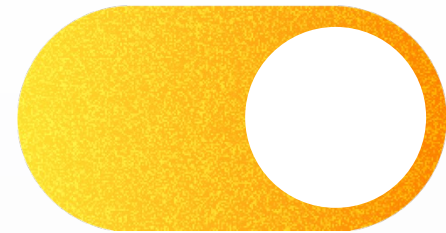
Александр Меньшиков

Максим Логунов

Multiplatform Time

Как завести мессенджер для Desktop + Mobile
и не прогореть

Александр
Меньшиков



Team Lead | iOS



Максим
Логунов

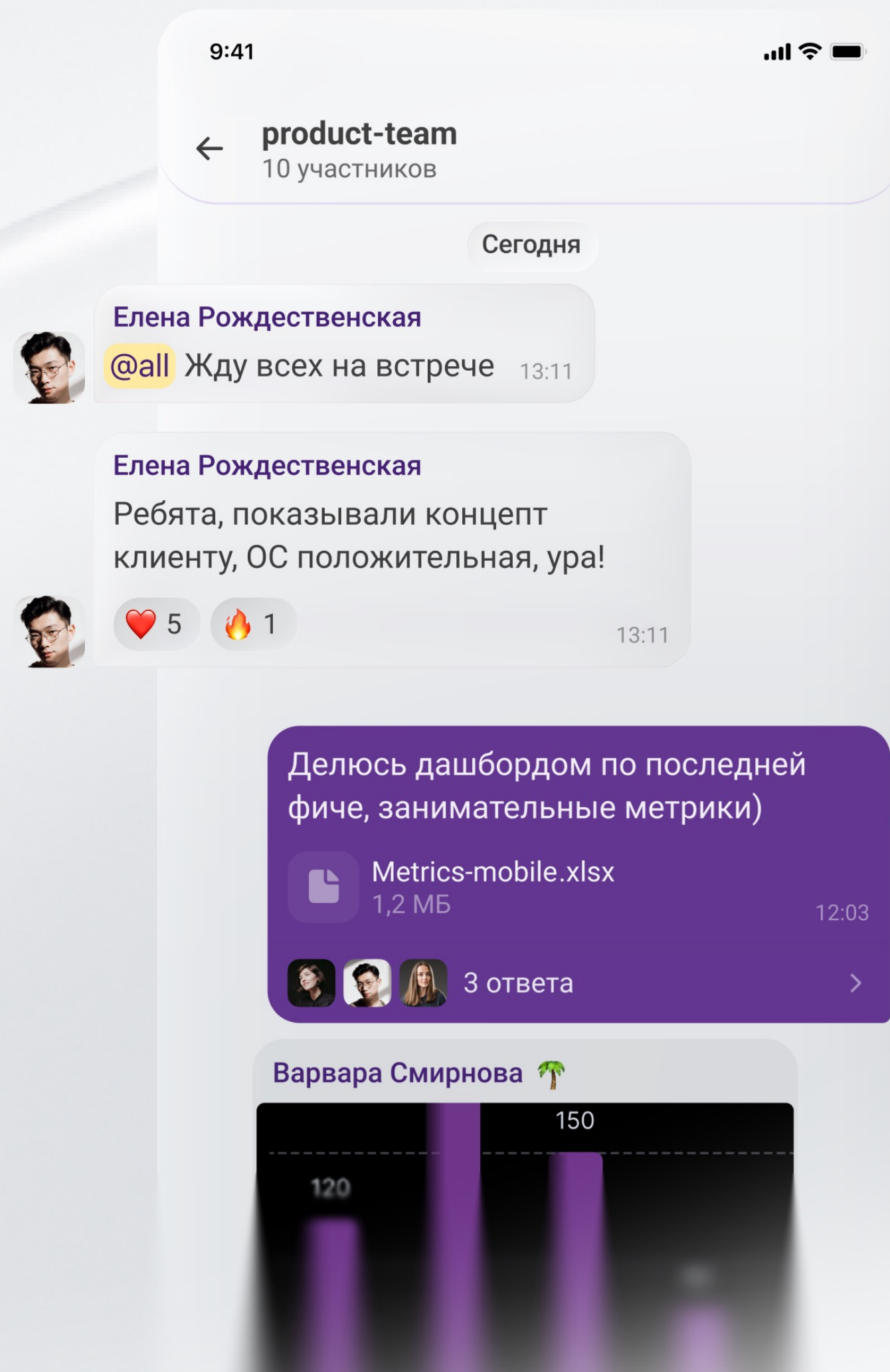


KMP | Android

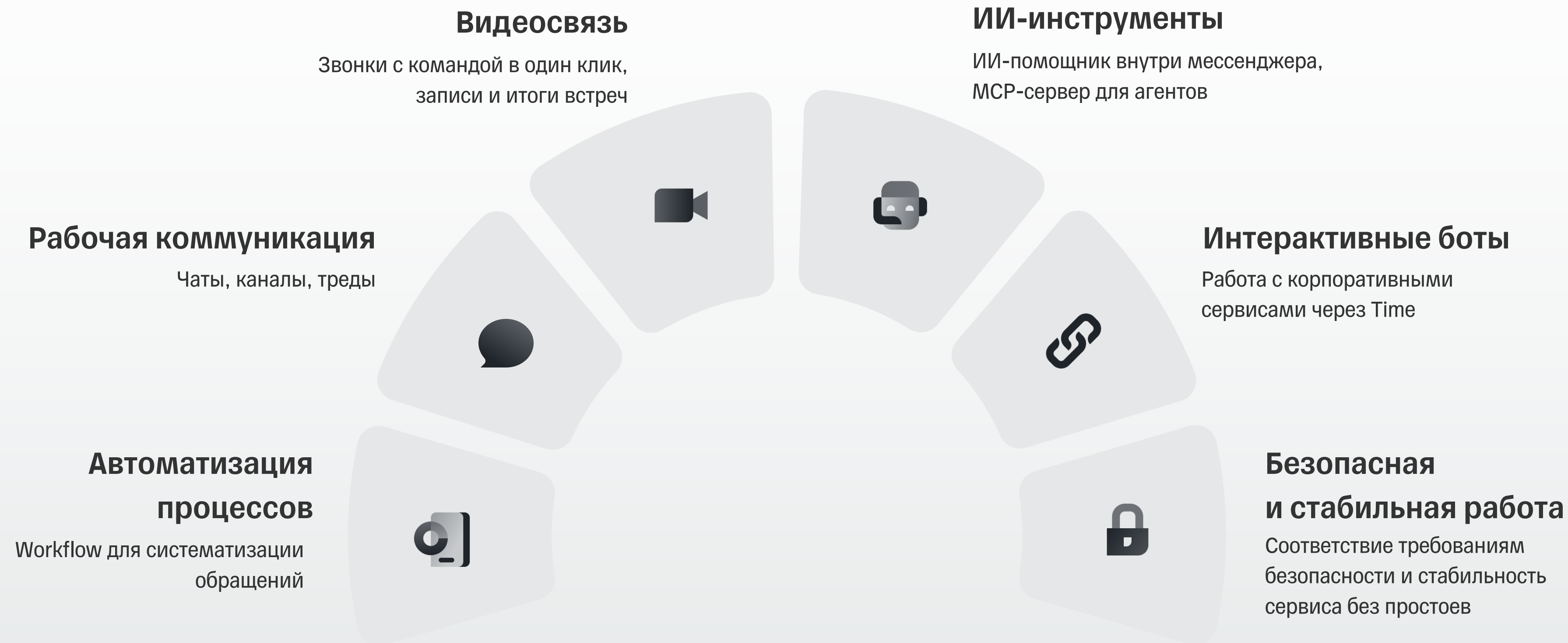


Что такое TIME?

- чаты, треды, реакции
- телефон, планшет, пк, браузер
- мультисерверность



Один мессенджер вместо десятка инструментов

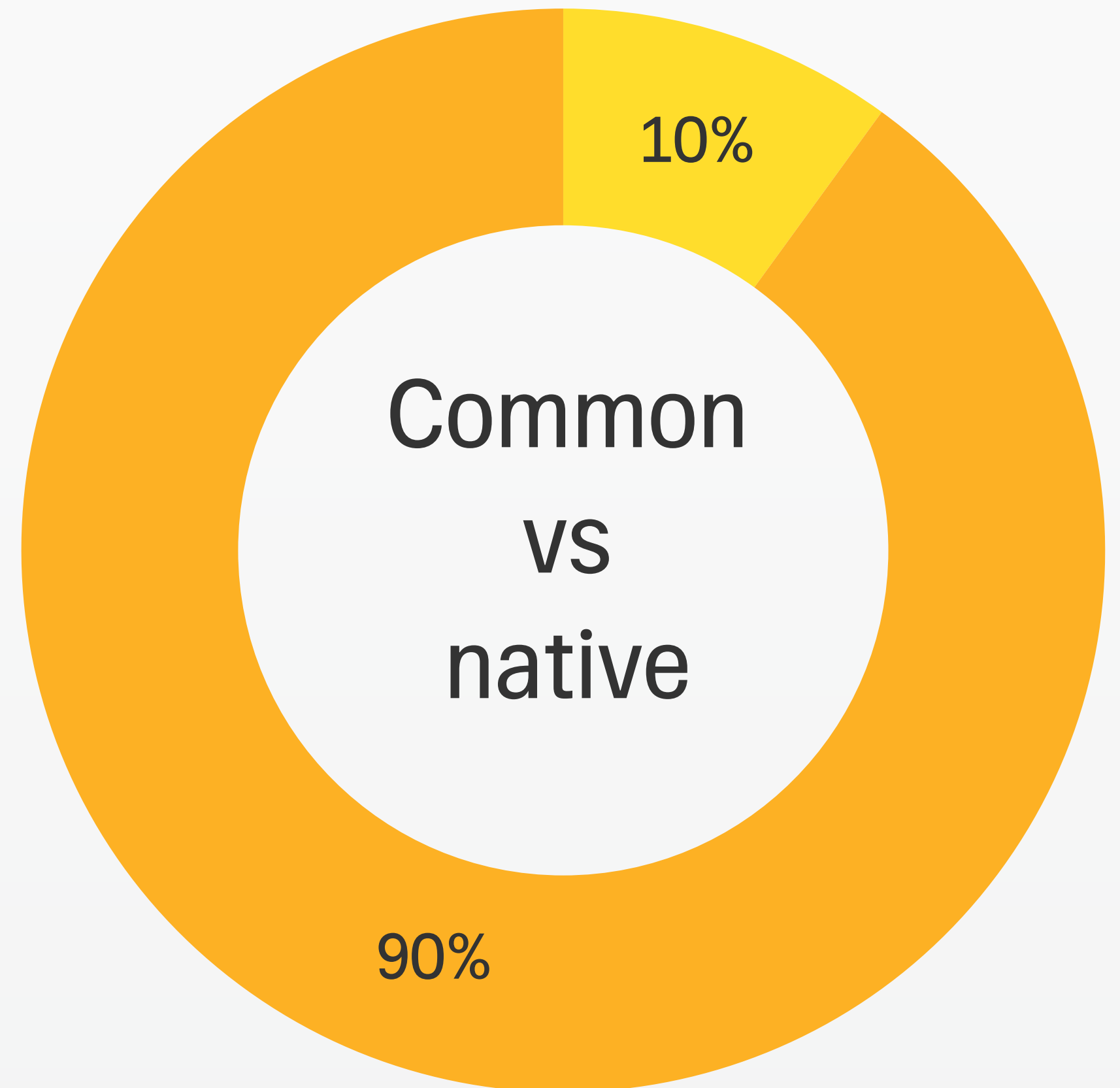


Нам доверяют



Что такое TIME?

