Research Square

Preprints are preliminary reports that have not undergone peer review.
They should not be considered conclusive, used to inform clinical practice,
or referenced by the media as validated information.

When Does Context Compaction Pay in Long-Horizon Agents?

Yuyao Wang

yuyaowang402@gmail.com

Independent

Research Article

Keywords: LLM agents, context compaction, long-horizon agents, reproducibility, prompt caching

Posted Date: October 27th, 2026

DOI: https://doi.org/10.21203/rs.3.rs-11487641/v1

License: This work is licensed under a Creative Commons Attribution 4.0 International License.
Read Full License

Additional Declarations: The authors declare potential competing interests as follows: Access to the GLM-5.3-Flash service used in this study was provided by Richinfo Technology Co., Ltd., the author's employer.

WHEN DOES CONTEXT COMPACTION PAY IN LONG-HORIZON AGENTS?

Yuyao Wang [1]

ABSTRACT

Context compaction replaces history with summaries, paying for their generation and changing an agent's subsequent work. We study when this exchange pays on nine long Terminal-Bench tasks, continuing agents from common saved states to termination under repeated compaction, and check the results with complete runs and GPU replays. Holding the realized request sequence of synchronous compaction (Sync) fixed, compaction would have saved gpt-6.1-sol USD 0.17 per state at 64K and was favored at 18 of 21 states; continued to termination, Sync instead cost USD 0.26 more with measured cache reads, more at 20 of the 21. It added requests, mostly read-only, at all 48 saved states; the excess persisted with a continue cue, with Codex's summary prompt, on four of five further tasks, and in the Codex CLI with its own compaction. In our runtime's complete runs, with observed requests and cache allocations fixed, Sol's Sync becomes cheaper only above a cache-read price of 34% of uncached input (16–87%; listed 5%); GLM's crosses at 0.5% (listed 20%). Price alone does not explain the split: at one common price vector, compaction paid only where much work remained after the switch. On one GPU, compaction made a fixed-work pair of sessions 19% faster by avoiding a full KV pool; in a separate replay of two whole continuations, summary generation and added requests made Sync take 1.8 times as long although it never queued. Compact early for cost only when enough work remains to repay summaries and changed work, preserve cached prefixes, and do not read lower KV occupancy as lower latency.

1 INTRODUCTION

Context compaction exchanges the cost of carrying history for the cost of summarizing it and any change in subsequent work. Agents working on terminal tasks for hours accumulate long histories of responses and tool output (Merrill et al., 2026); carrying them costs API tokens and serving memory, and more context does not reliably help the model (Hong et al., 2025). Runtimes replace older history with a model-written summary (Wu et al., 2025). This can keep an agent within its hard context window, but compacting earlier to save money or time requires a separate justification. The summary consumes generation time, can interrupt the agent, and changes the information behind its next action. Shorter requests create a potential saving; the resulting work determines whether the exchange pays.

Production systems use on-demand, keep-recent and background compaction, and rebuild the conversation around summaries in different ways (Anthropic, 2026a;b;c; OpenAI, 2026a; Google, 2026). Prior work shows that fewer tokens need not lower cost (Weinberger and Hozez, 2026), agents re-acquire dropped facts (Liu, 2026; Min et al., 2026), and masking and summarization differ over complete runs (Lindenbauer et al., 2025; Fan et al., 2026). Serving studies measure changes in outputs under KV compression (Gao et al., 2025), price fixed-prompt compression against caching (Song, 2026), or hide summary latency (Chen et al., 2026; Cim et al., 2026; Pan et al., 2026).

We present an empirical study organized around this exchange. A summary costing C and saving s at each later step repays after $R = C/s$ steps only while that work stays fixed. We compare this account with agents continued from common saved states to termination (Figure 2), check the result in independent complete runs, and replay request streams to expose serving bottlenecks. Thresholds of 32K–128K lie far below the models' 1M-token window, so compaction here is an optional saving, not a way to fit the window. Nine long Terminal-Bench tasks and complementary model checks support three connected contributions:

- **A savings budget tested against changes in work.** We decompose observed savings into a fixed-trace estimate, which holds Sync's realized requests fixed, and the cost difference between the two request sequences (Section 2). For Sol at 64K, the estimate favored Sync at 18 of 21 states; continued to termination with repeated compaction, Sync cost more at 20 with measured cache reads. GLM retained its savings; prompt reruns and the Codex CLI test whether Sol's added

[1]Independent. Correspondence to: Yuyao Wang <yuyaowang402@gmail.com>.

Preliminary work. Under review by the Machine Learning and Systems (MLSys) Conference. Do not distribute.

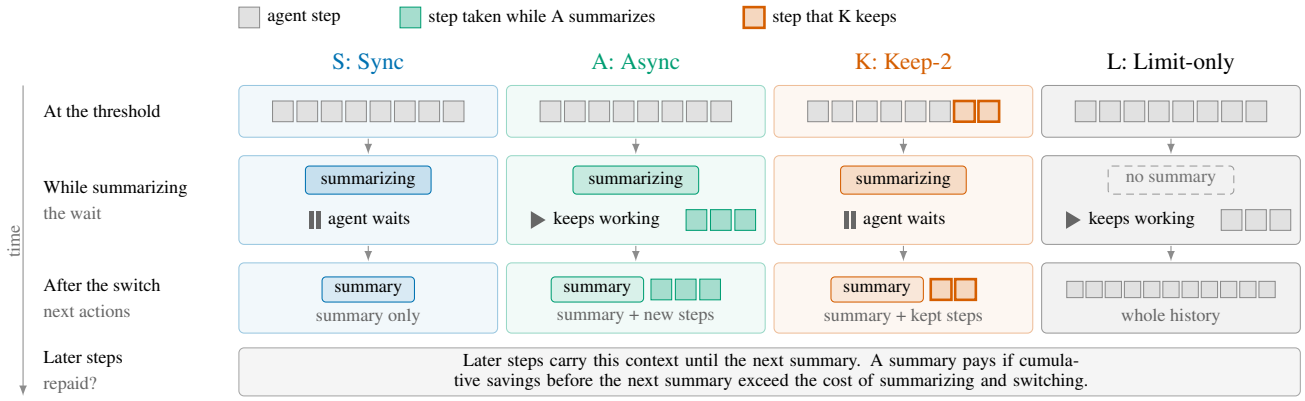


Figure 1. Four of the policies around one compaction, with time running down; numbers of steps are illustrative. When the context crosses the threshold, S and K wait while the summary is written, A keeps working, and L does not compact until its 1M-token limit. The first request after the switch then sees the summary alone (S), the summary followed by the steps taken meanwhile (A) or by the two steps before the threshold (K), or, for L, the whole history. The labels on the left mark what we measure: the wait and the repayment (Section 4), and the next actions and the work that follows (Section 5).

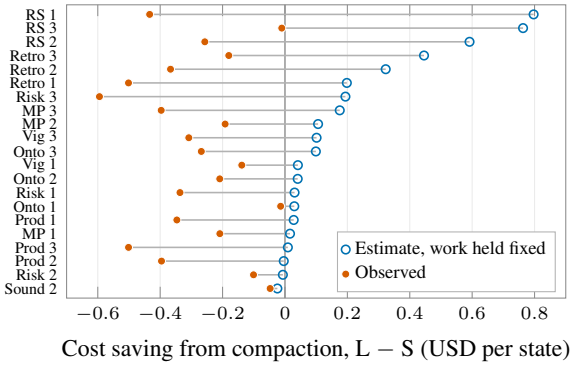


Figure 2. Cost saving of S over L at Sol's 21 states at 64K, by task and root, sorted by the estimate. Open: the fixed-trace estimate, S's realized requests priced with L's context. Filled: the observed saving from the same state, L's cost minus S's. Our cache accounting at Sol's listed prices, averaged over repeats; measured cache reads move the observed saving by at most $0.12 (Section 5). Sol at 32K and GLM: Figure 8.

work depends on one summary wording or on our runtime (Sections 5 and 6).

- **Price ranges for complete-run policy choices.** Freezing observed requests and cache allocation yields a break-even cache-read price. Sol's later complete runs cross at 34% for Sync and above 99% for Mask-10, against a listed 5%; GLM's Sync crosses at 0.5% (Section 6). At common prices, the models' split tracks the work left after the switch (Section 5).
- **Capacity relief separated from service demand.** Controlled GPU replays show both an 18.8% fixed-work speedup and, in a different workload, a 1.76-fold whole-continuation slowdown from summary generation and added requests despite eliminating queueing. A slot-limit intervention identifies when Async can overlap work at fixed KV capacity (Section 4.4).

Section 7 translates these results into runtime choices: retain history outside the measured cost-saving price range, preserve reusable prefixes, and budget summary service before relying on overlap.

2 Compaction and Its Return

Our agent is one model calling tools in a loop inside the task's Docker container. Tools execute shell commands and support background execution and retrieval of earlier outputs. A run ends when the model replies without calling a tool, and the task's verifier then grades the container. A *step* is one assistant response together with its tool results. The runtime keeps an append-only record of the interaction. Compaction replaces a prefix of this history with a summary written by the same model, through the same request path as the agent. The summary request carries the history being replaced and asks for a handoff in fixed sections: goals, technical background, files and code, errors and fixes, unfinished tasks, current work, next steps, and constraints (Appendix I). After every summary, the runtime adds a note that the summary is a lossy paraphrase to be checked against original records, with directions for rereading earlier tool output.

A summary has a one-time cost C: its summarization requests and the extra cost of the first request after the switch, whose new prefix is not yet cached. It saves s at each later step, the cost of carrying the removed history. Used for U

steps before the next summary replaces it or the run ends, it returns $Us - C$ if the later steps are the same as they would have been without it, and it repays once U exceeds $R = C/s$. We call $Us - C$ the *fixed-work estimate*.

Across a run, removed context stays removed. Let $\text{cost}_L(S)$ be the cost of a Sync run's agent requests sent with Limit-only's context: all history removed at earlier switches is added back and priced as cached input; $\text{cost}(S)$ and $\text{cost}(L)$ are the actual costs, summaries included. The observed saving is $G = \text{cost}(L) - \text{cost}(S)$. Define the *fixed-trace estimate* F and the work difference D:

$$F = \text{cost}_L(S) - \text{cost}(S),$$
$$D = \text{cost}_L(S) - \text{cost}(L),$$
$$G = F - D.$$

F is the net savings budget with Sync's realized requests fixed, already accounting for summary and switch costs; D compares the two request sequences at Limit-only's context and prices. Compaction saves when $D < F$; D can be negative if the work change is favorable. F is computed after the fact from Sync's requests, a diagnostic rather than an online trigger; forecasting is a separate problem (Section 4.3). At a flat cached-token price, $F = \sum_j (U_j^+ s_j - C_j)$, counting U_j^+ steps from summary j's first use to the end; tiered prices use request-by-request repricing (Appendix Q). The per-summary estimate counts a summary's savings until the next summary replaces it; the fixed-trace estimate counts the savings of all removed history through the end of the run. Both hold the later work fixed; same-state continuations and complete runs let it change.

Table 1 lists the policies and Figure 1 draws four of them around one compaction. The runtime switches only between requests, so a finished summary can wait until the next request begins; behavior measurements start at the first request that uses it.

Table 1. Policies: what happens when the context crosses the threshold, and what the agent sees after the switch. Letters label the policies in figures and tables.

Policy	At the threshold; after the switch
Sync (S)	waits for a summary of the history; sees the summary
Async (A)	keeps working while the summary is written; sees the summary, then the steps taken meanwhile
Keep-2 (K)	waits, keeping the last two complete steps out of the summary; sees the summary, then those two steps
Capped Keep-2 (KB)	as K, keeping steps only within 20% of the threshold
Mask (H)	in every request from the third tool batch on, masks tool output older than the last two tool batches (readable through a tool); summarizes only if the masked request is still above the threshold
Mask-10 (M10, M10+S)	as H, keeping ten tool batches; M10 summarizes only at the 1M-token limit, M10+S as H
Limit-only (L)	compacts only at the 1M-token limit; sees the whole history

3 Study Design

3.1 Tasks and thresholds

We use nine tasks from Terminal-Bench (Merrill et al., 2026), taken from the public task repository at commit `452bf30` (Table 2). From the public descriptions alone we picked 22 candidates that looked likely to need long, multi-step work and ran each once with GLM at a 1M-token context without compaction. The first five kept were those whose prompts grew past 128K tokens and whose environments fit our machine (Appendix F). Four more, chosen for their different patterns of growth, are held out for the switch-time forecast, frozen before their data were read.

The threshold is a working target set by the runtime: a policy acts when the runtime's estimate of the next request's input exceeds it. Every policy also has a hard limit of 1M tokens. We start at 128K, which each of the five main tasks passed without compaction, and halve it twice. At 32K the summary alone may use up to 12,288 tokens, more than a third of the threshold, so 32K is the practical lower end. Every run has an eight-hour limit.

3.2 Models and runs

GLM-5.3-Flash and gpt-6.1-sol (Sol, reasoning effort max) run the tasks; three other models provide checks.

- *Complete runs.* Sol runs every policy but Mask-10 on the nine tasks at 64K, three times each, and Mask also at 32K; a later batch runs Limit-only, Sync, and both Mask-10 policies three times each, and the Codex CLI runs Limit-only and two forms of Sync three times each. GLM runs Sync, Capped Keep-2, Mask, and Limit-only on all nine tasks at 64K, Keep-2 on seven, and Async on four; each is compared with Limit-only on the tasks both ran. GLM also starts Sync/Async pairs together at 32K, 64K, and 128K on all nine tasks.
- *Same-state continuations* start from Sol's first 32K and 64K crossings (48 states), GLM's first 64K crossing on each task, and nine later GLM states (Section 5); later runs continue one 64K state per task with other summary prompts, and the first 64K states of five further tasks. A saved state's source run is its *root*.
- *DeepSeek-Flash* checks repayment at a lower cache price in five Sync/Async pairs and five Limit-only runs.

Table 2. Tasks, and how long each ran under Limit-only, which does not compact below its 1M-token limit: median over three runs of agent time (minutes), completed requests, and the largest input of a request (thousands of tokens), per model. The lower four are held out.

			GLM-5.3-Flash			gpt-6.1-sol		
Label	Benchmark name	What the agent must do	Min.	Req.	Input	Min.	Req.	Input
MP	`mp-checkpoint-consolidation`	Rebuild one checkpoint from training shards	32	54	91	18	30	78
Vig	`interleaved-vigenere`	Build a breaker for an unknown cipher	240	254	345	16	19	44
RS	`rs-archive-clone`	Clone an archive tool by black-box probing	74	321	480	48	66	189
Coq	`coq-block-bound`	Prove a combinatorics theorem in Coq	110	554	525	16	23	36
Retro	`retro-console-soc`	Implement an 8-bit console SoC in Verilog	69	212	387	32	29	106
Risk	`risk-scorer-replay`	Make an evaluator match a black-box scorer	53	124	257	19	27	77
Sound	`sound-change-cascade`	Recover an ordered cascade of sound changes	78	66	546	11	12	49
Prod	`production-planning`	Write a plan into ERP, MES, warehouse DBs	24	24	103	16	21	77
Onto	`ontology-kg-querying`	Merge RDF data; write SPARQL queries	9	31	63	21	28	96

Table 3. Fixed prices used for all cost results, in USD per million tokens of uncached input, cached input, and output. r is the cached-input price as a share of the uncached price. DeepSeek's are its peak rates; its off-peak rates are half. Sol's cached price is its cache-read price; it also pays 2.50 for input written to the cache, and every price in the second Sol row applies to requests above 272K input tokens, with writes at 5.00.

Model	Uncached	Cached	Output	r
GLM-5.3-Flash	0.15	0.03	0.50	0.20
DeepSeek-Flash	0.30	0.006	1.20	0.02
gpt-6.1-sol	2.00	0.10	10.00	0.05
above 272K	4.00	0.20	15.00	0.05

- *Qwen3.8-27B and Qwen3-32B*, served with vLLM on one GPU, replay recorded GLM steps and two whole Sol continuations, alone and together, to measure server time and memory (Section 4.4).

Appendices A, K, W, O, O.3, and T list every run.

3.3 Outcomes and cost

Task quality is the official verifier reward. Cost, requests, and time run to termination under the common eight-hour budget, so they include unsuccessful and time-limited runs; requests count completed model requests, summaries included, and summary waits are measured separately. GLM and DeepSeek requests are priced from the provider-reported uncached input, cached input, and output at one fixed USD list price per model (Table 3), from the official pricing pages on 2026-10-06 (Z.AI, 2026; DeepSeek, 2026).

We price Sol's requests at the model's official Standard API prices, listed on 2026-10-08 (OpenAI, 2026d;e) (Table 3), with a cache accounting fixed before the runs (Appendix L): a request reads from the cache the longest complete input of an earlier request in the same run that it repeats as a prefix, if that request finished within the last 30 minutes, and writes the rest; a continuation starts with its root's cache at the saved state. The *break-even read price*, at which a policy's cost would equal Limit-only's, varies the read price with this allocation fixed (Appendix N).

Measured cache reads. To check this accounting, we replayed the 126 Sol continuations at 64K request by request through a ChatGPT subscription endpoint that reports cache reads, at the original spacing and after one request that repeated the root's last input under the same cache key; every replayed request had the input token count it had in the run (Appendix M). The provider read 92.9% of their input from the cache (Limit-only 96.3%, Sync 88.8%), against 93.9% under our accounting (96.8% and 90.4%). Replayed requests are priced at the listed prices of Table 3: the measured reads at the read price, the rest of their input at the uncached price, and their recorded output; so are the provider-reported cache reads of later runs.

Intervals and registration. Intervals are 95% bootstrap intervals from 10,000 draws with the tasks held fixed: same-state statistics resample roots within tasks, and complete-run contrasts resample repeats within tasks with the policies kept paired. Averages weight continuations, states, roots, and tasks equally at each level. GLM's same-state results, from one or two roots per task, are reported as counts. Before Sol's formal runs we registered six predictions (Appendix U): under 20% of Sync's summaries repay at 32K (Prediction 1), under half at 64K (2), over half without a cache (3); from the same state, Sync makes more requests than Limit-only (4) and saves less than its summaries' returns summed (5); fewer of Sync+2's first actions are read-only (6). The GLM forecast was registered before its held-out data were read; the fixed-trace estimate, the breakdown of added requests, and the break-even read prices were defined afterwards.

4 Sources and Limits of Savings

We first isolate savings from shorter requests: their price and lifetime conditions in API runs, and their dependence on service demand and capacity in GPU replays. Section 5 tests how changed agent work consumes or preserves the savings budget.

4.1 Prices determine the available budget

We split cost into agent requests and summarization requests, retries included (Appendix Y, Figure 11). With GLM, Limit-only paid almost entirely for cached input, the history carried in every request; compacting runs cost 0.15–0.26 USD per 100 steps, rising with the threshold for Sync and Async (Figure 3a). With DeepSeek, output made up most of the cost under every policy, and compaction shortens only the input (Figure 3b).

The cached-input price alone can move this balance. For a fixed set of requests with N_u uncached input, N_c cached input, and N_o output tokens, the cost is $C(r) = p_u N_u + r p_u N_c + p_o N_o$, where r is the cached price as a share of the uncached price p_u. With ΔN_u, ΔN_c, ΔN_o a compaction policy's tokens per agent step minus Limit-only's, its cost per step equals Limit-only's at $r^* = -(\Delta N_u + (p_o/p_u)\Delta N_o)/\Delta N_c$ and, when $\Delta N_c < 0$, is lower above it. These per-step comparisons omit changes in run length; Section 6 gives complete-run price ranges. Repricing each policy's pooled requests with only the cached price changed (Figure 3c), compaction became cheaper per step than Limit-only above 0.4–1.0% for GLM's runs and 3.0–3.3% for DeepSeek's. GLM's actual 20% lies far above both, and there Limit-only cost 3.0–5.9 times as much per step as compaction; DeepSeek's 2% lies below its break-even, and there Limit-only cost 0.8 times as much.

Waiting. Suppose a synchronous run takes foreground time T and spends W of it blocked on summaries. Hiding that wait saves at most W/T, provided the rest of the run stays the same. At 64K and above, every GLM and DeepSeek Sync run spent under 10% of T waiting (Figure 4a). GLM runs at 32K waited about 14%, and DeepSeek's Retro32, with 342 summary requests in 7.9 h, 43%.

