Clear Every Third Task: A Measured U-Curve in the Context Economy of Coding Agents

Evgenii Arsentev
ARSENTEV.AI, Barcelona, Spain
hello@arsentev.ai · ORCID 0000-0002-9120-7298 · https://arsentev.ai
Version 1.2, 15 September 2026. Corrects the modeled costs of versions 1.0 (DOI 10.5281/zenodo.22699668) and 1.1 (DOI 10.5281/zenodo.22749959). All versions: DOI 10.5281/zenodo.22699667.

Correction (2026-09-15)

Versions 1.0 and 1.1 of this report priced the measured token counters incorrectly. The errors are the author's. They were found on 15 September 2026, when the costs were recomputed from the raw session logs against Anthropic's published price list (https://platform.claude.com/docs/en/about-claude/pricing, retrieved 15 September 2026). There were two errors:

1. **Wrong price level.** Costs were computed at $3.00 and $15.00 per million input and output tokens, with cache reads at $0.30. The model used, `claude-sonnet-5`, is listed at $2.00 and $10.00. Its introductory price was made the standard price on 10 August 2026, and the previously scheduled increase to $3.00/$15.00 was cancelled. Cache reads cost 0.1× input ($0.20). On its own, this error scales every cost by the same factor, so it changes no percentage and no test result.

2. **Wrong cache-write duration.** Every cache write was priced at the 5-minute rate (1.25× input, $3.75 in versions 1.0 and 1.1). The session logs record the duration of every cache write. In all 36 runs, every cache write is a 1-hour write, which is billed at 2× input ($4.00). This error changes how much cache writes weigh against everything else. It therefore changes the percentages, the p-values and part of the conclusions.

Token counters, model calls, sessions, wall-clock times and test outcomes are unchanged. What changed:

Quantity	Versions 1.0 and 1.1	Version 1.2
Mean cost per run at 1 / 2 / 3 / 4 / 6 / 12 tasks per session (USD)	2.680 / 2.399 / 2.138 / 2.279 / 2.281 / 2.473	2.165 / 1.854 / 1.622 / 1.704 / 1.679 / 1.780
Condition with the lowest mean cost	every third task	every third task (unchanged)
Clearing after every task vs every third	+25.3%, p = 0.0022	+33.5%, p = 0.0022, Holm 0.011
Clearing every second task vs every third	+12.2%, p = 0.0108	+14.3%, p = 0.0022, Holm 0.011
Never clearing vs every third	+15.6%, p = 0.0108	+9.7%, p = 0.046, Holm 0.14: not significant
Every fourth / every sixth vs every third	+6.6%, p = 0.097 / +6.7%, p = 0.21	+5.1%, p = 0.17 / +3.5%, p = 0.45
Cost drop from every task to every third	20.2%	25.1%
Price of a cache write relative to a cache read	12.5×	20×
Spend at the optimum: cache read / cache write / output	44% / 23% / 33%	39.0% / 32.3% / 28.7%

Table 0: Figures affected by the correction. Holm: p-value adjusted over the five comparisons against the optimum.

What this changes in the conclusions. Versions 1.0 and 1.1 claimed that both extremes are significantly more expensive than the optimum. That claim is withdrawn for never clearing. Never clearing is 9.7% more expensive in the point estimate, but p = 0.046 before and 0.14 after Holm adjustment, so this design does not establish the difference. Clearing after every task and clearing every second task remain significantly more expensive than clearing every third task, including after adjustment. The lowest mean is still at three tasks per session, and three, four and six tasks remain indistinguishable. The headline gap between clearing after every task and the optimum grows from 25.3% to 33.5%.

Limitations now stated explicitly. "Zero failures" refers to tests that the agent wrote itself, and the number of tests per run differed between conditions. On average there were 116.8 tests per run when clearing after every task

and 106.8 when never clearing. The reference condition was chosen after the fact as the observed minimum.