- ➔ **240** Gradle-модулей
- ➔ **~90%** общего кода
- ➔ **1500** Swift-файлов



Карта доклада: где проходит граница

Compose: Android · Desktop

SwiftUI: iOS

Граница общего кода

01 Навигация — Decompose

02 Стейт — MVIKotlin

03 Граница с нативным iOS

04 Выводы

Как мы сюда пришли

Мультиплатформа была не итогом, а условием с первого дня

Сложилась модульная
структура, пошли фичи

Начало 2024

k2s-gen
выключен

Июнь 2026

Август 2023

Первый коммит
KMP добавили на
следующий день

Август 2025

Добавили свой
k2s-gen генератор

Сегодня

240+ модулей
3700+ коммитов



**Август 2023 — точка
отсчета, с которой
все начиналось**

Платформы и стек



Платформы

Android · Desktop · iOS · Wasm

UI:

- Compose MP на Android и Desktop
- Нативный SwiftUI на iOS



Стек

Kotlin Multiplatform · Decompose · MVIKotlin ·
Compose MP · SwiftUI

Инфраструктура:

Ktor · SQLDelight · Koin · SKIE

«Общий у нас не UI. Общее — всё, что под ним»

Это архитектурное решение, к которому мы пришли осознанно

План

**«Напишем один раз и
запустим везде»,
сэкономим примерно
половину бюджета**

Реальность

**~90% Kotlin-логики
пишется один раз. Но
оставшиеся 10% дают о
себе знать**

Что оказалось дорогим?

Это не «мелкая доработка
напильником». Это отдельный
инженерный проект со своим
бэклогом

Сборка

Собрать четыре платформы не
бесплатно, и на CI это
чувствуется каждый день

01

Интеграция с Xcode

KMP фреймворк, подписи,
ресурсы и Preview настраиваются
руками, меняются после
крупного обновления Xcode

02

Мост в Swift

Kotlin-классы превращаются в
то, с чем iOS-разработчику
работать неудобно

03

Нативный iOS

1500+ файлов на Swift,
которые нигде не
переиспользуются

04

**Даже с полностью нативным iOS
~90% кода написано один раз**

Но почему не Compose MP?