4.2 Repaying a summary

We estimate C, s, R, and U for every summary of a Sync run from the recorded usage around its switch (Appendix G); U counts the agent steps in the summary's cycle. At GLM's prices a summary needed a median of 2–6 steps to repay, fewer at higher thresholds, and in 40 of 41 Sync runs most summaries repaid before the next one replaced them (Figure 5b); on the held-out tasks, 147 of 158 did. At DeepSeek's prices, carrying context was nearly free while the summary's output was not: repaying took a median of 29 steps at 64K against 10 steps of use, and in each run at 32K and 64K at most 4% of summaries repaid (Figure 5c). With Sol, where a cache read costs 4% of a cache write, none of the 1,256 Sync summaries at 32K repaid (Prediction 1), and at 64K the share that repaid was 0.49 (0.38–0.65), against a predicted share below one half (Prediction 2). Priced without a cache, more than 99.8% of the same summaries repaid (Prediction 3): with the summaries unchanged, the price of carrying context decided whether they repaid. With every model, lower thresholds made repayment harder in two ways: each summary removed less context and was replaced sooner.

4.3 The budget is not yet an online trigger

Forecasting U from the room left below the threshold and recent growth per step, fixed on the five main tasks, did not consistently beat a fixed prior lifetime or the previous summary's: once the summary was ready, its balanced accuracy was 0.59 against 0.50 and 0.58 on GLM's held-out tasks, and 0.70 against 0.81 and 0.80 on Sol (Appendix G). The eleven of 158 GLM summaries that did not repay lost 0.18% of the Sync cost. Thus even repayment with fixed work does not yet provide a reliable switch-time predictor.

4.4 On a self-hosted server

A hosted API bills tokens; a self-hosted server pays for context in time and in KV-cache memory. We distinguish service demand from capacity relief: fixed-work replays measure both with agent work held constant; whole continuations test their balance when the policies do different work. Recorded outputs isolate serving costs, not task quality on the replay model. We first replay recorded GLM switch points on two open models served with vLLM (Kwon et al., 2023) on one 96 GB GPU: Qwen3.8-27B (Qwen Team, 2026), which uses full attention in 16 of its 64 layers, and Qwen3-32B (Yang et al., 2025), which uses it in all 64. At each point, the next 20 agent steps of the GLM run are replayed under Limit-only, Sync, and Keep-2 with their original responses and tool outputs, so only the context differs. Decoding time is reconstructed from times measured at fixed lengths and checked against full runs (Appendix D).

Service demand with fixed work. Decoding time grew linearly with context, four times as fast on Qwen3-32B as on Qwen3.8 (Figure 5a). Within the 20 replayed steps, none of the 78 Sync and Keep-2 replays on Qwen3.8 finished below Limit-only's time, against 15 of 54 on Qwen3-32B; extrapolated linearly, few summaries would have repaid within the steps they were used in the GLM runs (Figure 5d,e).

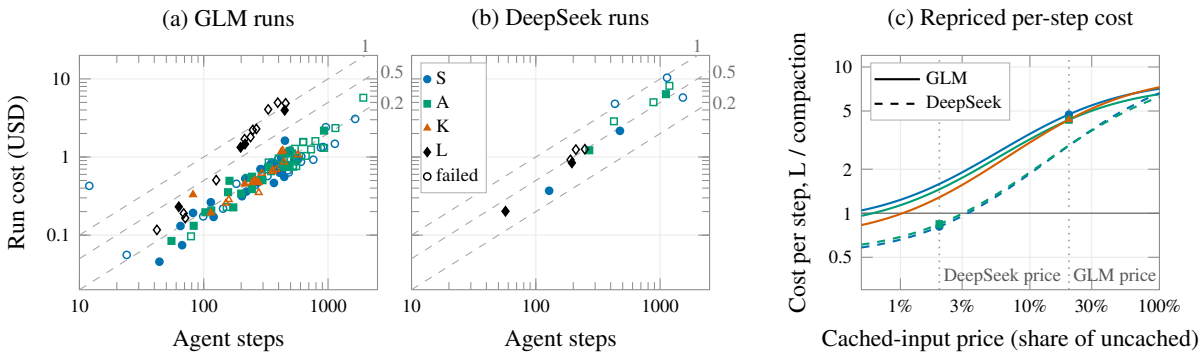


Figure 3. Cost at fixed list prices. (a,b) Total cost of each run against its agent steps, counting every agent and summarization request; open marks are failed runs, and dashed lines mark constant cost per 100 steps in USD. (c) Cost per step of L divided by that of S, A, or K after repricing the same requests with only the cached-input price changed, pooling each policy's runs and the L runs on the same tasks; above 1, compaction is cheaper per step. Marks show each provider's actual price.

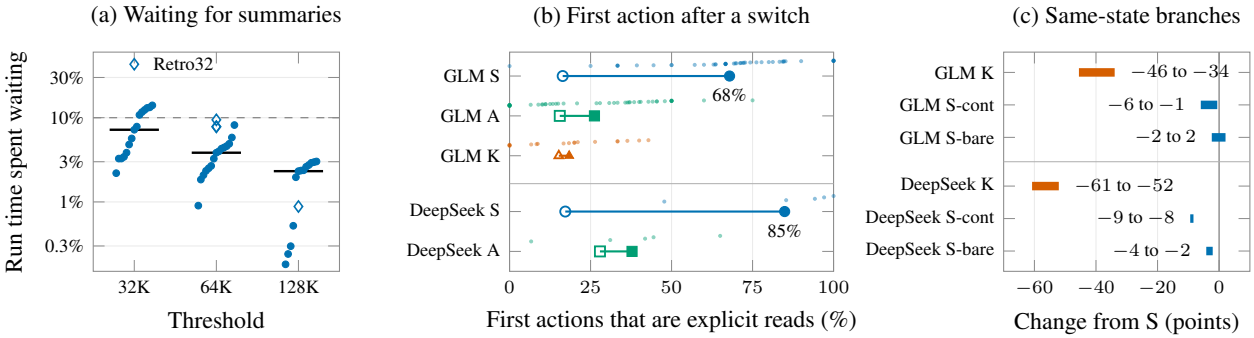


Figure 4. Waiting and the first action after a switch. (a) Share of each S run's foreground time spent waiting for ordinary summaries: dots are GLM runs, diamonds DeepSeek runs, bars GLM medians, and the dashed line marks 10%; one GLM S128 run ended before its first summary. (b) Share of first actions that were explicit read commands right after a switch (filled marks; small dots are single runs) and at a run's other steps (open marks, from the fourth step after each switch). (c) Same-state branches: change in the share of read-only first proposed actions relative to S when the agent keeps the last two steps (K), or when the two notices are removed (S-bare) or replaced with a continue instruction (S-cont); GLM branches come from the second repetition's S runs (Appendix B). Bars span the assignments of unclassified actions, not sampling intervals.

Capacity relief with fixed work. When the Vig and RS replays at 64K shared the GPU on Qwen3-32B, with the server set to run at most two sequences at once, the two growing Limit-only contexts filled the 86K-token KV-cache pool about 1,050 s in (Figure 6a), and from then on a request was waiting almost all the time until the RS session finished. The two Sync sessions peaked at 74% of the pool and never waited. The pair finished in 2,075 s with Limit-only and 1,685 s with Sync in both repetitions, 18.8% sooner; sharing the GPU lengthened each session by 22–24% under Limit-only and by 3% under Sync (Figure 6b), Limit-only's mostly after the pool filled. The benefit therefore includes avoiding a capacity boundary, beyond the per-token speedup of shorter context.

Admission capacity for background work. Async's summary, generated while the agent keeps working, needs a sequence slot of its own. With the 86K-token pool fixed and two repetitions per setting, two slots left Async's requests queued for 254–295 s, with no consistent advantage over Sync; four removed the queue, raised the KV peak from 74% to 78%, and let Async's pair finish 61–66 s before Sync's, while summed engine time stayed above Sync's; Sync and Limit-only changed by at most 0.3%.

Whole continuations: testing the net effect. To include the requests compaction adds, we replayed two whole Sol continuations at 64K on Qwen3.8, from Prod root 1 and Retro root 2, without tool gaps and with outputs forced to the recorded lengths. Alone, Sync's agent requests took 54 and 206 s less than the same requests with the removed history restored, but generating its one and three summaries took 252 and 555 s. Counting all its requests, 17 and 62 against Limit-only's 5 and 8, Sync took twice as long. Even

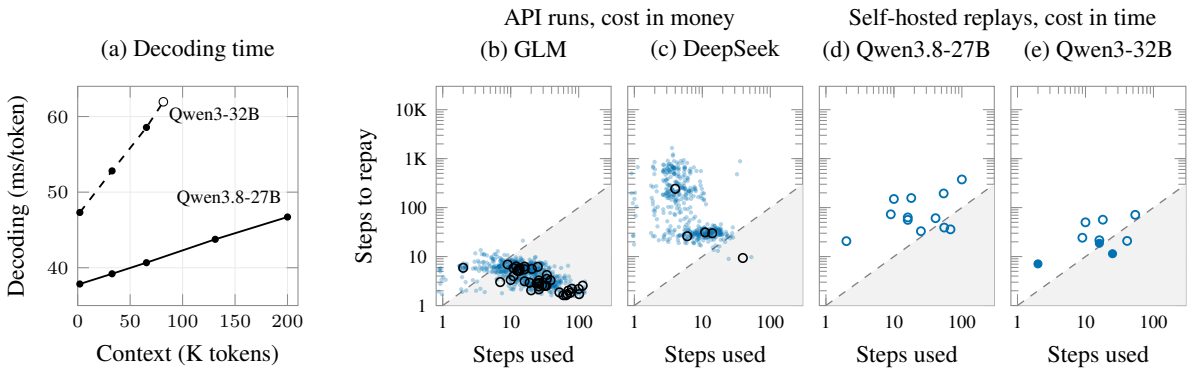


Figure 5. Repaying a summary. (a) Time to decode one output token against context length on one GPU; Qwen3.8-27B uses full attention in 16 of its 64 layers, Qwen3-32B in all 64; the open mark comes from a later session. (b–e) Estimated steps a summary must be used to repay its one-time cost, against the steps it was used before the next summary; in the shaded area, the estimate is below the steps used. (b,c) S runs' summaries (dots, jittered), priced as in Table 3; circles are run medians. (d,e) S replays of GLM switch points; the vertical position extrapolates the 20 replayed steps linearly, and filled marks are replays whose reconstructed time fell below L's within them. Steps used come from the GLM run; summaries cut short by the end of a run are left out here but stay in the costs of Figures 11 and 3.

Table 4. Whole Sol continuations replayed together on Qwen3.8-27B, mean of three pairs per cell. Pair: seconds until both sessions finish. Queue sums request waiting time, not delay of the pair's end; KV peak is the largest sampled pool use, and Cached is the share of prompt tokens served from the prefix cache. L sends 13 requests per pair; S sends 79, including four summaries.

Pool		Pair	Preempt.	Queue (s)	KV peak	Cached
128K	L	1,399	4	246	100%	65–72%
	S	2,457	0	0	89–90%	90%
180K	L	1,231	0	0	74–75%	93%
	S	2,454	0	0	63%	90%

with agent work fixed, summary generation exceeded the service time it saved; Sync's continuation also contained more requests. We then ran the two continuations together, three times per policy in each of two pools: 180K tokens and 128K, below the 154K needed by the two largest Limit-only requests together. In the smaller pool, Limit-only was preempted and queued, whereas Sync never waited but still took 1.76 times as long (Table 4). Capacity relief did not offset summary generation and additional work.

Capacity relief and lower service demand therefore require separate checks (Section 7). In a production SGLang deployment of GLM-5.3-Flash, over 542 hours that included our own runs (13% of its prompt tokens), active KV averaged 3.7% of the 8.3M-token pool (p95 16%, largest sample 90%), while reclaimable prefix cache filled the memory and served 95.6% of prompt tokens: the fixed-work pair's capacity boundary did not appear in these samples.

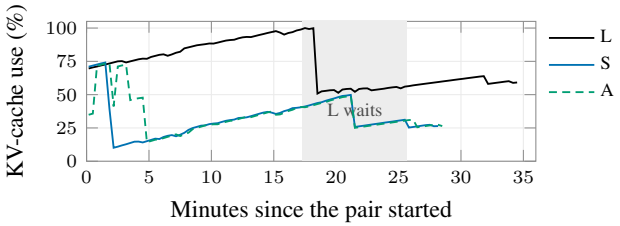
5 When Work Exhausts the Saving

A fixed-trace saving is a budget that changed work can exhaust. We test this from common saved states. Independent runs also differ before compaction: in Sol's complete runs at 64K, three repeats of the same task under Limit-only or Sync differed in cost by a median of $0.16, comparable to Sync's mean $0.17 excess (Section 6).

Same-state continuations. Each of Sol's 27 roots is a Limit-only run. When its context first crosses 32K, and again at 64K, we save the state: the agent's record up to the request that crossed, a copy of the container's file system, and the run's remaining time; processes and clocks are not restored, but at every state all recorded commands had finished (Appendix K). We continue each state to termination three times under Limit-only, Sync, and **Sync+2**, twice under Keep-2 and Async, and twice under Capped Keep-2 at 32K. Sync+2 (S+2 in tables) receives Sync's summary followed by the last two complete steps, at the first switch only, so Sync+2 minus Sync isolates what the two kept steps change. All 27 roots crossed 32K; 21 roots on eight tasks crossed 64K, and the other six, including all three Coq roots, finished first. Every continuation ran until the agent finished its work. Sync compacts again at every later crossing, so Sync minus Limit-only compares continuing with repeated compaction and without it.

Requests consume the saving. From the same state, Sync made more requests than Limit-only at every one of the 48 states: 22 more at 64K, 130 at 32K, and 75 over both (95% interval 61–90; Prediction 4), 66 of them agent requests and 9 summaries. Over both thresholds, Sync cost $1.16 more per state than Limit-only: priced at Limit-only's average cost per request, its extra requests add $2.74,

(a) Server memory, first repetition



(b) Each session, alone (open) and sharing the GPU (filled)

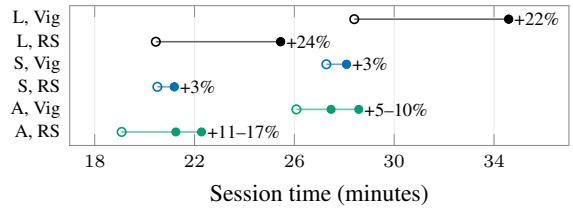


Figure 6. Two replayed agent sessions (Vig and RS at 64K) sharing one GPU on Qwen3-32B with two sequence slots, both under L, S, or A. (a) The server's KV-cache use while both sessions of a pair ran (largest value in each 20 s); under L, the two growing contexts filled the pool, and a request then waited until the RS session finished (shaded). (b) Time to complete each session's 20 steps alone and in two repetitions of the pair; labels give the slowdown from sharing.

and its cheaper requests take back $1.58.

Fixed-work and observed savings diverge. The fixed-trace estimate of Section 2 gives the saving Sync would have made had Limit-only done Sync's own requests (Appendix Q). At 64K, it favored Sync at 18 of the 21 states, by $0.17 per state (0.14–0.21). In fact Sync cost $0.26 more than Limit-only (0.21–0.33) and was cheaper at none of the 21 states (Figure 2). At 32K, of the 26 states where Limit-only could have carried Sync's work below its context limit, the estimate favored Sync at 13 and Sync was cheaper at three. The observed saving fell below the estimate at all 21 states at 64K and at 23 of the 26 at 32K. As registered in Prediction 5, the same-state difference was also less favorable to Sync than the sum of each summary's return over its own cycle, by $0.76 (0.53–1.05).

Measured cache reads. With cache reads measured by replay (Section 3), Sync cost $0.26 more than Limit-only per state (0.20–0.31) and more at 20 of the 21 states, against 21 under our accounting; per state the two differ by at most $0.12, and leaving out any one task keeps the mean between $0.24 and $0.29 (Appendix M).

What the added requests do. We label every tool call with a rule-based parser of shell commands, frozen before the runs; a request counts as read-only if all its calls only read existing state, which says nothing about whether the read was needed. Of Sync's 66 added agent requests per state, 48 (38–58) were read-only, and 28 read only files the run had already read or written, changed or not; at 64K, 11 of 20 were read-only and 3 were such rereads (Appendix R). Of 80 sampled read-only requests of both policies, three model judges without policy labels found 22 that retrieved text identical to an earlier recorded fragment and 25 that returned changed content or checked a result; 33 stayed undecided (Appendix R.1). At MP root 2, the 64K state with the median request difference, Sync spent ten requests after its first summary re-reading the directory, framework sources, and its own scripts, which Limit-only still held, and finished after 26 against 12, at $0.59 against $0.47.

Extra work persists across prompts. From eight preselected Sol 64K states, three continuations per policy with the notices replaced by a continue cue, and with Codex's compaction prompt (OpenAI, 2026b), left each Sync variant with 13–14 more requests than Limit-only and higher mean cost, with the same pass outcomes (Table 29; Appendix T.1). Codex's prompt shortened summaries by 29% without lowering mean cost. On this later runtime the excess was smaller than the 19 requests and $0.26 of the original continuations from these states; one Sync response lost its token counts, so its cost is a lower bound.

The effect on further tasks. On the same runtime, from the first 64K states of nine Limit-only roots on five further tasks, Sync made 16 more requests per continuation than Limit-only, more on four of the five tasks, with the same outcomes; it cost more on three of the four tasks with complete token counts (Appendix T.2).

Retention changes work but consumes context. Keeping Sync's last two steps lowered read-only first actions by 29–41 percentage points in GLM branches and by 0.25 in Sol's continuations (Prediction 6; Appendix B). Kept at every switch, Keep-2 reduced Sol's excess at 32K to 46 requests and $0.98 against Limit-only, less than half Sync's excess. Capping retention lowered the share of contexts above half the threshold from 22% to 3% (Appendix S). Retention can limit the penalty without eliminating it. Continuation success differences from Limit-only range from -0.07 to -0.04 at 32K and -0.05 to $+0.02$ at 64K.

GLM's saving survives changed work. GLM repeats Sol's 64K design from nine new Sync runs, one per task, saved at their first 64K crossing and continued under Limit-only, Sync, and Sync+2 three times each (Appendix W); 11 of 78 continuations ran out of the source run's eight-hour budget and count as failures. At GLM's prices, Sync cost less than Limit-only at eight of the nine states, $1.20 less on average and $0.61–1.35 with any one task left out

Table 5. Sol at 64K on the later runtime, 27 runs per policy. Pass: runs with verifier reward 1. Cost (USD): reported cache reads at Sol's listed prices; M10+S's is a lower bound (four requests lost their token counts). Break-even: the cache-read price, as a share of the uncached input price, above which the policy would cost less than Limit-only. Time (s), completed requests, and summaries (Summ.): means per run. Cached: cache-read share of input.

Policy	Pass	Cost	Break-even	Time	Req.	Summ.	Cached
L	22	0.77	–	1148	30.7	0	93%
S	20	0.94	34%	1319	45.7	1.56	88%
M10	18	3.12	>99%	1205	38.3	0	13%
M10+S	19	≥2.55	–	1365	44.1	0.59	26%

(Appendix W), and at any cache-read price above 0.4%; at DeepSeek's prices, at six states, $0.19 less. The work still changed: Sync made more read-only requests at eight states, 25 more per state, and more agent requests at six, up to 110 more; it made fewer at Coq and RS, where Limit-only's own continuations ran long. Of Sync's 142 summaries, 139 repaid by the per-summary account, and the fixed-trace estimate favored Sync at all nine states. Sync and Limit-only each passed 12 of 27 continuations, on different tasks. Sync took 23 minutes longer per state, mostly on MP, where all three Sync continuations ran out of time.

Remaining work tracks the split. Repriced at one common vector, USD 1, 0.05 and 5 per million uncached, cached and output tokens (Appendix X), the first 64K continuations of the eight tasks both models reached still diverged: Sync saved $1.69 per task with GLM, at five tasks, and lost $0.11 with Sol, at all eight. After the saved state, Limit-only made a median of 70 more requests with GLM and 7 with Sol. Sync's cost relative to Limit-only's fell as more work remained (Spearman -0.94 over 16 model–task pairs; Figure 9): Sync cost more at all eleven model–task pairs whose Limit-only continuations ended within 50 requests, where added work exceeded the fixed-trace saving, and less at all five GLM tasks with more. The work left is known only after the run (Section 4.3), and at later GLM states more added work can reverse it (Appendix W).

6 Price Ranges for Complete Policies

Independent complete runs of each policy test whether the same-state cost reversal persists from the task's start and locate its price boundary. Table 5 gives 108 Sol runs at 64K on the later runtime, with reported cache reads: nine tasks, three runs each of Limit-only, Sync, and the two Mask-10 policies, which keep ten tool batches as earlier work kept ten turns (Lindenbauer et al., 2025).

Table 6. GLM complete runs at 64K: each policy minus Limit-only per run, within the same task blocks, on the tasks where both ran; positive: the policy cost more. Per Limit-only run, GLM cost $2.38 at its prices with 244 requests and passed 6 of 27.

