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.

Break Later or Never: Prefix-Preserving Context Compaction for Coding Agents

Musa Cim

mtc5693@psu.edu

Pennsylvania State University

Burak Topcu

Pennsylvania State University

Mahmut Kandemir

Pennsylvania State University

Research Article

Keywords: LLM serving, prefix caching, KV cache, LLM agents, context management, performance characterization, coding agents, SWE-bench

Posted Date: November 11th, 2026

DOI: https://doi.org/10.21203/rs.3.rs-11684582/v1

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

Additional Declarations: The authors declare no competing interests.

Break Later or Never: Prefix-Preserving Context Compaction for Coding Agents

Musa Cim, Burak Topcu, Mahmut Taylan Kandemir
The Pennsylvania State University
University Park, PA, USA
{mtc5693, bvt5283, mtk2}@psu.edu

***Abstract*—Coding agents repeatedly invoke LLMs over growing, largely append-only histories (each request repeats the previous one and appends a new turn), making them natural beneficiaries of prefix caching. Yet the context compaction used to control this growth can undermine that benefit: rewriting earlier tokens invalidates reusable prefix KV and forces a costly re-prefill of the suffix (i.e., recomputing the KV of every token after the edit). We characterize this interaction across two models and two SWE-bench variants, finding that uncompacted agent trajectories achieve 98 to 99% prefix-cache reuse and that tool observations (e.g., file views and test logs) account for the largest share of context cost. As a simple prompt-level mitigation, *batched observation masking* (masking old observations once every few steps instead of every step) preserves 95 to 98% prefix reuse while reducing context length by up to 50%, serving cost by up to 18%, and task completion time by up to 21%. We then present *ObsDrop*, which moves compaction from the prompt into the KV cache: stale tool-observation KV is released while the logical token prefix remains unchanged, preserving prefix matching while reducing KV residency and attention work. We implement ObsDrop in SGLang and Sparse-vLLM and evaluate it across multiple attention backends and concurrent agent workloads. ObsDrop maintains task quality while improving cache residency, throughput, and tail latency under memory pressure. Its benefits grow with concurrency and KV pressure, reaching 2.45 to 2.85× higher throughput with FlashInfer (and up to 5.37× with the Triton backend under peak load), relative to no compaction. Our results show that effective context management for agents benefits from considering not only which tokens are retained but also how compaction interacts with the serving system. https://github.com/musacim/ObsDrop**

***Index Terms*—LLM serving, prefix caching, KV cache, LLM agents, context management, performance characterization, coding agents, SWE-bench**

I. Introduction

Coding agents repeatedly invoke an LLM over a growing history of reasoning, tool calls, and tool *observations* (the outputs that tools such as file viewers and test runners return) [1]. This workload appears unusually friendly to modern LLM serving: each request largely appends to the previous one, allowing *prefix caching* (reusing the key–value (KV) state already computed for an identical earlier prefix) to skip most of the work [2]. At the same time, long agent trajectories accumulate tens of thousands of context tokens, increasing KV-cache pressure and attention costs [3], [4]. Agent frameworks, therefore, compact their histories through mechanisms such as truncation, observation masking, and summarization [5]–[7]. These two optimizations, however, can work against each other: (1) compaction saves context by rewriting history, while (2) prefix caching saves computation only while that history remains unchanged.

We characterize this interaction using coding-agent workloads across two models and two SWE-bench variants. Without compaction, successive agent requests achieve 98–99% prefix-cache hit rates (§IV), as nearly the entire conversation history is reused across requests. Tool observations, especially file contents and test results, make up much of the accumulated context. Removing stale observations can therefore reduce context size, but prompt-level compaction disrupts prefix-cache reuse: modifying earlier tokens breaks prefix matching at the first edit, forcing the remaining suffix to be prefilled again. Consequently, a shorter prompt can be more expensive to serve. This tradeoff becomes more pronounced under concurrent load: retaining the full history increases KV-cache pressure and can exhaust available capacity, whereas frequent compaction increases redundant prefill computation, potentially shifting the performance bottleneck toward prefill [8]–[10].

Compaction should therefore account for not only what is removed, but also when and where the history changes. We first explore a simple prompt-level mitigation: *batching* observation masking (§VIII), i.e., masking old observations once every k' turns instead of after every turn. Across our workloads, it maintains 95 to 98% prefix-cache reuse, reduces mean context length by up to 50%, serving cost by up to 18%, and task completion time by up to 21%. This provides a practical option that requires no inference-engine changes, but it cannot eliminate the underlying conflict: whenever compaction rewrites the prompt, some reusable prefix state is lost.

Motivated by this, we present and evaluate *ObsDrop,* a KV-level compaction mechanism that decouples the agent's *logical history* (the token sequence it sends) from its *physical KV residency* (the KV entries the engine keeps in GPU memory). ObsDrop keeps the prompt and token positions unchanged for prefix matching but releases the KV entries corresponding to stale tool-observation contents and excludes those positions from attention. The next request can therefore match the complete previous prompt and prefill only newly-appended tokens while consuming KV memory and attention work only for the retained context. We implement ObsDrop in SGLang's RadixAttention cache and, independently, in Sparse-vLLM

(§X).

In our evaluation, ObsDrop substantially reduces live KV and attended-token volume while maintaining task success close to no compaction. As concurrency increases, its smaller physical footprint keeps more agent prefixes resident, avoiding the cache-eviction and re-prefill cycle that causes tail latency to rise sharply without compaction. Across SGLang and Sparse-vLLM, as well as multiple attention backends, ObsDrop provides modest gains at low load that grow as memory and attention pressure increase, reaching 2.45 to $2.85\times$ throughput gains with FlashInfer and a peak of $5.37\times$ with the Triton backend at 16 agents, relative to no compaction.

In summary, this paper makes the following **contributions**:

- **Characterization.** We characterize the interaction between context compaction and prefix caching in coding agents, showing that naturally high prefix reuse makes prompt edits an important and previously overlooked component of compaction cost (§IV).
- **Prefix-aware prompt compaction.** We show that batching observation masking is a simple way to amortize prefix invalidations, preserving 95 to 98% cache reuse while reducing context, serving costs, and task completion time (§VIII).
- **ObsDrop.** We design and implement a KV-level observation-dropping mechanism that reclaims stale observation KV while preserving the logical prefix used for cache matching, with independent implementations in SGLang and Sparse-vLLM (§X).
- **End-to-end evaluation.** We evaluate these techniques across various models, benchmarks, serving engines, attention backends, and concurrent agent workloads, and we show that prefix-preserving compaction improves KV residency, attention work, throughput, and tail latency while maintaining task quality (§X).

Effective agent context management benefits from considering *both* context reduction and the reuse of cached inference state.

II. Background

A. The Agent Loop as an Inference Workload

Coding agents operate in long-running loops, invoking the model again as each new observation is added to the context (reason, act, observe, repeat) [11], which makes them unusually inference-intensive. A recent study of SWE-bench Verified trajectories reports that agentic coding tasks consume roughly $1000\times$ more tokens than code reasoning or code chat, with input tokens rather than output tokens accounting for most of the cost [12]. The scale is already visible in production: by August 2026, the median researcher at one provider was using more than $600 per day of coding-agent inference at API prices, the 90th-percentile user exceeded $7,000 per day, and aggregate coding-agent runtime had grown to 3.1 agent-workdays per human workday [13]. These trends make the agent loop itself an increasingly important inference workload to optimize.

B. Prefix Caching

Autoregressive inference runs in two phases: (1) *prefill*, which processes the whole prompt in parallel to build its KV state, and (2) *decode*, which generates new tokens one at a time from that state. Modern serving engines avoid repeating this prefill work when requests share a common prefix. For example, vLLM's automatic prefix caching [14], [15] and SGLang's RadixAttention [16] (hashed KV blocks in vLLM, a radix tree of token sequences in SGLang) retain KV state from previously processed prefixes and reuse it when later requests begin with the same tokens, so only the uncached suffix must be computed. Successive agent calls append to an otherwise unchanged history, so each request nearly extends the previous one [17]. Prefix reuse, however, is *exact* (one changed token ends the match) and *position dependent* (the same tokens at a different offset do not match). In vLLM, cached KV state is organized into blocks whose identities incorporate both the block's tokens and its preceding prefix; consequently, reuse ends at the first changed block [18]. An append-only agent, therefore, needs to prefill only its newest suffix, whereas modifying an earlier part of the history forces the subsequent suffix to be recomputed (*re-prefill*). Inference cost thus depends not only on how many tokens remain in the context but also on where and how often it changes.

C. Context Compaction

An agent's history grows with every reasoning step, action, and observation, approaching the context-window limit and raising KV-cache memory and per-step prefill while much of it grows stale. Agent frameworks, therefore, reduce the retained history using mechanisms such as summarization (replacing old history with a model-written summary), truncation (dropping the oldest turns), or selective masking (replacing old tool outputs with a placeholder) [5], [6], [19], [20]. We use the term *context compaction* as an umbrella term for these operations: any transformation that reduces the amount of prior context carried forward while preserving enough task state for the agent to continue. The goal is not only to stay within the context window but also to control the growing token, memory, and serving costs of long-running trajectories. We study four such policies in §§V–VIII.

