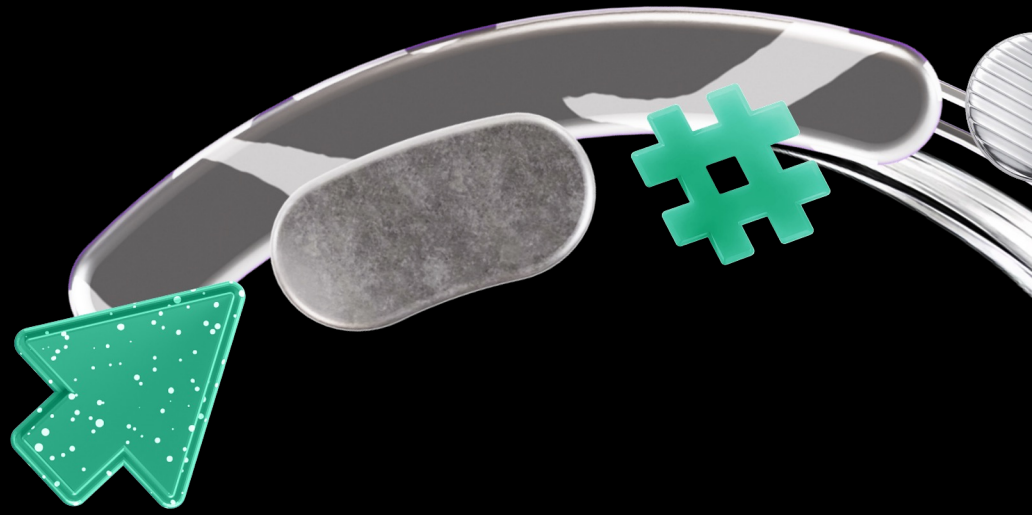
Система фейков

E2E без E2E: как мы тестируем продакшен-сценарии без тестовых пользователей и реальных зависимостей



Константин Горностаев
QA инженер, Авито

Константин Горностаев

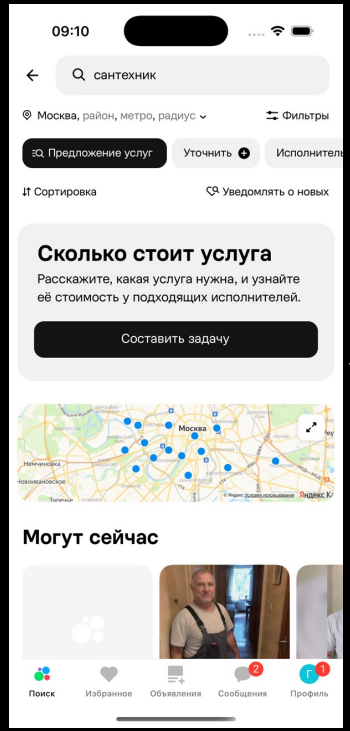
- В тестировании с 2017 года
- С нуля выстраиваю подходы и процессы тестирования продуктов
- Разрабатываю инструменты автоматизации тестирования
- Занимаюсь качеством агентных систем



