

What an agent actually costs

Seven versions of one production agent, measured on the same benchmark: including the two that got worse, and the chart that would have been wrong.

Nikolaos Broikos • broikos.gr August 2026 7 versions • 270 task-executions • 30 tasks

270

GRADED TASK-EXECUTIONS ACROSS SEVEN VERSIONS

22.8%

WHAT A DO-NOTHING AGENT SCORES ON THE SAME GRADERS

4

RECORDED ERRORS, EVERY ONE A STEP LIMIT

2 of 6

VERSION TRANSITIONS THAT MADE THE AGENT WORSE

01 What this is

Discharges claims D001, D002, D003.

This is the engineering record of one production agent, published whole.

Seven versions of a Windows automation agent were benchmarked over thirty-one hours on 1 to 2 August 2026, against the same thirty tasks, on the same model, with the same grader. **Two hundred and ten task records covering 270 task-executions**, because the last version runs everything three times. Every run recorded, per task: points earned against points possible, which tools were called and how many times, wall-clock duration, input and output tokens, and any error string.

The record was made **while building the agent, not for publication**. That is the only reason it is worth reading. Nobody sets out to publish the version of their software that scored six points worse than the one before; a record kept for engineering contains it because the engineer needed to know.

Two of the seven versions were worse than the version preceding them. Both are in the table.

What this paper is not

It is not a claim that this agent is good. The suite it is measured on is close to saturated, twenty-eight of thirty tasks reached a perfect score in at least one version, and a suite everything passes cannot rank much of anything. That limitation is stated here, in section 09, and in the data.

It is not a comparison with other agents. A companion paper in this programme measures three agent harnesses against each other on one model; this one measures a single agent against its own history. Those are different questions and the instruments do not transfer.

It is not a cost estimate for anyone else's workload. Thirty tasks on one machine with one model is a measurement of this agent, at that price list, in that week.

What it is

A worked example of what it costs to iterate an agent, with the iterations that failed left in.

The headline numbers, corrected as section 03 explains they must be:

versions benchmarked	7 (v3 to v10; v4 was never benchmarked)
tasks per version	30, worth 302 points
task records / executions	210 / 270
model	deepseek-v4-pro throughout
score range	87.1 → 97.6
versions that got worse	2 of 6 transitions
cost per task-execution	\$0.00920 to \$0.01309
best value	v7 , at \$0.00976 per 100 points
highest score	v10, at 25% worse value than v7
recorded errors across all 270 executions	4, every one a step limit

The finding that shapes the paper

Halfway through the record, the measurement protocol changed: **v10 runs every task three times, and v3 through v9 run each once.** Nothing in the result files announces this, and the run-level totals do not account for it.

Read without noticing, v10 costs **3.3× more** than v9 and takes **2.8× longer**. Both are artefacts. Per task-execution v10 costs 10% more and is slightly *faster*.

A chart of run cost against version number: the first chart anyone would draw from this data, and the one that would have illustrated this paper, shows a cost explosion that did not happen. That near-miss is section 03, and it is the reason the rest of the paper reports everything per execution.

Why publish it

Paper 03 of this programme carried a line that this paper is the test of: **a case study with no failures in it is marketing.**

The industry publishes capability claims and demonstrations. Regressions stay private, because a regression is embarrassing and a demo is not. The result is that anyone about to build an agent has no public reference for what the process actually looks like: how much a version costs, how often a change makes things worse, and what the failure actually looks like when it comes.

It looks like this. Seven versions, two regressions, and one of them caught and fixed inside twenty minutes.

How to read the numbers

Every figure in this paper is recomputed from the seven published result files by the scripts published alongside them. The tasks, the grader's check types and their point values, all 210 task-executions and the extraction code are in the data directory. **If a number here is wrong, the file that disproves it is published next to it.**

Costs are stated at the model's **uncached** rates, \$0.435 per million input tokens and \$0.87 per million output, consistent across two independently maintained price tables dated 1 and 6 August 2026. The benchmark applies no cache discount, so **every cost in this paper is an upper bound** and the real invoice was lower. Section 03 explains why that matters less than it sounds.

02 The instrument

Discharges claims D004, D005, D006, D007.

Numbers from a benchmark mean nothing until the benchmark is described. This one is published in full, tasks, prompts, check types and point values, so this section is a summary of an artefact a reader can open rather than a substitute for it.

Thirty tasks, 302 points, nine categories

Each task is a natural-language instruction given to the agent on a real Windows machine, with a maximum tool-call budget and a flag saying whether the task is permitted to change the system.

CATEGORY	TASKS	WHAT IT ASKS FOR
orientation	4	identify the machine, diagnose a symptom
telemetry	4	find and change privacy-related settings
debloat	4	remove preinstalled software
safety	4	refuse or qualify a dangerous instruction
packages	4	install, update and identify software
services	3	inspect and change Windows services
registry	3	read and modify registry state
defaults	2	set default applications
recovery	2	restore points and rollback

The mix is deliberate. **Four of the thirty tasks are safety tasks in which the correct behaviour is to not do what was asked**, or to qualify it first. An agent that scores well by being obedient will lose those points.

Twelve check types, and what they refuse to accept

A task's points are split across checks. The distribution says more about the instrument than any prose could:

CHECK TYPE	CHECKS	POINTS	WHAT IT DECIDES
tool_used	18	72	a specific tool was actually called
tool_arg_contains	19	60	the call carried the right argument
completed	26	36	the task finished at all
answer_contains	10	34	the reply names the right thing
any_tool_used	7	25	at least one of a permitted set was used
no_mutation	6	22	nothing on the machine changed
no_repeated_calls	9	20	the agent did not loop
max_tool_calls	8	16	it stayed inside the budget
any_call_contains	3	6	some call carried a required string
no_self_confirm	1	5	it did not approve its own dangerous action
tool_arg_not_contains	1	4	a forbidden argument was absent
tool_not_used	1	2	a specific tool was avoided

Only 34 of 302 points, 11%, come from what the agent says. The rest come from what it did: which tools it called, with which arguments, how many times, and whether the machine changed. That is the same discipline this programme's harness paper argues for, arrived at independently and a week earlier.

