Modern MVI и MVVM+
со всех сторон в 2023

Управляем состоянием приложения,
не привлекая внимания санитаров

Содержит следы BLoC, Redux, TEA/Elm, MVU, SAM, FSM

Linen

Артём Шендрик
Lead Android Engineer

Обо мне

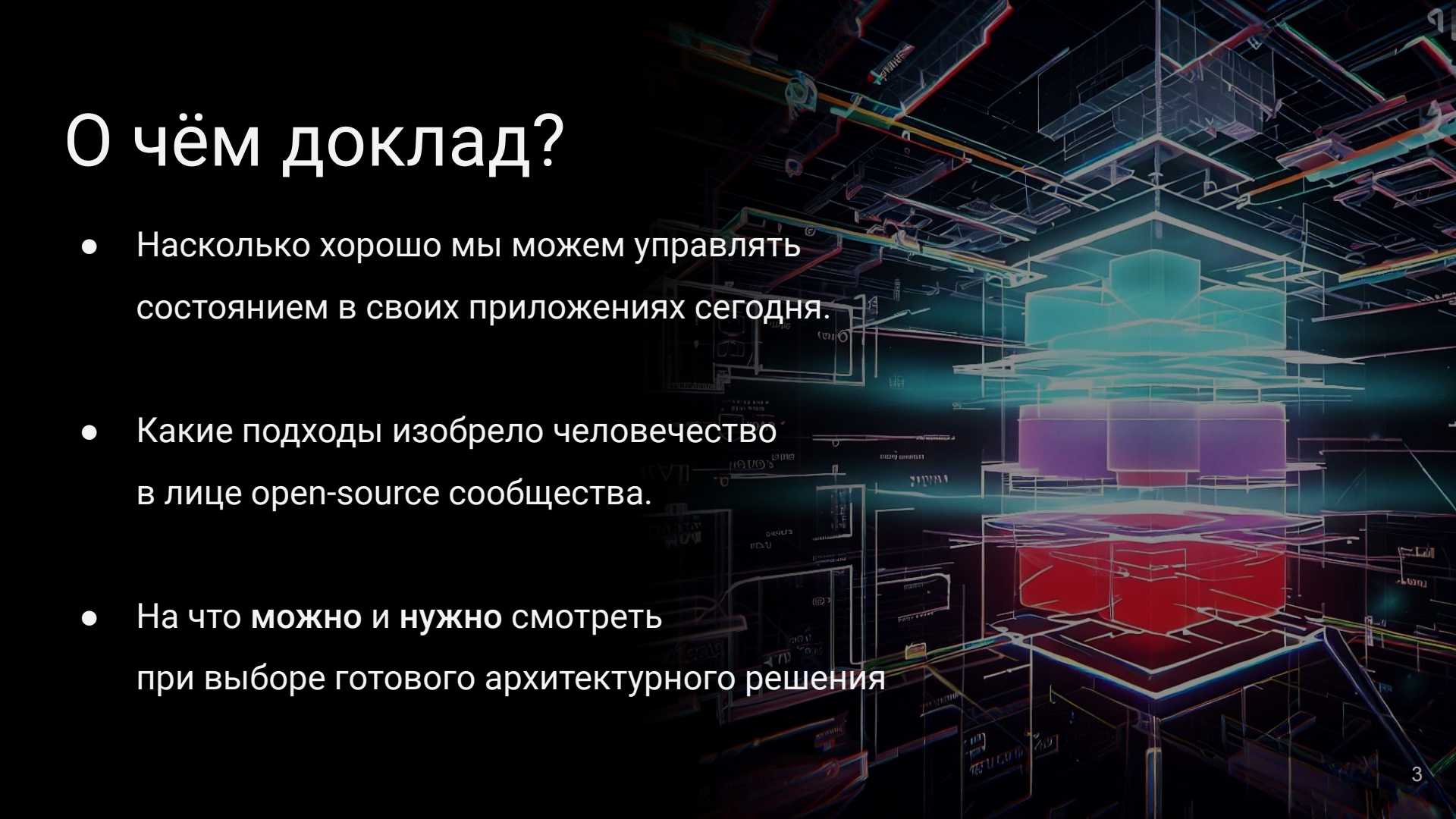
- Лид Android-направления в Linen Wallet
- Энтузиаст Kotlin Multiplatform
- Девять лет разработки на Kotlin
- Больше 13 лет разработки в разных доменах, на нескольких платформах
- Автор Fluxo *['fluksu]*, state-management библиотеки для KMP/KMM

О чём доклад?

- Насколько хорошо мы можем управлять состоянием в своих приложениях сегодня.

- Какие подходы изобрело человечество в лице open-source сообщества.

- На что **можно** и **нужно** смотреть при выборе готового архитектурного решения



Исследовано более **60** библиотек

Fluxo
Ballast
Orbit MVI
VisualFSM (Kontur)
FlowMVI (Respawn)
MVIKotlin
MviCore (Badoo)
Kotlin Bloc
Elmslie
Uniflow
FlowRedux (Freeletics)
Mobius.kt
Redux-Kotlin
Mavericks, ex. MvRx (AirBnb)
Reduktor (Ula)
CommonStateMachine
genaku/Reduce (Knot Kt)

StateMachine (Tinder)
KStateMachine
KSFM
kfin-state-machine
Workflow (Square)
Mad State-Machine (Freeletics)
KViewModel (Adeo)
oolong
Formula (Instacart)
Mobius (Spotify)
Kaskade
tunjid/Mutator
bamlab/kstate
Flywheel
Comachine
haroldadmin/Vector
Chassis

CoRed
RxRedux
gustavkarlsson/krate
bansa
sergejsha/knot
Grox (Groupon)
Mosby by Hannes Dorfmann
HAL by adrielcafe
Dalek by adrielcafe
Roxie
mboudraa/flow
Stateful4k
milosmns/kssm
pardom-zz/redux-kotlin
Statelin
Sesame Loop
Reduks

KotlinBloc
RxReduxK
Puerh
fededri/Arch
Quantum
MVFlow
FSMgasm
KmpMvi
myrealtrip/box
StateReducerFlow
Cyklic
Kontent
trafi/states

docs.google.com/spreadsheets/d/1gbwXU5Vp9QGvph1rLu0hv90KqrHNhkZyKTxk_I_yBJk

По 18 библиотекам собрана наиболее полная информация для детального сравнения

Fluxo
Ballast
Orbit MVI
VisualFSM (Kontur)
FlowMVI (Respawn)
MVIKotlin
MviCore (Badoo)
Kotlin Bloc
Elmslie

Uniflow
FlowRedux (Freeletics)
Mobius.kt (spotify/Mobius fork)
Redux-Kotlin
Mavericks, ex. MvRx (AirBnb)
Reduktor (Ula)
CommonStateMachine
genaku/Reduce (Knot Kt)
StateMachine (Tinder)

docs.google.com/spreadsheets/d/1gbwXU5Vp9QGvph1rLu0hv90KqrHNhkZyKTxk_I_yBJk

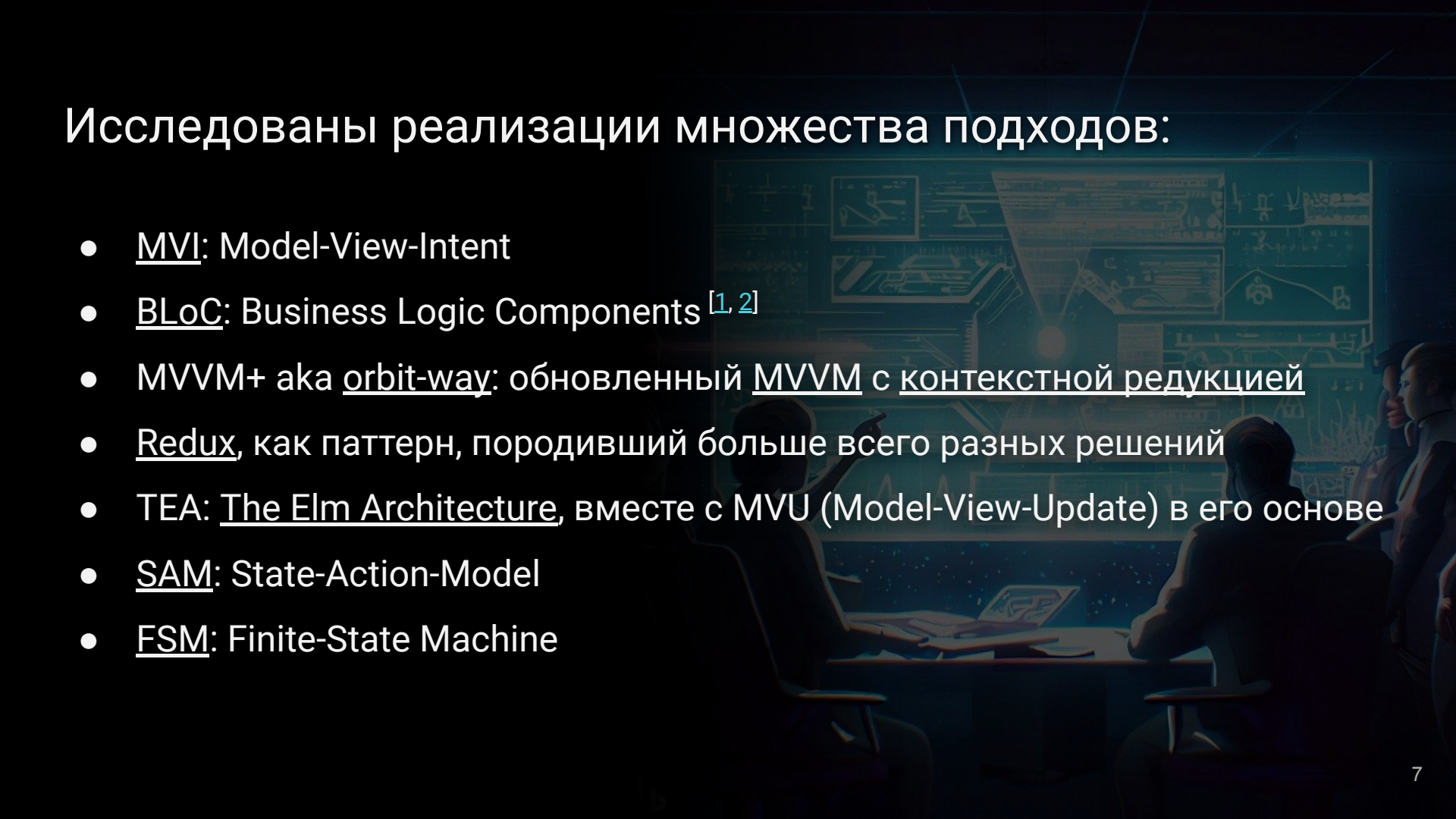
Для 13 библиотек написаны и развиваются бенчмарки

