

# Metamorphic oracles

AI may reduce the cost of authoring and maintaining behavioural challenges, giving teams reason to reassess the economics behind the test pyramid. Metamorphic relations, mutation and system-level observation can turn tests from examples of success into an adversarial evidence system.

---

## AUTHORED BY

Daniel Johnson & Gary McLean Hall

---

## AI DRAFTING

Astra, Fable & the models of the moment

---

## EVIDENCE CHECKED

15 September 2026. The date tells you when the cited material was reviewed, not when the future agreed to behave.

---

## INVALIDATED BY

When new evidence changes the practical cost or fault-detection value of these techniques.

---

## EXPIRY DATE

The reader's call. Before a million generated assertions are mistaken for one good oracle.

---

The founders own the argument; AI helped draft the prose. The sources and limits are here so you can challenge both.

---

## Contents

01. [The oracle problem](#)
  02. [Beyond example tests](#)
  03. [The old pyramid and its economics](#)
  04. [Inverting the question](#)
  05. [AI as oracle author, not oracle](#)
  06. [Documentation is no longer the same bottleneck](#)
  07. [Mutation proves the harness has teeth](#)
  08. [Building a behavioural census](#)
  09. [Production feedback closes the loop](#)
  10. [A practical harness](#)
  11. [Limits and counterarguments](#)
  12. [Conclusion](#)
  13. [References](#)
-

This method paper distinguishes established testing techniques from a hypothesis about AI-assisted harness economics. The review date records source and editorial checks, not an experiment proving that hypothesis. Numerical examples are constructed arithmetic; they are not commercial prices, customer measurements or tax guidance.

---

## The oracle problem

A test needs more than an input. It needs an oracle: a way to decide whether the observed output is acceptable. For simple arithmetic the oracle can be an exact value. For a search ranking, optimisation result, migration or complex workflow, the correct output may be costly or impossible to enumerate.

---

### The penny that breaks the relation

Splitting preserves the underlying amount but not this fee. Line boundaries matter because rounding happens before summation.

|                         |                                                                      |
|-------------------------|----------------------------------------------------------------------|
| <b>One line</b>         | $10\text{p} \times 10\% \rightarrow 1\text{p fee.}$                  |
| <b>Split line</b>       | $5\text{p} \times 10\% \rightarrow 1\text{p, twice. Total fee: 2p.}$ |
| <b>Invalid relation</b> | "Splitting a line preserves the fee."                                |

Constructed arithmetic · fictional fee rule, not tax treatment.

---

Metamorphic testing addresses that problem by checking relationships between executions. If the input changes in a way that should preserve or predictably transform the result, the relation becomes an oracle even when the exact answer is not known in advance. [Chen, Cheung and Yiu, 1998](#).

Sort a list, then permute the input: the result should remain the same. Duplicate every item in an aggregate: the count should change predictably. For a translation-equivariant operation, translating the input object should translate the output by the same amount. Re-run an idempotent command: the specified business effect should occur only once, while any permitted audit or retry records remain consistent with the contract.

These relations turn domain knowledge into executable challenges.

The relation has to state its preconditions. Sorting is invariant to input order only if the sorting contract defines how equal keys are handled or the comparison deliberately ignores their irrelevant order. An aggregate relation may fail when the domain includes

caps, rounding or deduplication. A useful oracle records those conditions alongside the transformation. Otherwise a mathematically attractive relation can repeatedly reject correct domain behaviour.

Consider a constructed invoice example. Reordering independent line items should preserve the total if the contract rounds each line in the same way and order carries no meaning. Splitting one line into two is a different transformation: it may change the total under line-level rounding. The first relation might be valid while the second is not. This distinction is the work of oracle design. “The numbers should stay the same” is not enough to make a generated test authoritative.

## Worked relation: when splitting a line changes the answer

Define an invented calculation rule solely for this example: multiply each line amount by 10%, round each result to the nearest penny with half-pennies rounded upward, then sum the rounded results. This is a fictional fee calculation, not a claim about tax treatment. Use integer minor units or exact decimal arithmetic so floating-point representation does not introduce another question.