Note the shape of the negative checks. `no_mutation`, `no_repeated_calls`, `max_tool_calls`, `no_self_confirm`, `tool_not_used` and `tool_arg_not_contains` together carry **69 points, 23% of the total, for not doing things**. An agent cannot reach a high score here by working harder.

The model, held constant

Every one of the 270 task-executions ran on deepseek-v4-pro. No version changed model, temperature tier or provider. Whatever moved between versions, it was not that.

This matters more than it might seem. The obvious confound in any longitudinal agent record is that the underlying model changed underneath the engineering, and the record then measures the vendor rather than the author. Here it did not.

Prices, and why every cost here is an upper bound

Costs are computed from recorded token counts at **\$0.435 per million input tokens and \$0.87 per million output tokens**. That table appears in the benchmark runner, committed 1 August 2026, and is independently corroborated by a separately maintained table in another project in this programme, fetched 6 August 2026. The two agree exactly.

The benchmark applies no cache discount. DeepSeek prices cached input at a small fraction of the uncached rate: the corroborating table puts it at \$0.003625 per million, about 120× cheaper, and these runs push between 577,000 and 2,240,000 input tokens each, much of which is a system prompt repeated across tasks.

So the honest statement is: **every cost figure in this paper is an upper bound, and the true spend was lower, probably substantially.** The result files do not record cached-token counts, so the discount cannot be reconstructed after the fact.

This is stated rather than corrected because the comparison the paper makes is *between versions measured the same way*. An uncached upper bound applied consistently across seven versions still ranks them correctly. A companion paper in this programme records what happens when cache metering is inconsistent between two things being compared: the answer came out **nine times too expensive** for one of them.

What the instrument cannot do

It is close to saturated. Twenty-eight of the thirty tasks reached 100% in at least one version, and only two, both in package management, never did. A suite that almost everything passes has little room left to discriminate, which is why the interesting movements in this record are regressions rather than gains.

It has no negative control. Nothing in the published record demonstrates that the graders *fail* on an agent that does nothing. Without that, a high score is consistent with a lenient grader, and the reader is asked to trust rather than check. The harness paper in this programme treats such a control as mandatory and refuses to report without it. **This suite should acquire one**, and section 09 says so plainly rather than leaving it to be discovered.

It runs on one machine. Every measurement is Windows 11 on the author's hardware. Nothing here establishes that the same agent behaves the same way on a different install, and a fresh machine would exercise exactly the paths, package installation, defaults, recovery, where this record is weakest.

03 The chart that would have been wrong

Discharges claims D008, D009, D010.

This section exists because the paper it was going to be would have opened with a false headline.

What the raw record says

Seven result files, each with a run-level total. Read them straight and plot cost against version:

VERSION	RUN COST	RUN MINUTES
v3	\$0.2845	13.5
v5	\$0.3550	14.7
v6	\$0.3928	13.4
v7	\$0.2761	9.9
v8	\$0.3566	13.0
v9	\$0.3255	10.5
v10	\$1.0714	29.0

The story writes itself. Six versions hold between \$0.28 and \$0.39, and then the final, best-scoring version costs **3.3× the one before it** and takes **2.8× as long**. Getting from 95.6 to 97.6 apparently cost triple. That is a genuinely interesting finding about diminishing returns, it fits every prior about scaling, and it is **not true**.

What actually changed

v10 runs **every task three times**. v3 through v9 run each once.

Nothing in the run-level summary says so. The repetition appears only inside each task record, as a list of three per-run scores and a spread value, in a field that is null in all six earlier files. The overall score of 97.6 is a mean of three runs; the cost of \$1.0714 is the sum of ninety task-executions, not thirty.

	V9	V10	RATIO
task-executions	30	90	3×
cost	\$0.3255	\$1.0714	3.3×
cost per execution	\$0.01085	\$0.01190	1.10×
wall time	10.5 min	29.0 min	2.8×
seconds per execution	20.9	19.3	0.92×

Per unit of work, v10 costs **10% more** than v9 and runs **8% faster**. The cost explosion is an artefact of counting three runs as one.

The evidence that this is the right reading

Two independent confirmations, which matters because the entire cost argument of this paper rests on the normalisation.

The data. Every v10 task record carries a three-element runs list; no earlier file does. Ninety executions, thirty tasks.

The author's own commit log. The repository behind the agent contains, timestamped to the same minute the v10 benchmark started:

6ec21f8, Add --repeat so the benchmark gives a number worth trusting

and thirty minutes later, when it finished:

edbda1f, *Stable baseline: 97.6 over 90 runs*

Ninety runs, in the engineer's own words, at the time. The normalisation is not an inference from the JSON; it is what the person who ran it recorded.

Why this is the most useful thing in the paper

The mistake was available, cheap and invisible. Every input needed to make it sits in the run-level summary, which is the part designed to be read. Every input needed to avoid it sits one level down, in a field that is null six times out of seven and therefore easy to skip.

And the false version is *better copy*. "Our best version cost triple" is a finding. "Our best version cost 10% more" is a Tuesday.

Three consequences worth carrying:

1. **A protocol change is a discontinuity in the data, and it must be recorded as one.** The benchmark gained -- repeat because repetition makes a number more trustworthy. The change made every earlier number less comparable, and nothing in the output flagged it. 2. **Normalise before you compare, and say what the unit is.** Every figure in this paper is per task-execution and labelled as such. Run-level totals appear only in this section, where they are the subject rather than the evidence. 3. **The most dangerous error is the one that produces a more interesting result.** Nothing about the false reading looked wrong. It agreed with expectations, it made a better chart, and the only thing standing between it and publication was a null field.

That last point is not unique to this paper. A companion study in this programme records two benchmark faults that each produced a confident, reproducible, wrong result, one of which had already been published. **Three separate measurement errors across two papers in one week, every one of them flattering, every one of them caught by a check that only existed because an earlier error had been caught.**

04 The corrected record

Discharges claims D011, D012, D013, D014.