		ΔCost (USD), prices			
Policy	Tasks	GLM	DeepSeek	ΔRequests	ΔSuccess
S	9	−1.99	−0.29	−24.7	+0.15
A	4	−0.24	−0.03	+10.8	0.00
K	7	−1.22	+0.11	+99.3	+0.05
KB	9	−1.91	−0.18	+6.5	+0.04
H	9	−1.51	+0.42	+204.1	+0.04

Success. Seven of the nine tasks had the same outcome under every policy, in all three runs; the policies differ only on Vig and Prod. Sync's success differed from Limit-only's by -0.07 (-0.15 to 0); both Mask-10 policies passed fewer runs (-0.15, -0.22 to -0.11; -0.11, -0.19 to -0.04).

Sync's price boundary. Sync cost $0.17 more per run than Limit-only (0.10–0.24), more on seven of nine tasks, and took 171 s longer. It made 15 more requests, 13.4 of them agent requests. It sent 15% less input than Limit-only but read less of it from cache. With these request sequences and cache allocations fixed, Sync would cost less only above a read price of 34% of uncached input (16–87%; Appendix N), against Sol's listed 5%; priced with no cache, it would cost $0.39 less. It depends on the task mix: without MP it is 25%, without RS 122%. An earlier 162-run batch under our cache accounting agreed (Appendix O.2).

The Codex CLI. In the open-source Codex CLI (OpenAI, 2026c), with its own tools, prompts, and compaction, Sol ran each task three times per policy (Appendix O.3). Its default server-side compaction, set to trigger at 64K, cost $0.14 more per run than Limit-only (0.02–0.26), more on seven tasks, with 14 more requests, and would cost less only above a read price of 18%; our runtime gave $0.17, seven tasks, 15 requests, and 34%. Its local compaction cost $0.51 more, $0.37 more than server-side compaction. They passed 20 and 21 of 27 runs, Limit-only 21.

Cache reuse sets Mask's price range. Mask-10 cost $2.36 more per run than Limit-only (1.97–2.87), more on all nine tasks. It sent about as much input as Limit-only but read only 13% of it from cache; priced with no cache, both would cost $3.49 per run. Mask-10+summary summarized in only 10 of 27 runs (Appendix O.1).

GLM's complete runs favor compaction. At GLM's prices the direction reverses (Table 6): every summarizing policy cost less than Limit-only, Sync by $1.99 (0.99–3.10), because Limit-only carried its long context into every request on the long tasks; Sync would have cost more only be-

low a read price of 0.5% (Appendix N). Sync passed 10 of 27 runs and Limit-only 6 (difference 0.15, -0.04 to 0.33).

7 Guidance for Runtimes

Choose earlier compaction by net savings and outcomes; outcomes rarely differed, but Mask-10 passed fewer runs.

Charge changed work against the savings budget. F already deducts summary and switching costs, so compaction saves only if the work difference D stays below it; matched continuations estimate D, request counts alone do not. At flat list prices, Sol's saving per state would have paid for about five of Limit-only's requests, and Sync made 20 more; GLM's would have paid for about 140, and Sync made three fewer (Appendix X.1). The work left tracked the cost difference (Section 5), but three features known at the crossing did not beat one arm per model and price: on held-out tasks it came within \$0.002 per state of the cheaper observed arm at list prices, \$0.045 at others.

Keep early compaction off outside its cost-saving price range. With request sequences and cache allocations frozen, the cost difference $\Delta A + r\Delta B$ in the read price r identifies the range (Appendix N), which for Sol's Sync lies far above the listed price. Preserve reusable prefixes: Mask-10 sends as much input as Limit-only but reads 13% of it from cache, not 93%, at four times the cost.

Budget service demand before relying on capacity relief or overlap. Avoiding a full KV pool sped up fixed work, but summary generation and added requests slowed whole continuations that never queued (Section 4.4). Hiding summary waits alone saves under 10% in the GLM and DeepSeek runs at 64K and above if later work stays fixed. Async also needs sequence slots and KV headroom; its overlap hides latency without removing service demand.

8 Related Work

Context management and cost. MemGPT, ReSum, and self-compacting agents manage a limited model context explicitly (Packer et al., 2023; Wu et al., 2025; Li et al., 2026); open-source harnesses size the kept history in tokens (Earendil Works, 2026; DeepSeek-AI, 2026; Nous Research, 2026); and sleep-time compute prepares context ahead of time (Lin et al., 2025). Serving engines reuse computation across requests (Zheng et al., 2024; Kwon et al., 2023; Gim et al., 2024; Li et al., 2025), prompt caching cuts agents' API cost (Lumer et al., 2026), agent evaluations vary across repeats (Bjarnason et al., 2026; Mustahsan et al., 2025), and agents should be judged on cost and performance together (Kapoor et al., 2025).

Closest studies. In Claude Code, tool-output reduction can raise complete-task cost through cache traffic and added retrieval and testing (Weinberger and Hozez, 2026); we separate history compaction's fixed-trace saving from changed work. TRACE continues branches from one compaction boundary for up to five actions and scores complete tasks, capped at 50 steps, by success and steps (Min et al., 2026); we continue matched states to termination under repeated compaction and price both sides. Liu measures re-querying over 24-turn episodes with dropped or digested state and oracle restoration (Liu, 2026); we price model-written summaries. Masking and rule-based elision can match summarization at lower cost over complete runs (Lindenbauer et al., 2025; Fan et al., 2026); with cache hits discounted, our Mask-10 cost over three times as much as Sync (Section 6). CAPC prices static-prefix compression against caching (Song, 2026); we price histories whose later work changes with the policy (Appendix Z).

9 Limitations

All tasks come from one benchmark, with two or three runs per setting; Sync's difference in success from Limit-only remains unresolved. Sol's earlier complete runs and 32K continuations lack measured cache reads. Saved states are nested within their source runs (Sol's 48 within 27, GLM's 18 within 12). The fixed-trace estimate assumes that history added back stays cached; price ranges hold each policy's observed work and cache allocation fixed. Action labels come from a frozen parser and model annotators that may share errors. Self-hosted results use one GPU.

10 Conclusion

For Sol at 64K, a fixed-trace saving of \$0.17 per state became a \$0.26 increase: cheaper context did not fund the changed work. Complete runs favor Sync only above a cache-read ratio of 34% for Sol in our runtime and 0.5% for GLM (listed 5% and 20%); at one common price vector, compaction paid only where much work remained after the switch. Without prefix reuse, Mask-10 cost four times as much as Limit-only. On one GPU, avoiding a full KV pool sped up fixed work by 19%; summaries and added requests made whole continuations take 1.8 times as long.

Code and Data Availability

The artifact (`https://huggingface.co/datasets/context-cost-study/context-compaction-artifact`) provides the runtime with usage instructions, offline analyses with frozen inputs, and redacted trajectories (Appendix J). Code is under Apache-2.0 (historical V1 scripts: MIT), new first-party data under CC BY 4.0, and other components under their own licenses.

ACKNOWLEDGMENTS

Access to the GLM-5.3-Flash service used in this study was provided by Richinfo Technology Co., Ltd., the author's employer.

REFERENCES

Mike A. Merrill, Alexander G. Shaw, Nicholas Carlini, Boxuan Li, Harsh Raj, Ivan Bercovich, Lin Shi, Jeong Yeon Shin, Thomas Walshe, E. Kelly Buchanan, Junhong Shen, Guanghao Ye, Haowei Lin, Jason Poulos, Maoyu Wang, Marianna Nezhurina, Jenia Jitsev, Di Lu, Orfeas Menis Mastromichalakis, Zhiwei Xu, Zizhao Chen, Yue Liu, Robert Zhang, Leon Liangyu Chen, Anurag Kashyap, Jan-Lucas Uslu, Jeffrey Li, Jianbo Wu, Minghao Yan, Song Bian, Vedang Sharma, Ke Sun, Steven Dillmann, Akshay Anand, Andrew Lanpouthakoun, Bardia Koopah, Changran Hu, Etash Guha, Gabriel H. S. Dreiman, Jiacheng Zhu, Karl Krauth, Li Zhong, Niklas Muennighoff, Robert Amanfu, Shangyin Tan, Shreyas Pimpalgaonkar, Tushar Aggarwal, Xiangning Lin, Xin Lan, Xuandong Zhao, Yiqing Liang, Yuanli Wang, Zilong Wang, Changzhi Zhou, David Heineman, Hange Liu, Harsh Trivedi, John Yang, Junhong Lin, Manish Shetty, Michael Yang, Nabil Omi, Negin Raoof, Shanda Li, Terry Yue Zhuo, Wuwei Lin, Yiwei Dai, Yuxin Wang, Wenhao Chai, Shang Zhou, Dariush Wahdany, Ziyu She, Jiaming Hu, Zhikang Dong, Yuxuan Zhu, Sasha Cui, Ahson Saiyed, Arinbjörn Kolbeinsson, Jesse Hu, Christopher Michael Rytting, Ryan Marten, Yixin Wang, Alex Dimakis, Andy Konwinski, Ludwig Schmidt. Terminal-Bench: Benchmarking Agents on Hard, Realistic Tasks in Command Line Interfaces. 2026. https://arxiv.org/abs/2601.11868.

Kelly Hong, Anton Troynikov, Jeff Huber. Context Rot: How Increasing Input Tokens Impacts LLM Performance. *Chroma*, 2025. https://www.trychroma.com/research/context-rot.

Xixi Wu, Kuan Li, Yida Zhao, Liwen Zhang, Litu Ou, Huifeng Yin, Zhongwang Zhang, Xinmiao Yu, Dingchu Zhang, Yong Jiang, Pengjun Xie, Fei Huang, Minhao Cheng, Shuai Wang, Hong Cheng, Jingren Zhou. ReSum: Unlocking Long-Horizon Search Intelligence via Context Summarization. 2025. https://arxiv.org/abs/2509.13313.

Zhuofu Chen, Rui Pan, Yinwei Dai, Ravi Netravali. Slipstream: Trajectory-Grounded Compaction Validation for Long-Horizon Agents. 2026. https://arxiv.org/abs/2605.08580.

Musa Cim, Burak Topcu, Chita Das, Mahmut Taylan Kandemir. Parallel Context Compaction for Long-Horizon LLM Agent Serving. 2026. https://arxiv.org/abs/2605.23296.

Zaifeng Pan, Qianxu Wang, Zhengding Hu, Chang Chen, Yue Guan, Yanbo Zhou, Steven Swanson, Yufei Ding. SmoothAgent: Efficient Long-Horizon LLM-Based Agent Serving with Lookahead Context Engineering. 2026. https://arxiv.org/abs/2607.00151.

Charles Packer, Sarah Wooders, Kevin Lin, Vivian Fang, Shishir G. Patil, Ion Stoica, Joseph E. Gonzalez. MemGPT: Towards LLMs as Operating Systems. 2023. https://arxiv.org/abs/2310.08560.

Tianjian Li, Jingyu Zhang, William Jurayj, Xi Wang, Chuanyang Jin, Mehrdad Farajtabar, Eric Nalisnick, Daniel Khashabi. Self-Compacting Language Model Agents. 2026. https://arxiv.org/abs/2606.23525.

Tobias Lindenbauer, Igor Slinko, Ludwig Felder, Egor Bogomolov, Yaroslav Zharov. The Complexity Trap: Simple Observation Masking Is as Efficient as LLM Summarization for Agent Context Management. *NeurIPS 2025 Workshop on Deep Learning for Code (DL4C)*, 2025. arXiv:2508.21433.

Run-Ze Fan, Zihao Zhang, Simin Ma, Yebowen Hu, Shouju Wang, Kaiqiang Song, Fei Liu, Hamed Zamani, Xiaoyang Wang. An Empirical Study of Harness Design for Coding Agents. 2026. https://arxiv.org/abs/2609.20804.

Kevin Lin, Charlie Snell, Yu Wang, Charles Packer, Sarah Wooders, Ion Stoica, Joseph E. Gonzalez. Sleep-time Compute: Beyond Inference Scaling at Test-time. 2025. https://arxiv.org/abs/2504.13171.

Shuyu Liu. What Does Context Compression Cost an Agent? Interaction Costs Unrevealed by Task-Completion Metrics. 2026. https://arxiv.org/abs/2608.16370.

Wei Gao, Xinyu Zhou, Peng Sun, Tianwei Zhang, Yonggang Wen. Rethinking Key-Value Cache Compression Techniques for Large Language Model Serving. In *Proceedings of Machine Learning and Systems (MLSys)*, 2025.

Guanghui Min, Liang Wu, Mayank Darbari, Chen Chen, Liangjie Hong. Toward Reliable Context Compression for Long-Horizon Agents: An Empirical Study of Execution Instability. 2026. https://arxiv.org/abs/2608.06503.

Lianmin Zheng, Liangsheng Yin, Zhiqiang Xie, Chuyue Sun, Jeff Huang, Cody Hao Yu, Shiyi Cao, Christos Kozyrakis, Ion Stoica, Joseph E. Gonzalez, Clark Barrett, Ying Sheng. SGLang: Efficient Execution of Structured Language Model Programs. *Advances in Neural Information Processing Systems (NeurIPS)*, 2024.

In Gim, Guojun Chen, Seung-seob Lee, Nikhil Sarda, Anurag Khandelwal, Lin Zhong. Prompt Cache: Modular Attention Reuse for Low-Latency Inference. *Proceedings of Machine Learning and Systems (MLSys)*, 2024.

Woosuk Kwon, Zhuohan Li, Siyuan Zhuang, Ying Sheng, Lianmin Zheng, Cody Hao Yu, Joseph E. Gonzalez, Hao Zhang, Ion Stoica. Efficient Memory Management for Large Language Model Serving with PagedAttention. *Proceedings of the 29th Symposium on Operating Systems Principles (SOSP)*, 2023.

An Yang, Anfeng Li, Baosong Yang, Beichen Zhang, Binyuan Hui, Bo Zheng, Bowen Yu, Chang Gao, Chengen Huang, Chenxu Lv, Chujie Zheng, Dayiheng Liu, Fan Zhou, Fei Huang, Feng Hu, Hao Ge, Haoran Wei, Huan Lin, Jialong Tang, Jian Yang, Jianhong Tu, Jianwei Zhang, Jianxin Yang, Jiaxi Yang, Jing Zhou, Jingren Zhou, Junyang Lin, Kai Dang, Keqin Bao, Kexin Yang, Le Yu, Lianghao Deng, Mei Li, Mingfeng Xue, Mingze Li, Pei Zhang, Peng Wang, Qin Zhu, Rui Men, Ruize Gao, Shixuan Liu, Shuang Luo, Tianhao Li, Tianyi Tang, Wenbiao Yin, Xingzhang Ren, Xinyu Wang, Xinyu Zhang, Xuancheng Ren, Yang Fan, Yang Su, Yichang Zhang, Yinger Zhang, Yu Wan, Yuqiong Liu, Zekun Wang, Zeyu Cui, Zhenru Zhang, Zhipeng Zhou, Zihan Qiu. Qwen3 Technical Report. 2025. https://arxiv.org/abs/2505.09388.

Qwen Team. Qwen3.8-27B. *Hugging Face model repository*, revision `1d4bf0f`, accessed October 4, 2026. https://huggingface.co/Qwen/Qwen3.8-27B.

Hanchen Li, Runyuan He, Qiuyang Mang, Qizheng Zhang, Huanzhi Mao, Xiaokun Chen, Hangrui Zhou, Huanchen Zhang, Alvin Cheung, Joseph Gonzalez, Ion Stoica. Continuum: Efficient and Robust Multi-Turn LLM Agent Scheduling with KV Cache Time-to-Live. 2025. https://arxiv.org/abs/2511.02230.

Elias Lumer, Faheem Nizar, Akshaya Jangiti, Kevin Frank, Anmol Gulati, Mandar Phadate, Vamse Kumar Subbiah. Don't Break the Cache: An Evaluation of Prompt Caching for Long-Horizon Agentic Tasks. 2026. https://arxiv.org/abs/2601.06007.

Yan Song. Cache-Aware Prompt Compression: A Two-Tier Cost Model for LLM API Caching. 2026. https://arxiv.org/abs/2607.15516.

Sarel Weinberger, Amir Hozez. Token Reduction Is Not Cost Reduction: An Empirical Study of End-to-End Efficiency in API-Based Coding Agents. 2026. https://arxiv.org/abs/2607.12161.

Sayash Kapoor, Benedikt Stroebl, Zachary S. Siegel, Nitya Nadgir, Arvind Narayanan. AI Agents That Matter. *Transactions on Machine Learning Research*, 2025. arXiv:2407.01502.

Bjarni Haukur Bjarnason, André Silva, Martin Monperrus. On Randomness in Agentic Evals. 2026. https://arxiv.org/abs/2602.07150.

Zairah Mustahsan, Abel Lim, Megna Anand, Saahil Jain, Bryan McCann. Stochasticity in Agentic Evaluations: Quantifying Inconsistency with Intraclass Correlation. 2025. https://arxiv.org/abs/2512.06710.

Anthropic. On-demand compaction. *Claude API documentation*, accessed October 3, 2026. https://platform.claude.com/docs/en/build-with-claude/compaction-on-demand.

Anthropic. Keep recent turns. *Claude API documentation*, accessed October 3, 2026. https://platform.claude.com/docs/en/build-with-claude/compaction-keep-recent-turns.

Anthropic. Background compaction. *Claude API documentation*, accessed October 3, 2026. https://platform.claude.com/docs/en/build-with-claude/compaction-background.

OpenAI. Codex: local compaction summary prefix. *Source repository*, commit `ba2b67f`, retrieved October 16, 2026. https://github.com/openai/codex/blob/ba2b67f9cda954bcdda43c2a65ac58e807b996bd/codex-rs/prompts/templates/compact/summary_prefix.md.

OpenAI. Codex: local compaction prompt. *Source repository*, commit `ba2b67f`, retrieved October 16, 2026. https://github.com/openai/codex/blob/ba2b67f9cda954bcdda43c2a65ac58e807b996bd/codex-rs/prompts/templates/compact/prompt.md.

OpenAI. Codex CLI, release 0.163.0. *Source repository*, tag `rust-v0.163.0`, commit `8308618`, retrieved October 17, 2026. https://github.com/openai/codex/tree/rust-v0.163.0.

Google. Gemini CLI: chat compression service. *Source repository*, main branch inspected October 3, 2026. https://github.com/google-gemini/gemini-cli/blob/main/packages/core/src/context/chatCompressionService.ts.

Earendil Works. pi coding agent: compaction. *Source repository*, commit b1efcf7, August 14, 2026. https://github.com/earendil-works/pi/blob/b1efcf7d7c5d7394fbb12ede0174e04d39ee7004/packages/coding-agent/src/core/compaction/compaction.ts.

DeepSeek-AI. DeepSeek Harness: basic compaction. *Source repository*, commit cd5ef81, August 28, 2026. https://github.com/deepseek-ai/deepseek-harness/tree/cd5ef8148158c3a752a658978873241fdf8e2bbc/packages/compaction/compaction-basic.

Nous Research. Hermes Agent: context compressor. *Source repository*, commit 4209d37, August 29, 2026. https://github.com/NousResearch/hermes-agent/blob/4209d371aa1bb8840ce8447555bdd863a1a96c38/agent/context_compressor.py.

OpenAI. GPT-6.1-sol. *API documentation, model page*, accessed October 8, 2026. https://developers.openai.com/api/docs/models/gpt-6.1-sol.

OpenAI. Prompt caching. *API documentation*, accessed October 8, 2026. https://developers.openai.com/api/docs/guides/prompt-caching.

Z.AI. Pricing. *API documentation*, accessed October 6, 2026. https://docs.z.ai/guides/overview/pricing.

DeepSeek. Models & pricing. *API documentation*, accessed October 6, 2026. https://api-docs.deepseek.com/quick_start/pricing.

AI USAGE DISCLOSURE

We used the AI assistants Claude Code and Codex throughout this work: to help plan and run the experiments, write and revise the runtime, experiment, and analysis code, analyze data and make figures, search and summarize related work, review results and drafts, and draft and edit the manuscript. Within the method, language models also serve as blind annotators of agent actions (Appendix H). We take full responsibility for the work.

Guide to the appendices. Tables write policy and threshold together: S64 is Sync at a 64K threshold. Runs and settings: Appendices A, K, W, and F. Prices and cache accounting: L, M, and N. The work after a switch: Q, R, R.1, S, and T. Complete runs, including the Codex CLI: O, O.3, and W. Remaining work at common prices and a deployment check: X and X.1. Forecasting and registered predictions: G, P, and U. First actions, annotation, and summary wording: B, C, H, V, and I. Self-hosted replays: D. Run-to-run variation: E. Cost composition and context trajectories: Y. Related work: Z. Artifact: J. Table 7 lists the experiments with where each is reported.

A Early GLM and DeepSeek Runs and Deployments

Keep-2 ran at 64K on all five tasks and at 32K on Coq; we report Coq32 separately so that it does not give Coq extra weight in five-task summaries. The branch experiments start from Sync runs of both models, plus one DeepSeek Coq64 Sync run used only as a branch source. Appendix F states how we handled interrupted runs.

