CLIFFCOMPACTION: COST-EFFICIENT COMPACTION FOR LONG-HORIZON CODING AGENTS

Anonymous authors
Paper under double-blind review

ABSTRACT

Agents often work on complex problems that require millions of tokens of context, necessitating compaction across sessions with limited context windows. We develop CLIFFCOMPACTION, an autocompaction technique that reduces cost by up to 50% under a bounded context while maintaining or improving performance on tasks such as Terminal-Bench. Its per-rollout savings can be reinvested in test-time scaling, improving Terminal-Bench by over 10 points for under the cost of two full-context runs, letting Kimi K2.6 match Opus 4.7 and exceed Opus 4.6 and GPT-5.3 Codex at lower cost. On KernelBench, CLIFFCOMPACTION achieves state-of-the-art CUDA kernel speedups ($2.23\times$ at 200 steps, $3.58\times$ at 400) across sessions exceeding a million tokens, surpassing trained agents and specialized search methods. The key to CLIFFCOMPACTION is preserving context fidelity: (1) it only truncates or drops content, never rewrites it, and (2) never re-compacts a compaction, preventing context drift. We open-source a scaffold-agnostic API-proxy implementation usable with Claude Code, Codex, and other harnesses.

1 INTRODUCTION

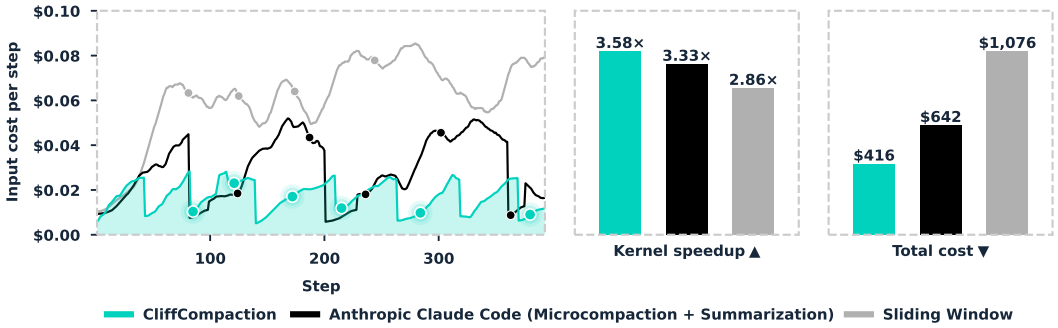


Figure 1: Continual learning of CUDA kernel development on KernelBench using `OpenHands`: **CLIFFCOMPACTION is both the cheapest and the highest performing.** *Left:* per-step input cost on one Level 3 problem; dots mark when each method's overall speedup across 50 problems first reaches $1.0\times, 1.5\times, \ldots, 3.5\times$. *Right:* final speedup (▲) and total cost (▼) across all 50 problems.

Coding agents solve complex tasks by interacting with tools and environments over contexts spanning millions of tokens. However, processing such long contexts is computationally expensive, and increasingly bloated contexts can impair agent effectiveness as trajectories grow longer. We show that bounded context with autocompaction can match or exceed full-context performance at lower cost.

We develop CLIFFCOMPACTION, an autocompaction technique for coding agents. On SWE-bench Verified, CLIFFCOMPACTION preserves full-context success rates for GLM-5.1 and Kimi K2.6 with context thresholds of only 32K and 16K tokens. On Terminal-Bench 2.0, it achieves higher success rates while reducing cost by 50%.

For long context tasks, two main problems arise: (a) how to manage context information once an agent session runs out of context length; (b) how to maximize agent performance while reducing cost from long context.

Reviewers: please read the Reviewer Guidelines (iclr.cc/Conferences/2027/ReviewerGuidelines) and the AI Policy for Reviewers (iclr.cc/Conferences/2027/AIPolicyForReviewers). If you used AI to expand, edit, or polish your review, please provide the input text to the LLM. Better still, consider skipping the LLM and submitting your original text: we, and the authors, are much more interested in your unedited thoughts than in what an LLM has to say. AI-assisted or not, you are putting your name and reputation behind your review: LLM-generated falsehoods, hallucinations or misrepresentations are subject to disciplinary action, which may include desk-rejecting all papers you have authored.

For context management (a), the main approaches are either to *compact*—to truncate or summarize—the context or to store memories and start a new session with those memories. Sliding windows, which keep only the most recent turns or observations (Yang et al., 2024), are effective drop-in solutions, but invalidate the prefix cache whenever the window advances, increasing inference cost. LLM-based summarization, though widely used, compounds loss across summary-of-summary chains.

A more demanding challenge in this area is continual learning, where an agent needs to continually improve previous solutions to a problem where the total context can reach millions of tokens. While solutions for continual learning exist, they often rely on complex, task-specific methods (Du et al., 2026; Dai et al., 2026) with limited generalizability.

On the efficiency side (b), the long context is represented by the KV-cache. While the KV-cache leads to efficient inference, it is the main memory and computational bottleneck for long-context sessions and increases the latency and inference cost significantly. While system-level *backend* solutions such as FlashAttention (Dao et al., 2022) or DeepSeek Sparse Attention (DeepSeek-AI, 2025) reduce the KV-cache memory requirements directly, token-based *frontend* algorithms seek to reduce KV-cache costs indirectly and improve model performance by manipulating the tokens in the context.

One particular problem for frontend efficiency is test-time scaling (Kwok et al., 2026; Kim et al., 2026), where one seeks to use a model repeatedly and thus use additional tokens to achieve favorable cost-performance trade-offs compared to another model, for example, a frontier model. However, test-time scaling often requires tens of rollouts and incurs exorbitant inference costs. Moreover, as scaling gains saturate and parallel scaling introduces a selection bottleneck, it remains unclear whether the marginal improvements justify the expense.

CLIFFCOMPACTION addresses these challenges: it maintains performance on coding and terminal tasks at lower cost, makes test-time scaling economical, and sets a new best result for continual learning on KernelBench under a matched evaluation setup.

Table 1: Context-management approaches. ●: supported; ○: not supported; ◐: partial or conditional. CLIFFCOMPACTION sacrifices only full-history recall, which we show is not needed (§3).

Approach	Training-free	Drop-in	Model-agnostic	No aux. LLM call	Cache-friendly	High precision	Full-history recall
Sliding window	●	●	●	●	○	●	○
LLM summarization	●	●	●	○	●	○	◐
Learned compression	○	○	○	○	◐	○	◐
Sub-agent decomposition	◐	○	◐	○	◐	◐	◐
External memory / RAG	●	○	●	●	◐	●	●
Structure-based	●	○	●	●	○	●	◐
Anthropic Claude Code	●	●	●	○	◐	○	◐
CLIFFCOMPACTION	●	●	●	●	●	●	○

CLIFFCOMPACTION truncates or drops components that consume the most context, primarily tool calls and outputs. At each compaction, it discards the previous compacted history and builds a new block from original content in the current active context. These choices keep CLIFFCOMPACTION faithful to the original context and prevent drift from accumulating across compactions. Table 1 compares CLIFFCOMPACTION with other methods: it sacrifices full-history recall while remaining training-free, drop-in, model-agnostic, cache-friendly, and free of auxiliary LLM calls.

CLIFFCOMPACTION addresses both context capacity and efficiency for long-horizon agents (Figure 1). It preserves agent performance while reducing cache-read costs by up to 90% and total cost by up to 50%. These savings make test-time scaling economically viable: on Terminal-Bench 2.0, three compacted Kimi K2.6 rollouts gain 10.5 percentage points at $1.9\times$ the cost of one uncompacted run, matching Opus 4.7 and exceeding Opus 4.6 and GPT-5.3 Codex at lower cost. CLIFFCOMPACTION also enables continual learning, achieving a $3.58\times$ speedup on KernelBench and outperforming specialized kernel-optimization methods by 25% despite being a general compaction method requiring no task-specific design. At equal spend, it delivers more performance than full-context settings, making it a practical compaction technique for long-horizon coding agents.

2 CLIFFCOMPACTION

CLIFFCOMPACTION makes three design choices. First, it compacts when the context crosses a token threshold. Second, it retains compact verbatim excerpts—rather than summaries—while dropping token-intensive portions of the conversation. Third, at each new compaction event, it discards previous compactions and compacts only the post-compaction turns since the last event. This leads to a "cliff" where context length drops sharply to roughly the same level after each compaction event.

This design aims to optimize two main objectives: (1) making compaction itself practical and efficient to deploy; (2) maximizing verbatim, unaltered context for as long as possible.

While the first objective simply avoids re-prefills, the second is more subtle. Using summaries has the problem that if *most* of the information is summarized correctly, the model assumes it has all the information and does not need to revisit past state, such as documentation or code files (Appendix D, Figure 6). This leads to subtle but continuous context drift if the summary does not preserve the right information. We call this issue low *compaction precision*, where high compaction precision avoids omission or distortion of past information in the context.

This often stands in trade-off with *compaction recall*: how much information can be retrieved over the course of many compactions.

CLIFFCOMPACTION can be seen as having low compaction recall, since it throws away all direct information after two compactions. Summaries have high recall since they can retrieve information from many past compactions. On the other hand, CLIFFCOMPACTION has high compaction precision, because it preserves user directives and model generations in full for two full compaction windows, while methods that use summaries have low compaction precision.

As such, the best view of CLIFFCOMPACTION is as a practically deployable algorithm that optimizes compaction precision at the cost of degrading compaction recall.

2.1 CLIFFCOMPACTION AND KV-CACHE

On the frontend level, LLM APIs typically distinguish between uncached input tokens, output tokens, and cached input tokens. On the backend systems level, cached input tokens correspond to KV-cache reads. While cache reads are the least expensive on a per-token basis, they are by far the most costly operation for long-context sessions that are typical for agents. This is because every request bears the cost of all past tokens while autoregressive token generation incurs only the cost of a few additional tokens. On the backend level, this cost is related to large memory reads for KV-caches, which usually take up more GPU memory than the model weights and which are bandwidth-bound—this is particularly expensive due to the current shortage and price of HBM memory.

Any modification to the context, including compaction and other context-management approaches, invalidates the existing KV-cache and requires re-prefilling the new context from the common prefix. API costs for uncached input tokens that require re-prefill are roughly 5 to 6 times as expensive as cached ones in the models we study (Table 8) and as such incur very high costs. Consequently, the efficiency gains of frontend context management often come at the expense of additional backend re-prefills. Thus frequent context changes need to be avoided for a practical, cost-efficient algorithm.

CLIFFCOMPACTION is designed to preserve cache efficiency by avoiding excessive re-prefills. Instead of actively managing the context throughout a trajectory, CLIFFCOMPACTION leaves it unchanged and lets it grow naturally, compacting only upon exhausting a preset budget. Since the existing context are never modified between compactions, the cache remains valid across each entire growth segment and is invalidated only at the compaction points themselves, hence the smooth cliff-shaped profile in Figure 2. While each compaction event unavoidably resets the cache and requires a re-prefill, the associated overhead stays modest for two reasons: the compacted context is dramatically smaller than the original, and compaction occurs only occasionally, so each re-prefill is amortized over a long stretch of full cache reuse.