III. Experimental Methodology

Experimental platforms. We use two systems at the Argonne Leadership Computing Facility (ALCF). The text-level studies (§§IV–IX) run on Aurora [21], with six Intel Max 1550 accelerators (twelve GPU tiles) per node. Each node hosts three vLLM [14] servers (v0.15) with TP4 across four tiles, a $131{,}072$-token context and prefix caching enabled. The KV-level study (§X) runs on Polaris [22], whose nodes each have four NVIDIA A100 GPUs. Both engines serve Qwen3-Coder-30B-A3B in BF16 on SWE-bench Multilingual as four attention replicas (TP1, DP4) with the experts sharded across the GPUs (EP4); SGLang also serves Devstral-Small-2-24B on SWE-bench Verified as two replicas of two GPUs each (TP2, DP2). On both platforms, OpenHands [23] agents are

each pinned to one server or replica, so their prefix stays in a single cache.

Models and benchmarks. We evaluate one dense and one mixture-of-experts model, so the results are not tied to a single architecture family: Devstral-Small-2-24B [24], a dense model, and Qwen3-Coder-30B-A3B [25], a mixture-of-experts model with about 3B active parameters, of comparable size. We run both on two benchmarks. SWE-bench Verified [26], [27], the most widely used coding-agent benchmark, contains tasks written in Python. SWE-bench Multilingual [28] contains tasks in several other programming languages. Using both keeps our results from depending on a single benchmark or a single programming language.

Metrics and normalized serving-cost model. We record token counts per call and the prefix-cache counters of each serving engine, and report per task the solve rate, steps (model calls), prompt length, prefix-cache hit rate, and task completion time (TCT). We combine the token counts in a *normalized serving-cost model*, $C = w_c T_{\text{cached}} + w_u T_{\text{uncached}} + w_o T_{\text{out}}$, where T_{cached}, T_{uncached} and T_{out} are a task's cached prompt, uncached prompt, and output tokens, with $1 : 10 : 50$ weights following hosted API prices [29], [30]. For the KV-level study, we also report the *live KV* of a call, $L = T_{\text{prompt}} - T_{\text{dropped}}$, its prompt minus the tokens whose KV was dropped, and the *attended tokens* of a task, $A = \sum_{\text{calls}} L$, its live KV summed over all calls.

IV. Anatomy of an Agent's Context Cost

A coding agent's context is a record of its own work: the model's *reasoning*, the *actions* (tool calls) it issues, and the *observations* the environment returns for each of them. Reasoning and tool calls are short; observations, such as file views and test logs, are much larger. Nothing leaves the context once it has entered. To see where the cost goes, we attribute it event by event across the runs that finished on their own under no compaction, charging each event e a cost $c_e = s_e \cdot n_e$, where s_e is its size in tokens and n_e is the number of calls that follow it (i.e., how often it is re-sent).

Most of the cost is re-reading a cached prefix: Across the settings we employ, each token is re-sent tens of times before its run ends. A new turn is read once as a cold prefill and stays cached afterward, because nothing ahead of it is ever modified under no compaction, which puts the hit rate at 98 to 99 percent in every setting. What such an agent costs is therefore the volume of tokens it re-reads, not recomputation.

The cost is mostly tool output: Fig. 1a shows how the attributed cost divides between reasoning, actions, and observations. Observations take the largest share for every model and benchmark pair, around half the attributed cost, while reasoning accounts for only a few percent. Actions are not negligible: two thirds of their cost is writing files (a reproduction script and repository edits). Within the observations, file views are the largest part by a wide margin; test runs come next, and searches, builds, and edit confirmations are small.

The cost is concentrated in a few observations: Fig. 1b lines up the observations from the most expensive to the cheapest and shows how much of the total the first slice accounts for, against a diagonal where every observation would cost the same. The curves are far from that diagonal and nearly identical across models and benchmarks: a handful of very long outputs, mostly file views, carry most of the cost while comprising only a small share of the observations.

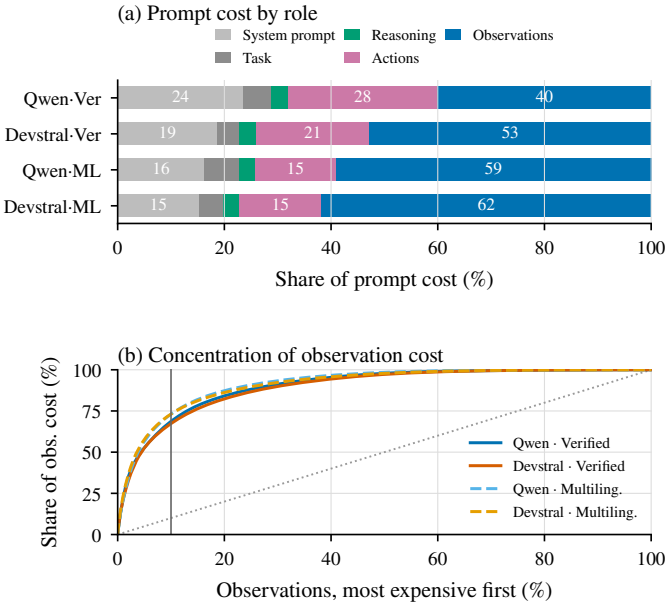


Fig. 1. Prompt cost without compaction.

When an observation arrives matters as much as its size: A second dimension is how long an observation stays in the context. Splitting observation costs by the quarter of the run in which they arrived, two outputs of the same size can cost very differently because n_e acts as a multiplier: an output in the first quarter is charged many times over, while one arriving late is charged comparatively little.

Key Takeaway: An agent only appends to the tail of its context, so the workload reuses its prefix almost perfectly and pays instead for the tokens it keeps re-reading. Compaction is costly on both sides: breaking the prefix cache forces a cold prefill of everything behind the edit, while postponing the break lets the re-read multiplier keep growing. When to break is therefore the design question.

V. Sliding Window

We begin with eviction at the granularity of a whole turn: instead of rewriting the trajectory, it discards the oldest turns and leaves the rest unchanged. A sliding window pins the system prompt and the task description at the head of the context and keeps only the k most recent turns behind them, where a *turn* is one (reasoning, action, observation) triple; as each new turn arrives, the window advances, and the turn that falls out of it is dropped entirely. This caps the prompt and the per-request KV footprint at k turns.

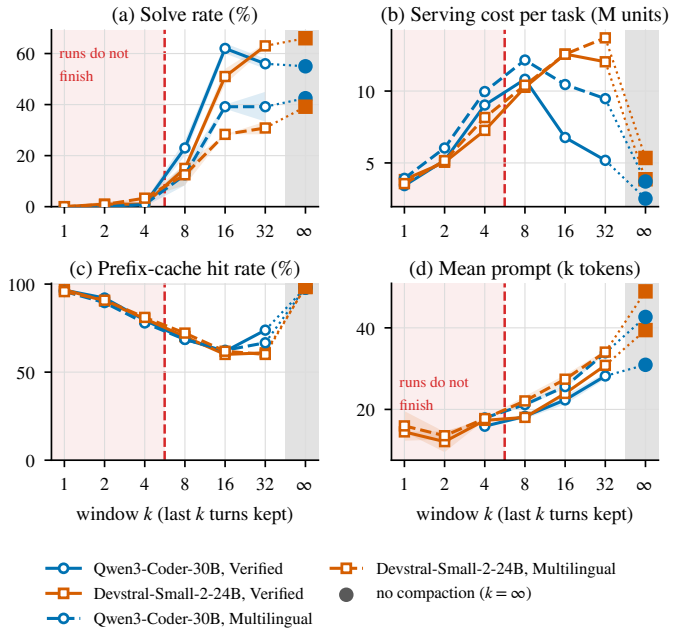


Fig. 2. Sliding window versus the window k. Red region: windows where almost no run finishes; filled marker at $k = \infty$: no compaction.

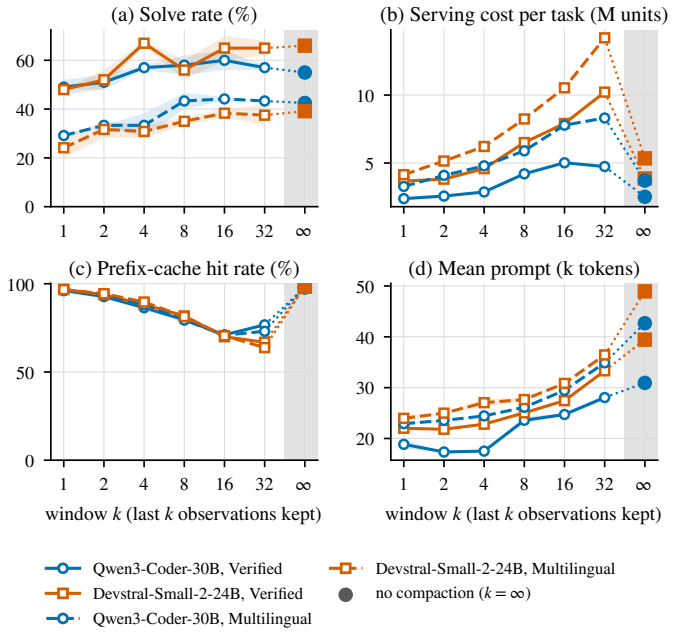


Fig. 3. Per-step observation masking versus the window k. Filled marker at $k = \infty$: no compaction; bands show run-to-run spread.