Everything below is per task-execution. Run-level totals appear only in section 03.

Seven versions

VERSION	RUN STARTED	REPS	SCORE	\$/EXECUTION	S/EXECUTION	TOOL CALLS/EXEC	Δ SCORE
v3	01 Aug 19:22	1	87.1	\$0.00948	27.0	3.37	,
v5	01 Aug 19:53	1	96.4	\$0.01183	29.4	4.67	+9.3
v6	01 Aug 20:14	1	90.1	\$0.01309	26.7	5.30	-6.3
v7	01 Aug 20:16	1	94.3	\$0.00920	19.7	3.30	+4.2
v8	01 Aug 20:30	1	95.9	\$0.01189	25.9	4.03	+1.6
v9	02 Aug 01:48	1	95.6	\$0.01085	20.9	4.00	-0.3
v10	02 Aug 02:06	3	97.6	\$0.01190	19.3	1.45	+2.0

v4 was never benchmarked. No result file names it and no commit does. It was an internal iteration that never reached a measurement, and the gap is left visible rather than closed by renumbering.

Progress is not a line

87.1 → 96.4 → **90.1** → 94.3 → 95.9 → **95.6** → 97.6

Two of six transitions made the agent worse. One of them, v5 to v6, cost 6.3 points *and* 11% more per execution, worse on both axes at once, which is the combination that should make an engineer stop.

The total gain across the whole record is **+10.5 points**, from 87.1 to 97.6. The largest single gain, +9.3 at v5, arrived in the first thirty-one minutes and accounts for **89% of everything the remaining five versions and six hours produced**. Everything after v5 is a fight over the last four points, and it is in that fight that both regressions happened.

Cost per point earned

Score alone ranks v10 first. Cost per unit of result does not.

VERSION	SCORE	\$ PER 100 POINTS	VS BEST
v7	94.3	\$0.00976	,
v3	87.1	\$0.01089	+12%
v9	95.6	\$0.01135	+16%
v10	97.6	\$0.01220	+25%
v5	96.4	\$0.01228	+26%
v8	95.9	\$0.01239	+27%
v6	90.1	\$0.01453	+49%

v7 is the most efficient version ever built, and nothing in the record announces it. It is not the top scorer, it has no celebratory commit, and it survived for fourteen minutes before being replaced. It scores 3.3 points below v10 and delivers each point **25% cheaper**.

The worst value is v6, the regression: it earned the fewest points per dollar of any version, including the first.

Cost and score are only loosely coupled

Three pairs from the table make the point:

- **v8 → v9**: score fell 0.3, cost fell 9%. Slightly worse, meaningfully cheaper.
- **v3 → v5**: score rose 9.3, cost rose 25%. The one clearly good trade in the record.
- **v9 → v10**: score rose 2.0, cost rose 10%. Defensible, and much less impressive than the raw files suggest.

Across all seven versions the correlation between score and cost per execution is weak and positive, but it is carried almost entirely by v3, the cheap early version that was also the worst. **Among the six versions after the first big jump, the cheapest is not the worst and the most expensive is not the best.**

Time did not track cost either

Seconds per execution ranges from 19.3 to 29.4, and the ordering barely resembles the cost ordering. v10 is the fastest per execution and mid-table on cost. v5 is the slowest and mid-table on cost. **Wall time is a poor proxy for spend**, because spend is driven by tokens and time is driven by tool calls and the machine's own latency, installing software takes as long as it takes regardless of how many tokens the decision consumed.

The tool-call collapse at v10

One number in the table is discontinuous rather than gradual: tool calls per execution falls from **4.00 at v9 to 1.45 at v10**, while the score rises.

Some of this is real, v10 immediately follows a commit that made setting default applications work properly, which removes retry loops. Some of it is likely an artefact of how calls are counted when a task runs three times.

The record does not settle which, and this paper will not pretend it does: it is flagged here as the one movement in the corrected table that has no confirmed explanation.

05 The regression that never reached the repository

Discharges claims D015, D016, D017, D018.

The most useful twenty minutes in this record produced the worst version in it.

What happened, by the clock

TIME	EVENT	SCORE
19:53	v5 benchmarked	96.4
	, <i>no commits</i>	
20:14	v6 benchmarked	90.1
20:16	commit 41e8f35 <i>Stop misreading script structure as commands</i>	
20:16:40	v7 benchmarked	94.3

Twenty-one minutes from the good version to the bad one. **Two minutes** from the bad result to the diagnosis. **Forty seconds** from the commit to the confirming run.

The regression was never committed

There are no commits between v5 and v6. The v6 benchmark was run against an **uncommitted working tree**: an experiment in progress, measured before it was saved.

That is the whole point. The change that cost 6.3 points and raised cost 11% exists nowhere in the repository's history. It was never merged, never shipped, and never had to be reverted, because it was measured first. The only trace it left is a result file and this paper.

An engineer without a benchmark makes the same change, sees nothing obviously wrong, and commits it. It then costs a fortnight to find, because by the time anyone notices, twenty other things have changed.

What broke, and whether the diagnosis holds

Two tasks collapsed and two others lost points:

TASK	CATEGORY	CHANGE
B24 Update installed software	packages	-88 points
B18 Inspect the Run keys	registry	-80 points
B14 Batch a multi-service change	services	-40
B20 Block hive deletion	safety	-40

The commit that followed names the cause: *"Stop misreading script structure as commands."*

That diagnosis is testable against the damage, and it survives. B18 reads registry Run keys and B24 drives a package manager, both require the agent to construct a shell invocation in which the **structure** of the script (pipes, quoting, subexpressions) is distinct from the **command** being run. An agent that confuses the two produces something that looks plausible and does not execute. B14 batches service changes, which is the same problem again.

The fourth is more interesting. **B20 Block hive deletion is a safety task**: the correct behaviour is to refuse or qualify a dangerous registry operation. A parsing bug in command construction cost 40 points on a task about refusing to do something. **A defect in how an agent builds commands degraded its ability to decline one.**

Capability and safety are not separable layers, and this is a small, dated, measured instance of it.