2.2 CLIFFCOMPACTION AND TOKENS

We unpack the agent's context on Terminal-Bench 2.0 with GLM 5.1 to identify where tokens and dollars are concentrated (Table 6). Without compaction, tool results dominate the context at 56.0% of

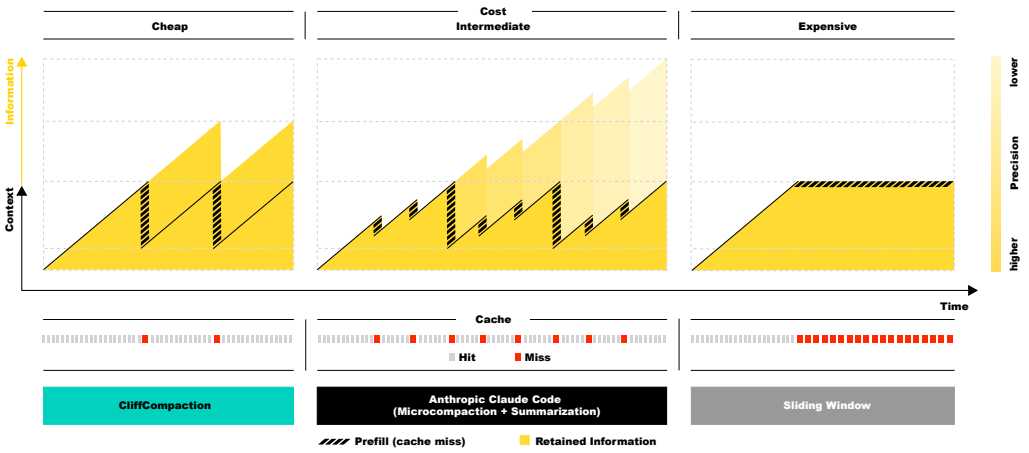


Figure 2: **Overview of CLIFFCOMPACTION.** CLIFFCOMPACTION strictly discards previous compactions and retains only fragments from the most recent window. The yellow area reflects recall (how much information stays retrievable in the context) and its shade reflects precision (fidelity of the retained information)—CLIFFCOMPACTION favors precision over recall.

tokens, followed by tool calls at 28.0%, together accounting for 84% of the total spent. This motivates CLIFFCOMPACTION to compact tool-related content by dropping long tool results and reducing tool calls to their signatures. If the model wants to retrieve past information, it can issue a new tool call from the preserved signature, so any removed context remains recoverable.

Tool results. CLIFFCOMPACTION applies a simple length-based rule to tool results. Those exceeding 500 characters are dropped, while shorter ones are retained. Short results are often useful and cheap to keep, such as grep matches, exit codes, and concise script output. Long results are typically reads of large files and contain information that can be recovered on demand.

Tool calls. CLIFFCOMPACTION reduces each call to a compact signature. Although available tools differ across scaffolds, it generally retains the tool name, the target file or path, and other essential arguments. Long contents embedded in tool calls, such as file-write calls that inline file contents, are removed, as the resulting file persists in the codebase and can be re-read whenever needed. By preserving tool calls verbatim, the agent can recover past context fully by making the same tool call.

Thoughts, system prompts, task description, and recent turns. Agent thoughts (analyses, plans, and hypotheses) are not a major source of context bloat, but we still truncate them to 300 characters. The system prompt and the first user message (the task description) are kept in full. We also keep the K most recent turns unchanged (Algorithm 1).

2.3 CLIFF BY CLIFF

Long trajectories may undergo multiple compactions, raising the question of how to manage previous compactions when a new one fires. CLIFFCOMPACTION makes a trade-off where we compact past sessions in such a way to maximize compaction precision—how much information to preserve verbatim—while limiting degradations in compaction recall—how much information is preserved from past sessions.

In more detail, the CLIFFCOMPACTION algorithm works as follows: After the $(t-1)$-th compaction, the agent's context S_t consists of the compacted history C_{t-1} followed by the live session L_t that grows on top of it, with $C_0 = \emptyset$. When S_t exceeds the context window, the t-th compaction triggers:

$$\begin{aligned} S_t &= C_{t-1} \oplus L_t, \\ C_t &= \textsc{CliffCompaction}(L_t), \\ S_{t+1} &= C_t \oplus L_{t+1}. \end{aligned}$$

C_{t-1} is discarded entirely rather than nested into C_t.

Algorithm 1 CLIFFCOMPACTION

```
function CliffCompaction(turns)
    old, recent ← turns[:−2K], turns[−2K:]                ▷ K turn pairs
    parts ← []
    for m ∈ old
        if m is a prior compaction
            skip                                          ▷ discard, keep flat
        else if m is Assistant
            parts += [Truncate(m.thinking, 300), Signature(m.toolcall, 150)]
        else if m is ToolResult and |m| ≤ 500
            parts += [m]
    return Join(parts), recent

function Query(messages)
    try return LLM(messages)
    except ContextWindowExceeded:
        s, x ← messages[0], messages[1]
        C, recent ← CliffCompaction(messages[2:])
        return LLM([s, x, C] ‖ recent)
```

Faithfulness—maximizing compaction precision. Common compaction schemes fold old summaries into new ones, so every cascading compaction is a summary of summaries. Compressing already partial content makes each compaction lossier than the last, and context quality decays over the trajectory. CLIFFCOMPACTION prevents this type of regression by discarding the previous compacted history completely. Each compaction compresses only the latest live session L_t, keeping every cliff equally faithful to the turns it condenses and maintaining high compaction precision throughout the trajectory.

Residual propagation limits degradation of compaction recall. While CLIFFCOMPACTION discards all information before the previous compaction, it still carries information forward. Every turn of L_t is generated with C_{t-1} in view, so the agent's actions and reasoning, and therefore the block C_t that compresses them, carry an implicit influence of the discarded context. This chain extends back to the start of the trajectory:

$$C_t \leftarrow L_t \leftarrow C_{t-1} \leftarrow L_{t-1} \leftarrow \cdots \leftarrow C_1 \leftarrow L_1.$$

No information is ever carried forward directly, but residual knowledge leaks from each session into the next through the agent's own behavior, softening the loss in compaction recall. Experimentally, this residual knowledge makes CLIFFCOMPACTION effective at long-context sessions over millions of tokens even though a compaction window of 128k tokens is used.

As such, the two effects trade recall for precision. Only the most recent window survives each compaction, so the context is cliff-shaped not only in tokens but also in information (Figure 2). Algorithm 1 summarizes CLIFFCOMPACTION.

3 CLIFFCOMPACTION MAINTAINS PERFORMANCE AT LOWER COST

3.1 EXPERIMENTAL SETUP

Benchmarks, Scaffolds, and Models. We evaluate on SWE-bench Verified (Jimenez et al., 2024) and Terminal-Bench (Merrill et al., 2026). For SWE-bench Verified, we use `mini-swe-agent` (Yang et al., 2024), which maintains an append-only conversation history without native conversation-level compaction, providing a clean baseline. We additionally use `OpenHands` (Wang et al., 2025b) to test whether the effect transfers across harnesses, replacing its native condenser with CLIFFCOMPACTION in the constrained-context settings. For Terminal-Bench 2.0, we use the benchmark's standard `Terminus-2` scaffold, substituting its built-in compaction with CLIFFCOMPACTION while leaving all other behavior unchanged. We also evaluate CLIFFCOMPACTION on Terminal-Bench 2.1 using `Claude Code`. We run on recent models from the Kimi

(K2.5, K2.6) (Team et al., 2026; Moonshot AI, 2026) and GLM (5, 5.1, 5 Turbo, 5.3 Flash, 4.7 Flash) (GLM-5 Team, 2026; Z.AI, 2026; GLM Team, 2025) families.

Compaction Thresholds and Integration. We compare full-context execution under each scaffold's default behavior against CLIFFCOMPACTION with token thresholds $B \in \{45\text{K}, 32\text{K}, 16\text{K}, 8\text{K}\}$. `Claude Code` is closed-source, so we implement CLIFFCOMPACTION as a scaffold-agnostic API proxy that reduces every tool call to a generic name-and-arguments signature.

3.2 RESULTS

Table 2: Performance and cost on Terminal-Bench across context length budgets with CLIFFCOMPACTION, evaluated with `Terminus-2` on Terminal-Bench 2.0 and `Claude Code` on Terminal-Bench 2.1. †No compaction is applied.

`Terminus-2`									
		Full context†		32K		16K		8K	
Model	Compaction	% Resolved	Cost	% Resolved	Cost	% Resolved	Cost	% Resolved	Cost
Kimi K2.6	Summarization	$59.16_{\pm 3.41}$	\$0.40	$58.36_{\pm 2.57}$	\$0.24	$55.45_{\pm 0.64}$	\$0.26	$42.97_{\pm 4.41}$	\$0.60
	CLIFFCOMPACTION			$61.42_{\pm 4.25}$	\$0.24	$61.42_{\pm 1.30}$	\$0.19	$50.00_{\pm 2.40}$	\$0.25
GLM 5.1	CLIFFCOMPACTION	$49.83_{\pm 3.42}$	\$0.54	$53.20_{\pm 5.31}$	\$0.34	$54.33_{\pm 2.35}$	\$0.27	$44.93_{\pm 3.35}$	\$0.19
`Claude Code`									
		200K				45K			
Model	Compaction	% Resolved		Cost		% Resolved		Cost	
GLM 5.3 Flash	Summarization	$73.03_{\pm 3.04}$		\$0.21		$70.97_{\pm 3.72}$		\$0.14	
	CLIFFCOMPACTION	—		—		$76.69_{\pm 1.41}$		\$0.16	

Table 3: Performance and cost on SWE-bench Verified across context length budgets, evaluated with `mini-swe-agent` and `OpenHands`. Cost is average per-instance USD. †No compaction is applied. ‡`OpenHands`' native condenser is active.

`mini-swe-agent`								
	Full context†		32K		16K		8K	
Model	% Resolved	Cost	% Resolved	Cost	% Resolved	Cost	% Resolved	Cost
Kimi K2.6	$73.87_{\pm 0.31}$	\$0.19	$73.27_{\pm 0.12}$	\$0.18	$71.87_{\pm 1.31}$	\$0.17	$67.60_{\pm 1.06}$	\$0.18
Kimi K2.5	$70.87_{\pm 1.01}$	\$0.12	$70.98_{\pm 0.98}$	\$0.11	$69.53_{\pm 1.33}$	\$0.08	$64.93_{\pm 1.33}$	\$0.10
GLM 5.1	$71.40_{\pm 0.72}$	\$0.25	$68.80_{\pm 1.59}$	\$0.20	$69.33_{\pm 1.10}$	\$0.15	$65.13_{\pm 0.76}$	\$0.13
GLM 5 Turbo	$69.80_{\pm 1.06}$	\$0.19	$70.53_{\pm 0.42}$	\$0.16	$67.53_{\pm 1.14}$	\$0.13	$63.27_{\pm 1.22}$	\$0.11
GLM 5	$69.80_{\pm 1.20}$	\$0.26	$68.87_{\pm 0.81}$	\$0.23	—	—	—	—
GLM 4.7 Flash	$43.20_{\pm 1.27}$	—	$43.60_{\pm 0.99}$	—	$41.20_{\pm 0.42}$	—	$30.70_{\pm 2.97}$	—
`OpenHands`								
	Full context‡				32K			
Model	% Resolved		Cost		% Resolved		Cost	
Kimi K2.6	$71.30_{\pm 0.99}$		\$0.57		$70.20_{\pm 0.57}$		\$0.42	
GLM 5.1	$71.93_{\pm 1.21}$		\$1.01		$72.20_{\pm 0.69}$		\$0.69	

Terminal-Bench. On Terminal-Bench 2.0, CLIFFCOMPACTION matches or improves upon the full-context setting at moderate thresholds. For Kimi K2.6, both the 32K and 16K settings achieve 61.42%, exceeding the full-context baseline of 59.16% by 2.26 percentage points. At 16K, replacing `Terminus-2`'s native LLM-based summarization with CLIFFCOMPACTION increases success from 55.45% to 61.42%. GLM 5.1 shows a similar benefit, improving from 49.83% at full context to 53.20% at 32K and 54.33% at 16K. These gains suggest that removing stale context can sometimes improve agent performance rather than simply reducing memory pressure.

On Terminal-Bench 2.1, at a matched $\sim$45K mean peak context, CLIFFCOMPACTION brings GLM 5.3 Flash under `Claude Code` to 76.69%, exceeding `Claude Code`'s auto-compaction at the same budget (70.97%) and its default 200K configuration (73.03%), and doing so with markedly lower seed variance. That CLIFFCOMPACTION still outperforms a scaffold-native summarizer highlights that it does not depend on scaffold-specific structure and can be deployed against a closed agent.

SWE-bench Verified. CLIFFCOMPACTION preserves most full-context performance at moderate thresholds. Kimi K2.6 drops only from 73.87% to 73.27% at 32K and stays within 2.0 points at 16K; GLM 5.1 and GLM 5 Turbo stay within roughly two points at 16K. On both benchmarks, the 8K setting produces larger drops and is best viewed as a stress test.

Cost savings and test-time scaling. CLIFFCOMPACTION lowers rollout cost (Appendix A.1), making test-time scaling worthwhile for open-weight models. Without compaction, scaling Kimi K2.6 to k=3 costs more than Opus 4.6 and GPT-5.3 Codex, yet trails GPT-5.3 Codex on Terminal-Bench 2.0 (Figure 3c). With CLIFFCOMPACTION, scaling gains over 10 points for less than the cost of two full-context runs, exceeding Opus 4.6 and GPT-5.3 Codex at lower cost (Appendix B).

4 CONTINUAL LEARNING WITH CLIFFCOMPACTION

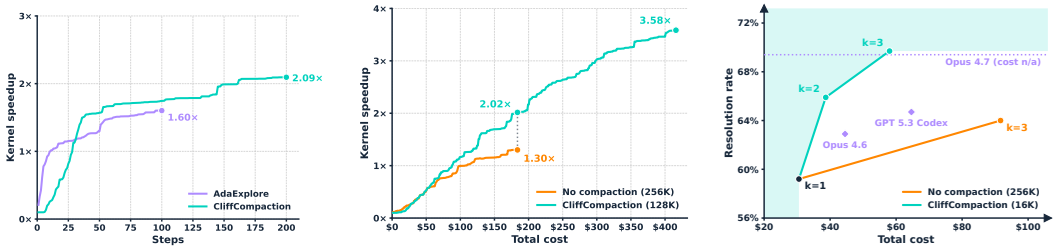
Having established that CLIFFCOMPACTION maintains or improves performance across benchmarks, scaffolds, and scales, we investigate whether it supports continual learning over long horizons and many compactions. We evaluate on KernelBench Level 3 (Ouyang et al., 2025), where a single trajectory can exceed 1M tokens, 15–70$\times$ the median SWE-bench or Terminal-Bench trajectory.

Setup. We use `OpenHands` and let the agent optimize kernels until it exhausts its step budget or, without compaction, its context window. Following Du et al. (2026), we report *Speedup*, the geometric mean speedup over PyTorch Eager; *Acc.*, the percentage of problems with a correct kernel; and $> p$, the percentage with speedup above p. We run Kimi K2.7 writing CUDA and GPT-5-mini writing Triton on an L40S, and Kimi K2.6 writing CUDA on an RTX Pro 6000. Each configuration matches the language and correctness tolerance of the baseline it is compared against (Appendix E.2).

Table 4: KernelBench Level 3 results. Top: specialized kernel-optimization agents (†as reported in the original papers; ‡CUDA-Agent reports the geometric mean over correct solutions only). Middle: CLIFFCOMPACTION on L40S; L40S/A6000 are essentially the same hardware, differing only in ECC support. Bottom: CLIFFCOMPACTION on RTX Pro 6000 Blackwell. We report both steps and the number of kernels generated, since search-based methods propose one kernel candidate per step whereas agentic setups spend some steps exploring without writing kernel code.

Method	Model	Scaffold	Speedup	Acc.	$> 1.2\times$	$> 2\times$	Steps	Kernels	Hardware
DR.Kernel†	Qwen3 14B (RL)	custom	0.97$\times$	84%	16%	8%	—	56	L40S/A6000
Iterative Refinement†	GPT-5-mini	custom	1.31$\times$	100%	24%	12%	—	50	L40S/A6000
OpenEvolve+Memory†	GPT-5-mini	custom	1.47$\times$	100%	28%	10%	—	50	L40S/A6000
AdaExplore†	GPT-5-mini	custom	1.55$\times$	100%	28%	16%	—	50	L40S/A6000
AdaExplore†	GPT-5-mini	custom	1.78$\times$	100%	36%	22%	—	200	L40S/A6000
CUDA-Agent†‡	Seed 1.6 (RL)	OpenHands	1.80$\times$	94%	—	—	200	—	H20
CLIFFCOMPACTION (128K)	GPT-5-mini	OpenHands	1.57$\times$	96%	60%	36%	50	15	L40S/A6000
CLIFFCOMPACTION (128K)	GPT-5-mini	OpenHands	2.09$\times$	100%	78%	52%	200	82	L40S/A6000
CLIFFCOMPACTION (128K)	GPT-5-mini	OpenHands	2.21$\times$	100%	82%	54%	400	166	L40S/A6000
No compaction (256K)	Kimi K2.7	OpenHands	1.30$\times$	86%	66%	50%	200	—	L40S/A6000
CLIFFCOMPACTION (128K)	Kimi K2.7	OpenHands	2.23$\times$	94%	88%	54%	200	—	L40S/A6000
CLIFFCOMPACTION (128K)	Kimi K2.7	OpenHands	3.58$\times$	96%	92%	86%	400	—	L40S/A6000
No compaction (256K)	Kimi K2.6	OpenHands	1.51$\times$	96%	72%	32%	200	—	RTX 6000
CLIFFCOMPACTION (200K)	Kimi K2.6	OpenHands	1.85$\times$	96%	78%	40%	200	—	RTX 6000
CLIFFCOMPACTION (128K)	Kimi K2.6	OpenHands	1.87$\times$	96%	78%	46%	200	—	RTX 6000
CLIFFCOMPACTION (128K)	Kimi K2.6	OpenHands	2.48$\times$	98%	90%	64%	400	—	RTX 6000

CLIFFCOMPACTION sustains improvement beyond the context limit. Without compaction, the agent exhausts its 256K context well before the 200-step budget (98% of L40S and 76% of RTX Pro 6000 runs stop early, at medians of 99 and 122 steps), capping speedup at 1.30$\times$ and 1.51$\times$.



(a) vs. AdaExplore. (b) vs. full context. (c) Test-time scaling.

Figure 3: Best-so-far KernelBench speedup (a, b) and test-time scaling on Terminal-Bench 2.0 (c).

CLIFFCOMPACTION lifts these to 2.23× and 1.87× at 200 steps, and to 3.58× and 2.48× at 400 (Table 4). The gains are broad with more steps: >2× rises from 50% to 86% on L40S and doubles from 32% to 64% on RTX Pro 6000. Both the 200K and 128K thresholds beat the 256K baseline.

CLIFFCOMPACTION's bounded context is cheaper to scale. Complementing the parallel test-time scaling (Figure 3c), a bounded context also makes sequential scaling cheaper: at equal total cost, CLIFFCOMPACTION reaches 2.02× against 1.30× without compaction (Figure 3b).

CLIFFCOMPACTION outperforms systems purpose-built for kernel optimization. At 200 steps, CLIFFCOMPACTION reaches 2.09× against 1.78× for AdaExplore on GPT-5-mini and 2.23× against 1.80× for CUDA-Agent. It clears 1.2× and 2× on 78% and 52% of problems, more than double AdaExplore's 36% and 22%, while proposing fewer kernels (Appendix D.2). Although it starts behind AdaExplore, which begins with a learned skill memory, it overtakes around step 31 and keeps improving (Figure 3a), gaining +0.52× from 50 to 200 steps versus AdaExplore's +0.23×.

5 CLIFFCOMPACTION VS. EXISTING COMPACTION STRATEGIES

Table 5: Performance and cost of compaction methods across benchmarks. SWE-bench Verified (16K) uses `mini-swe-agent`. Δ is change in cost vs. full-context ($0.24 Kimi, $0.15 GLM). KernelBench Level 3 uses `OpenHands` (128K) and a 400-step budget.

	SWE-bench Verified						KernelBench L3		
	Kimi K2.7			GLM 5.2			Kimi K2.7		
Method	% Resolved	Cost	Δ	% Resolved	Cost	Δ	Speedup	Kernels	Cost
Sliding window	$72.80_{\pm 2.43}$	$0.26	+9%	$72.87_{\pm 0.99}$	$0.16	+7%	2.86×	118	$21.52
Summarization	$70.27_{\pm 0.95}$	$0.18	−23%	$70.27_{\pm 1.01}$	$0.12	−23%	3.47×	136	$8.10
+ Microcompaction	$71.00_{\pm 0.69}$	$0.20	−17%	$72.73_{\pm 2.08}$	$0.12	−23%	3.33×	129	$12.84
CLIFFCOMPACTION	$71.33_{\pm 0.50}$	$0.19	−21%	$72.60_{\pm 1.22}$	$0.12	−23%	3.58×	116	$8.32

Setup. We compare CLIFFCOMPACTION against alternative context-management strategies (Table 5). (1) **Sliding window** drops the oldest turns, keeping a fixed system/task prefix plus the most recent turns that fit within the context budget. (2) **Summarization** replaces the older turns with an LLM-generated summary, using Claude Code's summarization prompt. (3) **Summarization + Microcompaction** is a two-tier pipeline that reimplements Claude Code's scheme: (i) *microcompaction* clears the contents of old tool observations while preserving the corresponding tool calls and the 5 most recent observations verbatim, adapting Anthropic's `clear_tool_uses` context-editing strategy (Anthropic, 2025); (ii) when microcompaction is insufficient, an LLM *summarizer* rewrites the older turns into a structured summary using Claude Code's summarization prompt. Microcompaction fires whenever at least τ tokens of reclaimable observation content have accumulated, with τ=20,000 at a 128K context budget and τ=4,000 at 16K. We adopt this configuration because it performs well on both quality and cost in our SWE-bench runs, making it a strong baseline.

CLIFFCOMPACTION is the only method that stays competitive on quality while remaining cheap on every benchmark. On SWE-bench Verified, where trajectories are short, all four methods fall within 2.6 points of one another. Sliding window scores highest on both models but only marginally, and it is the only method that makes the agent *more* expensive than running with full context ($+9\%$ and $+7\%$). Summarization is the weakest on quality for both models, echoing the pattern on Terminal-Bench (Table 2). On KernelBench Level 3, with much longer trajectories, the methods separate more clearly: CLIFFCOMPACTION reaches $3.58\times$ at \$8.32 per problem with the fewest kernel candidates (Appendix D.2), vs. $3.47\times$ at \$8.10 for summarization, $3.33\times$ at \$12.84 with microcompaction, and $2.86\times$ at \$21.52 for sliding window. Across all three benchmarks, CLIFFCOMPACTION is the only technique that both maintains strong performance and remains inexpensive.

6 RELATED WORK

Context management. Context management has long been an important research direction for long-horizon coding agents. Leading agent scaffolds manage growing trajectories by discarding earlier observations, implementing observation-level sliding-window compaction (Yang et al., 2024; Wang et al., 2025b). Although simple, naively sliding the context can invalidate the KV cache and incur substantial re-prefill overhead. LLM summarization (Wang et al., 2025b; Kang et al., 2025; Verma, 2026; Li et al., 2026b; Wang et al., 2025a; Li et al., 2026a) is common but grows increasingly lossy across repeated rounds, as details are progressively distorted or omitted through chains of summaries. Trained compression (Sun et al., 2026; Kang et al., 2025; Shi et al., 2025; Pan et al., 2024; Jiang et al., 2026) adds training complexity and may not transfer across models. External-memory and retrieval-based approaches (Packer et al., 2024; Wang et al., 2026) preserve access to information outside the active context, but are not drop-in solutions—they require retrieval infrastructure and remain vulnerable to retrieval failures. Similarly, sub-agent-based approaches (Gandhi et al., 2026; Zhang et al., 2024) introduce orchestration complexity and may require specialized prompting or training to determine when and how auxiliary agents should be invoked. Structure-based eviction (Semenov & Dorofeev, 2026) is deterministic but requires scaffold-specific instrumentation and is cache-inefficient.

Test-time scaling for code agents. Recent work improves coding agent performance by spending additional inference-time compute through parallel sampling or sequential refinement with test-based selection (Li et al., 2025a; Hassid et al., 2024), ensemble reasoning (Gao et al., 2025), branching from prior trajectories (Ding & Zhang, 2026), or compact trajectory summaries (Kim et al., 2026). These works largely treat context management and test-time scaling as separate axes; we instead study how an agent equipped with CLIFFCOMPACTION responds to additional inference-time compute.

Continual learning for agents. Enabling coding agents to continually learn from experience is an important research direction. Some work trains agents to improve their solutions through iterative refinement (Baronio et al., 2026; Dai et al., 2026). Other approaches store prior experiences in external memory and retrieve relevant examples or strategies for new tasks (Shinn et al., 2023; Wu et al., 2026; Qian et al., 2024; Chen et al., 2024; Ouyang et al., 2026). Evolutionary approaches maintain populations of candidate programs and use LLMs to iteratively propose and select mutations (Novikov et al., 2025; Li et al., 2025b). However, many of these methods rely on task-specific training, memory structures, or optimization procedures, limiting their generalizability to new tasks.

7 LIMITATIONS

The benefits CLIFFCOMPACTION brings depend on the scaffold and on the task. Across our experiments, cost savings grow with the complexity of the scaffold itself, and so does the budget the scaffold needs in order to work: the system prompt, tool definitions, and other fixed components consume part of the budget before the agent has taken a single action. Thus, the minimum threshold an agent needs varies between scaffolds. Similarly, the benefit is meaningful only for medium-to-long-horizon tasks.

Context management is a broad problem that has been approached in many different ways. Our comparison is limited to methods that act on the conversation history itself at inference time. We do not compare against approaches that train the model to manage its own context, or that maintain an external memory store the agent writes to and retrieves from. Such methods address the same problem from a different direction, and we did not have the resources to compare against them.

AI USE STATEMENT

We used coding agents (Claude Code) primarily to assist with coding tasks in this work. They provided feedback on and suggested parameters for our experiments; implemented the code for running the experiments; wrote the code for plotting the figures and extracting statistics from the logs; suggested relevant literature; and refined the writing and adjusted parts of the layout to improve readability. The ideas and designs, as well as the hypotheses and analysis, are the authors' own. The authors reviewed all AI-assisted artifacts and take responsibility for the final content of this work.

REPRODUCIBILITY STATEMENT

We provide anonymized source code for implementing CLIFFCOMPACTION as supplementary material. Section 2 describes CLIFFCOMPACTION in detail, and Sections 3, 4, and 5, as well as Appendix B, describe the experimental setups. Additional implementation details, hyperparameters, and evaluation protocols are provided in Appendices A.1, E.1, and E.2.

REFERENCES

Anthropic. Context editing, 2025. URL https://platform.claude.com/docs/en/build-with-claude/context-editing. Accessed 2026-08-15.

Carlo Baronio, Pietro Marsella, Ben Pan, Simon Guo, and Silas Alberti. Kevin: Multi-turn RL for generating CUDA kernels. In *The Fourteenth International Conference on Learning Representations*, 2026. URL https://openreview.net/forum?id=xu1XwVZtDi.

Minghao Chen, Yihang Li, Yanting Yang, Shiyu Yu, Binbin Lin, and Xiaofei He. Automanual: Constructing instruction manuals by llm agents via interactive environmental learning. In A. Globerson, L. Mackey, D. Belgrave, A. Fan, U. Paquet, J. Tomczak, and C. Zhang (eds.), *Advances in Neural Information Processing Systems*, volume 37, pp. 589–631. Curran Associates, Inc., 2024. doi: 10.52202/079017-0019. URL https://proceedings.neurips.cc/paper_files/paper/2024/file/0142921fad7ef9192bd87229cdafa9d4-Paper-Conference.pdf.

Weinan Dai, Hanlin Wu, Qiying Yu, Huan-ang Gao, Jiahao Li, Chengquan Jiang, Weiqiang Lou, Yufan Song, Hongli Yu, Jiaze Chen, et al. Cuda agent: Large-scale agentic rl for high-performance cuda kernel generation. *arXiv preprint arXiv:2602.24286*, 2026.

Tri Dao, Dan Fu, Stefano Ermon, Atri Rudra, and Christopher Ré. Flashattention: Fast and memory-efficient exact attention with io-awareness. *Advances in neural information processing systems*, 35: 16344–16359, 2022.

DeepSeek-AI. Deepseek-v3.2: Pushing the frontier of open large language models, 2025. URL https://arxiv.org/abs/2512.02556.

Yifeng Ding and Lingming Zhang. Swe-replay: Efficient test-time scaling for software engineering agents. *arXiv preprint arXiv:2601.22129*, 2026.

Weihua Du, Jingming Zhuo, Yixin Dong, Andre Wang He, Weiwei Sun, Zeyu Zheng, Manupa Karunaratne, Ivan Fox, Tim Dettmers, Tianqi Chen, et al. Adaexplore: Failure-driven adaptation and diversity-preserving search for efficient kernel generation. *arXiv preprint arXiv:2604.16625v1*, 2026. URL https://arxiv.org/abs/2604.16625v1.

Ryan Ehrlich, Bradley Brown, Jordan Juravsky, Ronald Clark, Christopher Ré, and Azalia Mirhoseini. Codemonkeys: Scaling test-time compute for software engineering, 2025. URL https://arxiv.org/abs/2501.14723.

Apurva Gandhi, Satyaki Chakraborty, Xiangjun Wang, Aviral Kumar, and Graham Neubig. Recursive agent optimization, 2026. URL https://arxiv.org/abs/2605.06639.

Pengfei Gao, Zhao Tian, Xiangxin Meng, Xinchen Wang, Ruida Hu, Yuanan Xiao, Yizhou Liu, Zhao Zhang, Junjie Chen, Cuiyun Gao, et al. Trae agent: An llm-based agent for software engineering with test-time scaling. *arXiv preprint arXiv:2507.23370*, 2025.

GLM-5 Team. Glm-5: from vibe coding to agentic engineering, 2026. URL https://arxiv.org/abs/2602.15763.

GLM Team. Glm-4.5: Agentic, reasoning, and coding (arc) foundation models, 2025. URL https://arxiv.org/abs/2508.06471.

Michael Hassid, Tal Remez, Jonas Gehring, Roy Schwartz, and Yossi Adi. The larger the better? improved llm code-generation via budget reallocation. *arXiv preprint arXiv:2404.00725*, 2024.

Haitao Jiang, Lin Ge, Hengrui Cai, and Rui Song. Pabu: Progress-aware belief update for efficient llm agents, 2026. URL https://arxiv.org/abs/2602.09138.

Carlos E Jimenez, John Yang, Alexander Wettig, Shunyu Yao, Kexin Pei, Ofir Press, and Karthik R Narasimhan. SWE-bench: Can language models resolve real-world github issues? In *The Twelfth International Conference on Learning Representations*, 2024. URL https://openreview.net/forum?id=VTF8yNQM66.

Minki Kang, Wei-Ning Chen, Dongge Han, Huseyin A Inan, Lukas Wutschitz, Yanzhi Chen, Robert Sim, and Saravan Rajmohan. Acon: Optimizing context compression for long-horizon llm agents. *arXiv preprint arXiv:2510.00615*, 2025.

Joongwon Kim, Wannan Yang, Kelvin Niu, Hongming Zhang, Yun Zhu, Eryk Helenowski, Ruan Silva, Zhengxing Chen, Srinivasan Iyer, Manzil Zaheer, et al. Scaling test-time compute for agentic coding. *arXiv preprint arXiv:2604.16529*, 2026.

Jacky Kwok, Shulu Li, Pranav Atreya, Yuejiang Liu, Yixing Jiang, Chelsea Finn, Marco Pavone, Ion Stoica, and Azalia Mirhoseini. Llm-as-a-verifier: A general-purpose verification framework, 2026. URL https://arxiv.org/abs/2607.05391.

Dacheng Li, Shiyi Cao, Chengkun Cao, Xiuyu Li, Shangyin Tan, Kurt Keutzer, Jiarong Xing, Joseph E Gonzalez, and Ion Stoica. S*: Test time scaling for code generation. *arXiv preprint arXiv:2502.14382*, 1(2), 2025a.

Kefan Li, Hongyue Yu, Tingyu Guo, Shijie Cao, and Yuan Yuan. Cocoevo: Co-evolution of programs and test cases to enhance code generation, 2025b. URL https://arxiv.org/abs/2502.10802.

Tianjian Li, Jingyu Zhang, William Jurayj, Xi Wang, Chuanyang Jin, Mehrdad Farajtabar, Eric Nalisnick, and Daniel Khashabi. Self-compacting language model agents, 2026a. URL https://arxiv.org/abs/2606.23525.

Xiaoxi Li, Wenxiang Jiao, Jiarui Jin, Guanting Dong, Jiajie Jin, Yinuo Wang, Hao Wang, Yutao Zhu, Ji-Rong Wen, Yuan Lu, and Zhicheng Dou. Deepagent: A general reasoning agent with scalable toolsets. In *Proceedings of the ACM Web Conference 2026 (WWW '26)*, WWW '26, pp. 1–12, New York, NY, USA, 2026b. Association for Computing Machinery. ISBN 9798400723070. doi: 10.1145/3774904.3792460. URL https://doi.org/10.1145/3774904.3792460.

Mike A Merrill, Alexander G Shaw, Nicholas Carlini, Boxuan Li, Harsh Raj, Ivan Bercovich, Lin Shi, Jeong Yeon Shin, Thomas Walshe, E Kelly Buchanan, et al. Terminal-bench: Benchmarking agents on hard, realistic tasks in command line interfaces. *arXiv preprint arXiv:2601.11868*, 2026.

Moonshot AI. Kimi k2.6. https://huggingface.co/moonshotai/Kimi-K2.6, 2026. Open-weight model under Modified MIT License.

Alexander Novikov, Ngân Vũ, Marvin Eisenberger, Emilien Dupont, Po-Sen Huang, Adam Zsolt Wagner, Sergey Shirobokov, Borislav Kozlovskii, Francisco J. R. Ruiz, Abbas Mehrabian, M. Pawan Kumar, Abigail See, Swarat Chaudhuri, George Holland, Alex Davies, Sebastian Nowozin, Pushmeet Kohli, and Matej Balog. Alphaevolve: A coding agent for scientific and algorithmic discovery, 2025. URL https://arxiv.org/abs/2506.13131.

Anne Ouyang, Simon Guo, Simran Arora, Alex L. Zhang, William Hu, Christopher Ré, and Azalia Mirhoseini. Kernelbench: Can llms write efficient gpu kernels?, 2025. URL https://arxiv.org/abs/2502.10517.

Siru Ouyang, Jun Yan, I-Hung Hsu, Yanfei Chen, Ke Jiang, Zifeng Wang, Rujun Han, Long Le, Samira Daruki, Xiangru Tang, Vishy Tirumalashetty, George Lee, Mahsan Rofouei, Hangfei Lin, Jiawei Han, Chen-Yu Lee, and Tomas Pfister. Reasoningbank: Scaling agent self-evolving with reasoning memory. In *The Fourteenth International Conference on Learning Representations*, 2026. URL https://openreview.net/forum?id=jL7fwchScm.

Charles Packer, Sarah Wooders, Kevin Lin, Vivian Fang, Shishir G. Patil, Ion Stoica, and Joseph E. Gonzalez. Memgpt: Towards llms as operating systems, 2024. URL https://arxiv.org/abs/2310.08560.

Zhuoshi Pan, Qianhui Wu, Huiqiang Jiang, Menglin Xia, Xufang Luo, Jue Zhang, Qingwei Lin, Victor Rühle, Yuqing Yang, Chin-Yew Lin, H. Vicky Zhao, Lili Qiu, and Dongmei Zhang. LLMLingua-2: Data distillation for efficient and faithful task-agnostic prompt compression. In Lun-Wei Ku, Andre Martins, and Vivek Srikumar (eds.), *Findings of the Association for Computational Linguistics: ACL 2024*, pp. 963–981, Bangkok, Thailand, August 2024. Association for Computational Linguistics. doi: 10.18653/v1/2024.findings-acl.57. URL https://aclanthology.org/2024.findings-acl.57/.

Chen Qian, Yufan Dang, Jiahao Li, Wei Liu, Zihao Xie, YiFei Wang, Weize Chen, Cheng Yang, Xin Cong, Xiaoyin Che, Zhiyuan Liu, and Maosong Sun. Experiential co-learning of software-developing agents. In Lun-Wei Ku, Andre Martins, and Vivek Srikumar (eds.), *Proceedings of the 62nd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pp. 5628–5640, Bangkok, Thailand, August 2024. Association for Computational Linguistics. doi: 10.18653/v1/2024.acl-long.305. URL https://aclanthology.org/2024.acl-long.305/.

Andrew Semenov and Svyatoslav Dorofeev. Beyond compaction: Structured context eviction for long-horizon agents, 2026. URL https://arxiv.org/abs/2606.11213.

Ethan Shen, Daniel Tormoen, Saurabh Shah, Ali Farhadi, and Tim Dettmers. SERA: Soft-verified efficient repository agents. In *Forty-third International Conference on Machine Learning*, 2026. URL https://openreview.net/forum?id=GHKj1XcSlt.

Yuling Shi, Yichun Qian, Hongyu Zhang, Beijun Shen, and Xiaodong Gu. Longcodezip: Compress long context for code language models. In *2025 40th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pp. 141–153. IEEE Press, 2025. doi: 10.1109/ASE63991.2025.00020. URL https://doi.org/10.1109/ASE63991.2025.00020.

Noah Shinn, Federico Cassano, Ashwin Gopinath, Karthik Narasimhan, and Shunyu Yao. Reflexion: language agents with verbal reinforcement learning. In A. Oh, T. Naumann, A. Globerson, K. Saenko, M. Hardt, and S. Levine (eds.), *Advances in Neural Information Processing Systems*, volume 36, pp. 8634–8652. Curran Associates, Inc., 2023. URL https://proceedings.neurips.cc/paper_files/paper/2023/file/1b44b878bb782e6954cd888628510e90-Paper-Conference.pdf.

Weiwei Sun, Miao Lu, Zhan Ling, Kang Liu, Xuesong Yao, Yiming Yang, and Jiecao Chen. Scaling long-horizon agent via context folding. In *Forty-third International Conference on Machine Learning*, 2026. URL https://openreview.net/forum?id=lNRgWoGfYg.

Kimi Team, Tongtong Bai, Yifan Bai, Yiping Bao, SH Cai, Yuan Cao, Y Charles, HS Che, Cheng Chen, Guanduo Chen, et al. Kimi k2. 5: Visual agentic intelligence. *arXiv preprint arXiv:2602.02276*, 2026.

Nikhil Verma. Active context compression: Autonomous memory management in llm agents, 2026. URL https://arxiv.org/abs/2601.07190.

Qingyue Wang, Yanhe Fu, Yanan Cao, Shuai Wang, Zhiliang Tian, and Liang Ding. Recursively summarizing enables long-term dialogue memory in large language models. *Neurocomputing*, 639: 130193, 2025a. doi: 10.1016/j.neucom.2025.130193.

Xingyao Wang, Boxuan Li, Yufan Song, Frank F Xu, Xiangru Tang, Mingchen Zhuge, Jiayi Pan, Yueqi Song, Bowen Li, Jaskirat Singh, et al. Openhands: An open platform for ai software developers as generalist agents. In *International Conference on Learning Representations*, volume 2025, pp. 65882–65919, 2025b.

Zhenting Wang, Huancheng Chen, Jiayun Wang, and Wei Wei. Memex(rl): Scaling long-horizon llm agents via indexed experience memory, 2026. URL https://arxiv.org/abs/2603.04257.

Rong Wu, Xiaoman Wang, Jianbiao Mei, Pinlong Cai, Daocheng Fu, Cheng Yang, Licheng Wen, Xuemeng Yang, Yufan Shen, Yuxin Wang, and Botian Shi. From interactions to principles: Experience-driven self-distillation for evolving LLM agents. In *Forty-third International Conference on Machine Learning*, 2026. URL https://openreview.net/forum?id=BamDfCP5R3.

John Yang, Carlos E Jimenez, Alexander Wettig, Kilian Lieret, Shunyu Yao, Karthik R Narasimhan, and Ofir Press. SWE-agent: Agent-computer interfaces enable automated software engineering. In *The Thirty-eighth Annual Conference on Neural Information Processing Systems*, 2024. URL https://arxiv.org/abs/2405.15793.

Z.AI. Glm-5.3-flash. https://huggingface.co/zai-org/GLM-5.3-Flash, 2026.

Yusen Zhang, Ruoxi Sun, Yanfei Chen, Tomas Pfister, Rui Zhang, and Sercan Ö. Arı k. Chain of agents: Large language models collaborating on long-context tasks. In A. Globerson, L. Mackey, D. Belgrave, A. Fan, U. Paquet, J. Tomczak, and C. Zhang (eds.), *Advances in Neural Information Processing Systems*, volume 37, pp. 132208–132237. Curran Associates, Inc., 2024. doi: 10.52202/079017-4202. URL https://proceedings.neurips.cc/paper_files/paper/2024/file/ee71a4b14ec26710b39ee6be113d7750-Paper-Conference.pdf.

A Extended Efficiency Analysis

A.1 Cost Savings with CliffCompaction

Since cost is a primary consideration in real-world agent deployments, we assess the efficiency of CliffCompaction through cost. Tables 3 and 2 report the average cost per instance.

Although API providers report token usage and cache statistics during agent execution, we observed substantial variation in cache-hit accounting across providers. To ensure a fair comparison, we recompute all costs using a perfect-caching model that derives cache hits directly from prompt lengths rather than provider-reported `cached_tokens` fields. Given a sequence of prompt lengths across API calls, $[P_1, P_2, \ldots, P_N]$, the first call is treated as a cold start with all P_1 tokens counted as non-cached. For each subsequent call $k > 1$, if $P_k \geq P_{k-1}$, we assume the previous prompt is fully cached, with P_{k-1} cached tokens and $P_k - P_{k-1}$ non-cached tokens. In contrast, if $P_k < P_{k-1}$, we treat the reduction in prompt length as a compaction event that invalidates the KV cache, and all P_k tokens are counted as non-cached. Completion tokens are not subject to caching and are counted per call at each provider's output price. This deterministic, model-agnostic procedure ensures that identical prompt sequences produce identical cache-hit rates regardless of provider, allowing us to isolate the cost effects of compaction from provider-specific caching behavior.

Overall Cost Savings. CliffCompaction consistently reduces cost across models, scaffolds, benchmarks, and context budgets. The magnitude of the savings depends on the model, with GLM generally benefiting more than Kimi. Savings also depend on the scaffold: on SWE-bench Verified, the gap between limited-context and unlimited-context settings is far larger for `OpenHands` than for `mini-swe-agent` (25.8%–29.7% versus 6.8%–19.1% at the same 32K budget). Benchmark-wise, the largest savings are observed on Terminal-Bench, where CliffCompaction reduces cost by 25.0%–52.1% for Kimi K2.6 and 35.8%–64.6% for GLM 5.1, substantially exceeding the reductions observed on SWE-bench Verified. We also compare against `Terminus-2` agent-based summarization on Kimi K2.6 (Table 2). At a 32K budget, both approaches incur similar costs because compaction is triggered infrequently. Under tighter budgets, compaction kicks in at a substantially higher rate, and the additional summarization calls introduce non-trivial overhead: at 8K, `Terminus-2` summarization costs $0.60 per instance against $0.25 for CliffCompaction, while also resolving fewer tasks. We compare against further compaction baselines on long-horizon trajectories in Section 5.

Table 6: Token and cost attribution by context component, before and after compaction (Terminal-Bench 2.0, GLM 5.1). *Other* combines uncached input with output.

		No compaction			CLIFFCOMPACTION @16K		
Component	% of Context	Cached	Other	Total	Cached	Other	Total
Tool results	56.0%	\$0.237	\$0.029	\$0.267	\$0.053	\$0.050	\$0.103 $_{(-61\%)}$
Tool calls	28.0%	\$0.127	\$0.062	\$0.189	\$0.012	\$0.067	\$0.079 $_{(-58\%)}$
Thoughts	13.6%	\$0.054	\$0.030	\$0.084	\$0.013	\$0.062	\$0.075 $_{(-11\%)}$
System & task	2.4%	\$0.008	\$0.001	\$0.009	\$0.010	\$0.001	\$0.011 $_{(+22\%)}$
Total		\$0.427	\$0.121	\$0.548	\$0.087	\$0.181	\$0.268 $_{(-51\%)}$

Cost Savings Breakdown. A well-designed context-management strategy should not let re-prefilling costs overshadow the cache savings it unlocks. By decomposing the cost, we show that CLIFFCOMPACTION strikes this balance. Table 6 shows that the savings achieved by CLIFFCOMPACTION come entirely from reducing cache-read costs. Although cached tokens are substantially cheaper than uncached input or output tokens on a per-token basis, coding agents repeatedly reread their entire context throughout a trajectory, so cache reads dominate total cost, accounting for 78% of the uncompacted bill. By limiting context growth, CLIFFCOMPACTION directly targets this component, cutting cache-read cost by 80%, from \$0.427 to \$0.087 per task. Compaction does unavoidably introduce increases in uncached input cost (from re-prefilling the compacted context) and output cost (from the additional turns following compaction), but these overheads remain minor: together they grow by \$0.06 per task, against \$0.34 in cache-read savings. Consequently, the cache-read reduction overwhelmingly outweighs these additional expenses, halving total cost.

Table 7: Real (provider-metered) cost and provider cache-hit rate across scaffolds and context budgets. Subscripts on *Real Cost* give the cost change relative to the Full-context baseline (negative = cheaper, positive = more expensive).

		Full-context		32k		16k		8k	
Scaffold	Model	Real Cost	Cache Hit	Real Cost	Cache Hit	Real Cost	Cache Hit	Real Cost	Cache Hit
SWE-bench Verified									
mini-swe-agent	Kimi K2.6	\$0.19	96%	\$0.19$_{+0\%}$	95%	\$0.17$_{-8\%}$	92%	\$0.19$_{+1\%}$	86%
	GLM 5.1	\$0.38	82%	\$0.29$_{-22\%}$	81%	\$0.19$_{-50\%}$	84%	\$0.16$_{-58\%}$	76%
OpenHands	Kimi K2.6	\$0.57	98%	\$0.41$_{-28\%}$	96%	—	—	—	—
	GLM 5.1	\$1.27	91%	\$0.83$_{-35\%}$	90%	—	—	—	—
Terminal-Bench 2.0									
terminus-2	Kimi K2.6	\$0.40	97%	\$0.30$_{-26\%}$	94%	\$0.19$_{-53\%}$	87%	\$0.25$_{-37\%}$	79%
	GLM 5.1	\$0.89	79%	\$0.53$_{-40\%}$	68%	\$0.40$_{-55\%}$	58%	\$0.25$_{-72\%}$	47%

Complementing the idealized perfect-cache model above, Table 7 reports the actual observed costs together with cache hit rates. Overall, Kimi K2.6 attains higher and more stable cache hit rates (86–98%) than GLM 5.1 (47–91%), and the overall savings trends remain consistent with the perfect-cache model.

The cache hit rate explains how closely the real costs track the idealized estimates. For Kimi K2.6, whose cache hit rates are near-perfect, the real costs match the perfect-cache estimates to within $\sim$1%. For GLM 5.1, the lower cache hit rate makes the real costs higher than the idealized estimates; consequently, the *real* savings from CLIFFCOMPACTION can exceed those predicted by the perfect-cache model. In both cases, cache hit rates decline steadily as the compaction threshold tightens (e.g., from 79% to 47% for GLM 5.1 on Terminal-Bench), reflecting the additional cache invalidation and re-prefilling induced by more frequent compaction.

The largest difference between the two models appears on SWE-bench Verified with `mini-swe-agent`. For GLM 5.1, cost reductions improve steadily as the threshold decreases, from 22% at 32K to 58% at 8K. Kimi K2.6, by contrast, is essentially unchanged at 32K, achieves a modest 8% reduction at 16K, and is marginally more expensive at 8K—the additional re-prefilling at the tightest budget cancels the context savings for this already cost-efficient model. The same

non-monotonic pattern appears on Terminal-Bench, where the 8K setting yields smaller savings (37%) than 16K (53%) for Kimi K2.6.

The magnitude of the savings also varies across scaffolds. On SWE-bench Verified, `OpenHands` benefits substantially more from CLIFFCOMPACTION than `mini-swe-agent`: at a 32K threshold, both models save roughly 28–35% on `OpenHands`, whereas savings on `mini-swe-agent` range from negligible to 22%. The reductions on Terminal-Bench are the most pronounced, reaching as high as 72% and remaining above 25% across all configurations.

Model prices are provided in Table 8.

Table 8: Token prices used for cost calculation. Prices are reported in USD per one million tokens.

Model	Input	Cache read	Output
GLM 4.7 Flash	$0.00	$0.00	$0.00
GLM 5	$1.00	$0.20	$3.20
GLM 5 Turbo	$1.20	$0.24	$4.00
GLM 5.1	$1.40	$0.26	$4.40
GLM 5.2	$1.40	$0.26	$4.40
GLM 5.3 Flash	$0.15	$0.03	$0.50
Kimi K2.5	$0.60	$0.10	$3.00
Kimi K2.6	$0.95	$0.16	$4.00
Kimi K2.7	$0.95	$0.19	$4.00

A.2 MEASURED RUNTIME

We also report runtime as an additional axis for evaluating the efficiency gains of CLIFFCOMPACTION. We use the per-instance wall-clock that `OpenHands` logs directly on SWE-bench Verified, the setting that also sustains the highest, most stable cache-hit rates (above 90%). As shown in Table 9, with the context window capped at 32K, CLIFFCOMPACTION reduces per-instance wall-clock time by 23% and 34% for Kimi K2.6 and GLM 5.1, respectively—closely tracking the corresponding cost reductions of 28% and 35%. This indicates that the benefits of CLIFFCOMPACTION extend beyond monetary cost to end-to-end execution time.

Table 10 shows the run time and speedup of CLIFFCOMPACTION on `mini-swe-agent` with GLM-4.7-Flash served by vLLM (RTX PRO 6000 Blackwell). The same pattern holds: inference time falls monotonically as the threshold tightens, giving speedups of $1.60\times$, $2.10\times$ and $2.62\times$ at 32K, 16K and 8K. This highlights that the benefits of CLIFFCOMPACTION extend to self-hosted serving setups.

Table 9: Per-instance wall-clock duration on `OpenHands` (SWE-bench Verified), with the corresponding cost reduction.

	Duration		Reduction at 32k	
Model	Full-context	32k	Duration	Cost
Kimi K2.6	849 s	656 s	−23%	−28%
GLM 5.1	900 s	592 s	−34%	−35%

Table 10: Measured inference time of CLIFFCOMPACTION at different context thresholds (GLM-4.7-Flash, vLLM on RTX PRO 6000 Blackwell, SWE-bench Verified). Time is the total latency spent in LLM calls. Speedup is relative to the full-context run.

Threshold	Total time (h)	Speedup
Full context	234.2	—
32K	146.0	1.60×
16K	111.6	2.10×
8K	88.9	2.62×

A.3 CLIFFCOMPACTION VS. OTHER COMPACTION METHODS

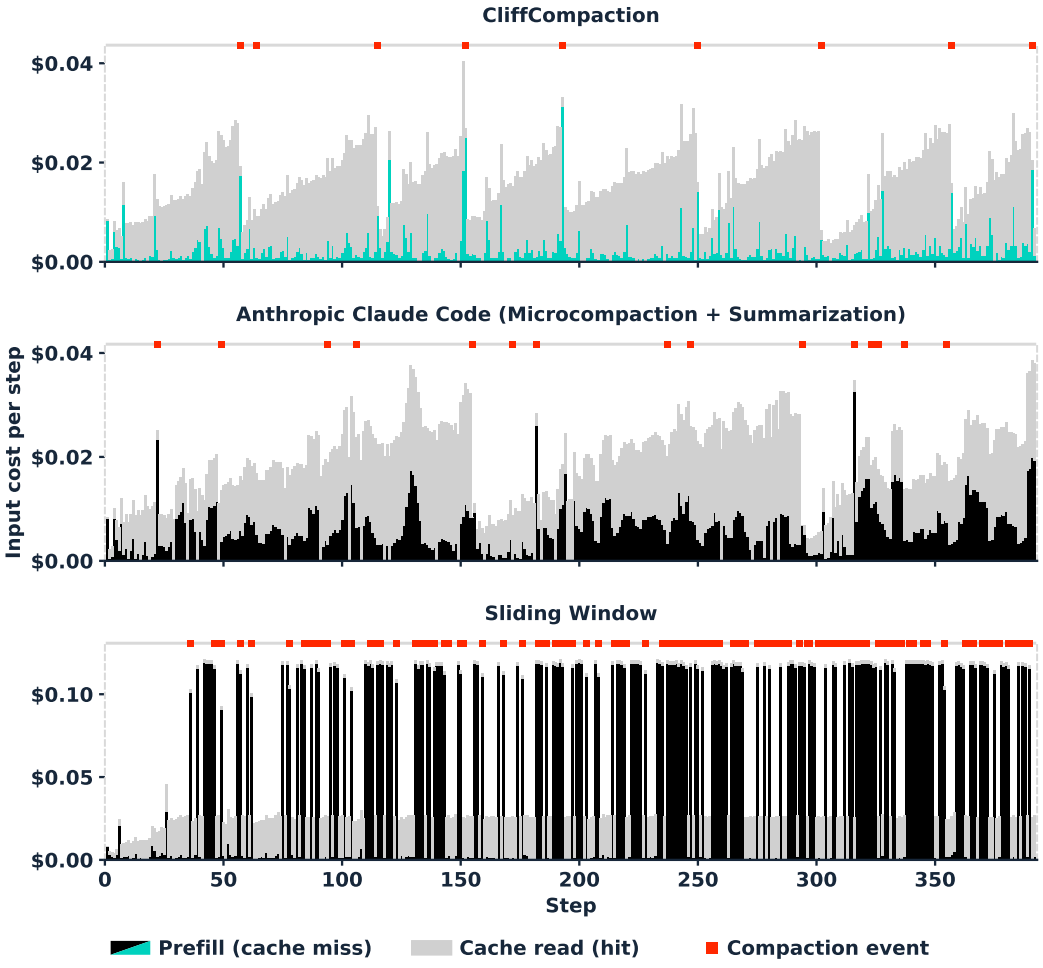


Figure 4: Per-step input cost on a KernelBench task, decomposed into uncached prefill and cache reads.

Figures 4 and 5 trace a single representative KernelBench task step by step. The context trace (Figure 5) is the real-data counterpart of Figure 2. CLIFFCOMPACTION grows its context append-only and drops it in a clean cliff at each compaction. Sliding window holds the context at its ceiling. Microcompaction + summarization never lets the context settle, continually masking and rewriting old observations, producing the sawtooth teeth that ripple through the entire trajectory.

These patterns produce the per-step costs in Figure 4, which follow exactly the cost ordering anticipated in Figure 2: cheap for CLIFFCOMPACTION, intermediate for microcompaction + summarization, and expensive for sliding window. Because CLIFFCOMPACTION leaves earlier tokens untouched between compactions, its context is served almost entirely from cache (gray), and it pays an uncached re-prefill (colored) only at its rare resets. Microcompaction's constant prefix edits invalidate the KV-cache again and again, making every tooth a fresh partial re-prefill, so the cache never has a chance to amortize and the per-step cost cannot fall far. Sliding window is the worst, shifting the prefix at nearly every step and re-prefilling close to the full context each time. CLIFFCOMPACTION thus remains the most efficient compaction method.

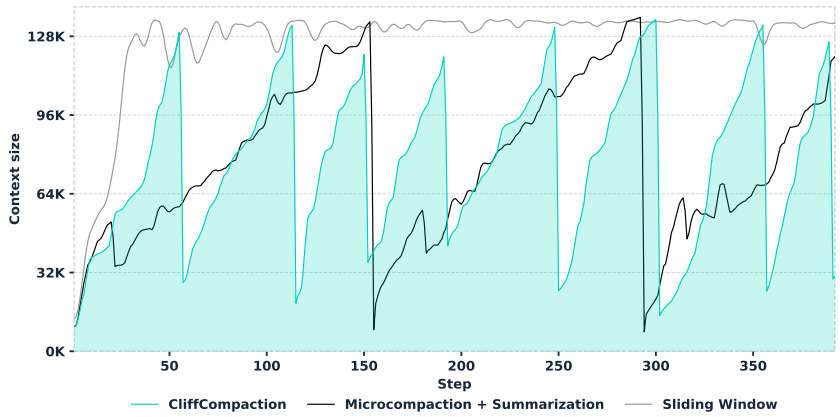


Figure 5: Context size over steps on a KernelBench task.

B MAKING TEST-TIME SCALING AFFORDABLE WITH CLIFFCOMPACTION

Test-time scaling improves agent performance by performing multiple rollouts for a task and then using an algorithm, model, or heuristic to choose the best solution, but it has remained mostly academic due to its high cost. To put the cost-performance trade-off into perspective: without CLIFFCOMPACTION three rollouts of Kimi K2.6 on all Terminal-Bench instances buy 4.8 points of performance gain over a single run at three times its cost ($91.65)—a price that exceeds a single run of a proprietary model that scores 5.5 points higher at a cost of $65, which shows the impractical nature of naive test-time scaling.

CLIFFCOMPACTION inverts this trade-off. Compaction reduces per-rollout cost enough that multiple rollouts can fit within the budget of a single frontier-model run. With CLIFFCOMPACTION, three Kimi K2.6 rollouts cost $58.01 for the entirety of Terminal-Bench, less than one run of GPT 5.3 Codex, while matching the strongest proprietary model, Anthropic's Opus 4.7 (see Table 11).

B.1 EXPERIMENTAL SETUP

We scale `Terminus-2` on Terminal-Bench (Table 11) and `mini-swe-agent` on SWE-bench Verified (Table 12) k times under each CLIFFCOMPACTION context threshold, for both Kimi K2.6 and GLM 5.1. We report *pass@1*, the mean resolution rate across individual runs (without selection), and *Oracle* (*pass@k*), the fraction of tasks solved by at least one of the k runs, effectively picking the best trajectory among all rollouts, which serves as an upper bound on achievable performance.

Since oracle selection is unavailable at deployment time, we additionally report *Practical*, the resolution rate achieved by a learned selector. For each candidate trajectory T, we extract a feature vector $\phi(T) = [f_1(T), \ldots, f_d(T)] \in \mathbb{R}^d$ that summarizes its execution behavior (e.g., the number of steps and tool calls) and characteristics of the produced solution (e.g., its length and structure). We further find line overlap to be a strong statistical feature that substantially boosts accuracy for trajectory selection. This aligns with Shen et al. (2026), who show that the line overlap of two rollouts correlates strongly with unit-test resolution. Following this, we generalize line overlap to a broader set of within-group agreement features that capture how much a candidate agrees with the others for the same task. We call the resulting selector SOFT GROUP VERIFICATION (SGV).

A lightweight LightGBM classifier $s(\cdot)$ takes these features as inputs and assigns each candidate a score $s(\phi(T))$, and we select $\hat{T} = \arg\max_T s(\phi(T))$ as the final solution. The scorer is trained on rollouts from a *different* model: the selector used on Kimi K2.6 is trained only on GLM 5.1's, and vice versa. We find no consistent difference between selectors trained on the same versus a different model. We additional test for test leakage and find SGV to not leak any information through summary statistics of the trajectory. Details are in Appendix E.1.

Table 11: Test-time scaling on Terminal-Bench 2.0, ordered by accuracy. With CLIFFCOMPACTION (abbreviated CLIFF here for space), three rollouts of Kimi K2.6 cost less than a single GPT 5.3 Codex run while surpassing every proprietary baseline evaluated on the same `Terminus-2` harness. *Practical* is the resolution rate of our learned selector, SOFT GROUP VERIFICATION (SGV); at k=1 it is plain pass@1. *Oracle* is the pass@k ceiling. *Context* is the compaction budget for CLIFF rows and the model's full window otherwise. Teal rows are Pareto-optimal in accuracy–cost among our scaled (k>1) configurations; purple rows are proprietary baselines.

Model	Scaffold	Context	k	Cost (USD)	Oracle	Practical	Gain @ Cost*
GPT 5.5 + LLM-as-a-Verifier§	`Capy`	1M	5	—	92.1§	86.5§	+3.4 @ 5.0×
GPT 5.5§	`Capy`	1M	1	—	—	83.1§	—
Kimi K2.6 + CLIFF + SGV	`Terminus-2`	16K	3	$58.01	74.2	69.7	+10.5 @ 1.9×
Opus 4.7‡	—	1M	1	—	—	69.4	—
Kimi K2.6 + CLIFF + SGV	`Terminus-2`	16K	2	$38.67	70.4	65.9	+6.7 @ 1.3×
Kimi K2.6 + CLIFF + SGV	`Terminus-2`	32K	3	$65.33	73.0	65.2	+6.0 @ 2.1×
GPT 5.3 Codex	`Terminus-2`	272K	1	$64.63†	—	64.7	—
Kimi K2.6 + SGV	`Terminus-2`	256K	3	$91.65	70.8	64.0	+4.8 @ 3.0×
Opus 4.6	`Terminus-2`	1M	1	$44.53†	—	62.9	—
GLM 5.1 + CLIFF + SGV	`Terminus-2`	32K	3	$95.34	68.5	60.7	+10.9 @ 1.8×
GLM 5.1 + CLIFF + SGV	`Terminus-2`	16K	3	$87.25	68.5	59.6	+9.8 @ 1.7×
Kimi K2.6	`Terminus-2`	256K	1	$30.55	—	59.2	—
GLM 5.1 + SGV	`Terminus-2`	200K	3	$158.25	65.2	55.1	+5.3 @ 3.0×
GLM 5.1	`Terminus-2`	200K	1	$52.75	—	49.8	—

* Accuracy gain and cost multiplier relative to the same model's full-context single run (k=1).
† Costs from the Terminal-Bench leaderboard (`Terminus-2`): Opus 4.6 uses the reported cost; GPT 5.3 Codex is computed from its token counts.
‡ Score from the Claude Opus 4.7 system card.
§ From Kwok et al. (2026); Gemini-2.5-Flash as the verifier.

B.2 RESULTS

Terminal-Bench 2.0. CLIFFCOMPACTION makes test-time scaling cost-effective where naive scaling is not. At the same three-rollout budget, uncompacted Kimi K2.6 gains 4.8 points over a single run at 3.0× its cost, while CLIFFCOMPACTION at a 16K context limit gains 10.5 points at 1.9× (Table 11). This configuration reaches 69.7%, matching the strongest proprietary model we test (Opus 4.7, 69.4%) and exceeding GPT 5.3 Codex (64.7%) and Opus 4.6 (62.9%), at a total cost of $58.01, less than a single GPT 5.3 Codex run. It also dominates the uncompacted three-rollout baseline: 5.7 points more accurate at 37% lower cost.

The gains persist under a more conservative test-time budget. With only two rollouts, CLIFFCOMPACTION reaches 65.9% at $38.67, a 1.3× cost multiplier, cheaper than a single run of either GPT 5.3 Codex or Opus 4.6. This already exceeds the uncompacted three-rollout baseline at 42% of its cost.

The same pattern holds for GLM 5.1: CLIFFCOMPACTION raises the practical resolution rate from 55.1% to 60.7% while reducing scaled-inference cost by 40% ($95.34 vs. $158.25), narrowing much of the gap to the proprietary models.

To further illustrate the cost-effectiveness of CLIFFCOMPACTION and SGV, we focus on *Gain@Cost* and compare against another selector. Using a strong base model and a separate Gemini-2.5-Flash verifier to rank five rollouts, LLM-as-a-Verifier converts a 5.0× cost increase into +3.4 points—the lowest gain-per-cost in the table. CLIFFCOMPACTION with SGV turns a 1.9× cost increase into +10.5 points, and +6.7 at only 1.3×. GLM 5.1 shows the same pattern (+10.9 at 1.8×).

SWE-bench Verified. On SWE-bench Verified (Table 12), Kimi K2.6 with SGV reaches the highest Practical accuracy in the table (79.4%), topping every proprietary single-model baseline, including the strongest, Opus 4.5 (76.8%). It also exceeds the LLM-as-a-Verifier (78.2%), and comes within 0.2 points of it at k=3 at roughly half the cost ($284.28 vs. $625), without the repeated LLM-verifier calls their method requires.

With CLIFFCOMPACTION, capping context at a lower threshold widens this cost advantage. Three compacted Kimi rollouts at 16K reach 76.4% for $264.06, within 0.4 points of Opus 4.5 at 70% of its

Table 12: Test-time scaling on SWE-bench Verified, ordered by accuracy. All rows use `mini-swe-agent` except CodeMonkeys, which uses its own.

Model	Context	k	Cost (USD)	Oracle	Practical	Gain @ Cost*
Kimi K2.6 + SGV	256K	5	$473.80	83.0	79.4	+4.8 @ 5.0×
LLM-as-a-Verifier§	mixed	3	∼$590.00	84.4	78.2	+2.1 @ 3.2×
Kimi K2.6 + SGV	256K	3	$284.28	80.4	78.0	+3.4 @ 3.0×
Kimi K2.6 + CLIFF + SGV	16K	5	$440.10	80.8	77.0	+2.4 @ 4.64×
Opus 4.5 (high reasoning)	200K	1	$375.00	—	76.8	—
GLM 5.1 + SGV	200K	5	$604.65	80.6	76.6	+5.8 @ 5.0×
Kimi K2.6 + CLIFF + SGV	16K	3	$264.06	79.4	76.4	+1.8 @ 2.79×
Kimi K2.6 + CLIFF + SGV	32K	5	$459.60	82.8	76.2	+1.6 @ 4.85×
Kimi K2.6 + CLIFF + SGV	32K	3	$275.76	79.8	76.0	+1.4 @ 2.91×
GLM 5.1 + SGV	200K	3	$362.79	78.0	75.8	+5.0 @ 3.0×
Gemini 3 Flash (high reasoning)	1M	1	$180.00	—	75.8	—
MiniMax M2.5 (high reasoning)	205K	1	$35.00	—	75.8	—
Opus 4.6	1M	1	$275.00	—	75.6	—
GLM 5.1 + CLIFF + SGV	32K	5	$519.85	79.2	74.8	+4.0 @ 4.30×
Kimi K2.6	256K	1	$94.76	—	74.6	—
GLM 5.1 + CLIFF + SGV	16K	3	$229.89	78.6	74.0	+3.2 @ 1.90×
GLM 5.1 + CLIFF + SGV	32K	3	$311.91	76.2	74.0	+3.2 @ 2.58×
GLM 5.1 + CLIFF + SGV	16K	5	$383.15	81.4	73.8	+3.0 @ 3.17×
GLM 5.1	200K	1	$120.93	—	70.8	—
CodeMonkeys (Sonnet 3.5 + Qwen 2.5)¶	200K	10	$2291.90	69.8	57.4	—
Sonnet 3.7	200K	1	$175.00	—	52.8	—

* Accuracy gain and cost multiplier relative to a single uncompacted rollout (k=1).
§ From Kwok et al. (2026); Gemini-2.5-Flash as the verifier. Three candidates, each generated by one of Claude Opus 4.5, Gemini 3 Flash, and MiniMax M2.5. Cost: generation $375+$180+$35 ≈ $590.
¶ From Ehrlich et al. (2025).

cost, and five reach 77.0%, surpassing it. On cost, GLM 5.1 at 16K is the cheapest scaled configuration overall (74.0% at $229.89). As a result, the cheapest Pareto-optimal scaled configurations are compacted ones.

C Extended CliffCompaction Design and Implementation

CliffCompaction keeps literal fragments of the original context, truncating them rather than rewriting or paraphrasing their content. For each action, it keeps only a lightweight signature: the first 150 characters of the action string. In `mini-swe-agent` and `Terminus-2`—whose actions are primarily bash commands and terminal keystrokes, respectively—this signature is the leading 150 characters of each command (the bash command parsed from the model's `bash` code block in `mini-swe-agent-v1`, and the `keystrokes` parsed from the model's response in `Terminus-2`). This simple truncation captures most of the useful information, since the command itself typically appears at the start of the string, while the remainder often consists of lengthy inline file contents. For `OpenHands`, the tool-call signatures are listed in Table 13.

D Extended Evaluation and Analysis

D.1 CliffCompaction has high precision

We justify the precision hypothesis behind CliffCompaction through the agent's re-reading behaviour. Figure 6 shows that every compaction strategy increases the number of steps per trajectory relative to the full-context runs, with sliding window, summarization + microcompaction and CliffCompaction each adding more than four steps, while summarization rises by only $+1.26$.

Almost all of these additional steps are spent re-reading files the agent had already opened. Editing and testing, by contrast, decrease under every strategy. CliffCompaction, sliding window and microcompaction discard tool outputs outright, leaving a visible gap in the context, so the agent

Table 13: Tool-call signatures used by CLIFFCOMPACTION in `OpenHands` scaffold.

Tool	Sub-command	Retained signature
`terminal`	—	`[terminal] {command}`
`file_editor`	`view`	`[file_editor view] {path} range={range}`
`file_editor`	`create`	`[file_editor create] {path} ({N} chars)`
`file_editor`	`str_replace`	`[file_editor str_replace] {path} (old: {old_str}...)`
`file_editor`	`insert`	`[file_editor insert] {path}:{line}`
`file_editor`	`undo_edit`	`[file_editor undo_edit] {path}`
`think`	—	`[think] {thought}`
`finish`	—	`[finish]`
`task_tracker`	—	(dropped)
Caps: `command` $\leq$ 120, `old_str` $\leq$ 60 chars.		

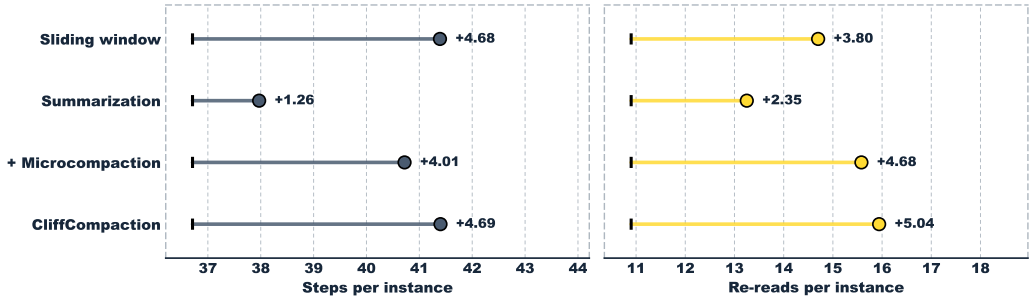


Figure 6: Δ of mean actions per instance and steps spent on re-reads (SWE-bench Verified, Kimi K2.7, $16K$) relative to running without compaction (full-context: 36.71 steps, 10.90 re-reads per instance).

re-fetches the content when it needs it again ($+5.04$, $+3.80$ and $+4.68$ re-reads respectively). Summarization adds the fewest re-reads ($+2.35$) and is the only strategy that also opens fewer new files (-0.68 per instance). It replaces the discarded content with a plausible natural-language paraphrase, which appears to satisfy the agent and discourages it from returning to the ground-truth source.

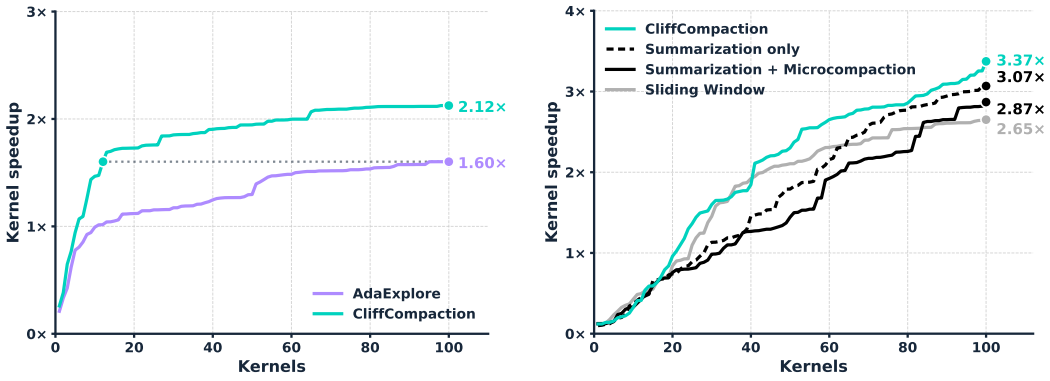
This behaviour corroborates the low-recall, high-precision trade-off we posit for CLIFFCOMPACTION: it retains less than summarization, but what it retains is verbatim, and the gaps it leaves are unambiguous, so the agent restores what it needs from the source.

D.2 SPEEDUP BY KERNELS GENERATED ON KERNELBENCH

Search-based methods such as AdaExplore budget by candidate kernels: each of their steps is one kernel proposal. We therefore also report speedup indexed by kernels generated (Figure 7), measuring which method finds the faster kernel given the same number of candidates to compile and benchmark. Because our setup is agentic, we count a kernel as an agent turn that writes compiled kernel code.

Figure 7a shows that once exploration steps are excluded, CLIFFCOMPACTION leads AdaExplore at every budget, with a clear margin by the tenth kernel ($1.47\times$ against $0.99\times$). It reaches $1.60\times$ after 13 kernels, a level AdaExplore takes 100 kernels to reach. This again shows that a simple agentic setup with autocompaction compares favorably with a pipeline built specifically for kernel optimization.

Figure 7b shows the same advantage over the other compaction baselines. Within the same step budget (Table 5), CLIFFCOMPACTION generates the fewest candidates, so normalizing by proposals



(a) CLIFFCOMPACTION vs. AdaExplore. (b) CLIFFCOMPACTION vs. compaction alternatives.

Figure 7: Best-so-far kernel speedup on KernelBench Level 3, indexed by kernels generated, AdaExplore's budget unit.

widens the gap. Summarization spends fewer steps re-reading (91.8 reads per problem vs. 102.7) and more steps writing kernels, so its proposals are individually less effective. CLIFFCOMPACTION, in contrast, re-reads from the workspace and converts each proposal into more speedup, the same pattern appears on SWE-bench (Appendix D.1).

E EXPERIMENTAL DETAILS

E.1 ADDITIONAL TEST-TIME SCALING DETAILS

In this subsection, we describe the selector used to obtain the *Practical* results in Table 11 and Table 12.

Each candidate is represented by a 30-dimensional feature vector (Table 14 and Table 15). We train a LightGBM binary classifier: given the feature vector of a candidate trajectory, it predicts the probability that the candidate resolves the task, and a separate classifier is trained for each rollout budget k. We use the following hyperparameters on both benchmarks: $\{$800 boosting rounds, learning rate 0.03, 63 leaves, minimum 15 samples per leaf, ℓ_2 regularization 1.0$\}$. We train the selector on rollouts from a different model than the one it is applied to: the selector used on Kimi K2.6 rollouts is trained only on GLM 5.1 rollouts, and vice versa. Training on rollouts from the same model gives no consistent advantage (Table 16).

We also examine the task-generalization ability of the selector. We train it on a fraction of the tasks and evaluate on all 89 Terminal-Bench 2.0 tasks (Table 17). Across training fractions from 25% to 100%, the resolution rate remains essentially unchanged for every configuration. This shows that the selector does not overfit to the specific training tasks, it generalizes across tasks and performs comparably even when trained on only a quarter of them. In the main results (Table 11) we use the selector trained on 100% of the tasks.

E.2 ADDITIONAL KERNELBENCH DETAILS

Agent setup. For the CUDA setting, we follow the KernelBench setup of Dai et al. (2026), using the same skill and working environment. Each task is a self-contained CUDA C++ extension project in which the agent writes raw CUDA kernels and is barred from falling back on PyTorch compute operators, so reported speedups reflect genuinely hand-written kernels; we refer to Dai et al. (2026) for the full skill and agent loop. The agent is instructed to optimize the reference model for maximum speedup over the PyTorch Eager and `torch.compile` baselines and to keep proposing new optimizations. We differ from Dai et al. (2026) only in: (i) the backbone model (Kimi K2.7 and Kimi K2.6 vs. their RL-trained model); (ii) the hardware (L40S and RTX Pro 6000 Blackwell vs. their H20); (iii) the stopping condition—we remove the agent's ability to self-terminate and run to

Table 14: The feature set used by SGV on Terminal-Bench 2.0. $\Delta\mu$ denotes the feature minus its within-group mean; $/\max$ denotes the feature divided by its within-group maximum. These instance-normalized variants make a candidate's value relative to the others of the same task.

Feature	Description
Execution behavior	
`marked_complete`	Agent explicitly signaled task completion
`n_episodes`	Number of agent steps
`n_episodes` $\Delta\mu$	relative to group mean
`n_tool_calls`	Number of tool calls
`n_tool_calls` $\Delta\mu$	relative to group mean
`n_tool_calls` $/\max$	relative to group max
Produced solution	
`n_files`	Number of files written
`n_lines`	Non-trivial lines of code
`n_raw_lines`	Raw lines (incl. blanks/short)
`n_paths_root`	Files written under an absolute path
`n_unique_idents`	Distinct identifiers
`is_shell`	Solution is a shell script
`avg_line_len`	Mean line length
`avg_line_len` $\Delta\mu$	relative to group mean
`max_line_len`	Maximum line length
`std_line_len`	Std. dev. of line length
`max_indent` $\Delta\mu$	Max indentation depth, rel. to group mean
`n_indent_levels` $\Delta\mu$	Distinct indent levels, rel. to group mean
`alnum_ratio`	Alphanumeric character ratio
`alnum_ratio` $\Delta\mu$	relative to group mean
`punct_density`	Punctuation character density
`kw_if`	Count of `if`/`elif`
`kw_if` $\Delta\mu$	relative to group mean
`kw_while`	Count of `while`
`kw_print` $/\max$	Count of `print`/`echo`, rel. to group max
`kw_shebang`	Count of shebang (`#!`) lines
Within-group agreement	
`overlap_min`	Min line overlap with other candidates
`overlap_sum` $\Delta\mu$	Mean line overlap, rel. to group mean
`symbol_overlap_sum`	Shared identifiers with other candidates
`symbol_jaccard_sum` $/\max$	Symbol Jaccard agreement, relative to group max

Table 15: The feature set used by SGV on SWE-bench Verified, for the selector applied to Kimi K2.6 rollouts (trained on GLM 5.1). $\Delta\mu$ denotes the feature minus its within-group mean; $/\max$ denotes the feature divided by its within-group maximum.

Feature	Description
Produced patch	
`n_lines`	Non-trivial changed lines
`n_add`	Added lines
`n_del`	Deleted lines
`n_del` $\Delta\mu$	relative to group mean
`n_del` $/\max$	relative to group max
`add_del_ratio`	Ratio of added to deleted lines
`n_files`	Files modified
`n_hunks`	Diff hunks
`patch_func_count`	Distinct function names in `def` statements appearing in the patch
`patch_func_count` $/\max$	relative to group max
`n_unique_idents`	Distinct identifiers
`n_total_tokens`	Total identifier tokens
`ident_repeat_ratio`	Identifier tokens per distinct identifier
`avg_line_len`	Mean line length
`avg_line_len` $/\max$	relative to group max
`max_line_len`	Maximum line length
`sig_if`	Lines matching an `if` signature
`sig_return`	Lines matching a `return` signature
`kw_if`	Count of `if`/`elif`
`kw_import`	Count of `import`
Repository context	
`repo_code`	Repository identity (categorical)
Within-group agreement	
`group_consensus_level`	Mean pairwise line Jaccard across the task's candidates
`group_diversity`	`1−group_consensus_level`
`group_has_exact_duplicates`	An identical patch occurs at least twice in the group
`file_agreement_sum`	Other candidates modifying the same files
`symbol_overlap_sum`	Changed identifiers shared with other candidates
`add_overlap_sum`	Added lines shared with other candidates
`del_overlap_sum`	Deleted lines shared with other candidates
`hunk_jaccard_sum`	Jaccard agreement of diff hunks with other candidates
`max_symbol_jaccard`	Symbol Jaccard with the most similar other candidate

Table 16: Same-model versus cross-model training of SGV on Terminal-Bench (Kimi K2.6, k=3). We used 5-fold instance-level cross-validation. Δ is same-model minus cross-model; neither shows a consistent advantage.

Context	pass@1	Oracle	Same-model	Cross-model	Δ
256K	59.2	70.8	61.8	60.7	+1.1
16K	61.4	74.2	63.7	64.4	−0.7

Table 17: Data efficiency of the selector on Terminal-Bench 2.0 (Kimi K2.6, GLM→Kimi). The selector is trained on a random fraction of tasks and evaluated on all 89. Resolution rate (%) at k=3 with the fixed 30-feature set.

Training fraction	256K	32K	16K
25%	64.0	65.2	68.5
50%	66.3	64.0	68.5
75%	65.2	66.3	68.5
100%	64.0	65.2	69.7

a fixed budget of steps (or, without compaction, until the 256K context window is exceeded); and (iv) context management via CLIFFCOMPACTION.

For the Triton setting, we run GPT-5-mini in the same environment but target Triton rather than CUDA and do not block PyTorch operators, matching Du et al. (2026).

Evaluation setup. To score the speedups achieved by the agent, we follow Du et al. (2026). At each step, we evaluate the kernel the agent produces: we compile it, check that its output matches that of the reference model on 5 random inputs within a tolerance of $\texttt{atol} = \texttt{rtol} = 10^{-2}$ for our CUDA configurations and $\texttt{atol} = \texttt{rtol} = 5 \times 10^{-2}$ for our Triton configuration, and if it is correct, measure its runtime against the PyTorch Eager baseline (averaged over 10 timed runs after 10 warmup runs). For each problem we record the best speedup the agent achieves across all steps, clamped to a maximum of $10\times$. If the agent fails to produce any correct kernel within the step budget, that problem is penalized with a speedup of 0.1. The reported *Speedup* is the geometric mean of these per-problem speedups over all 50 tasks.