

Ограничения покрытия тестами

Почему 100% покрытия кода не
означает проверенный продукт



О докладчике

Почему я полез в эту тему

Авенир Воронов

- CSCO в VeAI
- Разработка ИИ-агентов для JetBrains IDE
- Работал в HP, Motorola, Luxoft, Ерам, Корус



Классическая разработка



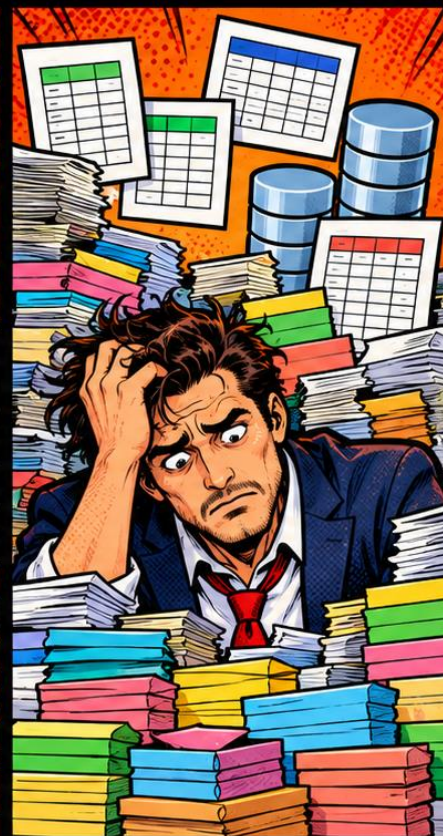
Стадии вайбкодинга



**Навайбкодил
MVP**



**Проверил
бизнес-логику**



**Обложился
тестовыми данными**



**Всё тормозит —
тестируем код!**

Классическая разработка



Как работают анализаторы

- **Покрытие в разных языках**
- Проблемы покрытия
- Что остается людям



Откуда берутся проценты

01

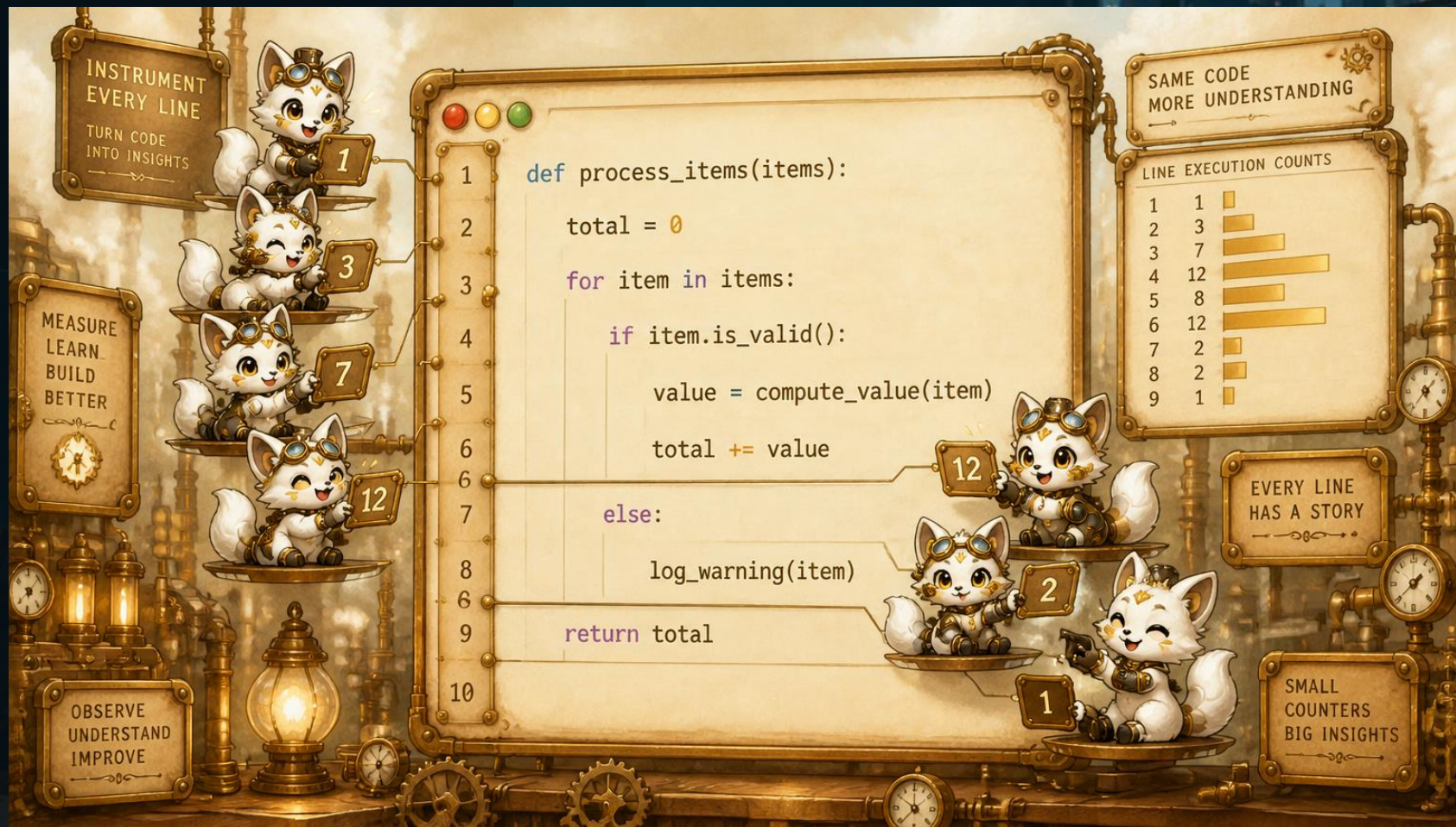
Вставить в код
невидимые счётчики

02

Запустить тесты

03

Прочитать счётчики



Разные языки дают разные точки вмешательства: машинный код, байткод, AST, callback интерпретатора.

gscov: покрытие кода C/C++

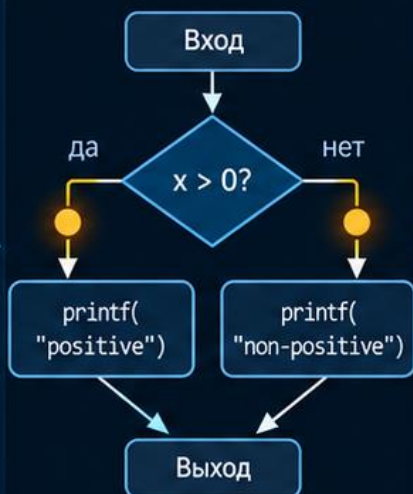
Исходный код (C)

myapp.c

```
1 #include <stdio.h>
2
3 int main(int argc, char** argv) {
4     int x = atoi(argv[1]);
5     if (x > 0) {
6         printf("positive\n");
7     } else {
8         printf("non-positive\n");
9     }
10
11     return 0;
12 }
```