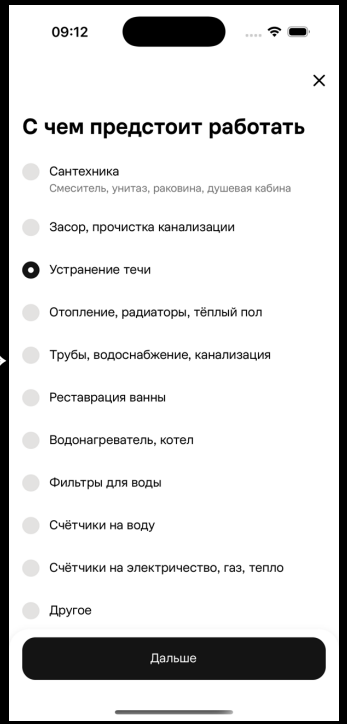
Более 100 категорий услуг

Заявки: один путь пользователя

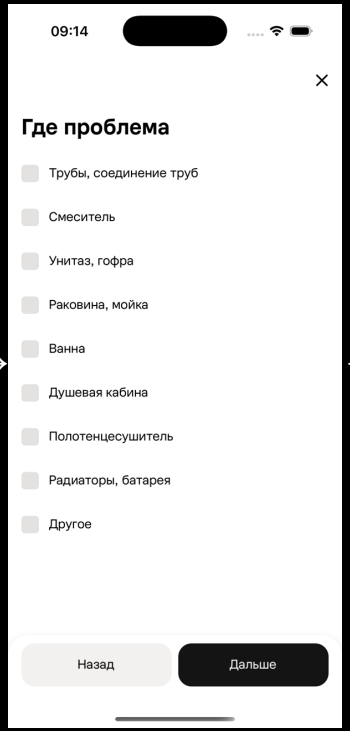
Вход



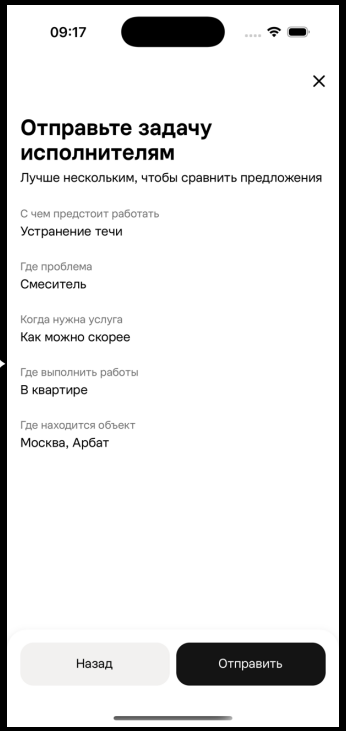
Анкета · экран 1



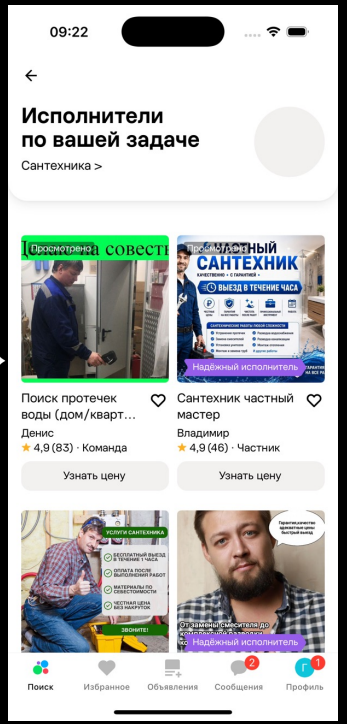
Анкета · экран 2



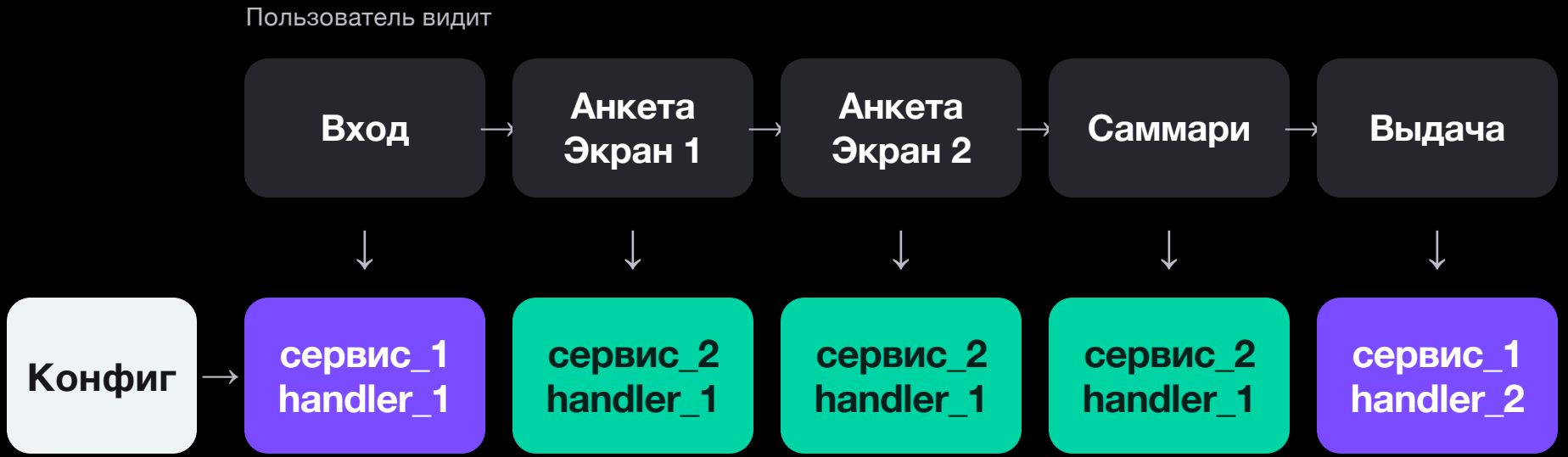
Саммари



Выдача



Один пользовательский путь — разные сервисы и хэндлеры



Поведение продукта задают и код, и конфиг

Конфиг задаёт поведение продукта

01

Точки входа

Где и при каких условиях предложить Заявку

Конфиг задаёт поведение продукта

01

Точки входа

Где и при каких условиях
предложить Заявку

02

Вопросы

Что спросить
и куда перейти дальше

Конфиг задаёт поведение продукта

01

Точки входа

Где и при каких условиях предложить Заявку

02

Вопросы

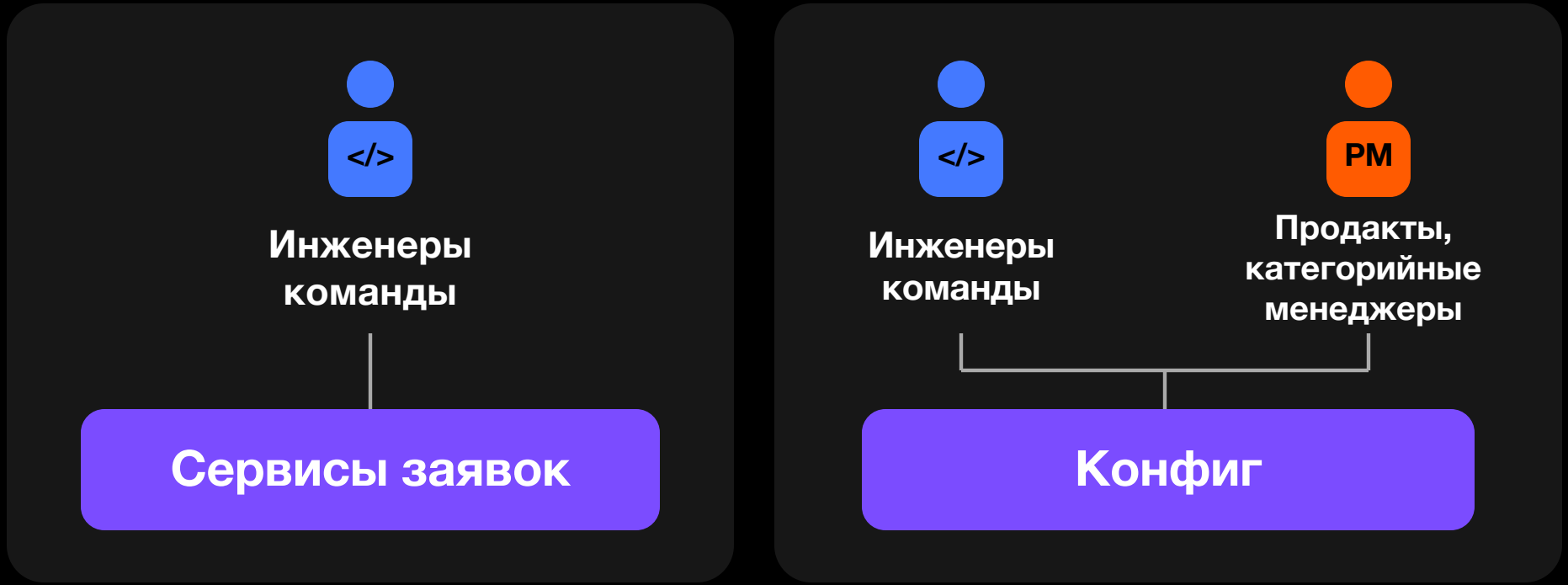
Что спросить и куда перейти дальше

03

Фильтры

Как настроить последующую выдачу

Кто отвечает за сервисы и конфиги



Что хотим автоматизировать

01

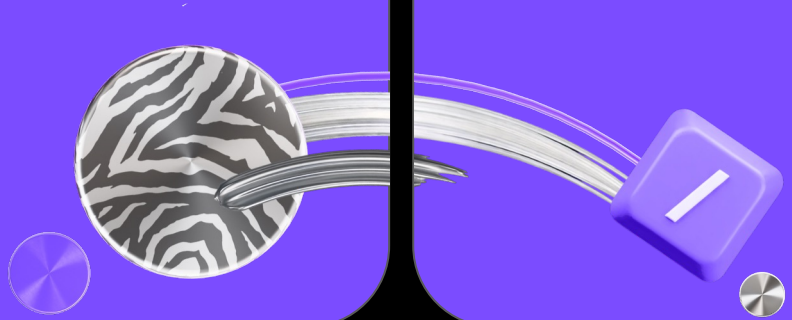
Проверять весь путь

Результаты действий пользователя
на всех этапах выбранного сценария.

