

Testing an agent against the rules it was given

Compiling an operating policy into an executable test environment

Constolve • October 2026

Every figure in this document comes from a recorded run and is quoted with its denominator. Where a measurement is scoped to one domain, one model family or one point in time, that scope is stated in the same sentence as the number.

1. The problem

A job that does not finish

Picture a coding agent running on a build machine. Its job that morning is to read every log file under a directory and summarise the errors it finds. It has done this every morning for weeks.

In production the directory contains a named pipe: a special file, left behind by another process, that looks like an ordinary file until you try to read it. The agent opens it the way it opens any file, and waits. Nothing is ever written to the pipe. There is no exception, no error message, no failed assertion. There is a job that does not finish, a queue backing up behind it, and an engineer looking at a process that appears to be working normally.

This is measured behaviour, not a thought experiment. In a comparison where thirteen agent systems each ran the same nine tasks twice, that single task accounted for **eleven of the thirty-one failures**, every one of them a timeout, affecting **six of the thirteen systems**. Two of the eleven are inferred from the task and the outcome rather than from a recorded timeout, because the systems they came from report no timeout field.

Nothing in that failure looks like a bug in the usual sense. The agent's reasoning was sound. It picked a reasonable tool. It even had a way to find out: a one-line check would have told it the file was a pipe. It simply had no reason to think it needed one.

Why testing an agent is not testing a program

A program has inputs and outputs, and you test it by supplying the first and checking the second. An agent breaks all four of those assumptions.

1. **The unit is a conversation, not a call.** A customer says their phone has no service. Twenty turns later something has been fixed or it has not. There is no single input to vary and no single output to assert on.
2. **The input is incomplete on purpose.** Real customers do not volunteer the fact that changes the answer. The person above did not mention they were travelling

abroad, which is exactly the fact the fix depends on. A test where the customer says everything up front is not a test of anything difficult.

3. **The output that matters is a change in the world, not a message.** An agent that says “I have reactivated your line” and did not reactivate the line has produced a fluent, confident, wrong answer. Grading the message re-rewards fluency. Grading the world does not.
4. **The failures happen at the tool boundary.** Not in the model’s reasoning, but at the seam where reasoning meets a real system: a file that is not a file, an argument that is subtly wrong, an action taken on the agent’s own side when it should have been requested of the customer.

And underneath all of it, one more thing: **there is a rulebook.** Every deployed agent has an operating policy, the document its owner wrote saying what it must do, what it must never do, and in what order. That document is the only thing that says what “correct” means for that particular agent, and it is the thing a general benchmark cannot know.

What a benchmark score cannot tell you

A **benchmark** here means a fixed, published set of tasks that anyone can down-load and run, with scores that different systems can be compared on. They are useful and we use one heavily. They also have three limits, and each of them is measurable.

1. **The task set is fixed and the policy is not.** A benchmark’s tasks were written once. An operating policy gets edited the week after a customer complaint. A score against a fixed task set cannot tell you whether last Tuesday’s policy edit helped.
2. **All-or-nothing marking throws away most of what happened.** A run that got four of five required actions right scores the same as one that got none. That is most of the information about what to fix.
3. **Scores move when nothing has changed.** This is the one that surprises people. Running the same unchanged agent, on the same unchanged tasks, three times, with the temperature set to zero, produces scores of 46%, 62% and 54%: a spread of **sixteen points**, larger than most improvements anyone reports.

It gets worse, in an instructive way. On a different domain, three identical runs each reported 68% to the nearest whole point, 232 passes across 342 task-runs. Underneath that steady headline, **thirty-five of the one hundred and fourteen tasks changed verdict between runs.** The headline was steady because the flips cancelled.

Figure 1 shows both halves of that: on the left, how far a score moves when nothing changes; on the right, a stable headline sitting on top of an unstable measurement.