The same safety task broke again

B20 lost 40 points at v6, recovered, and **lost 40 points again at v9**: a version whose preceding work was entirely test coverage and bug fixes.

A capability task that regresses is a bug. **A safety task that regresses twice, under two unrelated changes, is a fragile check or a fragile behaviour**, and either way it is the row in this record that most deserves attention. The published data does not settle which; nothing in the transcripts was coded for it. **It is stated here as an open finding rather than resolved by assertion.**

What it cost to find out

The v6 benchmark run took thirteen minutes. Against the alternative, committing a change that breaks package management, registry inspection and a safety refusal, and finding out later: that is the cheapest thing in this paper.

And one of the two regressions this record caught was caught before it entered the repository at all.

The generalisable part

Three things made this work, and none of them is the agent:

1. **The benchmark was cheap enough to run on an uncommitted tree.** At thirteen minutes it is affordable at experiment scale rather than release scale. A two-hour suite would have been run after the commit, not before.
2. **It reported per task, not per suite.** An overall score of 90.1 says something got worse. B24 -88, B18 -80 says *where*, and that is what made a two-minute diagnosis possible.
3. **The result was trusted enough to act on immediately.** The commit came before any attempt to argue the number was noise, which, at one repetition per task, it might have been. The next section shows what happened when that question was finally taken seriously.

06 What the tests bought

Discharges claims D019, D020, D021.

Between v8 and v9 the agent's author did what every engineering guide recommends. The benchmark scored it at **minus 0.3**.

The work

Five hours, four commits, and the largest quality push in the record:

TIME	COMMIT
02 Aug 00:21	<i>Cover the three things that fail silently</i>
02 Aug 01:14	<i>Ignore coverage artifacts</i>
02 Aug 01:46	<i>Coverage 33% → 66%, and four bugs the new tests found</i>
02 Aug 01:48	<i>Document the eight suites and the 66% coverage</i>

Unit-test coverage doubled. Four real bugs were found and fixed. Eight test suites were documented.

The result

	V8	V9
score	95.9	95.6
\$/execution	\$0.01189	\$0.01085
s/execution	25.9	20.9

The benchmark score fell 0.3 points. Cost fell 9% and wall time fell 19%.

Within the tasks, the movement was not flat. Two telemetry tasks improved sharply, B05 Disable telemetry using verified paths gained 32 points and B07 Advertising ID and activity history gained 20, while three others fell, including B20, the safety task, by 40.

What this does and does not mean

It does not mean the testing was wasted. Four bugs were found. Three things that fail silently are now covered. Cost per execution dropped 9% and stayed down. Those are real returns and they are visible in the same table.

It does mean the two are measuring different things, and that the relationship between them is weaker than the recommendation implies. Unit tests assert that functions behave as their author intended. The benchmark asserts that an agent completes a task on a real machine. A codebase can become considerably more correct in the first sense without moving the second, because the second is dominated by decisions the model makes at runtime, which no unit test reaches.

Four bugs found by tests is four defects that would have surfaced eventually, in a path the benchmark's thirty tasks do not exercise, on a machine configured differently, or in a case that only appears on a fresh install. Three of the four commits that followed this run were exactly that: bootstrapper defects *"that would only have appeared on a fresh install."*

The honest summary is that internal quality work and external task performance are different axes, and this record measured only one of them. A reader who wants a rule from this should not take "do not write tests." They should take:

Do not expect a coverage number to move a capability number, and do not treat a flat benchmark as evidence the testing failed.

The n=1 problem, which this section cannot escape

The score fell by 0.3 points on **a single run of thirty tasks**. At one repetition per task, a difference that small is not distinguishable from noise, and this paper will not claim it is.

That limitation is not incidental: it is the reason the next thing the author did, four commits later, was add --repeat to the benchmark *"so the benchmark gives a number worth trusting."* v10 is the first version in this record with any variance estimate at all: three runs per task, with a spread recorded per task.

So the correct reading of the v8 → v9 comparison is: **the testing work did not produce a detectable capability gain at the resolution this instrument had at the time**. The instrument acquired better resolution immediately afterwards, and by then the question had moved on.

That is a small, real example of a general problem in agent engineering. **The measurement improves after the decision it should have informed**, because the need for the better measurement only becomes obvious once a result is too small to interpret.

07 How an agent actually fails

Discharges claims D022, D023, D024.

Two hundred and seventy task-executions produced **four recorded errors**. Every one is the same failure.

The taxonomy, in full

COUNT	CATEGORY	RECORDED ERROR
2	packages	<i>looped until it hit the step limit</i>
1	packages	<i>hit the step limit without finishing</i>
1	debloat	<i>hit the step limit without finishing</i>

That is the complete list. There are no crashes, no malformed tool calls that the runner recorded as errors, no refusals logged as failures, and no cases of the agent doing something destructive that the grader caught as an error rather than a lost point.

The agent does not fail by doing the wrong thing. It fails by not finishing.

Why this is the more useful finding

The failure modes that dominate discussion of agents are the vivid ones: hallucinated commands, destructive actions, confident wrong answers. This record contains none of them as *errors*, and it is a record of an agent with shell access, registry access and package-manager access on a live Windows machine, including four tasks

explicitly designed to invite dangerous behaviour.

What it contains instead is exhaustion. The agent starts a task, makes progress, and runs out of budget before it can declare completion. Three of the four instances are package management, which is the slowest and least predictable category in the suite, installs that prompt, updates that enumerate, package managers that take their time.

This has a direct engineering consequence, and it is not "make the model smarter":

The binding constraint is the step budget, and the tasks that hit it are the ones with slow, variable external latency.

An agent that fails by running out of turns is fixed by better budgeting, by tools that do more per call, or by recognising a long-running external operation and waiting on it rather than polling, not by a better prompt.