Task success: Fig. 2a shows how the solve rate responds to the window size. Task success collapses at small windows: it stays below 4 percent for every model and benchmark pair when the window consists of at most four turns, and it never exceeds 1 percent when it keeps one or two turns. The agent re-reads files and repeats commands to retrieve data that have fallen outside the window, and at small windows, this never converges; the runs loop until they hit the step limit, which is what almost every run does at one or two turns.

Serving cost: Fig. 2b shows the serving cost of all runs. No compaction costs between 2.5 and 5.3 M units per task (millions of normalized cost units, §III). The sliding window reaches 13.7 M, several times as much, for far fewer solved tasks. Even at the windows where accuracy recovers ($k \geq 16$), it still costs roughly 2 to 3× as much as no compaction in every pair, so no window is both accurate and cheap. The reason is cold prefill. The window advances every step, and the oldest turn is removed from the very front of the history, right after the pinned prompt. Thus, everything cached behind that point is invalid, and each call re-prefills the whole window instead of reusing it. Fig. 2c makes this visible as a U shaped hit rate: near 100 percent at the narrowest windows, where the pinned prompt and task description comprise most of what is sent, and about 60 percent in the middle of the sweep. Whether it recovers at the widest windows depends on trajectory length since a window only evicts once a run passes k steps.

Prompt size: The window does stop the growth that no compaction allows. Fig. 2d shows the mean prompt below the no compaction runs at every window, with the gap widening as the window narrows. The tokens that remain are, nevertheless, both expensive and insufficient: they are billed as uncached, and they leave out what the agent has already learned.

Key Takeaway: A sliding window bounds the context but not the cost. It gives the shortest prompts of any policy we measured and is at the same time the most expensive to serve and the least likely to finish, because dropping old turns takes away what the agent still needs and turns a reusable prefix into a cold prefill. Prompt length and KV footprint alone are therefore misleading measures of a policy.

VI. Observation Masking

Observation masking removes only the *observations* and keeps the reasoning and actions, which preserves task success where the sliding window did not [7]. Each observation older than the window k is replaced by a placeholder ("Previous N lines omitted for brevity"), so at every step, one more observation is masked, k turns behind the newest one.

Accuracy needs reasoning and actions: Fig. 3a shows the solve rate against k. Accuracy recovers as the window grows and then holds at the no-compaction level: the agent needs the most recent observations, and keeping more of them adds nothing.

Eviction may lengthen the run: Observations are the largest part of the context (§IV), mainly consisting of file views and test runs. Once an observation is masked, the agent may need the same content again and repeat the action that produced it (e.g., viewing the same file or rerunning the same

test), so eviction shortens the prompt but may lengthen the run.

Serving cost and prefix reuse: Cost is lowest at the narrowest window (below no compaction in every pair), passes no compaction at $k = 4$ and keeps climbing (Fig. 3b), so the windows that restore accuracy are also the most expensive to serve. The hit rate explains why (Fig. 3c): the newly masked observation sits k turns behind the newest one, so everything after it is prefilled again on the next call (i.e., the same invalidation as the sliding window). The context still falls below no compaction (Fig. 3d), but stays above the sliding window because the reasoning and actions are never removed.

Key Takeaway: Masking preserves accuracy because it evicts the right content, but that is only half the problem. It still edits the prompt at every step and throws away the prefix reuse that dominates an agent's cost, and what it masks may be fetched again, so a shorter prompt can mean a longer run. Policies of this kind should be designed around prefix reuse.

TABLE I
LLM SUMMARIZATION AT CONDENSER BUDGET M AGAINST BATCHED MASKING.

Policy	ctx (k)	Steps	TCT (×)	Cost (×)	Solve (%)
SWE-bench Verified, Qwen3-Coder					
No comp.	28.8	47	1.00	1.00	55.0
M=32	13.8	63	1.39	1.27	57.0
M=48	17.1	53	0.94	0.91	55.0
M=64	19.3	49	0.94	0.87	62.0
M=96	22.3	46	0.94	**0.86**	**65.0**
Batched k=8	22.5	47	0.85	0.88	60.0
SWE-bench Multilingual, Qwen3-Coder					
No comp.	39.6	58	1.00	1.00	42.5
M=32	15.4	75	1.21	1.05	30.0
M=48	19.2	58	0.99	0.86	36.7
M=64	22.1	57	0.91	**0.79**	37.5
M=96	28.5	54	0.87	0.82	36.7
Batched k=8	25.4	60	0.93	0.82	**44.2**
SWE-bench Multilingual, Devstral					
No comp.	43.9	69	1.00	1.00	**39.2**
M=32	18.3	72	2.16	1.06	25.8
M=48	18.8	70	1.07	0.81	34.2
M=64	22.7	72	1.02	0.79	34.2
M=96	31.2	66	1.75	0.86	32.5
Batched k=8	27.2	73	1.22	**0.79**	38.3
SWE-bench Verified, Devstral					
No comp.	36.2	64	1.00	1.00	**66.0**
M=32	14.6	88	1.43	1.26	58.0
M=48	17.5	74	0.99	1.02	**66.0**
M=64	20.7	75	0.97	1.06	59.0
M=96	25.9	64	0.89	0.94	63.0
Batched k=8	25.4	63	0.98	**0.92**	60.0

VII. LLM SUMMARIZATION

LLM summarization is the compaction policy used in deployment by major providers such as Claude Code [5] and OpenAI Codex [31], and by open-source coding agents such as OpenHands [6]. When the history exceeds a budget M, the model summarizes everything between the first few steps and a recent window, which stay verbatim, and the agent continues from the summary. M counts *events* (a step is two events: an action and its observation), so the history reaches about $M/2$ steps before it is rewritten, with roughly half kept verbatim.

Summary quality becomes part of task success: The extent to which summarization preserves task success depends on several interacting factors: when compaction occurs in the agent's trajectory, the summary length, and the quality of the generated summary. Unlike masking, summarization can permanently discard information. A masked observation can be recovered by re-executing its action, which remains intact, whereas details omitted from a summary are lost. On SWE-bench Verified, summarization preserves the no-compaction solve rate (e.g., 55–65% for Qwen3-Coder at $M \geq 48$, compared with 55% without compaction). On SWE-bench Multilingual, however, every summary budget reduces task success (e.g., 30.0–37.5% for Qwen3-Coder, compared with 42.5% without compaction). Because the model chooses what to retain, summarization introduces run-to-run variability in the compacted context. Nevertheless, it achieves the expected reduction in context size (Table I, ctx).

Serving cost: A summary leaves a small context, so the next cold prefill covers only the summary and the recent window. Unlike masking, summarization also loads *decode*, since the model must first write the summary. At the larger budgets, summarization and batched masking have similar serving costs (e.g., 0.86 to 0.87× no compaction for Qwen3-Coder on Verified at $M \geq 64$, against 0.88× for batched masking).

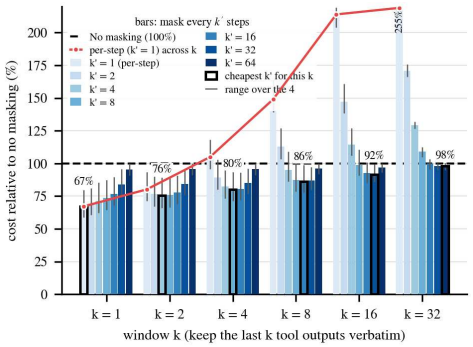


Fig. 4. Simulated cost of masking with window k (groups) and interval k' (bars), normalized to no compaction. $k' = 1$ is per-step masking (red line).

VIII. BATCHED OBSERVATION MASKING

The sliding window (§V) and observation masking (§VI) differ in what they remove, but both edit the prompt at every step. This section keeps what is removed fixed and changes only how often. We take the parameter from SWE-agent, whose `LastNObservations` history processor polls every few steps instead of every one, with the interval left to the user [32].

TABLE II

BATCHED OBSERVATION MASKING VS. NO COMPACTION. COST IS THE SERVING COST PER TASK (M UNITS, EVERY RUN BILLED); TCT IS THE MEAN WALL-CLOCK TIME PER TASK; CTX IS THE MEAN PROMPT LENGTH PER CALL; HIT% IS THE PREFIX-CACHE HIT RATE; STEPS AVERAGES FINISHED RUNS.