Compose MP

Desktop + Android

- Один UI на две платформы
- Тот же язык, что и логика
- Общие компоненты и тема

Счетчик “так исторически сложилось” — 1

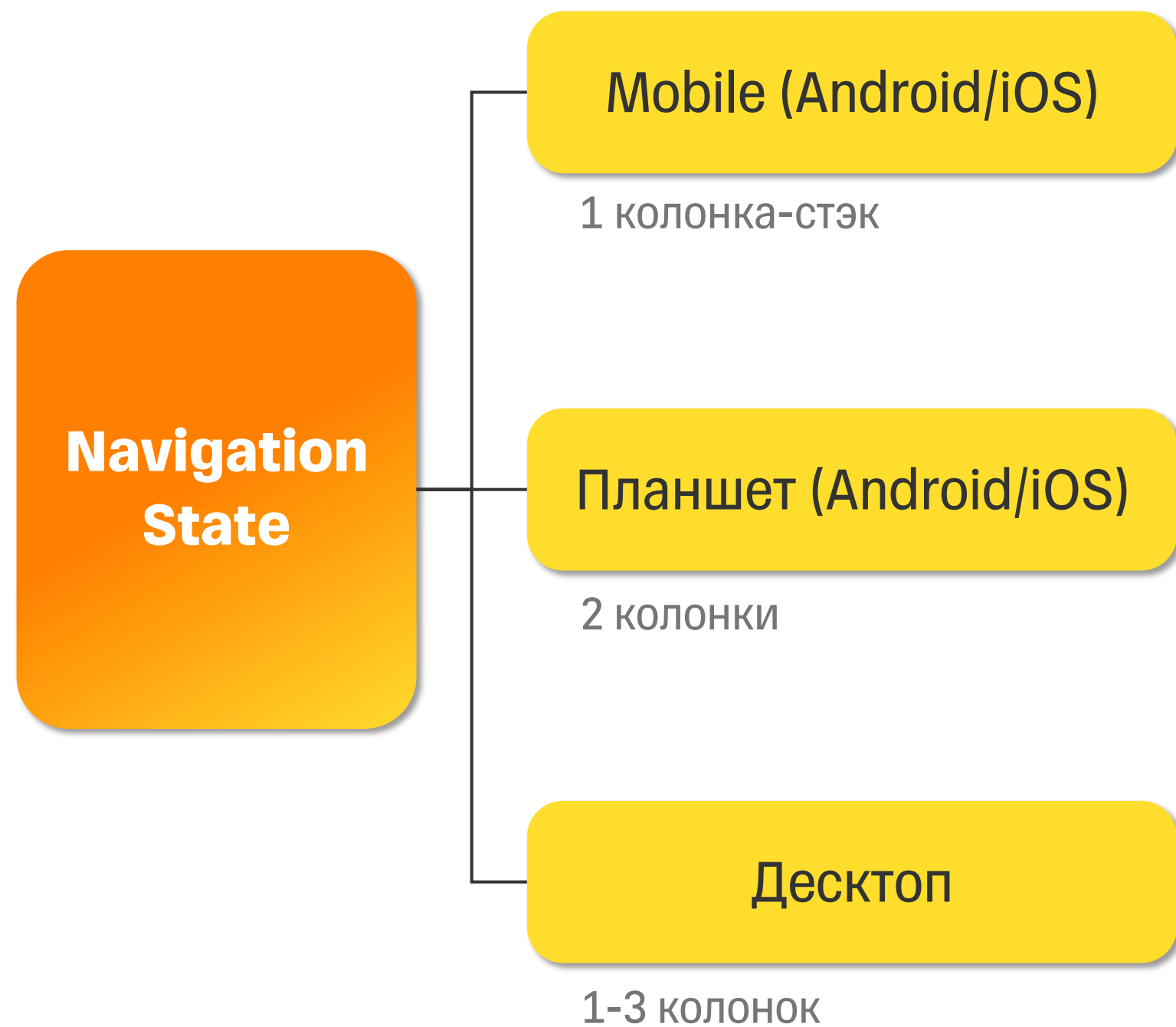
Нативный SwiftUI

iOS

- Родные жесты, accessibility
- Живой Preview
- Нативный интерфейс

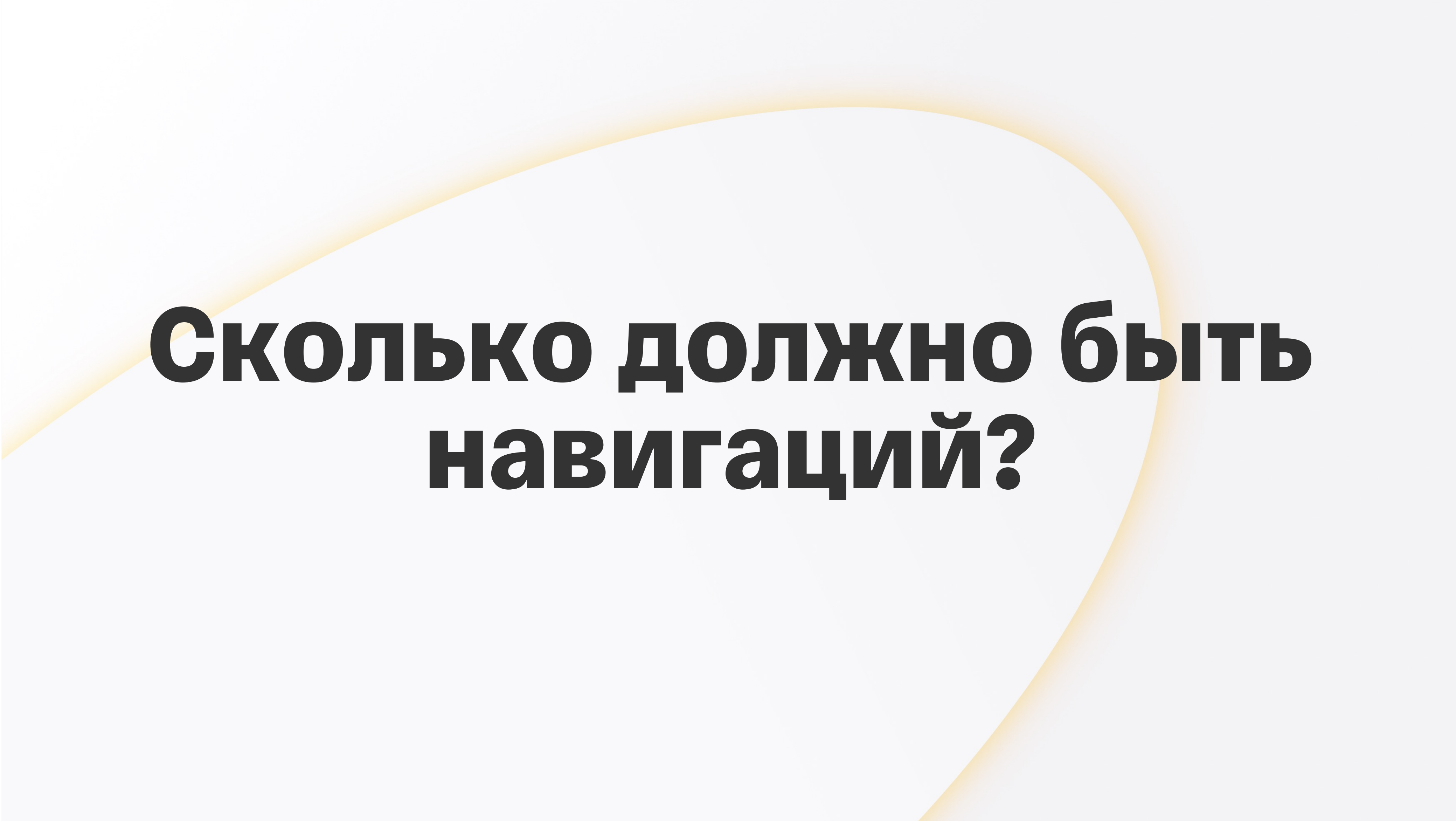


Push to Navigation



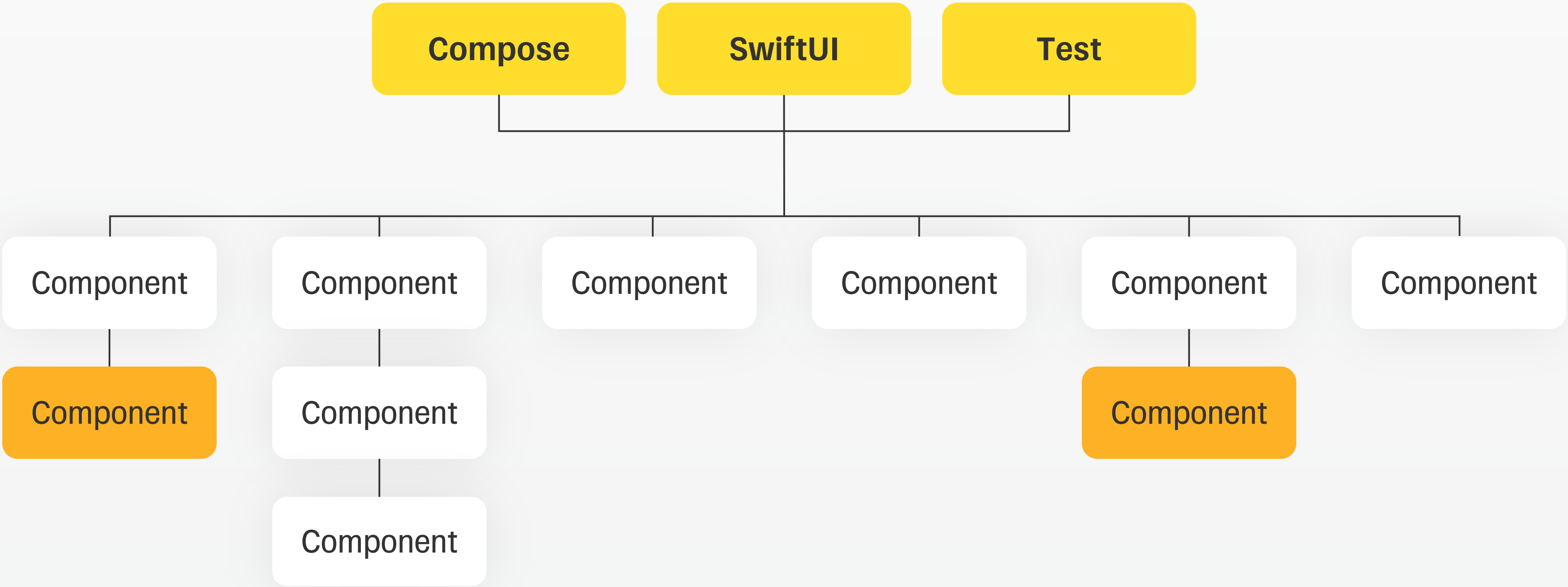
Одна навигация — три разных UI

Мессенджер — это классический master-detail, но его нужно реализовать под все таргеты



**СКОЛЬКО ДОЛЖНО БЫТЬ
НАВИГАЦИЙ?**

Decompose



Почему не Jetpack Navigation?

Причины

- ➔ В августе 2023 Jetpack Navigation для KMP ещё не было
- ➔ Jetpack Navigation остался Compose-only: состояние не отдать в SwiftUI
- ➔ Decompose как конструктор: компонент, lifecycle, стек, список, слот.
Нам нужны инструменты, а не готовый механизм