- Fluxo
- Ballast
- Orbit MVI
- VisualFSM (Kontur)
- FlowMVI (Respawn)
- MVIKotlin
- MviCore (Badoo)
- FlowRedux (Freeletics)
- Mobius.kt (spotify/Mobius fork)
- Redux-Kotlin
- Reduktor (Ula)
- genaku/Reduce (Knot Kt)
- StateMachine (Tinder)

github.com/fluxo-kt/fluxo/actions/workflows/benchmark.yml (GitHub CI)

github.com/fluxo-kt/fluxo/actions/runs/4907652284 (GitHub CI)

Исследованы реализации множества подходов:



- MVI: Model-View-Intent
- BLoC: Business Logic Components [1, 2]
- MVVM+ aka orbit-way: обновленный MVVM с контекстной редукцией
- Redux, как паттерн, породивший больше всего разных решений
- TEA: The Elm Architecture, вместе с MVU (Model-View-Update) в его основе
- SAM: State-Action-Model
- FSM: Finite-State Machine

Также:

- Найден ряд интересных решений и идей
- Обобщены подходы и функционал
- Получены инсайты на тему границ возможного и полезного
- Выдвинута и проверена гипотеза, из-за которой был создан Fluxo

Предупреждение:

- Сравнение ограничено реализациями написанными в основном на Kotlin.
- Сами принципы кроссплатформенные и не ограничены мобильными приложениями.
- Часть критериев субъективны, например, поддержка мультиплатформы или корутины в основе оцениваются как плюс.
- Но вы можете настроить итоговую оценку под себя, согласно своим критериям.

docs.google.com/spreadsheets/d/1gbwXU5Vp9QGvph1rLu0hv90KqrHNhkZyKTxk_I_yBJk

При чём здесь вообще state management?

Когда-то мы больше всего беспокоились
о разделении ответственности. Скажем с:

- MVC: Model-View-Controller
- MVP: Model-View-Presenter

Это важно и сейчас, но чем дальше,
тем больше мы думаем о реактивности,
однонаправленных потоках данных
и иммутабельности этих данных.

developer.android.com/topic/architecture

MVVM Architectural Pattern

- Текущая рекомендация для архитектуры Android приложений
- Вместе с разделением кода по слоям и фичам, обеспечивает:
  - хорошее разделение ответственностей
  - удобное переиспользование кода (DRY)
  - простоту покрытия кода тестами
  - корректную обработку особенностей жизненного цикла компонентов Android

developer.android.com/topic/architecture

MVVM Architectural Pattern

- Immutable data

```kotlin
/**
 * UiState for the Add/Edit screen
 */
data class AddEditTaskUiState(
    val title: String = "",
    val description: String = "",
    val isTaskCompleted: Boolean = false,
    val isLoading: Boolean = false,
    val userMessage: Int? = null,
    val isTaskSaved: Boolean = false
)
```

github.com/android/architecture-samples/../todoapp/addedittask/AddEditTaskViewModel.kt

MVVM Architectural Pattern

- Immutable data
- Single source of truth
- Reactive state (StateFlow)

```
// A MutableStateFlow needs to be created in this ViewModel. The source of truth of the current
// editable Task is the ViewModel, we need to mutate the UI state directly in methods such as
// `updateTitle` or `updateDescription`
private val _uiState = MutableStateFlow(AddEditTaskUiState())
val uiState: StateFlow<AddEditTaskUiState> = _uiState.asStateFlow()
```

github.com/android/architecture-samples/../todoapp/addedittask/AddEditTaskViewModel.kt

MVVM Architectural Pattern

- Immutable data
- Single source of truth
- Reactive state
- Unidirectional data flow
- Каждое действие выделено
  - Легко читать, изменять и тестировать
  - Контекстная редукция
- Thread-safe atomic update

```
fun snackbarMessageShown() {
    _uiState.update {
        it.copy(userMessage = null)
    }
}

fun updateTitle(newTitle: String) {
    _uiState.update {
        it.copy(title = newTitle)
    }
}

fun updateDescription(newDescription: String) {
    _uiState.update {
        it.copy(description = newDescription)
    }
}
```

github.com/android/architecture-samples/../todoapp/addedittask/AddEditTaskViewModel.kt

MVVM — Минусы

- Все нужные ограничения архитектуры только в уме. Нужно каждый раз следить за корректностью.
- Корректность запуска фоновых долгих процессов тоже ручная, легко ошибиться.
- Хорошо подходит для простых моделей, комплексные экраны с множественными источниками связанных данных очень легко превратить в кашу.
- При обновлении данных асинхронно и/или независимо от UI всё становится на порядок сложнее.

```
data class AddEditTaskUiState(
    val title: String = "",
    val description: String = "",
    val isTaskCompleted: Boolean = false,
    val isLoading: Boolean = false,
    val userMessage: Int? = null,
    val isTaskSaved: Boolean = false
)

/**
 * ViewModel for the Add/Edit screen.
 */
@HiltViewModel
class AddEditTaskViewModel @Inject constructor(
    private val taskRepository: TaskRepository,
    savedStateHandle: SavedStateHandle
) : ViewModel() {

    private val taskId: String? = savedStateHandle[TodoDestinationsArgs.TASK_ID_ARG]

    // A MutableStateFlow needs to be created in this ViewModel. The source of truth of the current
    // editable Task is the ViewModel, we need to mutate the UI state directly in methods such as
    // `updateTitle` or `updateDescription`
    private val _uiState = MutableStateFlow(AddEditTaskUiState())
    val uiState: StateFlow<AddEditTaskUiState> = _uiState.asStateFlow()

    init {
        if (taskId != null) {
            loadTask(taskId)
        }
    }

    // Called when clicking on fab.
    fun saveTask() {
        if (uiState.value.title.isEmpty() || uiState.value.description.isEmpty()) {
            _uiState.update {
                it.copy(userMessage = R.string.empty_task_message)
            }
            return
        }

        if (taskId == null) {
            createNewTask()
        } else {
            updateTask()
        }
    }

    fun snackbarMessageShown() {
        _uiState.update {
            it.copy(userMessage = null)
        }
    }

    fun updateTitle(newTitle: String) {
        _uiState.update {
            it.copy(title = newTitle)
        }
    }

    fun updateDescription(newDescription: String) {
        _uiState.update {
            it.copy(description = newDescription)
        }
    }
```

github.com/android/architecture-samples/../todoapp/addedittask/AddEditTaskViewModel.kt

BLoC: Business Logic Components

MVI: Model-View-Intent

- Внешние команды кодируются явно, в виде класса или объекта
- Появляется центральный узел, через который проходит поток команд
- Узлы удобно соединять друг с другом
- Команды легко тестировать, логировать, повторять (time-travel!)

```
internal interface DetailsStore : Store<Intent, State, Label> {

    // Serializable only for exporting events in Time Travel, no need otherwise.
    sealed class Intent : JvmSerializable {
        data class SetText(val text: String) : Intent()
        object ToggleDone : Intent()
        object Delete : Intent()
    }

    data class State(
        val data: TodoItem.Data? = null,
        val isFinished: Boolean = false,
    ) : JvmSerializable // Serializable only for exporting events in Time Travel

    // Serializable only for exporting events in Time Travel, no need otherwise.
    sealed class Label : JvmSerializable {
        data class Changed(val id: String, val data: TodoItem.Data) : Label()
        data class Deleted(val id: String) : Label()
    }
}
```

github.com/arkivanov/MVIKotlin/../sample/coroutines/shared/details/store/DetailsStore.kt

BLoC: Business Logic Components

MVI: Model-View-Intent

- Благодаря потоку команд мы можем
управляемо на них реагировать.
- Команды/Интенты легко отправить в фоновый поток или добавить debounce.
- Контейнер обеспечивает диспетчеризацию и безопасное обновление состояния.

```
executorFactory = coroutineExecutorFactory(mainContext) {
    onAction<Unit> {
        launch {
            val item: TodoItem? = withContext(ioContext) { database.get(itemId) }
            dispatch(item?.data?.let(Msg::Loaded) ?: Msg.Finished)
        }
    }

    onIntent<Intent.SetText> {
        dispatch(Msg.TextChanged(it.text))
        save()
    }

    onIntent<Intent.ToggleDone> {
        dispatch(Msg.DoneToggled)
        save()
    }

    onIntent<Intent.Delete> {
        publish(Label.Deleted(itemId))

        launch {
            withContext(ioContext) { database.delete(itemId) }
            dispatch(Msg.Finished)
        }
    }
},
```

github.com/arkivanov/MVIKotlin/../sample/coroutines/shared/details/store/DetailsStoreFactory.kt

MVI, BLoC — Минусы

- **Редьюсер** обрабатывает все возможные интенты. Легко может сильно разрастись.
- Сложнее понять, что происходит, когда отправляется интент. Нужно искать и читать места использования объекта, переход в IDE по методу не возможен.
- Увеличение когнитивной нагрузки для написания и поддержки кода. Требуется знание и соблюдение принципов паттерна.
- Зачастую требуется «перестраивать мышление».

```
reducer = { msg ->
    when (msg) {
        is Msg.Loaded -> copy(data = msg.data)
        is Msg.Finished -> copy(isFinished = true)
        is Msg.TextChanged -> copy(data = data?.copy(text =
        is Msg.DoneToggled -> copy(data = data?.copy(isDone
    }
},
```

- A case against the MVI architecture pattern (Apr 2021)
- MVI - The Good, the Bad, and the Ugly (Jul 2019)

MVVM+ aka Orbit-way

- Интент становится «**лямбдой**», анонимной функцией, внутри которой и происходит обработка команды, запуск долгих процессов и прочих «сайд-эффектов».

```
override val container = container<CalculatorState, CalculatorSideEffect>(CalculatorState())

fun add(number: Int) = intent {
    postSideEffect(CalculatorSideEffect.Toast("Adding $number to ${state.total}!"))

    reduce {
        state.copy(total = state.total + number)
    }
}
```

A note on styles of MVI, what is MVVM+ (Aug, 2021)

