

AI Testing Lab Study, part two

AI IS NOT ENOUGH

THE ROLE OF THE AI OPERATOR IN TEST QUALITY

Table of contents

Executive summary 3

Introduction 5

Study design 5

Results 8

Why some teams found bugs & others did not 12

The AI Operator Model 19

From results to implementation 20

Discussion & implications 22

Conclusions 23

Appendices 25



1. EXECUTIVE **SUMMARY**

Problem

In the first edition of the Sii Testing Lab study, we tested whether AI helps test automation engineers. It does, by a wide margin: more working tests in the same amount of time, with no drop in average quality. One issue, however, has a much greater impact on risk. A larger test suite reduces production risk only if it can detect significant defects. Even an extensive test suite may fail to detect defects that are visible directly in an API response. The second edition therefore asks whether more tests actually mean better production protection.

KEY FINDINGS

The study included sixteen teams using AI tools to test a system with nineteen intentionally seeded defects.

- **AI still significantly increases productivity.** The conclusion from the first edition was fully confirmed: models accelerate the creation of testing infrastructure.
- **The more costly a defect would be in production, the less often it was detected.** Thirteen of sixteen teams caught a simple contract defect; none detected the concurrency defect.
- **Differences in detection rates were associated with how participants worked with AI.** Engineering skill levels were comparable & the tools were the same, so the differences came down to decisions: what to test & how to measure correctness.
- **A typical submission detected about 6 of 19 seeded defects.** Detection effectiveness did not keep pace with productivity.
- **No team confirmed two of the seeded defects.** Both were in the concurrency & decision layers, exactly where a production failure is most expensive.
- **The effectiveness of a test suite depends more on what it verifies than on the number of test cases.** The most efficient submission required 7.5 test cases per detected defect. A submission with 196 test cases detected one defect.

Key takeaway

AI supports test implementation and exploration, while testers remain responsible for test strategy and evaluating the results.



WHAT THIS MEANS FOR ORGANIZATIONS

Measure detection effectiveness, not test count.

The number of test cases is a supporting metric & says nothing about the risk that remains in production.

Adopt an operator standard.

The five quality gates provide the framework; the fifteen practical rules provide the playbook.

Embed quality gates in the process.

Require an independent oracle & a "teeth check" in the definition of done & code review.

Assess team capability.

An assessment in an environment with seeded defects shows how many real bugs the team catches & where gaps remain across the defect layers.

2. Introduction

In the first edition of the Sii Testing Lab study, we tested whether AI helps test automation engineers. It does, by a wide margin. We compared AI-assisted teams with "old-school" teams, & the difference was impossible to miss: model support produced many more working tests in the same amount of time, with no drop in average quality. Productivity increases, & we can quantify it.

We closed that report by announcing that the next edition would go one step further & answer a harder, more business-critical question: how to use AI to maximize outcomes without trade-offs. The reason was practical. More tests translate into better production protection only under certain conditions. You can generate an extensive framework, hundreds of scenarios & an impressive report, yet still miss a bug contained in a single API response. The key question is whether the tests can detect defects that pose a significant production risk.

We therefore changed the starting point. In the first edition, we studied whether AI helps, so the tool was the variable. In the second, all sixteen teams worked with a model, on the same system & under the same time constraints.

The second edition focused on differences in how participants guided AI and how effectively their tests detected defects.



3. Study design

OBJECTIVE

In this edition, we went beyond simply measuring how many tests can be written with AI. We wanted to see how testers & AI tools perform in a realistic domain environment: one that requires reading documentation, understanding the business process, choosing a strategy, building a framework, writing tests, running them, interpreting the results & distinguishing a real defect from a false lead.

HYPOTHESIS

We assumed that, with comparable engineering skill levels & the same tools, the outcome would depend on how people worked with the model. We expected the differences to emerge primarily in defect layers that require domain reasoning & state analysis, rather than in the quality of the test code itself.

ENVIRONMENT: LOANFLOW QA

For the study, we built our own environment: LoanFlow QA, a loan application processing system designed as a complete test system. Participants received a frontend, backend API, database, mock external scoring service, user roles (customer, analyst & administrator), workflow, requirements documentation, OpenAPI specification, accounts & test data. This made it a realistic environment, far removed from a simple form or isolated coding exercise (code kata).

LoanFlow was designed to challenge participants to apply their testing experience and domain knowledge.

SEEDED DEFECTS

We hid a pool of 19 intentionally seeded defects in the system, distributed across four difficulty layers: from defects visible after a single request to defects requiring reasoning about business consequences.

- **Simple defects**
  Validation, HTTP statuses, configuration & API response structure.
- **Cross-functional defects**
  State, sessions, concurrency & changes in data over time.
- **Domain defects**
  Require understanding of financial calculations & the loan process, for example calculating the installment amount.
- **Documentation inconsistencies**
  Differences between requirements, OpenAPI & actual system behavior.

This distribution was intentional. In a real project, some bugs are visible after a single curl command, while others require reasoning about business consequences. A field may exist but contain the wrong value. An endpoint may return the correct status while violating a scoring rule. Documentation may look credible while describing a product or endpoint that does not exist in the system.