Три стека и фейковый экран

Sidebar Stack

Active

Content Stack

Active

Details Stack

Active

Три стека и фейковый экран

Sidebar Stack

Active

Pause

Content Stack

Active

Pause

Details Stack

Active

Sidebar Stack

Active

Pause

Content Stack

Active

Details Stack

Empty

Sidebar Stack

Active

Content Stack

Empty

Details Stack

Empty

Pause — **фейковый** экран, чтобы остановить СТЭК

Одна плоская история

Sidebar

Channel

Thread

Profile

Direct

Плоский стэк экранов через ItemsNavigation



Проекция в зависимости от видимых панелей



Sidebar Stack

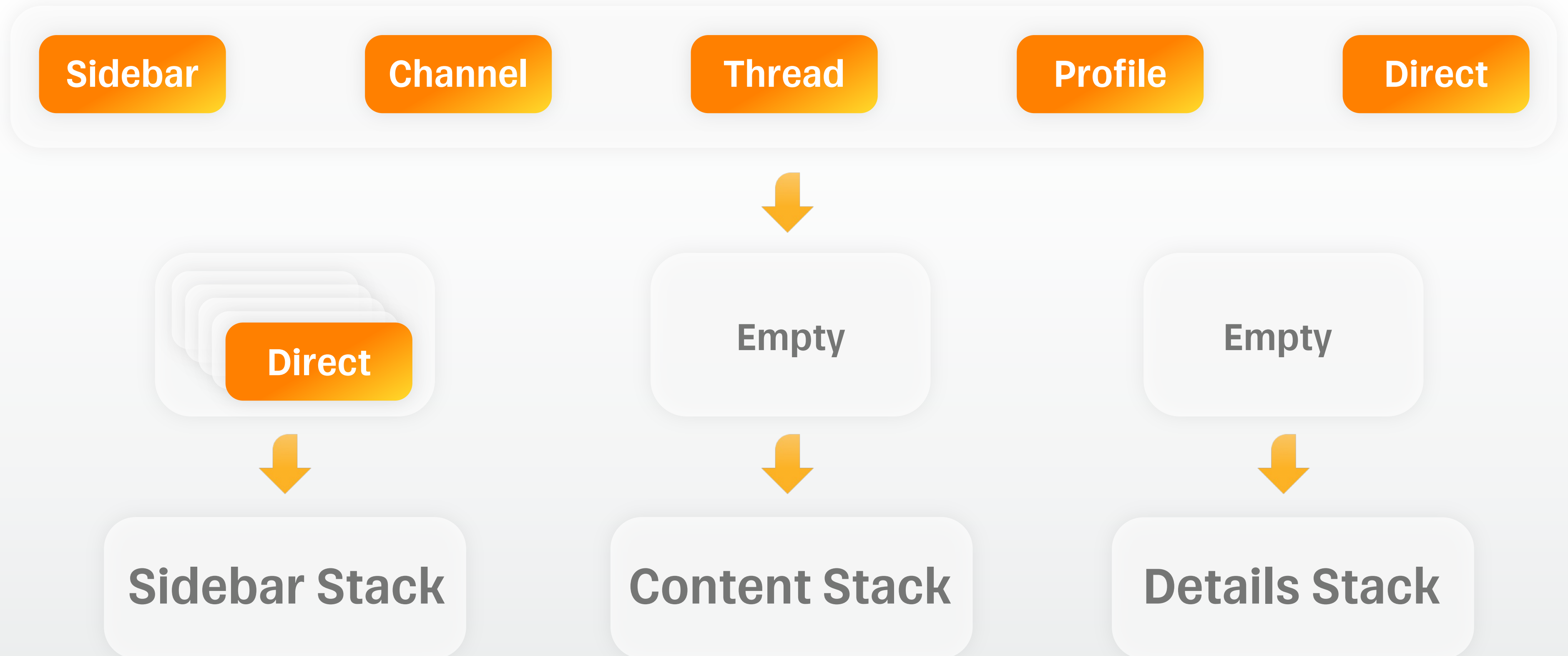


Content Stack

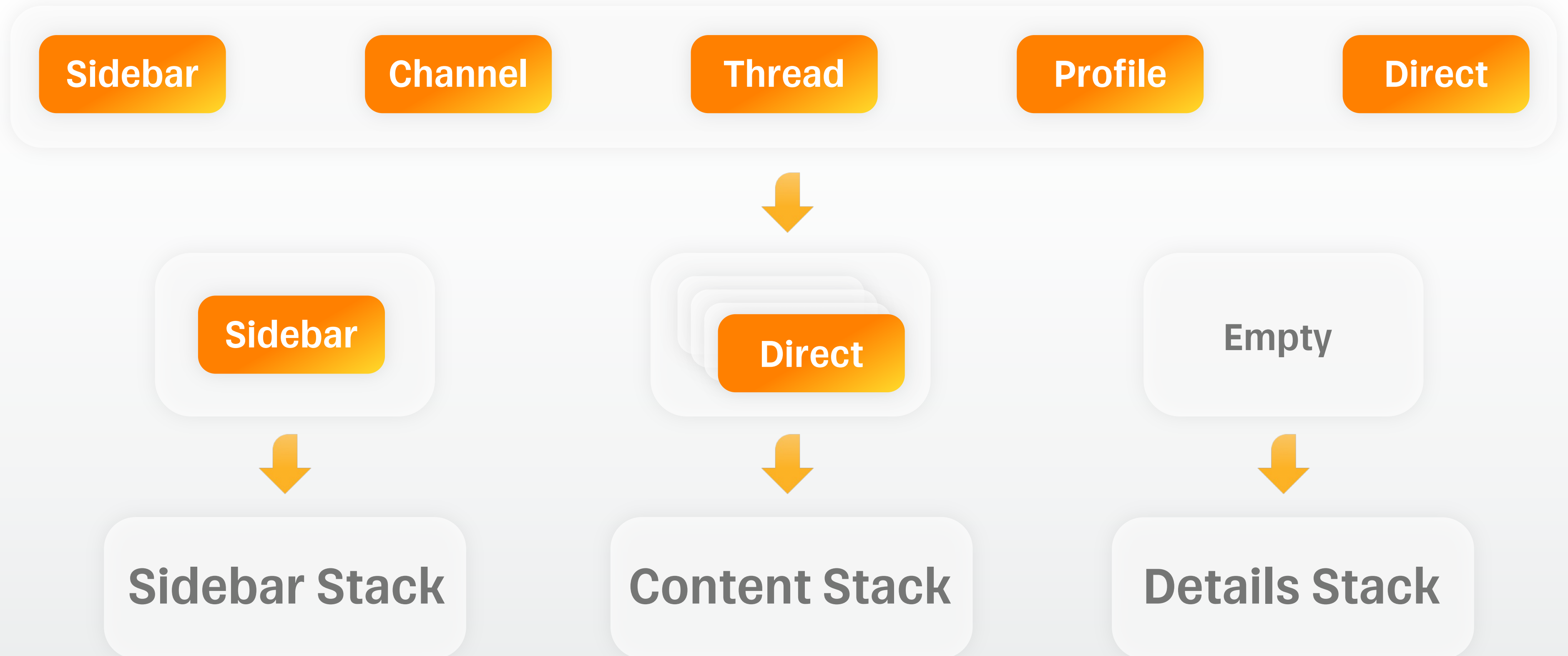


Details Stack

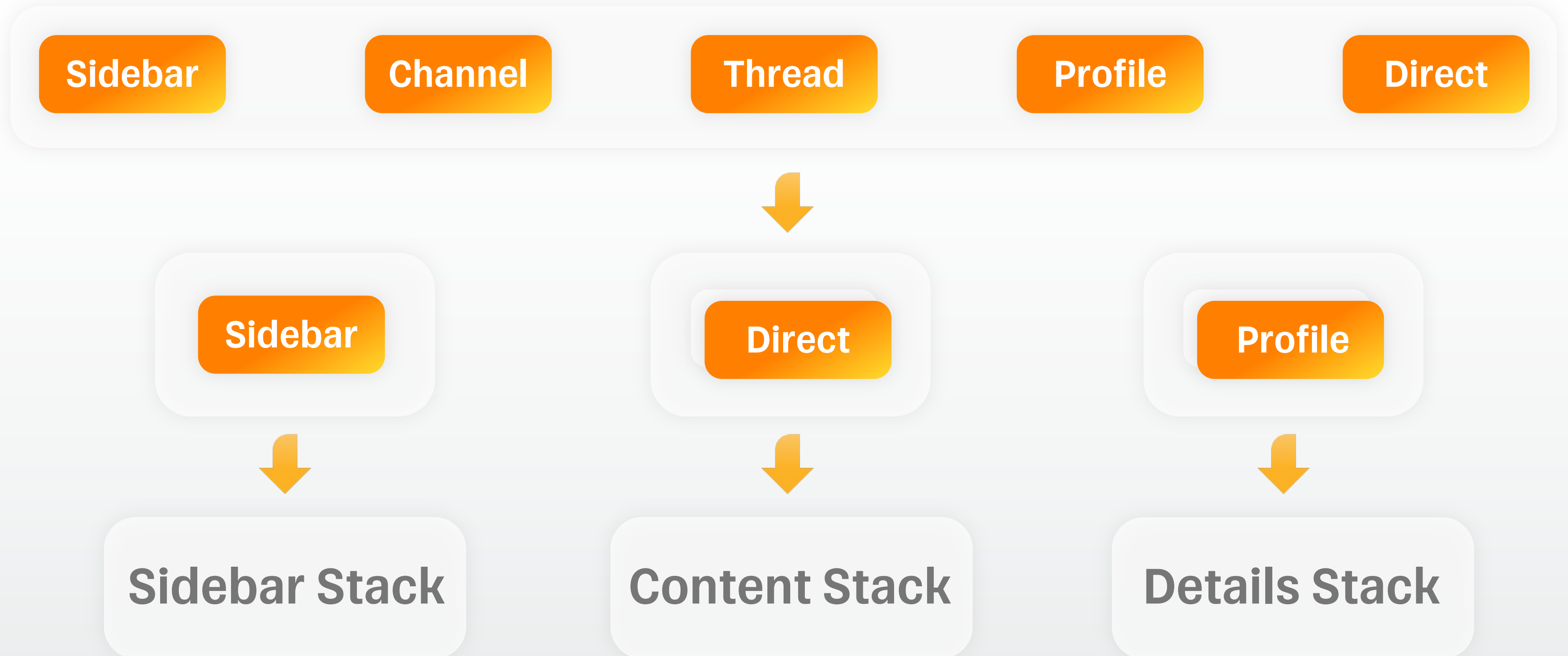
Одна плоская история

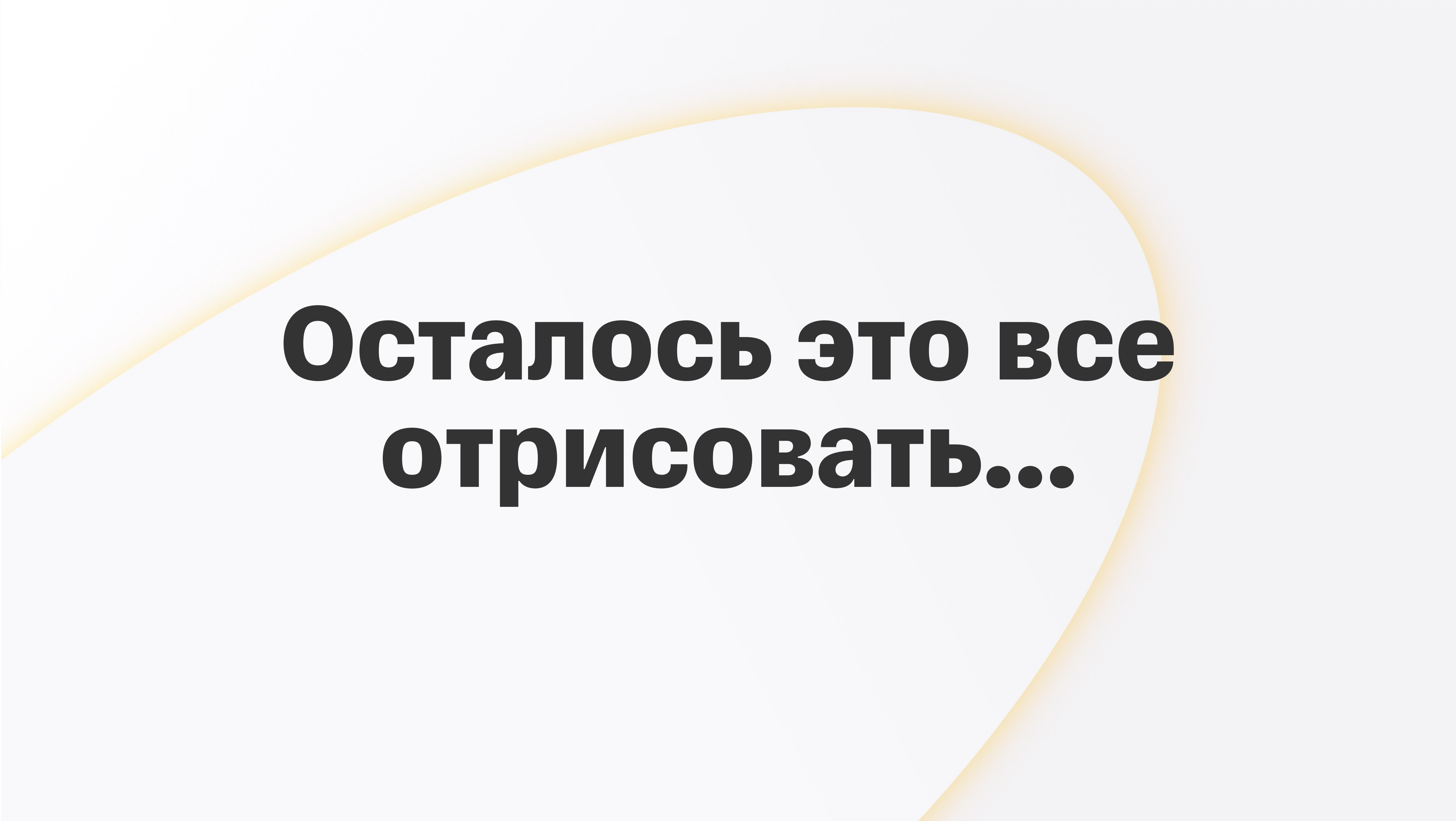


Одна плоская история



Одна плоская история





**Осталось это все
отрисовать...**

Свой layout: ширину задает UI

Но почему мы снова пишем свое решение?

