Software Testing

Oscar Chaparro

Worked at 8 hours, failed at 100

1991 Gulf War: a US **Patriot** battery misses an Iraqi **Scud**. **28 soldiers die.**

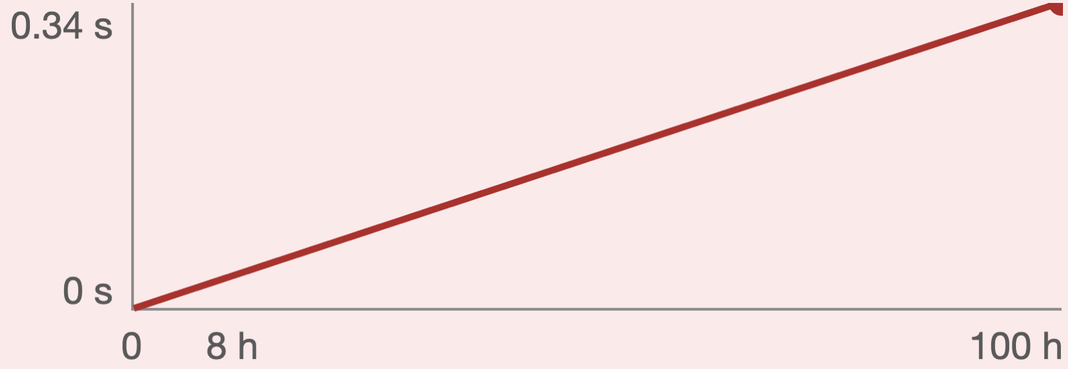
Timeline, 1991



Fix released Feb 16 · **arrived Feb 26, the day after**

Clock error vs. hours running

Clock counted **tenths of a second**; 0.1 **truncated** to 24 bits → each tick slightly short



After **100+ hours**: clock **0.34 s** off → tracking window **687 m** off

Why software is hard to test

Bridge



8 h ✓ ⇒ 100 h ✓

a static load doesn't drift: if it holds for 8 hours, it holds for 100

Software



8 h ✓ ⇏ 100 h ?

state builds up over time: 8 hours says nothing about 100

Small changes in inputs, time, or environment → very different behavior

"Program testing can be used to show the presence of bugs, but never to show their absence!" — E. W. Dijkstra, "The Humble Programmer," Communications of the ACM 15(10), 1972

What else makes testing hard

Difficulty	Where we cover it
Too many inputs: you can’t try them all	**Designing tests**
Knowing the right answer: the oracle problem	**Test oracles**
Hidden dependencies: databases, services, clocks are slow or hard to control	**Test doubles**
State and environment: time, order, machine change the result	**Suite health**
Change: every edit can break something far away	**Regression testing**

Agenda

1. Foundations
2. Designing tests from the specification
3. Coverage
4. Test oracles in practice
5. Mutation testing
6. Test doubles and integration
7. Keeping a test suite healthy
8. Testing in practice: what to run, who, and where

Foundations

What a test is, and what it checks

Fault, error, failure

```
function double(x: number) {
  return x * x;
}
```

Fault (bug)

the wrong code

`*` instead of `+`

→

Error

the wrong internal state it creates

`double(3)` computes `9`

→

Failure

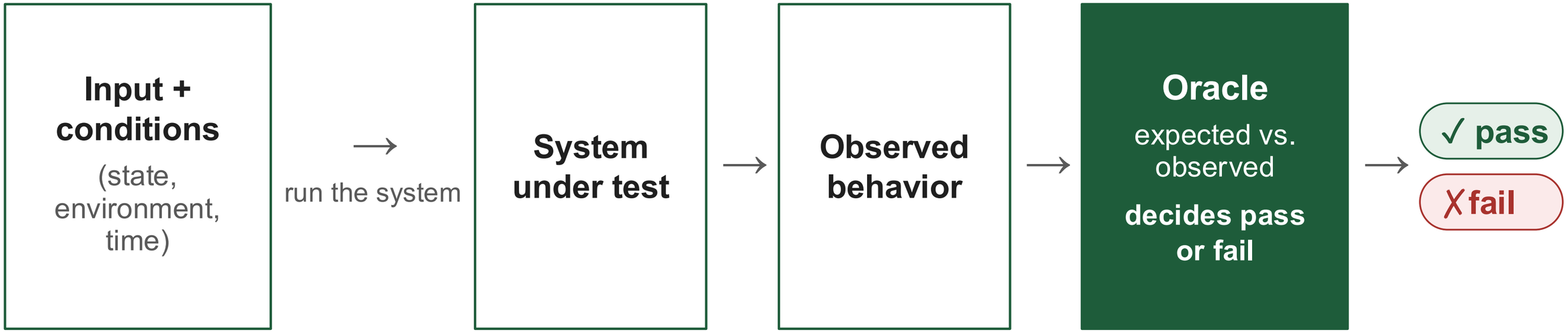
the wrong behavior a user can see

`screen shows 9, expected 6`

A fault can sit unnoticed until some input turns it into a failure

`double(2) = 4` — correct by luck.

What a test is



No oracle, no test — just a demo

Test levels

more realistic, slower, costlier

Level	Checks	Built from	Typical tools
Unit	one function or class	code + module spec	Vitest, Jest, go test
Integration	parts working together	interface specs	HTTP tests, real database
System / E2E	the whole app, as a user	requirements	Playwright, Cypress
Acceptance	does it meet the need?	acceptance criteria	users, product owners

Levels describe **scope**. At any level you can also test **non-functional** qualities: performance · load · endurance (long runs) · security · accessibility



What each level checks

Unit

- return values
- thrown errors
- state changes
- log output

Integration

- messages between parts
- database contents
- error codes
- contracts and schemas

E2E (end-to-end)

- screens and transitions
- data shown
- final outcomes
- backend state

The higher the level, the more there is to observe, and the more ways a check can be too weak or too brittle

What makes testing good

Effective

finds real faults

How to tell: break the code on purpose; does a test fail?

Repeatable

same input, same result — every time

How to tell: rerun 5 times; does any test flip?

Automated

cheap to run often

How to tell: does CI run it on every push?

Systematic

inputs chosen on purpose, not by luck

How to tell: can you name what each test covers?

Which of these does your project’s test suite fail today?

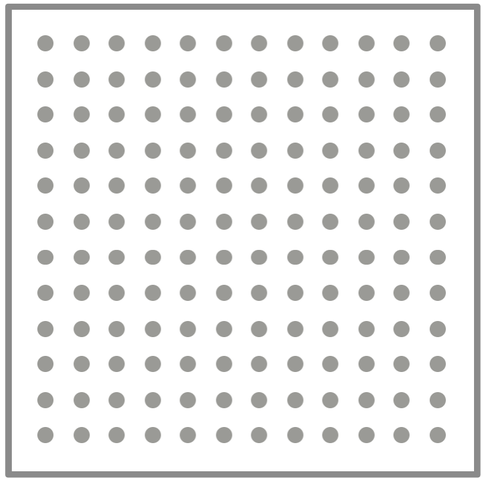
Designing tests from the specification

Black-box test design

You can’t test every input

3 inputs × 32 bits each → $(2^{32})^3 = 2^{96} \approx \mathbf{8 \times 10^{28}}$ combinations

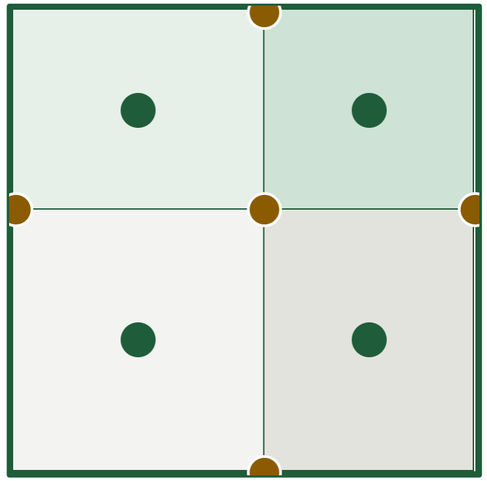
At a billion tests per second: about **2.5 trillion years**



all inputs

→

partition



groups

- **Group inputs that should behave the same**
- **Test one from each group — and the edges between groups**