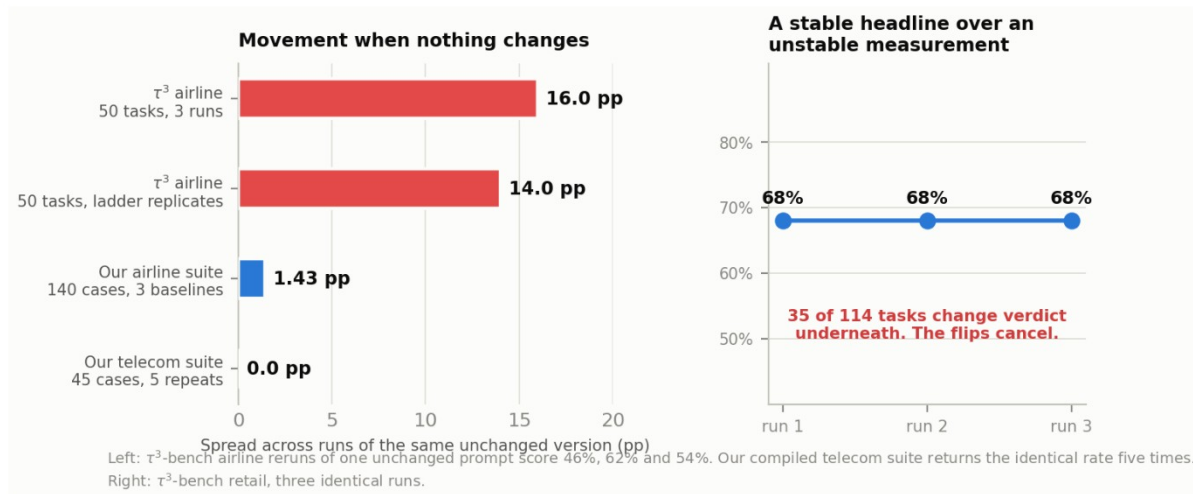


Figure 1: Left: how much a score moves when nothing changes. Right: a stable headline over an unstable measurement.

If you are choosing between two versions of an agent on a five-point difference, and your measuring setup moves sixteen points on its own, you are reading noise.

2. What we set out to do

One objective, stated plainly.

Given only an operating policy and the list of actions an agent can take, produce a set of test scenarios that tells you whether that agent follows that policy, with every scenario carrying machine-checkable evidence that it is solvable, and with enough repeatability that a real difference between two versions is visible above the noise.

Three constraints we imposed on ourselves, because they are what make the result trustworthy rather than merely impressive:

- **No hand-written test cases and no database.** If a customer has to supply fixtures, the method has not solved the problem they actually have.
- **Nothing accepted because a model said so.** Every artefact is accepted by being executed, or it is rejected.
- **Every claim reported with the number of runs behind it,** and anything that cannot be reproduced from a stored record is withdrawn rather than quietly kept.

3. The proposed solution

We take the two things a company already has, the **operating policy** and the **tool schemas** (the machine-readable list of actions the agent can take), and from those two things alone we build a working replica of the system the agent operates in, and then a suite of test scenarios inside it.

The idea in one sentence: **use a language model once, to write code, then throw the model out of the loop and let the code do the rest.**

That single decision is what makes the output checkable. A model asked to invent a test scenario has to invent a starting situation, a fault, a correct solution and a marking scheme all at once, and none of it is ever executed, so none of it can be checked. A model asked to write a working replica of a system produces something that either runs or does not. The uncertainty moves from “is this test any good” to “does this code execute”, and the second question has an answer.

There are three steps, and the important thing about them is where the model is and is not involved.

4. How it works

Step one: compile the system

A model reads the policy and the tool descriptions once, and writes a small working replica of the customer’s system. That replica has three parts, and it is worth naming them concretely because the rest of the paper leans on them.

- **The stored settings.** For a telecom support agent: whether a line is active or suspended, whether roaming is switched on, how much data has been used this month, whether the bill is paid.
- **The actions.** One function per tool the agent can call, carrying that tool’s stated preconditions and effects. `suspend_line` checks the line is active, then sets it to suspended and records the date.
- **The health checks.** A set of named conditions defining what “working normally” means. “The line is active.” “The bill is not overdue.” “Mobile data is reachable.” These are what a fault breaks and a repair restores.

Then the replica is **run**. Every action must parse, match its declared signature, and execute. Anything that does not is rejected and the model is asked again.

The worked example used throughout this paper. Our main test case is the telecom customer-support domain from a public benchmark: an agent that handles calls about phones with no service, data that will not connect, picture messages that will not send, and bills. We chose it because the benchmark ships its own hand-written test environment and its own hand-written suite of nineteen fault types, which means every claim we make about our generated version can be checked against a version somebody else wrote independently. The inputs are a policy document of **158 lines of prose** and **43 tool schemas**.

From those two inputs, compilation produced a **2,241-line program** implement-ing all forty-three actions, fourteen health checks, and five things a customer can actually perceive. Later fixes took it to 2,337 lines. For scale: the hand-written environment shipped with the benchmark, which this replaces, is **2,851 lines** and took a research team to write. The comparison is like for like, both counted directly from the files.

On the forty-three actions. Forty-three is the size of the agent’s whole toolbox, not the number of calls in a test. A typical task in this domain grades about **4.5 required actions**, of which roughly one is the agent’s own call and the rest are things it must get the customer to do. So no scenario uses all forty-three. What the forty-three buys is coverage of the space the agent could go wrong in, and wrong tool use is graded explicitly: a scenario fails an agent that calls the right tool with wrong arguments, that performs an action itself when the policy says to ask the customer to do it, or that calls things in an order the policy forbids. Section 6 reports how reliably the suite catches each of those.

The model is used once, at build time. After that, the replica is ordinary code, and it behaves identically every time it runs. Measured across three domains, **the replica contributed zero variance**: repeated runs of the same case produced no divergence in stored settings, in health checks, or in what any action returned, within a process and across processes. The telecom measurement rests on 30 cases run three times; the other two domains were checked on smaller samples and support the result less strongly.

Step two: break it, and record what fixes it

With a working replica, faults do not have to be invented. They can be **found**. The system changes every stored setting, calls every action, and records which health checks each change breaks. Then it searches for the sequence of calls that puts each one right again.