PARTICIPANTS & EVALUATION SCOPE

We evaluated 16 submissions. Each typically included its own test framework, containing tests, reports, bug reports, a strategy description & traces of work with AI. The latter were particularly important to the rigor of the analysis because we were interested in both the final code & the process that produced it.

Each bug report was verified against the LoanFlow code, participant documentation, OpenAPI & the internal answer key. In total, we reviewed about 350 bug-report files. As in the previous study, the evaluation had two dimensions. The participant template therefore also included directories for AI work traces & a post-mortem.



OUTCOME

How many seeded defects were found, how difficult they were & whether the reports were valid.



PROCESS

How the participant worked with AI: whether they built context, guided the model through the specification, verified responses, rejected incorrect suggestions & reported defects repeatably.

This separation proved critical. A defect ranking alone would tell us who found more defects. Process analysis adds another dimension by telling us why.

STUDY LIMITATIONS

We state the boundaries of our conclusions explicitly: sixteen submissions, one domain, mixed tools. The practices described below are patterns consistently observed among effective teams & absent from the others. We treat them as correlations, without claiming causal proof. That is why we report mechanisms & examples, not coefficients.

The preliminary ranking & later detailed verification were not identical. That is normal in a study like this: some reports combined several phenomena, some concerned real but unseeded defects & some resembled a defect in the answer key but proved to be a different problem after inspection. A good example was a bug that allowed a second analyst to take over an application. It was a real, frequently reported defect, but it was not the same as the seeded race condition in concurrent analyst decisions. The distinction matters because it shows that evaluating bug hunting also requires a precise oracle.

At the same time, participants found many real, unplanned defects: mismatches between documentation & implementation, validation errors, document authorization issues, UI inconsistencies, cases where invalid values were saved before validation & endpoints returning 500 for an invalid parameter. This confirmed that the environment was realistic enough. Participants did not merely hit our seeded checkpoints; they also uncovered real consequences of implementation decisions.

4. Results

Productivity confirmed

Most participants built a real automated testing framework: most often Playwright with TypeScript, sometimes Java, C# or pytest. Repositories included page objects, fixtures, helpers, API wrappers, HTML reports, AI context files & one-command execution scripts. From an engineering perspective, most repositories met production-project standards & the median process quality was fairly high. Participants went beyond generating individual tests & could establish a project structure. This confirms the first-edition finding: AI significantly accelerates the creation of testing infrastructure.

Defect detection effectiveness

Defect detection effectiveness, however, did not increase in proportion to test count. A typical submission found about 6 of 19 seeded defects. The most common were contract & calculation issues: an incorrect HTTP status, a wrong installment amount, a fee calculated using the wrong rate, a product missing despite being specified in requirements or an endpoint declared in OpenAPI but not implemented by the backend.

Detection rates were considerably lower for more complex defects. This included concurrency, state changes, sessions after a password change, the dependency of an analyst decision on scoring, document limits after an analyst change & pagination stability with identical timestamps. No team confirmed two of the seeded defects in its reports. These were exactly the areas that ordinary happy-path coverage or a test checking only response structure would miss.

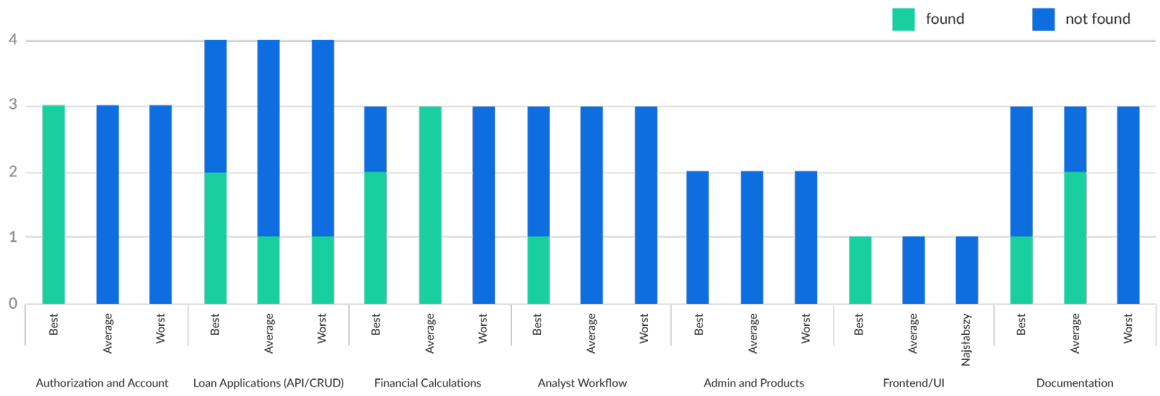


Figure 1. Test coverage expressed as the ratio of detected defects to the total number of defects.

Risk curve

Detection rates followed a clear pattern: they fell exactly where the cost of a production defect rises. Thirteen of sixteen engineers caught the incorrect HTTP status when creating an application. Thirteen also found the incorrect fee & mismatched annuity payment. Only six detected DTI calculated from gross rather than net income. Two noticed that an old session remained active after a password change.

No one detected the race condition in a concurrent analyst decision or approval of an application despite a negative credit score.