A single line of £0.10 produces a £0.01 fee. Split it into two lines of £0.05. Each fee is half a penny and rounds upward to £0.01, so their sum is £0.02. The underlying amount has not changed, but the contract makes line boundaries consequential. A generated assertion that splitting preserves the fee would reject the correct implementation.

| Transformation                          | Expected relation under this rule | Why                                                      |
|-----------------------------------------|-----------------------------------|----------------------------------------------------------|
| Reorder the same independent lines      | Fee remains identical             | Per-line rounding has no order dependency                |
| Add a zero-value line                   | Fee remains identical             | Its rounded contribution is zero                         |
| Duplicate every line as a distinct item | Fee doubles                       | Each original rounded contribution appears twice         |
| Split a line into smaller lines         | No general equality claim         | Rounding happens before aggregation                      |
| Change a display-only description       | Fee remains identical             | Description is excluded from calculation by the contract |

The duplication relation requires distinct line identities and no invoice-level cap or deduplication. If the real domain adds either, the relation must change. Preconditions are not footnotes to a test; they are the boundary of its authority. The generator should construct valid inputs satisfying them and produce separate examples for the cases outside them.

Now seed a defect: calculate the fee on the invoice sum and round only once. Reordering and adding a zero line still pass. On the two five-penny lines, the faulty path produces one penny, while the agreed line-rounding rule produces two. A reviewed example kills this mutation even though the obvious metamorphic relations do not. This is why relations complement known-answer examples. A family of invariances can be true of both the intended function and a different, incorrect function.

The next useful test asks whether the application preserves the inputs to that function. A browser or API may merge identical lines before calling it, changing a correct domain function's result. A narrow arithmetic property cannot detect that wiring defect. Exercise the application seam with the two separate line identities and inspect both the displayed total and persisted lines. The strongest observation depends on where the promise can be broken.

For a buyer, the exhibit explains what an oracle contributes. "We ran many cases" gives no reason to trust the total. "These independently reviewed examples establish the rounding boundary; these relations cover permitted transformations; this seeded aggregation defect is refused" supplies an inspectable argument. It still leaves other rules, such as adjustments and historical versions, to their own specification and evidence.

---

## Beyond example tests

Example tests are essential. They communicate concrete rules and regressions. Their weakness is selection: a human chooses a handful of cases, often from the happy path, and the suite can remain green around a large unexamined surface.

### Different routes to refusal

#### Known-answer example

The two five-penny lines must yield the reviewed two-penny fee.

#### Metamorphic relation

Reordering those independent lines must preserve that result.

Property-based testing generates many inputs against an invariant. Metamorphic testing generates related inputs and compares their outputs. Differential testing runs two implementations or versions against the same inputs and compares the resulting

observations. Mutation testing changes the implementation deliberately and asks whether the suite notices.

Together they answer different questions:

- examples: does this named case behave as agreed?
- properties: does an invariant hold across a generated space?
- metamorphic relations: do related executions move together correctly?
- differential tests: do two paths remain equivalent within observations?
- mutations: can the oracle detect a plausible defect?

None is sufficient alone. Their combination creates evidence with more refusal power than a large collection of shallow examples.

These techniques should share a model of the obligation without sharing every assumption of the implementation. An example can establish a known boundary result; a property can exercise many inputs; a differential check can compare a replacement; and a mutation can test whether the combined oracle notices a broken rule. They reinforce one another when each supplies a different route to refusal.

They can also agree for the wrong reason. If all expected values are generated by copying the production function, the suite may faithfully reproduce its mistake across thousands of cases. Keep the rationale independent: a domain rule, a small trusted calculation, a reviewed relation or an external observation. Independence does not require a different programming language or model vendor. It requires that the reason for accepting the result is not simply that the implementation produced it.

---

## The old pyramid and its economics

The familiar test pyramid recommends many small unit tests, fewer service tests and very few full user-interface tests. Martin Fowler's account explains the economic reason: high-level tests have historically been slow, brittle, difficult to diagnose and expensive to author.

Measure the cost beyond authoring

### Local property

Broad input exploration with fast failure localisation.

### System challenge

Observes wiring and user promises; include execution and diagnosis cost.

That advice was never a law about test worth. It was a response to cost. Fowler explicitly notes that if high-level tests become fast, reliable and cheap, fewer lower-level tests may be needed. [Fowler, Test Pyramid](#).

AI changes parts of that cost curve. Models can derive candidate scenarios from specifications, generate fixtures, operate interfaces, explain failures and update repetitive test code. Modern browsers, containers and ephemeral environments make system execution more reproducible. Observability can expose internal state without coupling every test to implementation details.