The result is a vocabulary of fault units, each one a pair: something that goes wrong, and the calls proven to put it right. For the telecom policy this recovered all seventeen of the nineteen fault types in the benchmark’s hand-written suite that an agent can repair, without ever reading that suite: sixteen completely, and one, an overdue bill, only in part, with the line suspension found but not the payment steps. The other two are resolved only by handing off to a human, so there is no repair to recover.

This step involves no model.

Step three: compose, and prove each scenario solvable

Fault units combine, and combining them produces a very large space of candidate scenarios. Most of that space is junk: combinations that cannot be repaired, or that the system could never have arrived at.

Every candidate is therefore **replayed** before it is allowed to exist:

1. Assert the world is healthy.

2. Apply the breakages. Assert it is now broken.
3. For each repair step in turn: assert the problem is not yet fixed, apply the step, assert the expected result.

A candidate **fails** this replay if any one of those assertions turns out false, and each failure means something specific and disqualifying. If the world does not break, there is no fault to find. If a repair step's expected result already held before the step ran, that step was unnecessary and the scenario would be marking the agent on work it does not need to do. If the last assertion fails, the scenario is not solvable by the solution it ships with. Failed candidates are discarded rather than patched, so a scenario reaches the suite only if it has already been solved once, deterministically, by its own solution.

On the telecom policy, the build our coverage result was measured on emitted **92,184** scenarios, each carrying a replay proof, and discarded every composition it could not prove. Figure 2 shows where the attrition happens.

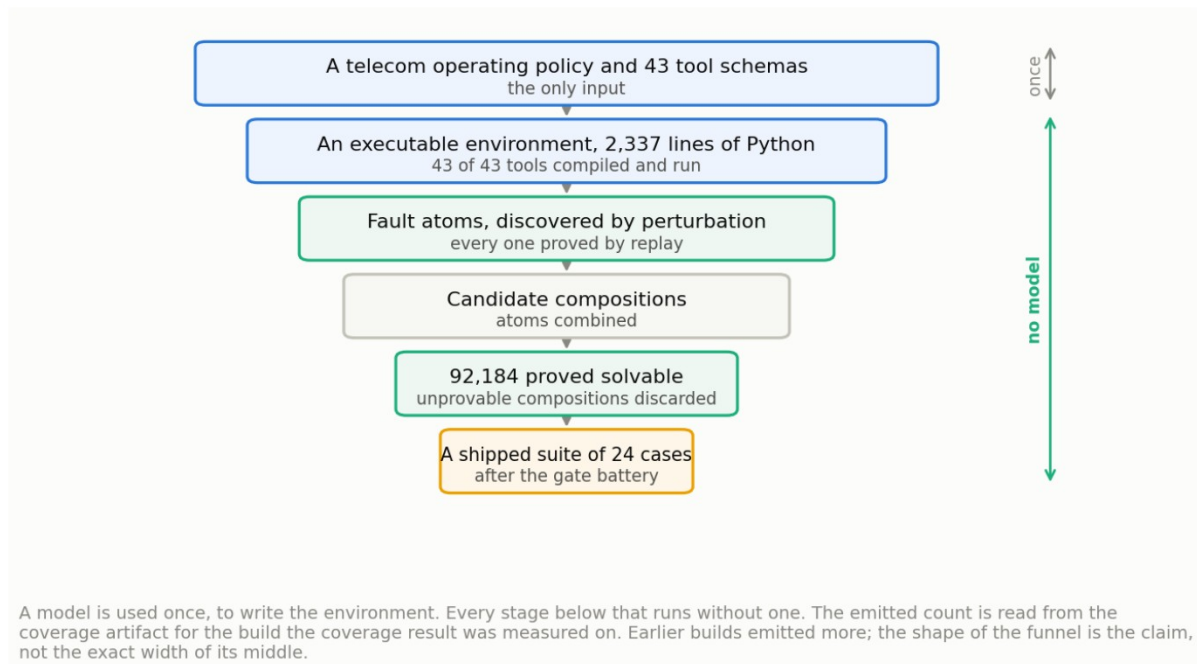


Figure 2: Figure 2. The compilation funnel, with real attrition.

This step also involves no model. One model call happens at the very end, to write the customer's opening description in natural language, and it is structurally prevented from touching the world, the graded actions or the pass criteria. If it could touch them, the replay proof would no longer describe the scenario that actually runs.

Two ideas that make this work

Both came from reading how a well-built public benchmark actually constructs its tasks, rather than from any measurement of our own.

A test world should be reached, not written down. There are two ways to set up a broken situation for a test. You can write the situation down as data, or you can call the actions that break it.

Writing it down seems simpler and is a trap. Suppose you want a test where a customer's line is suspended for non-payment. Written down as data, it is a few lines of JSON, and any of them can be wrong in a way nothing catches: a field named `roaming` when the system calls it `roaming_enabled`, a status of "Unpaid" when the only values the system allows are Active and Suspended, or a combination the system could never actually produce, like a suspended line with no suspension date. Every one of those has to be checked for separately, and we built roughly twenty such checks.