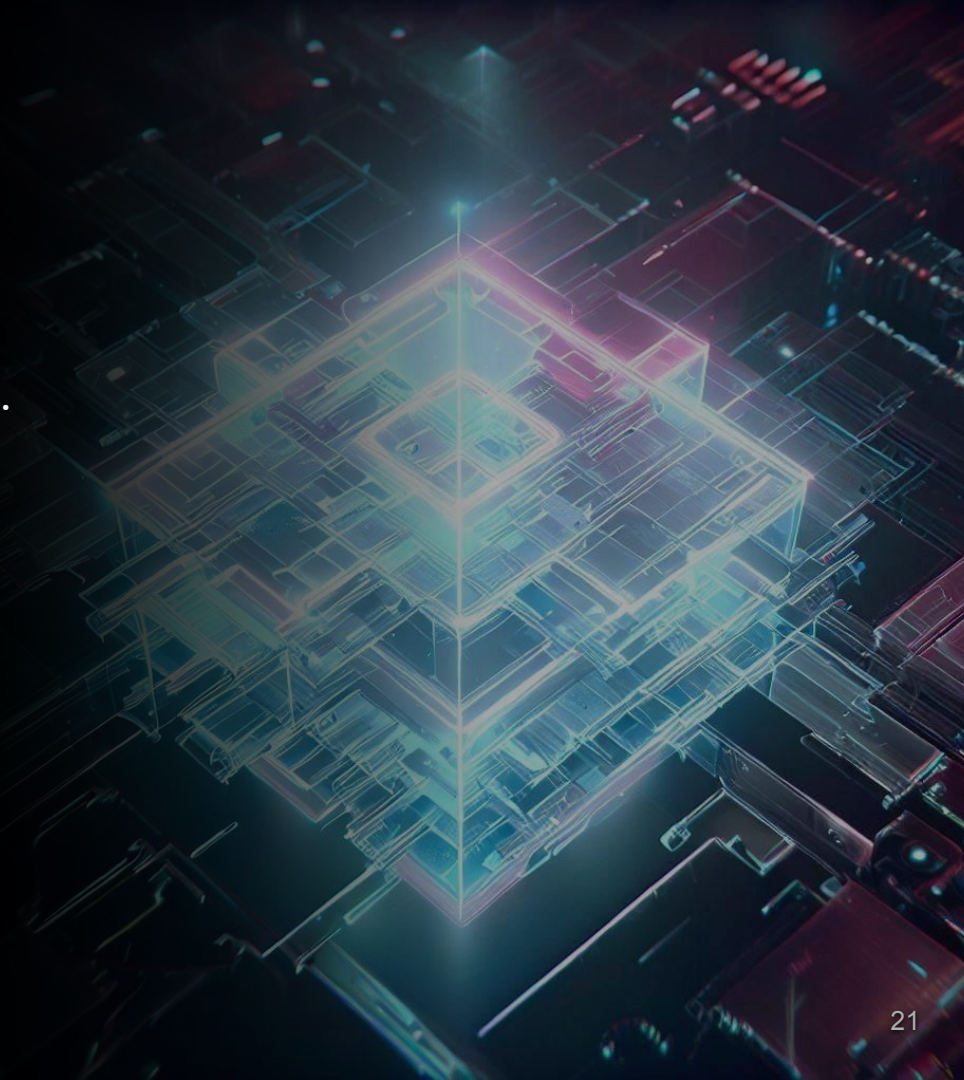
MVVM+ или Orbit-way

- Интент становится «**лямбдой**», анонимной функцией, внутри которой и происходит обработка команды, запуск долгих процессов и прочих «сайд-эффектов».
- Код снова читается линейно, более интуитивно. Изменение состояния происходит в контексте вызова (в отличие от группировки всего в один большой «редьюсер»).
- Легко делать гибкое и выразительное API, сохраняя безопасность всех действий.

A note on styles of MVI, what is MVVM+ (Aug, 2021)

MVVM+, какой ценой?

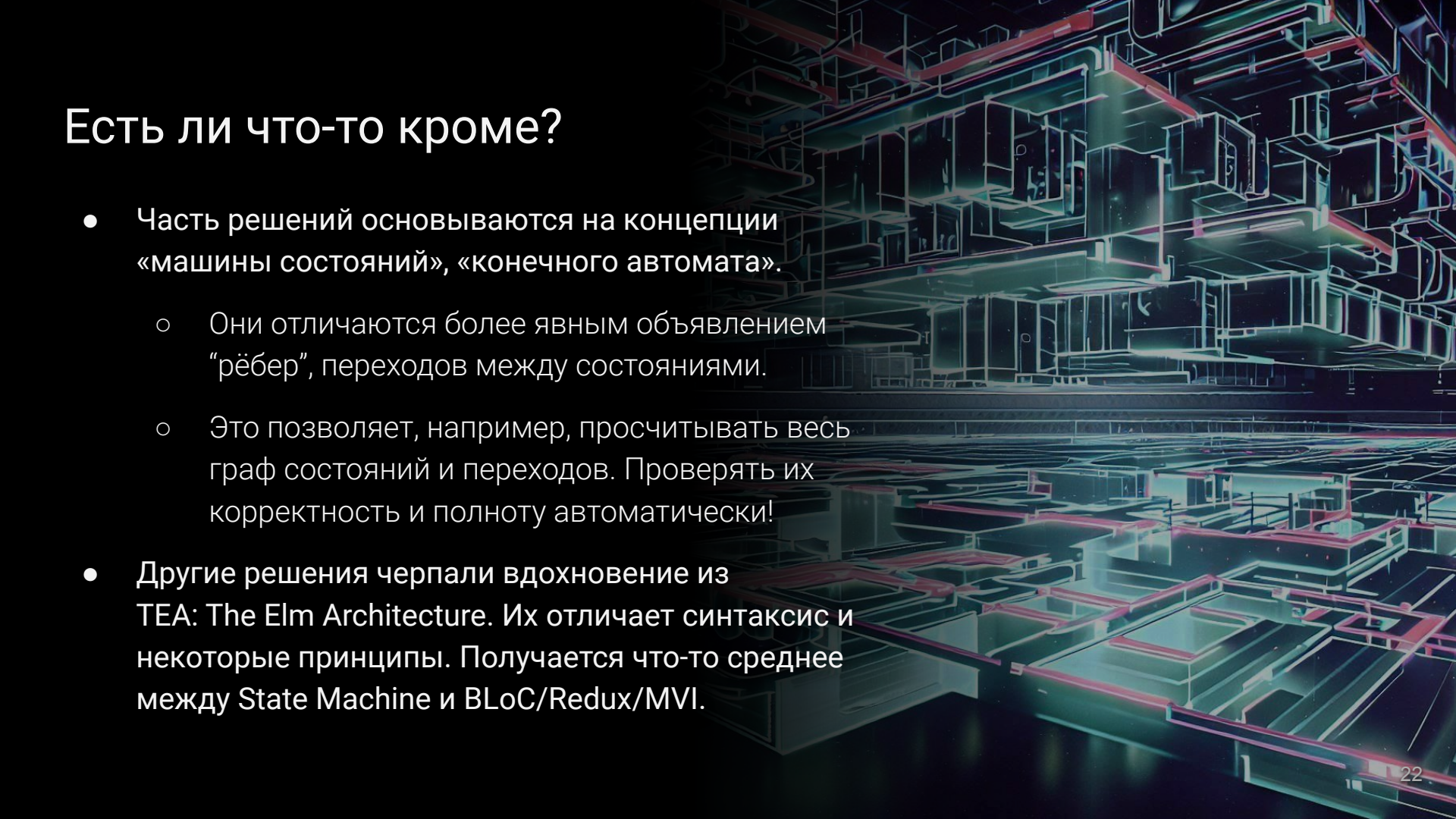
- На каждый интент с аргументами создаётся новый инстанс лямбды.
- Теряем удобное и информативное логирование интентов.
- Остаёмся без time-travel отладки.
- Уже совсем не похоже на Redux :)



A note on styles of MVI, what is MVVM+ (Aug, 2021)

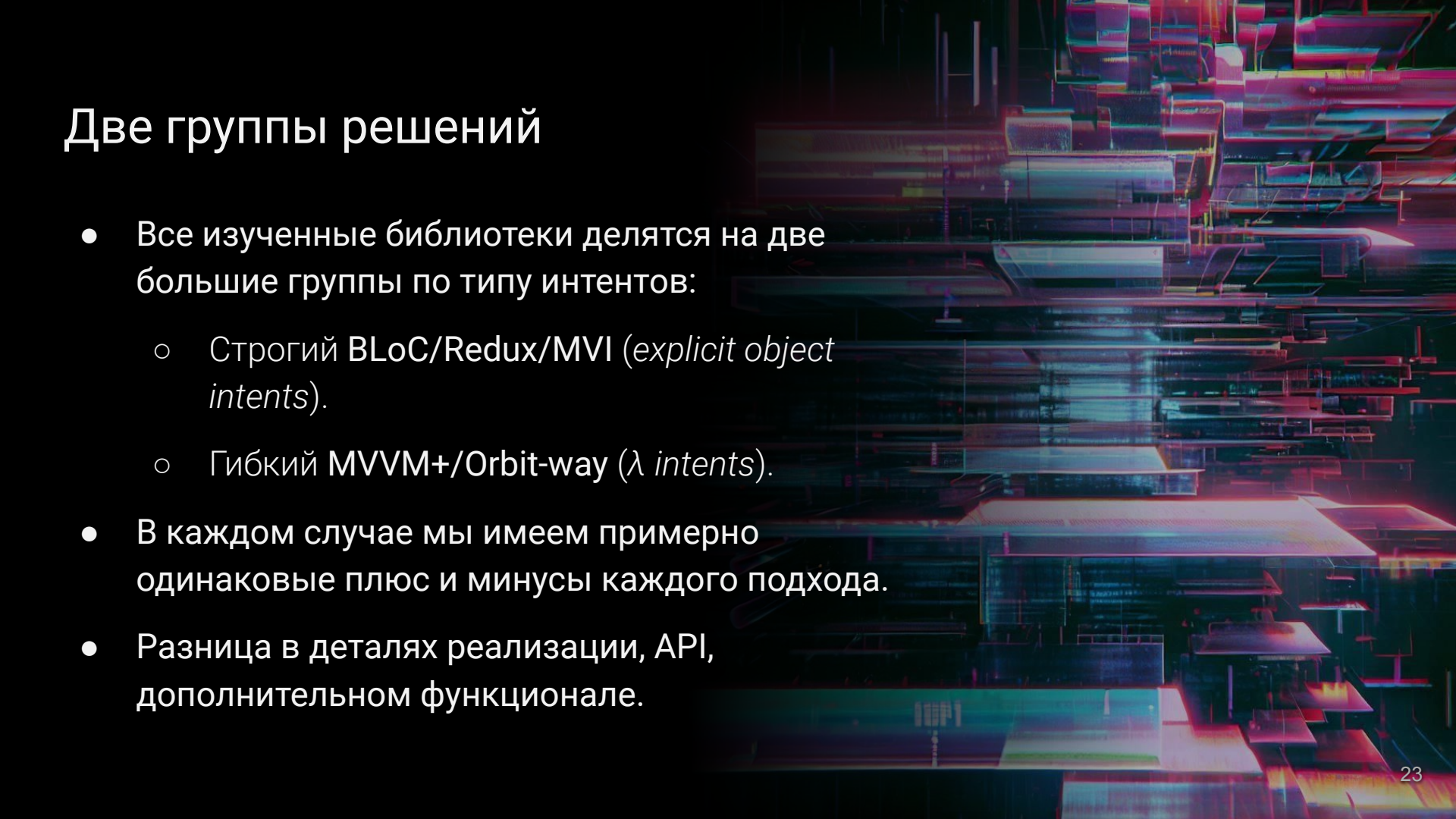
Есть ли что-то кроме?