01

Готовых решений
тогда не было

02

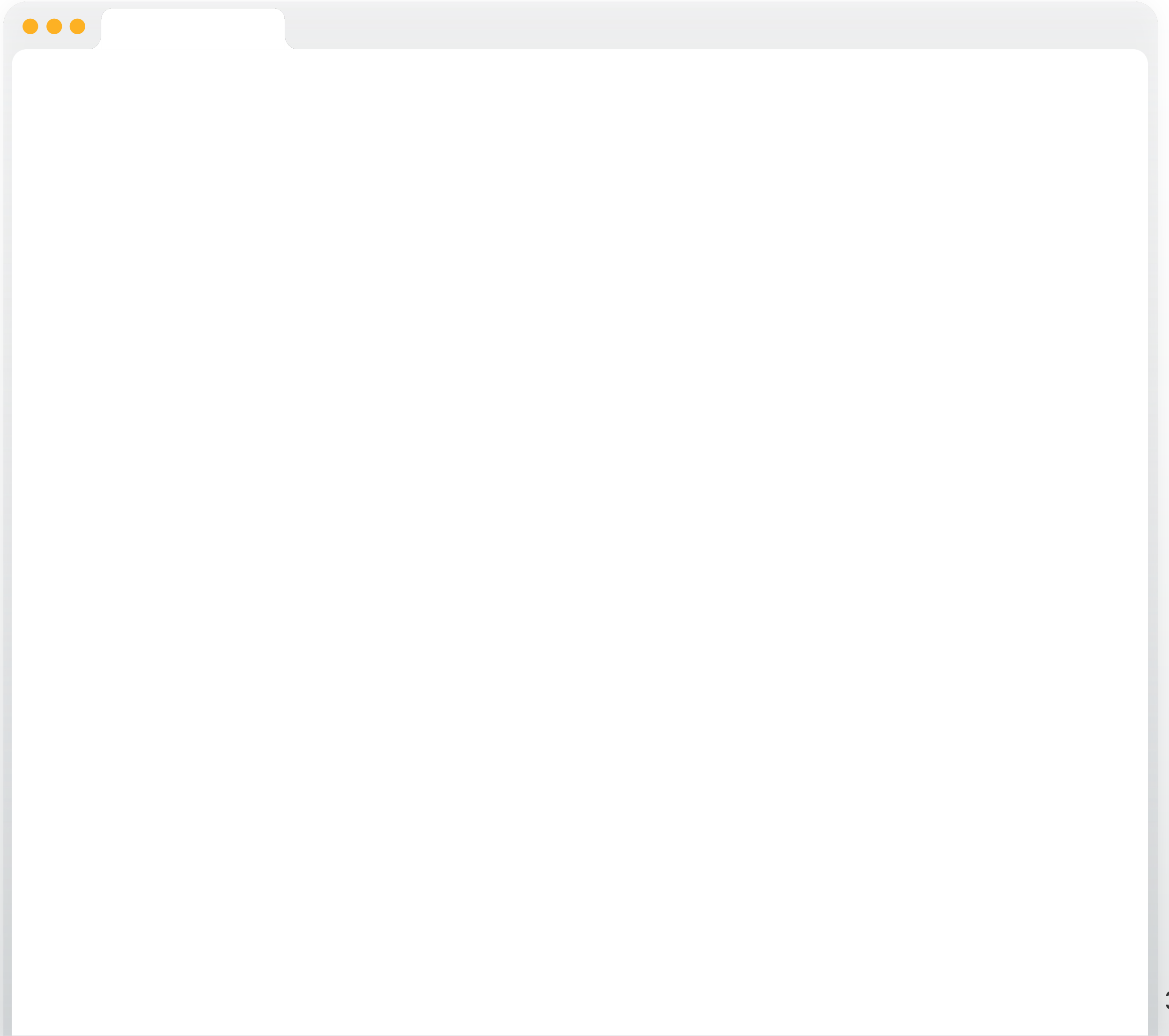
Нет возможности
менять ширину
панелей
пользователем

03

Динамическое
отображение
панелей

Демо

- ➔ Динамическое изменение количества панелей
- ➔ Сохранение стейта при переезде между панелями
- ➔ Бесшовный переход между состояниями



Как Compose переживает переезд?

Используем готовые инструменты из Decompose + Compose MP

SaveableStateHolder

Используется в реализации Children из Decompose для сохранения состояния



Movable Content

При переезде между панелями, Compose не пересоздаёт поддерево, а физически переносит его на новое место



**С Compose вроде все понятно,
а что со SwiftUI?**

Тот же стек — нативный SwiftUI

Compose

ThreePaneScaffold



Navigation State

commonMain



SwiftUI

iPhone: NavigationStack

iPad: три панели нативно

```
SwiftUINav ×
NavigationStack(path: Binding(
    get: { stack },
    set: { _ in component.dispatch(intent: MainIntent.Back()) }
))
```

Свайп назад не меняет состояние — **он просит его изменить**
Единственный источник правды остаётся в Kotlin

Разложение по девайсам: compact и regular

MainFeatureView

horizontalSizeClass



Navigation State

вьюмодель мержит стеки



Regular · iPad и Mac

SidebarNavigationView
три NavigationStack

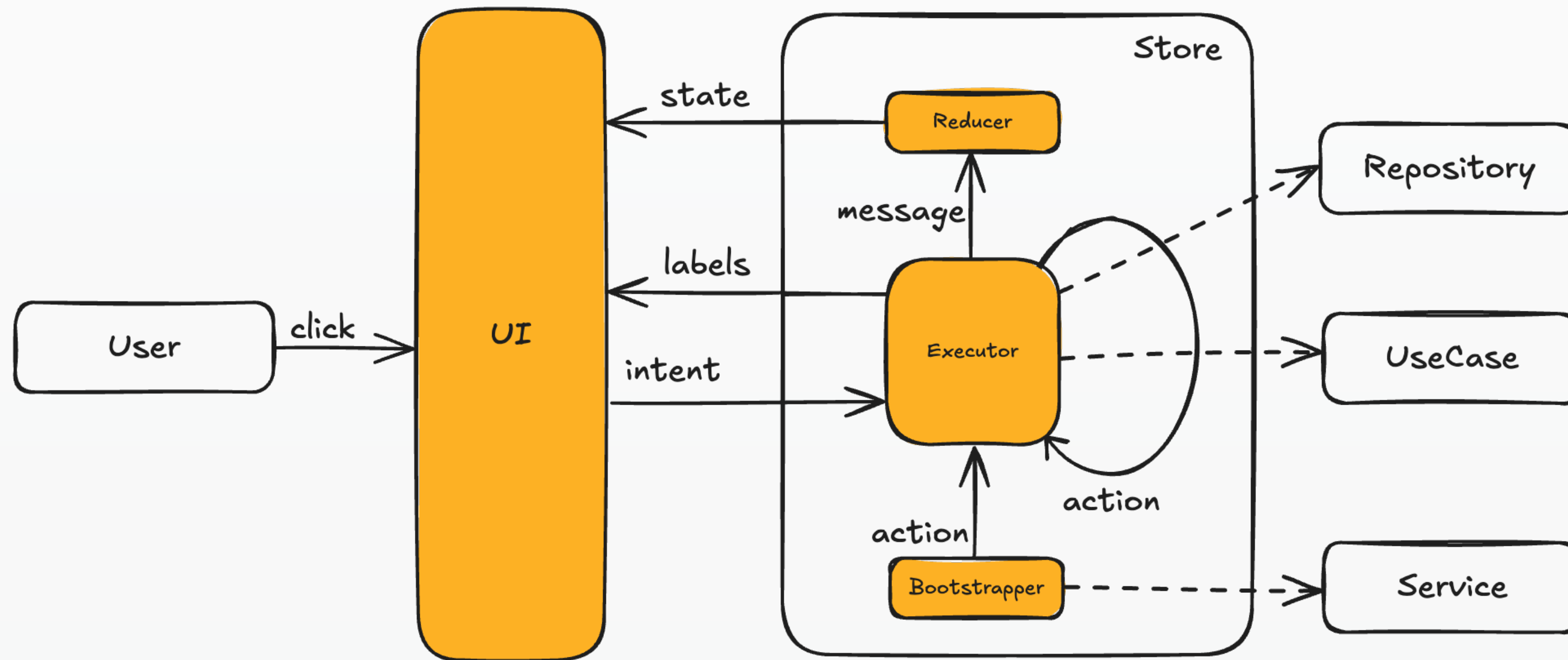
```
private func getUpdatedStack() -> [ChildCreated] {
    var stack = sidebarStack.dropFirst()
    stack.append(contentsOf: contentStack)
    stack.append(contentsOf: detailsStack)
    return stack.compactMap{withoutEmpty}
}
```

Compact: три стека Kotlin сливаются в один path

Regular: стеки остаются тремя нативными NavigationStack

MVI: как, зачем и почему?

MVIKotlin



Чем плох один Executor

Обработка всех событий в одном
классе

≡ Executor ×

```
fun executeAction(action: Action) {  
    when (action) {  
        is Action.FetchData -> fetchSmth()  
        is Action.SaveSettings -> saveSettings()  
        is Action.Retry -> retrySave()  
        ...  
    }  
}  
  
fun executeIntent(intent: Intent) {  
    when (intent) {  
        is Intent.SendMessage -> sendMessage()  
        is Intent.ClearInput -> clearInput()  
        is Intent.ShowMenu -> showMenu()  
        ...  
    }  
}
```

Чем плох один Executor

Контейнер для всех зависимостей

≡ Executor ×

```
class SomeExecutor(  
    private val userRepository: UserRepository,  
    private val chatRepository: ChatRepository,  
    private val postsRepository: PostsRepository,  
    private val connectionUseCase: ConnectionUseCase,  
    private val sendMessageUseCase: SendUseCase,  
    private val sessionManager: SessionManager,  
    private val dispatchers: Dispatchers,  
    private val postsMapper: PostsMapper,  
    private val usersMapper: UsersMapper,  
    ...  
) : Executor() {  
    ...  
}
```