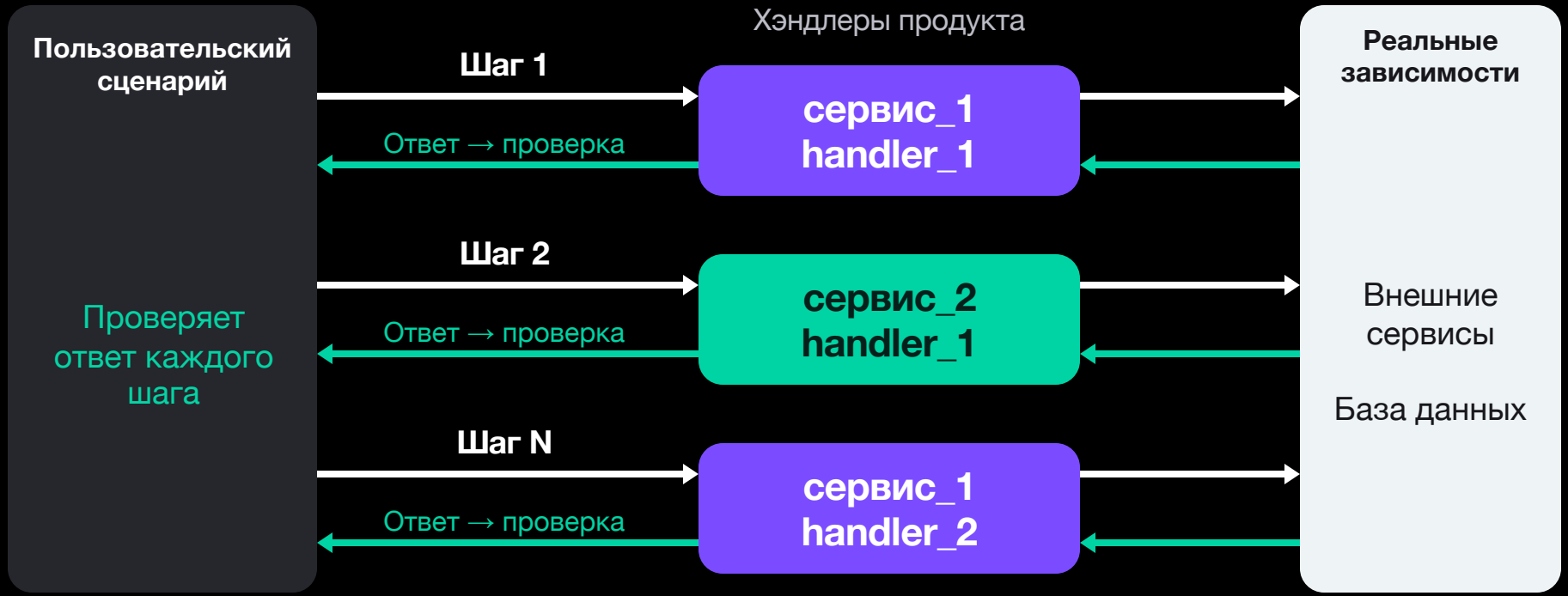
02

Повторять проверки

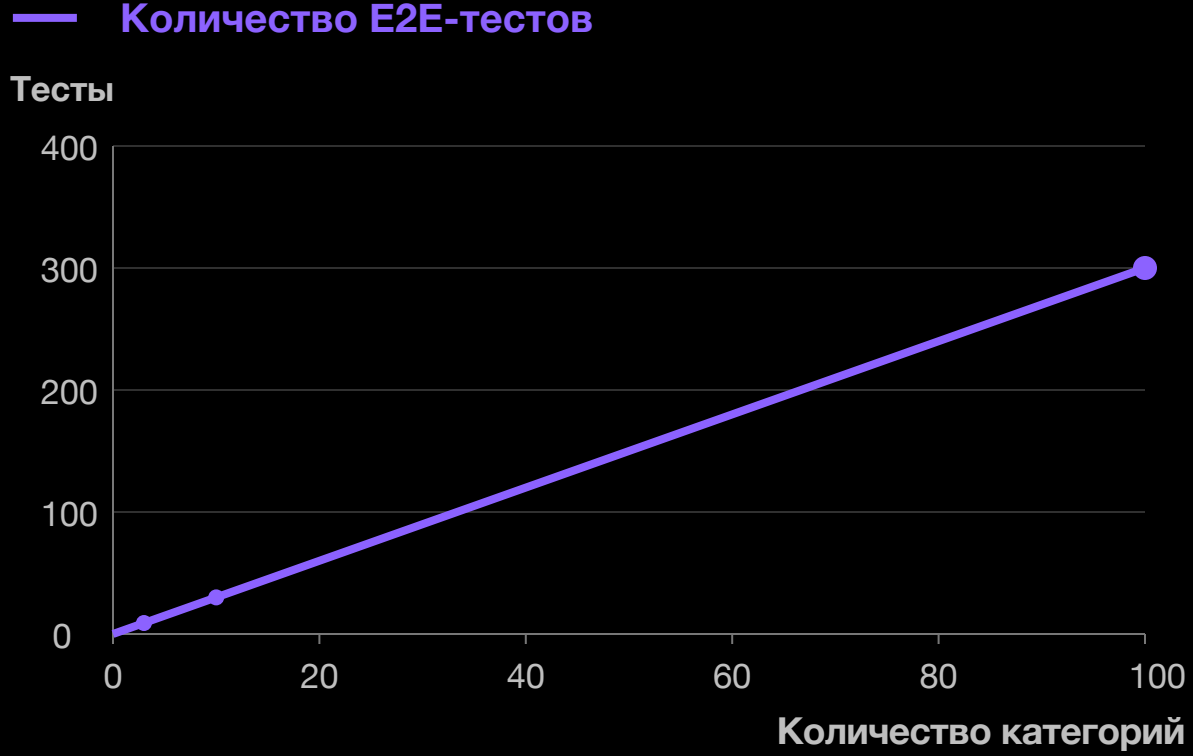
После изменений кода и конфигов.



E2E-тест воронки на платформах iOS, Android, MAV



100 категорий — уже 300 E2E-тестов



1 категория → 3 теста

по тесту на каждую платформу

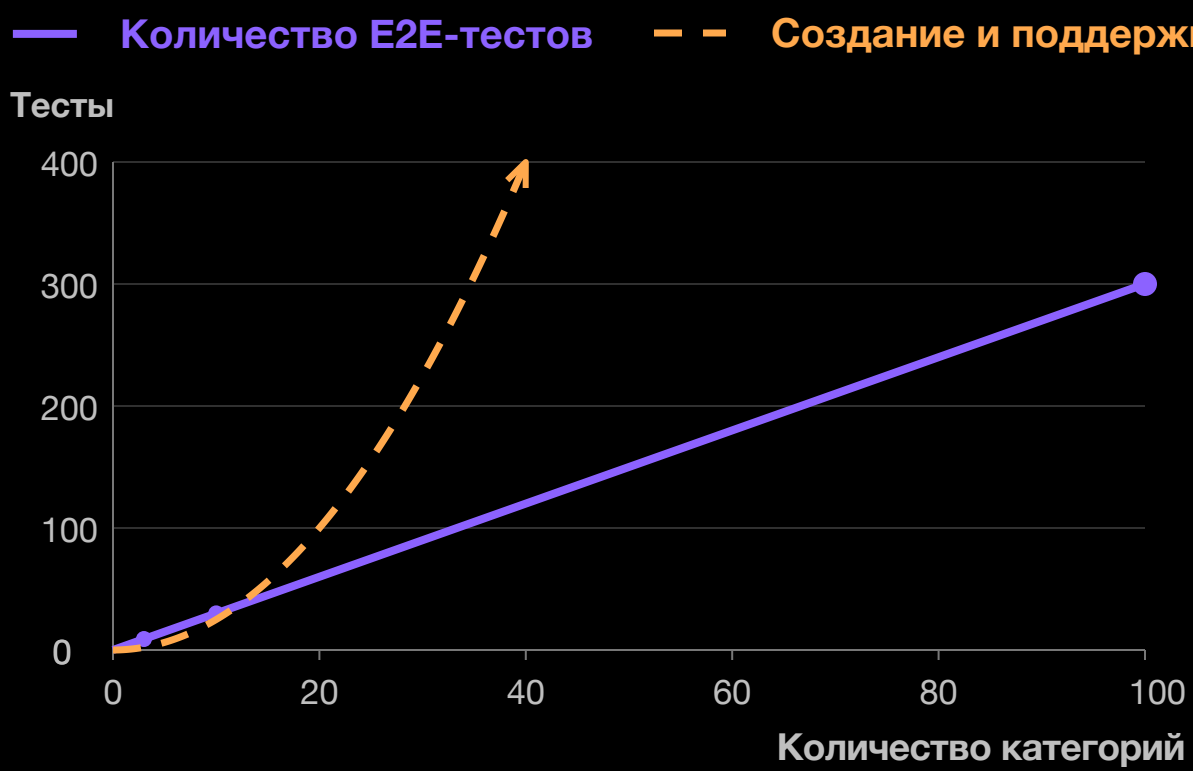
10 категорий → 30 тестов

В 10 раз больше тестов

100 категорий → 300 тестов

В 100 раз больше тестов

Затраты растут быстрее количества E2E-тестов



Растёт трудоёмкость создания новых тестов и поддержки существующих.

Что нам нужно от нового подхода

01

Сохранить проверку пользовательского пути целиком

От точки входа в Заявку до итоговой выдачи — все шаги и переходы выбранного сценария.

Одна воронка — множество пользовательских путей

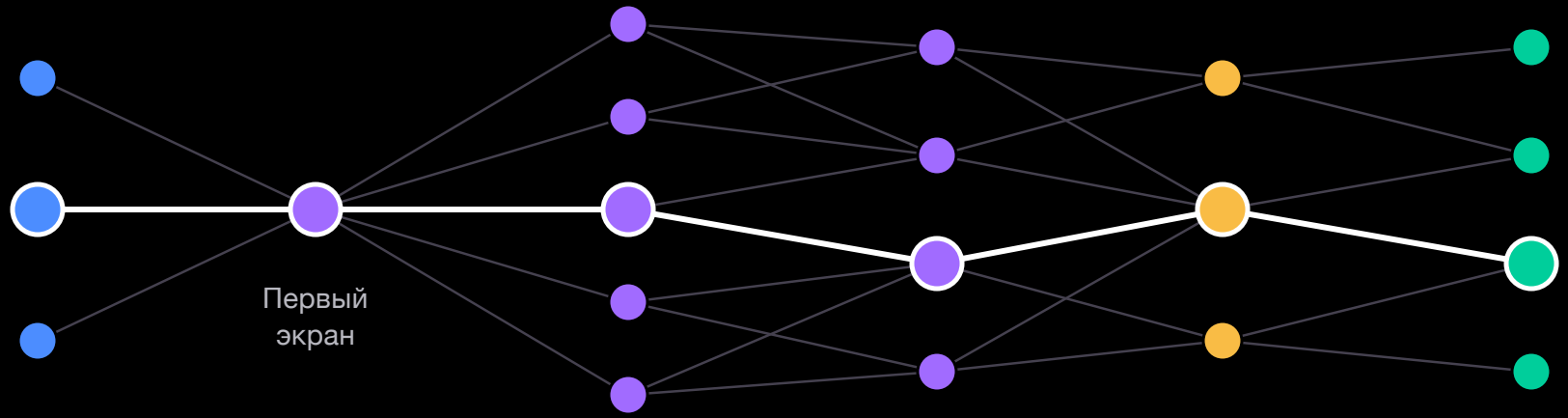
Разные точки входа и ответы создают разные варианты прохождения

Точки входа
Откуда пришёл пользователь

Анкета
Ответы и условия переходов

Саммари
Итоговые ответы

Выдача
Параметры поиска



Что нам нужно от нового подхода

01

Сохранить проверку пользовательского пути целиком

От точки входа в Заявку до итоговой выдачи — все шаги и переходы выбранного сценария.

02

Самостоятельный запуск проверок авторами конфигов

Автор конфига запускает тесты и понимает результат без обращения к команде разработки.

Что нам нужно от нового подхода

01

Сохранить проверку пользовательского пути целиком

От точки входа в Заявку до итоговой выдачи — все шаги и переходы выбранного сценария.

02

Самостоятельный запуск проверок авторами конфигов

Автор конфига запускает тесты и понимает результат без обращения к команде разработки.

03

Снизить затраты на создание и поддержку тестов

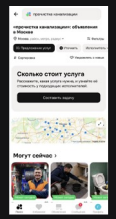
Меньше времени тратить на тесты при добавлении Категорий и изменении конфигов.

А если пройти тот же сценарий без UI?

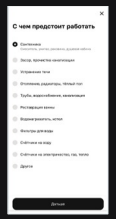
UI

Проверяем сценарий через интерфейс

Точка входа



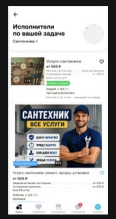
Анкета



Саммари



Выдача



Backend

Проверяем сценарий через обработчики →

Точка входа → Шаги анкеты → Саммари → Выдача

Что в конфиге описывает пользовательский путь?