This does not justify replacing unit tests with a wall of end-to-end scripts. It justifies reopening the allocation. Test where the important behaviour is visible, while retaining lower-level tests where they give faster localisation or prove a difficult invariant.

Authoring cost is only one term in that allocation. A system test can be cheap to generate and expensive to execute or diagnose. A local property test can explore a large input space quickly but miss the wiring between the domain and the user interface. Measure feedback latency and investigation effort alongside the behaviour covered. The economic case for a broader high-level harness becomes stronger when environments are reproducible, failures are attributable and drivers observe stable semantics. Without those conditions, generated test volume can make the old brittleness problem larger rather than solve it.

---

## Inverting the question

The useful inversion is not “put UI tests at the bottom”. It is to start with the product obligation and ask which observation gives the strongest evidence.

Place the observation at the promise

### **Keyboard completion**

Exercise the assembled interaction path.

### **Concurrent ledger balance**

Observe domain arithmetic and database effects under concurrent updates.

If the obligation is that a citizen can complete a service with a keyboard, a component unit test is insufficient. If the obligation is that a ledger balances under concurrent updates, a deterministic domain test and a database concurrency test may be stronger than a browser journey. If the obligation is behavioural equivalence during a refactor, black-box observations across old and new paths are central.

The portfolio of tests should follow evidence value, failure cost and feedback time. Shape is an outcome, not a doctrine.

---

## AI as oracle author, not oracle

Models can help extract relations from domain material. Given a specification for a benefit calculation, an agent can propose that increasing eligible income should not increase a means-tested award beyond defined exceptions. Given a workflow, it can enumerate illegal transitions. Given an API, it can vary ordering, duplication and irrelevant fields.

The relation's acceptance record

### Authority

A source, domain owner and stated preconditions.

### Counterexample

Preserve input, transformation and outputs before deciding which rule failed.

The proposal must be reviewed. A plausible invariant can be false in a domain with thresholds, caps or exceptions. The model is valuable because it expands the candidate set cheaply. The authority remains the specification, observed behaviour and domain owner.

Once accepted, the relation becomes ordinary deterministic code. The model does not decide at test time whether the output "looks right". The harness computes the relation and fails mechanically.

An accepted relation should be a versioned artefact with an owner and a counterexample policy. If a new case violates it, the team needs to distinguish a product defect from an invalid relation or an omitted precondition. The agent can propose that classification, but the evidence should show the input, transformation, outputs and rule being tested. "The model thinks this exception is reasonable" is not a useful acceptance boundary.

The same discipline applies to generated input distributions. Thousands of cases concentrated around ordinary values may add little beyond a few examples. Ask which boundaries and interactions the generator can reach: empty state, repeated identifiers, extreme values, historical versions and sequences with partial failure. Preserve failing seeds or concrete inputs so an intermittent discovery becomes a durable regression case. Generation is valuable because it explores a specified space, not because its output count is large.

---

## Documentation is no longer the same bottleneck

Historically, a broad behavioural harness could be unaffordable because somebody had to understand the system, encode fixtures, write drivers, maintain selectors and explain failures. AI can reduce parts of that mechanical burden; the net saving depends on review, execution and maintenance costs.

What AI can multiply

### Scarce input

One precise, independently justified domain invariant.

### Mechanical expansion

Fixtures, drivers and related cases that exercise that invariant.