GCC --coverage

Инструментация дуг управления при компиляции



Счетчики добавляются на выбранные дуги между базовыми блоками

Сборка с покрытием

Генерируется инструментированный исполняемый файл и служебный файл .gcno



myapp

Инструментированный исполняемый файл (с счетчиками)



myapp.gcno

Служебный файл с информацией о структуре программы (создается при компиляции)

Запуск программы

При выполнении собираются счетчики, результат сохраняется в .gcda



Запуск
./myapp 5



myapp.gcda

Файл с собранными счетчиками (создается при выполнении)

Отчет gscov

gscov читает .gcno и .gcda и показывает покрытие дуг и строк

```
1 #include <stdio.h>
2
3 int main(int argc, char** argv) {
4     int x = atoi(argv[1]);
5     if (x > 0) {
6         printf("positive\n");
7     } else {
8         printf("non-positive\n");
9     }
10
11     return 0;
12 }
```

- — дуга выполнена (есть переходы)
- — дуга не выполнена (переходов не было)

\$ gcc --coverage -o myapp myapp.c



Overhead 2-10%

Нужна отдельная сборка

gsov — машинный код

C/C++ · уровень компилятора

GCC при сборке добавляет подсчёт выбранных переходов между блоками и по ним вычисляет покрытие

Цена: нужна отдельная сборка с флагами покрытия

```
gcc --coverage -o myapp myapp.c
```

2–10%

OVERHEAD

НЮАНС :

```
if (vip) price -= 10;  
if (coupon) price -= 10;
```

Ветки проверили. Сочетание обоих факторов - нет.

JaCoCo: покрытие байткода JVM

Исходный код (Java)

MyService.java

```
1 public class MyService {
2     public int add(int a, int b) {
3         if (a < 0) {
4             return 0;
5         }
6         if (false) {
7             System.out.println(a + b);
8         }
9         int result = a + b;
10        return result;
11    }
12 }
```

Нет инструкций —
нет данных о покрытии

Компиляция

Байткод JVM

MyService.class

```
0:  aload_0
1:  iload_1
2:  ifge 8
5:  iconst_0
6:  ireturn
8:  iload_1
9:  iload_2
10: iadd
11: istore_3
12: iload_3
13: ireturn
```

JaCoCo (javaagent)

Перехватывает загрузку классов



- 1 Класс загружается в JVM
- 2 JaCoCo добавляет пробы покрытия в байткод в памяти
- 3 Модифицированный байткод выполняется в JVM

Байткод в памяти (с пробями JaCoCo)

```
0:  aload_0  ● ; probe
1:  iload_1
2:  ifge 8    ● ; probe
5:  iconst_0  ● ; probe
6:  ireturn   ● ; probe
8:  iload_1   ● ; probe
9:  iload_2
10: iadd      ● ; probe
11: istore_3  ● ; probe
12: iload_3
13: ireturn   ● ; probe
```

Выполнение и отчет о покрытии

Отчет (фрагмент)

```
— 1 public class MyService {
  2     public int add(int a, int b) {
  3         if (a < 0) {
  4             return 0;
  5         }
  6         if (false) {
  7             System.out.println(a + b);
  8         }
  9         int result = a + b;
  0         return result;
  1     }
  }
```

● — выполнена (покрыта)
● — не выполнена (не покрыта)
— — нет данных о покрытии
(нет соответствующего байткода)

\$ java -javaagent:jacoco.jar -jar myapp.jar



Overhead: обычно <10%

Без перекомпиляции

JaCoCo — байткод JVM

Java · уровень байткода

Java-агент перехватывает загрузку классов и переписывает байткод в памяти, вставляя инструкции пробы. Перекомпиляция не нужна.

Цена: dead code может исчезнуть ещё на уровне компилятора

```
java -javaagent:jacoco.jar -jar myapp.jar
```

<10%

OVERHEAD

НЮАНС:

интеграционный тест отмены заказа достигает нужной ветки, но не проверяет сумму возврата.

Istanbul: покрытие исходного JavaScript

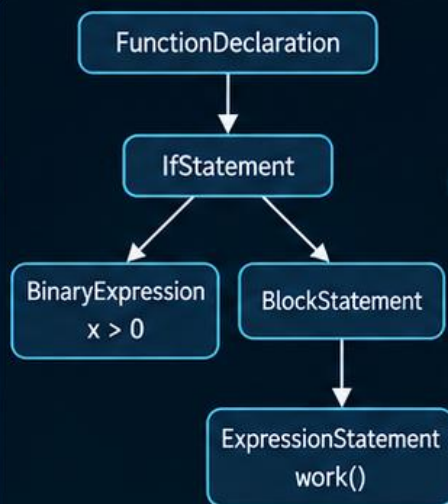
1. Исходный код (JavaScript)

Оригинальный исходный файл до выполнения

```
1 function check(x) {  
2   if (x > 0) {  
3     work();  
4   }  
5 }
```

2. Разбор в AST

Исходный код разбирается в абстрактное синтаксическое дерево (AST)



3. Вставка счётчиков

Код трансформируется: в ветви добавляются инкременты счётчиков покрытия

```
1 function check(x) {  
2   if (x > 0) {  
3     cov.b[0]++;  
4     work();  
5   }  
6 }
```

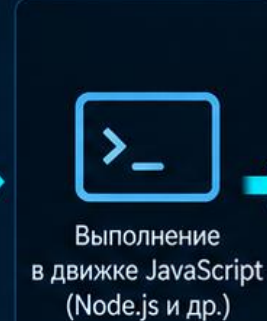
Дополнительные операции при выполнении



Счётчик ветви увеличивается каждый раз, когда выполняется эта ветвь.

4. Выполнение и отчёт

Трансформированный код выполняется обычным движком JavaScript. По завершении формируется отчёт о покрытии.



Отчёт о покрытии

```
1 ● function check(x) {  
2 ●   if (x > 0) {  
3 ●     cov.b[0]++;  
4 ●     work();  
5 ●   }  
6 ● }
```

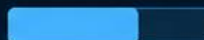
- — ветвь выполнена (покрыта)
- — ветвь не выполнена (не покрыта)



Накладные расходы во время выполнения

Из-за дополнительных операций (инкрементов счётчиков) выполнение замедляется по сравнению с обычным запуском.