ЭТАП	ЧТО ЗАДАЁТ КОНФИГ	ФРАГМЕНТ КОНФИГА
01 Точка входа	Доступность входа через виджет	`entryPoints` `"widgetEntryPoint": { … }`

Что в конфиге описывает пользовательский путь?

ЭТАП	ЧТО ЗАДАЁТ КОНФИГ	ФРАГМЕНТ КОНФИГА
01 Точка входа	Доступность входа через виджет	`entryPoints` `"widgetEntryPoint": { … }`
02 Анкета	Условия переходов между экранами	`screens[15].condition` `"rawCondition": "'Устранение течи' == p0001_r1"`

Что в конфиге описывает пользовательский путь?

ЭТАП	ЧТО ЗАДАЁТ КОНФИГ	ФРАГМЕНТ КОНФИГА
01 Точка входа	Доступность входа через виджет	`entryPoints` `"widgetEntryPoint": { … }`
02 Анкета	Условия переходов между экранами	`screens[15].condition` `"rawCondition": "'Устранение течи' == p0001_r1"`
03 Саммари	Настройки итогового экрана	`screens[47]` `"type": "Summary", "version": "33"`

Что в конфиге описывает пользовательский путь?

ЭТАП	ЧТО ЗАДАЁТ КОНФИГ	ФРАГМЕНТ КОНФИГА
01 Точка входа	Доступность входа через виджет	`entryPoints` `"widgetEntryPoint": { … }`
02 Анкета	Условия переходов между экранами	`screens[15].condition` `"rawCondition": "'Устранение течи' == p0001_r1"`
03 Саммари	Настройки итогового экрана	`screens[47]` `"type": "Summary", "version": "33"`
04 Выдача	Правила формирования поисковых фильтров	searchParams.conditionalParams · фрагмент `string(p0008_r1) contains 'Раковина, мойка'` → Услуга = Устранение протечки

Что в конфиге описывает пользовательский путь?

ЭТАП	ЧТО ЗАДАЁТ КОНФИГ	ФРАГМЕНТ КОНФИГА
01 Точка входа	Доступность входа через виджет	`entryPoints` `"widgetEntryPoint": { … }`
02 Анкета	Условия переходов между экранами	`screens[15].condition` `"rawCondition": "'Устранение течи' == p0001_r1"`
03 Саммари	Настройки итогового экрана	`screens[47]` `"type": "Summary", "version": "33"`
04 Выдача	Правила формирования поисковых фильтров	searchParams.conditionalParams · фрагмент `string(p0008_r1) contains 'Раковина, мойка'` → Услуга = Устранение протечки

Конфиг задаёт путь и правила переходов. Проверяем, как их выполняет код сервисов.

Проверяем переход на настоящем коде



Проверяем переход на настоящем коде



Собираем шаги в пользовательский сценарий

Точка входа → Проверить виджет

Анкета → Проверить переходы

Саммари → Проверить ответы

Выдача → Проверить фильтры

Для каждого шага

`input`
Входные данные

`output`
Ожидаемый ответ

`interactions`
Ответы зависимостей

Собираем шаги в пользовательский сценарий

Точка входа
Проверить виджет

→

Анкета
Проверить переходы

→

Саммари
Проверить ответы

→

Выдача
Проверить фильтры

Для каждого шага

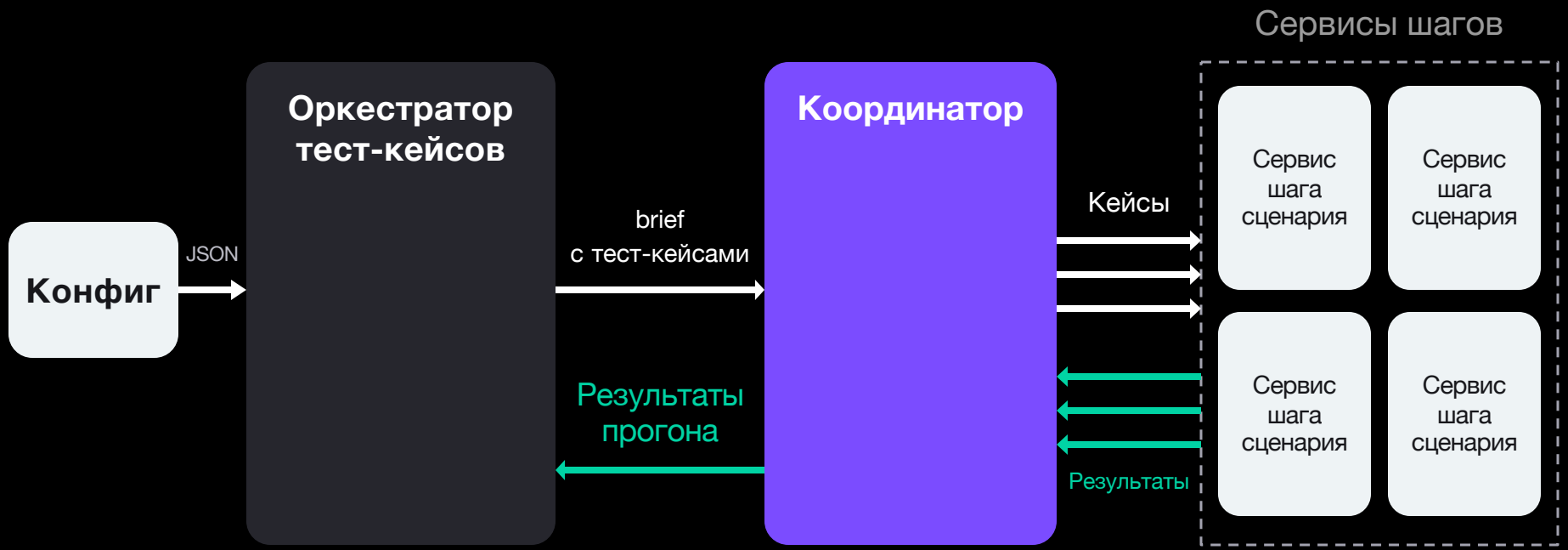
`input`
Входные данные

`output`
Ожидаемый ответ

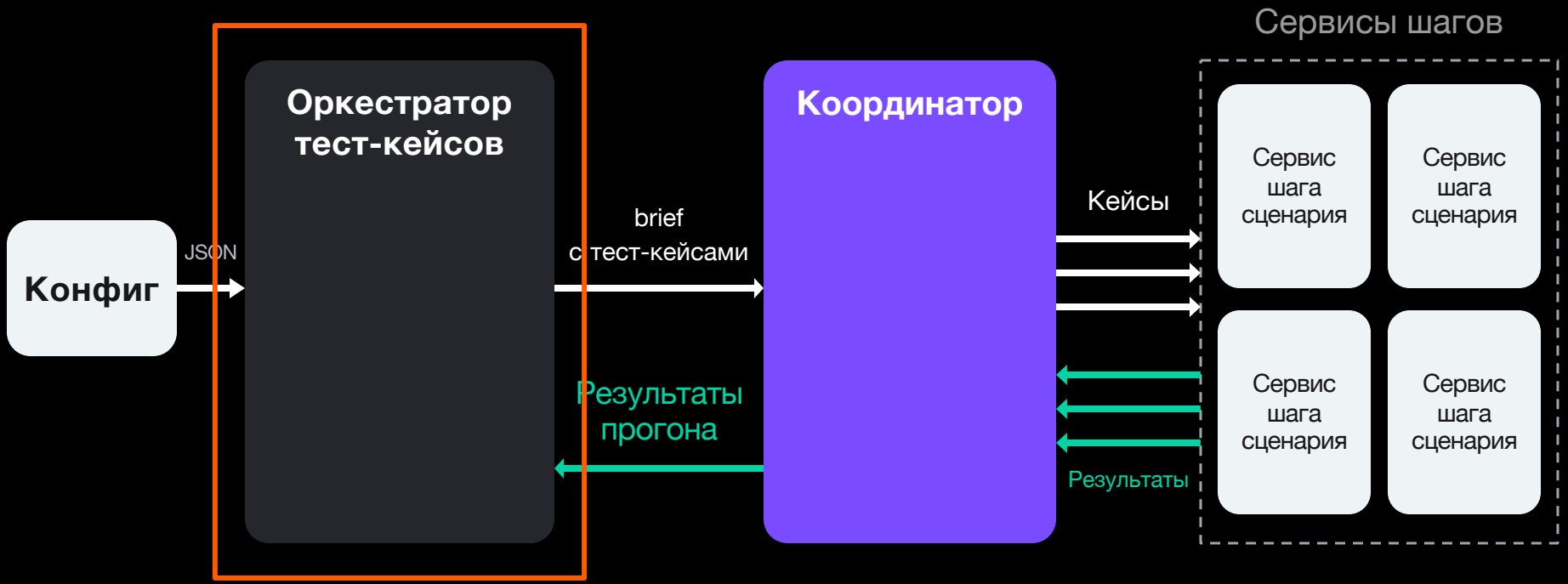
`interactions`
Ответы зависимостей

Как автоматически готовить и запускать сценарии для разных категорий?

