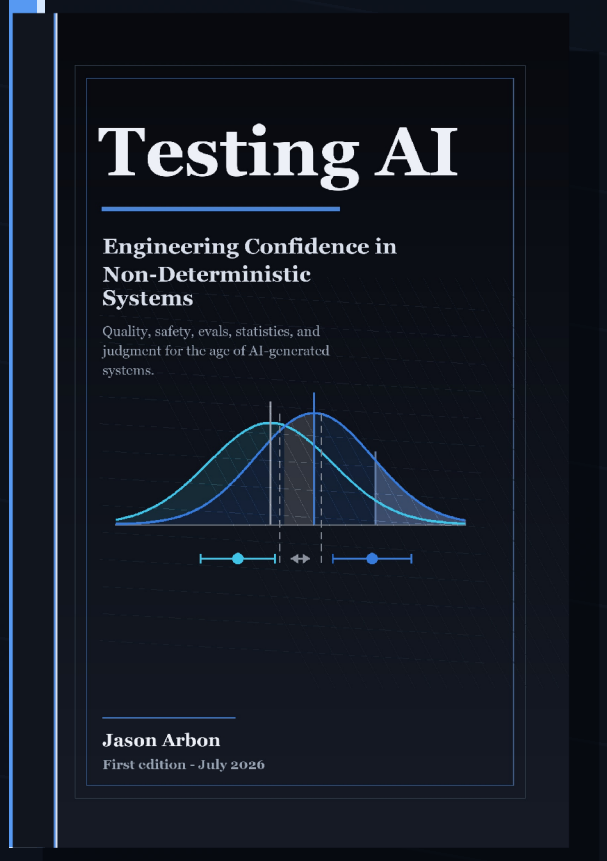
GENERATED CODE CHANGES THE JOB

AI makes code cheap. Validation is the scarce work.

A plausible diff can compile, pass a few tests, and still be wrong everywhere that the real system is complicated.

Swipe for the evidence AI-generated code needs before it deserves production.



THE ROLES ARE MERGING

AI builders become confidence engineers.

Classic handoff

- Developer builds
- Tester checks
- PM decides intent
- Security reviews later

AI-era loop

- Define quality and risk
- Direct coding agents
- Generate and inspect evidence
- Make a ship recommendation

The scarce skill is not producing a first draft. It is knowing what evidence makes the draft safe enough to keep.

Build the thing, understand how it can fail, and know what outcome the user actually needed.

PLAUSIBLE IS NOT CORRECT

Clean code can solve a nearby problem.

```
// Requirement: discount
only eligible items
function discount(cart) {
  return cart.total * 0.10;
}

// passes simple happy-path
test
```

```
// Keep the business
boundary explicit
function discount(cart) {
  return cart.items
    .filter(isEligible)
    .reduce(sumTenPercent,
0);
}
```

> The dangerous bug is not bad syntax. It is a tidy implementation that quietly made the wrong assumption.

Test neighboring cases: exceptions, boundaries, permissions, time zones, ordering rules, nulls, and product vocabulary.

FAILURE TAXONOMY

Ask: what assumption did the model make that an expert would not?

Requirement drift

It ignored an exception, business rule, or edge clause that changes the outcome.

Boundary mistake

Inclusive ranges, empty inputs, max lengths, pagination, or daylight-saving time.

Fake generality

A broad helper seems reusable but erases a critical domain rule.

Silent fallback

It catches an error, returns a default, and hides the failure behind a friendly result.

Behavioral evidence beats idiomatic-looking code. The best counterexample often looks almost like the original request.

INTEGRATION BOUNDARIES

Working in isolation is not working in your system.

Model world

- Imagined helper API
- Newest SDK pattern
- Happy mock response
- One ideal environment

Local reality

- Installed versions and types
- Existing wrappers and conventions
- Auth scopes, timeouts, retries
- Real sandbox and production-like config

Use type checks, contract tests, sandbox calls, lockfile review, and production-like configuration tests near the real boundary.

SECURITY AND PRIVACY

Functional code can still be unsafe code.

Authorization

Logged in is not authorized. Test role, tenant, ownership, and object-level boundaries.

Input and secrets

Challenge unsafe paths, commands, HTML, files, serialization, keys, logs, and default scopes.

Abuse cases

Ask what an attacker, rogue tenant, malicious document, or injected prompt can do with the path.

> Make the coding agent review its own completed patch as a skeptical security reviewer. It often catches a missing validation or unsafe default once it stops optimizing only for feature completion.

Tutorial code is usually written to demonstrate one idea. Production code must survive abuse, privacy obligations, and real operational failure.

ARCHITECTURE DEBT

The first change works. The second change tells you whether it belongs.

Fast local patch

Duplicated business logic, a new parallel helper, broad coupling, and a clever abstraction no one else can safely change.

System-fit change

Existing patterns, ownership boundaries, domain naming, narrow scope, and a clear path for the next modification.

Review change shape, not only behavior. Ask the agent to make one realistic follow-up change and see where the design fights back.

GENERATED TESTS

Coverage can be a confidence machine for the bug.

Implementation mirror

The agent writes code and tests that merely prove the code returns what it already returns.

Mocks, snapshots, and happy paths can make this look comprehensive.

Requirement challenge

Counterexamples, properties, boundaries, contract failures, historical defects, and mutations test whether the requirement holds.

Ask: would this test fail for a realistic defect?

AI-generated tests are useful starting material. They are not proof that the behavior, oracle, or coverage target is correct.

FROM THE FIELD

Automation can verify the wrong thing perfectly.

The oracle was wrong.

At BizTalk Server, an automated suite for a business-rules engine was green, but the expected behavior encoded in the tests was wrong. The automation had made the bad assumption look official.

Automation does not create truth. It scales whatever requirement, expectation, or blind spot you give it.

Requirement review	Negative cases	Mutation tests	Defect replay

Before trusting a generated test, ask the boring essential question: would it fail if the product violated the actual requirement?

INTERNAL SIGNALS AND VALIDATION LIMITS

Use model internals as triage evidence, never as proof.

Look inside

- Attention and activation patterns
- Concept probes and logit trends
- Known-good versus known-bad comparisons

Return to behavior

- Validate patterns with output tests
- Compare versions and fine-tunes
- Use drift to focus investigation

Interpretability can make a mistake less mysterious. It cannot prove that the model reasoned correctly or that a patch is safe.

Internal diagnostics, static analysis, formal methods, runtime tests, traces, and production monitoring are complementary evidence layers.

PUT IT INTO PRACTICE

Turn AI-generated code into evidence-backed software.

AI can create a huge diff overnight. The durable work is mapping the impacted contracts, failure modes, tests, security boundaries, and rollout risks. IcebergQA helps teams build that validation discipline.

Talk to IcebergQA