- Часть решений основываются на концепции «машины состояний», «конечного автомата».
  - Они отличаются более явным объявлением “рёбер”, переходов между состояниями.
  - Это позволяет, например, просчитывать весь граф состояний и переходов. Проверять их корректность и полноту автоматически!
- Другие решения черпали вдохновение из TEA: The Elm Architecture. Их отличает синтаксис и некоторые принципы. Получается что-то среднее между State Machine и BLoC/Redux/MVI.



Две группы решений

- Все изученные библиотеки делятся на две большие группы по типу интентов:
  - Строгий **BLoC/Redux/MVI** (*explicit object intents*).
  - Гибкий **MVVM+/Orbit-way** (*λ intents*).
- В каждом случае мы имеем примерно одинаковые плюс и минусы каждого подхода.
- Разница в деталях реализации, API, дополнительном функционале.



Гипотеза Fluxo

- Должна быть возможность объединить все сильные стороны строгого **BLoC/Redux/MVI** с гибкостью и удобством чтения **MVVM+/Orbit-way**, собрав преимущества обоих подходов!

Гипотеза Fluxo

- Должна быть возможность объединить все сильные стороны строгого **BLoC/Redux/MVI** с гибкостью и удобством чтения **MVVM+/Orbit-way**, собрав преимущества обоих подходов!
- Идеальное логирование удалось сделать для JVM/Android платформ.
- Можно выбирать, какой подход использовать для конкретного контейнера состояния и многое другое.

```
private class AddModel {

    private val container = container(initialState = 0)
    val state: StateFlow<Int> = container

    fun add(number: Int) = container.intent {
        updateState {
            it + number
        }
    }

    // pass "12" as arg. intent logged as:
    // add(number=12)
}
```

github.com/fluxo-kt/fluxo

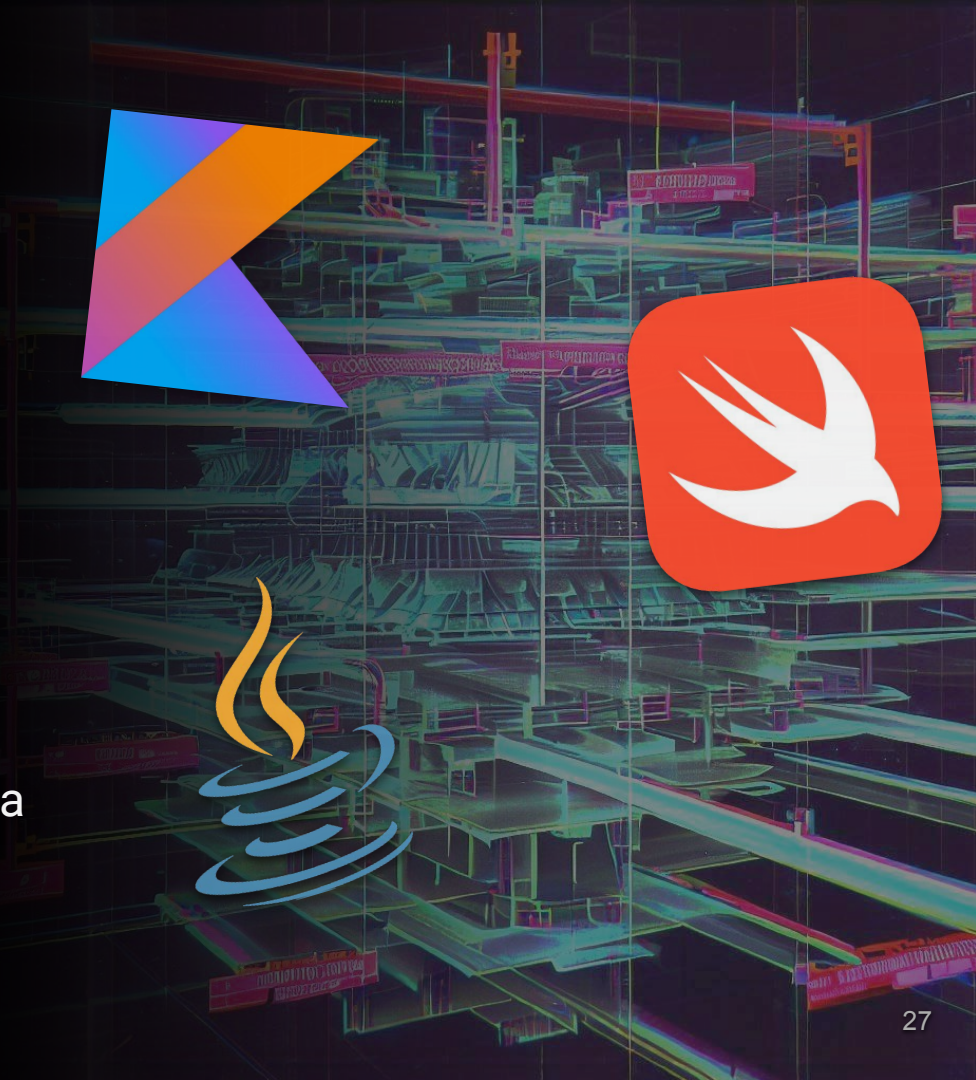
Вернёмся к исследованию

- Библиотек многие десятки!
- У всех разные функции и гора нюансов в реализациях.
- Хотел хорошо разобраться в этой теме и теперь могу помочь вам выбрать себе оптимальное решение!
- На что стоит обращать внимание?
    - В таблице собраны десятки важных и не очень критериев. Сейчас о ключевых.

docs.google.com/spreadsheets/d/1gbwXU5Vp9QGvph1rLu0hv90KqrHNhkZyKTxk_I_yBJk

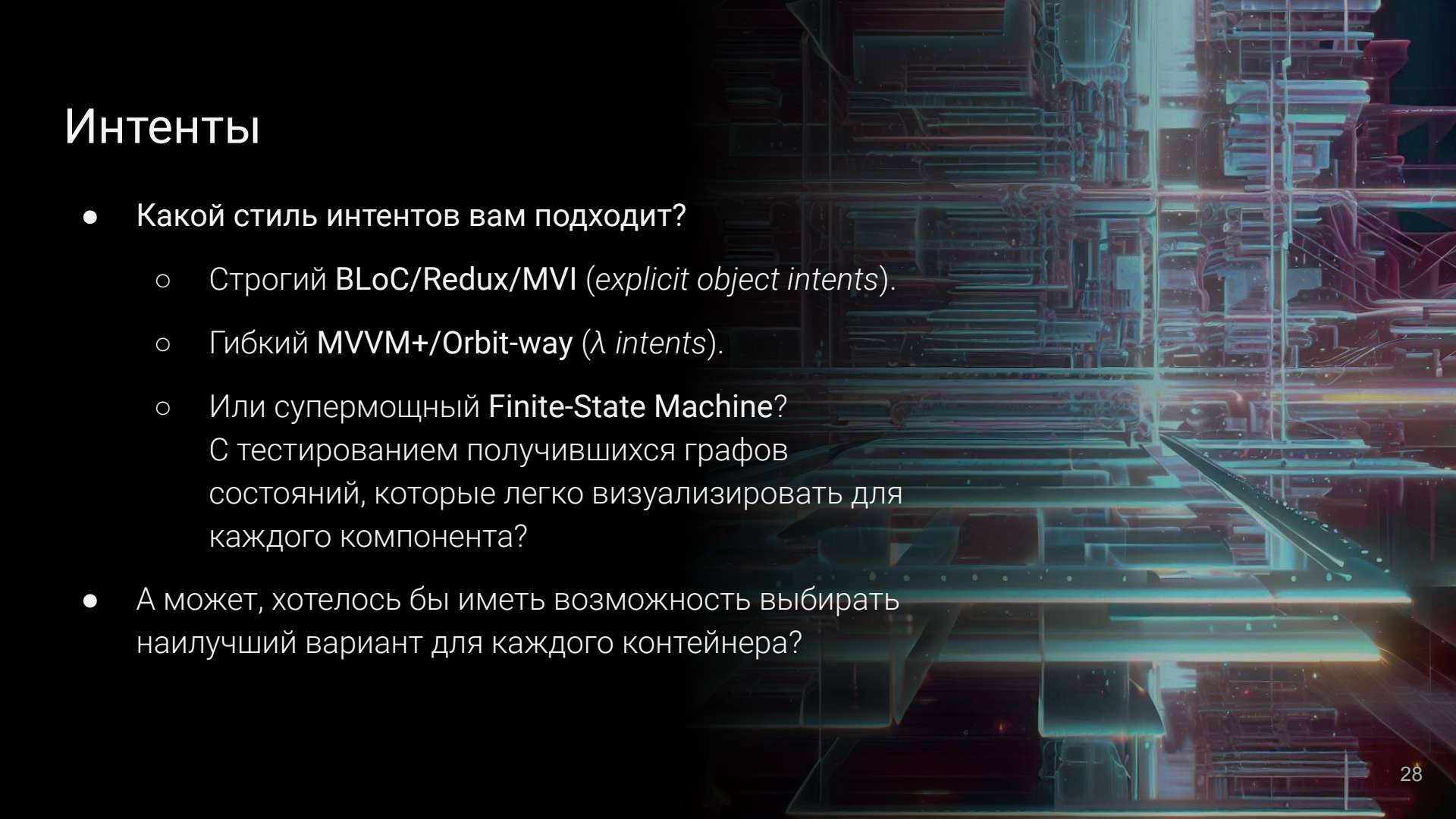
Платформа

- Язык библиотеки
- Поддержка Multiplatform
- Основа реактивности
  - Coroutines, RxJava, LiveData
- Поддержка Compose
- Активная разработка и поддержка
- Документация и сообщество