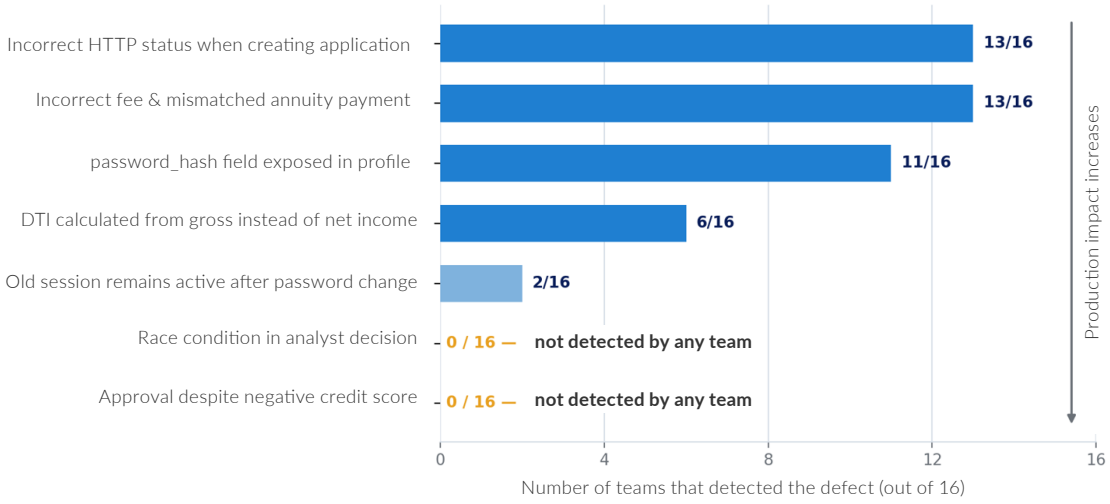


Figure 2. Detection rate of seeded defects across 16 teams.

Most participants successfully detected relatively straightforward API contract defects.

AI helps build tests but does not guarantee they test the right things.

Coverage efficiency

Analysis of the test suites led to an interesting conclusion: most participants did not write too many tests in the sense of empty filler. In many repositories, the ratio of genuinely useful tests to the total number of cases was high. The results did not suggest that participants were simply generating large numbers of irrelevant tests. The shape of coverage, however, mattered more than its size.

Weaker repositories could contain many tests, but they tested statuses, element visibility & JSON structure rather than business values. They did not verify calculations. They did not assert the absence of forbidden fields. They did not build multi-actor scenarios. They did not test concurrency. They did not examine moments when data changes after an application is created.

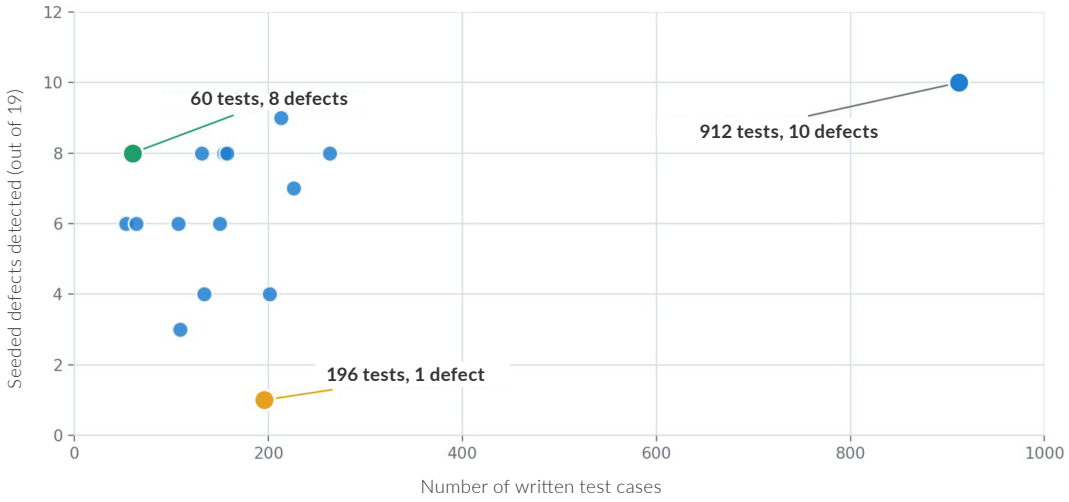


Figure 3. Number of written test cases vs. number of seeded defects detected (16 submissions).

A submission with 60 test cases detected eight defects. A submission with 196 cases detected one. Fewer tests with sharp assertions outperformed more tests with weak assertions.

Coverage efficiency: test cases per detected defect

Submission profile	Test cases per detected defect	Position in detection group
Leanest ("sharp assertions")	7,5	highest
Typical submission	approx. 25–40	middle of the pack
Heaviest ("many tests")	approx. 90 to 196	below the leanest

This has a direct impact on cost & maintenance. A small, precise test suite is cheaper to maintain & more effective than a large framework that does not test the right hypotheses. Optimizing for test-case count means optimizing for a metric that does not protect production.

Cost vs. effectiveness

The third dimension, alongside test count & detection effectiveness, is submission cost. Bubble size in the chart below represents cost on a relative scale, where 100% is the most expensive submission in the group. The cost shown in the chart was estimated based on token costs using the Anthropic API. Values were normalized to a relative scale where 100% represents the most expensive submission.