Call `suspend_line` instead and none of those checks are needed. A situation you arrived at by calling real actions cannot be a situation the actions are unable to reach. The checks become unnecessary rather than passed. Figure 3 puts the two side by side.

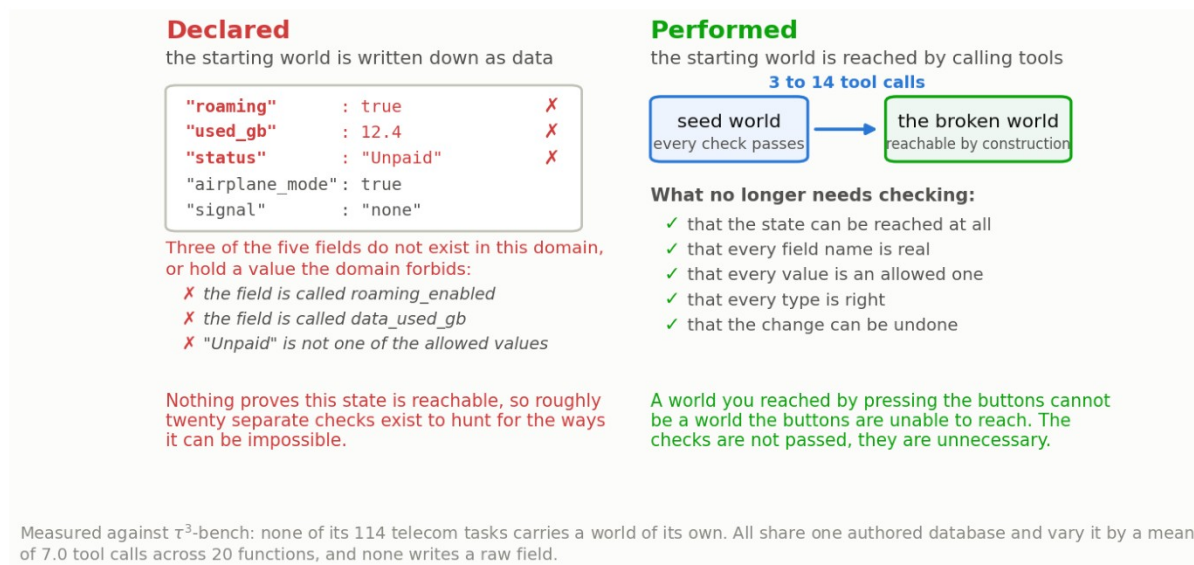


Figure 3: Figure 3. Two ways to set up a broken world.

Difficulty comes from the structure of the system, not from asking for it. A real system has three levels, shown in Figure 4.

- At the bottom, **the settings it stores**: roaming on or off, network type, data usage.
- At the top, **what a person can actually perceive**: whether a picture message sends. These are computed from several stored settings at once.
- In between, **the actions**, which change stored settings and only stored settings.

Whether a picture message can be sent depends on mobile data working, which itself depends on six separate settings, and on the network type, and on a configuration URL, and on app permissions. **No single action sets it.** A customer who cannot send picture

messages therefore has a problem with several causes, and every one has to be found and repaired before the thing they can see changes.