Чем плох один Executor

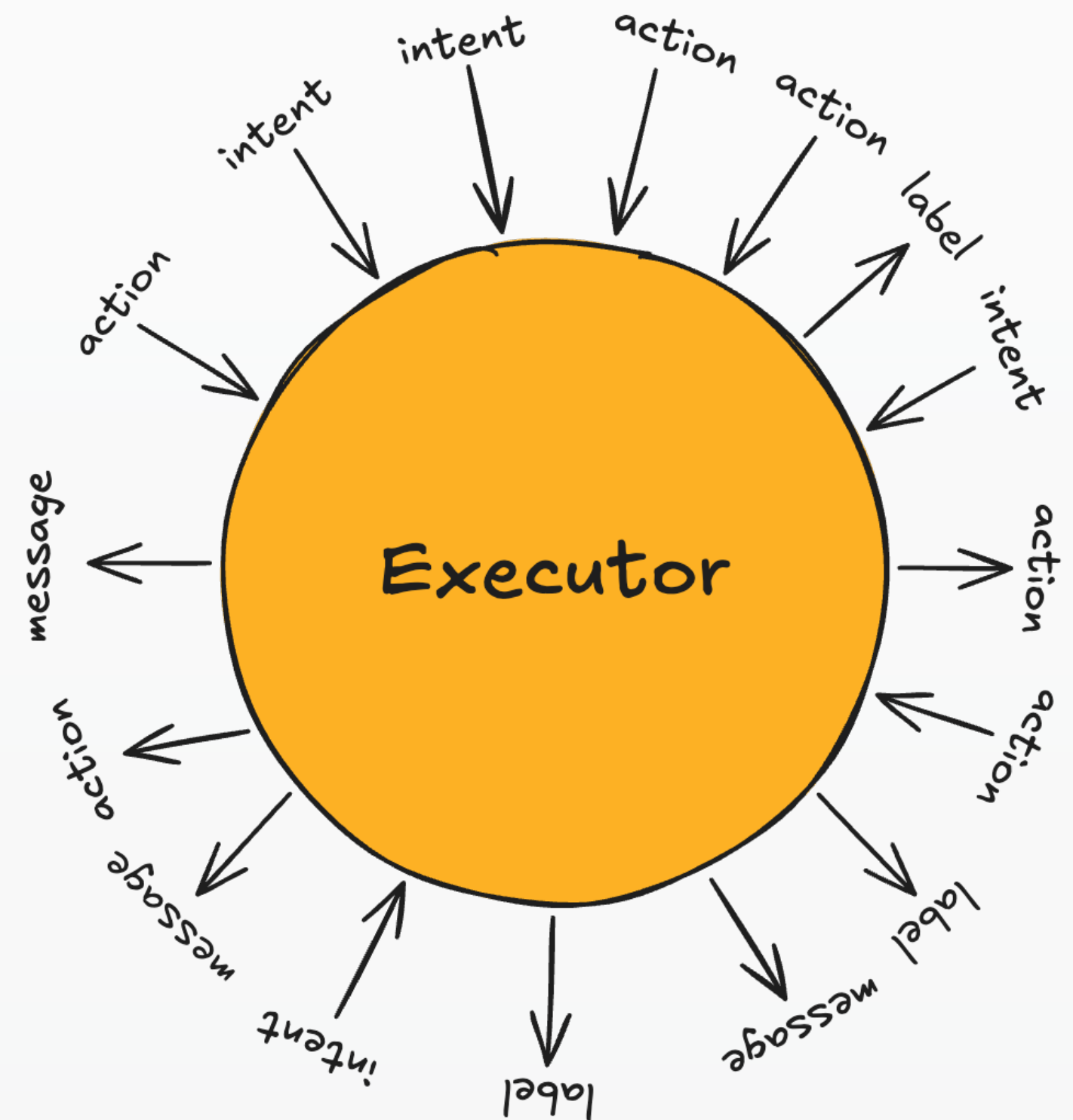
Сложно тестировать

≡ Executor ×

```
class SomeExecutorTest {  
  
    private val userRepository = mock()  
    private val chatRepository = mock()  
    private val postsRepository = mock()  
    private val connectionUseCase = mock()  
    private val sendMessageUseCase = mock()  
    private val sessionManager = mock()  
  
    ...  
  
    fun testUsers()  
    fun testMessages()  
    fun testPagination()  
}
```

Чем плох один Executor

- 01 Обработка всех событий в одном классе
- 02 Контейнер для всех зависимостей
- 03 Сложно тестировать
- 04 Частые конфликты
- 05 Сложно поддерживать



Чем плох один Executor

Тогда

250

строк — пик ChatExecutor
декабрь 2024



рост ≈ 50 строк в месяц

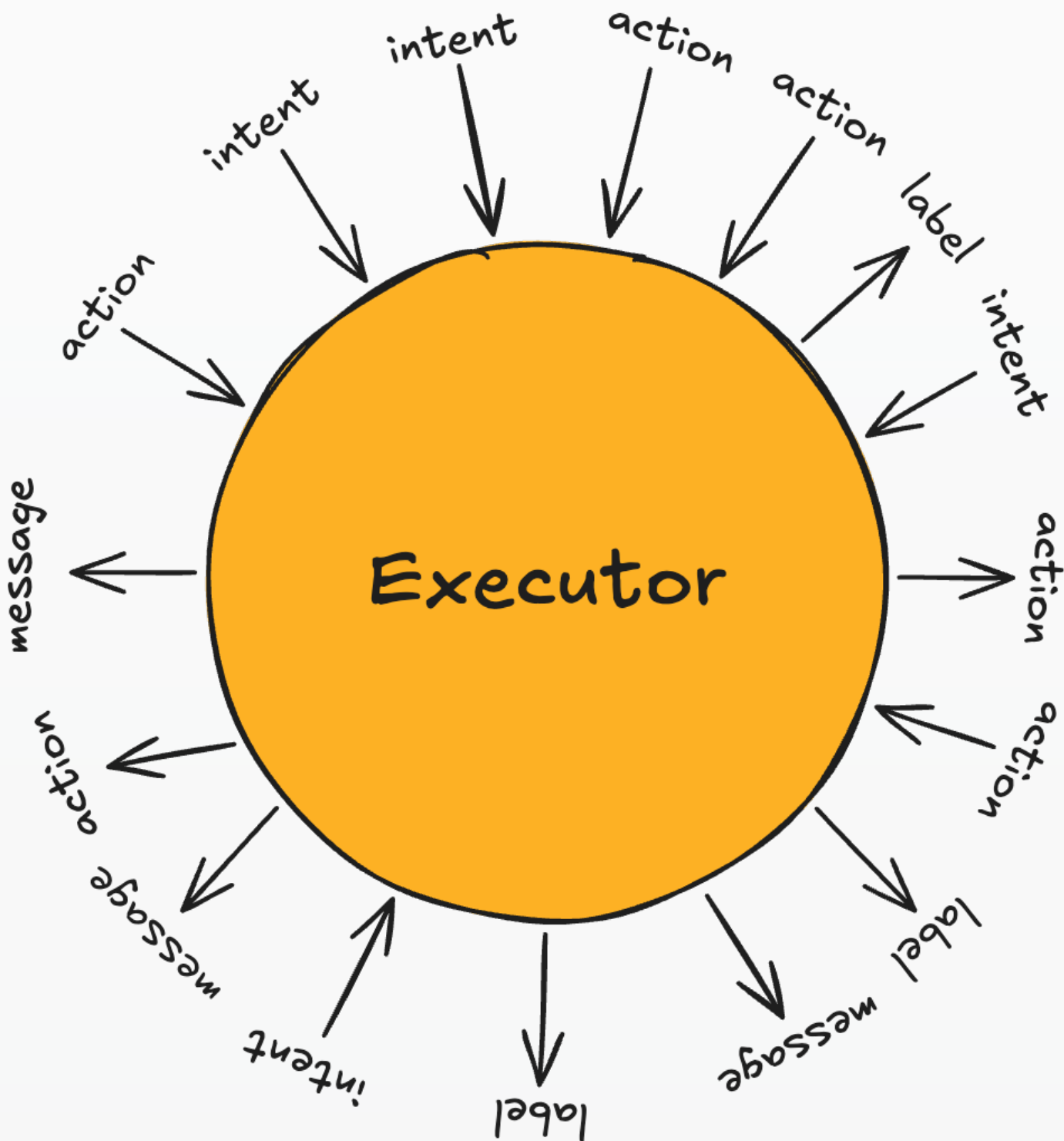
Сегодня

3317

строк кода

66

классов



Composite Executor

- Не содержит никакой логики
- Делегирует работу хэндлерам
- Параллельная обработка
- Новый хендлер = новый файл + одна строка в DI

IntentHandler

Реагирует на действия
пользователя



ActionHandler

Работает с внутренними
событиями стора



FlowHandler

Реактивные подписки, работа с
потоками данных



AnalyticsHandler

Сквозной трекинг всех событий
стора



Что получилось в результате?