Как работает система фейков



Оркестратор: собираем сценарии



Оркестратор: из конфига — в сценарии

JSON-конфиг продукта



Оркестратор тест-кейсов

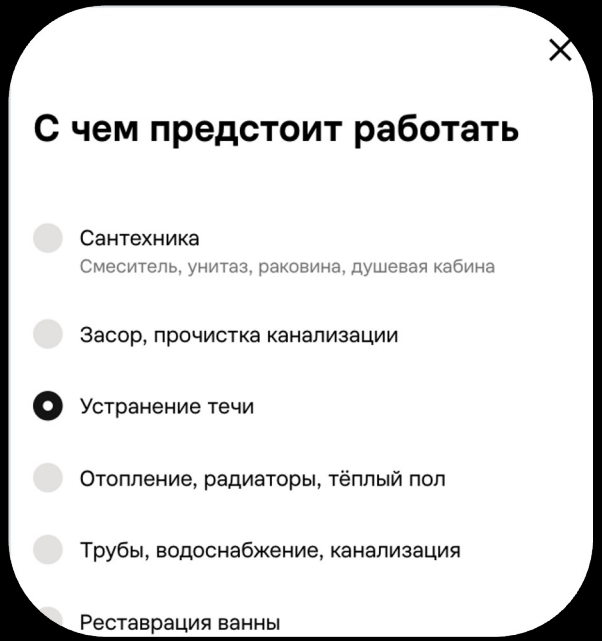
1. Генерирует тест-кейсы
2. Готовит ответы зависимостей
3. Собирает пользовательские сценарии

Пользовательские сценарии



brief с тест-кейсами

1. Генерирует тест-кейсы: входные данные



Фрагмент экрана анкеты

input — выбранный ответ

```
"pageNumber": 1,
"pageResult1": "Устранение течи"
```

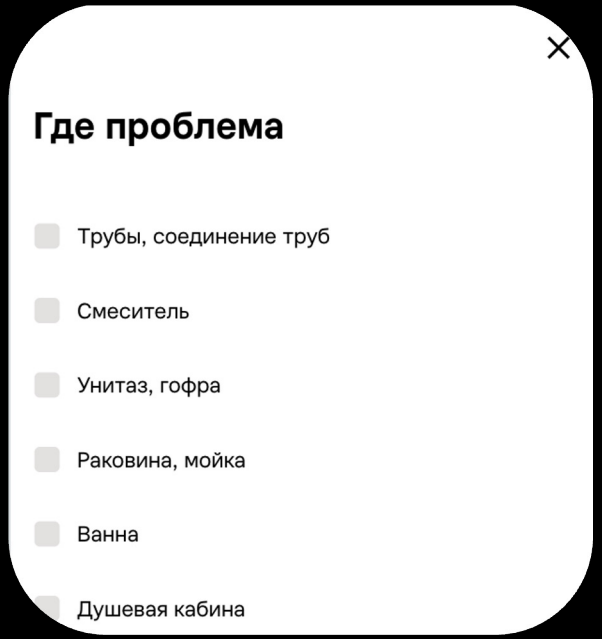
Условие из конфига

plumbing · screens[15] · фрагмент

```
"condition": {
  "isExist": true,
  "rawCondition":
    "'Устранение течи' == p0001_r1"
}
```

pageResult1 → p0001_r1: ответ первого экрана

1. Генерирует тест-кейсы: ожидаемый ответ



Фрагмент экрана анкеты

output — ожидаемый экран

success.main.params · фрагмент

```
"PageNumber": 16,
"PageTitle": "Где проблема"
```

Что задаёт этот кейс

После ответа «Устранение течи» ожидаем данные экрана «Где проблема» от хэндлера.

2. Готовит ответы зависимостей

`interactions` — подготовленные ответы внешних сервисов

ДЛЯ КАКОГО ВЫЗОВА

Сервис хранения заявок
`isUserFormOwner`

↓

```
{ "isOwner": true }
```

ДЛЯ КАКОГО ВЫЗОВА

Геосервис
`getDocumentsByTypesV1`

↓

```
{
  "documents": [{
    "key": {
      "avitoId": 637640,
      "type": "region"
    },
    "name": "Москва"
  }]
}
```

3. Добавляет готовый кейс в сценарий

```
suite := coordinator.TestSuite{
  TestCase: coordinator.TestCase{
    Input:        testCase.Input,
    Output:       testCase.Output,
    Interactions: testCase.Interactions,
  },
  ClientName: "services-requests-api",
  HandlerName: "RunNextPageTests",
}
step := int64(startIndex + i)
suite.TestCaseStep = &step

testSuites = append(testSuites, suite)
```

Берём готовый кейс: вход, ожидаемый ответ и ответы зависимостей.

3. Добавляет готовый кейс в сценарий

```
suite := coordinator.TestSuite{
  TestCase: coordinator.TestCase{
    Input:        testCase.Input,
    Output:       testCase.Output,
    Interactions: testCase.Interactions,
  },
  ClientName: "services-requests-api",
  HandlerName: "RunNextPageTests",
}
step := int64(startIndex + i)
suite.TestCaseStep = &step

testSuites = append(testSuites, suite)
```

Указываем сервис
и тестовую RPC-ручку.

3. Добавляет готовый кейс в сценарий

```
suite := coordinator.TestSuite{
  TestCase: coordinator.TestCase{
    Input:        testCase.Input,
    Output:       testCase.Output,
    Interactions: testCase.Interactions,
  },
  ClientName: "services-requests-api",
  HandlerName: "RunNextPageTests",
}
step := int64(startIndex + i)
suite.TestCaseStep = &step

testSuites = append(testSuites, suite)
```

Задаём порядок кейса.

`startIndex` — длина списка,
`i` — индекс нового кейса.