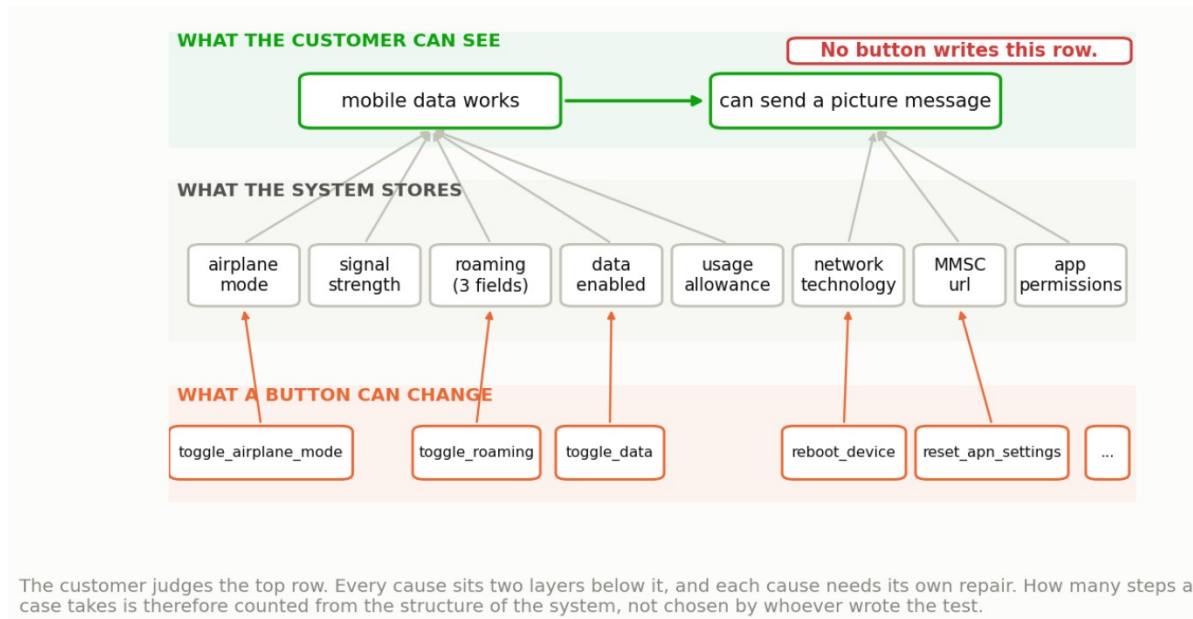


Figure 4: Figure 4. The three layers, with a real derivation.

That is why the number of steps a scenario takes should not be chosen by whoever writes the test. We tried asking a model for harder, deeper scenarios and got padding: extra steps that did not change anything. Grading what the customer can perceive produces depth automatically, because the number of repairs needed is simply the number of causes, and that is a fact about the system rather than a dial anyone sets.

Deciding which scenarios are worth running

A scenario can be perfectly well formed and still be worthless. Two different questions have to be asked, and they are not the same kind of question.

Question one: is the scenario sound? Every one of these is a property of a single replay with no agent involved, and all of them are cheap.

- Does it start from a healthy world?
- Is it solvable by its own solution?
- Is every graded step actually necessary?
- Do the arguments matter, or would any value pass?
- Can the customer perceive the repair?
- Is every statement in the scenario's text true of the world?

Question two: would the scenario tell anything apart? Would it distinguish an agent that read the rulebook from one that did not? This is a property of the *difference between two agents*, and a generator working from one correct solution has no way to see it.

The fix is to write the weaker agent down. We run three synthetic opponents, each doing strictly less than a real agent:

- one that does nothing at all;
- one that immediately hands off to a human;
- one that calls everything, in the wrong order.

Any pass criterion the do-nothing opponent satisfies is meaningless by construction. Any prohibition the reckless opponent fails to trip cannot be enforced. Both are checkable at zero cost, with no agent run.

And one check that grades the tests rather than the agent. Take a scenario's proven solution, corrupt it in a known way, and confirm the suite catches the corruption. Results are in Section 6.

5. How we tested it

This section defines the vocabulary and the setup, so the numbers in Section 6 can be read without guessing.

The domains. Three, all from the same public benchmark, all customer-support agents with a written policy and a set of tools.

- **Telecom.** Phones with no service, data that will not connect, picture messages that fail. 114 tasks, 43 tools. This is the domain that carries the coverage result, and the only one taken all the way through to a validated suite.
- **Airline.** Booking changes, cancellations, refunds and baggage, under rules that differ by fare class. 50 tasks, 14 tools. Used mainly to test how repeatable our setup is.
- **Retail.** Order modifications, returns and exchanges. 114 tasks, 16 tools. Used for the same purpose.

The basic units.

- A **task** or **scenario** is one customer situation, run as one conversation.
- A **run** is one execution of one scenario by one agent.
- A **repeat** is running the whole set again from the start with nothing changed. Repeats are how we measure noise.
- An **arm** is one side of a comparison. Every scenario is run twice: once by an agent holding the full operating policy, once by an identical agent holding a single sentence instead. The gap between the two arms is the headline number, and the

second arm is what proves the suite measures anything at all. If an agent that never read the rules scores the same as one that did, the suite is not testing the rules.

The scoring rules. Three, and they are not interchangeable, so we never pool them.

- **Actions completed.** Of the actions a scenario requires, how many happened. This is partial credit, and it is the one that tells you what to fix.
- **Cases solved.** All-or-nothing: every required action happened and no prohibition was broken.
- **The benchmark's own score.** Whatever the public benchmark publishes, used unchanged when we are checking ourselves against it.

The check we ran before anything else. Before pointing a new measuring setup at anything new, run it on material where the answer is already published and confirm it reproduces that answer. Ours reproduces the benchmark's own result on the benchmark's own tasks, which is what rules out the explanation that our environment, our simulated customer or our grader are the reason our numbers differ. Section 6 gives the figures.

6. Results

The setup reproduces the benchmark before it is pointed anywhere new

This is the result we would most want a sceptical reader to check first. Take the public benchmark's own tasks, run them inside our compiled environment with our simulated customer and our grading, and compare against the benchmark's published numbers.

On the strict measure our stack reproduces the benchmark closely and slightly exceeds it: a gap of **+20.6 points** between the two arms, against the benchmark's own **+17.3**, recomputed from the surviving run logs. Our environment, simulated customer and grader are therefore not what makes our own results different. Only after that did we point the same setup at scenarios we generated.

The suite separates an agent that has the policy from one that does not

Figure 5 shows all three ways of measuring it, and all three separate.