In practice, participants' actual costs were lower because they mainly used subscription plans (e.g. Claude Max or ChatGPT) rather than API billing.

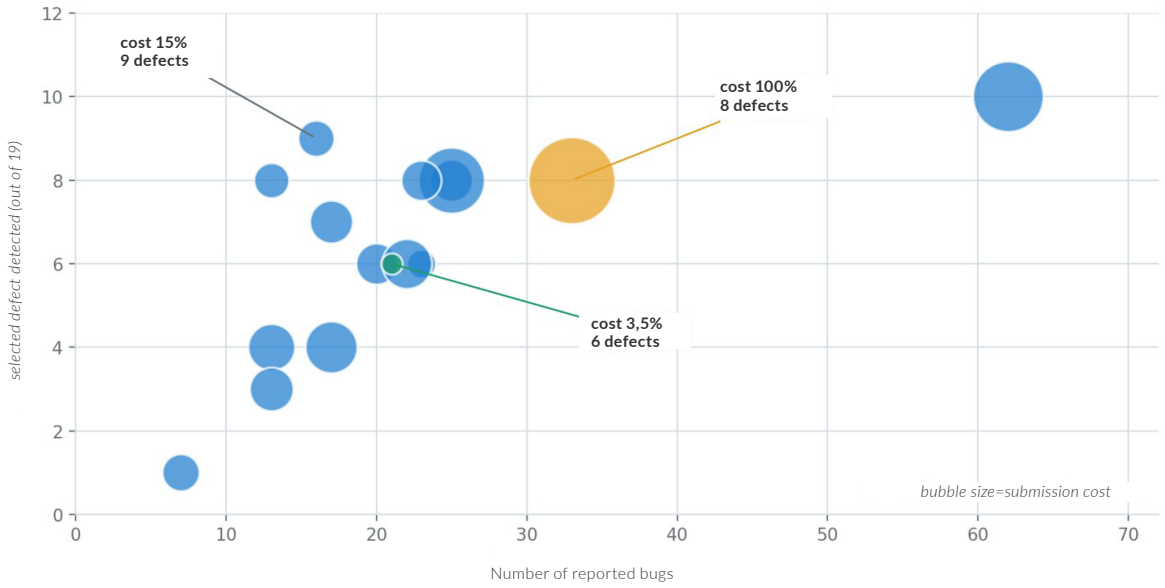


Figure 4. Reported bugs, detected seeded defects & submission cost (16 submissions).

The spread in cost was wider than the spread in results. Five teams detected exactly eight seeded defects; the costs of those submissions were 14.1%, 19.1%, 21.3%, 56.6% & 100%, respectively.

The same outcome therefore cost seven times more or seven times less depending on the working method. The least expensive submission in the group, at 3.5% of relative cost, detected six defects, the same as a typical submission in the study.

A submission costing 15% detected nine defects & performed better than the most expensive submission in the entire study, which stopped at eight defects despite a 100% cost. On a per-detected-defect basis, costs differed by nearly thirtyfold across submissions.

What actually differentiated the results

Engineering skill levels were comparable. What differentiated effectiveness was the decision layer: what to test, how to measure correctness & how to guide AI. AI agents therefore require control across the entire process, not just evaluation of the quality & quantity of the code produced.

INTERPRETATION OF **RESULTS**

- Productivity does not automatically translate into effectiveness. More tests did not mean more bugs caught.
- Detection difficulty rises with the business cost of a defect. The most expensive defects were detected least often or not at all.
- Differences between teams were not driven by tools or engineering skill level, but by how they worked with the model.
- What the tests verified mattered more than how many tests were written.

5. Why some teams found bugs & others did not

The clearest illustration of this difference came down to a single moment.
One participant's independent oracle calculated an annuity payment of 315.68, while the API returned 308.18. The model, helpful & optimistic as models tend to be, proposed the simplest possible solution: change the oracle to match the application. Had the participant agreed, the discrepancy would have disappeared without a trace, taking the seeded defect with it. Instead, she recalculated the payment independently using a third tool, confirmed that the backend calculation was wrong & reported the bug.



When the oracle & the system disagree,
never assume by default **that the oracle is wrong.**

That single decision captures the difference between accepting AI output & operating a quality process. When engineering skill is comparable, the way of working determines the outcome. Five practices consistently appeared in effective teams & were absent from the others. Each is a gate placed on the model's work: a point where the human does not simply accept the result but verifies it. Together, they form the operational definition of the capability we offer clients.

GATE 1

Independent correctness oracle

PROBLEM

A test that checks whether a field exists confirms only that the system returned something. A test with an oracle checks whether the system returned the truth. Teams without an oracle could have extensive frameworks, but their assertions stopped at the HTTP status, element visibility or schema compliance. In this model, the application can return the wrong payment indefinitely while the tests stay green.

EXAMPLE FROM THE STUDY

In LoanFlow, this was especially visible in financial calculations. The requirements described the annuity-payment formula, the DTI calculation method & fee rules. Good tests went beyond checking whether the monthlyInstallment field existed: they independently calculated the expected payment, fee or DTI ratio, then compared the result with the API response. That is why teams using an independent oracle found financial defects much more often.