The two tasks that never succeeded

Across all seven versions, **twenty-eight of thirty tasks reached a perfect score at least once**. The two that never did are both in the same category:

TASK	BEST EVER	LATEST (V10)
B21 Install two apps with correct IDs	90	88
B24 Update installed software	95	95

Package management is where three of the four step-limit errors occurred, and it is where the only two never-solved tasks live. **The failure concentrates**. One category out of nine accounts for the entire unsolved remainder of the suite and most of its errors.

That is worth more than a general observation about agent reliability. It says: on this workload, the marginal engineering effort belongs in package management, and effort spent anywhere else is polishing tasks that already score 100.

What the error field does not capture

The count of four is a count of *recorded* errors, and the field is written by the runner when a run terminates abnormally. It is not a count of everything that went wrong.

A task that scored 60 because it called the wrong tool records **no error**, it records lost points. Across the record there are hundreds of lost points and only four errors, so the great majority of failure in this suite is silent, graded rather than logged.

This is the correct design: the grader's job is to score the artefact, not to interpret the transcript, but it bounds what this section can claim. **The right statement is: of failures severe enough to abort a run, all were step-limit exhaustion**. The larger population of partial failures is visible only as scores, and nothing in the published record codes *why* a partial failure happened.

Coding that would require reading the transcripts, which are published but were not analysed for this paper. It is the single highest-value piece of unfinished work in this record, and section 09 lists it as such.

08 What the whole thing cost

Discharges claims D025, D026, D027.

The bill

versions benchmarked	7
task records / executions	210 / 270
input tokens	6,406,701
output tokens	316,133
wall-clock across all runs	104 minutes

A complete measured engineering record of a production agent: seven versions, two caught regressions, and a 10.5-point improvement.

The most expensive single version was v10, and it was expensive because it ran everything three times to produce a number worth trusting. The cheapest was v7, which was also the best value ever achieved.

Where the money goes

Input tokens outnumber output tokens **20 to 1**. At the published rates that makes input **91% of the bill**, despite output being priced at twice the rate.

This is characteristic of tool-using agents and it inverts the usual intuition about prompt engineering. An agent's cost is dominated not by what it writes but by what it re-reads: the system prompt, the tool schemas, and the accumulated transcript, sent again on every turn. With 3.37 to 5.30 tool calls per task-execution, each of those turns re-sends the context.

Two consequences follow directly, and both are visible in the record:

Cost tracks session length, not tool count. Across the seven versions, cost per execution correlates with **output tokens per execution at $r = 0.99$** , almost perfectly, and with tool calls per execution at only **$r = 0.34$** .

That is worth pausing on, because output is just 9% of the bill. Output tokens are not driving the cost; they are an almost exact *proxy* for how long a session ran, and session length is what determines how many times the system prompt and accumulated transcript get re-sent as input. **The thing to shorten is the session, and output length is the cheapest available measure of it.**

A cache discount would change the absolute numbers dramatically and the ranking hardly at all. Because the repeated content is the same system prompt and schemas across every task, it is exactly what a provider's cache is designed to serve. The corroborating price table in this programme puts cached input at about **1/120th** of the uncached rate. Applied uniformly, the bill falls a long way; applied uniformly, the ordering of seven versions measured the same way does not move.

The comparison that puts it in scale

The barrier to a longitudinal benchmark of this kind is not money and it is not compute.

That is the finding most likely to be useful to a reader deciding whether to measure their own agent. It is the discipline of running the suite before committing, and of keeping the results when they are unflattering.

Two of the seven versions here are unflattering. They are the reason the record is worth publishing.

What this does not price

Engineering time is absent. The record spans thirty-one hours of wall clock and twenty-one commits, and none of that labour is measured here. Compute is the smallest line in any honest accounting of what building this agent cost.

The machine is absent. These tasks install software, modify the registry and change services on a real Windows 11 install. The cost of having a machine that can be dirtied, and of restoring it, is not measured.

The failures that were not benchmarked are absent. v4 exists in the version numbering and in no result file. Whatever it cost to build and abandon is not here.

So the honest headline is narrow, and it is stated narrowly: **this instrument measures the compute cost of seven versions of a production agent, and nothing else about what it cost to build.**

09 The negative control, and what it says about every number above

Discharges claims D035, D036, D037, D038, D039.

An earlier draft of this paper listed the absence of a negative control as the weakest point in the instrument, and said the suite should acquire one. It now has one, it cost nothing to run, and it changes how every score in this paper should be read.

What was done

Three synthetic agents were graded by the **real graders against all thirty real benchmarks**, with **zero API calls**. Each is a Run object handed straight to the scoring function:

AGENT	BEHAVIOUR
null	does nothing at all, no tool calls, no answer
babbler	emits one plausible paragraph about Windows, telemetry and JSON; calls no tool
flailer	calls five real tools with junk arguments, then emits the same paragraph

If the suite measures what it claims to, all three should score near zero.

What they scored

AGENT	EARNED	OF 302	PERCENT
null	69	302	22.8%
babbler	111	302	36.8%
flailer	96	302	31.8%
<i>the measured agent, v10</i>			97.6%

An agent that does absolutely nothing scores 22.8%. An agent that says something plausible and touches nothing scores **36.8%**, half again as much as doing nothing, and more than the agent that at least called some tools.

Why: the negative checks pass vacuously

The reason is exact and it is arithmetic rather than interpretation. Six check types award points for **not** doing something, and a do-nothing agent satisfies every one of them trivially:

CHECK TYPE	PASSES VACUOUSLY	POINTS
no_repeated_calls	9 of 9	20
no_mutation	6 of 6	22
max_tool_calls	8 of 8	16
no_self_confirm	1 of 1	5
tool_arg_not_contains	1 of 1	4
tool_not_used	1 of 1	2
total	26 checks	69

Sixty-nine points, which is precisely the null agent's score. **Every point a do-nothing agent earns comes from a restraint check that costs nothing to satisfy by inaction.**

Section 02 of this paper described those 69 points admiringly, as 23% of the suite awarded for not doing things, and argued an agent could not score well by working harder. That is true and it was only half the picture. **The other half is that an agent can collect all of them by not working at all.**

The babbler is the more serious finding

The null agent's 22.8% is a floor that can be subtracted. The babbler's **36.8%** cannot, because it means **fourteen points of answer_contains checks are satisfiable by generic plausible prose** that was written without looking at the machine.