Пример: +30–60%

Обычный запуск  10 с

С покрытием  13–16 с

Зависит от программы и замера.



Важно

- Трансформация кода (вставка счётчиков) происходит до выполнения.
- Замедление — результат дополнительных операций во время выполнения.
- Существуют и другие подходы к сбору покрытия в JavaScript.

Istanbul — исходный код

JavaScript · AST

Istanbul парсит файл в AST и вставляет вызовы счётчиков прямо в исходник. Для JavaScript это надёжный способ наблюдения перед JIT-компиляцией.

Цена:
каждый счётчик — вызов функции на JavaScript

30–60%

OVERHEAD

НЮАНС:

выполнена ветка отклонения промокода, но скидка осталась в сумме заказа.

coverage.py: события интерпретатора Python

Исходный код (Python)

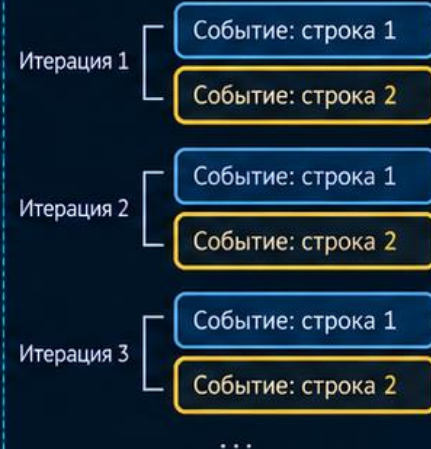
Обычная программа,
без изменений

```
1 for i in range(3):  
2     print(i)
```

Интерпретатор Python

Во время выполнения
генерирует события
для каждой выполняемой
строки

События выполнения строк



Сбор данных покрытия

coverage.py наблюдает
за событиями выполнения
строк и собирает информацию
о выполненных строках

`sys.settrace()`
(C tracer)

`sys.monitoring()`
(Python 3.12+)



coverage.py

Трассирующий
наблюдатель
(собирает данные
о выполненных строках)



События строк генерируются
много раз (например, в циклах),
поэтому сбор данных добавляет
время выполнения.

Файл данных

Собранная информация
сохраняется в файл
.coverage (SQLite)



.coverage

Файл SQLite
с данными покрытия
(создается
при выполнении)

Отчет о покрытии

coverage.py читает .coverage
и показывает, какие строки
были выполнены

```
1 ● for i in range(3):  
2 ●     print(i)
```

- — строка выполнена
(есть события)
- — строка не выполнена
(нет событий)

Пример влияния на время выполнения

Тот же код, разная конфигурация (конкретный пример)

Без покрытия:

10 с

CPython (пример):

+180% → 28 с

PyPy (пример):

+1300% → 140 с
(2 минуты 20 секунд)



Overhead +180...+1300%



Код программы не переписывается

Зависит от версии Python и нагрузки

Coverage.py — callback интерпретатора

Python · sys.settrace

Coverage.py не меняет код. Интерпретатор вызывает пользовательскую функцию перед каждой строкой; Coverage.py регистрирует callback и пишет номера строк.

Цена: на CPython заметно, на PyPy может стать экстремально дорого

180% / 1300%

OVERHEAD

НЮАНС:

тест прошёл обработчик пустого ответа API, но не проверил повторный ответ

Мнимая закономерность



Машинный код	gcov	2–10%
Байткод JVM	JaCoCo	<10%
Исходник (AST)	Istanbul	30–60%
Callback интерпретатора	Coverage.py	180–1300%

Точка наблюдения и стоимость зависят от языка, инструмента, версии и нагрузки; ни один из этих способов не проверяет ожидаемый результат

