RACER: RECOVERABLE KV CACHE MANAGEMENT FOR MULTI-TURN LLM AGENTS

Anonymous authors
Paper under double-blind review

ABSTRACT

Efficient context management is critical for multi-turn large language model (LLM) agents, whose tool definitions and growing interaction histories increase key-value (KV) cache usage. Recent work has explored text-level context compression and KV cache compression to reduce this cost. However, compression may lead to a decision that differs from full-context execution, which in multi-turn agents can change the environment and affect subsequent decisions through the resulting observation. We refer to this phenomenon as compression-induced trajectory drift. To address this problem, we propose Racer, a recoverable KV cache management framework compatible with different KV compression methods. For tool definitions, Racer retains query-relevant tool definitions with full KV representations while compressing the remaining definitions. For interaction histories, Racer protects selected historical information as native KV before generation, then uses a risk detector to identify potentially incorrect drafts and selectively restores historical evidence, allowing the model to regenerate its decision before execution. Extensive evaluations demonstrate that Racer improves mean task success by 43.7% across six KV-compression backends on long-context tasks, while achieving 1.67× higher task throughput than full-context execution under concurrent serving.

1 INTRODUCTION

Multi-turn large language model (LLM) agents interact with users and environments over multiple steps, making decisions based on previous actions and observations (Yao et al., 2022b; Yang et al., 2024). Their context includes tool definitions that describe available capabilities and interaction histories that record prior actions, observations, and results (Yao et al., 2022a; Wu et al., 2024; Maharana et al., 2024). As interactions continue, actions and observations accumulate in the history, increasing KV cache usage and attention overhead while reducing serving concurrency (Kwon et al., 2023; Liu et al., 2024; Xiao et al., 2025). KV cache compression can reduce this growing cost, but compressed history still expands as interactions continue (Hooper et al., 2024). Long-running agents therefore require higher compression ratios to operate under limited KV-memory budgets (Tang et al., 2025). This paper addresses how to support high-ratio KV compression for agent context while preventing local compression errors from accumulating and propagating across subsequent turns.

To reduce the cost of growing agent context, existing approaches can be broadly categorized into two classes. Text-level compression methods summarize or restructure previous trajectories (Hu et al., 2025; Kang et al., 2025; Ye et al., 2025a), while KV-level methods operate directly on the cache and retain only the most important KV states (Zhang et al., 2023; Xiao et al., 2024; Li et al., 2024; Cai et al., 2024). While improving inference efficiency, higher compression ratios inevitably reduce the information available to the model, increasing the risk of decision errors relative to full-context execution. Recent agent-aware KV methods further model multi-turn relevance to adapt which KV states are kept available during execution (Liu et al., 2026a; Huang et al., 2026; Liu et al., 2026b; Rusli et al., 2026). Nevertheless, these methods primarily focus on maintaining compressed context rather than preventing compression errors from propagating across subsequent decisions.

However, this limitation becomes critical in closed-loop agent execution. Standard KV compression methods are often evaluated on single-turn tasks such as question answering, where compression

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.

errors primarily affect the current output. In multi-turn agents, a compression-induced error can change the predicted action and alter the environment. The resulting observation is written back into the interaction history and used by subsequent decisions. Later actions are conditioned on an already deviated trajectory, allowing the initial error to accumulate across turns. We refer to this phenomenon as *compression-induced trajectory drift*. This suggests that simply managing or restoring KV states is insufficient once an incorrect action has already affected the environment. Our key intuition is to assess draft-failure risk under compressed context and selectively restore historical evidence, allowing the model to regenerate its decision before commitment to the environment.

To address these problems, we propose Racer (**R**isk-**A**ware **C**ompression **E**rror **R**ecovery), a recoverable KV cache management framework for multi-turn LLM agents compatible with different KV compression methods. Racer addresses two major sources of context overhead: tool definitions and interaction histories. For tool definitions, Racer preserves the full KV representations of selected query-relevant definitions while compressing the rest. For interaction histories, Racer protects selected historical information with full KV representations before generation, then selectively restores relevant evidence based on the draft's failure risk, enabling the model to regenerate its decision before execution under the same history-KV budget.

We evaluate Racer against compression-only execution across different KV-compression backends under matched history-KV budgets, and separately assess selective tool-definition compression against uniform compression. Experimental results demonstrate improved tool-name accuracy and consistent task-success gains across multiple KV-compression backends, alongside higher throughput under concurrent serving.