The deployments have separate roles (Table 8). The cost questions concern hosted APIs, where developers pay per token at the provider's cache prices, so the agents run on GLM-5.3-Flash and, as a second model whose cached input is far cheaper, on DeepSeek-Flash, with fewer settings for budget reasons. GLM-5.3-Flash was reached through an API gateway shared with other users, and DeepSeek-Flash through the provider's public API. The self-hosted models measure server time on recorded steps (Section 4.4). Running the agents on them would not fit the Limit-only reference runs, whose prompts reached 687K tokens with DeepSeek, within the 201K and 82K tokens that our single-GPU configurations serve.

B Branch and Action Measures

We report a model's branch contrasts only if at least 80% of its first actions are classified and coverage differs by at most 10 percentage points between conditions, a criterion set before developing the second parser. We average responses within each branch point, points within each source run, and then weight source runs equally. These behavior hypotheses arose from earlier runs and are exploratory; outcome, time, and usage measures were fixed before the main runs.

The first GLM branches, from Sync runs of the first repetition, fell short of this criterion. Before generating any new branch response, we therefore froze a replication on all 15 Sync runs of the second repetition, with the same conditions, sampling, criterion, and weighting. Its annotators decide only whether an action is read-only, and an action referred to them counts as classified only if all three agree (Appendix H).

In full runs, we classify the first tool call of each of the first three agent steps after a summary's first use with the same parser. The annotators reviewed only branch actions, so here we count explicit read commands, those the parser alone classifies as read-only; programs and scripts form a separate category. Three-step measures use switches followed by at least three steps before the next summary. Steps from the fourth onward after each switch give a run's usual rate. We average switches within a run and weight runs equally; Sync/Async differences are paired by task, threshold, and repetition.

Effects on the first action. What is kept at the switch steers the next action. We restarted GLM and DeepSeek Sync runs from sampled switch points and recorded two first responses per condition without running them: Sync and Keep-2 with fresh summaries, and Sync's own summary with the two notices our runtime appends removed or replaced by an instruction to continue. A frozen parser, with three blind model annotators for programs and scripts, labels whether the first proposed action only reads existing state (Appendix H). Keeping the last two steps lowered the share of read-only first actions by 34–46 points with GLM and by 52–61 with DeepSeek; removing or replacing the notices moved it by at most 6 points with GLM and under 10 with DeepSeek (Figure 4c). Adding the last two steps to Sync's own summary at 49 GLM points lowered it by 29–41 points, so the kept steps, not Keep-2's different summary, make the change (Appendix C). Sol's continuations confirm this with the agent running to the end: after the first summary, Sync's first tool action only read existing state in every classified case and Sync+2's in 79%; averaged by root and task, the difference was -0.25 (-0.29 to -0.21; Prediction 6; Appendix V); GLM's first crossings show the same (Appendix W). In complete GLM runs, Sync's first action after a switch was an explicit read 68% of the time, four to five times its rate at other steps (Figure 4b).

Effects on subsequent work. Kept once, the two steps changed the first action; later, Sync+2 minus Sync was \$0.00 (-0.55 to 0.46) at 32K, too uncertain to tell, and at 64K Sync+2 made 4 fewer requests (1.3 to 6.7) at about the same cost. Kept at every switch, the steps did more: at 32K, Keep-2 added \$0.98 and 46 requests against Limit-only, less than half of Sync's excess, and Async, which also keeps raw steps after the summary, added \$0.80 and 29. From the same state, each policy passed 0.04–0.07 less often than Limit-only at 32K and between -0.05 and $+0.02$ at 64K.

Table 7. Experiment batches: model and access, size, where they are reported, and what cost or time is based on. Our accounting: the cache rule fixed before the Sol runs (Appendix L).

Batch	Model and access	Size	Reported in	Cost or time
Early runs and branches	GLM-5.3-Flash (gateway), DeepSeek-Flash (API)	Table 8	§4, §5; App. A	provider usage at list prices
GLM complete runs at 64K	GLM-5.3-Flash	141 runs	§6; App. W	provider usage at list prices
GLM continuations	GLM-5.3-Flash	78 from first crossings, 78 from later states	§5; App. W	provider usage at list prices
Sol complete runs	gpt-6.1-sol, max, ChatGPT plan	162 at 64K, 27 Mask at 32K	App. O.2	our accounting
Sol same-state continuations	as above	27 roots; 405 at 32K, 273 at 64K	§5; App. K	our accounting
Sol cache replay	as above, cache-reporting endpoint	the 126 S and L continuations at 64K	§3.3, §5; App. M	measured cache reads
Sol rerun, other prompts	as above, later runtime	96 continuations from 8 states	§5; App. T.1	reported cache reads
Sol additional tasks	as above, later runtime	12 roots on 6 tasks; 36 continuations from 9 states	§5; App. T.2	reported cache reads
Sol later complete runs	as above, later runtime	108 runs	§6; App. O.1	reported cache reads
Codex CLI complete runs	gpt-6.1-sol, max, ChatGPT plan, Codex CLI 0.163.0	81 runs at 64K	§6; App. O.3	reported cache reads
GLM RS rerun	GLM-5.3-Flash	6 continuations from 1 state	App. W	requests and tokens
Self-hosted replays	Qwen3.8-27B, Qwen3-32B, vLLM, one GPU	Table 8; 2 whole continuations alone and in 12 shared pairs	§4.4; App. D	server time

C Kept Steps and the Summary Limit

Both tests start from the 54 points of the GLM replication and use its model and settings: two responses per condition, tools not run, and condition order from a fixed seed. They ran after the main runs, alongside the second set of full runs on the same API.

Sync's summary with kept steps. Each point is restarted twice with Sync's summary from the replication: alone, and followed by the last two complete steps verbatim. The summary already covers those steps, so their content appears twice; the position of the switch, the number of compacted messages in the summary's header, and the range that the notices point to also change with the kept steps. At five points, our runtime added the identifiers of recent failed commands that the summary did not name, so the summary text differed between the conditions; we dropped these points before sampling, leaving 49 points from 14 source runs. With the kept steps, the next request held 21,165 tokens on average instead of 7,284 (median 12,086 instead of 7,082; largest 142,705 instead of 13,403). We weight responses as in the branches. As for the replication, the 62 actions that the parser labeled as programs or could not parse went to three new blind annotators, with 40 read-only and 20 writing actions sampled to check the parser: two new sessions of `gpt-6.1-sol`, both with the client reasoning setting `ultra` (not an API parameter; Appendix H), and a new Claude session that reported `claude-opus-5-5`. The first two agreed on all 122 actions, and each agreed with the third on 117 ($\kappa = 0.93$); all sampled parser labels matched. Of the 196 first actions, 93.9% were classified, 96.9% for Sync and 90.8% with kept steps. The read-only share with kept steps minus Sync's lies between -41.4 and -28.9 points; the cautious variant classified 91.8%, with a spread of 8.2 points, and gave -44.9 to -28.0. With the parser alone, explicit reads fell from 76% to 34% of first actions, explicit writes rose from 4% to 21%, and programs and scripts from 16% to 38%; at the same points, the replication's Keep-2 responses had 28%, 23%, and 39%, against Sync's 73%, 4%, and 18%. The Keep-2 and Sync responses come from the replication, sampled four days earlier; the newly sampled Sync responses matched them, with 76% explicit reads against 73%.

A lower initial summary limit. This test varies the initial output limit under our runtime's retry rule; it does not test shorter summaries. From each of the 14 source runs, one of its replication points is chosen with a fixed seed, and the summary is generated twice, with initial output limits of 4,096 and 12,288 tokens, each followed by two next responses. When a summary reaches its limit, our runtime discards it and generates it again with twice the limit. This happened to 5 of the 14 summaries at 4,096 tokens, whose final versions had 4,645–7,519 tokens, and to none at 12,288. Over all summary requests, the 4K condition generated 76,141 output tokens against 54,926; the final summaries had 55,661 and 54,926, with medians of 3,552

Table 8. Early GLM and DeepSeek runs and the self-hosted replays, before the Sol study (Appendix K) and GLM's complete-run and continuation batches (Appendix W). Table 11 gives the finished runs and how they ended.

Runs	Settings and role
GLM S/A	5 tasks × 3 thresholds × 3 repetitions: 45 pairs, 90 runs. Main comparison.
DeepSeek S/A	MP64, MP128, Retro32, Vig64, RS64: 5 pairs, 10 runs. Second model.
GLM K	5 tasks at 64K, plus Coq32, × 3 repetitions: 18 runs. Kept recent steps.
GLM L	5 tasks × 3 repetitions: 15 runs, before the S/A runs. Reference without threshold compaction.
GLM held-out	4 tasks × 3 thresholds × 3 blocks: 36 S/A pairs, 72 runs, and L once per task and block, 12 runs. New tasks and the forecasting test.
DeepSeek L	5 tasks × 1 run: 5 runs, after the S/A runs. Reference without threshold compaction.
GLM kept steps	49 replication points, S's summary alone or with the last two steps, two responses each. Kept steps without a new summary.
GLM summary limit	14 replication points, initial limits of 4,096 and 12,288 tokens, two responses each. Summary length and cost.
Qwen3.8 replay	13 switch points from GLM S runs, each replayed as L, S, and K three times; S against A at 2 points. Server time.
Qwen3-32B replay	The 9 points from 32K and 64K runs, replayed as L, S, and K three times; at Vig64 also L, S, and K in full twice, and L, S, and A three times with tool gaps. Server time.
Qwen3-32B, two sessions	Vig64 and RS64 together: L and S four times each and A twice, over two sessions; each policy alone once per point. Server time under sharing.

and 3,492. The next requests held 202,562 and 201,180 input tokens in total. At list prices the summaries cost 0.152–0.172 and 0.119 USD and the next responses 0.019 and 0.020 USD; the range covers two requests whose cached input was not reported. All 28 final summaries were non-empty and adopted. Neither test measures task success: each samples only the next response.

D Self-Hosted Replays

Hosted APIs bill tokens and hide the time a server spends on them. To measure that time, we replay switch points from GLM Sync runs on two open models served with vLLM (Kwon et al., 2023) on one 96 GB NVIDIA RTX PRO 6000 GPU: Qwen3.8-27B (Qwen Team, 2026), which uses full attention in 16 of its 64 layers and linear attention in the rest, and Qwen3-32B (Yang et al., 2025), which uses full attention in all 64. From each Sync run of the first GLM repetition, we take the first summary adopted after at least 20 completed steps: 13 points for Qwen3.8, and the 9 from the 32K and 64K runs for Qwen3-32B. At each point we replay the next 20 agent steps three times each as Limit-only (keep the full context), Sync (switch to the summary), and Keep-2 (switch and keep the last two steps). The replay reuses the GLM run's summary and steps at their original lengths, so only the context differs. The Qwen models do not see GLM's reasoning text, so the context at the switch was only 15K–68K tokens. We time prefill for every request and reconstruct decoding from decode times measured at fixed context lengths; on Qwen3.8, full-length runs at two points matched the reconstruction within 0.14%. At two Qwen3.8 points we also time Sync against Async, with the summary and the agent sharing the GPU. On Qwen3-32B, one point (Vig64) is also measured in full: Limit-only, Sync, and Keep-2 twice without tool gaps, in separate sessions, to check the reconstruction, and Limit-only, Sync, and Async three times in wall-clock time with the original tool gaps. Finally, two agents share the GPU: the Vig64 and RS64 points are replayed together, both as Limit-only, Sync, or Async. Appendix F gives the serving configuration and measurement details.

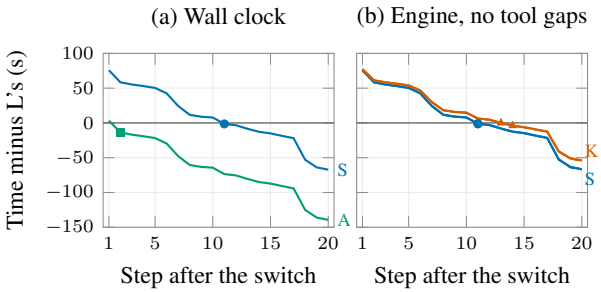


Figure 7. Time after one switch at Vig64 on Qwen3-32B relative to keeping the full context (L). (a) Wall clock with the original tool gaps, mean time differences from matched L across three runs per policy; these differences varied by under 0.4 s at every step. L is a matched run without a summary. S used its summary from step 1, after waiting for it, and A from step 2. (b) Engine time without tool gaps, two runs per policy in separate sessions; time differences from matched L varied by under 0.3 s at every step. Measured (solid) and reconstructed before the runs (dashed). Filled marks: first step below L in each run.

On a self-hosted GPU, the summary and the agent share the hardware. In the Qwen3.8 replays at Vig64 and Coq128, generating the summary took 67 and 151 s, and Sync waited for all of it. With Async, the next ten steps finished 60 and 142 s sooner than with Sync. The overlapping foreground requests took 4 and 6 s longer than matched requests without the background summary. On Qwen3-32B we measured all 20 steps after the Vig64 switch three times per policy, with the original tool gaps (Figure 7a); the time differences from matched Limit-only varied by under 0.4 s

across repetitions at every step. Sync waited 90 s for its summary, first finished below Limit-only at step 11, and completed the steps 67 s (4%) sooner. Async generated the summary alongside its first step, which took 3 s longer than Limit-only's, and completed the steps 139 s (8%) sooner than Limit-only and 72 s sooner than Sync.

Two sessions. When the Vig64 and RS64 replays shared the GPU, the two Limit-only sessions filled the KV pool in both repetitions; the server preempted one request in each and queued others. Sync used at most 74% of the pool and was neither preempted nor queued. The pair finished in 2,075 s with Limit-only and 1,685 s with Sync in both repetitions, 18.8% sooner; within the shared pair, the RS session finished 17% sooner with Sync than with Limit-only. Compared with running alone, sharing the GPU lengthened the Vig and RS sessions by 22% and 24% under Limit-only and by 3% under Sync. Run alone, Sync finished 67 s (3.9%) sooner than Limit-only at Vig64, as in the earlier sessions, and 4 s (0.3%) later at RS64. A second session repeated the Limit-only and Sync pairs to within 0.1% of the first. Cached tokens were 96% of Limit-only's prompt tokens and 93% of Sync's. Limit-only's delay came in two parts. Until about 1,050 s into each Limit-only pair, the pool was not full, and Limit-only's steps took a median 11% longer than alone, against 3–4% for Sync. When the two Limit-only prompts together reached the size of the pool, a request was waiting in almost every sample until the RS session finished; the steps in this window account for 280 of the Vig session's 371 extra seconds and 193 of RS's 301.

Sequence slots. Async behaved differently once the GPU was shared. Alone, it finished 72 s and 86 s sooner than Sync at Vig64 and RS64. Shared, each Async session could have a summary and a foreground request outstanding, four requests in all, while the server ran at most two sequences at once. To test the slot limit, we replayed the Async and Sync pairs with two and four slots on the same KV pool (5,403 blocks of 16 tokens, checked on every deployment), twice each in the deployment order 2, 4, 2, with two Limit-only pairs per setting. With two slots, Async's requests queued for 254 and 295 s in total, and the pair finished 1.8% later than with Sync in one repetition and 2.3% sooner in the other, as in an earlier session (1.7% and 2.3%); in the second, the RS session finished 64 s later with Async. With four slots, Async's queue fell below 1 ms, four requests ran at once, Async's KV peak rose from 74% to 78% of the pool without preemption, and Async finished the pair 61 and 66 s (3.6% and 3.9%) sooner than Sync; per session, 61–66 s sooner for Vig and 74–79 s for RS. Sync and Limit-only finished within 0.3% of their two-slot times. Async's summed engine time stayed above Sync's at four slots (3,022 and 3,005 s against 2,929 and 2,931 s): the background summary still shares the GPU's work, but no longer waits for a slot.

Whole continuations. The whole-continuation replays use Qwen3.8-27B (BF16, vLLM 0.30.0, a 200,704-token window, thinking disabled) on one GPU. For each of Prod, Retro, and RS, in that order, fixed before the replay, we took the first 64K state whose Sync and Limit-only continuations (repetition 1) had complete records and fit the window: RS roots 1 and 3 exceeded it, RS root 2 and Retro root 1 had a request with unknown output, leaving Prod root 1 and Retro root 2. Each workload ran twice on one server after a cache reset and a one-token request on the root's last input, without tool waits; the measure is the sum of request service times. Sol's encrypted reasoning was dropped from the inputs and the recorded output length, reasoning included, was forced; Sol's summaries have a median of 5.4K output tokens, 9% of them reasoning. Repetitions agreed within 0.4 s, and the engine's inference timer gave the same differences within 6 s (Table 9). Limit-only carrying Sync's work served 1.28M and 7.44M prompt tokens per replay, 96% and 98% from the prefix cache; Sync served 0.57M and 2.01M. Longer contexts slowed decoding by only 6–10% per output token. The original continuations ended with the same verifier outcome under Sync and Limit-only: both failed at Prod and both passed at Retro.

Table 9. Whole Sol continuations replayed on Qwen3.8-27B: requests (S's summaries in parentheses) and request service time (s), mean of two replays. Negative differences mean S took longer.

	Prod root 1	Retro root 2
Requests, L / S	5 / 17 (1)	8 / 62 (3)
L	554	1,168
L carrying S's work	899	1,998
S	1,098	2,346
S's agent requests	845	1,792
S's summaries	252	555
L carrying S's work − S	−199	−349
L − S	−543	−1,178

Whole continuations sharing the GPU. A later deployment of the same model and engine replayed the Prod and Retro workloads together, Limit-only with Limit-only and Sync with Sync, with two sequence slots, prefix caching, chunked prefill, and an 81,920-token window, which every request fits with its forced output (at most 80,257 tokens). Limit-only carrying Sync's work was left out: one of its Retro requests reaches 192,656 tokens, and a pool large enough for it would never make Limit-only wait. Before any policy request, a startup without generation measured the KV pool, and the two pools were fixed at 173 and 244 blocks of the hybrid cache, which the server reported as 127,677 and 180,076 tokens. The largest requests of the two Limit-only sessions need 153,856 tokens together, and

Sync's 126,158. The smaller pool ran first, then the larger, each in the order L, S, S, L, L, S. Each pair reset the prefix cache, sent one request on each root's last input, in reverse order in the second repetition, and started both sessions together; each session sent its requests one after another without tool gaps, and the pair ended with the later session's last response. Outputs were forced to the recorded lengths and discarded. All 552 workload requests completed. Table 10 gives the results; the three repetitions of each cell agreed within 11 s. In the smaller pool, Limit-only's Prod session spent 262–272 s in all before first tokens, against 9 s in the larger, and finished 43% later; its Retro session finished 14% later. Sync's sessions finished within 4 s of their times in the larger pool.

Table 10. Whole Sol continuations from Prod root 1 and Retro root 2 replayed together on Qwen3.8-27B, mean of three pairs per cell. Pair: time until both sessions finished (s). Preempt. and queue: per pair, from the engine's counters; queue sums the time the pair's requests waited, which is not the delay of the pair's end. KV peak: largest sampled share of the pool in use. Cached: share of prompt tokens served from the prefix cache. Per pair, L sends 13 requests and 0.85M prompt tokens, S 79 requests, four of them summaries, and 2.58M.

Pool		Pair	Preempt.	Queue (s)	KV peak	Cached
128K	L	1,399	4	246	100%	65–72%
	S	2,457	0	0	89–90%	90%
180K	L	1,231	0	0	74–75%	93%
	S	2,454	0	0	63%	90%

Scope. Decoding is reconstructed and checked against full-length runs at two Qwen3.8 points and one Qwen3-32B point, and repayment beyond 20 steps is extrapolated linearly. The two Qwen models differ in size, tokenizer, and position scaling as well as attention. The whole continuations cover two states that fit Qwen3.8's window, which leaves out the longest contexts, and their replays keep the recorded requests, so a summary's wording cannot change the work that follows.

A production deployment. With the operator's permission we exported the retained Prometheus history of a production deployment: aggregate metrics only, sampled every 15 s, without request content and without sending it any request. Its SGLang instance served GLM-5.3-Flash with a 1,048,576-token context limit and an 8,339,648-token KV pool for 542 observed hours between 24 September and 17 October 2026. Our own GLM runs went through it: their 11.5 billion reported prompt tokens are 13.4% of the 85.4 billion it counted, and 473 failed or cancelled calls without reported usage are left out. Active KV, which excludes reclaimable prefix cache, averaged 3.7% of the pool (median 1.3%, p95 15.9%), and its largest sample was 89.9%; physical occupancy averaged 99.6%, almost all of it reclaimable prefix cache. Prefix-cache hits covered 95.6% of prompt tokens, 2.4 points of them from host memory, so host offload was active for part of the period. Requests waited in the queue during 5.6% of the observed time, and the retraction gauge read zero in every sample. These observations place the replays' capacity boundary relative to one production deployment.