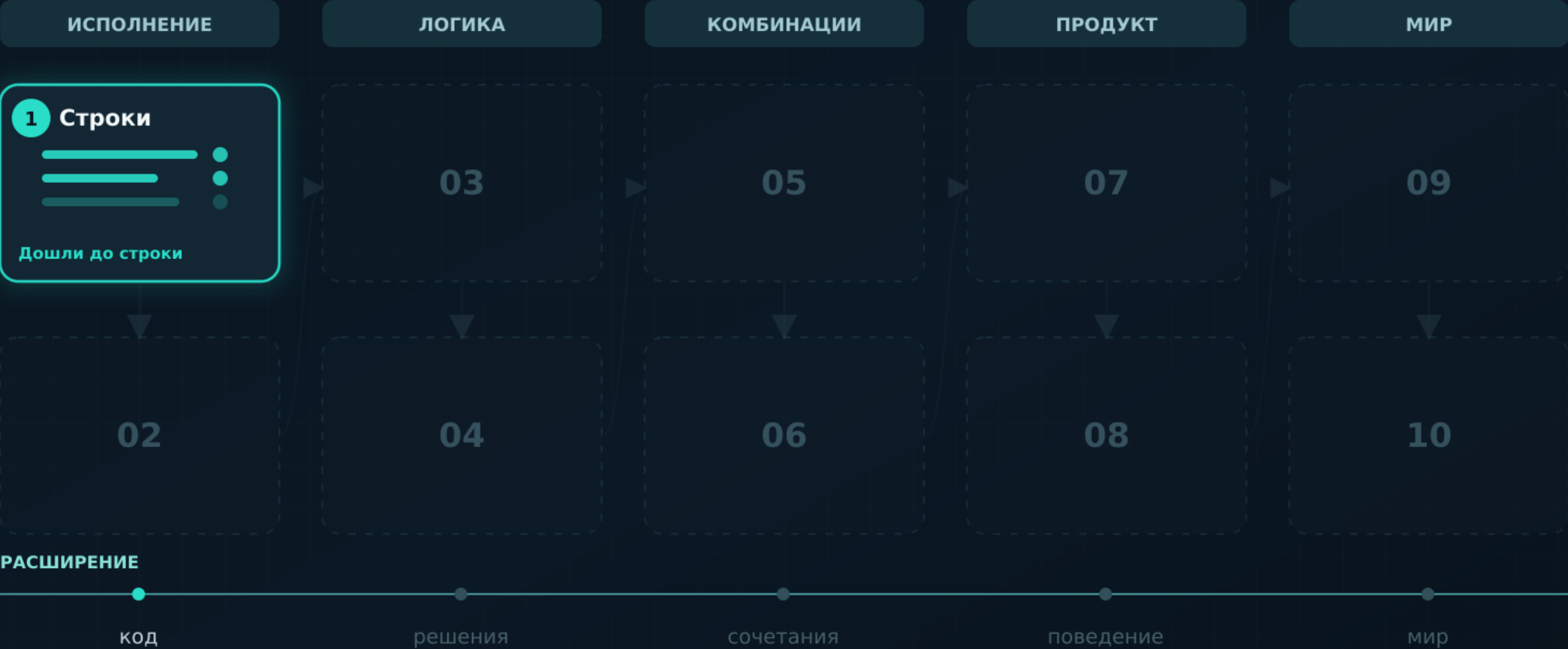
Как работают анализаторы

- Покрытие в разных языках
- **Проблемы покрытия**
- Что остается людям



КАК РАСТЁТ ПОЛЕ НАБЛЮДЕНИЯ

От исполнения строки — к правилам продукта и сигналам физического мира



ОТДЕЛЬНАЯ ОСЬ: меньше повторных наблюдений → ниже стоимость, но не шире покрытие

Проблема — измерение продолжается

после того, как ответ известен

Coverage.py: 180% overhead не потому, что счётчики медленные.

Счётчик щёлкает каждый раз. Строка выполнилась миллион раз? Миллион раз увеличили счётчик. Для ответа «покрыта или нет» достаточно единицы.

Это не баг Coverage.py — это фундаментальный недостаток классического подхода.

1,000,000

Выполнений

Счётчик щёлкает каждый раз

1

Достаточно

Для ответа "покрыто или нет?"

SlipCover — одноразовые зонды

инструмент измерения покрытия **строк и ветвей Python**, представленный на ISSTA 2023



180%

классический overhead

5%

so SlipCover

Скорость появилась. Но полнота всё ещё ограничена строками и ветками.

CoverUp (FSE 2025)

ЗАМЫКАНИЕ ЦИКЛА

1 SlipCover → непокрытые строки

2 AST → минимальный сегмент

3 LLM: «напиши тест»

4 Запуск теста

5 Покрытие выросло → сохранить

6 Нет → feedback с ошибкой

80%

медианное покрытие

47%

у конкурентов

Intel Processor Trace (Intel PT)

аппаратное решение той же проблемы

Процессор записывает ветвления в аппаратный буфер — без инструментирования исходника.

- работает с бинарниками без исходников
- не нужна перекомпиляция
- минусы: Intel, Linux, сложность настройки

SlipCover

программный

5%

Intel PT

аппаратный

≈5%

```
gcc -O0 -g -o hello hello.c
perf record -e intel_pt//u -- ./hello
perf script --itrace=i0ns -F time,sym,srcline,ip
```

Оба измеряют поток управления — не данные и не условия.

Towards Refined Code Coverage (TU Delft 2025)

предсказание без запуска

Random Forest классификатор:

«Будет ли строка покрыта тестами?»

Признаки: CBO (Coupling Between Objects), fanin, цикломатическая сложность, тип узла AST, позиция в файле.

До: запустить тесты → увидеть пробелы →
написать тесты

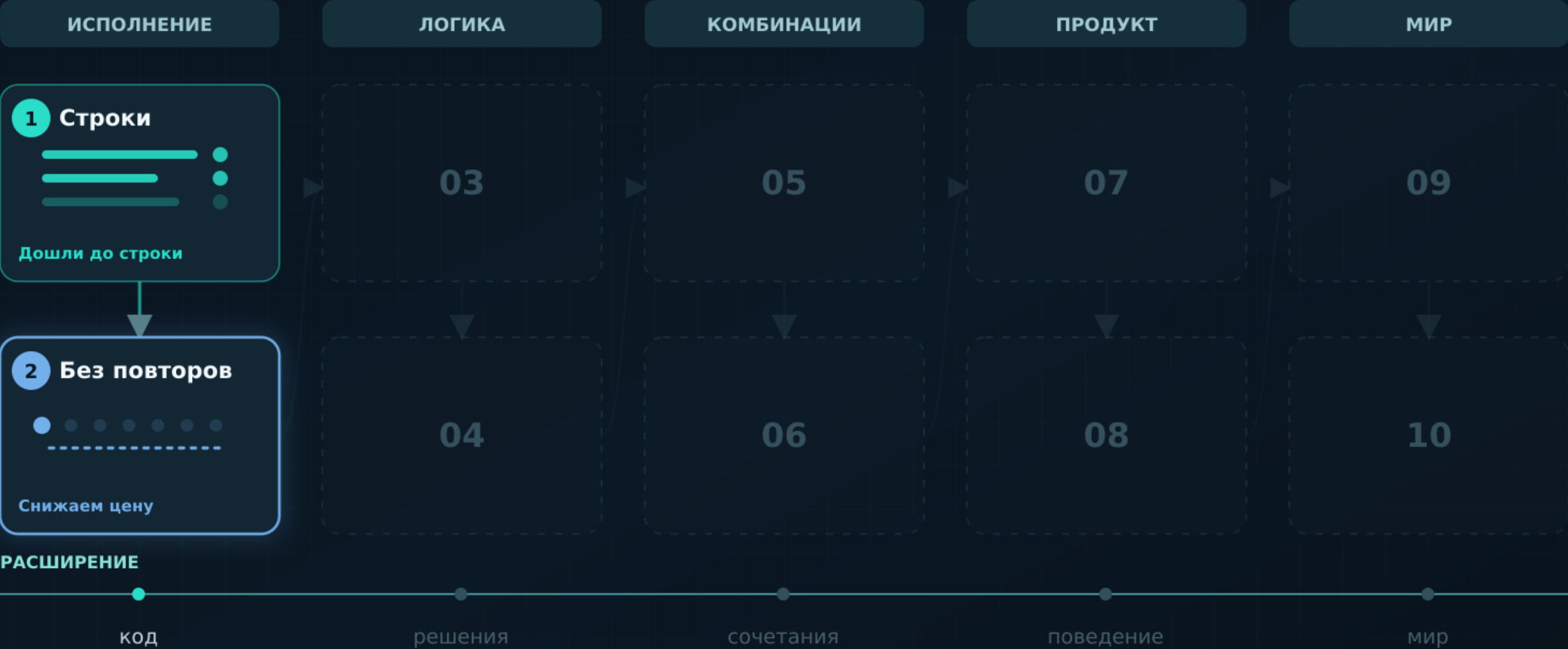
После: посмотреть на код → предсказать
пробелы → написать тесты заранее

90.2%

ТОЧНОСТЬ

КАК РАСТЁТ ПОЛЕ НАБЛЮДЕНИЯ

От исполнения строки — к правилам продукта и сигналам физического мира



ОТДЕЛЬНАЯ ОСЬ: меньше повторных наблюдений → ниже стоимость, но не шире покрытие

Что мы измеряем

строки, ветки, функции — и где этого мало

если (пользователь_админ И включена_2FA)

→ пустить в систему

иначе

→ отказать

Branch coverage говорит:

Пустили хотя бы раз? Да.