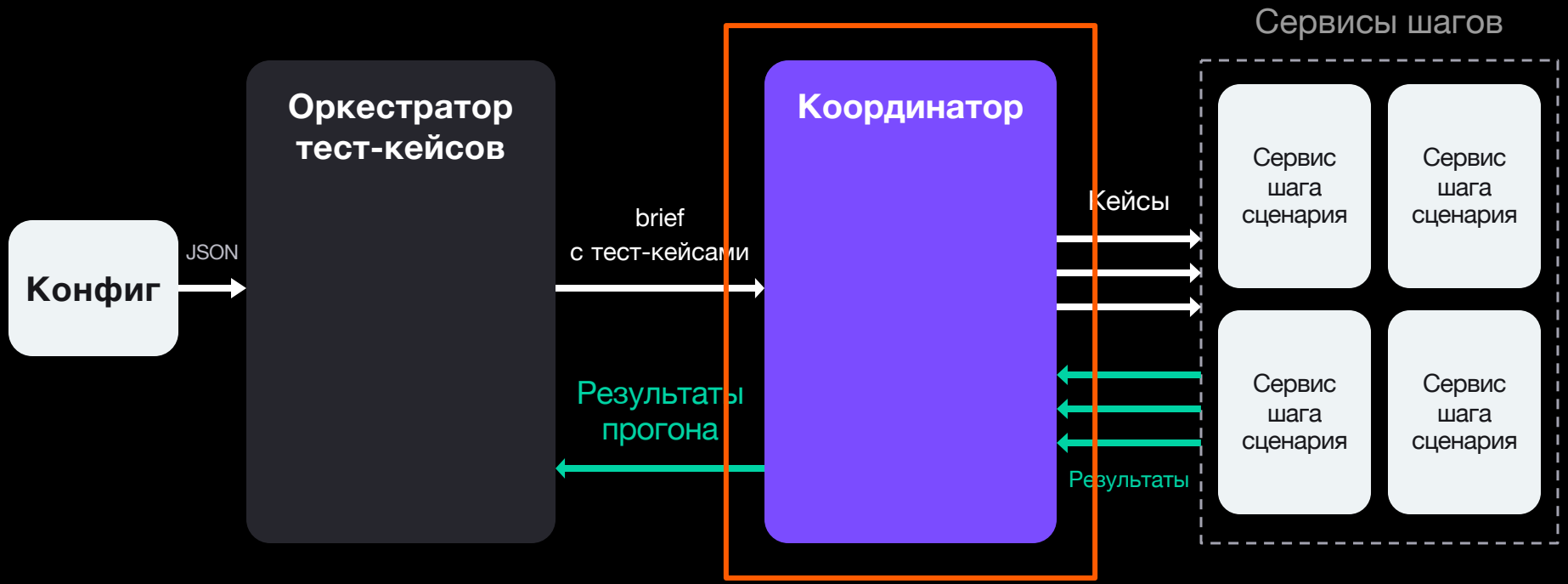
3. Добавляет готовый кейс в сценарий

```
suite := coordinator.TestSuite{
  TestCase: coordinator.TestCase{
    Input:        testCase.Input,
    Output:       testCase.Output,
    Interactions: testCase.Interactions,
  },
  ClientName: "services-requests-api",
  HandlerName: "RunNextPageTests",
}
step := int64(startIndex + i)
suite.TestCaseStep = &step

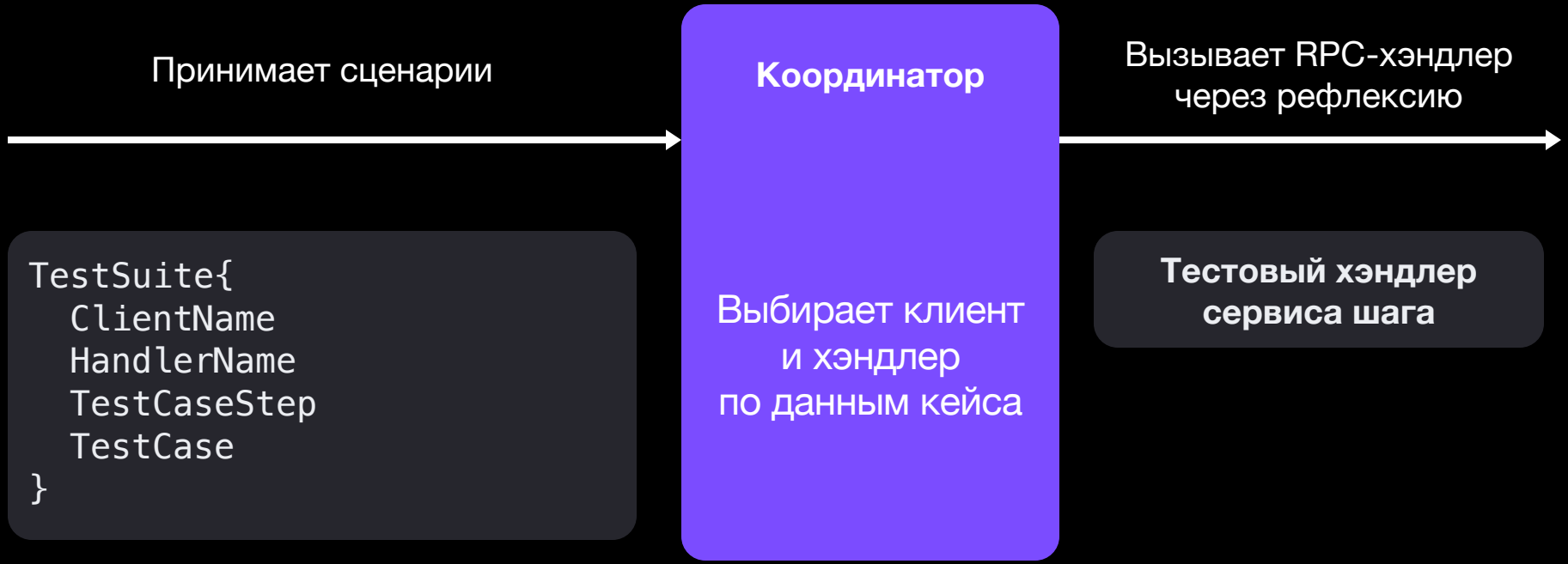
testSuites = append(testSuites, suite)
```

Добавляем шаг в сценарий.

Координатор: направляем кейсы



Координатор: из сценария — в RPC-вызов



Координатор: вызываем шаг сценария

```
client, ok := h.clients[suite.ClientName]
method := reflect.ValueOf(client).
  MethodByName(suite.HandlerName)

methodType := method.Type()
inType := methodType.In(1)
inValue := reflect.New(inType.Elem())
err := fillStruct(inValue, suite.TestCase)

results := method.Call([]reflect.Value{
  reflect.ValueOf(ctx),
  inValue,
})
```

По `ClientName` выбираем клиент сервиса.

По `HandlerName` находим его метод.

Координатор: вызываем шаг сценария

```
client, ok := h.clients[suite.ClientName]
method := reflect.ValueOf(client).
  MethodByName(suite.HandlerName)

methodType := method.Type()
inType := methodType.In(1)
inValue := reflect.New(inType.Elem())
err := fillStruct(inValue, suite.TestCase)

results := method.Call([]reflect.Value{
  reflect.ValueOf(ctx),
  inValue,
})
```

Узнаём, какую структуру запроса принимает метод.

Создаём её и заполняем данными кейса.

Координатор: вызываем шаг сценария

```
client, ok := h.clients[suite.ClientName]
method := reflect.ValueOf(client).
  MethodByName(suite.HandlerName)

methodType := method.Type()
inType := methodType.In(1)
inValue := reflect.New(inType.Elem())
err := fillStruct(inValue, suite.TestCase)

results := method.Call([]reflect.Value{
  reflect.ValueOf(ctx),
  inValue,
})
```

Вызываем метод, передавая контекст и запрос с данными кейса.

Рефлексия: один механизм вызова для разных шагов

В сценарии — имя метода

```
"RunNextPageTests"
```

Рефлексия

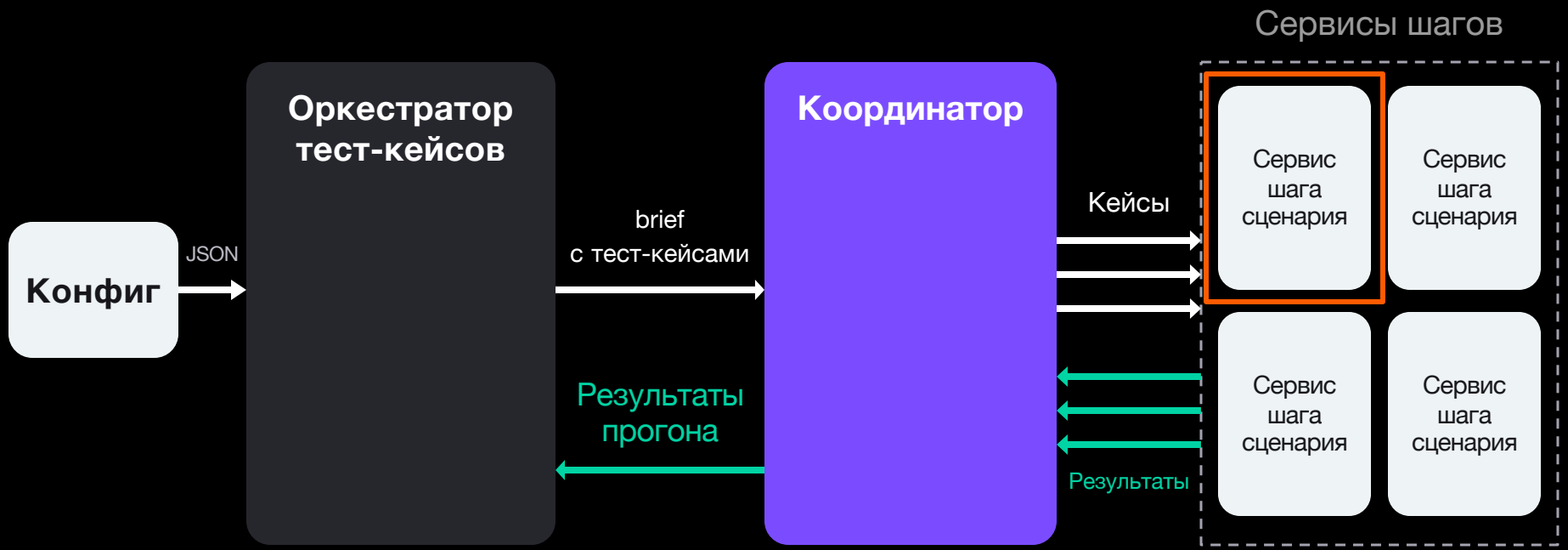


На выбранном клиенте

```
RunNextPageTests(...)
```

Координатор узнаёт тип запроса и вызывает нужный метод во время выполнения.