The recomputation scripts and their outputs are included in this record as `recompute-2026-09-15.zip`. They re-extract the tokens with the cache-write duration split, re-price them, and rerun the exact permutation tests with Holm adjustment. Corrected per-run costs are in `runs_ucurve_costs_v1.2.json`.

Abstract

Practitioners running LLM coding agents face a question with no published answer: how often should an agent's context be cleared? We ran a fixed suite of twelve programming tasks under six session-length policies — a fresh session every task, every two, three, four, six, and one session for all twelve — with six replicates each, holding the model, the tasks and their order constant. Cost is not monotone in session length. It falls from $2.17 per run when the context is cleared after every task to $1.62 at every third task, then rises to $1.78 when the context is never cleared. Exact permutation tests show that clearing every third task is significantly cheaper than clearing after every task (+33.5%, $p = 0.0022$, the smallest value this design can produce; 0.011 after Holm adjustment over five comparisons) and than clearing every second task (+14.3%, same p). Never clearing is 9.7% more expensive in the point estimate, but that difference is not significant ($p = 0.046$ unadjusted, 0.14 adjusted). The differences between clearing every three, four and six tasks are within noise ($p = 0.17$ and $p = 0.45$). The low region is therefore a plateau rather than a point. Clearing after every task sits clearly outside it; whether never clearing does too is not established at this sample size. No run produced a test failure: across all 36 runs, 4086 tests executed and 4086 passed. That figure is a total over the experiment, not a per-condition value. The suite is written by the agent as part of each task, so the number of tests varies from run to run (97 to 127), and this gate cannot establish that the conditions did equal work. The U-shape decomposes cleanly. Cache writes fall monotonically with longer sessions (253k → 88k tokens), because each fresh session re-primes the cache. Context carried per model call rises monotonically (38k → 71k tokens), because nothing is ever dropped. Their product, cache reads, is U-shaped with a minimum at three. On the model we used, a 1-hour cache write costs 20× a cache read per token, so neither term can be ignored. We release the run-level token counters and the harness. This version corrects the prices used in versions 1.0 and 1.1; see the Correction section above.

Keywords: LLM agents, inference economics, prompt caching, context management, reproducible measurement

1 Introduction

Every team running coding agents in production develops a folk rule about session hygiene. Some clear the context after every task, reasoning that a short prompt is a cheap prompt. Others never clear it, reasoning that the model should keep what it has learned about the codebase. Both rules are defended with confidence and neither, to our knowledge, has been measured.

The question is not academic. In agent workloads the dominant cost is not the tokens the model writes but the tokens it re-reads and re-primes. In our runs, output accounts for between a fifth and three tenths of spend (19.9% when clearing after every task, 28.7% at the optimum), and cached context accounts for the rest. Session length is the one lever that changes how much context is re-read. Every operator already has it: it costs nothing to change and requires no model access.

We measured it. Holding everything else fixed, we varied only how many tasks share a session. We report what happened to cost, to the structure of that cost, and to whether the work still got done.

Our contributions:

1. A controlled measurement showing that agent cost is **U-shaped** in session length. The interior optimum beats clearing after every task at conventional significance, including after adjustment for multiple comparisons.
2. The finding that the low region is a **plateau rather than a point**. Clearing every three, four or six tasks is statistically indistinguishable, and clearing after every task or every second task is significantly worse. Never clearing is more expensive in the point estimate but not significantly so.
3. A **decomposition** of that U into two monotone components pulling in opposite directions, which explains why the optimum is interior and why it is not obvious.

4. An **open dataset** of run-level token counters and the harness needed to reproduce the design on another model or task suite.

2 Setup

Task suite. Twelve independent programming tasks that build a small JavaScript utility package. Each task is stated in one sentence and each is verified by unit tests executed with `node --test`. Tasks and their order are identical in every run; nothing about the suite depends on the session policy.