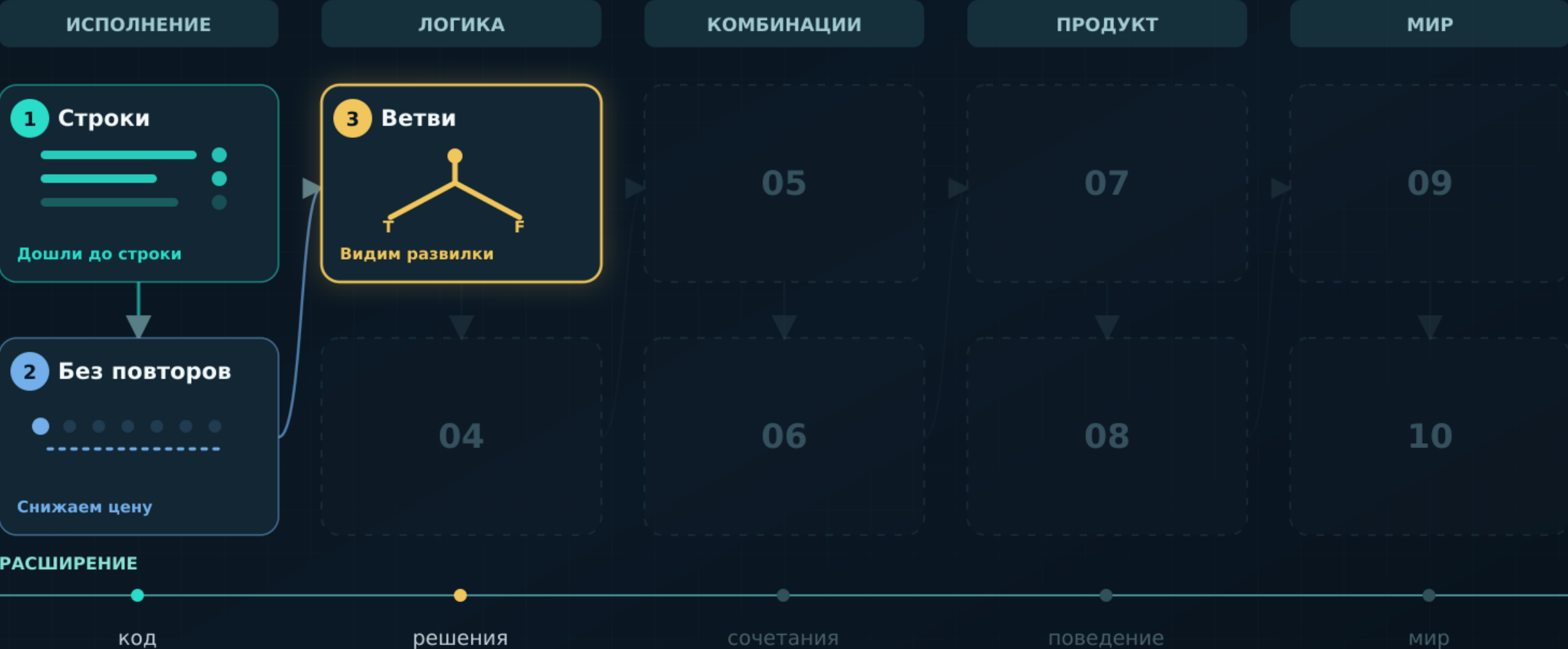
Отказали хотя бы раз? Да.

Вывод: 100% покрытия

**Но если проверка 2FA забыта, тест с гостем всё равно даст отказ.
Branch coverage зелёный. Баг живёт.**

КАК РАСТЁТ ПОЛЕ НАБЛЮДЕНИЯ

От исполнения строки — к правилам продукта и сигналам физического мира



ОТДЕЛЬНАЯ ОСЬ: меньше повторных наблюдений → ниже стоимость, но не шире покрытие

MC/DC (Modified Condition/Decision Coverage)

DO-178B/C

стандарт для авиации

каждое условие должно независимо влиять на результат

Для каждого условия внутри решения нужно:

- условие принимало true и false
- изменение только этого условия меняло итог решения
- в базовом режиме gcov, JaCoCo, Istanbul, Coverage.py это не измеряют

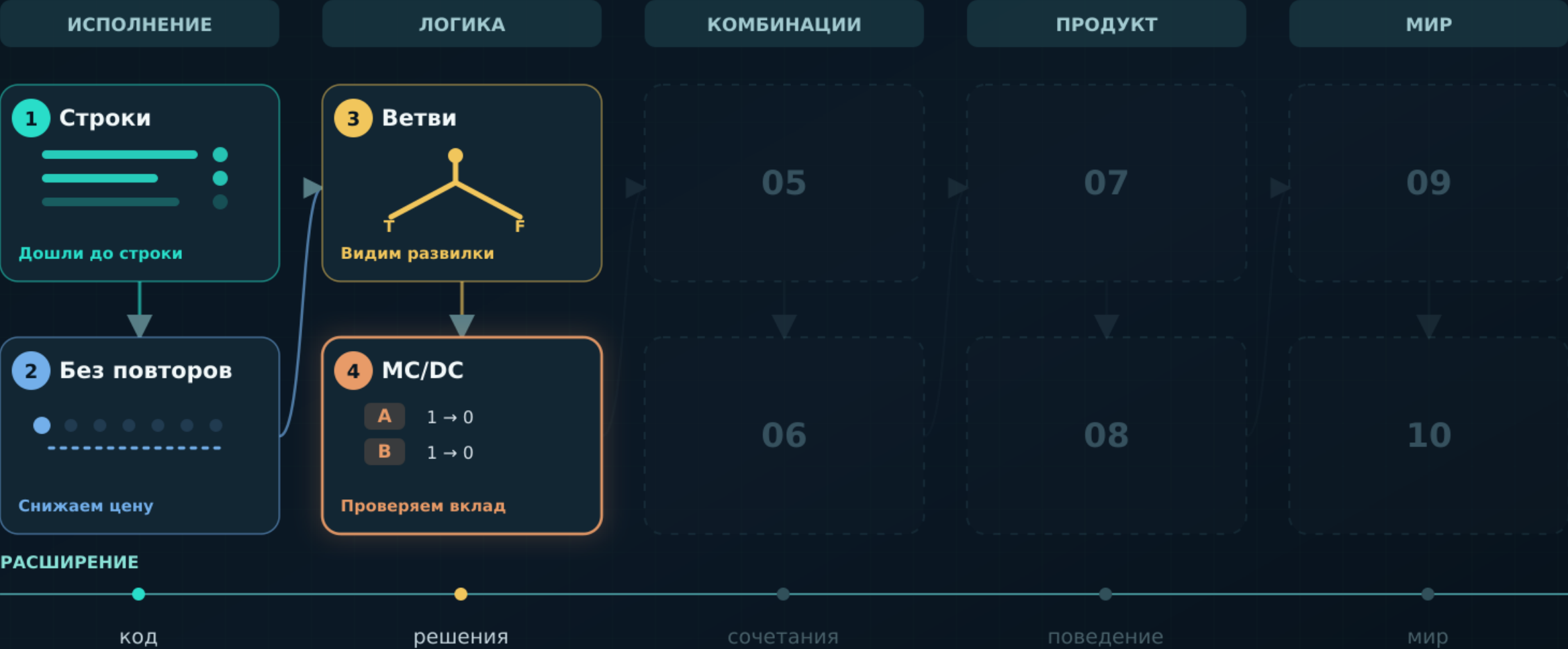
```
если (возраст >= 18) { balance -= 100; }  
если (доход >= 50000) { balance -= 100; }
```