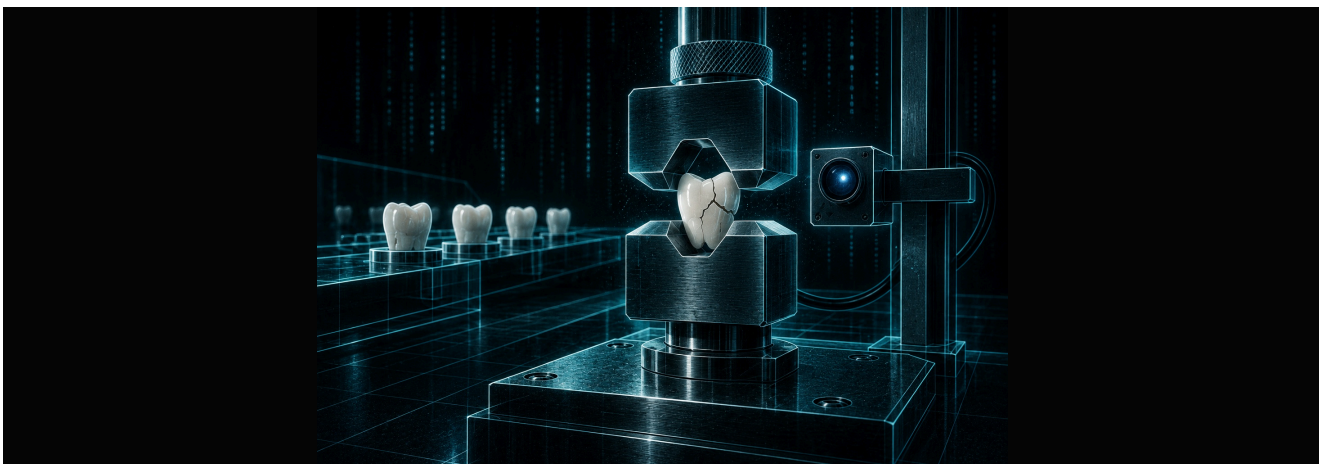
It can read schemas, routes, logs and existing tests to draft a system map. It can turn recorded examples into parameterised cases. It can generate accessible selectors from semantics rather than screen coordinates. It can compare changes with specifications and propose missing documentation.

That does not eliminate the need for human context. It changes its leverage. A domain expert who states one precise invariant can now cause hundreds of related cases to be exercised. The scarce input is the truth of the relation, not the typing of every test.

---

## Mutation proves the harness has teeth

A green suite shows that the current implementation agrees with its tests. It does not show that the tests would notice a relevant defect.



Mutation testing changes operators, conditions or returned values and reruns the suite. A surviving non-equivalent mutant may expose missing execution coverage or an oracle that fails to detect changed behaviour. Triage is needed to distinguish these cases. An observational study of almost 15 million mutants found that developers using mutation testing wrote more tests and improved their suites; it also found evidence linking mutants to real faults.

Mutation score must still be interpreted. Equivalent mutants cannot be killed because they do not change behaviour. Trivial mutations can inflate work. The useful practice is targeted mutation against critical rules and failure mechanisms, not a vanity percentage.

The cited industrial mutation study is observational. Its large corpus and reported association with improved testing practice make it relevant, but do not establish that imposing a mutation-score target causes better outcomes in every team. The practical inference here is to use selected faults to examine the oracle's sensitivity, then measure the maintenance and investigation burden locally. [Petrovic et al., 2021](#).

Agents can help triage survivors, propose higher-value mutants and write candidate tests. Independent gates must decide whether those tests actually kill the mutation without merely encoding the mutated implementation.

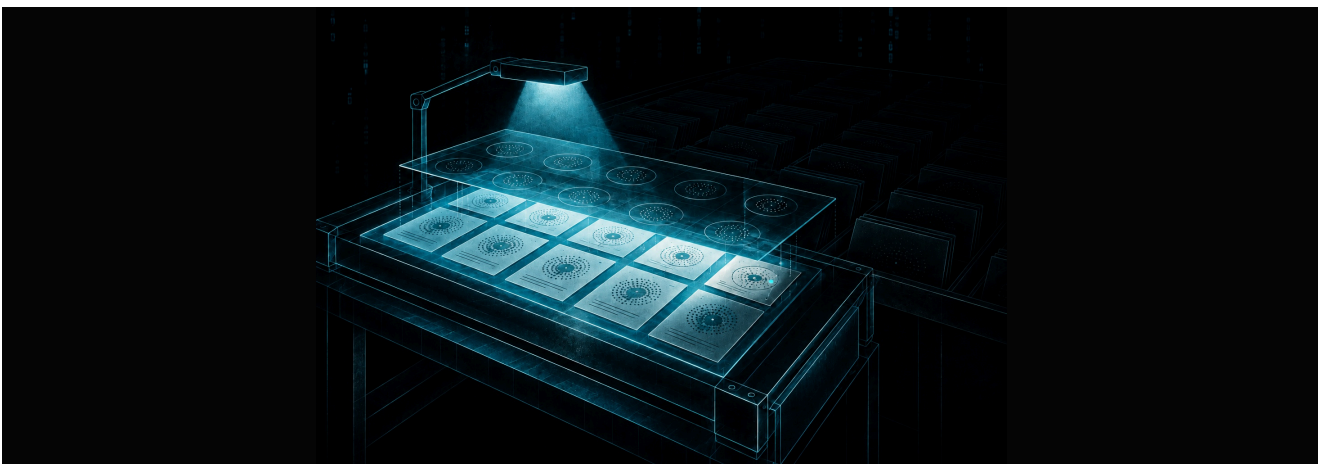
In a constructed permissions example, deliberately remove the ownership predicate from a query. A useful suite should reject the resulting cross-owner exposure. If it remains green, adding more allowed-user examples will not necessarily help; the oracle needs a forbidden access attempt and an assertion about the returned data. This is a direct connection between a plausible fault mechanism and a completion claim.