Model	Policy	API cost (M/task)	Δ	TCT (s)	Δ	ctx (k)	Δ	hit%	solve%	steps
SWE-bench Verified										
Qwen	No compaction	2.52	–	1174	–	30.9	–	97.9	55.0	48
	Batched, k=1	2.28	↓9%	1042	↓11%	19.1	↓38%	96.0	59.0	62
	Batched, k=2	2.28	↓10%	996	↓15%	19.8	↓36%	95.3	56.0	57
	Batched, k=4	2.33	↓7%	1010	↓14%	22.7	↓27%	95.4	58.0	52
	Batched, k=8	2.21	↓12%	998	↓15%	23.1	↓25%	95.7	60.0	49
	Batched, k=16	2.30	↓9%	1102	↓6%	25.5	↓17%	96.7	**62.0**	48
	Batched, k=32	2.39	↓5%	1077	↓8%	30.1	↓3%	97.8	57.0	47
Devstral	No compaction	3.86	–	1534	–	39.4	–	98.3	66.0	65
	Batched, k=1	3.40	↓12%	1420	↓7%	22.0	↓44%	96.5	54.0	83
	Batched, k=2	3.98	↑3%	1783	↑16%	25.1	↓36%	96.1	54.0	79
	Batched, k=4	3.70	↓4%	1560	↑2%	26.3	↓33%	96.0	54.0	77
	Batched, k=8	3.55	↓8%	1499	↓2%	25.9	↓34%	96.1	57.0	66
	Batched, k=16	3.45	↓11%	1386	↓10%	30.4	↓23%	96.8	**68.0**	65
	Batched, k=32	3.77	↓3%	1598	↑4%	34.4	↓13%	97.8	61.0	63
SWE-bench Multilingual										
Qwen	No compaction	3.71	–	1577	–	42.7	–	98.1	42.5	61
	Batched, k=1	3.18	↓14%	1433	↓9%	21.0	↓51%	96.1	40.8	85
	Batched, k=2	3.26	↓12%	1483	↓6%	22.6	↓47%	95.4	40.8	77
	Batched, k=4	3.15	↓15%	1442	↓9%	23.3	↓45%	95.3	37.5	72
	Batched, k=8	3.06	↓18%	1466	↓7%	26.8	↓37%	95.6	**44.2**	64
	Batched, k=16	3.31	↓11%	1473	↓7%	33.1	↓22%	96.8	37.5	63
	Batched, k=32	3.47	↓7%	1558	↓1%	38.2	↓11%	97.7	39.2	60
Devstral	No compaction	4.86	–	1201	–	44.9	–	98.5	39.2	75
	Batched, k=1	4.34	↓11%	1180	↓2%	22.9	↓49%	96.7	35.0	94
	Batched, k=2	4.35	↓11%	1099	↓9%	24.4	↓46%	95.9	28.3	89
	Batched, k=4	4.29	↓12%	1124	↓6%	26.0	↓42%	95.8	35.0	82
	Batched, k=8	4.46	↓8%	1154	↓4%	30.7	↓32%	96.2	35.0	78
	Batched, k=16	4.98	↑2%	954	↓21%	34.2	↓24%	97.1	34.2	83
	Batched, k=32	4.78	↓2%	1193	↓1%	42.5	↓5%	97.8	**42.5**	72

Masking in batches: Observation masking removes the right content, but at the worst possible rate: once per step. Batching applies the same rule once every k' steps instead (i.e., at steps t with $t \bmod k' = 0$); e.g., with $k = 4$ and $k' = 8$, masking happens at steps $8, 16, 24, \ldots$. Each time, all observations older than the newest k are masked together, and between applications, the prompt is only appended to. With $k' = 1$, batching reduces to per-step masking, so the batched policy contains it as a special case.

Choosing the batch size: A larger k' breaks the cache less often, but it also leaves more old observations in the prompt between batches. We replay the recorded trajectories with each candidate k', count which prompt tokens would be cached or recomputed, and price them with our cost model (§III). Fig. 4 shows that cost is U-shaped in k': masking every step pays for a cache break at every call, while a very large batch carries too many old observations. The best k' grows with the window and lies close to $2k$, within 0.4 percent of the cheapest on average; so, we fix $k' = 2k$.

Batching widens the window: The agent holds at least k observations and at most $k+k'-1$, which is $3k-1$ at $k' = 2k$, so the *visible window* averages $\bar{k} = k + (k' - 1)/2 \approx 2k$. Task success is preserved as a result: in every setting, the best window performs similarly to no compaction. At $k = 8$, finished runs also take within three steps of no compaction, so masking no longer lengthens the run. We ran six values of k with $k' = 2k$ on both models and benchmarks. As Table II shows, prefix reuse no longer decays with the window, staying between 95 and 98 percent where per-step observation masking fell towards 60 percent (§VI).

Less context, less recomputation, less time: Depending on k, accumulated context shrinks by up to 50% relative to no compaction (Table II, ctx). Preserving prefix reuse avoids additional prefill, reducing serving cost by up to 18% for most window sizes (cost) and task completion time by up to 21% on real hardware (TCT).

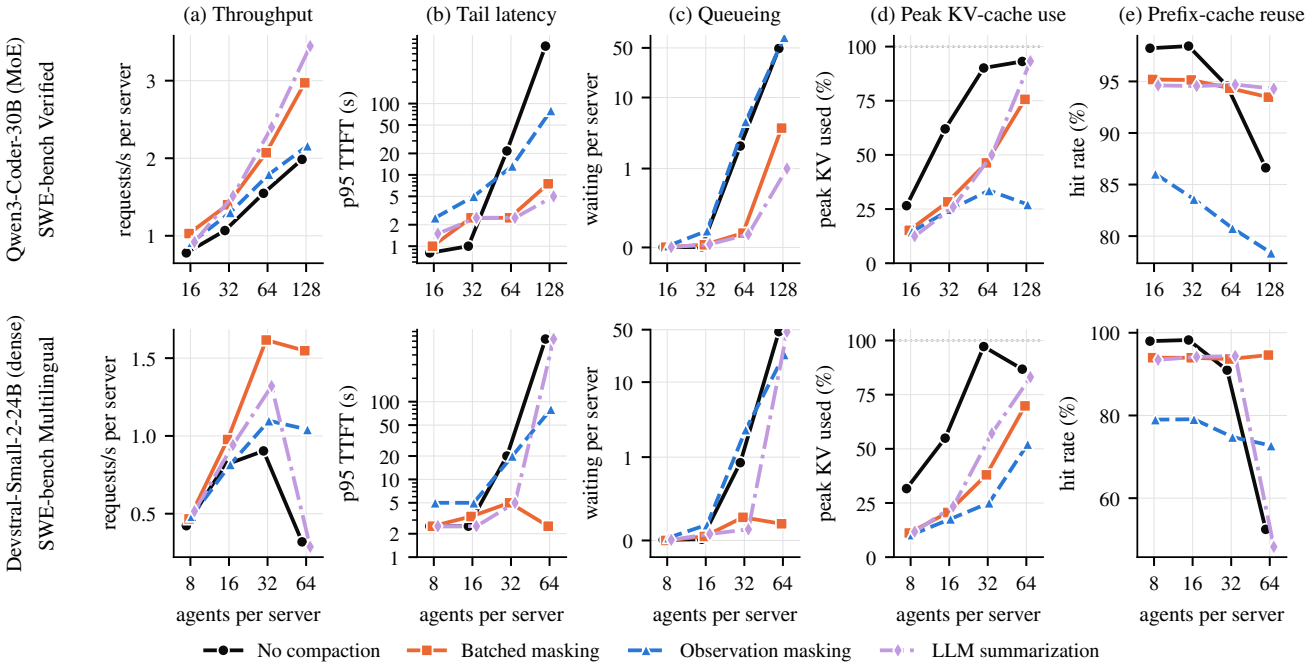


Fig. 5. Policies under concurrent load.

IX. Concurrency Experiments

We now put the compaction policies under concurrent load, where agents sharing a server compete for prefill compute and KV memory. We raise the number of agents per server until the server saturates (64 agents in the dense setting, 128 in the mixture-of-experts one), and replace each finished agent immediately so the load holds for the whole measurement. We pair each model with a different benchmark: Qwen3-Coder-30B on SWE-bench Verified and Devstral-Small-2-24B on SWE-bench Multilingual. All policies run at a single window, $k = 4$: no compaction, observation masking, batched masking ($k' = 8$) and LLM summarization. We leave out the sliding window (§V), since it solves almost nothing at these windows.

Throughput and latency under load: Fig. 5a–b compare throughput and tail latency under increasing load. The key distinction among policies is *where* they add load: memory for no compaction, prefill for observation masking, and decode for summarization, which must write its summaries first. At high load, no compaction performs worst in both settings: its full context fills the KV cache, and once the server evicts, agents re-prefill their whole context (Fig. 5e). Observation masking also incurs substantial prefill overhead because each prompt edit invalidates the cached prefix beyond the edit point, forcing the remaining suffix to be recomputed. Consequently, its performance remains close to that of no compaction. Interestingly, summarization and batched masking reverse their relative rankings across the two models. On the mixture-of-experts model, where relatively few parameters are activated and decoding is inexpensive, summarization achieves the highest throughput and lowest tail latency. In contrast, on the dense model, where token generation is more computationally expensive, batched masking outperforms summarization on both metrics.