Table 11. Passed runs out of finished runs. GLM's S and A pool the three thresholds, and K ran at 64K and on Coq also at 32K; DeepSeek's S/A pairs ran at MP64, MP128, Retro32, Vig64, and RS64 (–: none). L never compacted. The lower four tasks are held out and ran with GLM only. The last two rows count failed runs that hit the eight-hour limit or ended on an error; the other failed runs finished and failed the tests. In 16 runs (12 passed, 9 of them on MP and 3 on Sound), the agent finished while a background command it had started was still running; the harness stopped the command before grading.

	GLM				DeepSeek		
Task	S	A	K	L	S	A	L
MP	7/9	5/9	1/3	0/3	2/2	1/2	1/1
Retro	0/9	1/9	0/3	0/3	0/1	0/1	0/1
Vig	6/9	5/9	1/3	1/3	0/1	0/1	0/1
RS	1/9	0/9	0/3	0/3	0/1	1/1	0/1
Coq	8/9	7/9	5/6	3/3	–	–	1/1
Risk	6/9	7/9	–	1/3	–	–	–
Sound	4/9	2/9	–	1/3	–	–	–
Prod	1/9	2/9	–	0/3	–	–	–
Onto	0/9	0/9	–	0/3	–	–	–
8-h limit	3	7	2	0	0	1	0
Error	2	0	0	0	1	1	0

E Run-to-Run Variation

Elapsed time. Repetitions of the same setting varied widely. Across the 36 GLM settings, each run three times, the slowest run took a median of 3.2 times as long as the fastest, and one Async setting at 64K took 0.2 h in one repetition and 8 h in another. Within the 45 GLM Sync/Async pairs, Async finished sooner in 23, and Async's elapsed time minus Sync's had a median of 0.0 h (95% bootstrap interval -0.4 to 0.4 h), while the Sync runs spent a median of 3% of their time waiting for summaries. Elapsed-time differences alone are thus a weak basis for attributing gains to hidden waiting, so we report the wait bound alongside them.

Pass rates. Success depended mainly on the task (Table 11): with GLM, Coq passed in nearly every run under every policy, and RS in only one of 24. Of the 45 GLM Sync/Async pairs, both runs passed in 11, neither in 16, Sync alone in 11, and Async alone in 7. Most of the difference came at 32K, where Sync passed 9 of 15 runs and

Async 4 (Sync alone in 6 pairs, Async alone in 1); at 64K and 128K the two were within one pass of each other. With few runs per task, and Keep-2 and Limit-only scheduled apart from the Sync/Async pairs, pass rates did not order the policies consistently.

F Additional Study Details

Task screening required environments that fit 4 CPUs and 4 GiB of memory per run. GLM-5.3-Flash and DeepSeek-Flash ran with thinking enabled, temperature 1, and at most 131,072 output tokens per agent request. A streamed response had to finish within 5,400 s, with at most 180 s between reads; a failed output could be regenerated once per step and three times per run, and requests that failed in transport were retried. Shell commands time out after 120 s unless the agent asks for up to 600 s, and background commands after four hours. The repetition guard stops a run after 50 identical polls, each under five seconds, of the same finished background command, or 100 while other commands may still be running; it stopped one run, a GLM first-crossing continuation (Appendix W). Every planned run is in the results except the GLM Sync+2 continuations that the runtime declined before any model call. Attempts cut short by an infrastructure or runtime fault were rerun with the same inputs after the fault was fixed, and the reruns replace them; no run was repeated for its outcome. The DeepSeek Vig64 pair ended on an empty model response (Sync) and an API timeout (Async); both original runs were retained as failures. One GLM Sync run at 128K ended after eight minutes, before its first compaction, when Sync and Async still behave identically; it also stays in the results as a failure.

For the API repayment estimate, counting only the drop in input across the switch, without the last response, leaves 589 rather than 626 of 717 GLM summaries, and 5 rather than 7 of 450 DeepSeek summaries, repaying before the next one replaced them.

The self-hosted replays used vLLM 0.30.0 with BF16 weights and KV cache, prefix caching, chunked prefill, at most two concurrent sequences, and 8,192 batched tokens. Coq32 and Vig128 had no replay point: the Coq32 run stopped after 12 steps, and Vig128's only summary left 5 steps. Each summary request is generated at its original length and timed, and its text is discarded; the replay continues from the GLM run's summary. Qwen3.8-27B (revision `1d4bf0f`) served up to 200,704 tokens. Qwen3-32B (revision `9216db5`) served up to 81,920 tokens with YaRN position scaling (factor 4 over 32,768 positions), which also applies at shorter lengths. Both used the model's chat template with thinking turned off, and earlier reasoning was removed from the history. We measured decoding with neutral prompts of 2K, 32K, 64K, 128K, and 200K tokens on Qwen3.8 and 2K, 32K, and 64K on Qwen3-32B, each with a cold and a warm cache three times, generating 256 tokens. We do not extrapolate decoding beyond the measured lengths. The Limit-only context of two Qwen3-32B points, Coq64 and Retro64, grew past 64K; a later session on the same deployment measured decoding at 81,664 tokens (61.9 ms per output token; prefill 37.4 s with a cold cache and 0.17 s with a warm one), and only these two points are reconstructed with both sessions' measurements. On Qwen3.8, the 20-step full-length runs at Vig64 and Coq128 differed from the reconstruction by -0.13% to $+0.04\%$ of engine time. For repayment, the one-time cost of Sync or Keep-2 is the summary's engine time plus the extra prefill time of the first request after the switch, relative to Limit-only; the saving per step is Limit-only's time for the 20 steps minus the candidate's, with that first-request time removed, divided by 20.

For the comparison with Async on Qwen3.8 at Vig64 and Coq128, Sync and Async alternate in three ABBA blocks at each of two tool-call intervals, one second and the original, six runs per policy and interval. Each of the next ten responses is capped at the smaller of its original length and 1,024 tokens, and the time runs from the trigger to the end of the tenth step. A separate run on the full context without a summary measures how much the summary slows Async's steps. At Coq128, the summary was not ready before Async finished its ten steps; at Vig64, where Async switched within them, the ten steps finished 3 s sooner than the same steps without compaction.

On Qwen3-32B at Vig64, the later session measured the 20 steps after the switch at their original lengths: first Async and a matched Limit-only run on the full context, then Sync, with the original tool gaps and timed from the trigger to the end of the 20th step's tool gap; then Limit-only, Keep-2, and Sync without tool gaps, timed in engine time. A further session repeated the runs with tool gaps twice, first with Sync before Async and Limit-only, then with Sync after them, and another repeated Limit-only, Keep-2, and Sync without tool gaps. Sync's summary took 90 s. Async's took 93 s alongside the agent's first request, which took 3 s longer than the matched Limit-only request, and Async used the summary from the second step. Without tool gaps, the reconstruction made before the runs fell 0.53–0.62% below the measured totals of Limit-only, Sync, and Keep-2 in the first run and 0.44–0.53% in the second; refitting with the new calibration reduced this error for Limit-only but not for Sync and Keep-2 in both.

The two-session replay ran in a later session on the same deployment, with 86,448 token slots in the KV pool. Vig64 and RS64 were chosen before any timing because they are distinct tasks whose largest requests fit the window: prompt plus output reached 55K and 47K tokens under Limit-only,

27K and 24K under Sync, and 35K and 31K for the summaries. Each policy first ran alone at each point; then the two points ran together as Limit-only, Sync, Sync, and Limit-only. Each run reset the prefix cache once and started both sessions together; they then advanced independently with the original tool gaps, and a session ended at the end of its 20th step's tool gap. Sync generated its summary at the original length and waited for it; as in the earlier replays, the generated text was discarded. We sampled the server's KV usage and running and waiting requests about once per second and read its counts of preemptions and queue time per run. A second session on the same deployment ran Async alone at each point, then the pairs in the order Limit-only, Sync, Async and Async, Sync, Limit-only. Async submitted its summary in the background and kept the full context in each request until one began after the summary was ready.

All 84 held-out runs were graded. In two Sound Async runs the container ran out of its 4 GiB memory; one of them, at 32K, later hit the eight-hour limit. Ten requests in the held-out runs have no recorded usage, so their runs' costs are lower bounds.

G FORECASTING PROTOCOL

Ledger. For GLM and DeepSeek, a summary's one-time cost C is its summarization requests plus the extra cost of the first request after the switch: that request's uncached input above the run's median, priced at the uncached price minus the cached one. The removed context is the input of the last request before the switch plus its reported output, minus the input of the first request after; priced as cached input, it is the saving per step s. For Sol, C and s use v2's cache-write and cache-read prices. U counts the accepted agent steps in the summary's cycle.

Forecast. The rule, its priors, the prices, and the split were saved together before any per-summary data from the held-out tasks were extracted, and the rule was applied once, after all 36 held-out Sync runs had finished; nothing was refitted. Priors are medians from the main tasks' Sync runs by model and threshold. Sizes are the runtime's own token estimates; the saving converts them to provider tokens with a prior ratio.

Before generation, the forecast uses the current context, the fixed instructions and kept content, a prior for the summary's size, and the growth g: the median of the last at most eight increments since the last switch, with at least three required, else a prior (needed once in 160). Once ready, it uses the actual context after the switch and the summary's actual requests. Neither uses the first request after the switch. With T the threshold and *post* the context after the switch, $\hat{U} = \max(1, (T - \mathit{post})/g)$; the saving per step is the removed context priced as cached input; R is the summary's cost plus a prior for the extra uncached input of the first request, divided by that saving; and the forecast is that the summary repays if $\hat{U} \geq R$. The fixed prior uses the median lifetime and median R at the threshold; the previous-summary estimate uses the last completed interval of the run, or the fixed prior for a run's first summary (30 of 160).

The lifetime forecast is scored on completed intervals, from a summary's first use to the next summary request. The label counts the steps a summary was used until replaced or until the run ended, so 30 summaries cut short by the run's end stay in, 28 of which repaid. The six held-out Sync runs that never compacted, Prod and Onto at 128K, add no summaries. Two Sound64 summaries without complete usage have no label. Classification counts are normalized within each run, then runs and tasks are weighted equally; balanced accuracy is the mean of the recall on repaying and on non-repaying summaries from these weighted counts.

Table 12. Forecasting on the held-out tasks: 158 GLM summaries, 11 of which did not repay. Balanced accuracy weights runs, then tasks, equally; 0.5 is chance. "Before" is just before the summary is generated; "ready" is once it is ready. Lifetime errors are run medians averaged over runs and tasks, from 130 completed intervals. The fixed prior and the previous summary do not depend on the forecast time.

Estimate of $\hat{U}$	Balanced accuracy	Losers found (of 11)	Lifetime error steps / factor
Growth, before	0.49	0	9.6 / 1.8
Growth, ready	0.59	1	9.6 / 1.8
Fixed prior	0.50	0	9.8 / 1.9
Previous summary	0.58	1	7.3 / 1.8

Of the 11 non-repaying summaries, 10 came at 32K. On the summaries whose forecast was within a factor of two of the line, 40 before generation and 34 once ready, balanced accuracy was 0.47 and 0.64 for the growth forecast and 0.59 and 0.63 for the previous summary's lifetime. Once ready, the one non-repaying summary that both found was on Prod; on Risk and Sound neither found any. On the main tasks, where the rule was developed, balanced accuracy was 0.53 for the growth forecast and 0.75 for the previous summary with GLM, and 0.90–0.92 against 0.70 with DeepSeek's five Sync runs, which the held-out tasks do not test.

With Sol, the priors came from 275 summary cycles in each task's first source run (its 64K run and one 32K continuation per state), saved before the rule was scored once on the 870 labeled summaries from the other two source runs, 78 of which repaid; draws, states, roots, and tasks are weighted equally. Once the summary was ready, balanced accuracy was 0.70 for the growth forecast, 0.81 for

the fixed prior, and 0.80 for the previous summary.

Table 13 checks the forecast's terms against the ledger. The one-time cost and the saving per step agree closely; the errors come from the lifetime. Swapping in a summary's actual steps of use corrects most misclassified summaries and changes almost none of the others. The ledger leaves out the last tool result before the switch, so its removed context is slightly smaller than the runtime's own reconstruction; using the reconstruction instead changes few labels, all near the boundary, where steps used are between half and twice the repay steps.

H Action Annotation

Besides read-only, the categories are writing files, building or testing, polling, cancellation, reads of original tool output, and declaring the task complete. Contrasts are bounded by assigning undecided actions to either outcome; a separate check varies all program and script labels. The first parser left most commands unclassified. Its replacement was developed with condition labels hidden. Programs and scripts were then annotated in separate blinded sessions by two Codex CLI sessions running `gpt-6.1-sol` (client reasoning setting `ultra`) and a third annotator that reported using `claude-opus-5-5`. The first two agreed on all 141 program and script labels; the third agreed on 132 ($\kappa = 0.88$). All nine differences were between build/test and undecided. Only unanimous substantive labels were retained, leaving 118 of 141 classified.

For 17 further unresolved commands, 14 labels agreed once the third annotator's categories were mapped to the parser's; the rest stayed undecided. The third annotator also checked a random sample of 60 actions without changing the rule-assigned labels: all 40 read-only labels were confirmed, and 18 of 20 work labels mapped to writing.

For the GLM replication, a revised codebook was frozen before any new branch response. The parser's read-only and writing labels stand; every action that it labels as running a program, script, build, or test, or cannot parse, goes to three annotators, together with 40 read-only and 20 writing actions sampled to check the parser. Each annotator decides whether the whole action is read-only, not read-only, or unresolved, and a referred action counts as classified only if all three agree. The packet holds only the first tool call of each action under a new random identifier, without condition, source, or model. Two annotators were new Codex CLI sessions running `gpt-6.1-sol` through a ChatGPT subscription, and the third a Claude Code subagent running `claude-opus-5-5`. The three agreed on 172 of 179 actions; 159 received a read-only or non-read-only label and 20 stayed unresolved. Pairwise agreement was 97.2–97.8% ($\kappa = 0.95$–0.96), and all 60 sampled parser labels matched the annotators. A cautious variant, set in advance, also leaves unresolved any action with a low-confidence label or a plausible alternative. It classified 95.1% of first actions, with a spread of 7.4 points between conditions, and bounded Keep-2 minus Sync at -45.5 to -33.0 points. Under it, S-cont minus Sync spans -6.0 to $+0.3$ points, so we read no direction into the notice changes with GLM.

I Summary Wording

Every summary request repeats the agent's system prompt and the history being replaced, so that the provider can reuse the cached prefix, and ends with the user message below, reproduced verbatim. When failed tool observations carry identifiers that the summary must keep, the line "The following exact identifiers come from failed tool observations and must be preserved verbatim:" and one quoted identifier per line precede the last line.

> Organize this conversation into a structured handoff summary for the next model to continue the work. Output the following Markdown sections in this order, using concise bullets; write "None" for sections with no content.
> ## User Goals and Requests
> - The user's original and evolving goals; the user's latest original question that has not been fully answered.
> ## Technical Background
> - Technologies, conventions, and concepts needed to continue the work.
> ## Files and Code
> - Exact paths, important changes, and necessary code excerpts.
> ## Errors and Fixes
> - Errors, attempted fixes, actual validation results, and user feedback; preserve verbatim the latest error messages needed to continue.
> ## Unfinished Tasks
> - Work explicitly requested by the user but not yet completed; distinguish pending work, suggestions, and completed items.
> ## Current Work
> - What is being done, the current progress, tasks still running, and existing tool state.
> ## Next Steps
> - The next action consistent with the user's latest request; do not turn suggestions or items awaiting confirmation into authorization to execute.
> ## Key Constraints and Decisions
> - Still-valid user instructions, conditions, prohibitions, preferences, decisions and their reasons, and unresolved questions.
> Rules:
> 1. Faithfully preserve user corrections and still-valid constraints without guessing or inventing details; retain conditions, time limits, and scope.
> 2. Treat any earlier compaction summary in the input as an old handoff: merge still-valid information, update progress, remove content explicitly cancelled

Table 13. Forecast once the summary is ready, against the ledger: medians (quartiles) of forecast/ledger ratios, and label changes. Held-out: GLM's four held-out tasks; development: the five main tasks.

	GLM held-out	GLM dev.	DeepSeek dev.
Summaries	158	756	454
One-time cost C	1.01 (1.00, 1.04)	1.00 (0.98, 1.02)	1.00 (0.99, 1.02)
Saving per step s	1.05 (1.02, 1.09)	1.03 (1.02, 1.06)	1.06 (1.03, 1.19)
Lifetime U	1.00 (0.75, 1.64)	1.18 (0.78, 1.88)	1.11 (0.79, 1.69)
Misclassified	14	99	19
corrected by actual U	13	94	17
correct, changed by actual U	0	1	1
Removed, ledger/reconstruction	0.97 (0.94, 0.98)	0.97 (0.95, 0.98)	0.94 (0.84, 0.97)
Labels changed by reconstruction	1	5	1

> or superseded later, and do not cumulatively copy old summaries.
> 3. Exactly preserve paths, commands, errors, identifiers, values, and task templates needed to continue; do not describe existing execution or validation results as pending, and do not invent success.
> 4. Use English and be concise, but do not omit information needed to continue the task merely to meet a fixed length.
> 5. Output only the handoff summary; do not repeat these instructions or compaction wrapper tags, call tools, or take other actions.
> 6. For failed-tool evidence:v1:sha256: references, retain only the up to three recent references explicitly required below; omit older evidence hashes from the handoff. This limit does not apply to task-relevant paths, error codes, or other identifiers.
>
> Compact all content in this conversation preceding this instruction.

After every summary, the runtime appends the following two notices. The second appears when the original tool output can be read back; `<ref>` stands for the run's reference value. All other wording is reproduced verbatim.

> `[summary-fidelity]` Lossy paraphrase, not exact evidence. For exact values or quotes, verify original records; old assistant replies may also be approximate.

> `[source-evidence]` read_tool_evidence if available: `{"mode":"sources","ref":"<ref>"}`. Paginate sources, then read their evidence refs for original bytes. Listing is not reading.

Two variants of Sync change these notices: S-bare drops both, and S-cont replaces them with:

> Continue from the work already completed. Avoid repeating completed work. Use the summary to take the next appropriate action.

Both variants use the exact Sync summary, history cut, and message role of the same branch state. The continue cue is our own wording, modeled on the instructions that existing harnesses place after a summary. Because the variants change wording and length together, they measure the effect of the appended text as a whole. Both responses for a condition start from the same state and share its summary; no tool is executed. The rerun of Appendix T.1 places the same cue after every summary and runs the agent to the end.

J ARTIFACT

The artifact (https://huggingface.co/datasets/context-cost-study/context-compaction-artifact) contains an offline analysis archive, redacted historical trajectories, and the standalone runtime source. The analysis archive provides frozen numeric inputs, reduction scripts, protocols and reviewed evidence for the analyses listed in its README; its documented recomputation uses no model service. Its inventory records coverage and excluded private inputs.

The runtime includes dependency declarations, installation and manual-use instructions, a demo that makes no model-generation calls, focused offline tests and explicit provider configuration for new manual runs. It does not include a complete frozen study environment or historical live checkpoints, and therefore does not by itself rerun the study.

Historical trajectories are readable evidence projections, not exact provider wires or executable checkpoints. Credentials, identifying host paths and private URLs are redacted. So that the release republishes no grader source, the output of any tool call that reads files under `/tests/`, `/oracle/`, `/grader/` or `/hidden_tests/` is replaced by a placeholder; this changes only the released text, and the analyses use the original records. During the runs, the agent's instructions told it not to seek verifier tests, and Harbor uploads a task's tests only when grading starts. Original upstream author, copyright and license notices and public benchmark anti-training-contamination canaries are retained. First-party attribution is withheld in

the anonymous review copy. Component-specific licenses distinguish runtime and research code, first-party data and third-party material.

K Sol Runs

Table 14 lists Sol's runs. Each root is a Limit-only run that saves its state when its context first crosses 32K and again at 64K. Six roots, all three for Coq, two for Sound, and one for Vig, finished before 64K, so their 64K continuations do not exist; same-state averages use the 48 states that were reached.

Table 14. Sol runs. Continuations per state: L, S, and S+2 three each; K and A two each; KB two at 32K only. The 64K continuations were planned for all 27 roots; 21 reached 64K.

Set	Planned	Analyzed
64K, six policies × nine tasks × 3	162	162
H at 32K	27	27
L roots	27	27
Continuations at 32K (27 states)	405	405
Continuations at 64K (21 states)	351	273

Restore check. Restoring a saved state does not restore process memory or clocks. An audit of the root logs up to all 48 saved states found every recorded command finished and every tool call returned, so no state was excluded; at five states a finished command still had output left to read, which the saved state keeps. If restoring changes nothing, each root's own run past the state is exchangeable with its three Limit-only continuations; for a continuous measure without ties, it then falls inside their range with probability one half. For cost it did so at 24 of the 48 states. Requests are whole numbers and tied a continuation at 17 states, which raises that probability; they fell inside at 30. For success, the root's outcome matched the majority of its three Limit-only continuations at 46 states and all three at 43.