Agent. The Claude Code CLI in non-interactive mode, one invocation per task. Continuation within a session uses the CLI's continue flag; a new session is started simply by omitting it. This is the same mechanism an operator would use in production, not a research harness that simulates clearing.

Conditions. Six session lengths: 1, 2, 3, 4, 6 and 12 tasks per session, giving 12, 6, 4, 3, 2 and 1 sessions per run respectively. Six replicates per condition, 36 runs, 432 task executions in total.

Model and toolchain. A single model was held fixed across all conditions. It was invoked through a CLI alias that resolves in the session logs to `claude-sonnet-5`. This is the only model string present across the experiment logs, and it carries no dated snapshot suffix, so this record does not pin the exact weights. The modeled cost uses Anthropic's published list prices for `claude-sonnet-5`: $2.00 and $10.00 per million input and output tokens, cache reads at $0.20, and cache writes at $4.00, the 1-hour rate (2× input). The session logs record the duration of every cache write, and all cache writes in the 36 runs are 1-hour writes, so the 5-minute rate ($2.50) does not apply. Versions 1.0 and 1.1 used $3.00 and $15.00, with every write priced at a 5-minute rate of $3.75 and reads at $0.30; see the Correction section. The harness ran on Node v24.20.0 with Claude Code 2.1.252 on a Linux host. The package under test is a dependency-free ESM package. **Temperature was never set** and is not recorded anywhere, so we report none rather than guess one. We deliberately do not compare models here: the claim is about the shape of the curve, and mixing models would confound it.

Measurement. Token counters are read per model call and aggregated per run. Two details matter for correctness. First, records must be de-duplicated by message identity, because streaming produces multiple records for the same call. Second, when merging duplicates, the element-wise maximum must be taken across usage fields. The first record of a streaming call carries an incomplete output count, and taking the first record rather than the maximum understates output by roughly a factor of two. We flag this because it is easy to get wrong, and it silently biases exactly the quantity under study. Cache writes must also be split by duration, because the 5-minute and 1-hour rates differ.

Cost. We report a modeled cost computed from the measured counters and the model's published per-token prices, separating input, output, cache write and cache read. We report modeled rather than billed cost so the numbers can be recomputed from the released counters under any price list.

Statistics. With six replicates per condition, we use exact two-sided permutation tests on the difference of means, enumerating all 924 splits. This makes no assumption about the shape of the cost distribution, which matters because agent runs are not normally distributed. The smallest two-sided p this design can produce is 2/924 = 0.0022. The reference condition is the observed minimum, and the five comparisons against it are reported both unadjusted and with a Holm adjustment over those five comparisons.

Quality control. Every run ends with the full test suite. We treat the test outcome as a gate rather than a secondary metric, because a cost comparison is meaningless if the cheaper policy also does less work. The gate is weaker than that reasoning requires, and we say so here rather than leave it implicit: the suite is not an independent instrument. Each task prompt asks the agent to write the test file itself, with a stated minimum number of cases and no maximum, and to run `node --test`. A run that does less therefore does not go red; it writes fewer tests. The gate detects a broken interface or a dropped task, which is what it was installed for. It cannot by itself certify that two conditions checked the same amount.

3 Results

Table 1 gives the mean over six replicates for each condition.

Tasks per session	Sessions	Model calls	Cache write (tok)	Cache read (tok)	Context per call (tok)	Cost (USD)
1	12	94.0	252,533	3,622,795	38,478	**2.165**
2	6	84.3	169,670	3,578,718	42,415	1.854
3	4	71.5	131,100	3,160,517	44,192	**1.622**
4	3	71.0	123,478	3,466,808	48,795	1.704
6	2	71.0	105,746	3,796,905	53,309	1.679
12	1	63.0	87,846	4,451,367	70,712	1.780

Table 1: Means over six replicates. Cost is modeled from measured counters at $2.00 input, $10.00 output, $4.00 for 1-hour cache writes and $0.20 for cache reads per million tokens (corrected in version 1.2).