Решение	Условие	Нужно проверить
if 1	возраст >= 18	true и false
if 2	доход >= 50000	true и false

Граница: MC/DC не требует комбинировать разные if.
Комбинации между if — это уже path profiling.

КАК РАСТЁТ ПОЛЕ НАБЛЮДЕНИЯ

От исполнения строки — к правилам продукта и сигналам физического мира



ОТДЕЛЬНАЯ ОСЬ: меньше повторных наблюдений → ниже стоимость, но не шире покрытие

Path profiling

не какие ветки, а каким путём прошла программа

Два независимых if дают четыре пути.

МС/ДС смотрит на условия по отдельности.

Но баг может жить в комбинации: скидка есть, доставки нет.

Path profiling нумерует пути от входа до выхода функции.

Доставка? Нет

Доставка не нужна

Доставка? Да

Добавить стоимость доставки

Скидка? Нет

Без скидки

Path explosion

когда путей слишком много

1024

2^{10} путей

1M

2^{20} путей

1B

2^{30} путей

Решения: прореживание, гибридный режим, адаптивная гранулярность.

MC/DC становится «золотой серединой» между branch coverage и path coverage.

Ball-Larus / BLFuzz (Binary Level Fuzzing)

path profiling на практике

BLFuzz — это инструмент для фаззинг-тестирования (тестирования с помощью случайных или мутационных данных)



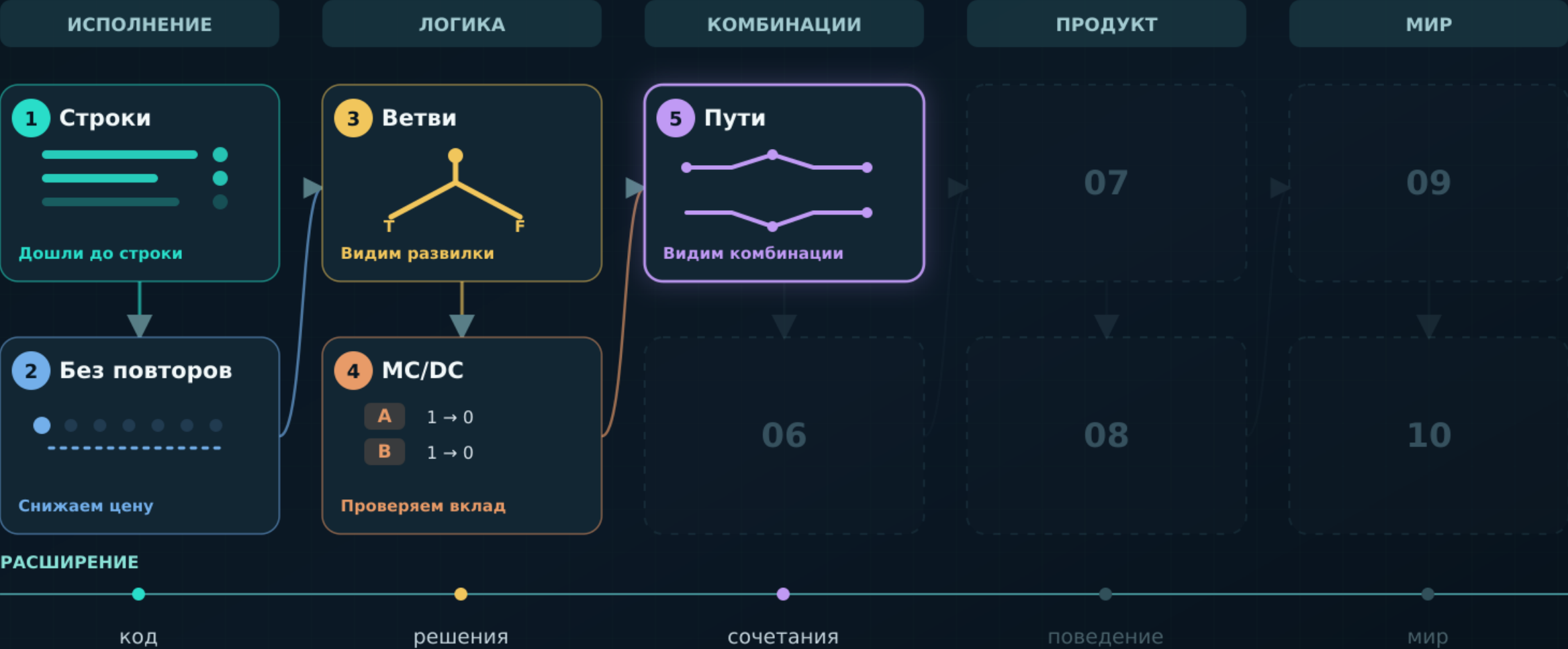
В конце функции `path_id` однозначно показывает, каким путём прошли. Overhead сопоставим с обычным `edge coverage`.