L Sol Cost Accounting

Sol ran on a ChatGPT subscription without a per-request bill, so its costs are priced at the API's listed prices. They use the cache accounting of Section 3, called v2 in the registered protocol: a request reads from the cache the longest complete input of an earlier request in the same run that it repeats as a prefix, if that request finished within the last 30 minutes, and writes the rest. It is an accounting rule fixed before the runs; Appendix M compares it with the provider's cache on replayed requests.

Without a cache. The no-cache account prices every input token at the uncached rate (2.00, or 4.00 above 272K input tokens), with no cache-read or cache-write charges, and keeps v2's output prices. It is therefore not v2 with the read price raised to the uncached one, which would still charge cache writes at 2.50. At 64K, Sync, Async, Keep-2, and Capped Keep-2 cost more than Limit-only in v2 and less without a cache (Table 15; Sync minus Limit-only $-\$0.26$, 95% interval -0.85 to 0.28). The difference comes from RS, where each policy saved \$6.3–7.0 per run against Limit-only without a cache; on the other eight tasks, Sync cost more on six, Async and Keep-2 on five, and Capped Keep-2 on seven. From the same states (Table 16), Sync and Sync+2 cost more than Limit-only in both accounts at 32K; at 64K, every policy costs more in v2 and less without a cache. At 64K, RS is the only task where Sync's continuations cost less than Limit-only's without a cache, by \$5.27; at 32K, S saved on four tasks, by at most \$1.84 (RS), and cost up to \$15.33 more (Retro). The added requests are the same in both accounts.

Table 15. Sol mean cost per run (USD) at 64K, and H at 32K: v2, without a cache, with interior prefixes credited, and with every interior prefix read from the cache.

Policy	v2	No cache	Interior	$f = 1$
L	0.77	3.52	0.77	0.77
A	0.96	3.00	0.96	0.96
KB	0.96	2.99	0.96	0.96
K	1.00	3.07	1.00	1.00
S	1.00	3.26	1.00	1.00
H	16.80	14.04	5.10	1.50
H32	15.21	13.98	6.61	2.00

Table 16. Sol same-state difference, policy minus L, USD per state, in v2 and without a cache.

State	Policy	v2	No cache
32K	S	2.09	3.68
	S+2	2.09	3.60
	K	0.98	0.38
	KB	0.78	0.19
	A	0.80	−0.01
64K	S	0.26	−0.25
	S+2	0.25	−0.37
	K	0.19	−0.60
	A	0.21	−0.51

What the provider reported. The responses of the endpoint used for the runs report cached tokens, but not in a form that can price requests. Over the 59,696 completed agent requests of Sol's runs and continuations, v2 counts 71% of input tokens as cache reads; the provider reported 14% as cached and no cache writes. In 84% of requests it reported exactly 1,664 cached tokens, whatever the input length, and the share reporting more did not depend on the time since the previous request (13–16% for gaps from un-

der 5 s to 10 minutes). Priced with these counts, almost all input would be uncached, close to the account without a cache. The registered Prediction 3 used these counts and the no-cache account (Table 32). Replayed through another endpoint, the same inputs were read from the cache much as v2 has it (Appendix M).

Interior prefixes. Mask hides earlier tool output, so its later requests repeat earlier inputs only up to the first masked item. *Interior* also credits the longest unchanged prefix inside an earlier request's input, not only complete inputs; on the 27 Limit-only runs at 64K, the per-item token counts it relies on added up to the reported totals. The column $f = 1$ assumes that every such interior prefix is read from the cache. Only Mask has interior prefixes; for every other policy these columns equal v2.

Requests without usage. Of 74,002 requests, 294 failed or ended without reported usage. They are listed with their run and excluded from cost subtotals.

M Measured Cache Replay

The replay covers the 126 continuations from Sol's 21 states at 64K, three Sync and three Limit-only continuations per state, with the 3,305 requests that completed in the runs. Each continuation was replayed in its own session under its own cache key. It began with one warming request that repeated the root's last input before the saved state, under the same key, and then sent the recorded requests in order, keeping the gaps between the original requests; no tool ran. Each request carried the recorded input items, encrypted reasoning included, at reasoning effort max. The replay's outputs were recorded and discarded, so every request carried the recorded history. The endpoint (`chatgpt.com/backend-api/codex/responses`) was reached through a different ChatGPT account from the runs and adds 17 tokens of instructions to each request; with these removed, each request's input equaled the run's to the token.

Our accounting is recomputed on the replay's own timeline, with the warming request, by the implementation used for the runs (Table 17). The endpoint reported no cache writes. The replay cost prices measured reads at the read price, other input at the uncached price, and the run's output; pricing the other input at the cache-write price, as our accounting does, gives an upper value. Outputs generated by the replay and the warming requests are measurement costs and are left out. Per state, the measured saving differs from the saving under our accounting in the runs by at most \$0.12, with a median of \$0.02. With measured reads, Sync cost more on average at every task, and leaving out one task at a time keeps the mean between \$0.24 and \$0.29.

Table 17. Replayed Sol continuations at 64K. Top: share of input tokens read from the cache, measured and under our accounting, on the same requests. Bottom: S minus L per state (USD; task-macro mean, 95% interval resampling roots within tasks) and states where S cost more, by cache reads and the price of other input.

		Requests	Measured	Ours
All		3,305	92.9%	93.9%
L		893	96.3%	96.8%
S		2,412	88.8%	90.4%
Reads	**Other input**	**S − L**	**Interval**	**S more**
Measured	uncached	0.26	0.20–0.31	20/21
Measured	cache write	0.29	0.23–0.34	20/21
Ours, replay	uncached	0.24	0.19–0.28	20/21
Ours, replay	cache write	0.27	0.22–0.32	21/21
Ours, runs	cache write	0.26	0.21–0.33	21/21

N Cache-Price Crossover

Every cost here is linear in the price of a cache read. With r that price as a fraction of the uncached input price, a run costs $A + rB$, where B is its cached input tokens priced as uncached; Sync minus Limit-only is $\Delta A + r\,\Delta B$ and changes sign at $r^* = -\Delta A/\Delta B$. Only the read price changes: each model keeps its own uncached, output, and (Sol) write prices and 272K tier, and every run keeps its recorded requests and cache allocation: for Sol, v2's in the earlier batches and the reported cache reads in the later complete runs and the Codex CLI. Averages use the paper's weights (Table 18).

With GLM, Sync's continuations carried far fewer cached tokens than Limit-only's, and their other costs were about equal: at the first crossings, ΔB was $-\$6.12$ and ΔA $+\$0.02$ per state, so Sync cost less on average once r exceeded 0.4%; seven states cross between 0.2% and 44%, and two favor Sync at any price (Table 19). At the later states, where Sync's added work was largest, the average crossover was 4.5%, between the ratios of DeepSeek's (2%) and GLM's (20%) list prices. With Sol at 64K, Sync's added requests and summaries cost $+\$0.28$ per state before any read and saved $\$0.50$ in reads at the uncached price, so Sync cost less on average only above $r = 56\%$ (95% interval 30–148%), against a listed 5%; Async and Keep-2 cross at 34% and 29%. In the earlier complete runs at 64K, Async, Capped Keep-2, Keep-2, and Sync cross at 30%, 30%, 37%, and 51% (95% intervals over the draws in which the average crosses: 8–219%, 13–95%, 23–79%, and 19–469%), and Mask only at 656%. In each, the saving in reads comes from RS: averaged over the other eight tasks, all four cost more than Limit-only at any read price. Most of that saving is RS's, where Limit-only's continuations carried long contexts and Sync crossed at 9.1%; at five of the eight tasks, Sync's continuations read more cached

tokens than Limit-only's, so Sync cost more at every price. At 32K the same held on average ($\Delta B = +\$1.87$). In the later complete runs, with reported cache reads, Sync crosses at 34% (95% interval 16–87%; 1.6% of draws do not cross below the uncached price) and Mask-10 at 99.7% (84–117%) (Appendix O.1). Leaving out one task moves Sync's crossover between 25%, without MP, and 122%, without RS, so here too the crossing comes from RS. In the Codex CLI (Appendix O.3), server-side compaction crosses at 18% (6–103%; 14–46% leaving out one task) and local compaction at 52% (26–156%). Every interval lies above the listed 5%. Pricing the input not read from the cache at the write price instead of the uncached price raises Sync's excess in the later runs to \$0.20 per run and its crossover to 39%, and in the Codex CLI to \$0.17 and 21% (local compaction: 63%). At the same r of 5%, Sync cost less at six of GLM's nine first crossings and at none of Sol's 21 states at 64K. A line that keeps one sign has no crossover; "any" and "none" in Table 19 mark these.

Table 18. S minus L as the cache-read price r varies (fraction of the uncached input price): mean difference in USD and the states (or task/repeat blocks of complete runs) where S cost less (Less), at Sol's listed 5% and GLM's 20%, and r^*, above which S costs less on average (none: S costs more at every r). Sol in v2's allocation; its earlier complete-run r^* has a 95% interval of 19–469% over the 97% of bootstrap draws in which the average crosses at all.

		$r = 5\%$		$r = 20\%$		
Model	Set (units)	S−L	Less	S−L	Less	r^*
GLM	First crossings (9)	−0.28	6	−1.20	8	0.4%
	Later states (9)	−0.03	7	−0.79	8	4.5%
	Complete 64K (27)	−0.46	19	−1.99	22	0.5%
Sol	64K states (21)	0.26	0	0.18	4	56%
	32K states (27)	2.09	3	2.37	3	none
	Complete 64K (27)	0.23	6	0.15	8	51%

Table 19. Where S costs less than L, per task: S minus L (USD per state) at the listed read price, and the read prices r at which S costs less (any: every $r \geq 0$; none: no r). GLM: one first-crossing state per task, at 20%. Sol: same-state averages in v2's allocation, at 5%; Coq never reached 64K.

	GLM, first		Sol, 64K		Sol, 32K	
Task	S−L	S less	S−L	S less	S−L	S less
MP	−0.52	$r > 1.3\%$	0.27	none	4.31	none
Vig	−0.01	$r > 14\%$	0.22	none	2.64	none
RS	−5.94	$r > 0.2\%$	0.23	$r > 9.1\%$	2.51	$r > 63\%$
Coq	−2.18	$r > 0.7\%$	–	–	0.08	$r > 26\%$
Retro	−1.48	$r > 0.9\%$	0.35	$r > 120\%$	7.01	none
Risk	−0.42	any	0.34	none	0.65	none
Sound	−0.23	any	0.05	none	0.07	$r > 81\%$
Prod	−0.01	$r > 8.6\%$	0.41	none	0.01	$r > 6.3\%$
Onto	0.00	$r > 44\%$	0.16	$r > 270\%$	1.57	none

O SOL COMPLETE RUNS

O.1 Later Batch at 64K

The later batch runs each of the nine tasks three times under Limit-only, Sync, Mask-10, and Mask-10+summary from the task's start, 108 runs, on the runtime of Appendix T without its warming request. Within each task and repeat, the four policies start in an order fixed by a seeded shuffle; up to 16 runs ran at once, each under its own cache key. The model, tools, verifier, threshold, and eight-hour limit are those of the earlier batch. A tool batch is one response's tool calls with their results; Mask-10 replaces older output with Mask's placeholder, and a batch whose results are incomplete is left as is and not counted. Mask-10+summary measures the threshold on the masked request and summarizes the raw history, as Mask does. Every run received an official reward. Four requests in three Mask-10+summary runs, at MP and RS, failed without token counts; the runs continued and were scored. Since a missing count cannot be negative, the policy's known cost, \$68.97 over 27 runs, bounds its mean from below. Break-even read prices vary the read price with each request's reported cache reads fixed; cost is linear in the read price, so each follows from the policy's cost at the listed and at the uncached read price. Sync cost more than Limit-only on seven tasks, on Vig by \$0.003, and less on Coq and Sound; Table 20 gives each task.

Table 20. Later batch by task: runs passed (of 3) and mean cost per run (USD); ≥ marks a lower bound.

	Passed				Cost			
Task	L	S	M10	M10+S	L	S	M10	M10+S
MP	3	3	3	3	0.60	0.96	3.55	≥2.22
Vig	3	2	0	1	0.42	0.42	1.25	1.29
RS	3	3	3	3	1.62	1.91	11.97	≥7.46
Coq	3	3	3	3	0.50	0.41	1.28	1.31
Retro	3	3	3	3	1.22	1.61	3.89	4.54
Risk	3	3	3	3	0.79	0.91	2.12	1.62
Sound	3	3	3	3	0.38	0.37	0.61	0.69
Prod	1	0	0	0	0.55	0.91	1.25	1.74
Onto	0	0	0	0	0.83	0.93	2.19	2.12

O.2 Earlier Batch at 64K

The earlier batch ran six policies on the nine tasks at 64K, three times each, on the original runtime, priced under our cache accounting (Table 21). A run that finishes before its context reaches 64K never summarizes (column Comp.), though Mask can still mask in such a run. Seven of the nine tasks had the same outcome under every policy, in all three runs; the policies differ only on Vig, where 0–2 of 3 runs passed, and RS, where 2–3 did.

Every summarizing policy cost more than Limit-only:

Sync by \$0.23 (95% interval 0.14–0.31), Keep-2 by \$0.22, Capped Keep-2 by \$0.18, and Async by \$0.18 (0.05–0.30). Sync made 20 more requests than Limit-only and took 539 s longer; Async, which keeps working while it summarizes, added the fewest requests (9) and the least time (215 s). The four would cost less than Limit-only only if a cache read cost more than 30–51% of an uncached token, a crossover that comes from RS, where Limit-only's runs carried the longest contexts; on the other eight tasks, each cost more at any read price (Appendix N). Capped Keep-2's cap applied at only one of its 43 compactions at 64K, so Section 5 tests it at 32K.

Mask kept the output of the last two tool batches and summarized as a fallback in 19 of its 27 runs. It made 226 requests per run against Limit-only's 29 and took 2.3 times as long, with the same success; even without summarizing, it made 31 against 18. Its cost depends on how much of the rewritten input the cache serves: \$16.80 under our accounting, \$5.10 when unchanged interior prefixes are read from the cache, and \$1.50 when every interior prefix is, against Limit-only's \$0.77 (Appendix L).

Table 21. Earlier batch: Sol at 64K, 27 runs per policy. Pass counts runs with verifier reward 1. Cost (USD) under our cache accounting at Sol's listed prices, time (s), completed requests, and summaries (Summ.) are task-macro means per run; Mask's summaries are its fallback summaries. Break-even: the cache-read price, as a share of the uncached input price, above which the policy would cost less than Limit-only on average. Comp.: runs with at least one summary.

Policy	Pass	Cost	Break-even	Time	Req.	Summ.	Comp.
L	19	0.77	–	1289	28.7	0	0
A	19	0.96	30%	1504	37.6	1.56	19
KB	18	0.96	30%	1720	41.1	1.59	20
K	18	1.00	37%	1819	41.0	1.67	19
S	18	1.00	51%	1829	48.2	1.70	21
H	19	16.80	>100%	3026	226.4	1.59	19

O.3 The Codex CLI

These runs replace our runtime with the open-source Codex CLI, release 0.163.0 (OpenAI, 2026c), which supplies the tools, prompts, history handling, and compaction; the model is Sol at maximum reasoning effort with a configured window of 872K tokens. The nine tasks, verifiers, container resources, and eight-hour limit are those of Section 6. Limit-only leaves the CLI's automatic compaction at the window, which no run reached; its largest request had 246K input tokens. Sync compacts when the context reaches 64K through the provider's server-side compaction, the CLI's default for this provider, which returns the summary encrypted. Sync-local (S-local) keeps the threshold and turns server-side compaction off, so the CLI asks the same model for a summary with its local prompt, the prompt of Appendix T.1. A proxy forwarded each request unchanged and recorded its provider-reported usage, including that of compaction requests, which the CLI's own token counter leaves out. Every request of the runs below completed with its usage, and every run received an official reward.

Each policy ran each task three times, 81 runs in all, up to 16 at once. Intervals resample repeats within tasks with the policies paired by repeat.

Table 22 gives the means and Table 23 each task. Each of the 53 server-side and 69 local compactions was carried into the next request. Eight tasks had the same outcome in every run; Onto and Prod failed in all, and Sync lost one Vig run. Sync cost \$0.14 more per run than Limit-only (0.02–0.26), less only at Coq and MP, and made 13.6 more requests (8.5–19.0), 11.6 of them agent requests. It sent 20% less input, but over its 27 runs read 14.3M fewer tokens from the cache and added 1.5M uncached input and 0.23M output tokens; priced with no cache, it would cost \$0.87 less per run. At Coq, where no run reached 64K, Sync still made 7 more requests than Limit-only, a measure of how far independent runs differ. Sync-local cost \$0.51 more than Limit-only (0.37–0.66), more on eight tasks, at Sound by \$0.005, and \$0.37 more than Sync (0.24–0.49), with 3.7 more requests (-1.6 to 8.5); most of its excess over Sync was uncached input, 8.9M against 4.5M tokens.

Table 22. Sol in the Codex CLI at 64K, 27 runs per policy. S: server-side compaction; S-local: local compaction. Pass counts runs with verifier reward 1. Cost (USD): reported cache reads at Sol's listed prices, compaction requests included. Break-even: the cache-read price, as a share of the uncached input price, above which the policy would cost less than Limit-only. Completed requests (Req.), compactions (Comp.), and cost are means per run; Runs: runs with at least one compaction. Cached: share of input read from the cache.

Policy	Pass	Cost	Break-even	Req.	Comp.	Runs	Cached
L	21	0.91	–	31.1	0	0	95%
S	20	1.05	18%	44.6	1.96	21	91%
S-local	21	1.41	52%	48.3	2.56	20	84%

O.4 Complete Policies at 32K

At 32K, each complete policy is a root's run up to its first 32K crossing followed by one continuation (Table 24). These are assembled from shared roots, not independent runs.

O.5 Mask at 32K

Mask at 32K passed 18 of 27 runs, made 343.5 completed requests per run, and took 4331 s. Against Mask at 64K, it made 117.1 more completed requests (95% interval 3.8–222.9), with a success difference of -0.04 (-0.11 to 0.04)

Table 23. Codex CLI runs by task: runs passed (of 3) and mean cost per run (USD).

	Passed			Cost		
Task	L	S	S-local	L	S	S-local
MP	3	3	3	1.04	0.94	1.49
Vig	3	2	3	0.50	0.56	0.71
RS	3	3	3	2.10	2.33	2.77
Coq	3	3	3	0.51	0.50	0.50
Retro	3	3	3	1.61	2.07	3.05
Risk	3	3	3	0.70	0.85	1.03
Sound	3	3	3	0.39	0.45	0.39
Prod	0	0	0	0.57	0.73	1.02
Onto	0	0	0	0.72	0.99	1.77

Table 24. Sol complete policies at 32K, assembled from roots and continuations; task-macro means per run.

Policy	Success	Cost v2 ($)	Requests	Time (s)
L	0.72	0.82	29.7	1444
A	0.65	1.62	58.9	1714
KB	0.65	1.59	75.6	2527
K	0.65	1.79	75.7	2869
S	0.68	2.91	159.6	4270

and a v2 cost difference of $-\$1.59$ (-8.9 to 4.8). Every Mask run at 32K wrote fallback summaries, 165 in all, against 43 in 19 of the 27 Mask runs at 64K.

P Sol Switch-Time Forecast

The forecast is the GLM rule of Section 4 with Sol's v2 write and read prices. Its priors for 32K and 64K were fitted on development summaries: the first round of the 64K Sync runs, and the first Sync continuation at each task's first root. Validation uses the other rounds and roots. Of 887 validation summaries, 870 had a label and a forecast. Near the boundary, with forecast lifetime between half and twice the repayment steps, balanced accuracy was 0.63 on 110 summaries once the summary was written. At the 32 branch states with a forecast for the first summary, the forecast's call agreed with the sign of the same-state difference at 0.75 of states, task-weighted; it called 7 states profitable, and Sync was cheaper than Limit-only at 2.

Q Fixed-Trace Estimate

For each Sync continuation, we reprice Sync's completed agent requests as Limit-only would have sent them: each request's input grows by the context removed at all earlier switches of the continuation, and that added history is read from the cache; the first request after a switch writes only the run's median new input instead of Sync's new prefix. The removed context at a switch is the per-summary rule's: the input and output of the last agent request before it minus the input of the first after. For Sol, each request is priced in v2, with the higher rate when its enlarged input exceeds 272K tokens. The estimate F is this repriced cost minus Sync's actual cost, summary requests included, and the same-state saving minus the estimate, $G - F$, is Limit-only's actual cost minus the cost of carrying Sync's requests at Limit-only's context. It compares two request sequences under the assumption that the added history stays cached, so it includes differences in how many requests each made, what they contained, and where each run stopped. Continuations whose per-summary labels are incomplete are left out of the estimate, as in Prediction 5.