The curve has an interior minimum. Cost falls by 25.1% from clearing after every task to clearing every third task, then rises again by 9.7% from every third task to never clearing. The most expensive policy is the one many teams adopt by default, a fresh session for every task, and it costs 33.5% more than the cheapest.

The left side is significantly worse, the middle is a plateau, and the right side is not established. Clearing every third task beats clearing after every task with $p = 0.0022$, the smallest value a 6-versus-6 permutation test can yield (0.011 after Holm adjustment). Every replicate at the optimum was cheaper than every replicate at the most expensive condition. It also beats clearing every second task ($p = 0.0022$, Holm 0.011), again with every replicate cheaper. Against never clearing, the difference is +9.7% with $p = 0.046$ unadjusted and 0.14 after Holm adjustment, so this design does not establish it. Against clearing every fourth or sixth task, the difference is far from conventional thresholds ($p = 0.17$ and $p = 0.45$). The honest reading has three parts. Anything from three to six tasks per session sits in the same flat bottom. Clearing after every task sits clearly outside it. Never clearing is more expensive in the point estimate, but not demonstrably so.

No failures anywhere, but the denominator is not constant. Across all 36 runs, 4086 tests executed and 4086 passed. No run and no condition produced a single failure. That total is a sum over the whole experiment, not a per-condition figure, and the per-condition totals are not equal:

Tasks per session	1	2	3	4	6	12	all
Tests executed	701	697	687	699	661	641	4086
Tests passed	701	697	687	699	661	641	4086

Table 2: Tests executed and passed per condition, six runs each. Per run the count ranges from 97 to 127; the mean per run falls from 116.8 (one task per session) to 106.8 (never cleared).

Because the agent writes the tests as part of each task, the size of the suite is an outcome of the run rather than a fixed property of the harness. Zero failures everywhere is therefore a weaker statement than "quality was identical". It rules out broken interfaces and dropped tasks. It does not rule out that some conditions checked less than others. We say what the counters support and no more. Wall-clock time showed no consistent trend with session length either. Condition means span 482–543 s, while individual runs span 409–690 s, so replicate variance exceeds the between-condition spread and the cost differences are not latency differences in disguise.

The saving is not marginal. Clearing after every task costs about a third more than the optimum. That is larger than most prompt-level optimizations report for a lever that costs nothing to pull, and it compounds, because the same policy applies to every agent run a team makes.

4 Why the curve is U-shaped

The U is the product of two monotone effects that pull in opposite directions.

Longer sessions mean fewer model calls. Calls fall from 94.0 to 63.0 as session length grows. A fresh session has to rediscover the state of the working directory, so short sessions spend calls on orientation that long sessions do not.

Longer sessions mean more context per call. Context carried per call rises from 38,478 to 70,712 tokens. Nothing is ever dropped inside a session, so every task's transcript is paid for again on every subsequent call in that session. A token that entered the context on the fifteenth call of a hundred-call run is charged roughly eighty-five more times.

Cache reads are the product of these two terms, and a product of a falling and a rising factor is U-shaped: 3.62M tokens at every-task clearing, 3.16M at the optimum, 4.45M with no clearing at all.

Clearing is not free either. Cache writes fall monotonically with session length, from 252,533 to 87,846 tokens, because every fresh session must re-prime the cache. On the model we used, a 1-hour cache write costs 20× a cache read per token ($4.00 versus $0.20 per million). That ratio keeps aggressive clearing expensive even though it does reduce context per call: at every-task clearing, cache writes alone account for $1.01 of the $2.17 run.

So the two ends of the range are expensive for different reasons, which is precisely why the problem is hard to notice in practice. A team that clears constantly sees a small context and concludes it is being frugal, while it pays in cache writes. A team that never clears sees no re-priming and concludes it is being frugal, while it pays in re-read context. Neither sees the other's bill. At the corrected prices the cache-write side is the heavier of the two, which is why the left arm of the U is steep and the right arm is shallow.