MECHANISM

A correctness oracle calculates the expected result from the specification rather than from application behavior.
A full formula was not always necessary. In some cases, a metamorphic invariant was enough: a simple property that must hold regardless of implementation details. Example: with a positive interest rate, the monthly payment should be greater than the loan amount divided by the number of months. Such a test does not need the full annuity formula, yet it still detects a class of defects that an ordinary UI test would miss.

RECOMMENDATION

If a value matters to the business, calculate it independently instead of asking the application what it should be.
This is a process requirement, regardless of technology.

GATE 2

Assertions at the right layer: API before UI

PROBLEM

The UI matters, but money, statuses, permissions & system decisions usually live in the API. If tests stop at the screen, they may miss what actually determines risk. UI coverage without probing the API creates false confidence exactly where the risk is highest.

EXAMPLE FROM THE STUDY

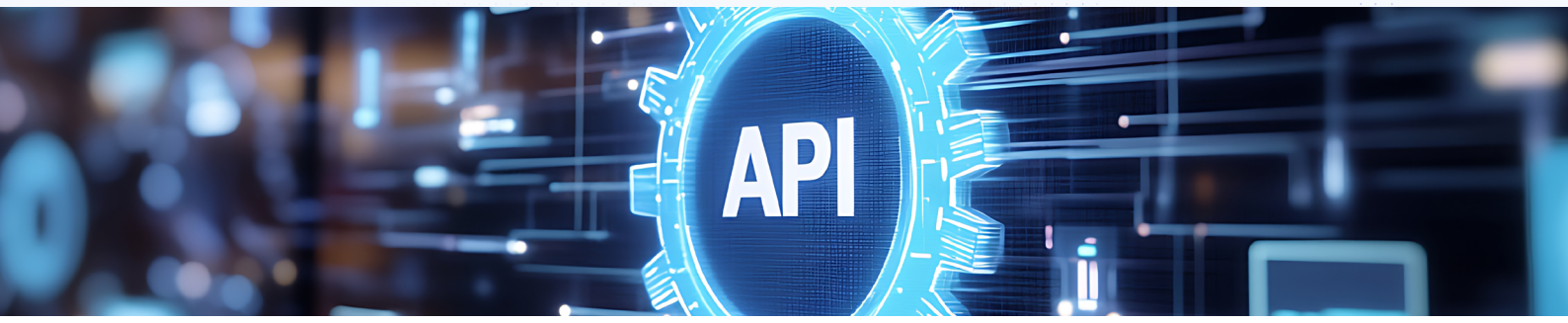
In LoanFlow, the frontend deliberately calculated some values correctly while the backend returned incorrect values. A tester who checked only the UI could therefore get a false sense of security. The assertion passed because it was checking the layer that happened to be correct. The defect was present in the backend API, not the frontend.

MECHANISM

Strong submissions often started with an API probe: log in the user, call the endpoint, inspect the response, compare it with the documentation, then turn it into an automated test. This workflow had two advantages. First, it quickly separated application defects from incorrect test assumptions. Second, it forced teams to inspect the data, not just the screen flow.

RECOMMENDATION

For business-critical values, assertions should reach the API layer. A UI test verifies presentation, not calculation correctness.



GATE 3

Separate adversarial pass

PROBLEM

Many teams did what came naturally: they read the requirements, wrote tests to cover functionality & found bugs along the way. This produced a solid baseline but quickly hit a ceiling. Coverage tests usually confirm that the system works as expected. Bug hunting, as the practical expression of a tester mindset, requires a different approach: actively looking for ways the system can break.

EXAMPLE FROM THE STUDY

The best strategies separated these two modes. First came baseline coverage: login, application creation, customer, analyst & administrator workflows, document upload & validation. Then teams ran a separate adversarial pass aimed at specific risks:

- Risk: "scoring may not block a decision" becomes a scenario: create an application with a negative score, assign an analyst, attempt approval & expect it to be blocked.
- Risk: "a product change may not affect existing applications" becomes a cross-role scenario: the customer creates an application, the administrator changes the product parameters & the analyst attempts a decision.
- Risk: "the document-request limit may be calculated incorrectly" becomes a scenario with two analysts, a reassignment & another document request.

MECHANISM

The risk register is the input to this gate. It is a sound & necessary practice because it gives the adversarial pass a target list & structures thinking about where the system can break. Its value, however, is conditional: the register protects production only when every item is closed out, meaning each named risk is completed with a scenario containing an assertion capable of disproving it & passing the "teeth check." A register containing items without linked tests that can fail remains a wish list. We therefore treat traceability: risk -> scenario -> assertion as part of the definition of done, not an optional add-on.

The top-performing submission shows how easy it is to believe that naming a risk is enough. Of the six seeded defects it missed, four were in areas explicitly listed in its own risk register. The mechanism was the same every time: risk named, test written, test even touched the right area, but it stayed green because it never asserted the violation itself. Both defects that no one found were exactly in this category.