The paragraph used asserts that the system is Windows 11, that telemetry, registry and service settings were checked, and that the catalogue is JSON. Every one of those is true of this machine and none of it was discovered, it was written into the control on purpose to test exactly this.

That is a real weakness in ten checks worth 34 points, and it is the one a reader should weigh when reading any aware-category result in section 04.

What this does to the paper's numbers

The scores in sections 04 to 08 are **not wrong**, and none of them changes. But the scale they sit on does not start at zero, so a raw score overstates how much of the achievable range an agent covered.

Rebasing each version onto the measured floor of 22.8%:

VERSION	RAW SCORE	SHARE OF REAL HEADROOM
v3	87.1%	83.3%
v5	96.4%	95.3%
v6	90.1%	87.2%
v7	94.3%	92.6%
v8	95.9%	94.7%
v9	95.6%	94.3%
v10	97.6%	96.9%

The **ordering is unchanged**, every comparison in this paper survives, and the two regressions remain regressions. What changes is the impression a reader takes from "97.6": the agent covered **96.9% of the range the suite can actually discriminate**, and the suite discriminates by **74.8 points**, not 97.6.

Every version-to-version delta in this paper is a difference between two numbers on the same scale and is therefore unaffected. The v5→v6 regression is -6.3 raw and -8.1 rebased; it did not get smaller.

What it establishes about the instrument

The suite is sound for comparison and misleading as an absolute. It separates a working agent from a do-nothing agent by 74.8 points, which is a real and large discrimination: that is the property a longitudinal record needs, and it is now demonstrated rather than assumed.

What it is not is a percentage of capability. **A vendor quoting "97.6% on our benchmark" from a suite with a 22.8% floor is quoting a number whose bottom quarter is free**, and nothing in the published output would reveal that. This suite is the author's own and it had exactly that defect for the whole of its recorded history.

What should change

1. **Report scores against the floor**, or drop the vacuous passes from the denominator. Either is a one-line change to the reporting and both are more honest than the raw percent. 2. **Make the negative checks conditional on the task being attempted**. `no_mutation` should award nothing to an agent that did not run, because it demonstrated nothing. 3. **Tighten `answer_contains`**. Ten checks are partially satisfiable by plausible prose; they should require a value that could only be obtained by looking. 4. **Run this control before every reported run**. It costs no API calls and takes under a second, and it is the difference between a number and a number worth trusting.

The control script is published with this paper. **It should have existed before the first benchmark run, not after the seventh**, and the only reason its absence did not corrupt any comparison here is that every version was measured on the same broken scale.

10 Limitations, and the errors in this paper

Discharges claims D028, D029, D030, D031.

What this record cannot support

One machine. Every measurement is one Windows 11 install. Package management, defaults and recovery: the categories where this agent is weakest, are exactly the ones most sensitive to machine state, and a fresh install would exercise them differently.

One model, one week. `deepseek-v4-pro` throughout, at prices dated 1 to 6 August 2026. Holding the model constant removes the worst confound and creates a narrower one: nothing here generalises to a different model.

`n=1` for six of seven versions. **Only v10 has repetitions, and therefore only v10 has any variance estimate. The 0.3-point difference between v8 and v9 is below the resolution of a single run, and this paper does not treat it as real.**

Where a difference is smaller than a few points, it should be read as unmeasured rather than measured.

The suite is close to saturated. Twenty-eight of thirty tasks reached 100% in at least one version. There is very little headroom left, and a version scoring 97.6 has more to do with the ceiling than with the agent.

The negative control was built after the fact, and it found something. Section 09 grades three do-nothing agents against the real graders: the null agent scores **22.8%**, the babler **36.8%**. Twenty-six restraint checks worth 69 points pass vacuously when an agent does nothing. The suite discriminates by 74.8 points rather than

97.6, and **every score in this paper sits on a scale whose floor is not zero**. Comparisons are unaffected; absolute percentages are overstated. This control should have existed before the first run.

The transcripts are unread. **Every run wrote a transcript and all of them are published. None was coded. So the great majority of failure in this record: the hundreds of lost points that were graded rather than logged as errors, has no diagnosis attached.** This is the largest piece of unfinished work in the paper.

A category imbalance. Nine categories from two to four tasks each. Package management, where all the interesting failure is, has four. Conclusions about where an agent fails are conclusions about a suite with four package-management tasks in it.

Errors made in producing this paper

Four figures were wrong in the first draft and are corrected here rather than quietly fixed, because the whole argument of section 03 is that this kind of error is easy and flattering.

1. Task-executions were reported as 210. That is the number of task *records*, thirty tasks times seven versions. The number of executions is **270**, because v10 runs each task three times. The same conflation the paper warns about in section 03, made by the author of section 03, in the section immediately after it.

2. Output tokens were reported as 213,809. The correct figure is **316,133**, and the input-to-output ratio is **20 to 1**, not 30 to 1. Input's share of the bill is **91%**, not 94%.

3. The cost correlation was stated backwards. The draft claimed cost per execution correlates strongly with tool calls and weakly with output tokens. It is the reverse: **$r = 0.99$ with output tokens and $r = 0.34$ with tool calls**. The corrected reading is more useful than the wrong one, output length is a near-perfect proxy for session length, and session length is what drives the re-sent input that dominates the bill.

4. A cross-paper spend comparison that was both imprecise and beside the point. It has been removed rather than corrected.

All four were caught by recomputing every number from the published files before the draft was assembled, not by review. That check now runs as part of building the paper.

The conflict of interest

The author built the agent, wrote the benchmark, wrote the graders, ran every version, and is publishing the result. There is no independent party anywhere in this record.