Carrying Sync's work, Limit-only would sometimes have held more context than it can. In Sol's Limit-only runs, the runtime compacts at about 1.02M tokens, below the 1.05M-token window; one 32K state, on Retro, passes that limit in at least one continuation, and the next largest reaches 0.95M. Table 25 reports all states and, separately, the states where Limit-only stays below its limit in every continuation; the subset is the same for any limit between 0.96M and 1.7M tokens. Figure 8 shows every state.

GLM's prices are flat, so the estimate there is the sum over summaries of $U_j^+ s_j - C_j$; every GLM Sync continuation compacts at its first request, whose preceding request belongs to the source run, so the context before that switch is the input of Limit-only's first request from the same state. At the six later Coq states, carrying Sync's work would have needed at most 0.45M tokens; the estimate averaged $0.74 and the saving $1.05, with source runs, then tasks, weighted equally. At each later Retro state, at least one continuation would have needed 1.56–2.38M tokens, beyond GLM's 1M-token window. At the nine first crossings, it would have needed at most 0.78M tokens; the estimate averaged $1.28 and the saving $1.20, tasks weighted equally.

R Added Requests

We label every tool call of the Sync and Limit-only continuations' agent requests with the frozen classifier of Prediction 6 (Appendix H): read-only (inspection and file search), writing files, running programs, scripts, builds, or tests (Run), polling or cancellation (Wait), and responses without a tool call or that the parser cannot read (Other). A request is read-only if all its calls are; otherwise it takes the first of Writing, Run, Wait, and Other among its calls. Of Sol's 15,517 Sync and Limit-only requests, 263 carry more than one call; labeling each request by its first call alone moves no count in Table 26 by more than 0.5 for Sol or 1.6 for GLM. A switch is the agent request that first uses a summary, from the runtime's adoption records (one of Sol's 1,419 Sync summaries could not be placed); a request is within five after a switch if it is that request or one of the next four, counted once where windows overlap. At 32K,

Table 25. Sol same-state saving of S over L (G, L−S, v2), the fixed-trace estimate F, and the per-summary sum, USD per state, with root-cluster 95% intervals. The per-summary sum adds each summary's return over its own cycle (Prediction 5's basis). The lower block keeps the states where L, carrying S's work, stays below its compaction limit in every continuation.

	32K (27 states)	64K (21 states)
Same-state saving G	-2.09 (-2.67, -1.61)	-0.26 (-0.33, -0.21)
Fixed-trace estimate F	8.57 (3.48, 16.12)	0.17 (0.14, 0.21)
$G - F$	-10.66 (-18.42, -5.47)	-0.42 (-0.53, -0.37)
Per-summary sum	-0.80 (-1.11, -0.57)	0.02 (0.01, 0.03)
States: $F > 0$ / $G > 0$ / $G < F$	14 / 3 / 24	18 / 0 / 21
	32K (26 states)	64K (21 states)
Same-state saving G	-1.98 (-2.55, -1.44)	as above
Fixed-trace estimate F	4.64 (2.94, 6.37)	
$G - F$	-6.62 (-8.41, -4.81)	
States: $F > 0$ / $G > 0$ / $G < F$	13 / 3 / 23	

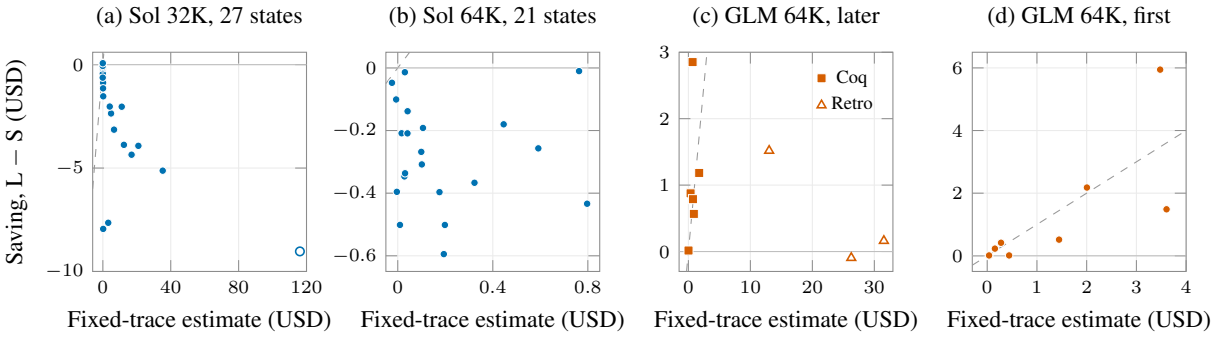


Figure 8. Same-state saving of S over L against the fixed-trace estimate, one point per state (mean over repeats). Sol under our cache accounting at listed prices, GLM at its prices: (c) nine later states of three earlier S runs, (d) the first crossing on each of the nine tasks. Open marks: carrying S's work, L would have passed its own compaction limit, so the estimate prices work L could not have done. The dashed line is equality; below it, the complete continuation was less favorable to S than the estimate. Scales differ by panel.

where Sync's summaries last a median of eight steps, these windows hold 59% of Sync's requests; Table 26 gives the read-only shares inside and outside the windows. A reread is a read-only request whose files, as the parser reads them, were all read or written earlier in the continuation or, for Sol, by the root before the saved state, whether or not the file changed in between; files written through scripts are not parsed. Table 26 gives Sync minus Limit-only per state with the registered averaging. These counts were defined after the results were known. The released per-request labels identify each request by run and position, which indexes the redacted trajectories (Appendix J).

R.1 Content of Read-Only Requests

The parser of Appendix R reads commands and paths, not the content they return. To see what read-only requests returned, we sampled them from all 288 Sync and Limit-only continuations of Sol at 32K and 64K, in which 7,899 of 15,517 agent requests are read-only (Table 27). We drew 20 requests per policy and threshold, spread nearly equally over tasks and at random within each, with a fixed seed and recorded inclusion probabilities. The 80 requests hold 82 tool calls from 23 of the 27 roots.

Two new gpt-6.1-sol sessions and one new claude-opus-5-5 session labeled every call independently from the same package: the request, the run's earlier records, and the summary and context in force, without policy, threshold, cost, outcome, or each other's labels. For each call they judged whether the returned content was identical to an earlier recorded output, updated, first seen, or undecidable; the extent of any match (whole file, fragment, or command output); and whether the agent was explicitly checking a result, each with a pointer to the evidence in the case. A label stands if at least two judges give it and its cited evidence holds; otherwise the call is undecided. A request is a check if its content had changed or it explicitly checked a result; otherwise identical-text retrieval if its content was identical; otherwise new if first seen; otherwise undecided. The three judges gave 65 of the 80 requests the same class, including requests all three left undecided.

Every identical match was to a recorded fragment; no re-

Table 26. Added agent requests, S minus L per state from the same state. Sol: root-cluster 95% intervals. GLM, counts only: means over the nine first-crossing states (tasks weighted equally) and over the six Coq and the three Retro later states (source runs, then tasks, weighted equally). Bottom rows: share of S's own requests that only read, within five after a switch and elsewhere, pooled over requests.

	Sol			GLM 64K		
	32K	64K	Pooled	First	Coq	Retro
Agent requests	113.7 (89.9, 141.2)	20.0 (17.7, 24.7)	65.8 (54.0, 79.5)	−2.9	−38.4	759.0
Read-only	84.9 (66.4, 105.9)	11.2 (10.1, 13.6)	47.5 (38.2, 58.0)	25.0	13.7	343.7
Rereads	52.3 (39.8, 65.9)	3.4 (3.0, 4.4)	27.7 (21.4, 34.5)	8.8	3.6	220.2
Writing	4.9 (2.4, 8.4)	2.3 (1.6, 3.2)	3.4 (2.0, 5.1)	12.3	−6.0	83.7
Run	6.8 (5.4, 8.3)	2.0 (1.3, 2.8)	4.3 (3.5, 5.1)	−7.6	−59.7	215.1
Wait	2.5 (1.8, 3.4)	0.8 (0.4, 1.5)	1.6 (1.2, 2.1)	−30.6	0.0	0.0
Other	14.6 (12.0, 17.1)	3.7 (2.7, 5.4)	8.9 (7.5, 10.2)	−2.1	13.6	116.6
Within five after a switch	78.5 (63.7, 95.6)	9.9 (9.2, 11.5)	44.1 (36.8, 52.5)	25.9	18.0	146.7
Read-only	59.4 (46.6, 74.6)	6.5 (6.0, 7.8)	32.8 (26.3, 40.3)	9.5	8.9	82.8
S read-only share, within five	76%	67%	75%	37%	50%	56%
S read-only share, elsewhere	49%	23%	42%	19%	20%	35%

Table 27. Content of Sol's read-only requests. Sampled: counts among 20 requests per column; none was labeled new. Weighted: point estimates of requests per continuation, tasks weighted equally, without intervals; at S32 two tasks carry most of the estimate (see text).

	L32	S32	L64	S64
Continuations	81	81	63	63
Read-only requests	87	6,965	33	814
Sampled: identical text	9	6	3	4
Check	2	7	9	7
Undecided	9	7	8	9
Weighted: identical text	0.39	54.70	0.06	2.31
Check	0.04	8.59	0.23	3.06
Undecided	0.65	22.70	0.18	6.29

quest returned a file read completely earlier, unchanged. No request counted as new, because a case does not carry a search of the whole history, so content not found earlier stays undecided. Whether the summary held the retrieved content was partial or undecided for 80 of the 82 calls, so the audit does not say which requests an omission from the summary caused.

Table 27 also scales the sample to all read-only requests: each task's counts are weighted by inverse inclusion probability and divided by its continuations, and tasks are weighted equally, a task without read-only requests counting zero. These are point estimates without intervals. They put Sync's excess of identical-text retrieval at 54.3 requests per continuation at 32K and 2.25 at 64K. At 32K, two tasks with 1,608 and 2,700 read-only Sync requests contributed two sampled requests each, all identical-text retrieval, and account for 53.2 of Sync's 54.7. The frame is every read-only request, not the rereads of Table 26, so these estimates are not reread counts.

S Capped Keep-2 at 32K

Table 28 describes the compactions of Sol's 32K continuations, pooled over compactions, next to each policy's same-state difference from Limit-only. Capped Keep-2 keeps the last two complete steps unless they exceed 20% of the threshold, 6,400 tokens. Against Keep-2 from the same states, Capped Keep-2 cost \$0.20 less (−0.42 to 0.01), completed −0.1 requests (−8.7 to 8.4), took 342 s less (−757 to 67), and its pass rate differed by 0.00 (−0.07 to 0.07).

T Later Same-State Runs

Two later studies continue saved Sol states on a later runtime, which differs from that of the original continuations in three ways: requests went to the cache-reporting endpoint of Appendix M instead of the original one; each continuation began with one warming request that repeated the root's last input under its own cache key; and the repetition guard also counts polls of command handles that no command returned (Appendix W). The model (gpt-6.1-sol, reasoning effort max), the 64K threshold, the source run's remaining time, the tools, and the verifier are unchanged. Costs are priced as in the replay: reported cache reads at the read price, other input at the uncached price, and output at the output price; warming requests are left out. The later complete runs (Appendix O.1) use this runtime without the warming request.

T.1 Other Summary Prompts

The rerun of Section 5 takes one 64K state per task, fixed before the runs as the lowest-numbered root that crossed 64K: root 1 on seven tasks and root 2 on Sound; Coq has no 64K state. Each state is continued three times under four policies, 96 continuations in all: Limit-only; Sync; Sync with the two notices of Appendix I replaced by the continue

Table 28. Compactions in Sol's 32K continuations. Context after the switch is the input of the first request that uses the summary; steps of use count accepted agent steps until the next summary or the end. The last three rows are same-state differences from L.

	S	S+2	K	KB	A
Compactions per continuation	15.8	15.8	9.6	8.4	6.9
Kept two / one / no steps	–	–	520/0/0	320/103/30	–
Median context after switch	7.5K	7.7K	13.6K	11.9K	17.8K
Above half the threshold	0%	1%	22%	3%	65%
Median steps of use	8	8	6	7	6
Replaced after one step	0%	1%	7%	1%	3%
Cost ($, v2)	2.09	2.09	0.98	0.78	0.80
Requests	129.5	126.0	46.0	45.9	29.1
Success	−0.04	−0.04	−0.07	−0.07	−0.07

cue given there; and the same with the instruction that ends every summary request replaced by Codex's compaction prompt at commit `ba2b67f` (OpenAI, 2026b); when the history is serialized into that final user message, only the instruction before it changes. Both changes apply at every summary, not only the first. The system prompt, the history sent for summarizing, the output allowance, the reasoning effort, and the summary handling stay as in Sync, and no target length is added to Codex's prompt.

All four policies ran on the later runtime. Up to 16 continuations ran at once, in an order shuffled within each state and repetition.

The eight states are the units. Each value averages a state's three continuations and then weights the states equally; with one state per task, the resampling of roots within tasks used elsewhere does not apply, so Table 30 gives every state instead of intervals. Within a state, the largest of the three continuations' request counts exceeded the smallest by a median of 3.5 to 7.5 over the states, depending on the policy. Table 29 gives the means. Onto and Prod failed in every continuation under every policy, and the other six tasks passed in all. Sync made 12.3 more agent requests than Limit-only and 1.6 summary requests, more requests at all eight states, and took 2.9 minutes longer, longer at seven. One Sync continuation at Risk lost a response without token counts, continued, and passed; no other continuation lost one. Summing Sync's known costs gives $0.52 per continuation, a lower bound. Over the seven states with complete counts, Sync cost $0.11 more than Limit-only, more at six. Replacing the notices with the cue changed Sync by -0.8 requests and $-\$0.025$ (seven states), and Codex's prompt changed the cue variant by $+0.5$ requests and $+\$0.002$; both differences changed sign across states. Against the cue variant, Codex's prompt shortened the summaries from 4,319 to 3,071 output tokens (29%; each continuation's mean per summary, averaged over its state's three continuations and then over the states) and their cost from $0.10 to $0.08 per continuation, but not the continuations' cost. Against the original continuations from the same states, Limit-only made about as many requests and Sync fewer, so the rerun's excess is smaller.

Table 29. Rerun from eight 64K states, three continuations per state and policy; task-macro means per continuation. S-cont: S with the continue cue in place of the notices; S-Codex: S-cont with Codex's compaction prompt. Pass: continuations with reward 1. Agent and Summ.: agent and summary requests. Time: agent time (minutes). Cost (USD): reported cache reads at Sol's listed prices; S's is a lower bound. Summary: output tokens per summary, reasoning included. The last two rows are the original continuations from the same states, with cache reads measured by replay.

Policy	Pass	Agent	Summ.	Time	Cost	Summary
L	18/24	12.6	0	10.0	0.40	–
S	18/24	24.8	1.6	13.0	≥0.52	4,387
S-cont	18/24	24.1	1.6	13.1	0.51	4,319
S-Codex	18/24	24.6	1.7	12.4	0.51	3,071
L, original	18/24	11.6	0		0.36	–
S, original	18/24	29.2	1.8		0.62	

T.2 Additional Tasks

Six Terminal-Bench tasks outside our nine were fixed before any Sol run (Table 31): four whose GLM runs had grown past 64K, and two for other kinds of work. Each has two Limit-only roots, run from the task's start. A root's state is saved at its first 64K crossing, as in Section 5; if a command is still running there, the root has no state, and no later point replaces it. Each state is continued twice under Limit-only and twice under Sync on the later runtime. Averages take a state's two continuations per policy, then a task's roots, then the tasks with equal weight. Sync made 16.45 more completed requests per continuation than Limit-only and 2.0 more summaries. One Limit-only continuation at `pretrain-shard-corruption` lost a response without token counts, so that task's cost difference is unknown.

Table 30. The rerun per state: mean requests (summaries included) and cost (USD) of three continuations. S's cost at Risk is a lower bound.

	Requests				Cost			
State	L	S	cont	Codex	L	S	cont	Codex
Vig 1	7.3	19.3	21.0	18.7	0.17	0.32	0.33	0.33
MP 1	2.3	13.7	10.3	8.3	0.07	0.22	0.16	0.13
Onto 1	12.0	12.7	10.3	18.7	0.35	0.20	0.16	0.34
Prod 1	6.7	16.0	14.7	17.3	0.27	0.42	0.37	0.38
Retro 1	18.3	48.7	45.7	50.3	0.62	0.98	0.95	1.01
Risk 1	7.7	22.7	18.3	20.0	0.30	$\geq$0.54	0.58	0.54
RS 1	45.3	75.0	83.0	73.7	1.43	1.46	1.45	1.32
Sound 2	1.0	3.7	2.3	3.0	0.01	0.06	0.05	0.03

U Registered Sol Predictions

The six predictions and the analysis rules were frozen on 2026-10-08 at 14:13 UTC, before the first formal Sol request, and were not changed afterwards; a later addendum on the same day added the interior-prefix cost account. The freeze was recorded in our repository and in a record that lists all 162 formal runs as not yet started. The artifact provides anonymized copies of the initial and addendum freeze records, with content hashes of the referenced files in their recorded snapshots. By the registered rule, a prediction matches when its point estimate lies strictly on the predicted side; intervals are reported beside it and not tested (Table 32). All six point estimates lay on the predicted side; for P2 the interval spans the cutoff, so that share is not distinguishable from one half. P5 compares the same-state difference with the per-summary sum, not with the fixed-trace estimate, which was defined after the results (Appendix Q).

V Sol First-Action Annotation

For each matched Sync/Sync+2 pair (144 pairs), we take the first tool call after the first summary is used. The frozen parser of Appendix H labels inspection actions read-only and work, polling, cancellation, evidence reads, and responses without a tool call not read-only. It could not decide 83 actions, mostly runs of programs and scripts. These, mixed with 45 parser-labeled controls, went to three annotators in separate blinded sessions, as for GLM: two new delegated sessions requested as `gpt-6.1-sol` and a Claude Code subagent requested with the `opus` alias (`claude-opus-5-5`), each seeing only the action text and the codebook of Appendix H. An action counts as classified only if all three agree: 94 read-only, 18 not read-only, and 16 unresolved. The three agreed on 112 of the 128 actions; in each of the 16 others, one `gpt-6.1-sol` session marked as unresolved, mostly because it calls a program whose effects are not visible, an action the other two labeled read-only. All 45 controls matched the parser. With these labels, 96.5% of Sync's and 92.4% of Sync+2's first actions are classified. By threshold, the difference was -0.19 at 32K and -0.34 at 64K. The cautious variant of Appendix H gives -0.22, with bounds of -0.28 to -0.15. The second and third actions, labeled by the parser alone, are classified below the 80% criterion and not reported.

GLM's 24 first-crossing pairs (Appendix W) went through the same procedure after the runs; it was not registered for GLM. The parser could not decide 18 first actions. With 30 parser-labeled controls, they went to two new `gpt-6.1-sol` sessions and a new `opus` subagent, each seeing only the action text and the codebook. All 30 controls matched the parser. Of the 18, the three agreed on 8 read-only and 7 not read-only; 3 stayed unresolved, two of them actions that both `gpt-6.1-sol` sessions could not decide and the subagent labeled read-only. With these labels, 23 of Sync's and 22 of Sync+2's 24 first actions are classified.

W GLM Runs

Table 33 lists the GLM runs used in Sections 5 and 6. Ten complete runs reached the eight-hour limit, four under Mask, three under Capped Keep-2, two under Limit-only, and one under Sync, and count as failures with the cost they incurred. Calls that ended without reported usage, 107 of 38,470 in the 64K runs, 32 of 26,974 in the later-state continuations, and 65 of 12,491 in the first-crossing continuations, are left out of the costs.

First-crossing continuations. Nine new Sync runs at 64K, one per task, were each saved at the first request whose prepared input crossed 64K, before any summary, and then stopped; these source runs are not scored. Each continuation keeps the rest of its source run's eight-hour budget: 3.5 hours on Sound, whose source took 4.5 hours to cross, and 7.1–7.8 hours elsewhere. Eleven continuations reached that limit and count as failures with the cost they incurred: on MP three Sync, two Sync+2, and one Limit-only; on Sound two Sync+2, one Limit-only, and one Sync; on RS one Limit-only. Continuations stopped by the runtime's repetition guard or ending with background commands still running keep their verifier reward. At Retro, the context first crossed 64K at the fourth request, and the last two steps held 97K of its 100K tokens (runtime estimate), so the runtime declined Sync+2's replacement, which would not have shrunk the context, in all three repeats. Carrying Sync's work, Limit-only would have held at most 0.78M tokens, within GLM's 1M-token window, at every state. Sync+2 cost $0.05 more than Sync on average and made 17 more requests, with differences in both directions across the eight states where it was defined; it passed 10 of 24 continuations against Sync's 12, fewer on RS and Sound and more on MP. After the first switch, the share of

Table 31. Additional tasks: roots that gave a 64K state (of 2), completed requests and summaries per continuation, passed continuations, and Sync minus Limit-only cost per continuation (USD, reported cache reads at Sol's listed prices).