Mutation results also require triage. A surviving change may be equivalent under valid inputs, outside the stated obligation or invisible because the test environment never reaches it. Those classifications should be supported, not guessed from a score. An equivalent mutation does not justify inventing a test that asserts implementation detail solely to kill it. Conversely, declaring a difficult survivor “equivalent” without examining the reachable states can conceal an important gap. The useful outcome is a better-supported rule and oracle, not a cosmetically improved percentage.

---

## Building a behavioural census

For an existing system, begin by observing what it does.



Capture representative requests, commands, events and workflows with sensitive data removed or controlled. Record outputs and durable effects. Identify equivalence classes and edge cases. Add domain properties and metamorphic transformations. Replay the

census against the current implementation to establish stability.

During a replacement, run the old and new paths against the same corpus. Compare not only status codes but normalised bodies, database effects, emitted events, timing bounds and error semantics. Investigate every divergence. Some differences are defects; some expose undocumented legacy behaviour; some are intentional improvements requiring an explicit decision.

A zero-divergence result is always bounded by the census. It does not prove universal equivalence. That limitation is a strength because it makes the claim falsifiable.

A census needs a sampling argument. Identify which operations, roles, state classes and failure sequences it includes, then make known omissions visible. Raw traffic volume can be misleading: a large corpus dominated by one common read may contain little evidence about rare state-changing operations. Grouping observations by obligation helps reveal that imbalance without pretending that every possible state can be enumerated.

Normalisation is an especially consequential part of the comparator. Removing timestamps can be legitimate when they are incidental; it can be dangerous when a deadline or ordering guarantee is the behaviour under test. Removing generated identifiers may hide a broken relationship between records unless the normaliser preserves identity relationships consistently. Treat each ignored difference as a claim about irrelevance. Review and test that claim, including a deliberately material change that the normaliser must not erase.

---

## Production feedback closes the loop

No pre-release suite contains reality. Production reveals new sequences, scale, integrations and human behaviour.

Turn an incident into a reusable challenge

### Keep the fixture

The concrete failing sequence remains reproducible.

### Recover the mechanism

Find other obligations vulnerable to the same failure pattern.

Convert incidents and representative observations into new examples or relations. Compare telemetry with service-level objectives. Use shadow traffic where lawful and safe. Record which parts of the behaviour space remain unobserved.

Google's research on continuous testing shows that test selection and feedback time matter even at enormous scale: very few tests fail, failures correlate with proximity to changed code, and running everything on every change can be impractical. A mature

harness therefore selects fast gates for each unit while scheduling broader censuses at appropriate boundaries.

Production-derived cases should enter through the same controlled process as authored fixtures. Preserve the relevant sequence and state while removing or containing sensitive data. A support incident often reveals not just a new input but a new ordering: a retry after a timeout, a policy update during an open case or a stale permission cache. Capturing only the final request can lose the mechanism that made the failure possible.

Broader execution should be scheduled according to the claim being made. Fast unit gates support a local change; a release census supports a larger integration boundary; a post-release observation can reveal behaviour neither covered. These are complementary records, not interchangeable badges. If a broad census has not run against the proposed release, an earlier result should remain labelled with its actual revision and scope. Evidence architecture includes the timing and provenance of checks as well as their code.

---

## A practical harness

An evidence-oriented harness contains:

Audit a normaliser

### Permitted omission

Only variation shown to be irrelevant to the observed obligation.

### Required challenge

A deliberately material change must remain visible after normalisation.