The mitigations are structural rather than assurances:

- **Every input is published:** the thirty tasks with their prompts and checks, all seven result files, the extraction and analysis code, and the tidy 210-row record. Every figure in this paper can be recomputed by someone who does not trust it.
- **The regressions are in the body**, not an appendix. Two of seven versions are worse than their predecessor and both have their own tables.
- **The best-scoring version is not presented as the best version.** v10 tops the score table and is 25% worse value than v7, and the paper says so.
- **The commit log corroborates the one methodological decision everything rests on.** The normalisation in section 03 is not an inference; the engineer recorded "97.6 over 90 runs" at the time.

What none of that fixes: a saturated suite written by the same person as the agent will tend to be a suite the agent passes. **The correct response is an independent task set, and this paper does not have one.**

What would make the next version of this record better

In order of value:

1. ~~A negative control.~~ **Done, and it changed the reading of every score** (section 09). The remaining work is to act on it: make the restraint checks conditional on the task being attempted, and tighten the ten answer_contains checks that plausible prose can partly satisfy. 2. **Repetitions on every version, not just the last.** The --repeat flag now exists. Re-running v7 and v9 at three repetitions would cost about **\$0.85** and would settle whether the differences this paper declines to interpret are real. 3. **Code the transcripts.** The failure taxonomy currently rests on four error strings out of 270 executions. 4. **Harder tasks.** Twenty-eight of thirty at ceiling means the suite has largely stopped measuring. 5. **A second machine.** Ideally a fresh install, where three of the four commits after the last benchmark suggest the real defects live.

11 What follows for anyone building an agent

Discharges claims D032, D033, D034.

Seven recommendations, each tied to something in this record rather than to general opinion. Where the evidence is thin, it says so.

1. Build the benchmark before you need it, and make it cheap enough to run on uncommitted work

The regression at v6 was caught because a thirteen-minute, forty-cent suite could be run against a working tree that had not been committed yet. **The bad change never entered the repository.**

A two-hour suite would have been run after the commit. A suite that costs real money would have been run at release. The value here came almost entirely from being cheap enough to run *speculatively*, and that is a design constraint, not an afterthought.

Evidence: strong. One clean instance, fully dated, with the commit log to corroborate it.

2. Report per task, never only per suite

An overall score of 90.1 says something broke. B24 -88, B18 -80, B14 -40, B20 -40 says where, and it made a two-minute diagnosis possible.

Per-suite scores are for tracking. **Per-task scores are for debugging**, and the difference between them is the difference between knowing you have a problem and knowing what it is.

Evidence: strong.

3. Normalise before you compare, and record protocol changes as discontinuities

The measurement protocol changed at v10 and nothing in the output said so. Read straight, the record shows a 3.3× cost explosion that did not happen.

Two habits prevent it: **state the unit** on every figure, everything in this paper is per task-execution, and **treat a change to the measurement as an event in the data**, recorded alongside the results it makes incomparable.

Evidence: strong, and self-implicating. The author of that section then miscounted executions in the next one.

4. Budget for exhaustion, not for error

All four recorded failures across 270 executions were step limits. None was a crash, a malformed call or a destructive action, on a suite that includes four tasks designed to invite exactly that.

The tasks that hit the limit are the ones with slow, variable external latency: package installs, software updates, removals. **An agent that fails by running out of turns is not fixed by a better prompt.** It is fixed by budgets that account for slow external operations, by tools that do more per call, or by waiting on a long-running operation rather than polling it.

Evidence: moderate. Four errors is a small number, but their concentration in one category is unambiguous.

5. Watch the safety tasks separately from the capability score

B20 Block hive deletion regressed twice, under two unrelated changes: a command-parsing bug at v6 and a testing pass at v9. Both times it lost 40 points; both times the overall score barely moved.

A safety behaviour that degrades under changes with nothing to do with safety will not be visible in an aggregate. It needs its own line. And note what caused the first one: **a defect in how the agent constructed commands degraded its ability to refuse one.** Capability and safety are not separable layers.

Evidence: moderate. Two occurrences, one instrument, and the record cannot distinguish a fragile check from a fragile behaviour. That ambiguity is itself the reason to watch it.

6. Do not expect test coverage to move a capability score

Coverage went from 33% to 66% and four bugs were fixed. The benchmark moved **-0.3 points**, which at one repetition per task is not distinguishable from noise.

This is not an argument against testing, four real defects were found, cost per execution fell 9%, and three of the four commits after the last benchmark fixed bootstrap bugs the benchmark could never have reached. It is an argument against expecting one number to move the other. **They measure different things, and a flat benchmark after a testing push is the expected result, not a disappointing one.**

Evidence: weak. One transition, n=1, a difference below the instrument's resolution. Stated because the direction is surprising, not because the magnitude is established.

7. Publish the versions that got worse

Two of seven versions here are worse than their predecessor. They are the only reason this record is worth anything to a reader.

A record of monotone improvement teaches nothing, because it does not resemble the process. What a reader about to build an agent needs to know is that roughly **one change in three made things worse**, that the worst one was caught in twenty minutes by a suite costing forty cents, and that the best-scoring version was not the best version.

None of that is visible in a capability claim, and none of it is expensive to produce. **What it costs is not money, it is publishing the run where the number went down.**

Evidence: this whole paper.

The one-line version

Measure before you commit, report per task, normalise before you compare, and keep the results that embarrass you: because the alternative is an engineering record that looks like marketing and teaches nobody anything, including its author.

12 Claim ledger

Every factual claim in this paper, with where it came from and whether it was checked. The ledger is the contract: a claim not in it is not made, and a claim in it can be recomputed from the published result files by the code published beside them.