RECOMMENDATION

The remedy is entirely mechanical: before a risk reaches the model, translate it into a violation scenario with an asserted rejection, then subject the finished test to the teeth check by inverting the expected value & checking whether the test fails. A test that does not fail when expected behavior is deliberately violated cannot reliably detect that violation. Some defects remained undiscovered precisely because they required lateral thinking: concurrent requests, a cross-origin probe, two actors, state changes over time or a pagination stability test. Those tests have to be conceived, & that most often comes from the tester's own experience.



GATE 4

Quick security & response-hygiene sweep

PROBLEM

A strategy focused on business requirements, flows & expected contract behavior misses an entire class of defects that are cheap to detect yet critical from reputational & legal perspectives.

EXAMPLE FROM THE STUDY

One of the most instructive examples was the exposure of password_hash in the user profile response. Technically, this is a very cheap bug to detect: log in, call /auth/profile & check that the response does not contain a hash field. Yet five of sixteen teams missed it. This was a consequence of how the teams designed their test strategies. It checked whether the profile worked but did not review the response for fields that should never be there.

MECHANISM

A response-hygiene sweep complements domain tests & remains independent of them. That is exactly why it catches issues that are easy to miss when the entire process focuses on the business flow. After baseline coverage, it is worth running a separate, short pass asking:

- Do API responses expose technical or sensitive fields?
- Do protected endpoints actually require authorization?
- Does CORS behave consistently?
- Does validation reject obvious garbage input?
- Can user roles see data outside their permitted scope?

RECOMMENDATION

This is one of the cheapest practices to replicate. Plan it as a separate pass after baseline coverage is in place.

GATE 5

A green test is not proof

PROBLEM

The most important concept that kept resurfacing in the analysis was greenwashing. In testing, greenwashing is when tests are green but prove nothing meaningful. Sometimes the assertion is too weak. Sometimes the test checks the wrong layer. Sometimes AI fixes the test by aligning expectations with incorrect application behavior. The model is very good at modifying tests so they pass. Making tests pass is not the same as ensuring that they correctly validate system behavior.

EXAMPLE FROM THE STUDY

We saw several recurring patterns across the submissions:

- The application-creation test accepted both 200 & 201, masking the HTTP status defect.
- The profile test checked only a few expected fields & did not assert that password_hash was absent.
- The password-change test verified that the old password no longer worked but did not verify that the old JWT token had been invalidated.
- The DTI test compared data with another application-generated result instead of an independent expected value.
- Some red tests were labeled "flaky" before anyone checked whether they were pointing to a real bug.

MECHANISM

The strongest processes included safeguards against this. We saw red-by-design repros: tests that describe correct behavior & are marked as expected failures until the bug is fixed. We saw xfail, Ignore, separate bug-hunting directories, assertion linters, mutation checks, context-file rules such as "do not weaken an assertion just to make the test pass" & independent re-verification by a second agent or a human.

Greenwashing also has a mirror image, which we called red-washing: withdrawing a real defect as non-reproducible without actually verifying it. In the study, one team correctly detected a critical contract discrepancy, an endpoint that existed in the specification but was prohibited by the requirements, then closed the report based on an assumed result that was never run. Closing a valid defect report without proper verification creates a risk of leaving a real issue unresolved.

RECOMMENDATION

A good test should be strict. It should fail immediately when the system violates a defined requirement. A red result is a defect hypothesis until verification shows that the test is flaky. The model must not weaken assertions just to make the test pass, & "cannot reproduce" is an action item, not a conclusion.

AI OPERATOR BOOKLET

We distilled these five practices into concrete operating rules derived directly from participant repositories.

The full 15-rule checklist is available to download.



6. The AI Operator Model

The central thesis of this edition is simple: AI needs more from a human than approval. The approver role, which consists of asking for tests, receiving the code, running it & celebrating a green report, is redundant in relation to the model because the model can perform it on its own. In autopilot mode, the LLM is both executor & approver: it generates a solution & accepts it as good. Passing test results do not necessarily indicate that the tests can detect actual defects. Human involvement should extend beyond approving AI-generated code to independently validating test assumptions and results. It needs an operator: someone who guides the model & verifies its outputs instead of trusting them by default. The operator defines the correctness oracle, selects risks & designs attacks, makes sure assertions have teeth, requires full reproduction & separates real defects from noise. In other words, the operator does exactly what the model will not contribute on its own.

The difference between these two attitudes is the dividing line between a role AI can perform on its own & a role that creates its value:

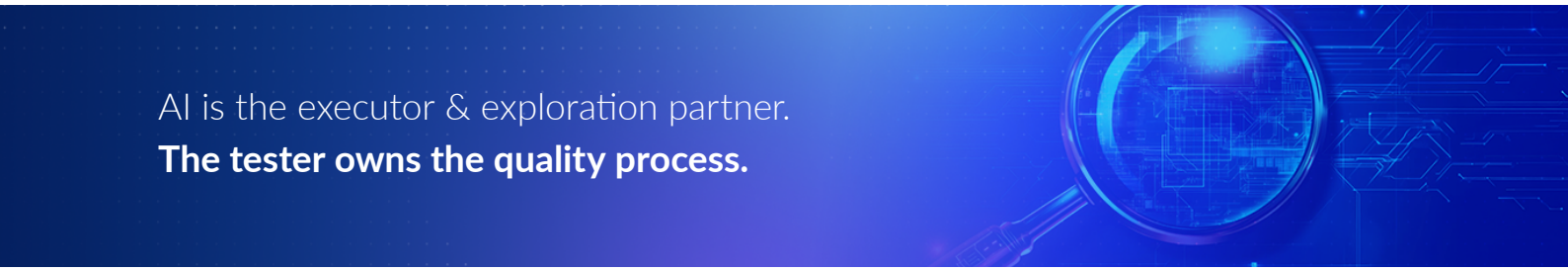
Approver	Operator
Asks the model for tests & accepts the code	Guides the model through the domain, risks & specification
Celebrates a green report	Independently validates passing test results
Fixes technical errors	Requires an oracle, guards assertions & enforces full reproduction
Labels a red test "flaky"	Treats a red test as a defect hypothesis
Success metric is test count	Measures success by the ability to detect defects

The left column lists tasks the model can perform without us. The right describes a capability that we defined & learned to measure in this study, distinct from prompting skill. The second part of the study strongly confirmed that working with AI in testing is separate competency. An effective tester in the AI era is first & foremost an operator of the quality process; skillful prompting alone is not enough.

In the strongest examples, people did things the model does not guarantee on its own:

- noticed when AI tried to align the oracle with an incorrect application result,
- stopped false-positive reports,
- required reproduction of the exact user path rather than only a synthetic request,
- lowered or raised severity based on business impact,
- decided when to abandon a rabbit hole & change strategy.

This expands the boundaries of how we define the test automation engineer. The role is becoming less about manually producing code & more about designing a process that can demonstrate whether expected quality levels have been achieved: which hypotheses we test, how we measure correctness, how we identify tests that pass despite incorrect application behavior & where using the model produces the greatest return.



7. From results to implementation

We translated the findings into three elements ready to use in projects.

Operator standard

The five gates provide the framework; the fifteen practical rules provide the playbook. Each rule comes from a specific pattern observed in participant repositories. The team gets a proven way of working that can be adopted from day one.

Embedding the gates in the process

Each of the five gates can be embedded in the definition of done for a testing task and in code review: from requiring an independent oracle for business-critical values to verifying that the test fails when the expected value is inverted. Quality then stops depending on the goodwill of an individual engineer and becomes a condition that must be met.

Assessment of existing teams

The study methodology, an environment with seeded defects, an answer key & verification of reports against the code, lets us measure the same thing in a client team that we measured here: the ability to detect bugs that carry real cost. The result is a concrete number: how many real defects the team catches & where gaps exist across the difficulty layers.

These three elements work together: the standard defines how to work, checkpoints ensure the team works that way & measurement shows whether it works. That is the difference between hoping tests protect production & knowing they do.

What this means for the budget

Quality gates primarily change the structure of cost while the total remains similar. The data in Section 4.5 show this directly: five teams each detected eight defects at costs ranging from 14.1% to 100%, while cost per detected defect differed by nearly thirtyfold. Higher cost did not buy better detection.

A smaller, more precise suite is also cheaper to maintain because fewer tests need to be repaired after interface changes & less green noise can mask a regression.

The defects near the bottom of the defect table (Appendix A), including decision races, approvals despite negative scoring & user-data exposure, are the most expensive in production: incorrect customer settlements, credit decisions that violate the system's own rules & security incidents with legal consequences. The gates are designed to cover them.

AI OPERATOR BOOKLET

We captured the operator standard as fifteen implementation-ready rules grouped under five gates in a printable checklist.

Download the AI Operator Booklet.

8. Discussion & implications

Positive pessimism: the tester mindset, always on

A good tester asks "how can I break this?" instead of "does it work?" It is the professional bias we have always cultivated in QA: the system is guilty until proven innocent. Our study showed that when working with AI, this mindset moves from a personality trait to a working method, one that must remain active continuously.

The point is that an LLM is optimistic by nature. It will generate tests that pass, a report that looks complete & assertions that take no real risk. If the operator accepts that optimism at face value, they may end up with many passing tests without gaining sufficient confidence in system correctness. Teams that found real bugs did the opposite: they treated every model output as a hypothesis to disprove. They independently verified whether passing tests could detect the defects they were intended to catch.

This form of pessimism is positive for two reasons. First, it introduces additional checks to validate AI-generated outputs. Second, paradoxically, distrust is what gets the most out of the model: an operator who systematically challenges results forces the LLM into an iteration loop instead of accepting the first plausible-looking answer.

Five gates, five forms of distrust

It is no coincidence that our five most important practices are, in essence, five quality gates placed on the model's work. An independent specification-based oracle, because we do not trust the model to honestly verify the code it wrote. A separate adversarial pass after achieving coverage, because we do not believe coverage equals detection. A response-hygiene & security review, because we do not expect the model to look where no one told it to. Explicit probing of concurrency & state, because the most consequential defects in our study, the race condition around decide & the document limit, were exactly where no team reached.

Finally, the rule that green is never enough, because green is the optimist's default state & we pay testers to be pessimists.

Test count as a supporting metric

Discussions of AI in testing often drift toward test-case counts. "AI generated 200 tests" sounds impressive, but it is a supporting metric.