At the optimum the spend splits roughly 39% cache read, 32% cache write and 29% output. No single term dominates, which is the signature of a balanced point rather than a corner solution.

5 Threats to validity

Resolution of the plateau and of the right arm. Six replicates are enough to separate clearing after every task from the middle, but not to rank conditions inside the middle. At the corrected prices they are also not enough to separate never clearing from the optimum. Distinguishing three from four or six tasks per session, or confirming the right arm of the U, would need more replicates or a task suite with a larger effect. We do not claim that three is better than four.

Post-selection and multiple comparisons. The reference condition (three tasks per session) was selected as the observed minimum of six sample means, so the comparisons against it are made after selection. The Holm adjustment over the five comparisons controls the family-wise error for those comparisons, but it does not account for how the reference itself was selected.

The verification suite is agent-authored. The agent under test writes the tests, to a stated minimum with no maximum. The suite therefore differs between runs in size (97 to 127 tests; on average 116.8 per run when clearing after every task and 106.8 when never clearing) and is not a fixed instrument. This limits the quality gate, not the cost measurement: the token counters that produce every cost in Table 1 do not depend on the suite. It does mean the gate cannot certify equal work across conditions, and a design with a fixed, externally authored suite would be strictly better.

One model, one suite. The location of the minimum almost certainly depends on the price ratio between cache writes and cache reads, and on how much per-session orientation a task suite demands. We claim the U-shape and the mechanism, not that three is a universal constant. The released harness is designed to let others place their own minimum.

Modeled cost and the price card. We compute cost from counters and published prices rather than from invoices. This is a deliberate trade: the numbers can be recomputed under any price list, but billing-side effects are not captured. This report is itself an example of the risk. Versions 1.0 and 1.1 applied the wrong price level and the wrong cache-write rate. Correcting them did not move the minimum, but the difference for never clearing lost significance. Conclusions about the ends of the curve are sensitive to the write/read price ratio, so the duration of cache writes must be read from the logs, not assumed.

Task independence. Our twelve tasks are largely independent. A suite with strong sequential dependencies would give longer sessions a larger advantage and shift the minimum to the right.

Agent nondeterminism. The agent chooses how many calls to spend per task, so model calls are an outcome rather than a controlled variable. This is why we report replicate spread and use permutation rather than parametric

tests.

6 Related work

Work on long-context models focuses on whether a model can use a long context; we ask what it costs to keep giving it one. Work on prompt caching reports cache hit rates and latency; we treat the cache as a priced resource and ask how session policy moves spend between its two sides. Agent benchmarks measure task success at fixed cost or cost at fixed success. The session-length lever is orthogonal to both and, unlike model choice or prompt engineering, is free to change.

7 Availability

Run-level counters for all 36 runs are released together with the measurement methodology: token counts by category, model calls, sessions, wall clock and test outcomes, with run identifiers hashed and no session content. The `modeled_cost_usd` field of `runs_ucurve.json` is kept as released and reflects the incorrect prices of versions 1.0 and 1.1. Corrected per-run costs, with cache writes split by duration, are in `runs_ucurve_costs_v1.2.json`, and the recomputation scripts are in `recompute-2026-09-15.zip`. A companion report decomposing where agent spend actually goes is archived at https://doi.org/10.5281/zenodo.22688704. The research index is kept at https://arsentev.ai/research.

8 Conclusion

Session length is a free lever, and the best setting is neither of the two settings people actually use. Clear the context, but not after every task. On our suite, anything from every third to every sixth task sits in the flat bottom, while clearing after each task costs 33.5% more. Never clearing cost 9.7% more in our runs, but six replicates cannot separate that difference from noise. More useful than any single number is the reason for the shape. Clearing trades cache writes against re-read context, the two move in opposite directions, and the price ratio between them decides where the balance falls. Any team can measure its own minimum in an afternoon. As this correction shows, it should price cache writes at the rate its logs actually record.