+27%

уникальных багов vs AFL++

КАК РАСТЁТ ПОЛЕ НАБЛЮДЕНИЯ

От исполнения строки — к правилам продукта и сигналам физического мира



ОТДЕЛЬНАЯ ОСЬ: меньше повторных наблюдений → ниже стоимость, но не шире покрытие

Data Coverage (USENIX Security 2024)

за пределами потока управления

```
if (promocode == 'HALYAVA')  
    return 20;
```

Branch и path coverage видят путь.
Но не знают, насколько вход близок к правильному значению.

HALYAVO

совпали 6 из 7 байт

Фаззер помечает такой тест-кейс как интересный и мутирует именно его.

98.31

из 100 на Google FuzzBench

Конвергенция инструментов

ОТ ОТДЕЛЬНЫХ ИССЛЕДОВАНИЙ К ЕДИНОМУ ЦИКЛУ

01

Предсказывает
пробелы до
запуска

02

Генерирует
тесты через LLM

03

Измеряет с
малым overhead

04

Видит пути, не
только ветки

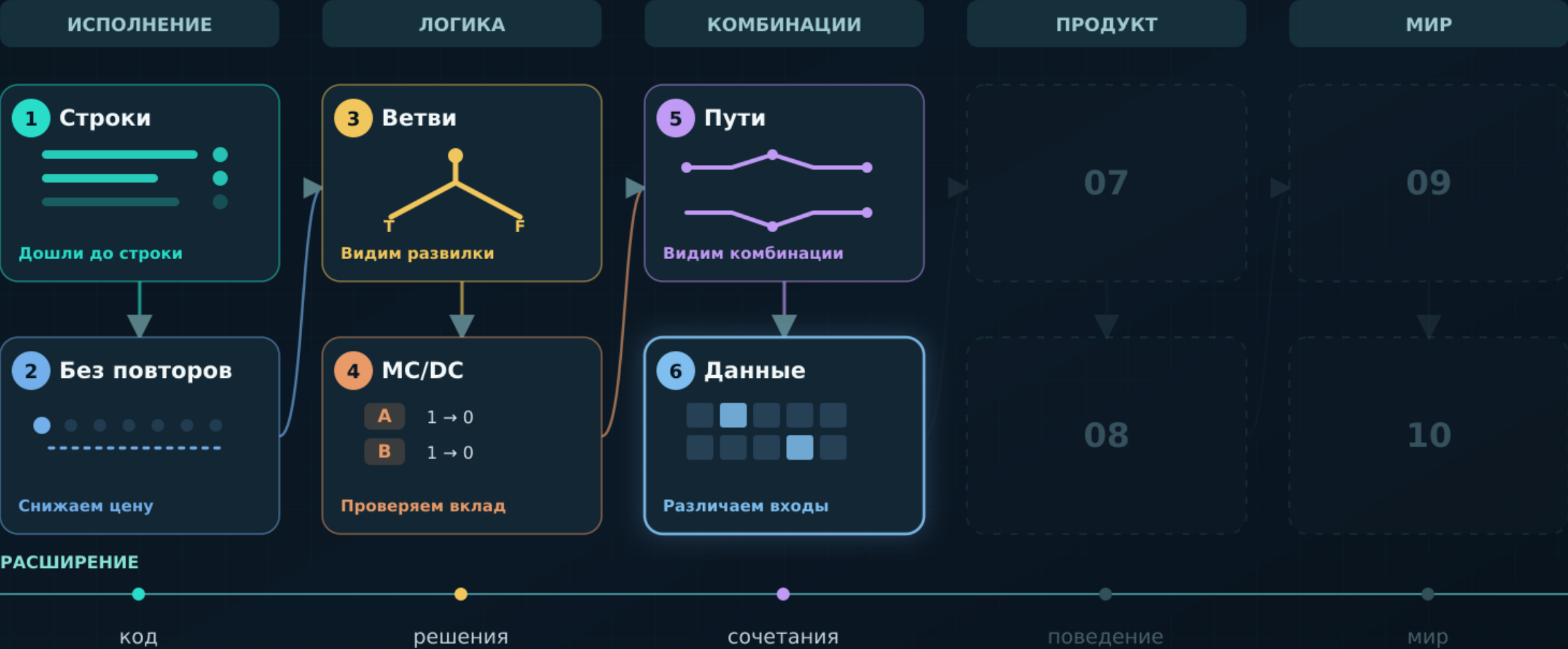
05

Отслеживает
данные, не
только код

**Человек пишет тесты для бизнес-логики.
Покрытие становится автоматической задачей.**

КАК РАСТЁТ ПОЛЕ НАБЛЮДЕНИЯ

От исполнения строки — к правилам продукта и сигналам физического мира



ОТДЕЛЬНАЯ ОСЬ: меньше повторных наблюдений → ниже стоимость, но не шире покрытие

FOX (CCS 2024)

направленный поиск покрытия

Не новый тип покрытия, а умный поиск.

- Scheduler оценивает frontier branch
- Mutator делает направленные мутации
- Особенно полезно на длинных цепочках проверок

+26%

покрытия vs AFL++

КАК РАСТЁТ ПОЛЕ НАБЛЮДЕНИЯ

От исполнения строки — к правилам продукта и сигналам физического мира



ОТДЕЛЬНАЯ ОСЬ: меньше повторных наблюдений → ниже стоимость, но не шире покрытие

Как работают анализаторы

- Покрытие в разных языках
- Проблемы покрытия
- **Что остается людям**



Business coverage

что за пределами кода?

Code coverage

«строка 42 выполнялась»

Business coverage

«возврат товара от VIP-клиента
с частичной оплатой сертификатом»

Тесты мапятся на бизнес-процессы, а не на строки.

Governance coverage

регуляторные требования вместо строк

Авиация
DO-178B/C

Медицина
IEC 62304

Финансы
PCI-DSS

«Каждый путь к отмене транзакции должен быть аудируем» — это не про строки.

Domain coverage

какие области продукта реально покрыты

90%

Оплата

5%

Возврат

0%

Многовалютность

100% line coverage по возврату не отменяет того, что домен может быть почти не проверен.

КАК РАСТЁТ ПОЛЕ НАБЛЮДЕНИЯ

От исполнения строки — к правилам продукта и сигналам физического мира



ОТДЕЛЬНАЯ ОСЬ: меньше повторных наблюдений → ниже стоимость, но не шире покрытие

За пределами текста

урок от робототехники

Проблема анализаторов покрытия — частный случай общей проблемы:

собираемого контекста недостаточно, чтобы понять, что реально происходит.