Сервис шага: выполняем проверку



Сервис шага: запускаем настоящий хэндлер

Входные данные,
ожидаемый ответ,
фейки зависимостей

```
RunTestsIn{
  Input
  Output
  Interactions
}
```

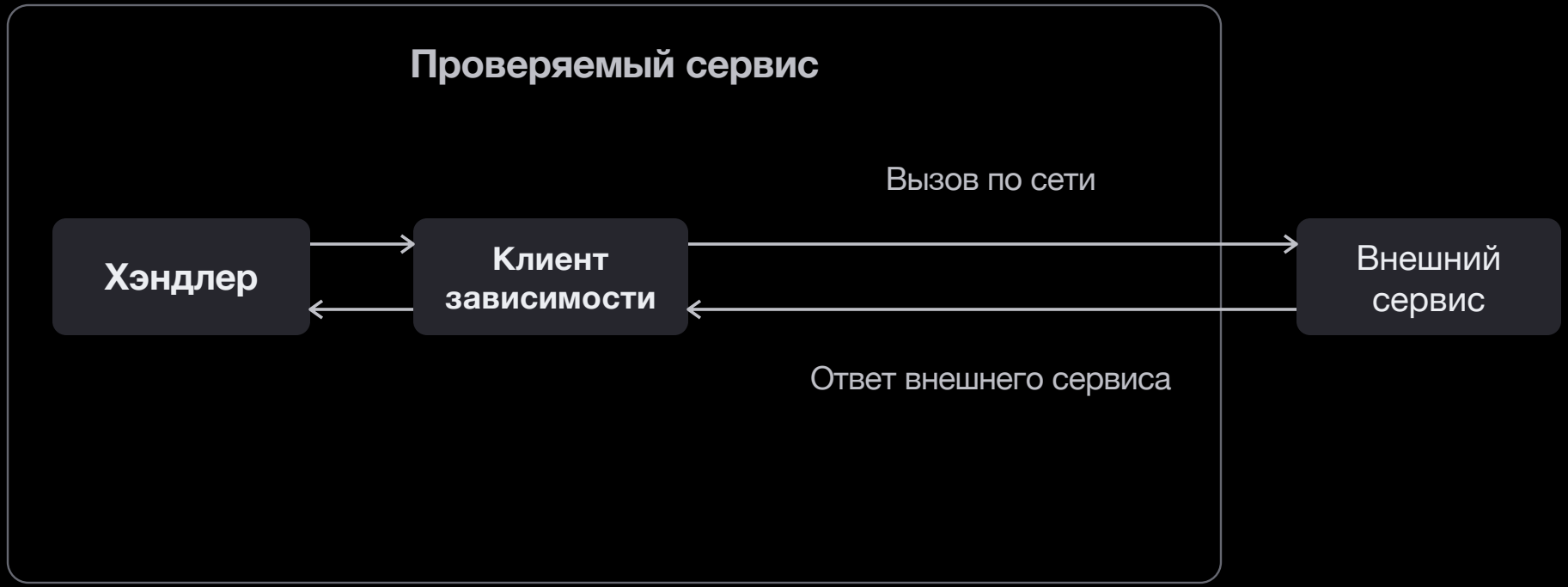
Сервис шага сценария

test handler

1. Создаём изолированное окружение
2. Создаём и вызываем настоящий хэндлер с данными кейса
3. Подменяем ответы внешних сервисов
4. Сравниваем фактический ответ с ожидаемым

Результат
шага сценария

1. Создаём изолированное окружение



1. Создаём изолированное окружение



1. Создаём изолированное окружение

```
mwFactory := wrapper.ClientMiddlewareFactory(
  in.Interactions,
)

requestStorageClient := helpers.SafeNew(
  briefrpc.ClientFor[storage.Server](
    mwFactory.Option(storage.Name),
  ),
)
```

`in.Interactions`

Ответы зависимостей
из тест-кейса.

Создаём storage-клиент
с подключённой подменой.

2. Собираем бизнес-логику с подготовленным клиентом

```
storageAdapter := requests_storage.New(
  requestStorageClient,
)

orderService, err := order.New(
  storageAdapter, …,
)
```

Тот же код бизнес логики,
что и в продакшене.

Подмена подключена
к клиенту зависимости.

2. Создаём и вызываем настоящий хэндлер

```
nextPageHandler := next_page_v2.New(
  orderService, metric,
)
req := &RequestWithQuery[…]{Data: in.Input}
resp := &ResponseM[…]{}

err := nextPageHandler.Handle(ctx, req, resp)
```

`orderService` — логика с подготовленным клиентом.

`req` — данные из `in.Input`
`resp` — результат шага

3. Подменяем ответы внешних сервисов



3. Подменяем ответы внешних сервисов

```
key := p.name + "." + name
resp, ok := p.interactions[key]

if !ok {
  return nil, fmt.Errorf(
    "mock response not found for %s", key)
}
```

`p.name` — сервис
`name` — RPC-метод

```
services-requests-storage
  .isUserFormOwner
            ↓
  { "isOwner": true }
```

3. Подменяем ответы внешних сервисов

```
out := reflect.New(
  reflect.TypeOf(outNil).Elem(),
).Interface()

if err := utils.FromJson(out, resp);
    err != nil {
  return nil, err
}

return out, nil
```

`outNil` задаёт тип ответа.

Заполняем `out` из JSON `resp`

```
{ "isOwner": true }
```

↓

`out.IsOwner` = true

4. Сравниваем фактический ответ с ожидаемым

Ожидаемый · in.Output

success.main.params · фрагмент

```
"PageNumber": 16,
"PageTitle": "Где проблема"
```

4. Сравниваем фактический ответ с ожидаемым

Ожидаемый · in.Output

success.main.params · фрагмент

```
"PageNumber": 16,
"PageTitle": "Где проблема"
```

Фактический · resp.Data

success.main.params · фрагмент

```
"PageNumber": 16,
"PageTitle": "Где проблема"
```

4. Сравниваем фактический ответ с ожидаемым

Ожидаемый · in.Output

success.main.params · фрагмент

```
"PageNumber": 16,
"PageTitle": "Где проблема"
```

Фактический · resp.Data

success.main.params · фрагмент

```
"PageNumber": 16,
"PageTitle": "Где проблема"
```

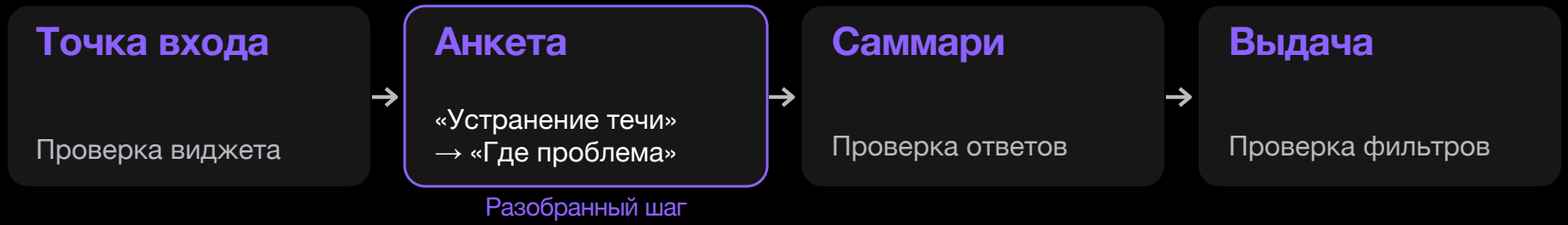
```
if err := tests_helpers.AssertJSONEqual(
  in.Output, resp.Data,
); err != nil {
  out.Status = dto.Status_FAILED
} else {
  out.Status = dto.Status_OK
}
```

При HTTP 200 сравниваем два JSON.

`out.Status` — результат тестовой ручки.

Результат сценария складывается из шагов

Разобранный переход — один из шагов пользовательского пути



Результат сценария складывается из шагов

Разобранный переход — один из шагов пользовательского пути

Точка входа
Проверка виджета

→

Анкета
«Устранение течи»
→ «Где проблема»

→

Саммари
Проверка ответов

→

Выдача
Проверка фильтров

Разобранный шаг

Для каждого шага оркестратор сохраняет результаты прогона конфига

Номер шага
`testCaseStep`

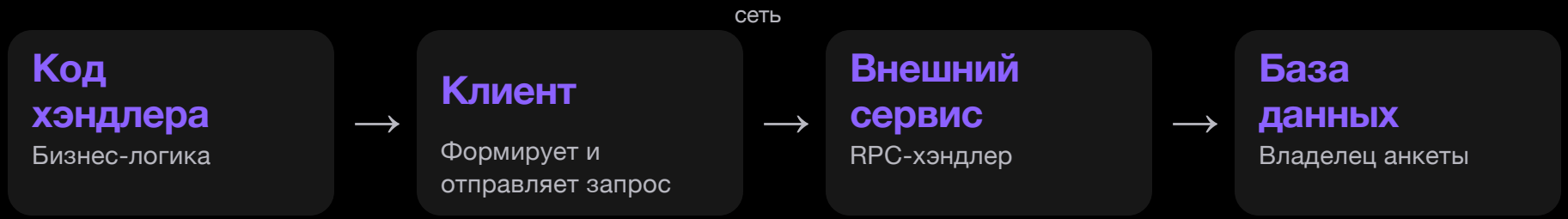
Статус проверки
`status: ok / failed`

Описание ошибки
`errorMessage`

Одна система — разные способы подмены

Стабы, фейки и моки

Обычный вызов: идём во внешний сервис



ЧТО ДЕЛАЕТ ВНЕШНИЙ СЕРВИС

Получает владельца анкеты из хранилища и сравнивает его с userID из запроса.