400+

хендлеров
~20 000 строк логики

40+

сторон живут на этом
паттерне

65+

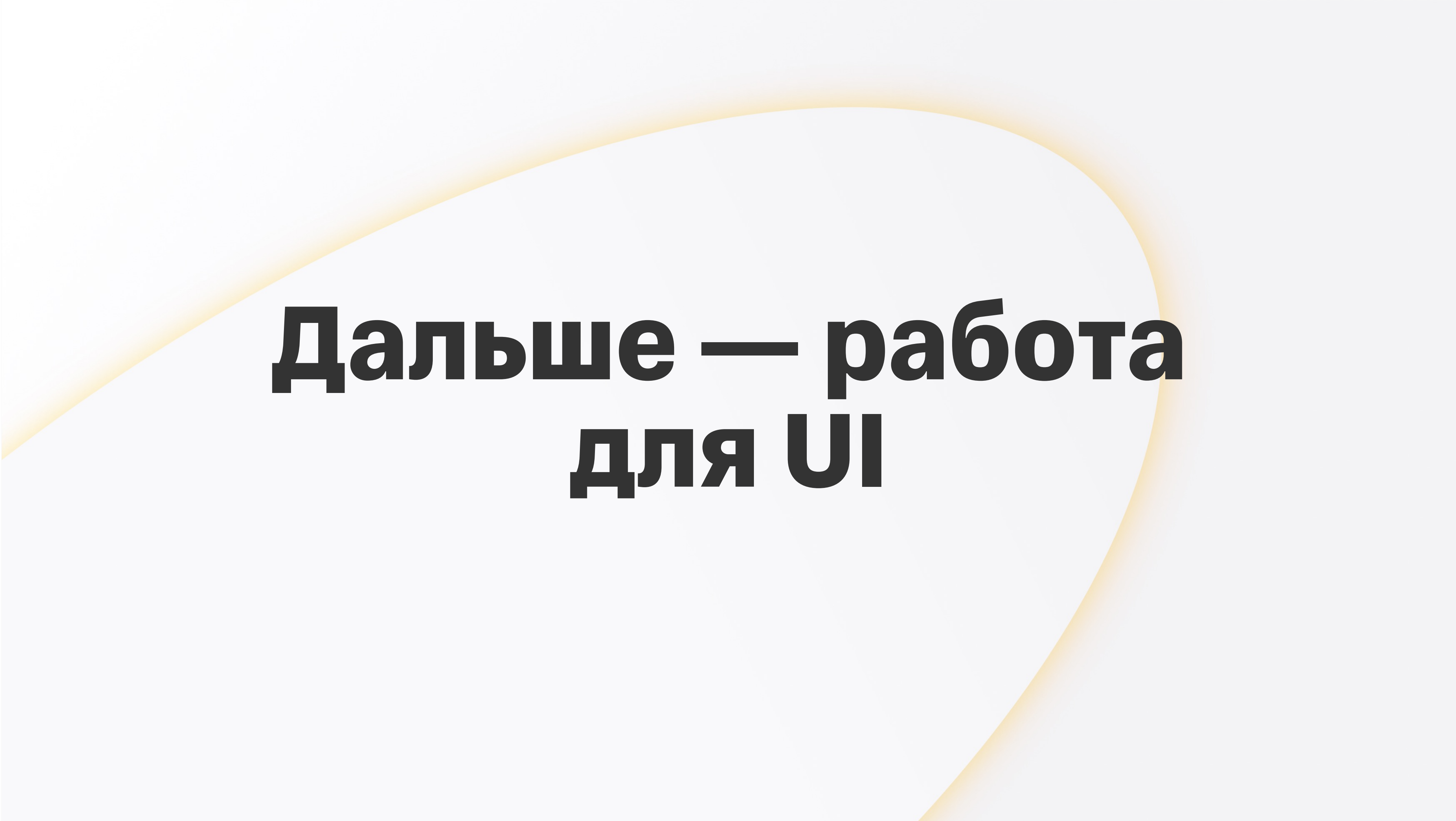
хендлеров на одном
экране чата

142

строки — самый
большой Executor
сегодня

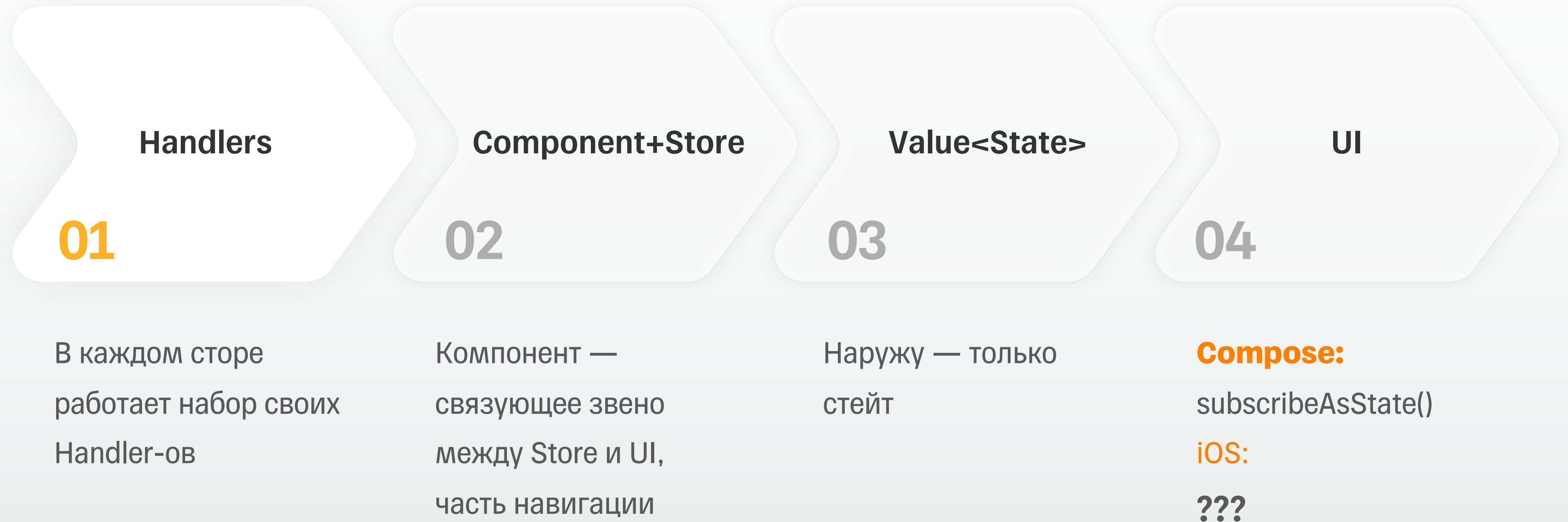
ChatExecutor ×

```
class ChatExecutor(  
    intentsHandlers: List<IntentHandler>,  
    actionHandlers: List<ActionHandler>,  
    flowHandlers: List<FlowHandler>,  
    analyticsHandler: AnalyticsHandler,  
) : CompositeExecutor<  
    ChatIntent,  
    ChatAction,  
    ChatState,  
    ChatMessage,  
    ChatLabel>(  
    intentsHandlers,  
    actionHandlers,  
    flowHandlers,  
    analyticsHandler,  
)
```



**Дальше — работа
для UI**

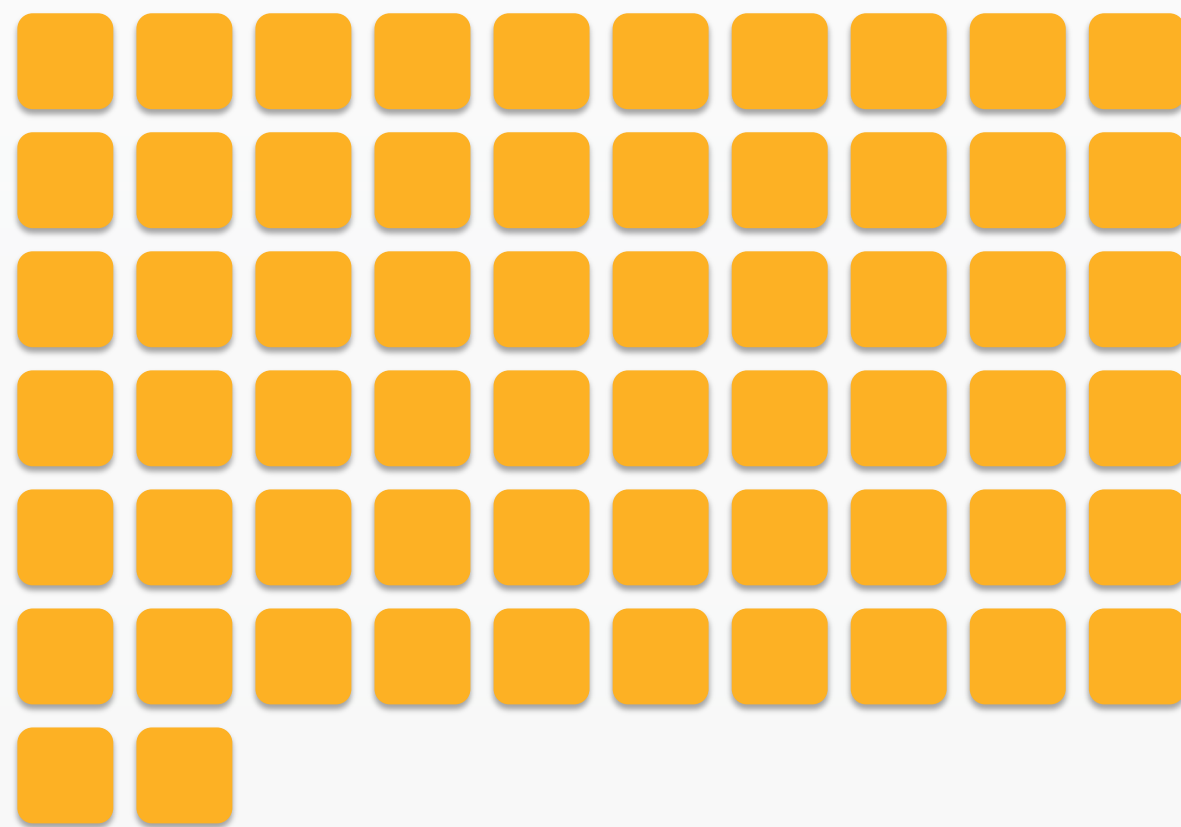
Что мы отдаем наружу



Где кончается общий код?

Граница с нативным iOS: мост и его цена

Как Kotlin попадает в Xcode?



60+ KMP-модулей



MirageShared

один статичный XCFramework

Static фреймворк
ломает превью

85% iOS-кода — обычный Swift

200 / 1400

Только 15% Swift-
файлов вообще знают
про KMP

19 / 19

SPM-пакетов —
чистый Swift без
Kotlin

app-таргет

единственное место,
где живёт Shared
фреймворк

Как mirage-таргет получает Kotlin

app-таргет Mirage

build phase



gradlew

:shared:embedAndSign...



MirageShared

статичный XCFramework
Framework Search Paths

```
Compact ×

// Build Phase: Compile Kotlin Framework
JAVA_VERSION=21 "$SRCROOT/../../gradlew" \
    :shared:embedAndSignAppleFrameworkForXcode

// Framework Search Paths:
// shared/build/xcode-frameworks/$(CONFIGURATION)
```

Сборкой управляет Gradle — Xcode только дёргает таск по кнопке
SPM-пакеты фич собираются отдельно и не знают про Kotlin

Что берёт на себя SKIE

```
Default  x  SKIE

@interface MirageChatEvent : NSObject
@end

@interface MirageMessageAdded
    : MirageChatEvent
@property (readonly) NSString *messageId;
@end

// sealed → набор классов,
// exhaustive-проверки нет:
if ([e isKindOfClass:Add.class]) {}
else if ([e isKindOfClass:Read.class]) {}
// новый case ловится в рантайме

// suspend → колбэки, Flow недоступен,
// имена с Obj-C-префиксами
```



```
Default  SKIE  x

switch onEnum(of: event) {
case .messageAdded(let e):
    render(e)
case .readChanged(let e):
    markRead(e)
}
// честный exhaustive switch

// suspend → async/await
let state = try await component.state()

// Flow → AsyncSequence
for await msg in component.messages {
    append(msg)
}
```


Что дописываем руками

Состояние: Value →

CoreStateValue

CoreStateValue ×

```
protocol CoreStateValue<ValueType>: AnyObject {  
  
    var value: ValueType { get }  
  
    func subscribe(  
        observer: @escaping (ValueType) -> Void  
    ) -> CoreCancellation  
}
```