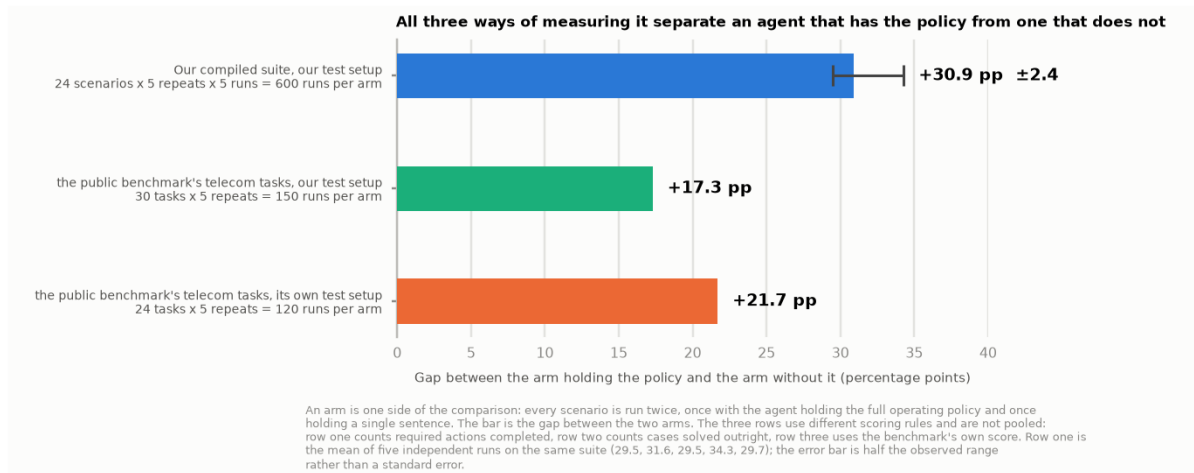


Figure 5: Figure 5. The gap between an agent holding the policy and an agent without it, under three different scoring rules.

The top row is our own compiled suite, scored on actions completed: a gap of +30.9 points. That figure is the mean of five independent runs on the same suite, each of 24 scenarios at five repeats, 120 runs per arm. The five runs gave +29.5, +31.6, +29.5, +34.3 and +29.7, and the bar carries half their observed range, ± 2.4 , rather than a standard error.

The lower two rows are the benchmark's own telecom tasks, once through our setup and once through the benchmark's own, at 150 and 120 runs per arm. They use different scoring rules from the top row and from each other, which is why the three bars are stacked rather than compared: the point is that the separation survives every way of measuring it, not that the three numbers are the same quantity.

The suite is more repeatable than the benchmark it was checked against

The benchmark's airline domain moves ten to sixteen points between identical runs. Our airline suite moves **1.43 points** across three baselines, recomputed from the run database rather than from our own summaries, and one arm of our telecom suite returned the identical rate five times in a row: a spread of **zero**.

Repeatability is not the same as sensitivity, and the difference matters. A more repeatable setup makes small real differences visible. It does not by itself prove that the differences it shows are the ones you care about.

The suite catches deliberate corruption of its own answers

Take each scenario's proven solution, corrupt it in a known way, and check the suite notices. On the telecom suite of 24 scenarios used for the separation runs: 185 of 202 non-equivalent corruptions caught, 91.6%. By kind of corruption, this is where the tool-use questions get answered:

Corruption introduced	Caught
-----------------------	--------

A required call dropped entirely	72 of 76
A call made with wrong arguments	33 of 41
An action performed by the wrong party	48 of 52
Handing off to a human too early	19 of 19
Two calls swapped in order	13 of 14

The 17 survivors fall into three groups. Eight change an argument: the size of a data top-up, or the customer and line passed to a usage lookup. Eight concern two checks the customer performs on the phone, the status bar and the SIM status, which in one handoff scenario the suite neither requires nor attributes to the right party. One swaps a line resumption with a data top-up. An earlier build of the suite caught 248 of 251, but it is not the build the separation runs used. We report them because a suite that caught everything would be the more suspicious result.

Compilation reaches ground a model writing tests directly did not

We ran thirty-one numbered rounds of refining the prompt and the filters that check its output, with the benchmark's own telecom tasks as the target. Coverage moved from zero to three of one hundred and fourteen. That figure rests on a run note rather than a stored result file, so treat the direction as the finding and the exact count as approximate.

Compiled from the same policy, the generator emits, for 105 of 113 of those tasks, a scenario requiring the same set of repair calls by tool name; on the 94 tasks the benchmark grades on a repair, it is 90 of 94. Figure 6 puts the two side by side.

Two things about that comparison need stating. The denominator is 113, not 114, because one benchmark task is made only of faults resolved by escalating to a human, so it has no repair to compare. Fifteen of the 105 are tasks the benchmark grades on escalation alone, which is why 90 of 94 is the fairer figure. And the two bars are deliberately different tests: the upper one asked a model to judge whether a generated scenario covered a benchmark task, while the lower one asks whether the set of repair calls is identical, which needs no judge and can be checked by anyone. It does not compare arguments or call order, and the generator was revised while coverage was measured against these same tasks, so it is not a held-out result.