Task	States	L req.	S req.	S summ.	Passed (L, S)	ΔCost
`pretrain-shard-corruption`	2	17.8	24.3	1.75	4/4, 4/4	unknown
`vf2-speedup-networkx`	2	13.0	33.8	1.75	4/4, 4/4	+0.21
`telecom-entity-resolution`	2	24.0	72.0	4.50	4/4, 4/4	+0.65
`fin-saccr-rwa`	2	5.5	14.0	1.00	4/4, 4/4	+0.10
`data-anonymization`	1	7.0	5.5	1.00	0/2, 0/2	−0.03
`wal-recovery-ordering`	0	–	–	–	–	–

Table 32. Registered predictions and results. Proportions are task-macro means; differences are per state. Intervals are 95% bootstrap intervals; the last column says whether the interval lies on the predicted side of the cutoff.

	Prediction (v2 unless stated)	Side	Result	Interval
P1	S32 summaries that repay	< 0.20	0.00 (0 of 1,256)	on side
P2	S64 summaries that repay, complete runs	< 0.50	0.49 (0.38, 0.65)	spans
P3	Same, no cache and provider counts, at 32K and at 64K	> 0.50	$\geq$0.998 in all four	on side
P4	Same-state requests, S−L	> 0	74.7 (61.4, 90.1)	on side
P5	Same-state difference minus per-summary sum	< 0	-0.76 (-1.05, -0.53)	on side
P6	Read-only first actions, S+2−S	< 0	-0.25 (-0.29, -0.21)	on side

Table 33. GLM runs at 64K.

Set	Runs	Tasks
S, H, L	27 each	all nine
KB	27	all nine (MP, Vig later)
K	21	all but MP and Vig
A	12	Onto, Prod, Risk, Sound
First crossing: L, S	27 each	9 states, one per task
First crossing: S+2	24	8 states (not Retro)
Later states: L, S	27 each	9 states (Coq 6, Retro 3)
Later states: S+2	24	8 states (not the first Coq state)

first tool actions that only read existing state was 78% for Sync and 59% for Sync+2. Where Sync made fewer requests than Limit-only, Limit-only's own continuations ran long. At RS, one Limit-only continuation polled 152 command handles that no command had returned, 957 times in all and one of them 696 times, each time receiving an unknown-handle error, until it reached its time limit; the repetition guard counts only polls of real commands and did not stop it. It cost $14.32, against $2.20 and $2.94 for the other two, and every Sync continuation at RS cost under $0.75. Without this continuation, Limit-only's mean at RS is $2.57, and Sync's average saving over the nine states falls from $1.20 to $0.76 at GLM's prices; Sync still costs less at eight states. Leaving out one task at a time, Sync's average saving stays between $0.61, without RS, and $1.35. No continuation polled a command started before its saved state, which holds no running command. A later rerun, with the repetition guard also counting polls of handles that no command returned, continued the same RS state three times under each policy. No continuation polled invented handles at length, the guard stopped none, and none passed.

Two of the three Sync continuations used a summary at the first switch; in the third, the first summary failed the runtime's quality check, and Sync first used one 15 steps later. Sync sent 1,079 requests and Limit-only 773; Limit-only's continuations sent at least 284M input tokens and Sync's at least 32M. Eight failed requests lack token counts, so we do not price these continuations. At Coq, Limit-only ran programs in 123 more requests than Sync. Table 34 gives each state; averages weight the continuations within a state and the tasks equally.

Table 34. GLM first-crossing continuations at 64K, per task: mean cost per continuation at GLM's prices, mean agent requests, and passes of three.

	Cost ($)			Requests			Passed		
Task	L	S	S+2	L	S	S+2	L	S	S+2
MP	0.88	0.36	0.49	147	236	299	2	0	1
Vig	0.15	0.14	0.10	45	110	65	3	3	3
RS	6.49	0.54	0.66	539	331	395	0	1	0
Coq	2.54	0.36	0.34	315	214	210	3	3	3
Retro	1.99	0.51	–	226	336	–	0	0	–
Risk	0.56	0.14	0.19	84	85	113	3	3	3
Sound	0.34	0.11	0.25	57	57	93	0	2	0
Prod	0.04	0.03	0.03	11	16	11	1	0	0
Onto	0.04	0.04	0.04	12	25	26	0	0	0

Later-state continuations. The nine later states come from three earlier Sync runs at a 64K threshold, two on Coq and one on Retro. Limit-only continues without ordinary compaction from a history that, except at the first Coq state, was already compacted. Sync+2 receives the first summary of its matched Sync continuation plus the

last two complete steps; its cost and time include that summary, which Sync generated. At the first Coq state, the last two steps held 84K of the 87K-token context, so Sync+2 would have replaced only about 900 tokens of history with one of Sync's summaries, which held 2.4K–2.9K tokens. The runtime declines a replacement that does not shrink the context, and did so before any model call in all three repeats; Sync+2 is therefore not defined at this state. Averages weight continuations within a state, states within a source run, source runs within a task, and the two tasks equally. Requests include summary requests, as for Sol. Every Coq continuation passed and every Retro continuation failed. At the three Retro states, Sync took 3.5 times as long as Limit-only. At the middle one, Sync compacted 31 times, and after a switch it typically re-read the processor and video sources or re-ran the simulator against the reference to find where it had left off. Each of Limit-only's requests there, carrying its long context, cost 4.3 times one of Sync's, so at GLM's prices Sync cost $0.10 more than Limit-only, and at DeepSeek's 3.1 times as much. Over the nine states at GLM's prices, Sync cost $0.79 less than Limit-only on average, and 326 of its 330 summaries repaid by the per-summary account. At DeepSeek's prices, Sync cost $0.82 more on average and still less at five of the six Coq states. On Limit-only's baseline, the Coq states lie three above and three below equality; at Retro the estimate of $13–32 prices work Limit-only could not have done. At Retro, Sync+2 made a third fewer requests than Sync and cost $0.54 less; at Coq it matched Sync.

X Remaining Work and a Deployment Check

Section 5 compares the two models at one price vector. We take the first 64K state of every root on the eight tasks both models reached at 64K (Sol: 21 states; GLM: 8; Sol's Coq roots finished first, so GLM's Coq is plotted but not counted) and price every continuation, summaries included, at USD 1, 0.05 and 5 per million uncached, cached and output tokens, without input tiers or cache-write charges. Failed or cancelled calls without reported token counts are left out of both arms: 10 for Sol and 38 for GLM, 16 of them in GLM's Limit-only continuations of RS. Each model keeps its own tokenizer and cache allocation, so a common price removes the price difference, not every difference between the models. Remaining work counts Limit-only's completed requests from the saved state until the continuation ended or reached its limit. Draws are averaged within a root and roots within a task.

Figure 9 plots Sync's cost relative to Limit-only's against that count. The rank correlation is -1.00 within Sol, -0.74 within GLM, and -0.94 over the 16 model–task pairs. The count and Limit-only's cost come from the same draws, so we repeated the calculation with the count from one Limit-only draw and the cost from the root's other two: the pooled correlation was -0.79, -0.95 and -0.95 for the three choices of draw. By the fixed-trace estimate of Section 2, added work exceeded the fixed-work saving ($D > F$) at all eight Sol tasks and at GLM's Prod, Onto and Vig; the observed difference G does not depend on that estimate. At a cache-read price of 0.20 the pooled correlation is -0.97 and the crossing moves to shorter remainders: GLM's Prod (11 requests) and Vig (45) and Sol's RS (46) then cost less under Sync.

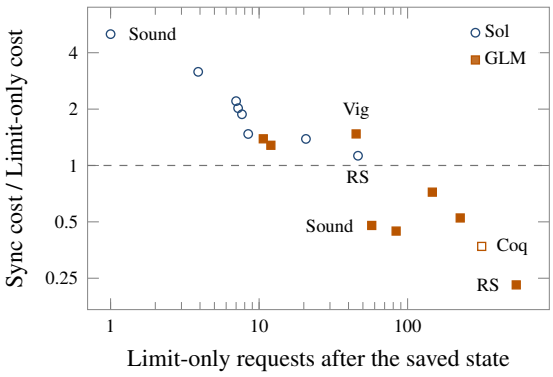


Figure 9. Sync's continuation cost relative to Limit-only's from the first 64K state, against the requests Limit-only made after that state, per task, at common prices (USD 1, 0.05 and 5 per million uncached, cached and output tokens). Circles: Sol; filled squares: GLM on the same eight tasks; open square: GLM's Coq. Below the dashed line, Sync cost less. Both axes are logarithmic.

X.1 A Deployment Check

Cost is linear in prices, so a continuation's cost at a flat price list is $N_u p_u + N_c p_c + N_o p_o$. From six flat price points, an uncached price of 1 with cached prices of 0.05, 0.20 and 1 and output prices of 5 and $10/3$ (a rank-3 system), we solve these token totals for Limit-only, for Sync, and for Sync's requests with the removed history added back ($\mathrm{cost}_L(S)$) at every first 64K state (Sol: 21; GLM: 9); the totals reproduce all six price points to within 10^{-6}. With them, G, F and D follow for any flat price list, and each state falls into one of three cases, by its net saving first: $G > 0$, Sync cheaper; $G \le 0 < F$, where added work exhausts the fixed-trace saving; or $G \le 0$ and $F \le 0$, where the summary and switch are not repaid even with fixed work. Since D can be negative, $F \le 0 < G$ occurs, at two GLM states below a cached price of about 1% of uncached, and counts as Sync cheaper. The artifact includes the totals and a script that prices them. Flat prices omit Sol's cache-write premium and long-input tier, so Sol's numbers here differ slightly from Section 5.

At flat list prices, Sol's F averaged $0.17 per state, enough

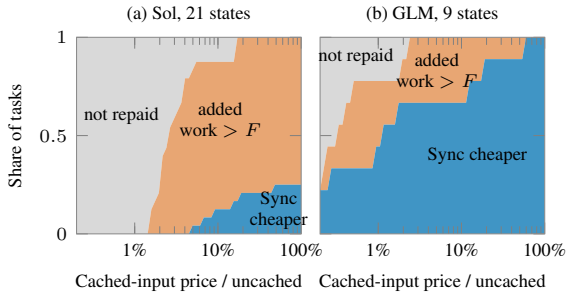


Figure 10. Share of tasks in each case at the first 64K state, against the cached-input price as a share of the uncached price, with output priced at five times uncached input. Blue: Sync cheaper ($G > 0$). Orange: the fixed-trace estimate favors Sync, but added work exceeds it ($G \leq 0 < F$). Gray: the summary and switch are not repaid even with fixed work ($G \leq 0$, $F \leq 0$). States are weighted so that each task counts once.

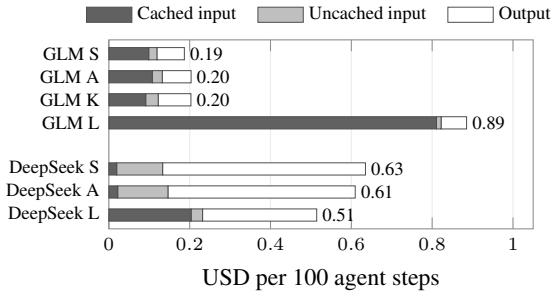


Figure 11. What the steps paid for: cost per 100 agent steps at the list prices of Table 3, split by token type and pooling each policy's agent and summarization requests over its runs; DeepSeek's L covers the four tasks with S/A pairs.

for about five of Limit-only's requests at their average cost; Sync made 20 more agent requests, and 18 of the 21 states fall in the middle case (two in the last, one with $G > 0$). GLM's F of \$1.28 would have paid for about 140 such requests; Sync made three fewer, and eight of the nine states have $G > 0$. Figure 10 varies the cached-input price. For Sol, below about 1.5% of the uncached price even fixed work does not repay; above 4%, 70–83% of the task weight stays in the middle case, and Sync is cheaper at no more than a quarter of it even without any cache discount. For GLM, Sync is cheaper at two thirds of the tasks from about 2% and at all of them above 60%, and at Sound and Risk even at the lowest cached prices, because its changed request sequence cost less there ($D < 0$); the middle-case tasks in between are the three whose Limit-only continuations ended soonest (Appendix X).

Choosing before the work is known. A runtime has to choose at the crossing. On held-out tasks (leaving one task out within a model) we compare always Limit-only, always Sync, one arm per model and price chosen on the other tasks, a threshold on one feature known at the crossing fitted on the other tasks, and the cheaper arm per state from observed means, which is optimistic. The features are the requests completed before the state, the agent time elapsed, and the input added per request; across tasks, their rank correlations with Limit-only's remaining requests lie between -0.38 and $+0.17$. Table 35 reports the cost per state. One arm per model and price came within \$0.002 per state of the per-state choice at list prices and within \$0.045 at the other prices. The feature rules matched it or cost more in 17 of 18 comparisons; the exception, elapsed time for Sol at a read price of 20%, cost more at the other five prices.

Y Cost Composition and Context Trajectories

Figure 11 splits the cost per agent step by token type, and Figure 12 shows context size over GLM Coq runs at 64K and 32K. Summarization accounted for 3–4% of GLM's Sync and Async cost at 128K, 7–9% at 64K, and 18–23% at 32K.

Table 35. Cost per state on held-out tasks (USD at the listed flat prices per million uncached, cached and output tokens; tasks weighted equally). Feature: the range over the three one-feature rules. Per state: the cheaper arm from observed means.

Model	Prices	Always L	Always S	By model	Feature	Per state
Sol	2 / 0.10 / 10	0.432	0.660	0.432	0.454–0.470	0.431
GLM	0.15 / 0.03 / 0.5	1.448	0.248	0.248	0.248–0.274	0.248
Sol	1 / 0.05 / 5	0.216	0.330	0.216	0.227–0.235	0.215
GLM	1 / 0.05 / 5	3.046	1.205	1.205	1.205–2.235	1.173
Sol	1 / 0.20 / 5	0.396	0.472	0.396	0.353–0.443	0.351
GLM	1 / 0.20 / 5	9.875	1.917	1.917	1.917–2.102	1.914

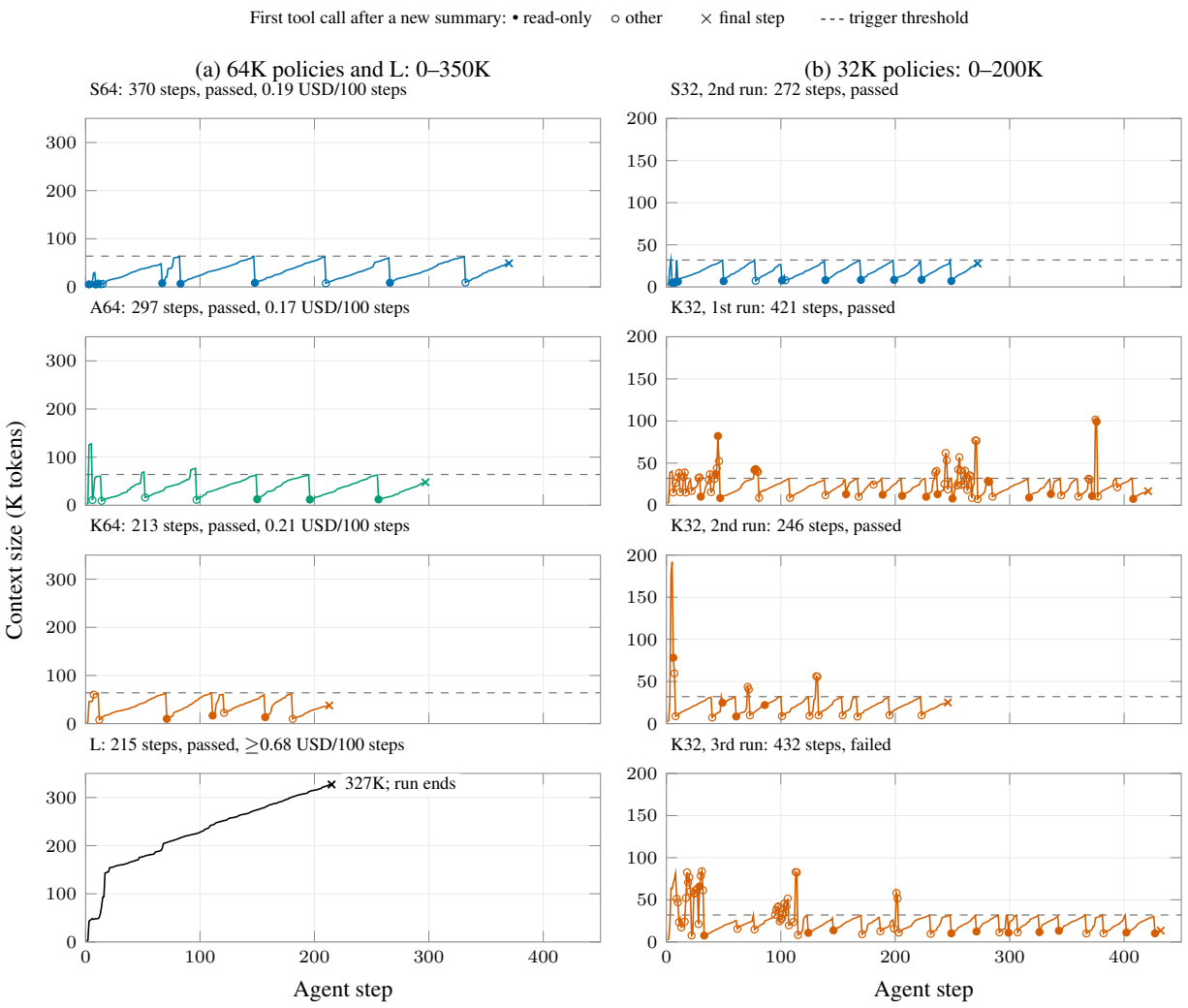


Figure 12. Runtime-estimated context at each GLM Coq agent step, about 15% above the provider's reported input. Axes are linear, with a common y range within each column. Circles mark first use of a new summary; their fill classifies its first tool call, as in the legend. (a) The first S, A, and K runs at 64K and the first L run; all passed. L ended after 215 steps at 327K without compaction. $\geq$ marks a cost lower bound where a failed request reported no usage. (b) The S32 run that compacted and all three K32 runs; the other S32 run failed after 12 steps. K's circle clusters are summaries replaced after a single step, mostly when the two kept steps were large (Section 5).

Z Comparison with Related Work

Table 36 compares the settings and measurements of the closest studies with ours.

Table 36. How this work differs from the closest studies. ✓: measured or modeled; (✓): partly; –: not reported in the paper. Rep.: repeated compaction within one long agent run; Cost: money at provider prices (cache: with cached-input prices; (cache): for some settings); Out.: task outcome; Beh.: actions after compaction; Cont.: how far runs continued from a matched compaction state; Time: the wait or latency that compaction itself adds, not overall task time; Pay.: per-compaction repayment accounting; Pred.: that accounting predicted before the outcome and checked on held-out tasks.

Study	Setting	Context operation	Rep.	Cost	Out.	Beh.	Cont.	Time	Pay.	Pred.
Complexity Trap (Lindenbauer et al., 2025)	SWE-bench V.	masking, summary, hybrid	✓	(cache)	✓	(✓)	–	–	–	–
Fan et al. (Fan et al., 2026)	SWE-bench V., TB 2.1	5 strategies, 4 budgets	✓	✓	✓	(✓)	–	–	–	–
Slipstream (Chen et al., 2026)	SWE-bench V., BrowseComp	sync / async / validated	✓	–	✓	–	–	✓	–	–
Parallel CC (Cim et al., 2026)	HotpotQA, LoCoMo	parallel summarization	✓	–	(✓)	–	–	✓	–	–
Liu (Liu, 2026)	planning env., ALFWorld	drop / fact-preserving	(✓)	–	✓	✓	24-turn episode	–	–	–
TRACE (Min et al., 2026)	AppWorld; complete tasks, 50-step cap	compression prompts	✓	–	✓	✓	$\leq$5 actions	–	–	–
Weinberger & Hozez (Weinberger and Hozez, 2026)	Claude Code; coding tasks, SWE-bench Go	tool-output reduction	–	cache	✓	✓	–	–	–	–
Lumer et al. (Lumer et al., 2026)	agent sessions, 3 providers	cache placement	–	cache	–	–	–	(✓)	–	–
CAPC (Song, 2026)	assistants, RAG, τ-bench	static-prefix compression	–	cache	✓	–	–	–	(✓)	(✓)
This work	9 main + 5 additional Terminal-Bench tasks, 3 APIs, 2 self-hosted models	S, A, K, KB, H, M10, M10+S, L	✓	cache	✓	✓	to task end; repeated com-paction	✓	✓	✓