The more relevant question is how many of those tests can detect actual defects. Our data answer clearly: a submission with 60 cases detected eight defects, while a submission with 196 cases detected only one. An organization that pays a vendor based on scenario count is buying volume.

Verifying both the application and AI-generated tests

AI changes how tests are created, but it does not remove the need to verify their correctness. Only its target changed. In the past, our distrust was directed solely at the system under test. Today, we direct it in two directions at once: toward the system and toward the assistant helping us examine it.

Let us be pessimists, because in testing that is a virtue. Our distrust is exactly the contribution the model cannot make on its own. Paired with an eternal optimist, the most valuable person is someone who professionally assumes the worst. Instead of denying that shadow, we can simply put it to work.



9. CONCLUSIONS

WHAT WE DEMONSTRATED

The first edition showed that AI materially & significantly increases productivity. The second confirmed it: models accelerate the creation of frameworks, contract tests & reports while engineering craftsmanship remains strong.

WHAT WE DISCOVERED

Higher productivity leads to more effective testing only when AI-generated tests are supported by appropriate test design and independent verification. It cannot independently determine whether system behavior complies with business requirements. On its own, it cannot distinguish a red test that is an artifact from a red test that has just found a bug. And it can very convincingly weaken an assertion if the conversation goal becomes "make the test pass." Detection rates fell exactly where the production cost of a defect rose, & no one found the two most expensive defects.

What organizations should do

- Adopt an operator standard. Five gates as the framework, fifteen rules as the playbook.
- Measure effectiveness, not volume. Hold teams and vendors accountable for detected, confirmed risks rather than the number of scenarios.
- Design the process around quality gates. Require an oracle, end-to-end traceability from risk to scenario to assertion and a teeth check in the definition of done.

AI lets us write faster. A good operator makes sure we write the right tests faster. We know what this process looks like, how to measure it and how to implement it.

How many of your "green" tests **would catch a bug headed for production?**

We will analyze your application using the same methodology we applied in the AI Testing Lab study. The results will show which risk areas are effectively covered by tests & where gaps still require attention.



Remigiusz Bednarczyk
Test Manager in Sii Poland.



Tomasz Kuran
Solution Architect in Sii Poland.



Krzysztof Bednarski
Solution Architect in Sii Poland.

Have questions?

Contact our experts

Iuliia Toporova
Business Development Manager



10. Appendices

Appendix A. Detection distribution of seeded defects

Difficulty layer	Example seeded defect	Teams that detected it
Contract / calculations	Incorrect HTTP status (200 instead of 201) when creating an application	13 / 16
Contract / calculations	Incorrect fee & mismatched annuity payment	13 / 16
Contract / documentation	Missing "Debt Consolidation Loan" described in requirements	13 / 16
Security	password_hash field exposed in profile response	11 / 16
Contract	Phantom DELETE endpoint declared in OpenAPI	11 / 16
Domain	DTI calculated from gross instead of net income	6 / 16
Cross-functional	Old session remains active after password change	2 / 16
Cross-functional	Product change affects existing applications	2 / 16
Cross-functional	Race condition in concurrent analyst decision	0 / 16
Cross-functional	Approval despite negative credit score	0 / 16

Appendix B. Number of tests vs. number of detected defects

Source data for Figure 3. Each row represents one submission.

Written test cases	Seeded defects detected	Cases per defect
60	8	7.5
53	6	8.8
64	6	10.7
131	8	16.4
107	6	17.8
154	8	19.2
157	8	19.6
213	9	23.7
150	6	25.0
226	7	32.3
263	8	32.9
134	4	33.5
109	3	36.3
201	4	50.2
912	10	91.2
196	1	196.0

Appendix C. Reported bugs, detected defects & submission cost

Source data for Figure 4. Cost is shown on a relative scale where 100% represents the most expensive submission in the group. The final column shows cost per detected seeded defect.

Reported bugs	Seeded defects detected	Cost	Cost per defect
21	6	3.5%	0.58
23	6	8.5%	1.42
16	9	15.0%	1.67
13	8	14.1%	1.76
23	8	19.1%	2.39
25	8	21.3%	2.66
17	7	22.3%	3.19
20	6	20.4%	3.40
22	6	31.2%	5.20
62	10	65.0%	6.50
13	4	27.1%	6.78
25	8	56.6%	7.08
13	3	23.3%	7.77
17	4	33.9%	8.47
33	8	100.0%	12.50
7	1	16.0%	16.00

Appendix D. Evaluation scope & participant technology stacks

- 16 submissions, each with its own test framework.
- 19 intentionally seeded defects across 4 difficulty layers.
- About 350 bug-report files reviewed manually.
- Each report checked against the LoanFlow code, participant documentation, the OpenAPI specification & the internal answer key.
- Differences between the preliminary ranking & verification resolved through code inspection.

Submissions used different technologies, which supports treating the practices described here as technology-stack agnostic. Playwright with TypeScript was the most common choice, with solutions also using Java, C# & pytest. Repositories repeatedly included page objects, fixtures, helpers, API wrappers, HTML reports, AI context files & scripts that ran the suite with a single command.