ID	STATUS	CLAIM	SOURCE
D001	verified	Seven versions of a Windows automation agent were benchmarked over thirty-one hours on 1 to 2 August 2026 against the same thirty tasks, the same model and the same grader	data/runs/
D002	verified	210 task records covering 270 task-executions, because the last version runs everything three times	data/versions-by-task.csv
D003	verified	Two of the seven versions scored worse than the version preceding them, and both are published	data/analyse.py
D004	verified	The suite is thirty tasks worth 302 points across nine categories, four of which are safety tasks where the correct behaviour is to refuse or qualify	data/benchmark-tasks.json
D005	verified	Only 34 of 302 points, 11 per cent, come from what the agent says; the rest come from what it did	data/benchmark-tasks.json
D006	verified	Every one of the 270 task-executions ran on deepseek-v4-pro; no version changed model, temperature tier or provider	data/runs/
D007	verified	Costs are computed at 0.435 and 0.87 per million input and output tokens with no cache discount applied, so every cost figure in the paper is an upper bound	data/extract.py
D008	verified	Read from the run-level totals, v10 costs 3.3 times v9 and takes 2.8 times as long	data/runs/
D009	verified	v10 runs every task three times and v3 to v9 run each once; nothing in the run-level summary says so	data/runs/2026-08-02_020605_v10-stable-baseline.json
D010	verified	Per task-execution v10 costs 10 per cent more than v9 and runs 8 per cent faster	data/analyse.py
D011	verified	The seven versions scored 87.1, 96.4, 90.1, 94.3, 95.9, 95.6 and 97.6, and v4 was never benchmarked	data/versions-by-task.csv
D012	verified	The largest single gain, 9.3 points at v5, arrived in the first thirty-one minutes and accounts for 89 per cent of what the remaining five versions produced	data/analyse.py
D013	verified	v7 is the best value in the record at 0.00976 per hundred points, 25 per cent cheaper per point than the top-scoring v10, and nothing in the record announces it	data/analyse.py
D014	NOT-CLAIMED	Tool calls per execution fall from 4.00 at v9 to 1.45 at v10 while the score rises, and the record does not settle why	data/versions-by-task.csv
D015	verified	The v6 regression was benchmarked against an uncommitted working tree and never entered the repository	git history of the agent
D016	verified	Twenty-one minutes from the good version to the bad one, two minutes from the bad result to the diagnosis, forty seconds from the commit to the confirming run	git history and run timestamps
D017	verified	A defect in how the agent constructed shell commands cost 40 points on B20, a safety task about refusing a dangerous registry operation	data/versions-by-task.csv

ID	STATUS	CLAIM	SOURCE
D018	NOT-CLAIMED	B20 lost 40 points at v6 and again at v9 under two unrelated changes; whether that is a fragile check or a fragile behaviour is not settled	data/versions-by-task.csv
D019	verified	Unit test coverage doubled from 33 to 66 per cent and four bugs were fixed; the benchmark score moved minus 0.3 points	data/versions-by-task.csv
D020	LIMITATION	At one repetition per task a 0.3-point difference is below the instrument's resolution and is not treated as real	data/analyse.py
D021	verified	Internal quality work and external task performance are different axes, and this record measured only one of them	data/versions-by-task.csv
D022	verified	Across 270 task-executions there were four recorded errors and every one was a step limit	data/runs/
D023	verified	Twenty-eight of thirty tasks reached a perfect score in at least one version; the two that never did are both package management	data/versions-by-task.csv
D024	LIMITATION	The error count counts only runs that terminated abnormally; the great majority of failure in this suite is graded rather than logged	data/extract.py
D025	verified	The whole record consumed 6,406,701 input tokens and 316,133 output tokens across 104 minutes of wall clock	data/extract.py
D026	verified	Input is 91 per cent of the bill despite output being priced at twice the rate, because a tool-using agent re-reads its context on every turn	data/analyse.py
D027	verified	Cost per execution correlates with output tokens per execution at $r = 0.99$ and with tool calls per execution at $r = 0.34$	data/analyse.py
D028	LIMITATION	One machine, one model, one week, and $n=1$ for six of the seven versions	data/runs/
D029	LIMITATION	The suite is close to saturated: twenty-eight of thirty tasks reached 100 per cent in at least one version	data/versions-by-task.csv
D030	LIMITATION	Every run wrote a transcript, all are published, none was coded, so the majority of failure in this record has no diagnosis attached	data/runs/
D031	DEFECT-CORRECTED	Four figures were wrong in the first draft and are corrected in the body: the execution count, the output token total, the direction of the cost correlation, and a cross-paper comparison that was removed	data/analyse.py
D032	verified	A benchmark cheap enough to run on uncommitted work is what caught the only regression that never entered the repository	data/runs/
D033	verified	A safety behaviour that degrades under changes unrelated to safety will not be visible in an aggregate score and needs its own line	data/versions-by-task.csv
D034	verified	Roughly one change in three in this record made the agent worse	data/versions-by-task.csv
D035	verified	Three synthetic agents were graded by the real graders against all thirty real benchmarks with zero API calls	data/negative-control.py
D036	verified	An agent that does nothing at all scores 22.8 per cent, and an agent that emits one plausible paragraph and touches nothing scores 36.8 per cent	data/negative-control-results.json
D037	verified	Every point the null agent earns comes from twenty-six restraint checks worth 69 points that pass vacuously when nothing is attempted	data/negative-control-results.json

ID	STATUS	CLAIM	SOURCE
D038	verified	Rebasing on the measured floor leaves the ordering unchanged and every version-to-version delta unaffected; the suite discriminates by 74.8 points rather than 97.6	data/negative-control-results.json
D039	DEFECT-CORRECTED	The negative control should have existed before the first benchmark run rather than after the seventh, and the only reason its absence corrupted no comparison is that every version was measured on the same broken scale	data/negative-control.py

Creative Commons Attribution 4.0. No sponsor, client or vendor paid for, reviewed or approved this work. 39 claims, all of them recomputable from the published data.

ABOUT THE RECORD

Two hundred and seventy task-executions across seven versions, one model held constant, recorded while building the agent rather than for publication.

WHAT IT SHOWS

Cost tracks session length rather than tool count. All four recorded failures were step limits. The best-scoring version is 25 per cent worse value than the best version.

CORRECTIONS AND CONFLICTS

The author built the agent, the benchmark and the graders. Four figures were wrong in the first draft and are corrected in the body, and the negative control that should have come first arrived after the seventh run.

© 2026 Nikolaos Broikos. Text and data under CC BY 4.0, code under MIT. The data, code and claim ledger are in [data/paper-05/](#) so any figure here can be recomputed, checked or disputed.