Most of what a naive setup reports is not real

Three findings from measuring our own setup, all of which we would rather not have found.

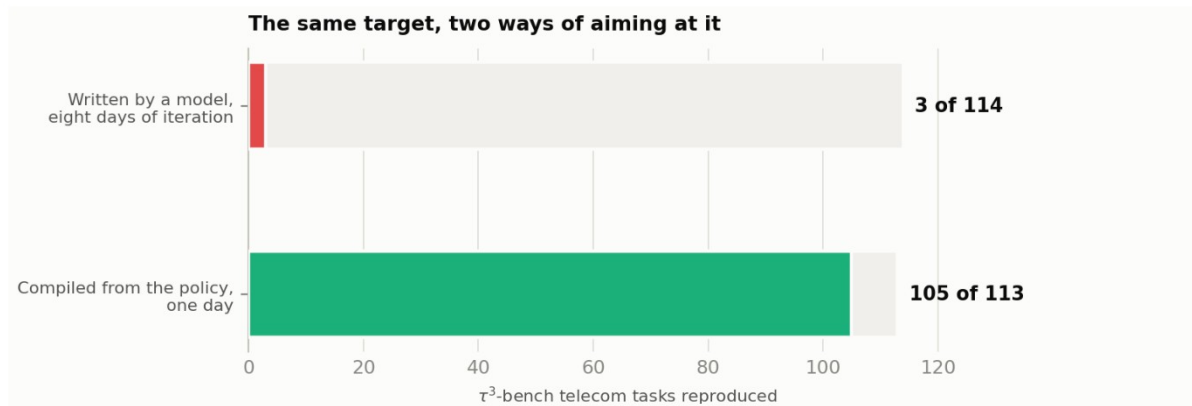


Figure 6: Figure 6. The same target, two ways of aiming at it.

A failure list is not a failure list until it has been replayed. Replaying every reported failure three times, without changing anything: on the **airline** domain, **7 of 19** reported failures happen again every time; on **retail**, **12 of 33**. The rest either recur intermittently or never recur.

The two populations behave differently enough that we do not average them, and the reason is worth stating. A failure that reproduces every time is usually structural: the agent is missing a rule, or takes a step in an order the policy forbids, and it does so on every run because nothing about it depends on chance. A failure that does not reproduce is usually one of three things: the conversation ran out of turns at a slightly different point, the simulated customer phrased something differently and the agent took a different branch, or the grader itself was inconsistent. The third of those is our defect, not the agent's, and the next finding is the clearest case of it. We are not yet able to attribute each non-reproducing failure to one of the three causes, and until we can, the honest statement is that fewer than half of a raw failure list survives replay.

A judge that cannot see the rule still returns a verdict. When a language model is asked to judge whether an agent broke a rule, and is not shown the rule, it answers anyway. Its failure verdicts survive an identical rerun **3 times in 19**, against **16 in 33** when it can see the rule (Figure 7). It was scoring the absence of a rule as a violation of it. How widespread the condition was cannot be recovered: the rule text handed to the judge was never stored, so the share of affected criteria is not recomputable and we do not quote one.

An agent that can do nothing cannot break a rule. On one suite that scored 95%, twenty of sixty conversations ended with the agent structurally unable to act, of which **seventeen were scored as passes**. Excluding them raised the pass rate on the remaining forty conversations to 100%, which is the opposite of the direction anyone expected. Put the other way: of the fifty-seven passes behind that 95%, **forty-four came from**

conversations in which no tool ever returned usable data. Thirteen conversations had usable data at all, and all thirteen passed.

We report these because they are the reason to take the rest seriously. A measuring setup that has not been turned on itself is not a measuring setup.