1. a versioned catalogue of obligations and relations;
2. generators for valid, invalid and boundary inputs;
3. drivers at the domain, interface and system levels;
4. normalisers for unstable but irrelevant output;
5. comparators for values, state and events;
6. targeted mutations and seeded faults;
7. reproducible environments and data;
8. reports that preserve input, versions and result; and
9. a route for a human to classify genuine divergences.

Every relation needs a rationale and source. Every normaliser needs scrutiny because ignored fields can hide drift. A flaky test should be repaired or replaced. If it is removed, record how its obligation remains covered—or explicitly accept and review the resulting evidence gap.

A concrete harness record can connect an obligation to its whole execution chain. It identifies the input generator and seed, transformation, driver, environment, normaliser, comparator and expected relation. The report preserves observed outputs and durable effects. When a run fails, a reviewer can reproduce the discrepancy without reconstructing the agent session that wrote the test.

The harness should make investigation cheaper than suppression. Distinguish environment failures from behavioural failures; retain enough context to diagnose each; and provide an explicit route for approving a changed relation. A flaky driver that intermittently loses the action it is supposed to perform is not evidence against the product rule. A consistently reproducible state divergence is. Collapsing both into one red indicator encourages indiscriminate reruns and teaches the team to treat refusal as noise.

---

## Limits and counterarguments

Generated tests can restate the implementation rather than challenge it. Recorded behaviour can preserve defects. Metamorphic relations can be wrong. High-level tests can remain slow and brittle. AI-written maintenance can create a larger volume of low-value checks.

A large suite can agree for the wrong reason

### Shared mistake

Expected values copied from the production implementation.

### Independent rationale

A domain rule, trusted small calculation or reviewed relation.

The controls follow from those risks: derive tests from independent specifications and observed obligations; review relations with domain owners; use mutation to test refusal power; keep drivers semantic; measure flakiness and feedback time; and delete tests that do not support a decision.

The aim is not maximum test count. It is maximum justified confidence per unit of time.

An economic comparison should retain the same release obligation while varying where and how it is tested. Include authoring, execution, false-failure diagnosis, environment maintenance and repair after legitimate contract changes. A generated browser suite may reduce writing effort but increase all the other terms. A domain relation may cheaply explore many inputs but require a separate integration challenge to establish that users reach the tested calculation.

Keep fast local feedback for faults it can detect and use broader observation for obligations that cross boundaries. Report which defect classes each layer actually caught, not just its execution count. If higher-level checks become cheaper and more reliable, the allocation may change; if they remain unstable, adding more generated journeys weakens the feedback loop. This is a testable allocation hypothesis, not evidence that AI has already overturned testing economics everywhere.

The strongest counterargument is that a larger automated harness creates its own maintenance system. Relations become stale, drivers drift and diagnosis consumes expertise. The proposal is worthwhile only if the evidence helps the organisation make better release and recovery decisions at an acceptable total cost. Measure time to investigate failures, recurring flakiness, faults found and the burden of updating legitimate contract changes. Test count is a poor substitute for those observations.

There is also an irreducible semantic limit. A relation can be perfectly implemented and still encode an unjustified domain assumption. Metamorphic testing cannot manufacture policy legitimacy or recover a requirement nobody supplied. Its contribution is narrower and useful: it makes a stated relationship executable across many cases, then permits adversarial checks of that execution. The organisation still owns the truth and scope of the relationship it chooses to enforce.

---

## Conclusion

Where AI reduces total harness cost, teams can test more of the behaviour that matters—but only if generated assertions are not treated as evidence by default.

Begin with obligations. Turn domain relationships into metamorphic oracles. Combine them with examples, properties, differential observation and mutation. Test at the level where the product promise is visible. Use AI to reduce the mechanical burden, then make the accepted oracle deterministic and capable of refusing the work.

The pyramid was an economic response. When the economics change, the evidence architecture should change with it.

---

## References

- Chen, Cheung and Yiu, [Metamorphic Testing: A New Approach for Generating Next Test Cases](#), 1998; uploaded 2020.
- Petrovic et al., [Does Mutation Testing Improve Testing Practices?](#), 2021.
- Jain et al., [Mind the Gap: The Difference Between Coverage and Mutation Score Can Guide Testing Efforts](#), 2023.
- Fowler, [Test Pyramid](#), 2012.
- Memon et al., [Taming Google-Scale Continuous Testing](#), 2017.