DROID

UMI

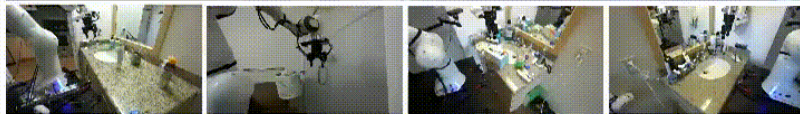
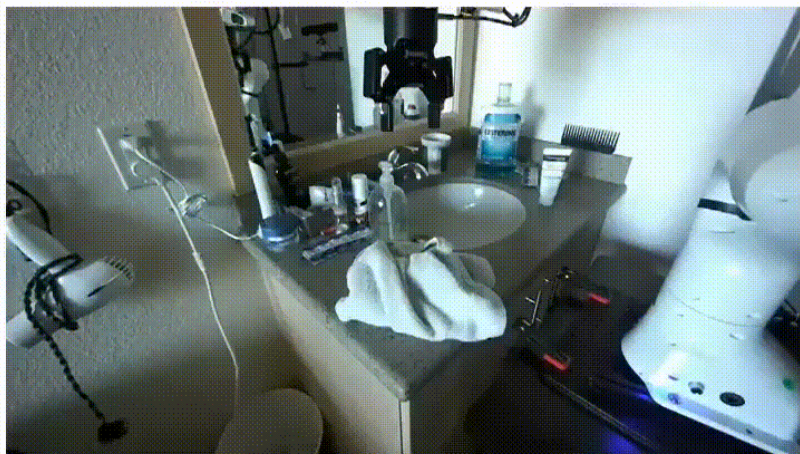
OSMO



Droid (2024)

Сбор данных для обучения роботов

Bathroom



DROID

Distributed Robot
Interaction Dataset



76k Episodes



564 Scenes



52 Buildings



13 Institutions

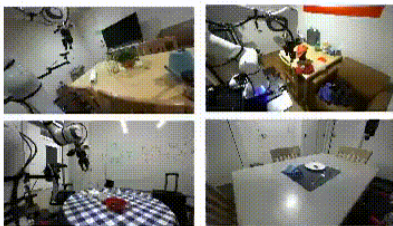


86 Tasks / Verbs

Kitchen



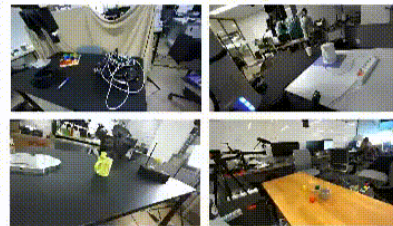
Dining Room



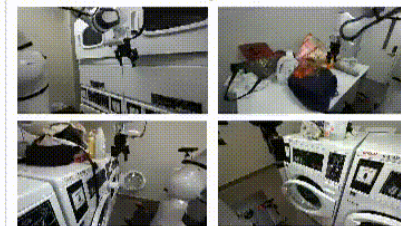
Bedroom



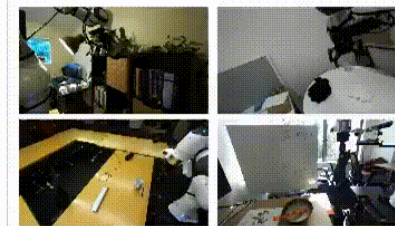
Laboratory



Laundry Room



Office



UMI (2024)

Коллаборативное обучение

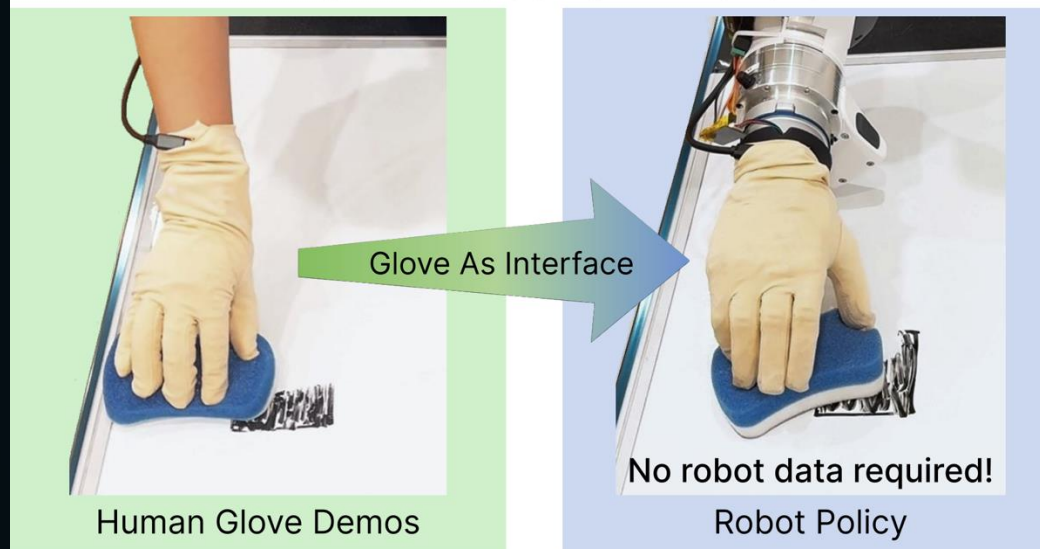


OSMO (2025)

Сенсорные данные



Human-to-Robot Skill Transfer



Три этапа робототехники

и точная аналогия с покрытием кода

DROID

**«нужно больше
данных»**

line/branch coverage:
количество как прокси
качества

UMI

**«данные можно
собирать дешевле»**

CoverUp и LLM: дорогой труд
заменяется машинным

OSMO

**«проблема в отсутствующей
модальности»**

Data / Business / Governance:
измеряем не то

КАК РАСТЁТ ПОЛЕ НАБЛЮДЕНИЯ

От исполнения строки — к правилам продукта и сигналам физического мира



ОТДЕЛЬНАЯ ОСЬ: меньше повторных наблюдений → ниже стоимость, но не шире покрытие

Как работают анализаторы

- **Покрытие в разных языках**
- **Как оптимизируют скорость**
- **Покрытие за пределами кода**



Классическая разработка



Практический вывод

Покрытие — иерархия, не одна цифра. 100%
line coverage не значит «багов нет».

Посмотрите на отчёт: какой это уровень? Есть
ли MC/DC?

Посмотрите за пределы кода: какие
бизнес-сценарии реально проверены?

Code coverage без business coverage — робот с
камерой, но без осязания.

Кто начнёт измерять контекст сейчас — будет на два года впереди.