Where the queue comes from: Fig. 5c shows the number of requests awaiting admission at each load level, while Fig. 5d shows the corresponding KV-cache occupancy. All policies queue beyond some load, but for different reasons. Observation masking accumulates queued requests despite using only a fraction of the available KV cache, indicating that it is primarily *compute-bound*. In contrast, no compaction nearly saturates the KV cache while incurring little recomputation, making it *memory-bound*. On the dense model, LLM summarization behaves similarly: its contexts grow until each rewrite, staying nearly as large as without compaction, so KV-cache capacity is quickly exhausted while expensive decoding further limits request processing. Summarization has the shortest queues on the mixture-of-experts model, but only batched masking keeps queues short on both.

Cache and reuse: Reuse separates the policies in Fig. 5e: batched masking and summarization stay near 94 percent, observation masking falls into the seventies, and no compaction collapses once the pool is full and the server starts evicting blocks. A queued agent then loses the prefix it had already paid for and prefills its whole context again (i.e., the tail latency of Fig. 5b).

Key Takeaway: A compaction policy decides where the extra work goes, and thus which resource saturates first: memory for no compaction, prefill for observation masking (queuing despite free memory), and decode for summarization, which fast decoding absorbs more easily. Batched masking adds little to either and kept queues short in both settings.

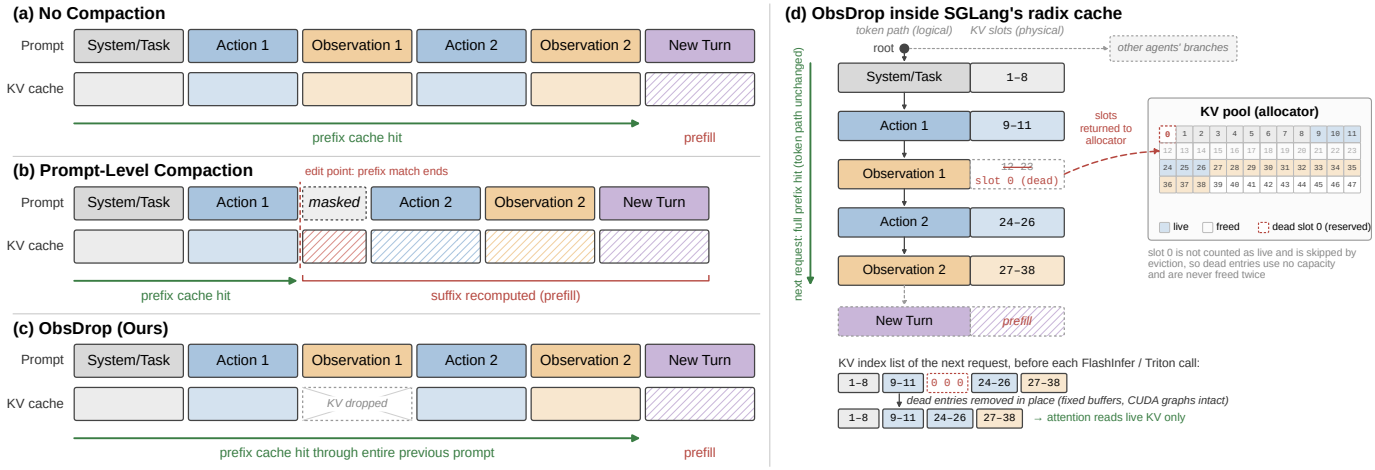


Fig. 6. Next-turn cache state under each policy, and how ObsDrop frees observation KV while keeping the prefix matchable.

X. KV-Level Observation Dropping

So far, every policy tested compacts the context at the text level (i.e., by rewriting the API prompt). In this section, we move compaction one layer down, into the *serving engine's KV cache*.

Preserving prefix reuse: Text-level compaction rewrites the prompt, so any edit breaks prefix reuse and forces a re-prefill of the suffix (Fig. 6b, against no compaction in Fig. 6a). *ObsDrop* instead leaves the prompt untouched (Fig. 6c): after each turn, it releases the KV entries of all but the K most recent observation contents, while keeping (1) every other token and (2) the identities and positions of the dropped ones. The next request therefore matches the full cached prefix and prefills only the new turn, and the dropped positions are excluded from attention, so the model holds less *live KV*.

Serving engines and design choices: We use SGLang [16] as the primary implementation platform because its RadixAttention cache represents cached conversations as a radix tree whose nodes map *logical* token sequences to *physical* KV-slot indices, so KV residency can change while the matched tokens remain the same. In contrast, vLLM's automatic prefix cache is organized around hash-identified KV blocks (i.e., keyed by their tokens and preceding prefix [18]). Stock vLLM cannot release KV entries inside a cached prefix while keeping it matchable. We additionally extend ObsDrop to Sparse-vLLM [33], an independent sparse-first engine, to test whether its benefits generalize beyond a single serving system. Because KV-level baselines (e.g., SnapKV [34] and H2O [35]) are not supported by the official SGLang and vLLM implementations, our KV-level comparison also runs in Sparse-vLLM.

SGLang implementation: Fig. 6d shows the change to the radix cache. After each turn is cached, ObsDrop identifies observation spans from the `<tool_response>` and `</tool_response>` token pairs. For observations older than the retained window, it walks the conversation's radix-tree path and (1) replaces the KV-slot indices of their contents with slot 0, which SGLang reserves for padding and ObsDrop treats as a *dead slot*, and (2) returns the original physical slots to the allocator. Cache accounting counts only live slots, while the allocator ignores slot 0 during eviction, so dead entries neither consume capacity nor get freed twice. Before each FlashInfer [36] or Triton [37] attention call, dead entries are removed in place from the request's KV index list, preserving the fixed buffers captured by CUDA graphs. Under attention data parallelism, all requests of a conversation go to the same replica, so its dead-slot state stays local.

Sparse-vLLM implementation: We extend ObsDrop to Sparse-vLLM with an endpoint that returns a prompt's exact token ids, so observation spans can be located, and an ObsDrop layer that, after each turn, releases the KV of observation contents older than the window while leaving the logical prefix unchanged. Under attention data parallelism, each release runs on the replica that holds the conversation.

Near-zero runtime overhead: Tracking dead slots naively adds GPU to CPU synchronization and repeated tree traversal to every request, which stalls the scheduler. To avoid synchronization in cache accounting, each radix-tree node stores a host-side record of its dead slots, so the live size of a node and the dead slots of a request are computed on the CPU. To avoid repeated traversal, after each turn ObsDrop finds observation boundaries in one pass over the prompt, walks the conversation's radix-tree path once, and frees only the observation that has just fallen out of the last K, skipping those already dropped. To avoid synchronization in attention, dead entries are removed from the KV index list in a single GPU pass, and the new lengths come from the host-side counts, so the attention kernel is planned without reading back from the GPU.

Quality and efficiency under a fixed KV budget: SGLang compares ObsDrop with *text-level* policies (batched masking and LLM summarization) and Sparse-vLLM with *KV-level* baselines (SnapKV and H2O). We first vary each policy's retention budget (Fig. 7, left). Across both SGLang workloads,