Equivalence classes



nextDate(month, day, year), year 1812–2050 → the date of the next day

Month

- 30-day
- 31-day
- February

Day

- 1–28
- 29
- 30
- 31

Year

- common
- leap (÷4)
- 1900 (÷100)
- 2000 (÷400)

invalid: < 1812 · > 2050

Weak vs. strong combinations

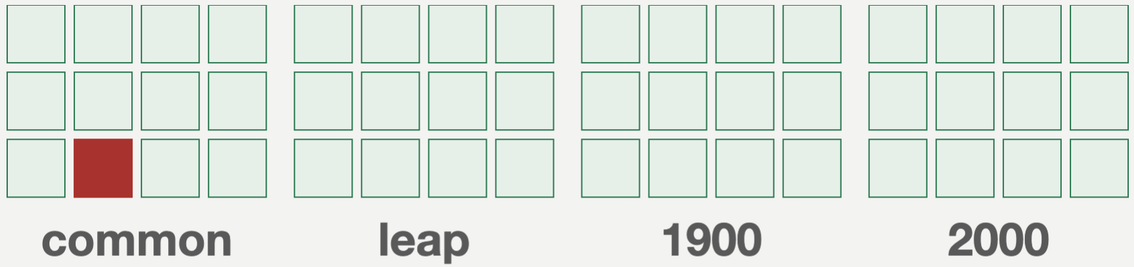
Weak

every class appears at least once → **4 tests**

Test	Input	Expected
W1	6/14/1900	6/15/1900
W2	7/29/1912	7/30/1912
W3	2/30/1913	invalid
W4	6/31/2000	invalid

Strong

every combination of classes → **48 tests**



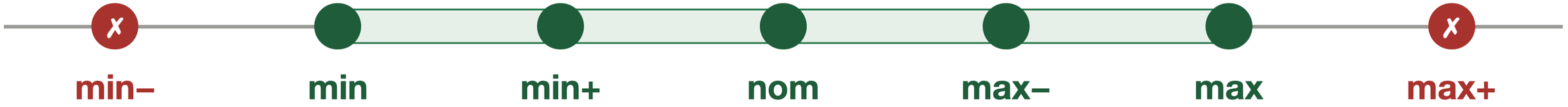
rows: month · columns: day

✗ Feb · 29 · common year

Weak is cheap; **strong** finds faults that need two inputs together (Feb 29 in a common year)

Boundary values

Faults cluster at the **edges** of classes



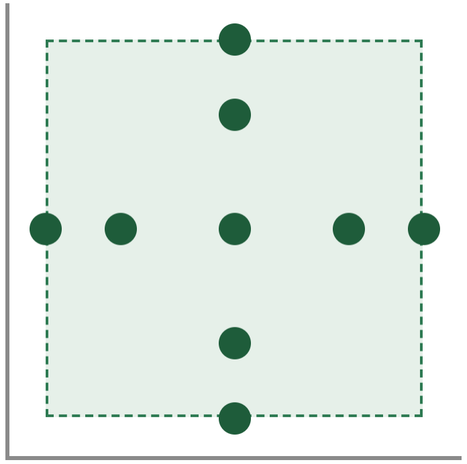
Normal: min · min+ · nominal · max− · max

Robust: also min− and max+ (invalid)

Two inputs: normal vs. worst case

Normal

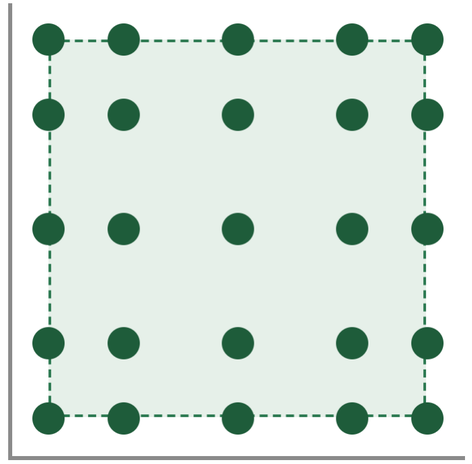
vary one input at a time



9 tests (4n + 1)

Worst case

every combination



25 tests (5^n)

With 5 inputs: normal = **21** tests · worst case = **3,125**. Use worst case when failures are costly.

Boundaries for nextDate

	min	min+	nom	max−	max
month	1	2	6	11	12
day	1	2	15	30	31
year	1812	1813	1931	2049	2050

Normal (13 tests)

`6/31/1931` → **invalid**

`12/15/1931` → **12/16/1931**

Worst case only (125 tests)

`12/31/1812` → **1/1/1813**

`12/31/2050` → **out of range**

Boundaries of the range miss 1900 and 2000: combine with equivalence classes

December 31, 2008



Dec 31, 2008: every **Zune 30 GB** (Microsoft's music player) froze on startup, all at once

The clock code counted days in the year. On **day 366** of a leap year, a loop never never ended

Microsoft's advice: wait until Jan 1 and restart

The boundary nobody tested: the last day of a leap year

Source: Microsoft via NBC News, 2009

Outline: boundaries in practice

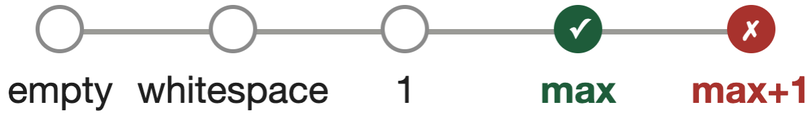
Archive a document with an optional **reason**, up to a maximum length

server/ routes/ api/ documents/ documents.test.ts

```
it("should allow a reason at the maximum length",
  async () => {
    const reason = "a".repeat(
    DocumentValidation.maxArchivedReasonLength);
    …
  });
it("should reject a reason over the maximum length",
  async () => {
    const reason = "a".repeat(
    DocumentValidation.maxArchivedReasonLength + 1);
    …
  });
```

TESTS IN THE REPO

- no reason
- empty
- only whitespace
- trims spaces
- **exactly max**
- **max + 1** rejected
- child documents
- restore clears it



The limit comes from a **constant**, not a typed number

Medusa: dates have edges too

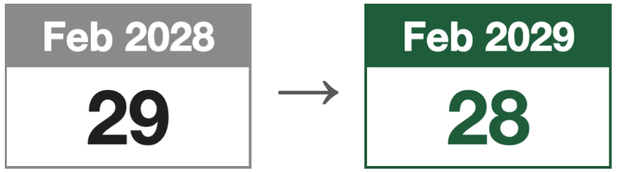
Subscriptions renew weekly, monthly, or yearly

TESTS IN THE REPO

```
describe("computeNextBillingDate", () => {
  it("adds 7 days")
  it("adds one month")
  it("clamps to the end of a shorter month")
  it("rolls over to next year")
  it("clamps a yearly interval from a leap day")
  it("doesn't change the input")
});
```



monthly: **no Feb 31** → last day of Feb



yearly: **no Feb 29, 2029** → Feb 28

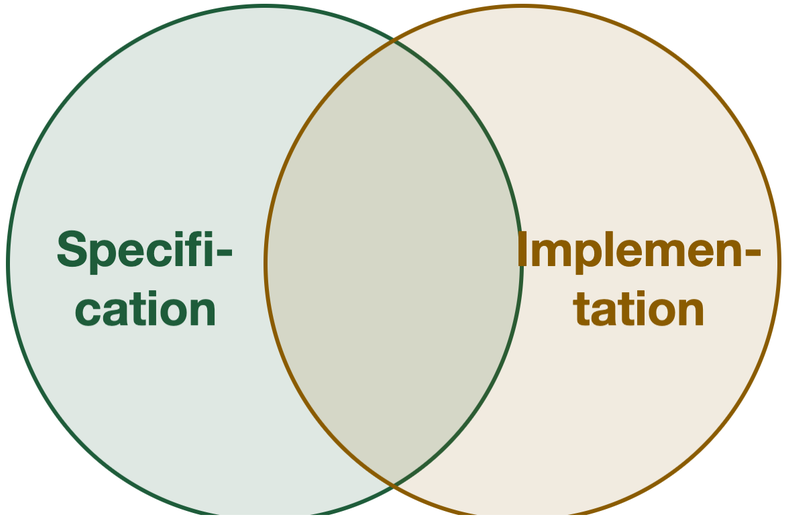
Other boundaries to test: a monthly plan starting **Feb 29** · a plan from **Jan 31** : after Feb 28, does it renew on on Mar 28 or Mar 31?

What each approach can’t see

Specified, not implemented

→ only **black-box** finds it

e.g., the spec says refunds over $500 need approval; the code never checks



Implemented, not specified

→ only **white-box** finds it

e.g., a debug ?admin=1 flag left in the code

Specification = what the system should do (requirements, acceptance criteria, API docs)

Black-box (from the specification): can’t see code that does something the spec **never asked for**

White-box (from the code): can’t see something the specification asked for but **is missing**

Use both

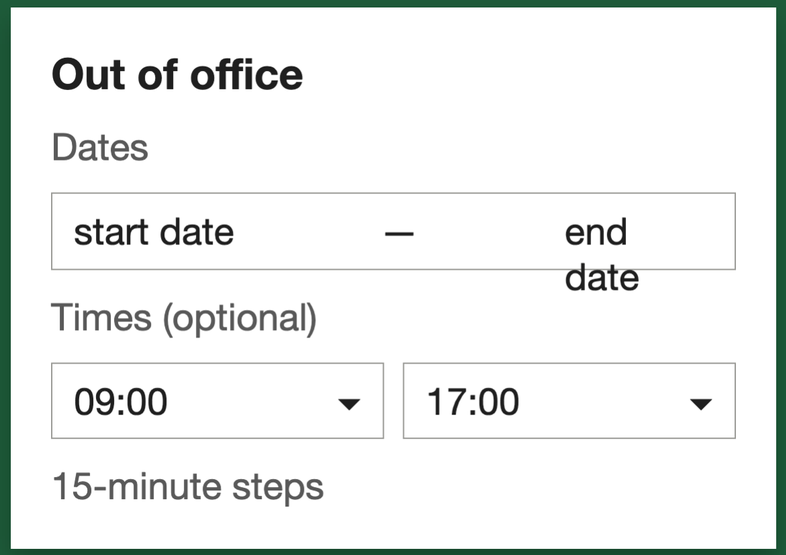
Exercise: an out-of-office window

Cal.diy: a host marks time as “out of office” so no one can book it

Inputs: start date · end date · start time · end time (times are optional: no times = whole days)

Rules: the end must come after the start · times are “HH:mm” · an end time of 23:59 means end of day

Your task: list equivalence classes and boundary values. Which **combinations** matter?



An out-of-office window — answers

Input	Classes	Boundaries
Times	absent (whole days) · present	00:00 · 23:59 ✓ · 24:00 ✗ · “9:00” ✗
Same day	end after start ✓ · end before start ✗	**end = start** ✗ · start + 1 minute ✓
Different days	end date after start date ✓ · before ✗	window across midnight ✓ (23:00 → 02:00)
End time	ordinary · 23:59 (end of day)	23:58 vs. 23:59

same day × time order is where the faults are

Cal.diy’s tests cover: end before start · end = start · 23:59 · past midnight. **Not in the handler tests:** time format, which the schema checks instead.

Also worth testing: end date = start date · only one time given · DST days · overlapping windows

Can tools find the classes for you?

Tools can’t read intent. **People choose the classes from the spec**; tools help with the rest.

You list classes and boundary values → the tool picks combinations

PICT | ACTS

you map each row to real inputs and the expected result

The tool reads the code → finds its boundaries

EvoSuite | Pynguin | KLEE

every `if` becomes a boundary

You give a rule → the tool searches for inputs that break it

fast-check | fuzzers

tries thousands of inputs, biased toward edges (0, empty, max)

Code-based tools can’t find missing classes: they’d never propose day 366 unless the code handles it

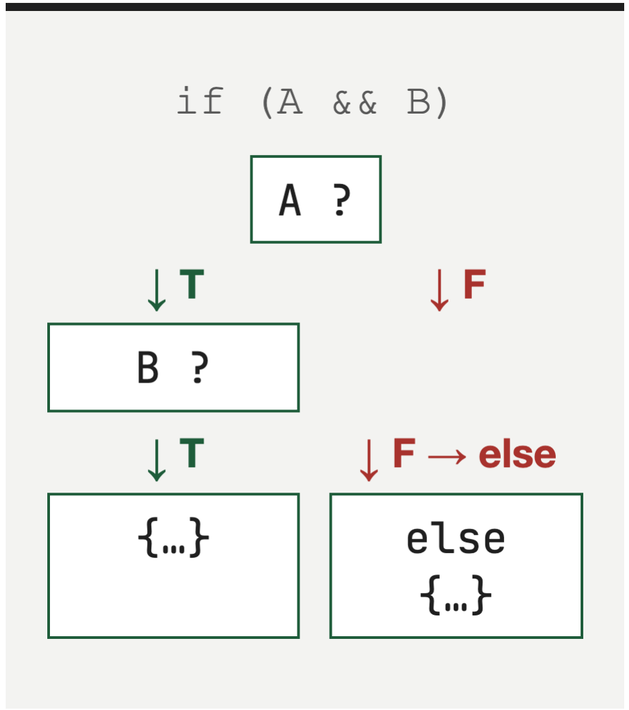
Coverage

What ran — and what that doesn’t tell you

Coverage criteria

stronger, costlier

Criterion	Every … executed at least once
Statement	line
Branch	true and false side of each decision
Condition	true and false value of each part (A, B in A && B)
MC/DC	each part shown to change the outcome on its own
Path	route through the code (usually infinite with loops)



Apple’s goto fail (2014)

sslKeyExchange.c, Apple Secure Transport

```
if ((err = SSLHashSHA1.update(&hashCtx, &signedParams)) != 0)
    goto fail;
    goto fail;
…
err = sslRawVerify(…);
```

always runs

never runs

Checks that the server really owns its certificate. Used by Safari, Mail and apps on iOS and macOS.

- One duplicated `goto fail;` skipped the final signature check
- → fake servers accepted on iOS and macOS
- A **bad-signature** test would fail; coverage shows the check **never ran**

Source: M. Bland, “Finding More Than One Worm in the Apple,” ACM Queue, 2014

Why 100% coverage is hard

Rare failures

disk full, network down: need fault injection

```
if (e.code === "ENOSPC") …
```

Defensive checks

branches that “can’t happen”

```
if (!user) throw Error("unreachable")
```

Platform or config

Windows-only code, feature flags

```
if (platform === "win32") …
```

Dead code

nothing reaches it: delete it

```
function oldExport() {…}
```

How much is enough? Google: 60% acceptable · 75% commendable · 90% exemplary. **100%** is rare: SQLite’s core took 10 months of work.

Chasing 100% leads to tests that touch lines without checking them.
Aim for coverage of changed and risky code.

Use coverage to find gaps, not as a goal

1. Write tests from the **specification**
2. Measure **coverage**
3. For each uncovered part: **which requirement does it serve?** → write that test

↺ repeat

```
test("applies the discount", () => {
  const total = applyDiscount(cart, 0.1);
  expect(total).toBeLessThan(100);
});
```

covered ✓, checked weakly ✗

a 20% discount or a rounding bug **still passes**

Measure with: Vitest `--coverage` (V8 or Istanbul) · patch coverage on PRs (Codecov)

Coverage shows what ran, not how well it was checked

From uncovered code to a requirement

Safety-critical (DO-178C, avionics)

DO-178C: the standard for certifying aircraft software

Each line or branch the tests never ran must be explained as:

- **Missing test:** add or change a test
- **Missing requirement:** fix it, then test it
- **Dead code:** no requirement: remove it
- **Deactivated code:** on purpose but switched off (e.g., a feature flag): show it can't turn on

Typical web team

1. Issue lists the criteria
2. PR links the issue
3. Patch coverage shows untested lines
4. Review: which criterion?

Agent help: give it the uncovered lines *and* the criteria.

Tools find what didn't run; people decide why and what to do: a missing test? a missing requirement? code to delete?

Sources: Ansys; LDRA; Rapita; Codecov docs; Ivanković et al., FSE 2019

Test oracles in practice

Deciding what “correct” means

An oracle has two parts

Oracle

Information	Procedure
what the correct behavior is	what to observe, and how to compare

```
expect(total).toBe(42)
```

where did 42 come from?

An oracle is more than an `expect`: snapshots, schemas, reference implementations, invariants, a person's judgment

A weak oracle is easy to write; a good one is hard:
what to expect, what to observe, how to compare

Our study: how practitioners engineer test oracles

Practices, Challenges, and Needs in Functional Test Oracle Engineering: Insights from Practitioners

Anonymous Author(s)

***Abstract*—Functional test oracles are essential for determining whether software behaves as expected. However, engineering functional oracles is challenging because expected behavior is often hard to identify, verify, and maintain as systems evolve. Although prior work has studied test oracles and general software testing practices, little is known about how practitioners engineer functional test oracles in GUI-based systems in practice. We conducted a mixed-method study with 51 software practitioners, including follow-up interviews with 10 participants. We found that practitioners determine expected behavior from many sources, including requirements, project knowledge, UI artifacts, architecture documents, existing behavior, and stakeholder clarification. They verify a wide range of system behavior, from return values and exceptions to UI workflows, backend state, logs, and service interactions. Tools and generative AI support some testing and oracle-related activities, but practitioners still rely heavily on human judgment to define expectations, inspect outputs, verify AI-generated results, and interpret failures. In addition, practitioners struggle with unclear or undocumented expected behavior, frequent changes, high maintenance effort, complex behavior across components or workflows, and conflicting expectations across sources. These findings suggest that oracle engineering should be treated as a first-class engineering activity and that future support should help practitioners capture, assess, maintain, and trace expected behavior knowledge.**

changes, API interactions, logs, and external effects [4, 5]. Thus, practitioners need to reason about both user-facing behavior and underlying system behavior when deciding whether a test passed or failed [6]. Prior work has proposed and evaluated many techniques for GUI testing and test oracles [1, 4, 7–12], and prior empirical studies have examined testing practice more broadly [13–15]. However, we still lack an empirical account of how practitioners engineer functional test oracles in GUI-based systems: how they determine expected behavior, decide what behavior to check, judge correctness, use tools, maintain oracles, and address oracle-related challenges.

In this paper, we study *oracle engineering* as the set of activities involved in determining expected behavior, selecting the system behavior to observe, judging correctness, and maintaining those judgments as systems evolve. Understanding these activities is important for identifying gaps in current support and for guiding future tools, methods, processes, research, and training/education.

To address this gap, we conducted a mixed-method study with 51 software practitioners with diverse roles, experience levels, domains, and project contexts. All participants had

E2E: Where does the expected value come from?

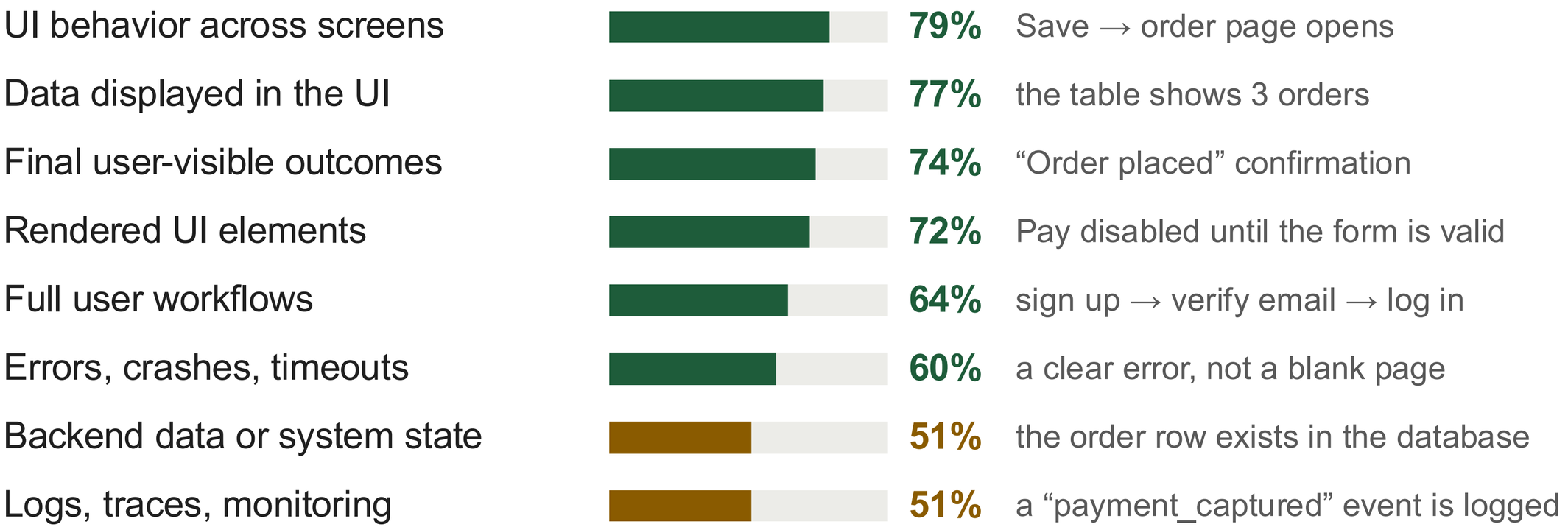


E2E tests · n = 47 · source: our study

🔄 observe the software → record as expected value → test passes → bug stays

If the expected value comes from what the software does now, the test agrees with the bug

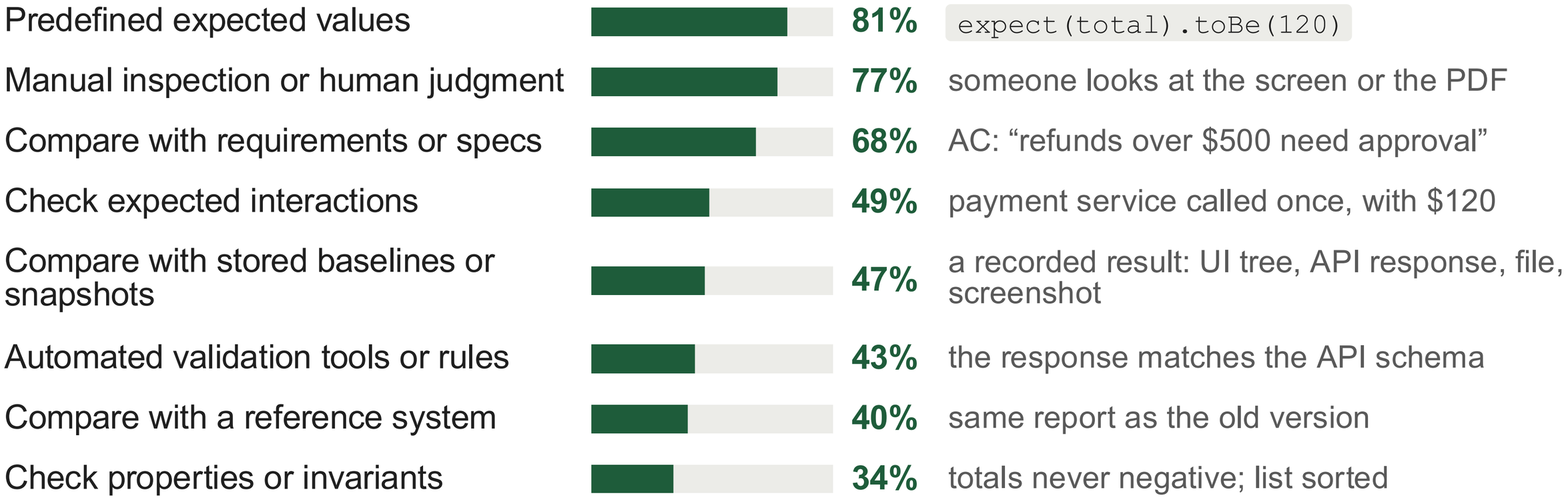
E2E: What practitioners check



E2E tests · n = 47 · source: our study

E2E oracles aren’t only visual: half also check backend state and logs.

E2E: How teams judge correctness



Rare: AI tools to validate outputs (17%)

E2E tests · n = 47 · source: our study

Strong oracle ≠ strict oracle

Place an order for 2 items. Check the API response.

✗ Too strict

```
expect(res.body)
  .toEqual(
    savedResponse);
```

compares every field, even `id` and `createdAt`

✗ fails every run

✗ Too shallow

```
expect(res.status)
  .toBe(201);
```

wrong total or missing items items still pass

✗ misses bugs

✓ Right

```
expect(res.body)
  .toMatchObject({
    status: "paid",
    total: 120,
    items: [{ sku: "A1" },
            { sku: "B2" }],
  });
```

checks only the fields the spec names

✓ checks the spec

A good oracle checks the right things with the right precision . More checks, or stricter ones, don't don't make it better.

Same test, weak or strong check (1/2)

✗ WEAK	✓ STRONG	PROBLEM
`expect(sendEmail).toHaveBeenCalled();`	`expect(sendEmail).toHaveBeenCalledWith("ana@x.com", expect.stringContaining("Order #101"));`	passes if the wrong person gets the email

`// a has $500`
`await expect(transfer(a, b, 800)).rejects.toThrow("insufficient funds");`
	`// a has $500`
	`await expect(transfer(a, b, 800)).rejects.toThrow("insufficient funds");`
	`expect(balanceOf(a)).toBe(500);`
		passes even if money left `a` before the error

Same test, weak or strong check (2/2)

✗ WEAK	✓ STRONG	PROBLEM
`expect(items.length).toBe(3);`	`expect(items.map(i => i.sku))` `  .toEqual(["A1", "B2", "C3"]);`	passes with the wrong items
`expect(fn).toThrow();`	`expect(fn).toThrow(` `  "max length exceeded");`	passes for any error, even a crash

Can’t list every item? Check the ones that matter, or a property of all of them (all belong to this user, sorted, no duplicates). duplicates).

GUI checks: weak or strong (1/2)

✗ WEAK

```
expect(toast).toBeVisible();
```

✓ STRONG

```
expect(toast)
  .toHaveText("Order saved");
await page.reload();
expect(row).toContainText("$120");
```

PROBLEM

nothing saved still passes

After clicking **Add**, the total updates when the server replies

```
await page
  .waitForTimeout(2000);
expect(await total
  .textContent())
  .toBe("$120");
```

```
// retries until it matches
await expect(total)
  .toHaveText("$120");
```

flaky (fails on a slow machine) and **slow**



GUI checks: weak or strong (2/2)

✗ WEAK	✓ STRONG	PROBLEM
`page.locator(".btn-primary")`	`page.getByRole("button", { name: "Save" })`	breaks on any restyle

Check that the sales chart **looks** right (compared to a saved screenshot)

`await expect(page).toHaveScreenshot();`	`await expect(chart).toHaveScreenshot({ mask: [timestamp], maxDiffPixels: 100 });`	**fails without a bug:** any pixel change (a timestamp, an ad)

A person still reviews screenshot diffs before accepting a new baseline.

Comparisons that mislead (1/2)

Snapshot updated without reading

Snapshot = a saved output used as the expected value; value; `-u` accepts every new output, bugs included

```
$ npx vitest -u
✓ 12 snapshots updated
```

Fix: review the snapshot diff like code

Expected value copied from the code

You ran the code once and pasted what it returned

```
// ran it once, pasted the result
expect(calcTax(order))
  .toBe(4.32);
```

Fix: compute it from the spec: 8% of $50 = $4.00

A test can have an assertion and still check nothing.

Comparisons that mislead (2/2)

An assertion that never runs

If `save` stops throwing, the catch never runs

```
try { await save(bad); }
catch (e) {
  expect(e.code).toBe(400);
}
```

Fix: `rejects.toMatchObject({ code: 400 })` or `expect.assertions(1)`

Order that isn't guaranteed

The database returns rows in any order

```
expect(tags)
  .toEqual(["a", "b"]);
```

Fix: sort first, or compare as a set

A test can have an assertion and still check nothing.

Oracles drift and conflict

Drift

the requirement changes (a new issue, a comment, a comment, a meeting), but the spec and tests don't



Week 2, issue #42: free shipping over **$75** . Test written.

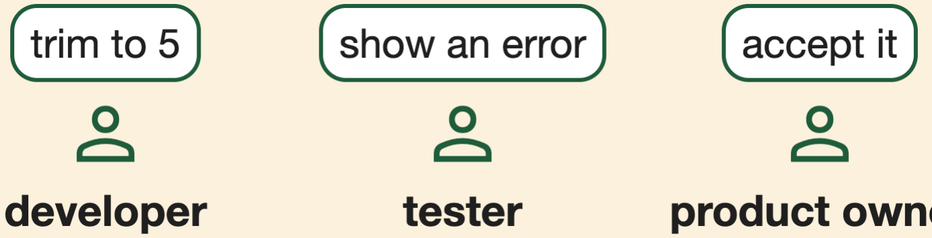
Week 6, issue #58: free shipping over **$50** . Code updated, test not.

→ the old test fails on correct code, or someone "fixes" the code back

Conflict

developer, tester, and product owner each have their own "correct"

A user enters a 7-digit zip code:



→ the test encodes whoever wrote it

What teams do:

✓ settle it with the people who decide

✓ link each check to its source

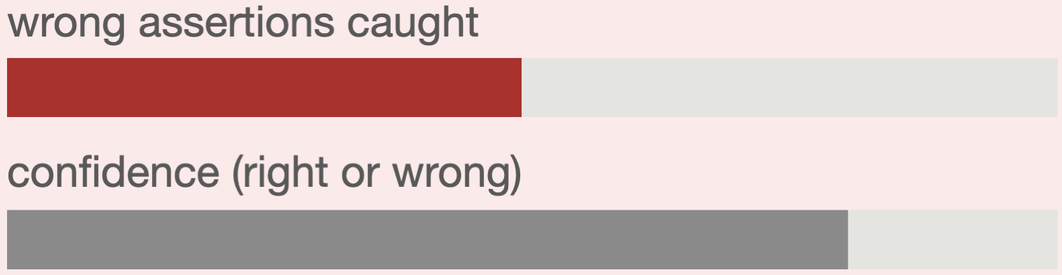
✓ update tests when the requirement changes

When AI writes the checks

49%

of wrong assertions caught

86 programmers judged AI-written **postconditions** (rules the output must satisfy), catching only **49%** of wrong ones, with the **same confidence** whether right or wrong



“the same AI writing and running the tests is not trustworthy!” — Outline team

In review

- ✓ **Try to break it:** find a valid input where the assertion gives the wrong verdict
- ✓ **Slow down** : wrong assertions get accepted fast
- ✓ Check it against the **spec** , not the AI’s explanation

Kaufman, Brun, Murali, Endres, “Programmers Are Poor and Overconfident Judges of LLM-Generated Assertions,” arXiv, 2026

Exercise: what should the test expect?

Medusa's dashboard shows **total revenue**. A test creates these orders.
What should `totalRevenue()` return?

#1	`$100`	paid
#2	`$50`	payment not taken yet
#3	`$80`	paid, then $30 refunded
#4	`€40`	paid; the store's currency is USD
#5	`$60`	canceled

Write down one number. Defend your answer.

What should the test expect? — answers

#1	`$100`	paid	✓ counts	**`$100`**	captured payment counts
#2	`$50`	payment not taken yet	✗ excluded	**`$0`**	authorized only: no money taken
#3	`$80`	paid, then $30 refunded	✓ counts	**`$50`**	net of the refund
#4	`€40`	paid, in euros	✗ excluded	**`$0`**	other currencies are ignored
#5	`$60`	canceled	✗ excluded	**`$0`**	canceled orders don’t count

Expected total **$150**

Every disagreement is a missing requirement. The code can’t settle it.

Is your test too weak or too brittle?

Too weak? (misses bugs)

- ✓ **Break the code on purpose:** a test should fail. Mutation testing (next) does this automatically.
- ✓ **Red flags:** checks only that it ran, succeeded, or returned *something*
- ✓ **Expected value:** can you derive it from the spec by hand? If not, it was likely copied
- ✓ **Coverage ≠ checked:** a covered line with no assertion on its result

Too brittle? (fails without a bug)

- ✓ **Broke after a refactor?** It checks *how*, not *what*
- ✓ **Rerun it** (5×, another machine): flips mean flaky
- ✓ **Snapshot updated on most PRs?** It covers too much: snapshot only the part under test
- ✓ **Find elements as a user would:** “the Save button” , not `.btn-primary`

Have someone else review test changes, including snapshot diffs.



Mutation testing

Testing the tests

Who tests the tests?

All three suites have **100% coverage**. A bug is planted: `-` → `+`

```
function discount(total, rate) {
  return total - total * rate;
}
```

Suite A · weak check

```
expect(discount(100, 0.1))
  .toBeGreaterThan(0);
```

✗ still green

Suite B · boundary only

```
expect(discount(100, 0))
  .toBe(100);
```

✗ still green

at rate 0, `-` and `+` give the same result

Fix: add a typical rate,
`discount(100, 0.1)` → 90

Suite C

```
expect(discount(100, 0.1))
  .toBe(90);
```

✓ goes red

Only C catches it. Mutation testing plants bugs like this automatically.

Would your tests notice?

Make a small change to the code — a **mutant**: `>` → `>=`, `+` → `-`, delete a line

ORIGINAL

```
if (qty > max) total = total + fee;
```

MUTANTS

```
if (qty >= max) total = total + fee;
```
→ ✓ killed

```
if (qty > max) total = total - fee;
```
→ ✓ killed

```
// line deleted
```
→ ✗ survived

Mutation score = killed ÷ (killed + survived)



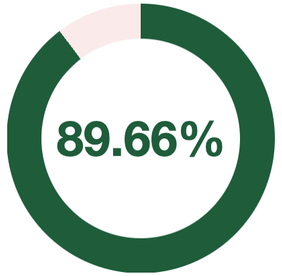
Equivalent mutant (false positive): no input can tell it apart. `max = a >= b ? a : b` → `a > b ? a : b`: when `a === b`, both return the same value

Show no input differs, then **mark it with a reason**

Actual Budget: StrykerJS on the budget engine

packages/ loot-core/ stryker.config.json

```
{
  "testRunner": "vitest",
  "mutate": [
    "src/server/budget/**/*.ts",
    "!src/**/*.test.ts"
  ],
  "reporters": [
    "html", "clear-text", "progress"]
}
```



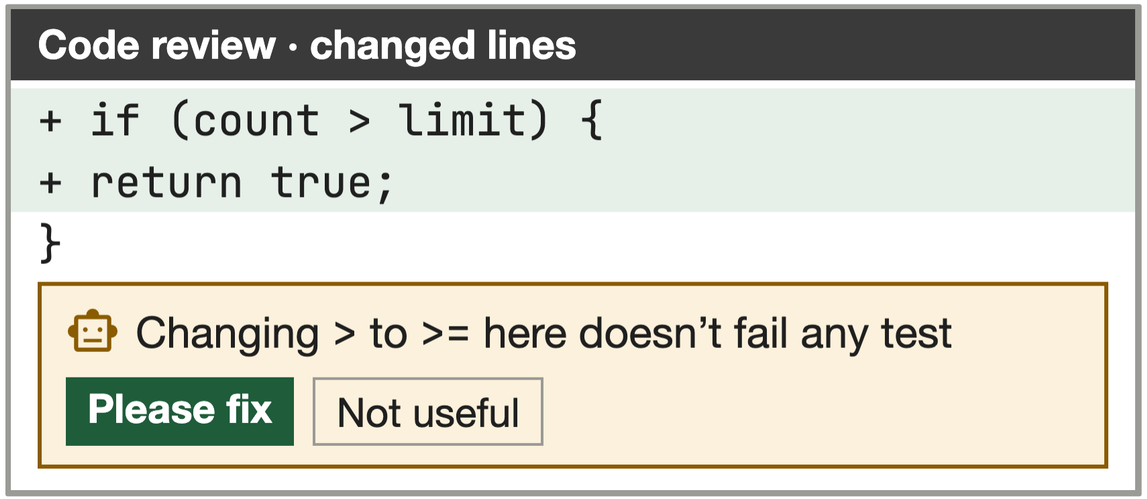
Scoped to the budget engine: **29 mutants · 89.66% score · 14 seconds**

SURVIVORS

- `<` → `<=`
- `a?.b` → `a.b`

Each survivor: a missing test, a weak test to strengthen, or an equivalent mutant

Mutation testing at Google



82% rated useful

Median: **2** surviving mutants per change

60% of useful mutants get fixed; others are judged not not worth a test

1. Developer sends a change for review
2. Mutate only changed, covered lines
3. Survivors appear as review comments
4. Reviewer can click “Please fix”; the author writes or strengthens a test, by hand

Petrović, Ivanković, Fraser, Just, “Practical Mutation Testing at Scale: A view from Google,” IEEE TSE 48(10), 2022

Exercise: which mutant survives?

```
function normalizeReason(
  reason: string, max: number) {
  const r = reason.trim();
  if (r.length === 0) return null;
  if (r.length > max)
    throw new Error("too long");
  return r;
}
```

Tests

`normalizeReason("", 1000)` → null

`normalizeReason("a".repeat(1001), 1000)`
→ throws "too long"

`normalizeReason("Done", 1000)` → "Done"

`normalizeReason(" Done ", 1000)` → not null

Mutants

M1 `length > max` → `length >= max`

M2 remove `.trim()`

M3 `length === 0` → `length === 1`

Which survive? What test kills each survivor?

Which mutant survives? — answers

Mutant	Result	Why	Fix
M1 `>=`	✗ survives	**missing test:** nothing tests exactly `max`	add `"a".repeat(1000)` → accepted
M2 no trim	✗ survives	**weak test:** has spaces, checks only “not “not null”	strengthen: `.toBe("Done")`
M3 `=== 1`	✓ killed	the `""` test catches it	—

One survivor needs a new test; the other needs a stronger one

What to do with a survivor

1 Read the mutant: what behavior would it break?	**Understand it first:** name the input where the behavior differs (e.g., < → <=: exactly at the limit)
2 Does any test run this line with an input that matters?	**No → missing test:** add one, usually at a boundary (M1)
3 A test runs it but still passes?	**Weak test:** strengthen the assertion or pick a better input (M2)
4 Same behavior for every input?	**Equivalent mutant:** nothing to test; mark it
5 Real, but not worth a test?	**Skip or suppress it** (logging, timeouts: Google's "arid" code)

Expected value from the **spec** , not from what the code does now

Rerun to confirm it's killed. Don't chase 100%.

Test doubles and integration

Testing parts alone — and together

Why use test doubles?

A **test double** replaces a real dependency (database, payment service, clock, email sender) during a test.

Why teams use them

- **Slow:** a database or network call per test adds up
- **Unpredictable:** time, randomness, network
- **Unavailable:** not built yet, another team's, paid
- **Risky:** charges cards, sends emails
- **Hard to steer:** make a payment decline or time out

Their limits

- ✗ A double is **your assumption** about the real thing
- ✗ It **drifts** when the real service changes
- ✗ **Over-mocking** checks *how* code calls things, not *what* it does
- ✗ Passing tests give **false confidence**

Each double is an assumption about an interface; integration tests check it. Integrating = replacing doubles with real parts, one at a time.

Kinds of test doubles

`checkout(order)` charges a card and emails a receipt

checkout ⟶ `charge($20)`

checkout ⟵ `{ status: "paid" }`

Give your code fake answers

Stub: a fixed reply

```
charge() → "paid"
```

Fake: a simple working version

```
in-memory database
```

Record what your code sends

Use when the requirement **is** a call: send one email, charge once, never call X

Spy: check the call afterwards

```
send called once
```

Mock: expected calls set beforehand

```
charge($20), once
```

Check the **result** (state) or the **calls** (behavior). In everyday use, “**mock**” often means any double (`prismaMock`, `jest.mock` , Mock Service Worker).

Test doubles in your projects

OUTLINE

stub (network)

MSW (Mock Service Worker)



intercepts outgoing HTTP calls in in tests

```
setupServer(…)
// localhost:
//   passthrough
// others: mocked
```

GITEA

fake clock

fake clock



`page.clock` controls time in a Playwright test

```
await page.clock
   .install();
…
await page.clock
   .runFor(200);
```

CAL.DIY

stub

mocked database



`prismaMock` returns canned query results

```
prismaMock
   .eventTypeFavorite
   .findMany
   .mockResolvedValue([]);
```

Design for testability

Doubles are easy when the code **takes** its dependencies instead of **grabbing** them.

1. **Pass dependencies in as parameters**
2. **Keep logic in pure functions, separate from the UI**

✗ BEFORE

```
function lateFee(
  dueDate: Date,
) {
  const today = new Date();
  …
}
```

✓ AFTER

```
function lateFee(
  dueDate: Date,
  today: Date,
) {
  …
}
```

```
expect(lateFee(jan1, jan31)).toBe(15);
```

testable with plain values

Outline kept the gallery filter logic “in a pure helper, separate from the UI, so it is unit-testable”



Works with the double, fails with the real thing

But a double is only as good as **your assumption** about the real thing.

Cal.diy

mocked database checks the query · **real Postgres** checks what only a database can show:
show:

- ✓ two clicks on ★ at the same time → one favorite
- ✓ deleting an event type removes its stars

Medusa

renewals tested only against a payment double, never real Stripe



✗ never tested

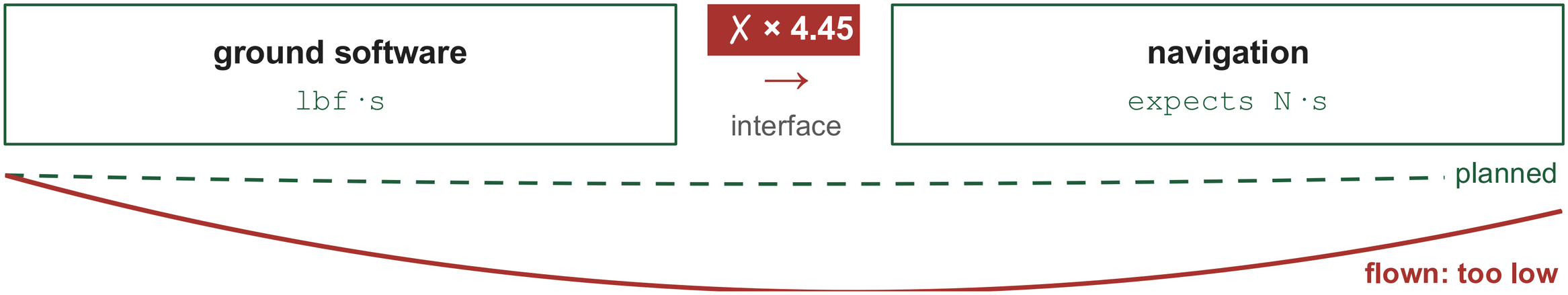
Practice: run integration tests against a real database (e.g., Testcontainers)

Mars Climate Orbiter (1999)

An integration failure: each part worked; the interface between them didn't

One team's software produced thruster data in **pound-force seconds**

The interface specification required **newton-seconds** → off by a factor of **4.45**



The orbiter flew too low and was lost on **September 23, 1999** . Investigators: end-to-end testing of that software against the against the interface specification "did not appear to be done".

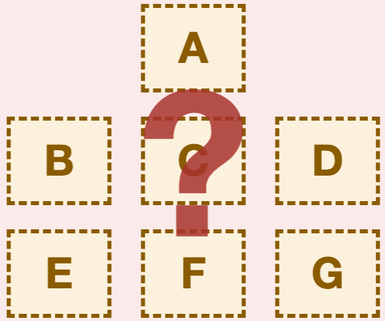
Each side tested against its own assumption of the other, like a double. Only running the real parts together against the interface spec would have caught it.

Source: NASA Mishap Investigation Board, 1999

Big bang or incremental?

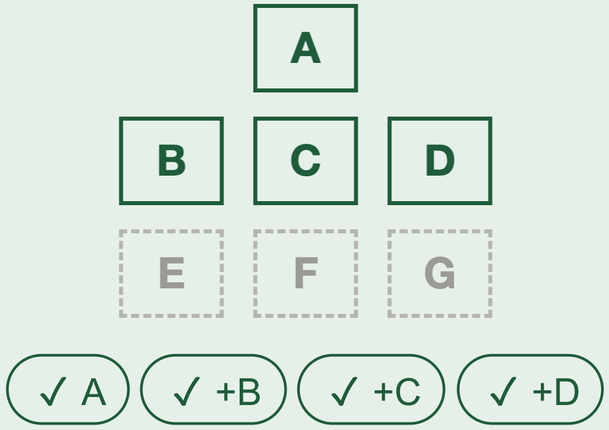
✗ Big bang

put everything together at the end → a failure could be anywhere



✓ Incremental

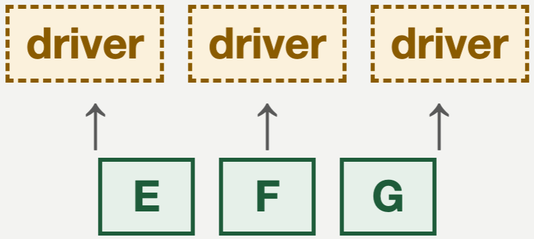
add one piece at a time → a new failure is near the new piece



Bottom-up or top-down?

Bottom-up

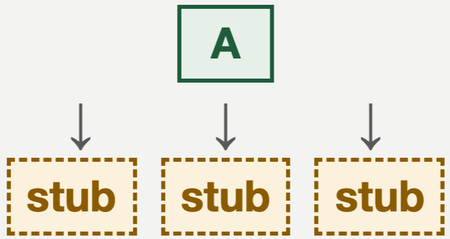
lower parts first, tested with **drivers**



Gitea split close reasons by layer: storage → user interface → automatic closes

Top-down

top parts first, lower parts replaced with **stubs**



Today: integrate continuously — every merge runs the tests (**CI**: continuous integration)

commit → **build** → **test** → **merge**

Exercise: what to fake, what to check for real

For each: which **double** would you use, and what would you still **check against the real thing**?

1. **Cal.diy:** an email reminder goes out 24 hours before a booking

2. **Outline:** a slideshow shows “Couldn’t load image” when the image server is down or times out

3. **Medusa:** a declined renewal payment ends the subscription

4. **Cal.diy:** a user double-clicks ★, so two requests arrive at once. They should end up with **one** favorite, not two

What to fake, what to check for real — answers

Scenario	Fast test with a double	Also confirm with the real thing
1 Reminder 24 h before	**fake clock** + **spy** on the email sender	email provider’s sandbox: the email arrives
2 Image server fails	**MSW** returns a network error or a delay for the image URL	E2E with a really broken URL (404)
3 Declined payment	**stub** provider that declines	Stripe test mode with a decline test card
4 Double-click ★	**fake** that rejects a duplicate (user, event type): your code handles the error	**real Postgres** : the unique constraint on (user, event type) holds when both requests arrive together

Doubles test your code; a real-component test checks the double’s assumption.

Keeping a test suite healthy

When “red” stops meaning anything

When red becomes normal

Red should mean one thing: **your change broke something**.

When the build is always red, people stop trusting it, and new failures go unnoticed.

Checks · your PR #214	
✗ `unit tests` failing for 3 weeks	
✗ `lint` failing since last sprint	
✗ `e2e (webkit)` flaky	
✓ `build`	
✗ `integration tests` **new**	**is this mine?**

Seen in our projects: Actual Budget · Medusa · Excalidraw · Gitea

Flaky tests

Why does a test go red when nothing broke?

Flaky: passes and fails on the same code

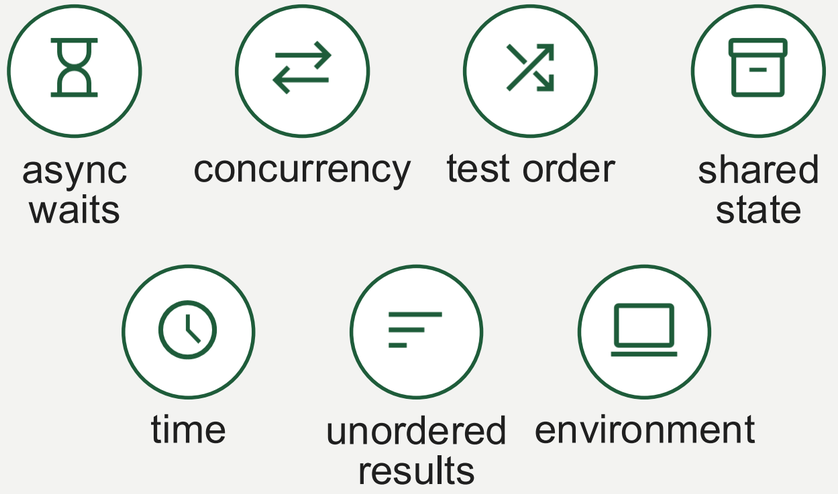
GOOGLE (2016)

1.5% of test runs flaky

16% of tests have some flakiness

84% of pass → fail changes involved a flaky test

Common causes



Sources: J. Micco, “Flaky Tests at Google and How We Mitigate Them,” Google Testing Blog, 2016 · Luo, Hariri, Eloussi, Marinov, “An Empirical Analysis of Flaky Tests,” FSE 2014

Flaky causes, with examples (1/2)

⌛ Async wait

✗ FLAKY

```
await page
  .waitForTimeout(500);
// fails if the server
// takes 600 ms
```

Fix: wait for the condition: `await expect(total).toHaveText("$120")`

⇄ Concurrency

✗ FLAKY

```
// two requests add 1 at once
expect(count).toBe(2);
// sometimes 1
```

Fix: make the update atomic (transaction or lock); run run the test many times

Flaky causes, with examples (2/2)

Test order / shared state

✗ FLAKY

```
// test A adds a user
// test B:
expect(users)
  .toHaveLength(0);
```

Fix: fresh data per test: reset in `beforeEach` , or unique IDs

Time

✗ FLAKY

```
expect(formatDate(new Date()))
  .toBe("Nov 12");
// fails tomorrow
```

Fix: fix the clock (`vi.setSystemTime`) and the time zone (`TZ=UTC`)

Unordered results → **Gitea** (next) · Environment → **Same environment, same result** (after Gitea)

Less common: resource leaks · network · randomness · floating point · file I/O

Gitea: a flaky test, found and fixed

A milestone's **burndown chart** counts open issues per day by replaying **issue events** (opened, closed, reopened) in time order.

Events in the **same second** tie → their order depends on the database. **PostgreSQL** differed → **wrong chart, chart, test failed**

✗ BEFORE

```
cmp.Compare(
  a.CreatedUnix, b.CreatedUnix)
```

✓ AFTER

```
cmp.Or(
  cmp.Compare(
    a.CreatedUnix, b.CreatedUnix),
  cmp.Compare(a.ID, b.ID))
```

Fix: break ties by id (the order they were saved)

Same environment, same result

Some tests fail only on certain machines. Fix: run them under the **same conditions** everywhere.

Time zone

Broke when: a date test passed locally, failed on CI (UTC)

Cal.diy, Outline

TZ=UTC vitest run

Operating system

Broke when: Windows file permissions differ from Linux

Windows file permissions broke Gitea's tests → moved to WSL (Windows Subsystem for Linux)

Windows → WSL

Whole environment

Broke when: teammates had different Node or Postgres versions

Dev Containers — same tools for everyone (Medusa's showcase)

.devcontainer/ devcontainer.json

Known bugs and limits: mark them

Can’t fix it right away? Mark it, don’t leave it red.

Expected failure

When: a known bug you can’t fix yet

```
test.fail();
// bug: …/issues/104
```

Risk: low

Guardrail: link the issue; you’re told when the bug is fixed (the test passes unexpectedly)

Skip with a reason

When: it can’t run in one environment

```
test.skip(browserName === "webkit",
  "macOS WebKit … Tab skips buttons");
```

Risk: forgotten skips

Guardrail: require a reason or issue link; review skips regularly

Both from Gitea: tests/e2e/multiselect-aria.test.ts



Flaky tests: tolerate, but track

Retry on failure

When: the test fails now and then

```
// playwright.config.ts
retries: 2
```

Risk: hides real races

Guardrail: report “passed on retry” as **flaky** ; fix tests that keep needing retries

Quarantine

When: too flaky to block merges

```
// separate, non-blocking job
quarantine: ["checkout.spec.ts"]
```

Risk: becomes permanent

Guardrail: an **owner** and a **deadline** to fix or delete

Every exception is visible and has an owner, or it’s just a hidden red build.

Testing in practice: what to run, who, and where

Regression, priorities, people, and environments

Regression testing

After every change: does the change work, and **did it break anything else?**

a change

→ ✓ **its own tests**

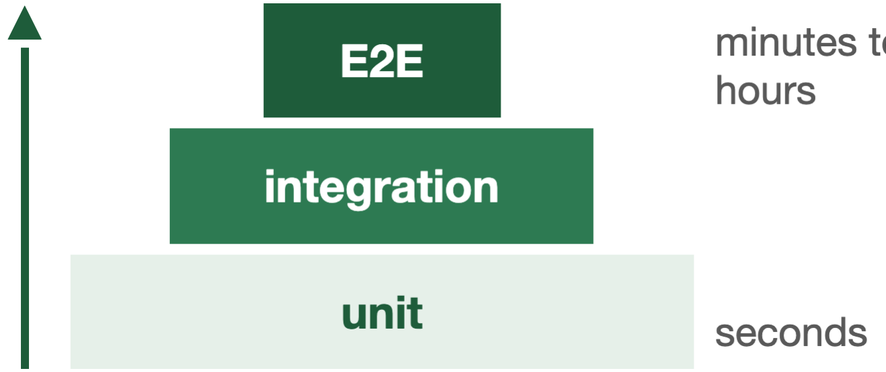
→ ✗ **an unrelated module's tests**

Medusa: the full suite caught a regression **the feature's own tests missed**

Medusa's lesson: rerun the full suite after every change

As the suite grows, that gets slow: next, how teams choose what to run

You can’t run everything every time



Pyramid: many fast unit tests · fewer integration · few slow E2E

Medusa: fast and slow suites

`fast suite` → **runs first**

`slow suite` → **separate, slower job**

Cal.diy: E2E on demand

`unit + integration` → **every push**

`E2E` → **PR labeled ready-for-e2e**

Large teams: only affected tests

change A → run A + what depends on A

`affected tests` → **every push**

Test the riskiest parts first

Run the riskiest tests most often.

Priority	Tests	Run
1	**Smoke tests:** the app starts; happy paths of critical features (log in, check out)	every commit
2	**Core features in depth:** edge cases, errors	every pull request
3	less critical features	nightly / before release

Raise priority for: parts that failed before · parts that changed · failures that would be costly

How teams choose and schedule tests

Find what's risky

Critical user journeys
sign up, log in, pay

Bug history and hotspots
where bugs cluster

What changed
the diff and what depends on it

Production incidents
each outage gets a test

Schedule it

Tags
`@smoke`, `@slow`

CI triggers
push → PR → merge → nightly → release

Required checks
must pass to merge: fast, reliable tests

When manual testing is the right call

Some checks need a person. Automate the rest.

Right for

- **How it looks:** layout, broken screens
- **How it feels:** confusing or slow flows
- **A real screen reader:** listening to it
- **Exploratory testing:** finding what nobody thought to test

Automate what you can, and help people judge the rest

Gitea: checks the label picker says “collapsed” after Escape

`goto`	screenshot
`click`	
`fill`	
`expect ✗`	

Traces show a person what failed

Who tests?

Developers own tests



write tests with every change; reviewers can block a change without tests

Google: “expected to include both the feature code and tests”

Test engineers



build test tools and infrastructure; plan risk-based testing

work inside product teams

Separate QA team



independent testers; manual and exploratory testing

often test release candidates before launch

Many teams combine all three.

Source: Software Engineering at Google, ch. 11 and 23

Test environments and data

Teams test in several places, each closer to real use.

closer to

Environment	What it is	Used for
Local	your machine	every change, while coding
CI	a fresh machine per run	every push and PR
Staging	a copy of the production setup	release candidates, manual QA
Production	real users	canary rollouts, feature flags, monitoring

Closer to production: more realistic · slower and costlier · mistakes affect more people

Data, at any level: synthetic · seeded · a copy of production (**anonymized** : personal data removed or masked) · real (production)

Sources: Software Engineering at Google, ch. 14 and 23; Google for Developers

What a test strategy says

Vague

“Unit and integration tests were written. All changes were tested.”

Useful

- ✓ **What** tests, for which changes
- ✓ **When** they run (every push, PR, nightly)
- ✓ **What’s manual**, and why
- ✓ **Who** writes, reviews, and runs which tests
- ✓ **Where** (environments, data)
- ✓ **How the tests were checked** (oracles, mutation)

Tools: testing

Need	Tools
Unit and integration	● Vitest ● Jest ● go test pytest JUnit
End-to-end	● Playwright Cypress
Test doubles	● MSW (Mock Service Worker) Testcontainers built-in mocks
Coverage	● Vitest/Istanbul coverage go test -cover
Mutation testing	● StrykerJS PIT (Java)
Fuzzing and properties	● go test -fuzz libFuzzer OSS-Fuzz fast-check Hypothesis
Contracts	Pact

● already used in your projects