Для владельца возвращает `{ "isOwner": true }`

Стаб: возвращаем подготовленный ответ

Код хэндлера
Бизнес-логика

→

Клиент
Адаптер и RPC

→

Стаб в middleware
Подмена RPC

→

Готовый JSON
Ответ из кейса

ЗАРАНЕЕ ЗАДАЁМ ОТВЕТ

```
services-requests-storage.isUserFormOwner
```

```
{ "isOwner": true }
```

Текущий стаб не учитывает userID и formID.
Запрос во внешний сервис не отправляется.

Фейк: вычисляем ответ по тестовым данным

```
func isUserFormOwner(
  formID, userID int64,
) bool {
  owners := map[int64]int64{
    42: 123456,
    43: 654321,
  }
  ownerID, exists := owners[formID]
  return exists && ownerID == userID
}
```

Получаем `formID` и `userID` из запроса.

Фейк: вычисляем ответ по тестовым данным

```
func isUserFormOwner(
  formID, userID int64,
) bool {
  owners := map[int64]int64{
    42: 123456,
    43: 654321,
  }
  ownerID, exists := owners[formID]
  return exists && ownerID == userID
}
```

Задаём тестовые пары:
«анкета — её владелец».

Фейк: вычисляем ответ по тестовым данным

```
func isUserFormOwner(
  formID, userID int64,
) bool {
  owners := map[int64]int64{
    42: 123456,
    43: 654321,
  }
  ownerID, exists := owners[formID]
  return exists && ownerID == userID
}
```

Находим владельца анкеты и сравниваем с `userID`.

```
42, 123456 → true
42, 654321 → false
```

Фейк учитывает данные запроса

Передали анкету другого пользователя

Текущий стаб

```
isOwner = true
```

Заданный ответ

Фейк

```
isOwner = false
```

Ответ по тестовым данным

С фейком тест на успешный переход упадёт

Мок: проверяем ожидания к вызову

```
ctrl := gomock.NewController(t)
storage := NewMockStorage(ctrl)

storage.EXPECT().
  IsUserFormOwner(int64(123456), int64(42)).
  Return(true).
  Times(1)
```

Создаём контроллер
и мок хранилища.
`t` — текущий тест.

Мок: проверяем ожидания к вызову

```
ctrl := gomock.NewController(t)
storage := NewMockStorage(ctrl)

storage.EXPECT().
  IsUserFormOwner(int64(123456), int64(42)).
  Return(true).
  Times(1)
```

Ожидаем метод с аргументами:
`userID` = 123456
`formID` = 42

Мок: проверяем ожидания к вызову

```
ctrl := gomock.NewController(t)
storage := NewMockStorage(ctrl)

storage.EXPECT().
  IsUserFormOwner(int64(123456), int64(42)).
  Return(true).
  Times(1)
```

Возвращаем `true`,
чтобы код продолжил работу.

Мок: проверяем ожидания к вызову

```
ctrl := gomock.NewController(t)
storage := NewMockStorage(ctrl)

storage.EXPECT().
  IsUserFormOwner(int64(123456), int64(42)).
  Return(true).
  Times(1)
```

Требуем ровно один вызов.

Другие аргументы, лишний или пропущенный вызов → тест падает.

Для этого перехода достаточно стаба

ЗАДАЁМ УСЛОВИЕ

Пользователь — владелец анкеты

```
{ "isOwner": true }
```

→

ПРОВЕРЯЕМ ПЕРЕХОД МЕЖДУ ЭКРАНАМИ

«Устранение течи»
↓
«Где проблема»

В ответе настоящего хэндлера

Моделировать хранилище владельцев не требуется.

Что команда получила?

≈30%

ручного тест-плана

Проверяем автоматически вместо ручного прохождения.

до 2 ч

экономии

На раскатку в прод: проверка основного сценария.

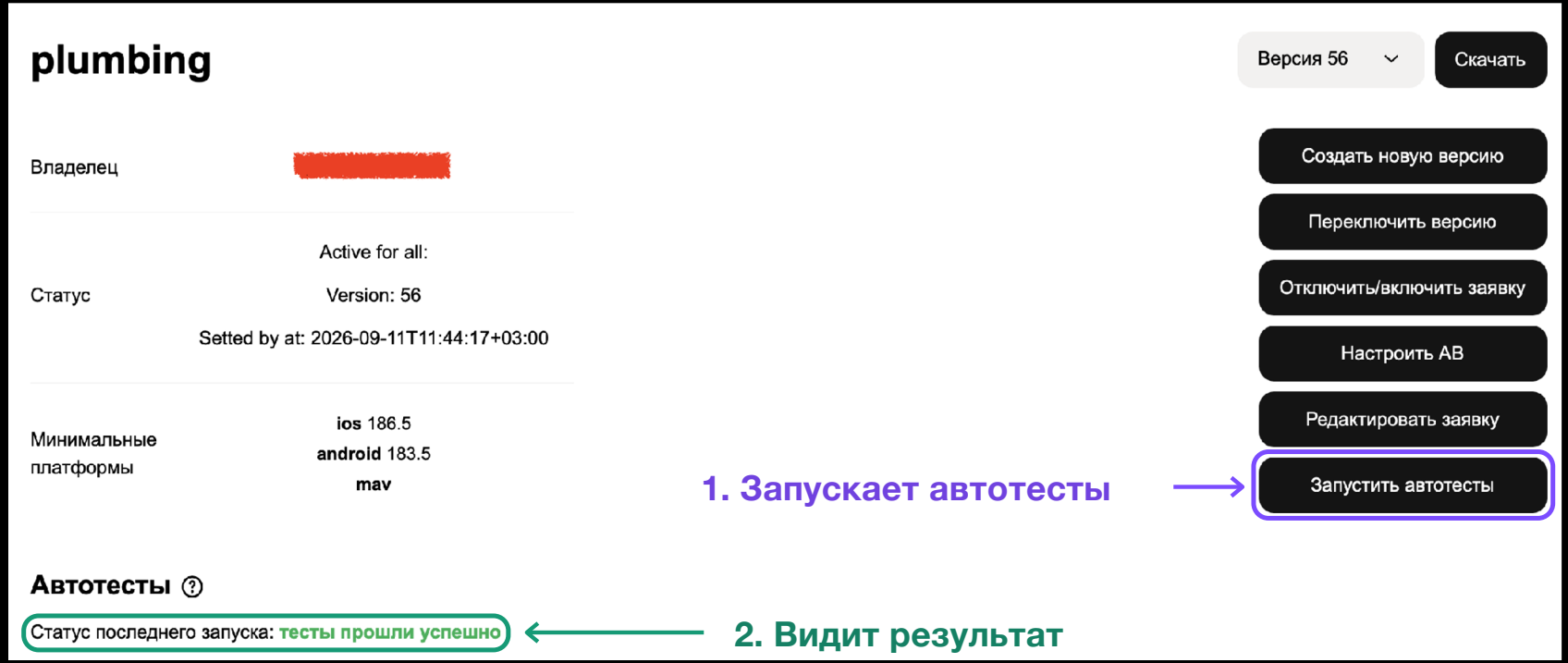
1

кнопка запуска

Авторы конфигов запускают автопроверки сами, без ожидания команды разработки.

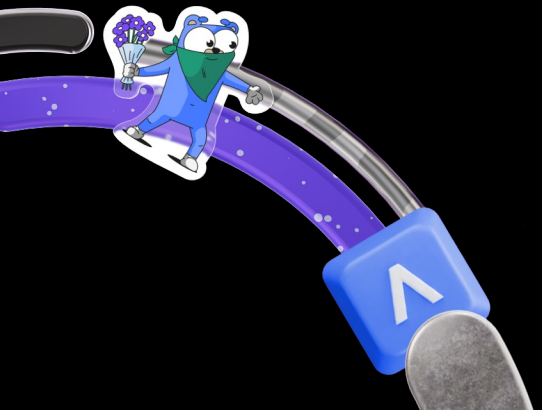
Автор конфига запускает тесты из админки

Запуск и статус последней проверки — на странице конфига



Что получаем

- ⊕ Прогон на проде в изоляции
- ⊕ Без тестовых пользователей и БД
- ⊕ Дедупликация внешних вызовов
- ⊕ Гибкая подмена зависимостей



Цена и границы

- ⊖ Поддержка генератора и тестовых хэндлеров
- ⊖ Подготовка и актуализация ответов
- ⊖ Не проверяем интеграции с подменёнными сервисами

Когда стоит вкладываться в такой подход

01

Изменения требуют широкого регресса.

02

Поведение продукта задают и код, и конфиг

03

Проверяем бизнес-логику

Ответы зависимостей можно задавать в тестах.

Экономия на регрессе должна оправдывать поддержку генератора, тестовых хэндлеров и подмен.

avito.tech

Продуктовый конфиг может стать источником тестовых сценариев

Константин Горностаев

QA инженер, Авито