Что дописываем руками

@retroactive Identifiable
для ForEach

☰ Extensions ×

```
extension SidebarChannel: @retroactive Identifiable {}
extension SidebarCategoryUiItem: @retroactive Identifiable {}
extension Team: @retroactive Identifiable {}
extension MainComponentFeature: @retroactive Identifiable {}
extension PostReaction: @retroactive Identifiable {}
extension PostAttachedFile: @retroactive Identifiable {}
extension ChatListItem: @retroactive Identifiable {}
extension PostMenuItem: @retroactive Identifiable {}
extension ServerWithTeams: @retroactive Identifiable {}
extension TeamItem: @retroactive Identifiable {}
extension SearchCategoryPage: @retroactive Identifiable {}
extension UserCategoryItem: @retroactive Identifiable {}
extension ChannelCategoryItem: @retroactive Identifiable {}
extension BotCategoryItem: @retroactive Identifiable {}
...
```


Что дописываем руками

Маппинг навигации:

Feature Config →

SwiftUI экран

ChildView ×

```
public func renderChildView(
    item: ChildCreated<any PaneConfig, MainComponentFeature>
) -> some View {
    switch onEnum(of: item.instance) {
    case .chatFeature(let feature):
        ChatFeatureView(component: feature.component)
    case .userProfileFeature(let feature):
        DirectProfileDetailsView(
            viewModel: ViewModel.create(feature.component)
        )
    case .teamSelectorFeature(let feature):
        TeamSelectorView(
            viewModel: ViewModel.create(feature.component)
        )
    ...
    }
}
```




**Каждый апдейт
стейта замедлял UI**

Почему на iOS апдейт стейта был дороже

Compose

перерисовывается только то, что изменилось в Compose-функции

SwiftUI

стейт приезжает одним объектом — тело экрана пересчитывается целиком



Как компонент соединяется с фичей

Kotlin component

Value<State>



прослойка

Wrapper + Mapper



ReminderFeature

чистый Swift-пакет
ReminderViewAssembly

```
Assembly ×
ReminderViewAssembly.makeModule(
    state: MappedStateValue(
        source: StateValueWrapper(component.state),
        transform: ReminderStateMapper.map
    ),
    actions: ReminderActions(
        handleSaveDuration: { component.dispatch(intent: .Confirm()) }
    )
)
```

Фича видит только свои Swift-типы и замыкания
Весь клей — мапперы «+MirageShared.swift»

Пайплайн данных: Value → CoreStateValue

component.state

Value<T> · Kotlin



Wrapper

CoreStateValue



Feature

ReminderViewModel
@Observable

```
Bridge ×
// Core, pure Swift:
protocol CoreStateValue<ValueType>: AnyObject {
    var value: ValueType { get }
    func subscribe(observer: @escaping (ValueType) -> Void)
        -> CoreCancellation
}
```

Фича зависит только от протокола Core — Kotlin внутрь не попадает
Для превью и тестов — PreviewStateValue: заглушка со значениями руками

МAPPING: Kotlin-стейт → Swift-структура

ReminderState

Kotlin



Mapper

+MirageShared.swift



ScreenState

Swift-структура
СВОИ ТИПЫ

```
Mapper ×
// ReminderScreenState+MirageShared.swift
ReminderScreenState(
    title: state.title.localized(),
    items: state.items.map { ReminderDurationItem(item: $0) },
    selectedDate: state.dateTime?.instant.toDate ?? Date(),
    isLoading: state.isLoading
)
```

Маппер — единственное место, где фича видит Kotlin-типы
Строки — .localized(), время — в Date, sealed — onEnum(of:)

ViewModel фичи: ТОЛЬКО СВОИ ТИПЫ

CoreStateValue

ValueType = ScreenState



ViewModel

@Observable



ReminderView

SwiftUI-экран
чистый Swift

```
ViewModel ×

@Observable final class ReminderViewModel {
    init(state: some CoreStateValue<ReminderScreenState>,
        actions: ReminderActions) {
        updateState(state.value) // first value
        state.subscribe { [weak self] in self?.updateState($0) }
    }
}
```

Фича работает с данными как с обычным Swift: СВОИ ТИПЫ, СВОИ ПОЛЯ
Стеки — тот же путь через KMPItemPublisher: дедупликация + главный поток

A decorative yellow arc with a gradient, starting from the left edge and curving upwards and to the right, framing the text.

**Общий паттерн —
триггер кодогенерации**

k2s-gen: кодоген, который мы убили через 9 месяцев

Успешно используем
k2s-gen во многих фичах

Понимаем, что поддерживать
генератор становится неудобно

26.08.2025

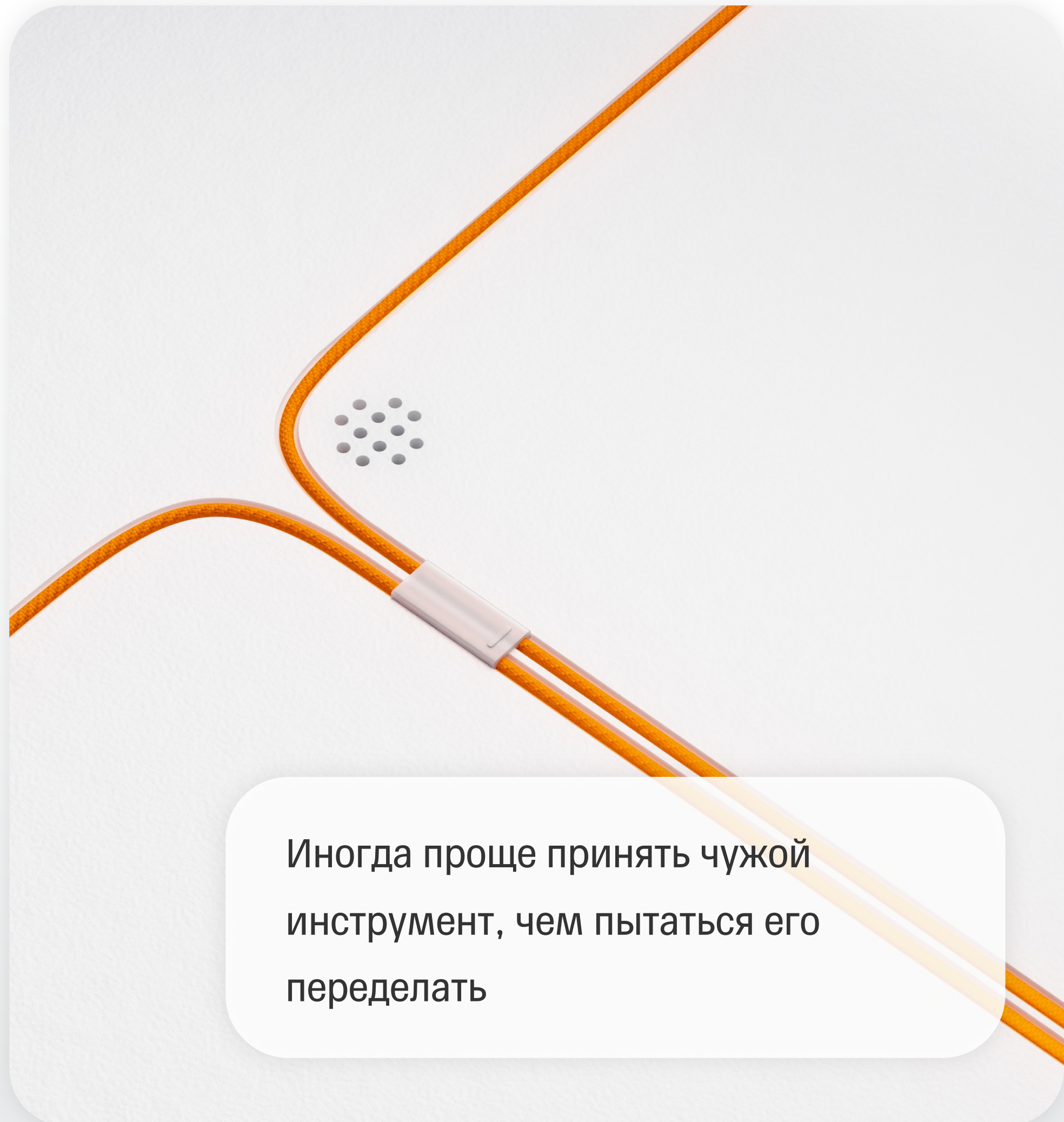
Создан KSP-процессор
на SwiftPoet

9 месяцев

SKIE и Xcode меняются,
приходится чинить каждый раз

04.06.2026

k2s-gen — @Deprecated



Иногда проще принять чужой
инструмент, чем пытаться его
переделать

Генератор поверх генератора стареет быстрее, чем окупается

Двести строк мы написали один раз,
и с тех пор не трогаем вообще

Итог по границе: Kotlin ↔ Swift

01

Взяли готовое

SKIE:

- sealed → enum
- suspend → async/await
- Flow → AsyncSequence
- чистые имена

02

Написали сами

- 181 строка для стейта
- 27 расширений Identifiable
- switch навигации на 25 экранов

03

Похоронили

k2s-gen — 9 месяцев и
@Deprecated. Самодельный
интероп оказался дорогим

Выводы

01

Шарьте логику: **~90%** Kotlin — в commonMain

04

Держите дорогое за флагом: WASM и macOS-native **выключены**

02

Не шарьте UI на iOS насильно. SwiftUI поверх общего домена — это **осознанный выбор**



05

Самодельный интероп может оказаться **недолговечным**

03

Развивайте инфраструктуру **раньше фич**

06

Используйте **готовые** инструменты для достижения результата

Общее 90%

- Навигация (Decompose)
- Стейт (MVIKotlin)
- Репозитории, UseCase, сервисы
- БД, Ktor, WebSocket

VS

Нативное 10%

- UI (iOS)
- Push-инфраструктура
- Системные сервисы

**Всё, что ниже кнопки, — общее.
Кнопка и всё, что трогает
системные ресурсы, — нативное**

**Логику мы
написали один раз.**

**Просто запустили
её три раза.**

Александр · @a_mnshkv

Максим · @TKOTW