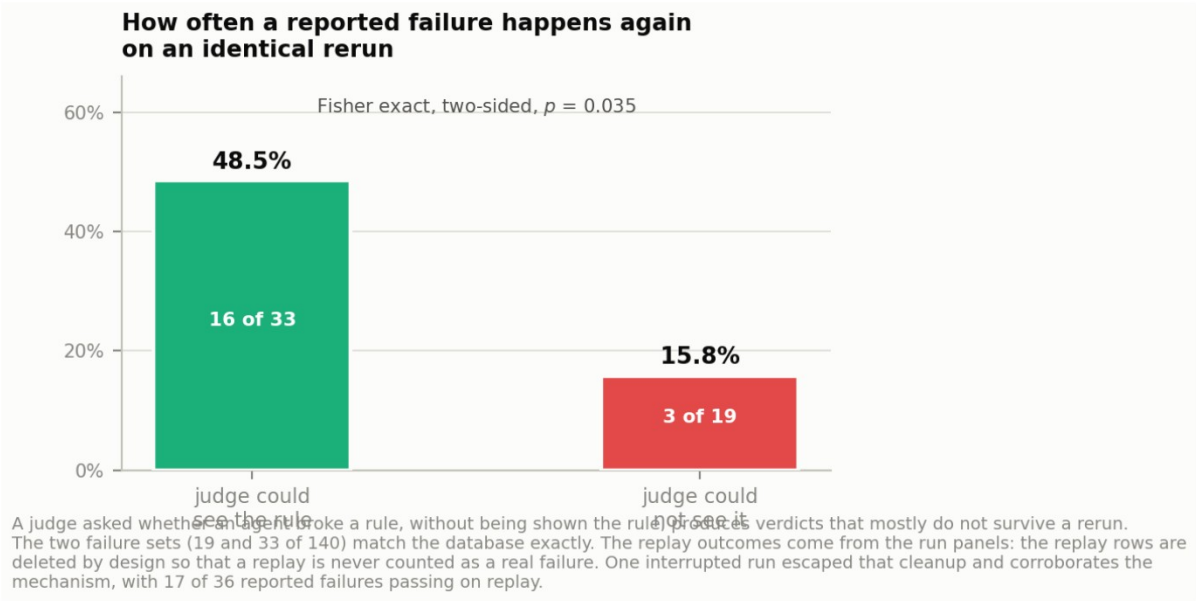


Figure 7: Figure 7. What happens when the judge cannot see the rule it is enforcing.

What the evidence supports	The nearest claim we decline to make
35 subjects have repeated execution evidence, out of a catalogue of 96 entities.	That all 96 are executable, or that the 35 form one ranking.
13 systems ran a shared 9-task fixture twice: 234 runs, 203 passes, 31 failures.	That these can be pooled with results measured on each system's own tasks.
One deployment cleared a bounded safety optimisation, 16 of 18 to 18 of 18.	That the optimisation generalises. The one transfer we tried failed its gate.
A judge configuration reached zero false positives across five repeats.	That it is certified. It was selected on a development set, not a held-out one.
Generation evidence across nine models, reported descriptively.	That any model is the best test generator. No adjudicated measure exists.

Each row is a boundary we wrote down before publishing, not after being asked.

Figure 8: Figure 8. Where the evidence stops.

7. What we have not shown

Beyond that table, five limits stated plainly.

One domain, one model family. The compiled environment works on three domains. The full chain through to a validated suite and a measured separation has been carried out on one.

A larger suite is not a better suite. Our generator reaches tens of thousands of repair sequences the benchmark does not contain. We have not characterised them, and “more” is not “better” until somebody does.

A separation is not a certification. In the one place where we ran a formal significance test against a benchmark on the same contrast, our suite ranked the agent versions correctly and could not certify the difference at the sample size we used, while the benchmark could. That result is what pushed us to rebuild the generator, and it is why we describe what we have as a valid way to *order* two agents rather than a valid way to *decide* about one.

The customer is simulated. Simulated users are not human users. They diverge on impatience, on error recovery, and on how goals actually get accomplished. Every measurement here is pre-deployment and comparative: is version B better than version A, under a fixed and declared model of the customer. None of it predicts live human success rates, and we will not present it as though it does.

Two of three domains carry a caveat we have not yet cleared. Because the benchmark is public, a model writing an environment can recall identifiers from it. We audited for this. The telecom domain, which carries the coverage result, is effectively clean: its single hit is a product name rather than a record. Two other domains recalled real record identifiers, and their counts are not quoted until their environments are regenerated under a rule that makes identifiers synthetic by construction.

8. What a pilot looks like

You supply an operating policy and the tool schemas for one agent. Nothing else, and in particular no database, no fixtures, and no hand-written test cases.

We compile an environment from those two inputs and show you the environment itself, because it is ordinary readable code and you should check whether it re-ssembles your system. We then generate a suite, each scenario carrying its own proof that it is solvable, and run your agent against it twice: once holding the policy, once holding a single sentence.

What comes back is three things: the gap between the two arms with its spread, a per-scenario record of which required actions happened and which did not, and an explicit list of what the evidence does not support.

If the gap is small, that is the finding, and we will say so. A suite that cannot tell your agent apart from one that never read your rules is not a suite you should be buying, and the fastest way for both of us to learn that is to run the second arm.

Notes on the numbers in this document

Every figure quoted here has been checked twice: against the stored result file it came from rather than against our own summaries of it, and then against the run database underneath that. Where a figure exists only in a run note and neither the result file nor the database can produce it, we have either replaced it with one that is stored or left it out. One that survives on a run note alone is marked in the text: the coverage figure for the model-written suite.

That check changed real numbers. A frequently repeated coverage claim of nineteen fault types out of nineteen is seventeen of the seventeen an agent can repair; the other two matched only through a bookkeeping flag. A perfect mutation score of 260 out of 260 was measured on an earlier build of the suite; on the build used for the separation runs it is 185 of 202. Going to the database found that a run we had recorded as covering 140 cases had lost four rows from one of its tables, so a spread we published as 1.8 points is 1.43. One correction went the other way: a separation of +30.9 points that an earlier draft could not reproduce does reproduce from all five stored runs, using the project's own scoring tool. The rest moved in the direction that makes our case weaker, which is why the checking is worth doing. The full list of what we withdrew, and why, is available to a design partner on request.