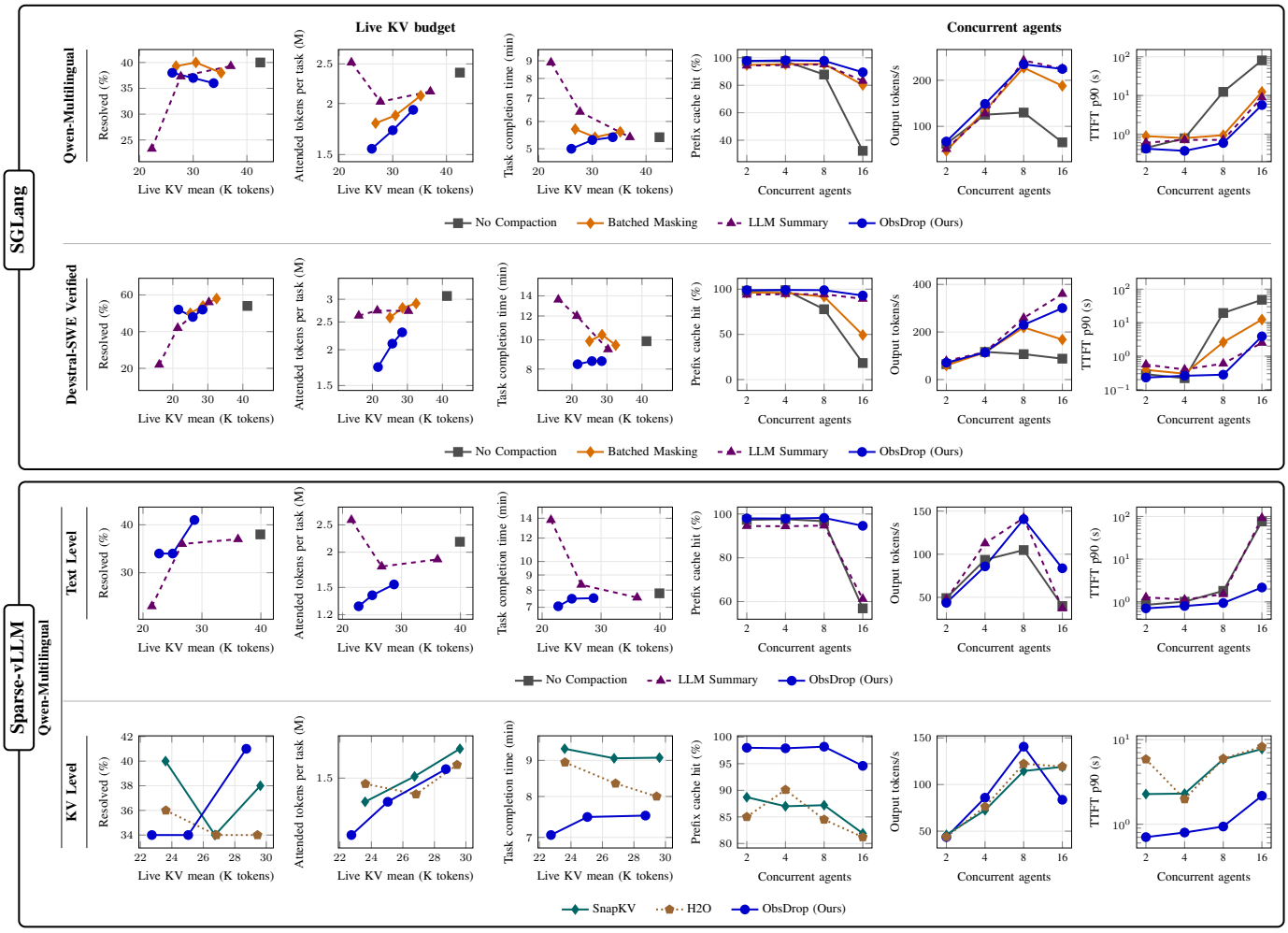


Fig. 7. SGLang (top two rows) and Sparse-vLLM text level and KV level policies against the live KV budget (left) and concurrent agents (right).

ObsDrop substantially reduces live KV (e.g., 26 to 34K against 43K tokens on Qwen-Multilingual) while keeping the solve rate close to no compaction. More importantly, these KV savings translate directly into less computation: ObsDrop consistently reduces attended tokens per task (i.e., 19 to 35% less on Qwen-Multilingual and 25 to 44% less on Devstral-SWE Verified). Across every engine and workload, ObsDrop has *the lowest attended-token volume and the shortest task completion time of all compared policies*. The same trend holds in Sparse-vLLM, showing that the benefit is not tied to a single serving system. Against the KV-level baselines, ObsDrop also achieves a lower attended-token volume while maintaining comparable task quality. Overall, these results show the central benefit of ObsDrop: it reduces physical KV residency and attention work without rewriting the logical agent history, so memory and computation fall together without the large quality loss usually associated with aggressive context reduction.

Concurrency reveals the main systems advantage. We next increase the number of concurrent agents from 2 to 16 (Fig. 7, right) to stress prefix-cache residency and expose the cost of KV pressure. ObsDrop is consistently the most robust policy as concurrency grows: its prefix-cache hit rate remains near 95–100% through most of the sweep, whereas no compaction collapses once the cache becomes pressured and text-level policies lose reuse whenever they rewrite history, which shows directly in tail latency. Across both SGLang workloads and both Sparse-vLLM comparisons, ObsDrop delivers the lowest p90 TTFT at nearly every concurrency level; the only exception is Devstral-SWE Verified at 16 agents, where LLM summarization is slightly lower. At 16 agents, Qwen-Multilingual on SGLang retains 89% prefix-cache reuse with ∼6 s p90 TTFT, while no compaction falls to 32% reuse and ∼80 s TTFT. On Sparse-vLLM's text-level comparison, ObsDrop similarly retains about 95% reuse with ∼2 s TTFT, while no compaction and summarization suffer severe cache loss and tens-of-seconds of tail latency. Against SnapKV and H2O, ObsDrop maintains substantially higher prefix reuse across 2, 4, 8, and 16 agents and the lowest TTFT throughout the sweep. Its KV savings preserve reusable inference state under concurrency, preventing eviction–re-prefill cycles.

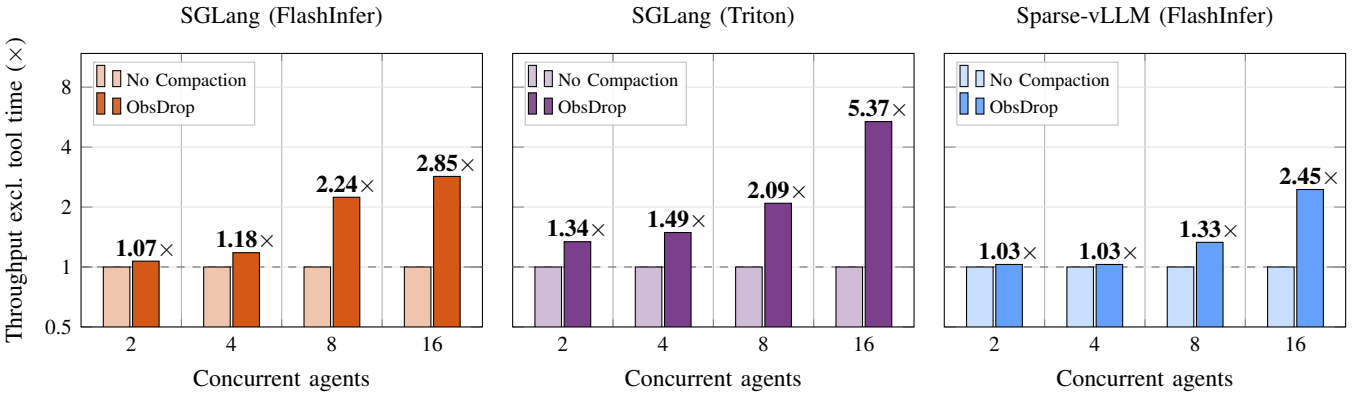


Fig. 8. Output token throughput excluding tool time, with ObsDrop relative to no compaction on the same engine and kernel at each concurrent agent count.

Cross-engine and kernel evaluation: Fig. 8 shows the resulting throughput gain over no compaction. With FlashInfer, the gain follows memory pressure (1.03 to 1.18× at two or four agents), reaching 2.85× on SGLang and 2.45× on Sparse-vLLM at 16 agents. With Triton, the gain is larger at every load (1.34 to 1.49× at two or four agents, 5.37× at 16), because Triton's decode cost grows faster with context length, so attending to fewer tokens speeds up every step. ObsDrop's gain thus has two parts: a *memory* part on every engine and a *compute* part whose size depends on the attention kernel.

XI. Related Work

Agent context management. Prior work reduces growing contexts through prompt compression [38] (dropping tokens that a small model deems unimportant), external memory [39] (paging state in and out of a store), and observation masking [7] (replacing old tool outputs with placeholders). We differ by studying how compaction frequency affects prefix reuse (§VIII) and by dropping stale observations directly from the KV cache (§X), without an auxiliary compression model.

KV-cache compression and sparse attention. Existing methods reduce KV cost by retaining *model-selected* tokens (i.e., chosen by attention scores or the current query): H2O keeps recent and heavy-hitter tokens [35], SnapKV selects important positions from attention patterns [34], PyramidKV allocates different KV budgets across layers [40], and Quest dynamically selects KV pages based on the current query [41]. Such selection can also degrade instructions unevenly [42]. Effective as they are, their selection mechanisms require specialized cache-management or sparse-attention support and do not address the logical cached prefix used for request-level prefix reuse. CacheBlend instead reuses non-prefix KV by partial recomputation [43].

How ObsDrop differs. Prior work either (1) modifies the prompt (context management) or (2) chooses which KV entries to retain (KV compression). ObsDrop instead targets request-level prefix reuse across iterative agent calls: it uses the agent's observation boundaries to decouple *logical* prefix matching from *physical* KV residency, so stale observation KV is reclaimed while the token sequence used for prefix matching stays unchanged. Needing no sparse-attention support, it fits the existing prefix cache of a general-purpose engine (§X). Neither stock SGLang nor vLLM exposes a mechanism to release the KV entries of selected tokens inside a cached prefix while keeping that prefix matchable for later requests. To the best of our knowledge, ObsDrop is the first open-source implementation of such a mechanism for SGLang, and we release it.

XII. Limitations and Future Work

Our simulation study uses $k' = 2k$ as a practical batching interval, but the optimal k' may vary across agent trajectories and workloads. We also use API cost as an efficiency metric; although pricing and cached-to-uncached token cost ratios may change, the underlying trade-off between context compaction and prefix reuse remains. We further observe a strong correlation between API cost and task completion time on real hardware, supporting the broader applicability of our findings. Future work could choose the batch size k' dynamically per trajectory and skip masking during the early steps of a trajectory. It also includes porting ObsDrop to vLLM, which requires decoupling block identity from KV residency.