Our main contributions are summarized as follows:

- We identify compression-induced trajectory drift as a key challenge of KV cache compression in multi-turn agents, where local errors can alter decisions and propagate across subsequent turns.
- We propose Racer, a recoverable KV cache management framework compatible with different KV compression methods, supporting selective tool-definition compression and pre-action history recovery.
- We design a controller that combines pre-generation history protection with risk-triggered recovery, retaining selected historical information and selectively restoring decision-relevant evidence based on draft-failure risk before execution under a fixed history-KV budget.
- Extensive experiments demonstrate that Racer improves mean task success by 43.7% across six KV-compression backends on long-context tasks, while achieving 1.67× higher task throughput than full-context execution under concurrent serving.

2 Background and Problem Formulation

2.1 KV Cache Compression for Agent Context

In a multi-turn agent, each LLM call takes as input the system prompt, tool definitions, interaction history, and current-turn user query. Since the tool definitions and interaction history persist across decisions and are repeatedly reused, compressing their KV states can reduce the context maintained during execution. We denote the context at decision step t as

$$C_t = (T, H_t, U_t), \tag{1}$$

where T denotes the tool definitions, H_t denotes the completed interaction history, and U_t denotes the remaining uncompressed context, including system instructions and the current turn. We consider KV compression for two components:

$$\bar{T} = \mathcal{C}(T), \qquad \bar{H}_t = \mathcal{C}(H_t), \tag{2}$$

where $\mathcal{C}(\cdot)$ denotes a KV cache compression operator.

2.2 Compression-Induced Trajectory Drift

We examine the effect of history-KV compression on the multi-turn Base benchmark of BFCL (Patil et al., 2025). With approximately 2.4× history-KV compression, representative KV compression

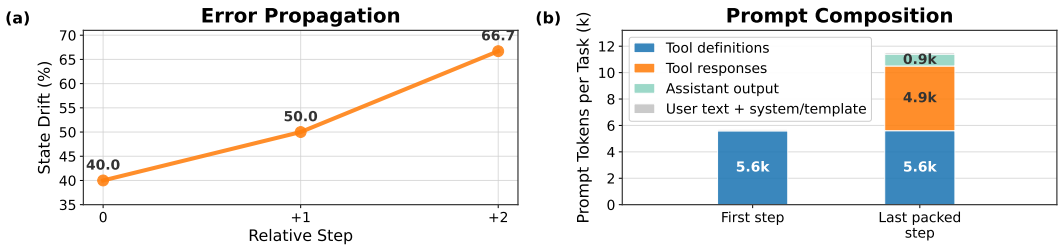


Figure 1: **Compression error propagation and prompt composition.** (a) After the first compression-induced divergent action, state drift increases from 40.0% to 66.7% over the next two steps. (b) Tool definitions form a static context floor, while tool responses and assistant outputs account for most of the growth in later steps.

methods reduce task success from 39.0% under full-context execution to 15.0–25.0%. We further investigate whether compression errors remain isolated to individual decisions or persist across subsequent steps of closed-loop execution. We align trajectories at the first compression-induced divergent action and define *state drift* as the fraction whose environment state differs from the corresponding full-context trajectory. Figure 1(a) shows that state divergence persists beyond the first divergent action, illustrating how compression errors can affect subsequent turns.

This behavior follows from the closed-loop nature of multi-turn agents. At decision step t, the agent produces an action a_t, receives an observation o_t, and appends both to the interaction history:

$$H_{t+1} = H_t \| (a_t, o_t). \tag{3}$$

A compression-induced action error may change the environment and produce a different observation, which enters the interaction history and influences subsequent decisions. We refer to this process as *compression-induced trajectory drift*. Since restoring context cannot undo an action that has already changed the environment, recovery should occur before the affected decision is committed to the environment.

2.3 Problem Formulation

At decision step t, the model generates an uncommitted draft from a compressed context $\bar{C}_t$. Before the resulting action is executed, a recovery policy η may selectively restore information from the original context, producing a recovered context C'_t for regeneration. Both contexts must satisfy a history-KV budget B_{hist}, accounting for retained historical KV and restored evidence. Given a trajectory ξ and its task-completion score $R(\xi)$, we formulate the objective as

$$\max_{\eta} \ \mathbb{E}_{\xi \sim p_\eta}[R(\xi)] \quad \text{s.t.} \quad M_{\text{hist}}(C) \le B_{\text{hist}}, \quad \forall t,\ C \in \{\bar{C}_t, C'_t\}, \tag{4}$$

where $M_{\text{hist}}(C)$ counts the active historical KV positions, including recovered evidence, in token-equivalent units. The constraint applies during both draft generation and regeneration whenever recovery is admitted. Our goal is to improve task accuracy while limiting compression-induced error propagation under a constrained history-KV budget.

3 Racer Framework

Racer manages tool definitions and interaction histories through compressed KV representations and selective access to archived source text (Section 3.1). It combines selective pre-generation allocation (Section 3.2) with risk-triggered history recovery (Section 3.3), guided by a learned detector (Section 3.4), to revise potentially incorrect decisions before execution. Figure 2 illustrates the overall framework.