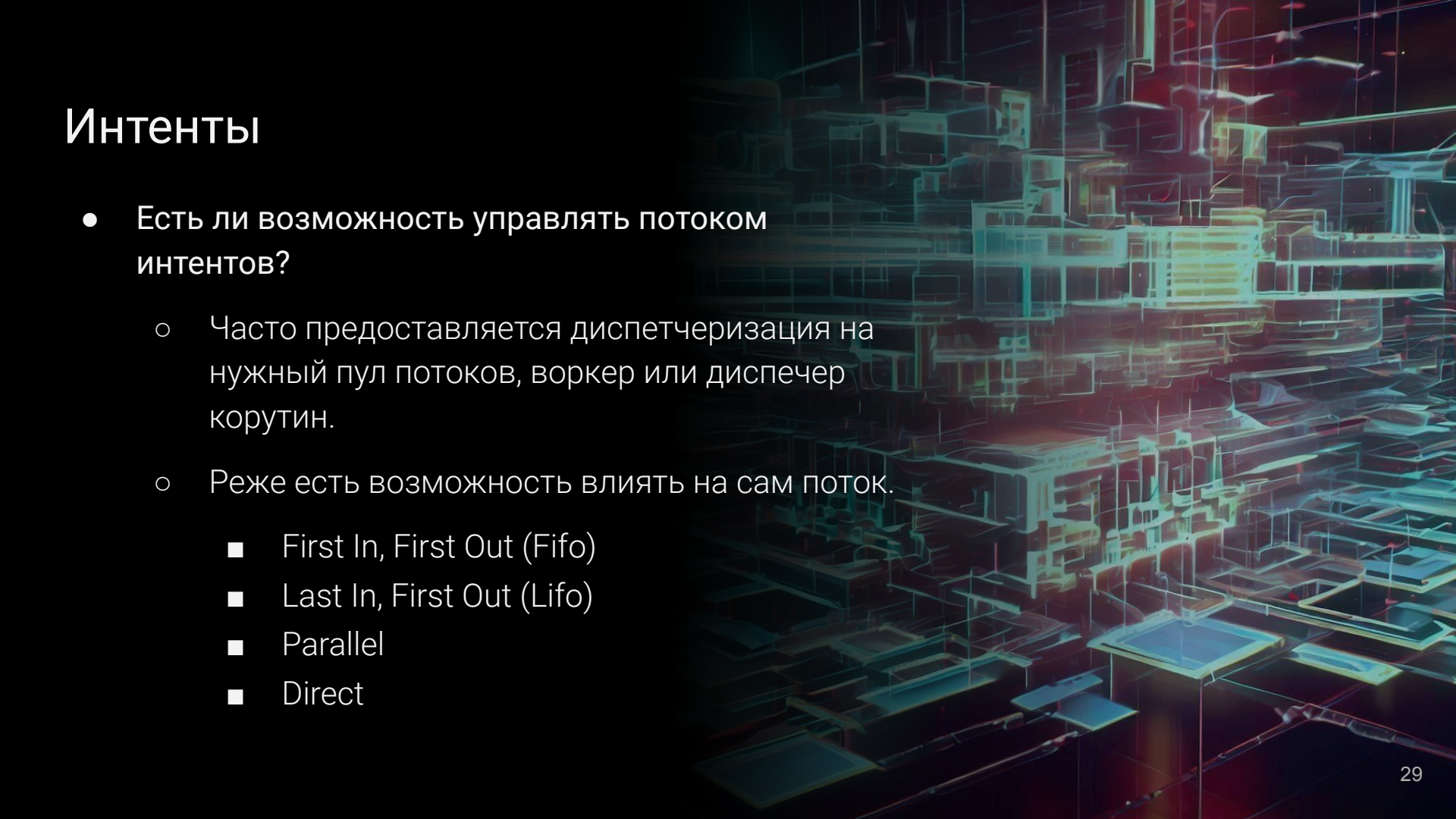
Интенты

- Какой стиль интентов вам подходит?
  - Строгий **BLoC/Redux/MVI** (*explicit object intents*).
  - Гибкий **MVVM+/Orbit-way** (*λ intents*).
  - Или супермощный **Finite-State Machine**?
    С тестированием получившихся графов состояний, которые легко визуализировать для каждого компонента?
- А может, хотелось бы иметь возможность выбирать наилучший вариант для каждого контейнера?



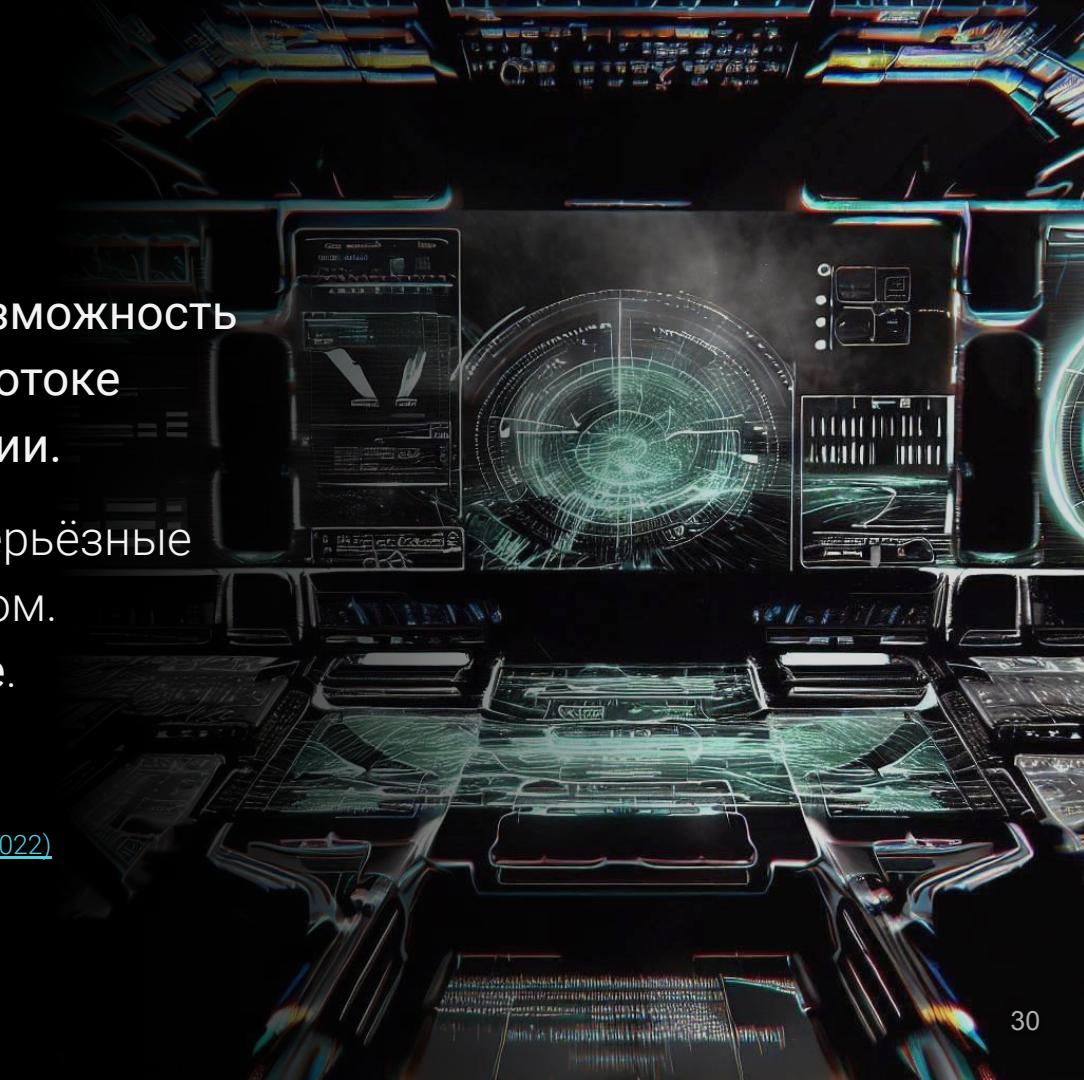
Интенты

- Есть ли возможность управлять потоком интентов?
  - Часто предоставляется диспетчеризация на нужный пул потоков, воркер или диспечер корутин.
  - Реже есть возможность влиять на сам поток.
    - First In, First Out (Fifo)
    - Last In, First Out (Lifo)
    - Parallel
    - Direct



Интенты

- Для Android важно иметь возможность выполнять интенты в Main-потоке сразу же, без диспетчеризации.
    - Иначе можно поймать серьёзные проблемы с **Text** виджетом. Как минимум, в **Compose**.



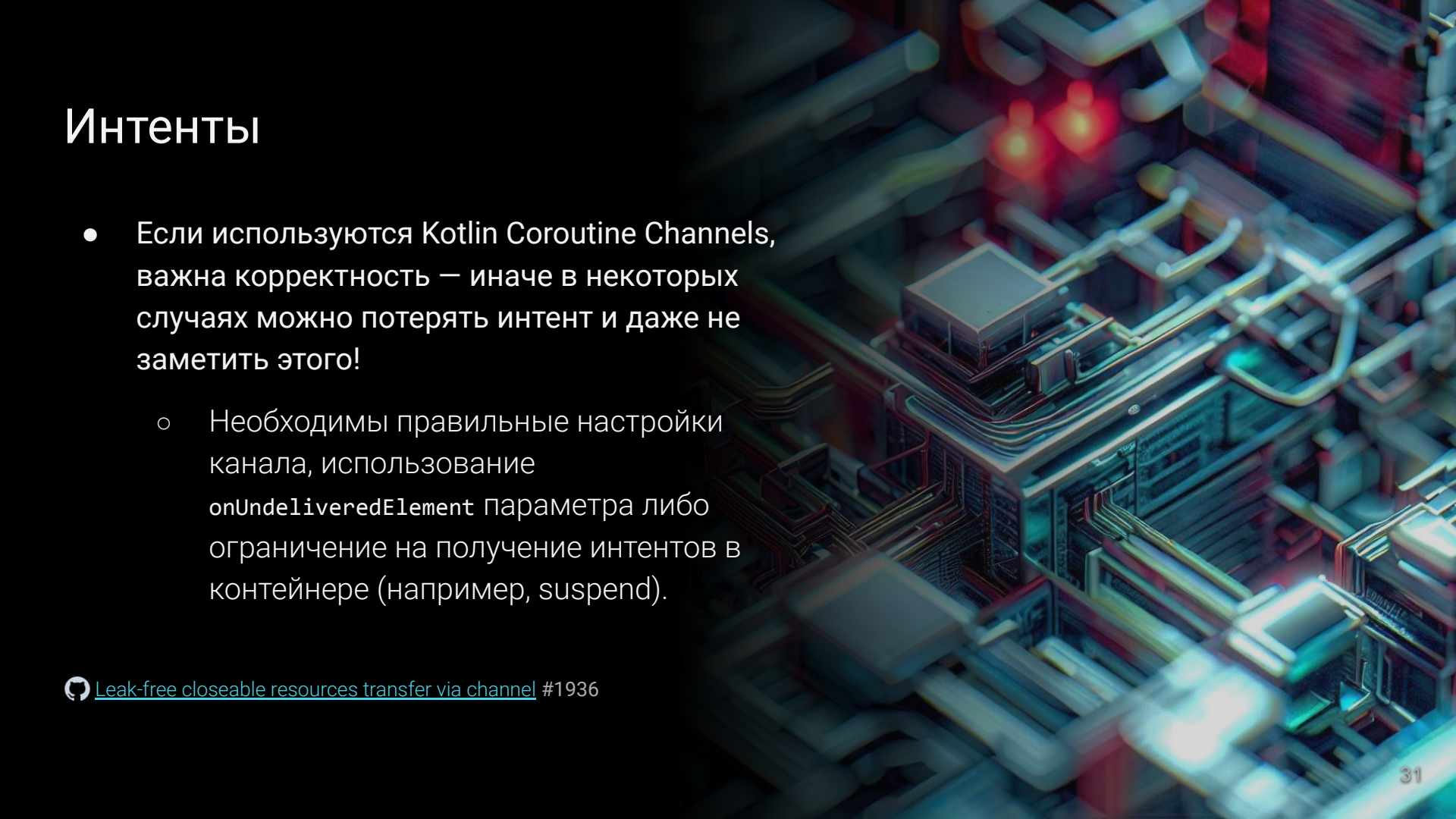
Effective state management for TextField in Compose (Sep, 2022)