XIII. Concluding Remarks

We characterize how context compaction interacts with prefix caching in coding agents and showing that rewriting history can shorten prompts while invalidating cached KV and increasing serving costs. We then show that *batching* observation masking amortizes these invalidations at the prompt level and present *ObsDrop*, which avoids them at the KV level by dropping stale observation KV without changing the logical prompt. Across two serving engines and two attention backends, ObsDrop improves cache residency, throughput, and tail latency under concurrent agent workloads while maintaining task quality. More fundamentally, ObsDrop gives SGLang a primitive it did not have, namely releasing KV inside a cached prefix while keeping that prefix matchable, and opens the door to *a new class of KV-level context management policies for agents*.

XIV. Acknowdlegments

This research used resources of the Argonne Leadership Computing Facility, which is a U.S. Department of Energy Office of Science User Facility operated under contract DE-AC02-06CH11357.

References

[1] X. Wang, Y. Chen, L. Yuan, Y. Zhang, Y. Li, H. Peng, and H. Ji, "Executable code actions elicit better LLM agents," in *Forty-first International Conference on Machine Learning, ICML 2024, Vienna, Austria, July 21-27, 2024*, ser. Proceedings of Machine Learning Research, R. Salakhutdinov, Z. Kolter, K. A. Heller, A. Weller, N. Oliver, J. Scarlett, and F. Berkenkamp, Eds., vol. 235. PMLR / OpenReview.net, 2024, pp. 50 208–50 232. [Online]. Available: https://proceedings.mlr.press/v235/wang24h.html

[2] B. Gao, Z. He, P. Sharma, Q. Kang, D. Jevdjic, J. Deng, X. Yang, Z. Yu, and P. Zuo, "Cost-efficient large language model serving for multi-turn conversations with cachedattention," in *Proceedings of the 2024 USENIX Annual Technical Conference, USENIX ATC 2024, Santa Clara, CA, USA, July 10-12, 2024*, S. Bagchi and Y. Zhang, Eds. USENIX Association, 2024, pp. 111–126. [Online]. Available: https://www.usenix.org/conference/atc24/presentation/gao-bin-cost

[3] R. Qin, Z. Li, W. He, J. Cui, F. Ren, M. Zhang, Y. Wu, W. Zheng, and X. Xu, "Mooncake: Trading more storage for less computation - A kvcache-centric architecture for serving LLM chatbot," in *23rd USENIX Conference on File and Storage Technologies, FAST 2025, Santa Clara, CA, February 25-27, 2025*, H. S. Gunawi and V. Tarasov, Eds. USENIX Association, 2025, pp. 155–170. [Online]. Available: https://www.usenix.org/conference/fast25/presentation/qin

[4] T. Dao, D. Y. Fu, S. Ermon, A. Rudra, and C. Ré, "Flashattention: Fast and memory-efficient exact attention with io-awareness," in *Advances in Neural Information Processing Systems 35: Annual Conference on Neural Information Processing Systems 2022, NeurIPS 2022, New Orleans, LA, USA, November 28 - December 9, 2022*, S. Koyejo, S. Mohamed, A. Agarwal, D. Belgrave, K. Cho, and A. Oh, Eds., 2022. [Online]. Available: http://papers.nips.cc/paper_files/paper/2022/hash/67d57c32e20fd0a7a302cb81d36e40d5-Abstract-Conference.html

[5] Anthropic, "Compaction," Claude Platform Documentation, https://platform.claude.com/docs/en/build-with-claude/compaction, 2026.

[6] OpenHands Project, "Context condenser," OpenHands Documentation, https://docs.openhands.dev/sdk/guides/context-condenser, 2025.

[7] T. Lindenbauer, I. Slinko, L. Felder, E. Bogomolov, and Y. Zharov, "The complexity trap: Simple observation masking is as efficient as LLM summarization for agent context management," *CoRR*, vol. abs/2508.21433, 2025. [Online]. Available: https://doi.org/10.48550/arXiv.2508.21433

[8] M. Luo, X. Shi, C. Cai, T. Zhang, J. Wong, Y. Wang, C. Wang, Y. Huang, Z. Chen, J. E. Gonzalez, and I. Stoica, "Autellix: An efficient serving engine for LLM agents as general programs," *CoRR*, vol. abs/2502.13965, 2025. [Online]. Available: https://doi.org/10.48550/arXiv.2502.13965

[9] A. Agrawal, N. Kedia, A. Panwar, J. Mohan, N. Kwatra, B. S. Gulavani, A. Tumanov, and R. Ramjee, "Taming throughput-latency tradeoff in LLM inference with sarathi-serve," in *18th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2024, Santa Clara, CA, USA, July 10-12, 2024*, A. Gavrilovska and D. B. Terry, Eds. USENIX Association, 2024, pp. 117–134. [Online]. Available: https://www.usenix.org/conference/osdi24/presentation/agrawal

[10] Y. Zhong, S. Liu, J. Chen, J. Hu, Y. Zhu, X. Liu, X. Jin, and H. Zhang, "Distserve: Disaggregating prefill and decoding for goodput-optimized large language model serving," in *18th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2024, Santa Clara, CA, USA, July 10-12, 2024*, A. Gavrilovska and D. B. Terry, Eds. USENIX Association, 2024, pp. 193–210. [Online]. Available: https://www.usenix.org/conference/osdi24/presentation/zhong-yinmin

[11] S. Yao, J. Zhao, D. Yu, N. Du, I. Shafran, K. R. Narasimhan, and Y. Cao, "React: Synergizing reasoning and acting in language models," in *The Eleventh International Conference on Learning Representations, ICLR 2023, Kigali, Rwanda, May 1-5, 2023*. OpenReview.net, 2023. [Online]. Available: https://openreview.net/forum?id=WE_vluYUL-X

[12] L. Bai, Z. Huang, X. Wang, J. Sun, R. Mihalcea, E. Brynjolfsson, A. Pentland, and J. Pei, "How do AI agents spend your money? analyzing and predicting token consumption in agentic coding tasks," *CoRR*, vol. abs/2604.22750, 2026. [Online]. Available: https://doi.org/10.48550/arXiv.2604.22750

[13] OpenAI, "Research acceleration: The view inside openai," https://openai.com/index/research-acceleration-view-inside-openai/, Sep. 2026, accessed: 2026-10-05.

[14] W. Kwon, Z. Li, S. Zhuang, Y. Sheng, L. Zheng, C. H. Yu, J. Gonzalez, H. Zhang, and I. Stoica, "Efficient memory management for large language model serving with pagedattention," in *Proceedings of the 29th Symposium on Operating Systems Principles, SOSP 2023, Koblenz, Germany, October 23-26, 2023*, J. Flinn, M. I. Seltzer, P. Druschel, A. Kaufmann, and J. Mace, Eds. ACM, 2023, pp. 611–626. [Online]. Available: https://doi.org/10.1145/3600006.3613165

[15] vLLM Project, "Automatic prefix caching," vLLM Documentation, https://docs.vllm.ai/en/latest/features/automatic_prefix_caching.html, 2025.

[16] L. Zheng, L. Yin, Z. Xie, C. Sun, J. Huang, C. H. Yu, S. Cao, C. Kozyrakis, I. Stoica, J. E. Gonzalez, C. W. Barrett, and Y. Sheng, "Sglang: Efficient execution of structured language model programs," in *Advances in Neural Information Processing Systems 37: Annual Conference on Neural Information Processing Systems 2024, NeurIPS 2024, Vancouver, BC, Canada, December 10 - 15, 2024*, A. Globersons, L. Mackey, D. Belgrave, A. Fan, U. Paquet, J. M. Tomczak, and C. Zhang, Eds., 2024. [Online]. Available: http://papers.nips.cc/paper_files/paper/2024/hash/724be4472168f31ba1c9ac630f15dec8-Abstract-Conference.html

[17] E. Lumer, F. Nizar, A. Jangiti, K. Frank, A. Gulati, M. Phadate, and V. K. Subbiah, "Don't break the cache: An evaluation of prompt caching for long-horizon agentic tasks," *CoRR*, vol. abs/2601.06007, 2026. [Online]. Available: https://doi.org/10.48550/arXiv.2601.06007

[18] vLLM Project, "Automatic prefix caching: Design," vLLM Documentation, https://docs.vllm.ai/en/latest/design/prefix_caching.html, 2026.

[19] LangChain, "Context engineering in agents," LangChain Documentation, https://docs.langchain.com/oss/python/langchain/context-engineering, 2026.

[20] Anthropic, "Context editing," Claude Platform Documentation, https://platform.claude.com/docs/en/build-with-claude/context-editing, 2026.