3.1 Recoverable Context

At decision step t, Racer maintains an original-text archive $\mathcal{A}_t$, compressed historical KV states Z_t, and a native-KV workspace $K(W_t)$. The archive stores tool definitions and completed interaction events with their source identifiers and temporal order outside the active KV cache. The workspace contains selected original historical content, prefilled through the base model to obtain native KV.

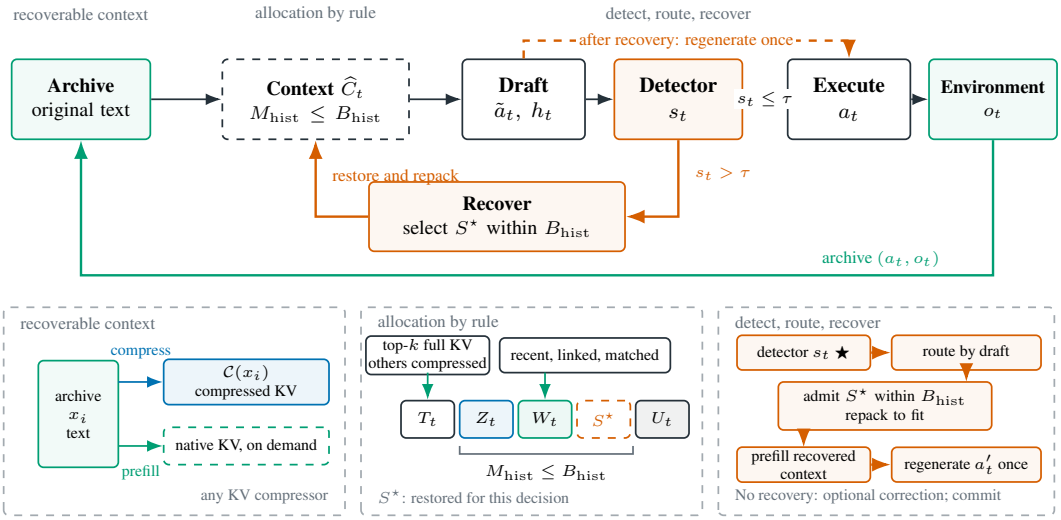


Figure 2: **Overview of Racer.** Top: the model generates an uncommitted draft from compressed context. A risk detector triggers evidence recovery when needed, allowing the model to regenerate once before execution. Otherwise, optional source-backed correction may be applied. The executed action and observation are archived. Bottom: recoverable context (left), pre-generation allocation (center), and risk-triggered recovery (right). ★ denotes the learned risk detector.

Let T_t denote the assembled tool context, containing full native KV representations of the top-K relevant tool definitions and compressed KV representations of the remaining definitions. Together with the uncompressed live input U_t, the active context is

$$\widehat{C}_t = \text{Compose}(T_t, Z_t, K(W_t), U_t)\,, \quad M_{\text{hist}}(\widehat{C}_t) \le B_{\text{hist}}. \tag{5}$$

Here U_t contains the system instructions and live input suffix, while M_{hist} measures active historical KV positions, excluding tool definitions and live input.

3.2 Pre-generation Allocation

Tool definitions. To identify which tool definitions require full fidelity, a fixed lexical ranker selects up to $K = 3$ definitions from the catalog $T = \{d_1, \ldots, d_N\}$ using the selection signal q_t:

$$S_t = \text{TopK}_{d_i \in T}\ s_{\text{lex}}(q_t, d_i). \tag{6}$$

The latest user request supplies the selection signal q_t. Selected definitions retain their complete native KV representations, while all remaining definitions are compressed. Each tool definition contributes one representation to the active tool context. The tool allocation stays fixed across draft generation and any recovery or regeneration within the same decision, and is refreshed at the next decision. Let f denote the fraction of tool-definition tokens retained at full fidelity and r the compression ratio applied to the remaining definitions. The approximate tool-KV compression ratio is

$$R_{\text{tool}} \approx \frac{1}{f + (1-f)/r}. \tag{7}$$

Equation 7 approximates catalog-level compression; measured tool-KV compression also depends on serialization overhead. The fraction of full-KV definitions determines the trade-off between retained information and KV memory.

Interaction history. For interaction histories, a pre-generation allocation rule protects selected recent and request-relevant sources as native KV, while the compression backend represents the remaining history. The policy first attempts event-level allocation and applies the available capacity adjustments. If complete events still cannot fit, it falls back to finer-grained source-message allocation. Source-backed cues about observed values and failed operations are admitted alongside retained history within B_{hist}; the live input is accounted for separately.

3.3 PRE-ACTION RECOVERY

The model first generates an uncommitted draft $\tilde{a}_t$ from $\widehat{C}_t$. Before execution, a draft-failure risk detector determines whether to trigger historical evidence recovery. The proposed tool and arguments guide retrieval of relevant archived evidence. When recovery is triggered and feasible, Racer revises the history within B_{hist}, allowing the model to regenerate the pending action once.

3.3.1 RISK DETECTION

The detector combines a low-dimensional PCA projection $P(h_t)$ of the final prompt-token hidden state with the draft's average negative log-likelihood ℓ_t, turn-stop indicator e_t^{stop}, and valid-action-parsing indicator e_t^{parse}:

$$z_t = \left[P(h_t); \ell_t; e_t^{\text{stop}}; e_t^{\text{parse}}\right]. \tag{8}$$

After standardization with fixed training statistics, a linear detector produces

$$s_t = \sigma\left(w^\top \widehat{z}_t + b_{\text{det}}\right). \tag{9}$$

Recovery is triggered when $s_t > \tau$.

3.3.2 EVIDENCE RETRIEVAL

The held draft supplies the proposed tool and arguments for evidence retrieval. Racer ranks complete eligible events by lexical relevance to the current request, held draft, and latest complete observation. A tool event contains its call and all observations. Events already available as native text or cancelled by a request revision are excluded. The first ranked event that yields a feasible, changed context is selected as the recovered evidence $S^\star$.

3.3.3 BUDGETED RECOVERY AND COMMIT

Given the selected evidence $S^\star$, Racer repacks the history under the same history-KV budget:

$$\widehat{C}'_t = \text{Repack}(\widehat{C}_t, S^\star, B_{\text{hist}}), \qquad M_{\text{hist}}(\widehat{C}'_t) \le B_{\text{hist}}. \tag{10}$$

Restored evidence shares B_{hist} with the remaining history, which is represented by the compression backend within the residual capacity. The model prepares the revised KV context and regenerates once.

When event recovery is not performed, fixed source rules may correct eligible string arguments using exact request literals or unambiguous entries from previous observations. Corrections are committed after their source bindings are verified, without an additional model call.

Only the final action is committed to the environment. The resulting interaction enters the archive, and the next decision applies normal allocation again. Appendix C details the runtime and evidence admission.

3.4 TRAINING AND POLICY CONFIGURATION

Racer is compatible with different KV-compression backends, which are trained or configured independently. Within the recovery policy, only the risk detector is learned; allocation, retrieval, evidence admission, and source verification follow fixed rules.

We collect detector examples at decisions whose preceding user-turn prefixes pass the benchmark checker. From each compressed-context draft $\tilde{a}_t$, we run a no-recovery continuation and label the current user turn:

$$y_t = \mathbb{1}\left[\text{the current user turn fails following } \tilde{a}_t\right]. \tag{11}$$

The detector predicts this outcome before recovery. We fit PCA and normalization on the training split, then train the linear classifier with regularized binary cross-entropy while keeping the model and compressor frozen. Regularization is selected by grouped cross-validation; the detector and threshold are then fixed. Appendix E gives the data, label construction, and training configurations.

3.5 Efficient Execution

The concurrent-serving implementation combines KV reuse with asynchronous history compression. Before generation, the runtime obtains the compressed blocks required by the selected context and reuses matching KV for the initial native prefix preceding the first compressed block. While the model generates, a background worker compresses additional eligible history chunks and caches the resulting blocks for subsequent decisions. Incremental context preparation reuses tokenized history, and persistent runtime workers submit concurrent sessions to a shared SGLang engine. Appendix C.4 describes the cache organization and scheduling; Section 4.3 evaluates the resulting system under concurrent agent workloads.

4 Experiments

4.1 Experimental Setup

Benchmarks and model. We evaluate closed-loop agents on BFCL Base tasks, BFCL Long Context tasks (Patil et al., 2025), and ToolSandbox scenarios (Lu et al., 2025) using their official evaluators, and tool-definition compression on recorded decisions from Agent-LLM-Traces (Bandel et al., 2026). The budget sweep and component study use BFCL Base. The serving study uses a fixed, stratified subset of 64 BFCL Long Context tasks for the concurrency sweep, and all 200 tasks grouped by context length for a separate sweep at fixed concurrency. Each condition has 3 batches. All methods use Qwen3-4B-Instruct-2507 (Yang et al., 2025); closed-loop experiments run through SGLang (Zheng et al., 2024). Appendices B and F provide runtime settings and additional results.

Baselines and budgets. We compare CommitKV (Huang et al., 2026), StreamingLLM (Xiao et al., 2024), H2O (Zhang et al., 2023), SnapKV (Li et al., 2024), PyramidKV (Cai et al., 2024), and C^2KV (Du et al., 2026) with and without Racer. Table 1 uses a history budget of $B_{\text{hist}} = 256$ tokens, including restored evidence. System instructions, tool definitions, and live inputs remain uncompressed. The tool-definition study separately evaluates C2KV, CacheBlend + SnapKV (Yao et al., 2025), and EPIC + SnapKV (Hu et al., 2024), each with and without selective allocation, alongside an uncompressed reference.

We also compare these KV methods with budget-constrained adaptations of HiAgent (Hu et al., 2025) and ACON (Kang et al., 2025) at $B_{\text{hist}} = 256$ on BFCL Base, BFCL Long Context, and ToolSandbox. Appendix B.4 describes their compression and retrieval settings.

Metrics. We report task success on BFCL and mean official scenario similarity on ToolSandbox. Serving efficiency is measured by completed-task throughput, per-task latency, and in-use KV (Section 4.3) (Agrawal et al., 2024; Zhong et al., 2024). The budget sweep additionally reports peak active model KV and effective history compression. The tool study reports Token-F1, ROUGE-L, tool-name accuracy, and mean per-decision tool-KV compression. Appendix D defines the measurements.

4.2 Closed-loop Performance across Compression Methods

We evaluate whether Racer improves closed-loop performance across KV-compression backends under a fixed history-KV budget, with HiAgent and ACON as text-level baselines. Only interaction histories are compressed; tool definitions and live inputs remain uncompressed. All methods use a fixed 256-token history budget, measured in active KV positions for KV-level methods and rendered tokens for text-level methods. Paired evaluations use the same tasks, model checkpoint, decoding settings, and GPU model. The detector and its calibration are transferred unchanged from the learned-compression setting across backends.

As shown in Table 1, Racer consistently improves task success across all six KV-compression backends on both BFCL benchmarks under the same history-KV budget. On BFCL Long Context, mean task success increases from 10.5% to 15.1%, achieving a 43.7% relative improvement over compression-only execution. On BFCL Base, mean task success increases from 20.3% to 24.2%. On ToolSandbox, Racer improves similarity on four of six backends, increasing the mean from 0.538

Table 1: **Closed-loop performance under a fixed history budget.** Only denotes compression without Racer. Parentheses indicate the ratio of mean history length in Full runs to the 256-token budget. The cap counts active history-KV positions for KV methods and rendered history tokens for text methods, including restored evidence. Full is uncapped. BFCL reports task success (%), while ToolSandbox reports mean official similarity (0–1). Δ denotes the performance change, with positive gains shown in **bold**. Mean is the unweighted average across the six KV backends.

	BFCL Long Context (21.45×)			BFCL Base (2.23×)			ToolSandbox (1.82×)		
Method	Only	+ Racer	Δ ↑	Only	+ Racer	Δ ↑	Only	+ Racer	Δ ↑
Full (uncapped)	32.5	–	–	39.0	–	–	0.599	–	–
KV-level compression									
CommitKV	11.5	15.0	**+3.5**	25.0	26.5	**+1.5**	0.535	0.552	**+0.017**
StreamingLLM	13.0	15.0	**+2.0**	24.0	25.0	**+1.0**	0.556	0.561	**+0.005**
H2O	11.5	16.0	**+4.5**	20.0	25.5	**+5.5**	0.552	0.533	-0.019
SnapKV	11.5	14.0	**+2.5**	19.5	23.0	**+3.5**	0.517	0.541	**+0.024**
PyramidKV	12.5	16.0	**+3.5**	15.0	21.0	**+6.0**	0.561	0.542	-0.019
C^2KV	3.0	14.5	**+11.5**	18.5	24.0	**+5.5**	0.505	0.580	**+0.075**
Mean	10.5	15.1	**+4.6**	20.3	24.2	**+3.8**	0.538	0.551	**+0.013**
Text-level compression									
HiAgent	15.0	–	–	21.5	–	–	0.621	–	–
ACON	3.5	–	–	11.0	–	–	0.515	–	–

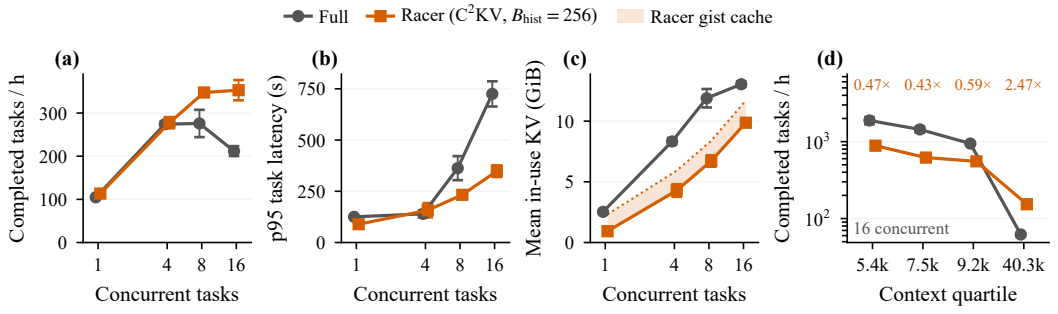


Figure 3: Serving on BFCL Long Context. (a)–(c) Concurrency sweep over 64 fixed tasks: completed tasks per hour, 95th-percentile task latency, and time-weighted mean in-use KV (Appendix G.2); the shaded band adds Racer's reusable gist cache. (d) Completed tasks per hour for each context-length quartile of all tasks at 16 concurrent tasks, on a logarithmic axis; ticks give each quartile's median maximum prompt length under Full, and annotations give the ratio of Racer to Full. Points are means over 3 batches; bars show one standard deviation.

to 0.551. Racer also improves task success across all six backends on BFCL Base at history budgets of 128 and 512 tokens (Appendix F.1).

Racer makes KV compression more competitive with text-level history compression. Without Racer, mean performance across the six KV backends is below HiAgent on both BFCL benchmarks; with Racer, it becomes comparable to HiAgent on BFCL Long Context and exceeds it on BFCL Base. HiAgent remains ahead on ToolSandbox.

4.3 Serving Efficiency

We serve BFCL Long Context tasks concurrently using a single SGLang engine on one GPU, comparing Full with C^2KV under Racer at $B_{\text{hist}} = 256$. Figure 3(a)–(c) sweeps concurrency over a fixed, length-stratified subset of 64 tasks; Figure 3(d) serves each context-length quartile of all 200 tasks separately at 16 concurrent tasks. Appendix G gives the configuration, metric definitions, and complete results.

Throughput and tail latency under concurrency. At sixteen concurrent tasks, Racer completes 1.67× as many tasks per hour as Full and lowers the 95th-percentile task latency from 725 s to 348 s.

Table 2: **Selective tool-definition KV compression on Agent-LLM-Traces.** Results are reported on common evaluation decisions within each experimental setting. Quality metrics are reported as percentages. Comp. denotes the arithmetic mean of per-decision tool-KV compression ratios.

Method	Tool Name ↑	Token-F1 ↑	ROUGE-L ↑	Comp. ↑
Full	52.75	53.92	52.94	1.00×
C2KV	38.46	45.13	43.92	2.42×
C2KV + Racer	52.75	49.55	48.47	2.42×
CacheBlend + SnapKV	39.56	44.98	43.89	7.95×
CacheBlend + SnapKV + Racer	43.96	43.31	42.33	7.95×
EPIC + SnapKV	42.86	46.25	44.95	7.95×
EPIC + SnapKV + Racer	43.96	46.89	45.94	7.95×

Its mean in-use KV is 24% lower, or 10% lower when its reusable gist cache is included, and the mean engine queue falls from 7.31 to 1.22 requests.

Recovery overhead. All timings are end-to-end and therefore include recovery. Racer regenerates 18–22% of decisions at every concurrency level, and the median time from the end of a discarded draft to the end of its regenerated action rises from 0.6 s with one task to 2.8 s with sixteen (Figure 6 in Appendix G.3).

The benefit is confined to the longest contexts. Racer is slower than Full in the three shorter quartiles (0.43–0.59× Full's throughput), where Full's mean engine queue stays below 0.3 requests. In the longest quartile (median 40.3k tokens), Full queues 11.15 requests on average, and Racer completes 2.47× as many tasks per hour. The serving benefit is thus a trade-off observed on congested long-context workloads, not a general speedup.

4.4 Tool-Definition KV Management

We evaluate Racer on Agent-LLM-Traces by compressing only tool definitions. The framework retains the top-3 query-relevant tool definitions in full while compressing the remaining definitions. We examine C2KV, CacheBlend + SnapKV, and EPIC + SnapKV under their respective experimental configurations. The CacheBlend-based configuration uses a value-difference KV fusion variant with SnapKV (Yao et al., 2025), while the EPIC-based configuration uses prefix recomputation following EPIC's KV reuse strategy (Hu et al., 2024). Appendix B.8 provides the implementation details.

Table 2 reports prediction quality and tool-KV compression. With C2KV, selective allocation increases tool-name accuracy from 38.46% to 52.75% and Token-F1 from 45.13 to 49.55. Both SnapKV-based configurations show increases in tool-name accuracy. EPIC + SnapKV also shows improvements in Token-F1 and ROUGE-L, whereas CacheBlend + SnapKV exhibits decreases in both text metrics. These results illustrate the backend-dependent effects of selective tool allocation.

4.5 Ablation Studies

We ablate three components of Racer's intervention framework: event recovery, argument correction, and protection. Event recovery uses risk-triggered event retrieval and regeneration; when event recovery is not performed, argument correction may replace an eligible tool-call argument with a source-verified request literal or observed list entry. Protection retains selected historical sources within the compressor's KV budget before generation. Starting from compression alone, Table 3 cumulatively enables these components in that order on BFCL Base at $B_{\text{hist}} = 256$.

Each addition increases mean task success. Event recovery raises mean success across the four compressors from 22.1% to 23.2%. Enabling argument correction increases it to 23.6%, and adding protection reaches 25.0%, a gain of 2.9 percentage points over compression alone. StreamingLLM reaches 26.0% with event recovery and argument correction and 25.0% after adding protection. Appendix F.4 reports the remaining component combinations, paired task outcomes, and the C^2KV decomposition.

Table 3: **Ablation study.** BFCL Base task success (%) ↑ on multi-turn task episodes per backend at $B_{\text{hist}} = 256$, one greedy run per cell. Rows cumulatively enable event recovery, argument correction, and protection. Mean averages the four compressors. Evaluation details and additional combinations are in Appendix F.4.

Intervention setting	SnapKV	H2O	StreamingLLM	CommitKV	Mean
Compression only	19.5	20.0	24.0	25.0	22.1
+ Event recovery	20.5	<u>21.0</u>	<u>25.5</u>	<u>26.0</u>	23.2
+ Argument correction	<u>21.0</u>	<u>21.0</u>	**26.0**	**26.5**	<u>23.6</u>
+ Protection (full)	**23.0**	**25.5**	25.0	**26.5**	**25.0**

5 Related Work

Agent Context Management. Text-level methods such as HiAgent (Hu et al., 2025), ACON (Kang et al., 2025), AgentFold (Ye et al., 2025a), and Context-Folding (Sun et al., 2025) reduce interaction histories through summarization or folding. Memex(RL) (Wang et al., 2026) maintains indexed external experiences for on-demand retrieval. KV-level approaches, including AgentKV (Liu et al., 2026a), CommitKV (Huang et al., 2026), and online KV compaction (Liu et al., 2026b), adapt history retention, while SideQuest (Kariyappa & Suh, 2026) uses auxiliary reasoning to evict stale tool responses.

KV Cache Compression and Recovery. StreamingLLM (Xiao et al., 2024) retains attention sinks and recent tokens, while H2O (Zhang et al., 2023), SnapKV (Li et al., 2024), and PyramidKV (Cai et al., 2024) selectively retain important KV states. C^2KV (Du et al., 2026) provides compressed, reusable document-level KV blocks, while EPIC (Hu et al., 2024) enables position-independent KV reuse. RepairKV (Rusli et al., 2026) restores offloaded KV states during decoding pauses under matched active-cache budgets. In contrast, Racer uses a lightweight risk detector to assess uncommitted action drafts, selectively restores historical evidence relevant to the proposed tool and arguments, and regenerates the action under the same history-KV budget before execution.

6 Conclusion

In this work, we identify compression-induced trajectory drift as a key challenge in KV cache compression for multi-turn LLM agents. We show that compression errors can propagate across subsequent decisions once incorrect actions affect the environment. To address this, we propose Racer, a recoverable KV cache management framework combining selective tool-definition compression, pre-generation history protection, and pre-action intervention through risk-triggered evidence recovery or source-verified argument correction. Under the same history-KV budget, Racer improves task success across all six KV-compression backends on both BFCL benchmarks, achieving a 43.7% relative improvement in mean task success on BFCL Long Context. Concurrent serving experiments further demonstrate 1.67× higher task throughput than full-context execution.

AI USE STATEMENT

Large language models assisted with language editing, research ideation, and implementation. The authors reviewed the technical content and results and take responsibility for the paper.

REFERENCES

Amey Agrawal, Nitin Kedia, Ashish Panwar, Jayashree Mohan, Nipun Kwatra, Bhargav Gulavani, Alexey Tumanov, and Ramachandran Ramjee. Taming {Throughput-Latency} tradeoff in {LLM} inference with {Sarathi-Serve}. In *18th USENIX symposium on operating systems design and implementation (OSDI 24)*, pp. 117–134, 2024.

Elron Bandel, Asaf Yehudai, Lilach Eden, Yehoshua Sagron, Yotam Perlitz, Elad Venezian, Natalia Razinkov, Natan Ergas, Shlomit Shachor Ifergan, Segev Shlomov, et al. General agent evaluation. *arXiv preprint arXiv:2602.22953*, 2026.

Zefan Cai, Yichi Zhang, Bofei Gao, Yuliang Liu, Yucheng Li, Tianyu Liu, Keming Lu, Wayne Xiong, Yue Dong, Junjie Hu, et al. Pyramidkv: Dynamic kv cache compression based on pyramidal information funneling. *arXiv preprint arXiv:2406.02069*, 2024.

Chuheng Du, Junyi Chen, Hanlin Tang, Kan Liu, Tao Lan, Lin Qu, Chaoyue Niu, Shengzhong Liu, Guihai Chen, and Fan Wu. C2kv: Compressed and composable kv cache reuse for efficient llm inference. In *Proceedings of the 32nd ACM SIGKDD Conference on Knowledge Discovery and Data Mining V. 2*, pp. 991–1002, 2026.

Coleman Hooper, Sehoon Kim, Hiva Mohammadzadeh, Michael W Mahoney, Yakun S Shao, Kurt Keutzer, and Amir Gholami. Kvquant: Towards 10 million context length llm inference with kv cache quantization. *Advances in Neural Information Processing Systems*, 37:1270–1303, 2024.

Junhao Hu, Wenrui Huang, Weidong Wang, Haoyi Wang, Tiancheng Hu, Qin Zhang, Hao Feng, Xusheng Chen, Yizhou Shan, and Tao Xie. Epic: Efficient position-independent caching for serving large language models. *arXiv preprint arXiv:2410.15332*, 2024.

Mengkang Hu, Tianxing Chen, Qiguang Chen, Yao Mu, Wenqi Shao, and Ping Luo. Hiagent: Hierarchical working memory management for solving long-horizon agent tasks with large language model. In *Proceedings of the 63rd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pp. 32779–32798, 2025.

Weizhong Huang, Jinchao Zhang, and Xiawu Zheng. Commitkv: Lifecycle-aware kv cache compression via commit transitions for multi-turn agents. *arXiv preprint arXiv:2608.07855*, 2026.

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.

Sanjay Kariyappa and G Edward Suh. Sidequest: Model-driven kv cache management for long-horizon agentic reasoning. *arXiv preprint arXiv:2602.22603*, 2026.

Woosuk Kwon, Zhuohan Li, Siyuan Zhuang, Ying Sheng, Lianmin Zheng, Cody Hao Yu, Joseph Gonzalez, Hao Zhang, and Ion Stoica. Efficient memory management for large language model serving with pagedattention. In *Proceedings of the 29th symposium on operating systems principles*, pp. 611–626, 2023.

Yuhong Li, Yingbing Huang, Bowen Yang, Bharat Venkitesh, Acyr Locatelli, Hanchen Ye, Tianle Cai, Patrick Lewis, and Deming Chen. Snapkv: Llm knows what you are looking for before generation. *Advances in Neural Information Processing Systems*, 37:22947–22970, 2024.

Taowen Tony Liu, Jeffrey TH Wong, Can Xiao, Bowen Yang, Hao Mark Chen, and Yiren Zhao. Agentkv: Phase-aware kv eviction for agentic llms. *arXiv preprint arXiv:2609.14872*, 2026a.

Yujian Liu, Jiabao Ji, Li An, Rohit Jain, Gungor Polatkan, Siyu Zhu, and Shiyu Chang. Practical online kv cache compaction for llm agents: An empirical study. *arXiv preprint arXiv:2608.00902*, 2026b.

Zirui Liu, Jiayi Yuan, Hongye Jin, Shaochen Zhong, Zhaozhuo Xu, Vladimir Braverman, Beidi Chen, and Xia Hu. Kivi: A tuning-free asymmetric 2bit quantization for kv cache. *arXiv preprint arXiv:2402.02750*, 2024.

Jiarui Lu, Thomas Holleis, Yizhe Zhang, Bernhard Aumayer, Feng Nan, Haoping Bai, Shuang Ma, Shen Ma, Mengyu Li, Guoli Yin, et al. Toolsandbox: A stateful, conversational, interactive evaluation benchmark for llm tool use capabilities. In *Findings of the Association for Computational Linguistics: NAACL 2025*, pp. 1160–1183, 2025.

Adyasha Maharana, Dong-Ho Lee, Sergey Tulyakov, Mohit Bansal, Francesco Barbieri, and Yuwei Fang. Evaluating very long-term conversational memory of llm agents. In *Proceedings of the 62nd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pp. 13851–13870, 2024.

Shishir G Patil, Huanzhi Mao, Fanjia Yan, Charlie Cheng-Jie Ji, Vishnu Suresh, Ion Stoica, and Joseph E Gonzalez. The berkeley function calling leaderboard (bfcl): From tool use to agentic evaluation of large language models. In *Forty-second International Conference on Machine Learning*, 2025.

Andrew Rusli, Shreyan Paliwal, and Michael Jiao. Cache you later: Post-compression kv repair for long-context agentic llm inference. In *AdaptFM: Resource-Adaptive Foundation Model Inference*, 2026.

Weiwei Sun, Miao Lu, Zhan Ling, Kang Liu, Xuesong Yao, Yiming Yang, and Jiecao Chen. Scaling long-horizon llm agent via context-folding. *arXiv preprint arXiv:2510.11967*, 2025.

Hanlin Tang, