Testing Strategy

Verified 2026-11-02

Where each test belongs — layer costs, pyramid vs trophy, mocking rules, test data, time, flaky-test policy, coverage, AI-written code and CI shape.

Contents

Mental model 2

Test layers and their cost 2

Pyramid or trophy 3

What to mock 4

Test data and fixtures 5

Async and time 6

Flaky-test policy 7

Coverage as a signal 8

Testing AI-generated code 9

CI pipeline shape 9

Further reading 10

Mental model

Every test buys confidence and pays for it in run time, flakiness and maintenance, and the price rises with how much of the system the test touches. So place each test at the **lowest layer that can prove the behaviour**, and climb a layer only when the lower one cannot catch the bug you are worried about. A suite is good when a green run lets you ship without a manual check, and a red run points at the defect.

Test layers and their cost

Layer	Proves	Cost
Unit	One function or class is right	Milliseconds, stable
Integration	Your code works with a real DB, queue, framework	Seconds, mostly stable
End-to-end	A user journey works through the whole stack	Slow, flakiest
Contract	Consumer and provider agree on an API	Cheap, needs both sides

Google separates a test's **size** (resources it may use) from its **scope** (code it exercises). Size is the better predictor of cost: a small test runs in one process with no network or disk; a medium test stays on one machine and may call `localhost`; a large test may span machines. Larger tests are flakier because they depend on more infrastructure.

```
Bug                       Cheapest layer
Off-by-one in rounding    unit
Wrong SQL                 integration (real DB)
Checkout button dead      end-to-end
API JSON field renamed    contract
```

Pyramid or trophy

Model	Bulk of the tests	Bet
Test pyramid	Unit	Fast, isolated tests are cheap to keep
Testing trophy	Integration	Confidence comes from parts working together
Both	Few end-to-end	Full-stack tests are slow and brittle

The pyramid says write many small tests and fewer as granularity rises. The trophy (Kent C. Dodds) adds a base of static analysis — types and lint — and puts the bulk in integration tests, arguing that unit tests prove little if the pieces never meet. They disagree on the middle, not on the top: both keep end-to-end tests few and focused on critical journeys.

```
Pyramid          Trophy
   /e2e\           e2e         few
  /integ.\     integration     most (trophy)
 /  unit  \       unit         most (pyramid)
                  static       types + lint
```

> **Tip:** Pick per codebase. Pure-logic libraries suit the pyramid; a UI that mostly wires framework, API and state together suits the trophy. Argue about where a specific bug would have been caught, not about shapes.

What to mock

Dependency	Default
Code you are testing	Never mock
Your own database	Real instance (container)
Third-party API, payments, email	Fake or stub at your adapter
Clock, randomness, IDs	Inject and control
Your own modules called by the unit	Usually real

Mocks remove the dependency and with it the confidence that you use it correctly. Prefer the real implementation, then a **fake** (a working in-memory version), and reach for interaction mocks last. Mock at the boundary you own — an adapter around the vendor SDK — not the SDK itself, so the test survives an SDK upgrade.

```
interface Mailer {
  send(to: string, body: string): Promise<void>
}
class FakeMailer implements Mailer {
  sent: { to: string; body: string }[] = []
  async send(to: string, body: string) {
```

```
    this.sent.push({ to, body })
  }
}
const mailer = new FakeMailer()
await mailer.send('a@x.io', 'Welcome')
mailer.sent.length  // 1
```

> **Gotcha:** A test that asserts `toHaveBeenCalledWith` on every collaborator tests the implementation, not the behaviour. It breaks on every refactor and passes when the real collaborator rejects the call.

Test data and fixtures

Practice	Why
Each test creates its own data	No order dependence
Builders with sensible defaults	Tests state only what matters
Unique IDs per test	Safe under parallel runs
No shared mutable fixture	One test cannot break another

Shared seed data turns into a hidden global: one test edits it and an unrelated test fails, but only in the full run. A builder function with defaults keeps each test short and makes the one field that matters visible.

```ts
type User = { id: string; name: string
  role: 'admin' | 'member'; active: boolean }

const aUser = (
  o: Partial<User> = {},
): User => ({
  id: crypto.randomUUID(), name: 'Ana',
  role: 'member', active: true, ...o,
})

const admin = aUser({ role: 'admin' })
admin.active  // true — default, not noise
```

Async and time

Problem	Fix
`sleep(2000)` before assert	Wait for the condition, with a timeout
`new Date()` inside logic	Inject a clock
Timers, debounce, retry	Fake timers, advance explicitly
Background job “eventually” done	Poll the observable result

A fixed sleep is either too short (flaky on a slow CI runner) or too long (slow everywhere). Code that reads the clock directly cannot be tested at midnight, month end or a DST switch; pass “now” in, and the test chooses it.

```ts
type Clock = () => Date

const isExpired = (exp: Date, now: Clock) =>
  now().getTime() >= exp.getTime()

const fixed: Clock = () =>
  new Date('2026-03-29T01:00:00Z')
const exp = new Date('2026-03-29T00:59:59Z')
isExpired(exp, fixed)  // true
```

Flaky-test policy

Step	Rule
Detect	Track pass-on-retry, not just failures
Quarantine	Remove from the blocking gate, with an owner
Fix or delete	Time-boxed; a quarantined test proves nothing
Prevent	Isolate data, no sleeps, mock the third party

A flaky test is one that both passes and fails on the same code. DORA lists tolerating unreliable tests as a pitfall: a pass must mean releasable and a failure must mean a real defect. Google pulls flaky tests out of presubmit while they are investigated rather than letting them train engineers to click “re-run”.

```
FLAKY-412  checkout.spec.ts › applies coupon
owner:     @payments-team
```

```
seen:      7 retries-to-pass in 14 days
cause:     shared promo code across workers
due:       2026-11-16, else delete
```

> **Warning:** Automatic retries with no tracking hide real race conditions in production code, not just in tests. Count flakes and treat the count as a defect metric.

Coverage as a signal

Use	Verdict
Find untested files and branches	Good
Review coverage of changed lines	Good
A ratchet that blocks drops	Acceptable
A company-wide % target	Gets gamed

Coverage says which lines ran, not whether anything was checked. A test with no assertions raises coverage and proves nothing. Read the report for **what is missing** — an untested error branch in payment code matters more than the overall percentage.

```
src/billing/refund.ts   78%  uncovered: 41-58
  41  if (amount > charge.amount) {
  ...   throw new RefundExceedsCharge()
-> no test for over-refund: add one
```

Testing AI-generated code

Check	Why
Tests written or reviewed by a human first	They are the spec
Diff touches tests and code together	Look for weakened asserts
Deleted or `.skip`ped tests	Agents “fix” red by removal
Edge cases you named are present	Generated tests mirror the code

DORA 2025 found AI adoption raises delivery throughput but still correlates with lower stability, and that strong automated testing and fast feedback loops are what keep the extra change volume from turning into instability. Tests generated from the implementation tend to assert whatever the code does today, bugs included, so write or approve the behaviour-level tests yourself.

```
Agent task: "add refund limit"
Human:  write 3 failing tests (limit, equal,
        over-limit error) and commit them
Agent:  implement until green; must not
        edit tests/refund.test.ts
Review: test diff empty? asserts intact?
```

CI pipeline shape

Stage	Runs	Blocks merge
Static	Types, lint, format	Yes

Stage	Runs	Blocks merge
Fast	Unit + small integration	Yes
Full	Integration with real services	Yes
Journey	Few end-to-end, contract checks	Yes, or post-merge gate

Order stages cheapest first so most failures surface in seconds. DORA's test-automation guidance is that developers should get feedback from automated tests in under ten minutes, locally and in CI. Shard slow stages rather than dropping them, and keep developers, not a separate team, owning the suite.

```
pr:     static -> fast -> full    (< 10 min)
merge:  journey e2e + contract    (sharded)
deploy: smoke test in production
```

Further reading

- Martin Fowler — The Practical Test Pyramid
- Kent C. Dodds — The Testing Trophy
- Software Engineering at Google — Testing Overview
- Software Engineering at Google — Test Doubles
- DORA — Test automation