[21] Argonne Leadership Computing Facility, "Aurora," accessed: 2026-10-05. [Online]. Available: https://www.alcf.anl.gov/aurora

[22] ——, "Polaris," accessed: 2026-10-05. [Online]. Available: https://www.alcf.anl.gov/polaris

[23] X. Wang, B. Li, Y. Song, F. F. Xu, X. Tang, M. Zhuge, J. Pan, Y. Song, B. Li, J. Singh, H. H. Tran, F. Li, R. Ma, M. Zheng, B. Qian, Y. Shao, N. Muennighoff, Y. Zhang, B. Hui, J. Lin, R. Brennan, H. Peng, H. Ji, and G. Neubig, "Openhands: An open platform for AI software developers as generalist agents," in *The Thirteenth International Conference on Learning Representations, ICLR 2025, Singapore, April 24-28, 2025*. OpenReview.net, 2025. [Online]. Available: https://openreview.net/forum?id=OJd3ayDDoF

[24] M. AI and A. H. AI, "Devstral: Fine-tuning language models for coding agent applications," *CoRR*, vol. abs/2509.25193, 2025. [Online]. Available: https://doi.org/10.48550/arXiv.2509.25193

[25] Qwen Team, "Qwen3-coder-30b-a3b-instruct," https://huggingface.co/Qwen/Qwen3-Coder-30B-A3B-Instruct, 2025.

[26] C. E. Jimenez, J. Yang, A. Wettig, S. Yao, K. Pei, O. Press, and K. R. Narasimhan, "Swe-bench: Can language models resolve real-world github issues?" in *The Twelfth International Conference on Learning Representations, ICLR 2024, Vienna, Austria, May 7-11, 2024*. OpenReview.net, 2024. [Online]. Available: https://openreview.net/forum?id=VTF8yNQM66

[27] OpenAI, "Introducing swe-bench verified," Aug. 2024. [Online]. Available: https://openai.com/index/introducing-swe-bench-verified/

[28] J. Yang, K. Lieret, C. E. Jimenez, A. Wettig, K. Khandpur, Y. Zhang, B. Hui, O. Press, L. Schmidt, and D. Yang, "Swe-smith: Scaling data for software engineering agents," in *Advances in Neural Information Processing Systems 38: Annual Conference on Neural Information Processing Systems 2025, NeurIPS 2025, San Diego, CA, USA, December 2-7, 2025 / Mexico City, Mexico, November 30 - December 5, 2025*, D. Belgrave, C. Zhang, L. N. Montoya, H. Lin, R. Pascanu, P. Koniusz, M. Ghassemi, N. Chen, I. V. M. Ruíz, and A. Loaiza-Bonilla, Eds., 2025. [Online]. Available: http://papers.nips.cc/paper_files/paper/2025/hash/8b86cf5ace600c48fd188efbb8dedec8-Abstract-Datasets_and_Benchmarks_Track.html

[29] Anthropic, "Claude api pricing," https://platform.claude.com/docs/en/about-claude/pricing, 2026, accessed: 2026-10-05.

[30] OpenAI, "Api pricing," https://developers.openai.com/api/docs/pricing, 2026, accessed: 2026-10-05.

[31] ——, "Introducing Codex," https://openai.com/index/introducing-codex/, 2025.

[32] J. Yang, C. E. Jimenez, A. Wettig, K. Lieret, S. Yao, K. Narasimhan, and O. Press, "Swe-agent: Agent-computer interfaces enable automated software engineering," in *Advances in Neural Information Processing Systems 37: Annual Conference on Neural Information Processing Systems 2024, NeurIPS 2024, Vancouver, BC, Canada, December 10 - 15, 2024*, A. Globersons, L. Mackey, D. Belgrave, A. Fan, U. Paquet, J. M. Tomczak, and C. Zhang, Eds., 2024. [Online]. Available: http://papers.nips.cc/paper_files/paper/2024/hash/5a7c947568c1b1328ccc5230172e1e7c-Abstract-Conference.html

[33] CURRENTF, "Sparse-vLLM," 2026. [Online]. Available: https://github.com/CURRENTF/sparse-vllm

[34] Y. Li, Y. Huang, B. Yang, B. Venkitesh, A. Locatelli, H. Ye, T. Cai, P. Lewis, and D. Chen, "Snapkv: LLM knows what you are looking for before generation," in *Advances in Neural Information Processing Systems 37: Annual Conference on Neural Information Processing Systems 2024, NeurIPS 2024, Vancouver, BC, Canada, December 10 - 15, 2024*, A. Globersons, L. Mackey, D. Belgrave, A. Fan, U. Paquet, J. M. Tomczak, and C. Zhang, Eds., 2024. [Online]. Available: http://papers.nips.cc/paper_files/paper/2024/hash/28ab418242603e0f7323e54185d19bde-Abstract-Conference.html

[35] Z. Zhang, Y. Sheng, T. Zhou, T. Chen, L. Zheng, R. Cai, Z. Song, Y. Tian, C. Ré, C. W. Barrett, Z. Wang, and B. Chen, "H2O: heavy-hitter oracle for efficient generative inference of large language models," in *Advances in Neural Information Processing Systems 36: Annual Conference on Neural Information Processing Systems 2023, NeurIPS 2023, New Orleans, LA, USA, December 10 - 16, 2023*, A. Oh, T. Naumann, A. Globerson, K. Saenko, M. Hardt, and S. Levine, Eds., 2023. [Online]. Available: http://papers.nips.cc/paper_files/paper/2023/hash/6ceefa7b15572587b78ecfcebb2827f8-Abstract-Conference.html

[36] Z. Ye, L. Chen, R. Lai, W. Lin, Y. Zhang, S. Wang, T. Chen, B. Kasikci, V. Grover, A. Krishnamurthy, and L. Ceze, "Flashinfer: Efficient and customizable attention engine for LLM inference serving," in *Proceedings of the Eighth Conference on Machine Learning and Systems, MLSys 2025, Santa Clara, CA, USA, May 12-15, 2025*, M. Zaharia, G. Joshi, and Y. C. Lin, Eds. OpenReview.net/mlsys.org, 2025. [Online]. Available: https://openreview.net/forum?id=RXPofAsL8F

[37] P. Tillet, H. Kung, and D. D. Cox, "Triton: an intermediate language and compiler for tiled neural network computations," in *Proceedings of the 3rd ACM SIGPLAN International Workshop on Machine Learning and Programming Languages, MAPL@PLDI 2019, Phoenix, AZ, USA, June 22, 2019*, T. Mattson, A. Muzahid, and A. Solar-Lezama, Eds. ACM, 2019, pp. 10–19. [Online]. Available: https://doi.org/10.1145/3315508.3329973

[38] H. Jiang, Q. Wu, C. Lin, Y. Yang, and L. Qiu, "Llmlingua: Compressing prompts for accelerated inference of large language models," in *Proceedings of the 2023 Conference on Empirical Methods in Natural Language Processing, EMNLP 2023, Singapore, December 6-10, 2023*, H. Bouamor, J. Pino, and K. Bali, Eds. Association for Computational Linguistics, 2023, pp. 13 358–13 376. [Online]. Available: https://doi.org/10.18653/v1/2023.emnlp-main.825

[39] C. Packer, V. Fang, S. G. Patil, K. Lin, S. Wooders, and J. E. Gonzalez, "Memgpt: Towards llms as operating systems," *CoRR*, vol. abs/2310.08560, 2023. [Online]. Available: https://doi.org/10.48550/arXiv.2310.08560

[40] Z. Cai, Y. Zhang, B. Gao, Y. Liu, T. Liu, K. Lu, W. Xiong, Y. Dong, B. Chang, J. Hu, and W. Xiao, "Pyramidkv: Dynamic KV cache compression based on pyramidal information funneling," *CoRR*, vol. abs/2406.02069, 2024. [Online]. Available: https://doi.org/10.48550/arXiv.2406.02069

[41] J. Tang, Y. Zhao, K. Zhu, G. Xiao, B. Kasikci, and S. Han, "QUEST: query-aware sparsity for efficient long-context LLM inference," in *Forty-first International Conference on Machine Learning, ICML 2024, Vienna, Austria, July 21-27, 2024*, ser. Proceedings of Machine Learning Research, R. Salakhutdinov, Z. Kolter, K. A. Heller, A. Weller, N. Oliver, J. Scarlett, and F. Berkenkamp, Eds., vol. 235. PMLR / OpenReview.net, 2024, pp. 47 901–47 911. [Online]. Available: https://proceedings.mlr.press/v235/tang24l.html

[42] A. Chen, R. L. Geh, A. Grover, G. V. den Broeck, and D. M. Israel, "The pitfalls of KV cache compression," in *Proceedings of the 64th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers), ACL 2026, San Diego, California, United States, July 2-7, 2026*, M. Liakata, V. P. Moreira, J. Zhang, and D. Jurgens, Eds. Association for Computational Linguistics, 2026, pp. 41 530–41 553. [Online]. Available: https://doi.org/10.18653/v1/2026.acl-long.1926

[43] J. Yao, H. Li, Y. Liu, S. Ray, Y. Cheng, Q. Zhang, K. Du, S. Lu, and J. Jiang, "Cacheblend: Fast large language model serving for RAG with cached knowledge fusion," in *Proceedings of the Twentieth European Conference on Computer Systems, EuroSys 2025, Rotterdam, The Netherlands, 30 March 2025 - 3 April 2025*. ACM, 2025, pp. 94–109. [Online]. Available: https://doi.org/10.1145/3689031.3696098