github.com/cashapp/molecule/issues/63

github.com/orbit-mvi/orbit-mvi/issues/82

Интенты

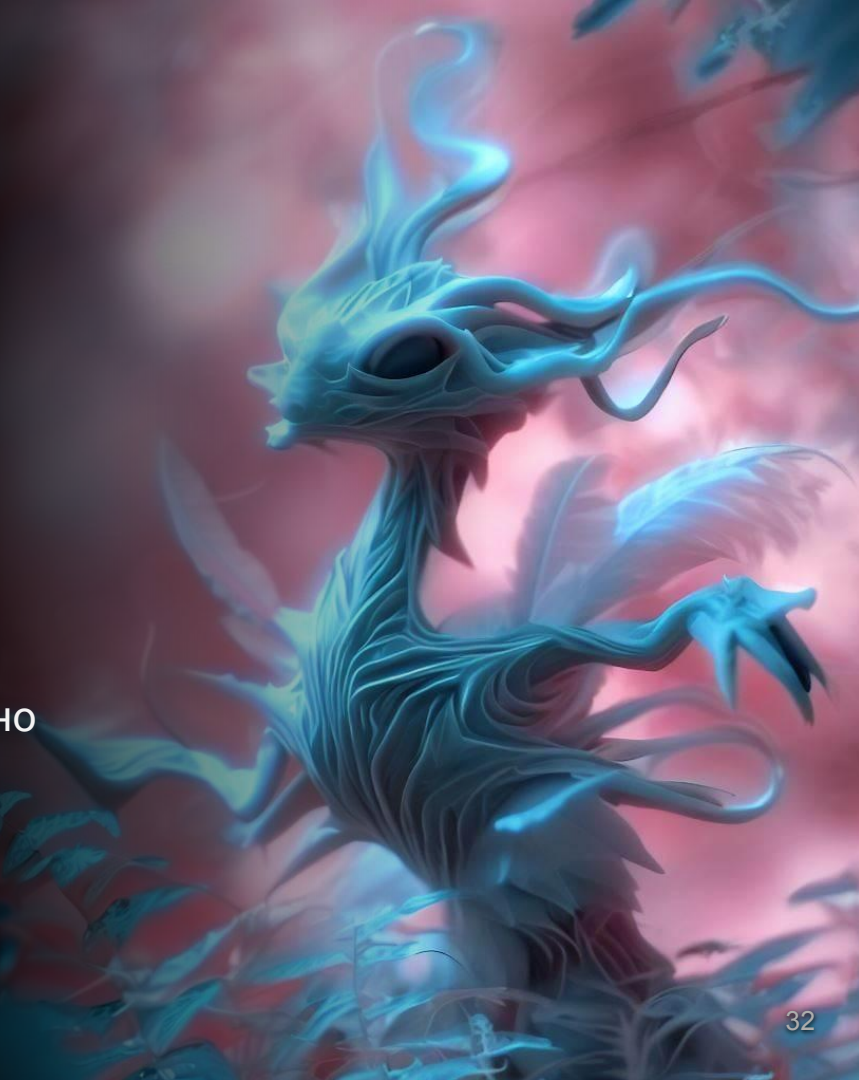
- Если используются Kotlin Coroutine Channels, важна корректность — иначе в некоторых случаях можно потерять интент и даже не заметить этого!
    - Необходимы правильные настройки канала, использование `onUndeliveredElement` параметра либо ограничение на получение интентов в контейнере (например, suspend).



Leak-free closeable resources transfer via channel #1936

Сайд-эффекты

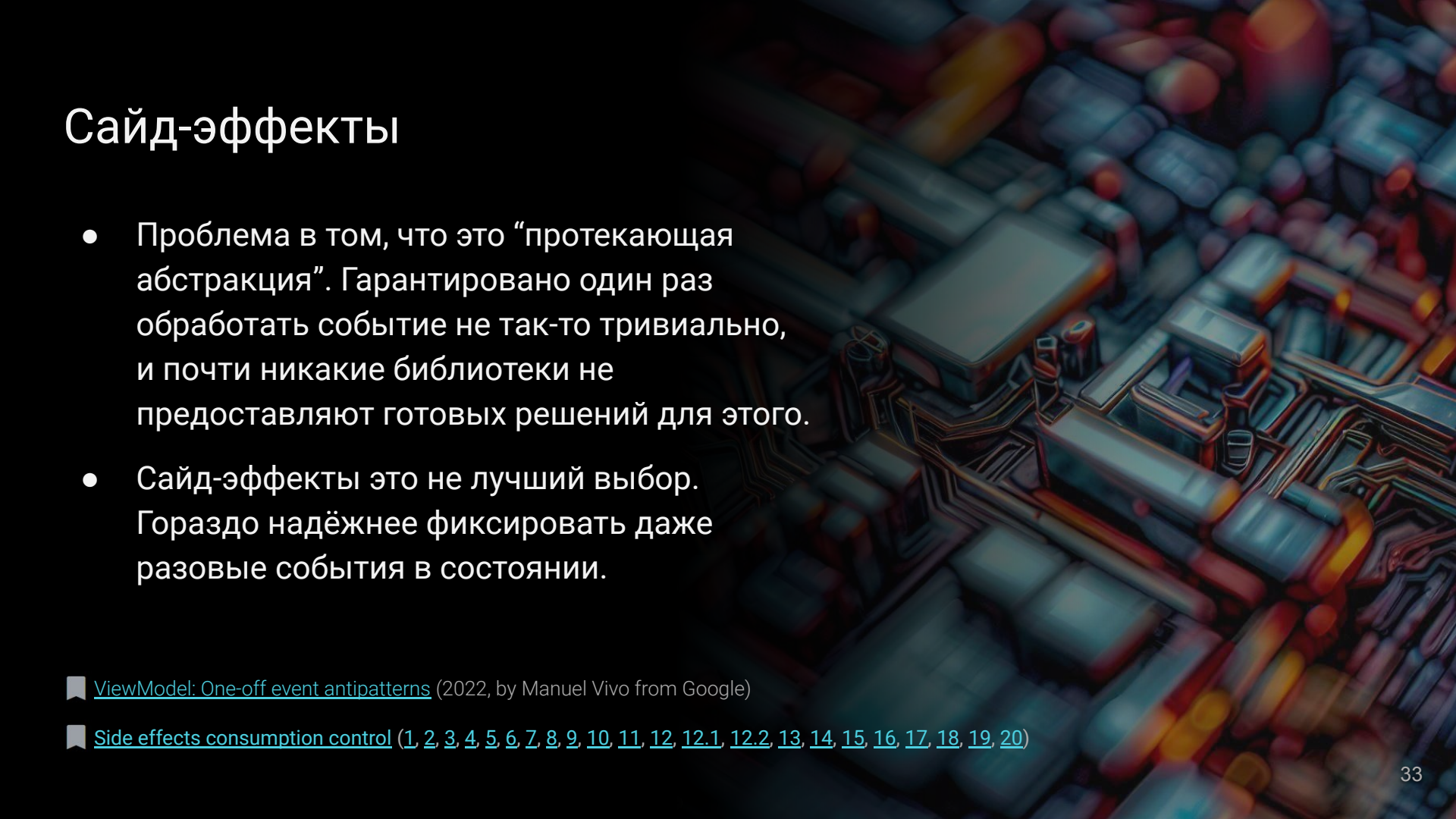
- Во многих реализациях кроме потока состояний есть ещё отдельный поток «событий», т.н. **сайд-эффектов**.
- Обратите внимание, что использование побочных эффектов обычно считается антипаттерном!
- Но иногда это может быть полезно, особенно при рефакторинге старой кодовой базы.



ViewModel: One-off event antipatterns (2022, by Manuel Vivo from Google)

Google Guide to app architecture, UI events > Other use cases > Note (Apr 2023)

Сайд-эффекты



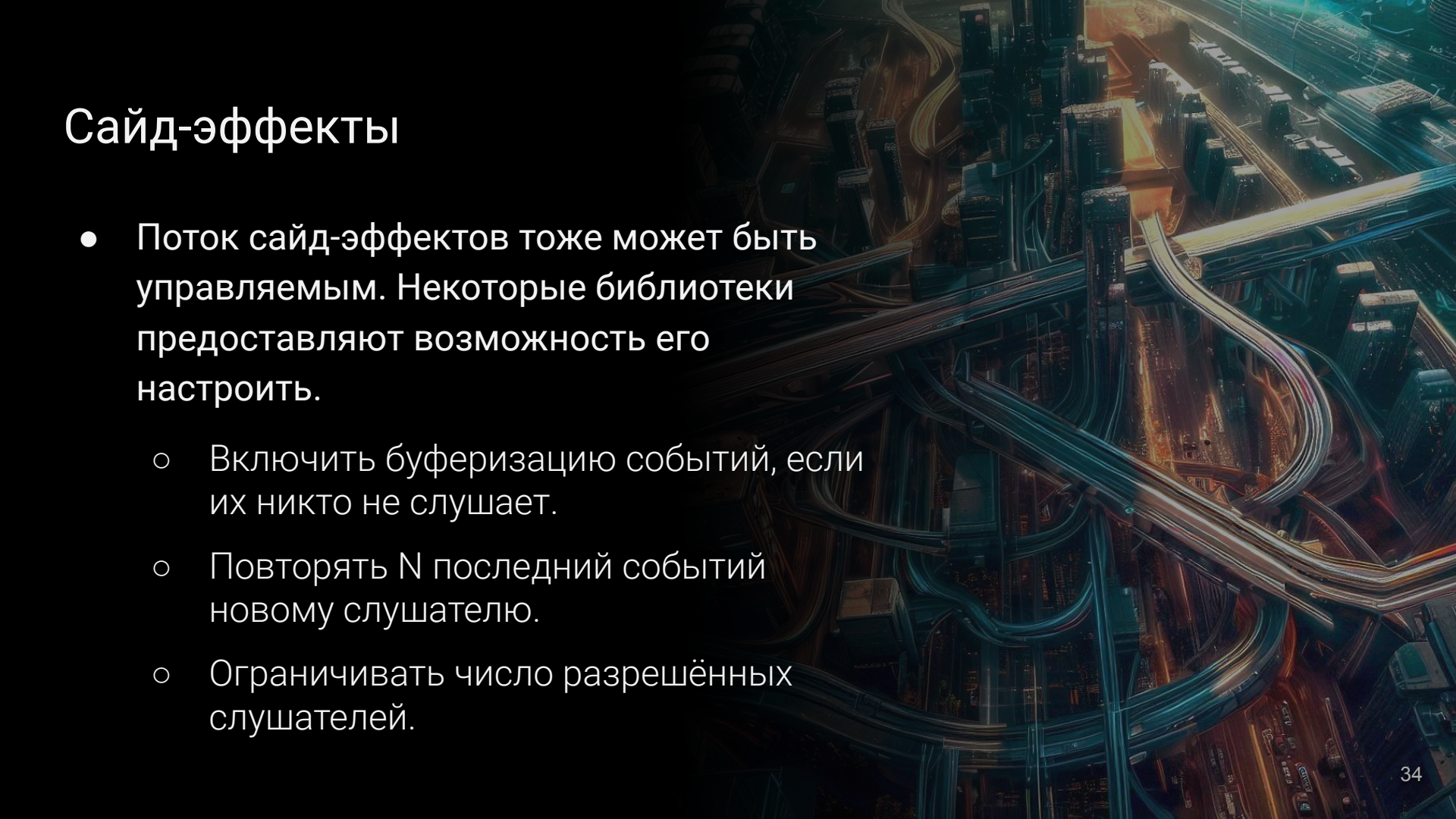
- Проблема в том, что это “протекающая абстракция”. Гарантировано один раз обработать событие не так-то тривиально, и почти никакие библиотеки не предоставляют готовых решений для этого.
- Сайд-эффекты это не лучший выбор. Гораздо надёжнее фиксировать даже разовые события в состоянии.

ViewModel: One-off event antipatterns (2022, by Manuel Vivo from Google)

Side effects consumption control (1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 12.1, 12.2, 13, 14, 15, 16, 17, 18, 19, 20)

Сайд-эффекты

- Поток сайд-эффектов тоже может быть управляемым. Некоторые библиотеки предоставляют возможность его настроить.
  - Включить буферизацию событий, если их никто не слушает.
  - Повторять N последний событий новому слушателю.
  - Ограничивать число разрешённых слушателей.

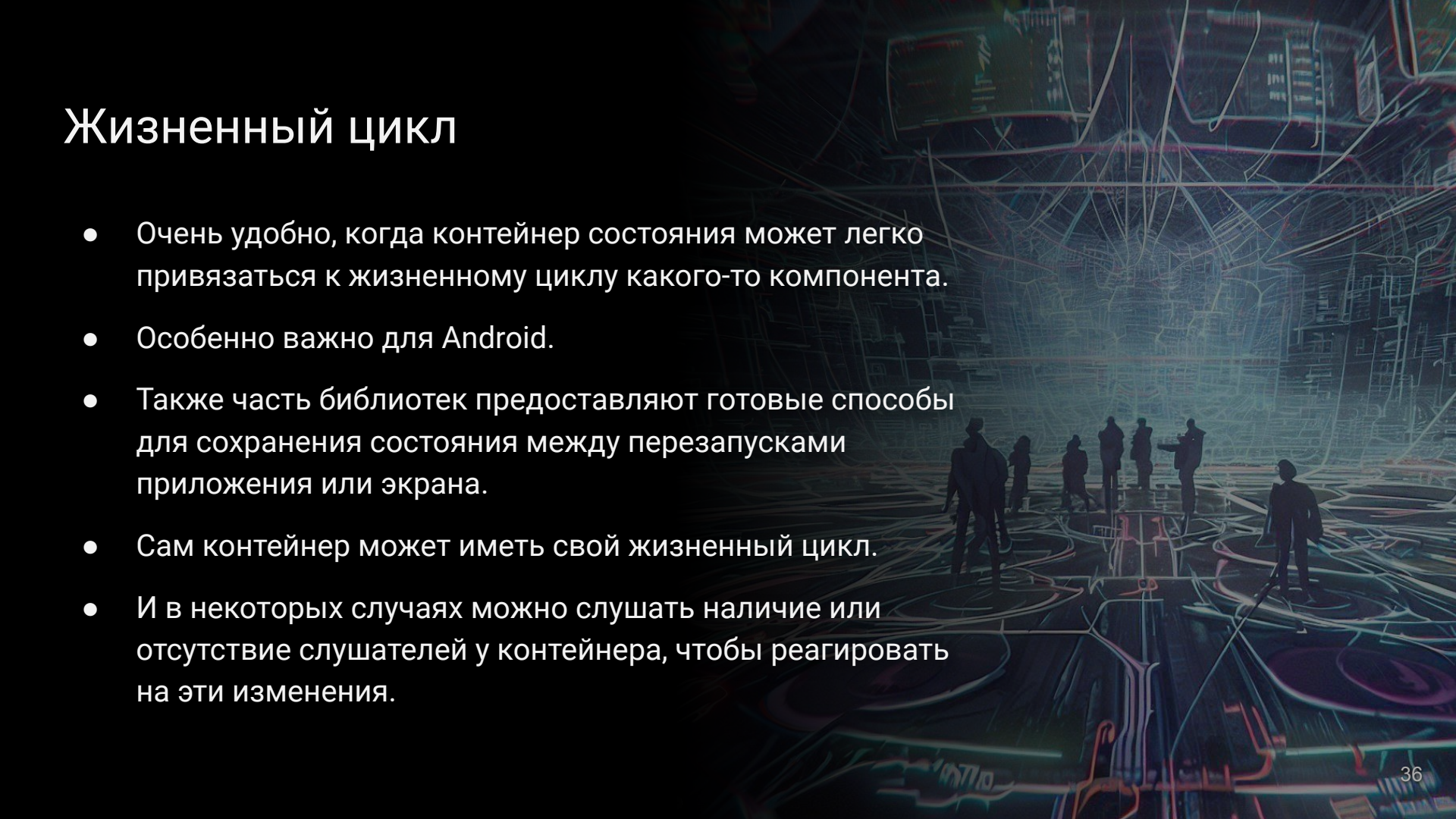


Long-running jobs

- Для реальных приложений необходимо запускать какие-то фоновые задачи в ответ на интенты. Обращаться к данным, что-то загружать, вычислять.
  - Почти все библиотеки дают возможности для подобных действий.
  - В некоторых случаях – это очень мощная и гибкая система управления задач.
    - Ограничение запуска нужным состоянием.
    - Защита от запуска одинаковых задач.
    - Авто-завершение задачи при запуске новой.

Жизненный цикл

- Очень удобно, когда контейнер состояния может легко привязаться к жизненному циклу какого-то компонента.
- Особенно важно для Android.
- Также часть библиотек предоставляют готовые способы для сохранения состояния между перезапусками приложения или экрана.
- Сам контейнер может иметь свой жизненный цикл.
- И в некоторых случаях можно слушать наличие или отсутствие слушателей у контейнера, чтобы реагировать на эти изменения.



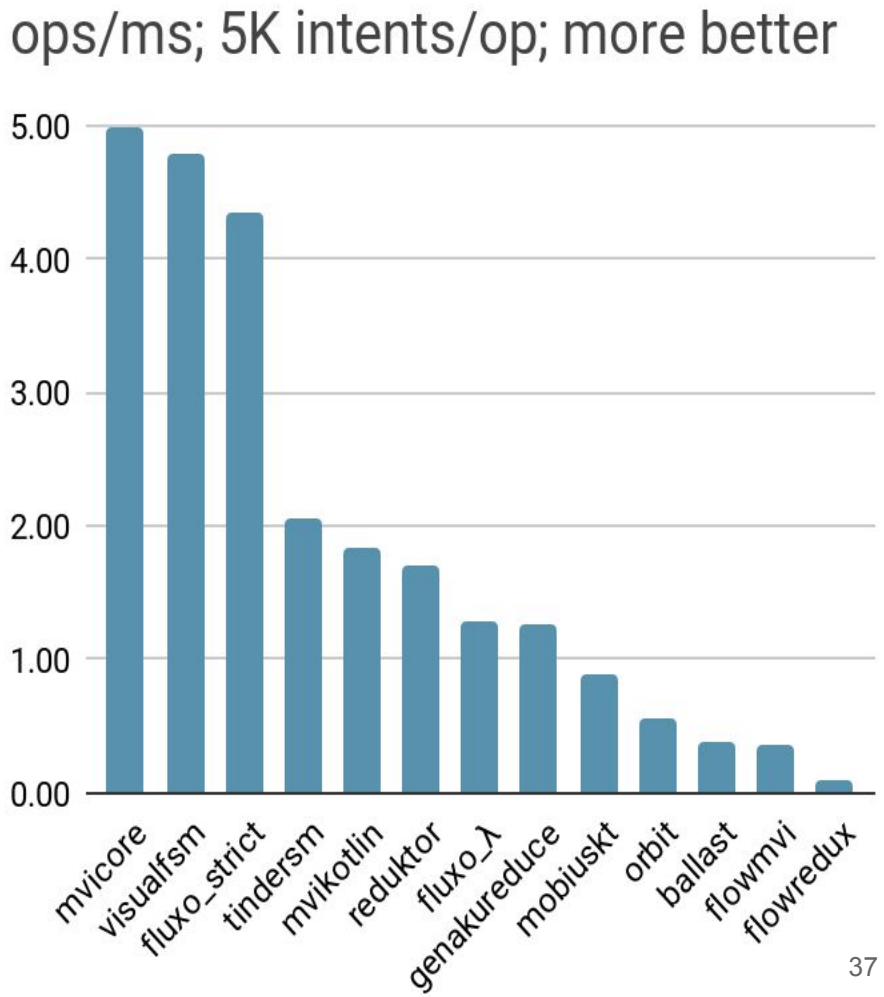
Производительность

Все протестированные библиотеки легко обрабатывают сотни тысяч, а некоторые и десятки миллионов простых интентов в секунду.

Тем не менее, какие-то из библиотек лучше оптимизированы и встречается разница до нескольких порядков!

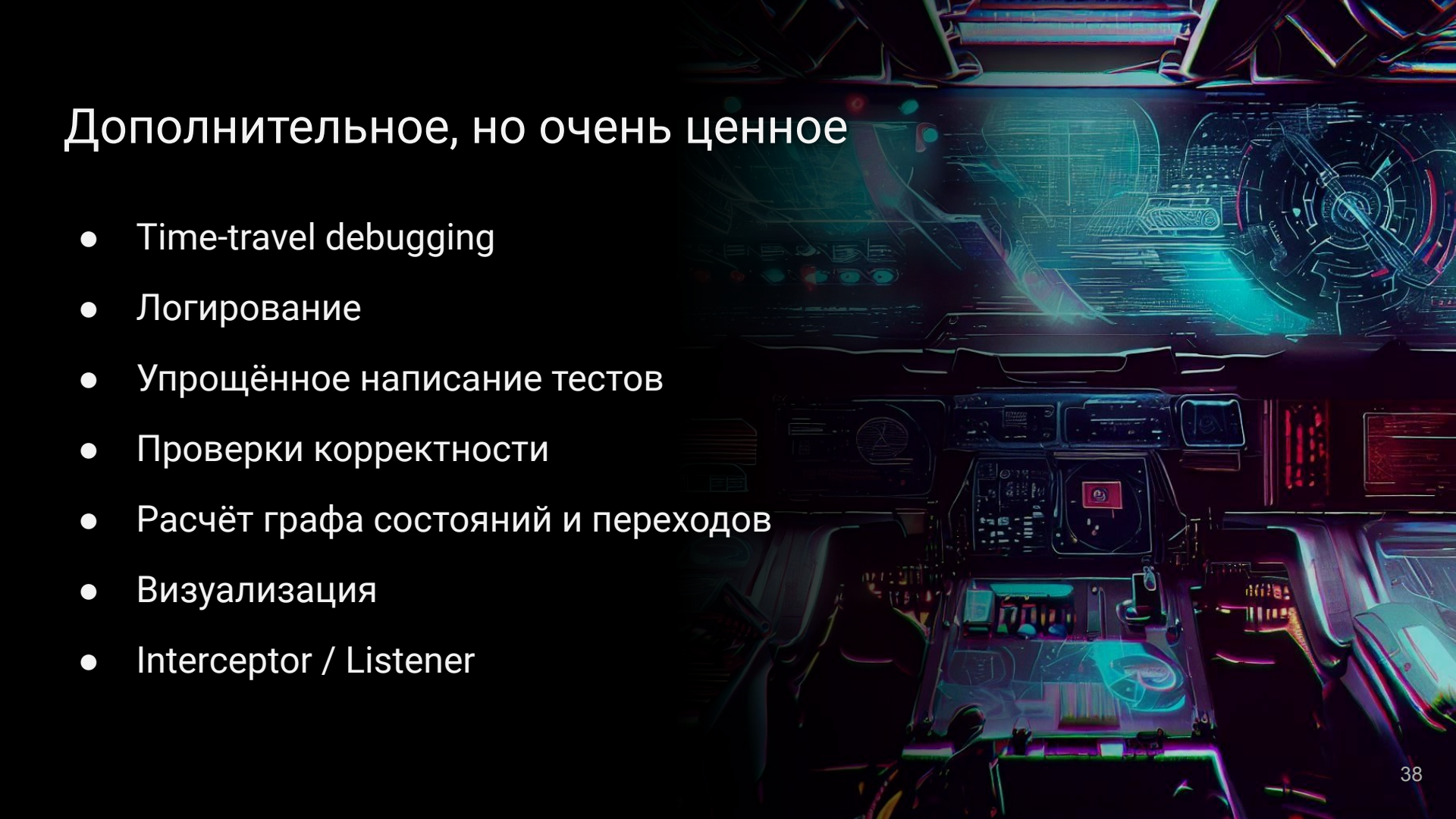
github.com/fluxo-kt/fluxo/actions/workflows/benchmark.yml (GitHub CI)

github.com/fluxo-kt/fluxo/actions/runs/4907652284 (GitHub CI)



Дополнительное, но очень ценное

- Time-travel debugging
- Логирование
- Упрощённое написание тестов
- Проверки корректности
- Расчёт графа состояний и переходов
- Визуализация
- Interceptor / Listener



Основные принципы

- Single source of truth

  Выделяем состояние на определённом уровне архитектуры приложения и храним в одном конкретном месте/контейнере.

  От вас зависит, будет ли это состояние одного виджета, всего экрана или целого приложения. Есть подходы для работы с любым вариантом.

  Текущие best-practices — хранить минимальные кусочки.

Основные принципы

- Single source of truth
- Неизменяемое состояние

  Immutable state — обязательное требование для корректности всех изменений, потоко-безопасности и общей надёжности всей архитектуры.

  Нельзя менять состояние иначе как централизовано через контейнер.

Основные принципы

- Single source of truth
- Неизменяемое состояние
- Реактивность

  Современные реактивные подходы вроде корутин или Rx/Reactive-имплементаций позволяют легко и надёжно получать обновления состояния, фильтровать или выполнять что-то на основе потока изменений, а также корректно обрабатывать жизненный цикл разных компонентов.

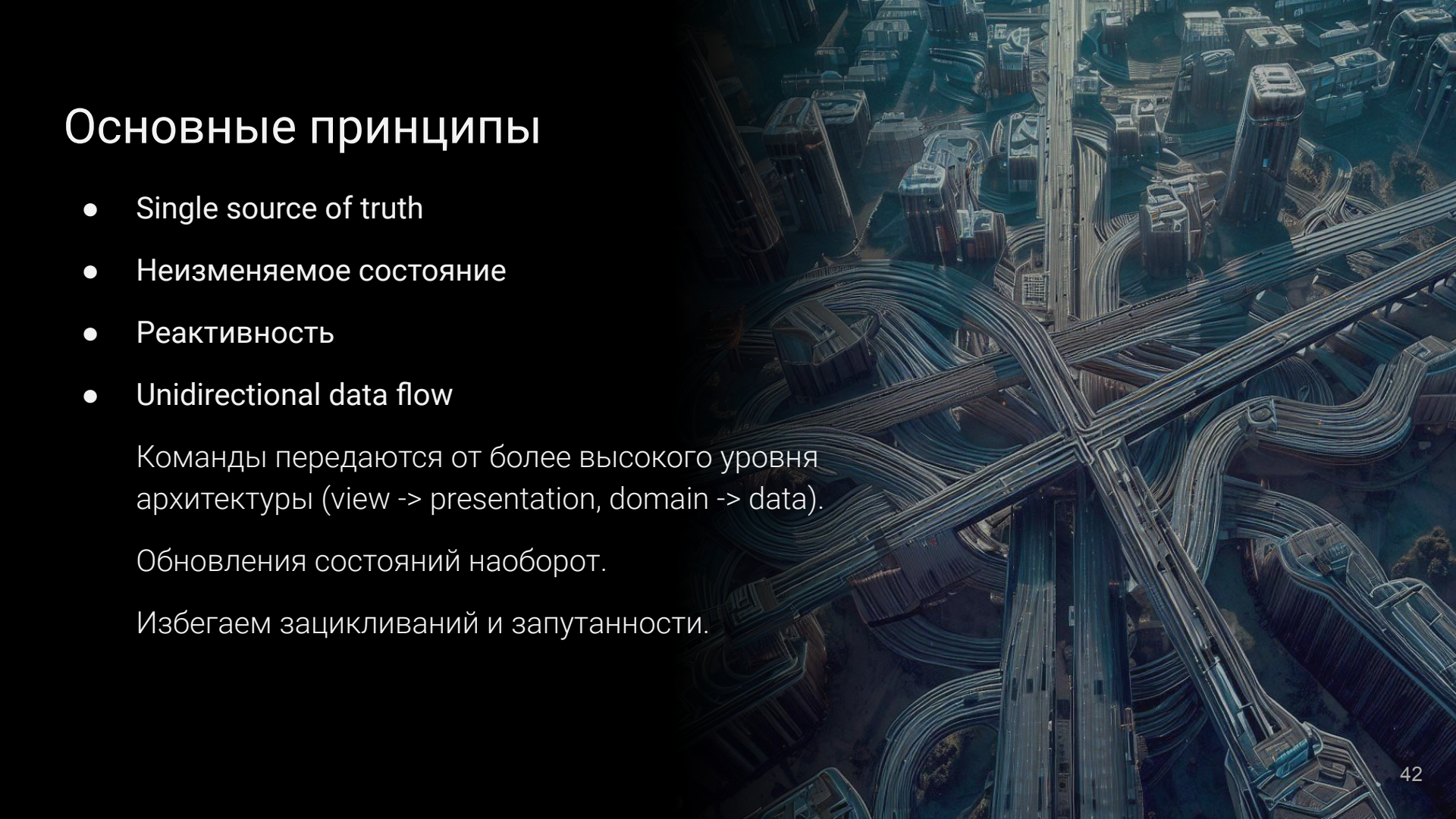
Основные принципы

- Single source of truth
- Неизменяемое состояние
- Реактивность
- Unidirectional data flow

  Команды передаются от более высокого уровня архитектуры (view -> presentation, domain -> data).

  Обновления состояний наоборот.

  Избегаем зацикливаний и запутанности.



Дополнительно

- Чистота функций
  - Мы можем сделать все части максимально простыми и изолировать сайд-эффекты.
  - Корректный контейнер состояния очень легко покрывать тестами и работать с ним.
  - Простота отладки и воспроизведения любых пользовательских ситуаций из одного лишь лога — отдельное удовольствие ;)

Short-list, рекомендации

- BLoC/Redux/MVI
  - MVIKotlin, Ballast, FlowMVI, Fluxo
- MVVM+/Orbit-way (*λ intents*)
  - Orbit MVI, Kotlin Bloc, Fluxo
- Finite-State Machine, Elm
  - VisualFSM, Elmslie


docs.google.com/spreadsheets/d/1gbwXU5Vp9QGvph1rLu0hv90KqrHNhkZyKTxk_I_yBJk

Short-list, рекомендации

- Управление потоком интентов
  - Ballast, FlowRedux, Fluxo
- Управление потоком сайд-эффектов
  - FlowMVI, FlowRedux, Fluxo
- Гарантированная доставка сайд-эффекта
  - Fluxo


docs.google.com/spreadsheets/d/1gbwXU5Vp9QGvph1rLu0hv90KqrHNhkZyKTxk_I_yBJk

Short-list, рекомендации

- Long-running jobs
  - VisualFSM, Ballast, Reduktor, Fluxo
- Subscriptions lifecycle
  - Orbit MVI, CommonStateMachine, Fluxo

docs.google.com/spreadsheets/d/1gbwXU5Vp9QGvph1rLu0hv90KqrHNhkZyKTxk_I_yBJk

Short-list, рекомендации

- Time-travel
  - MVIKotlin, Ballast, MviCore
- Graph tools
  - VisualFSM, StateMachine (Tinder)

docs.google.com/spreadsheets/d/1gbwXU5Vp9QGvph1rLu0hv90KqrHNhkZyKTxk_I_yBJk

Спасибо за внимание

Артём Шендрик

t.me/samally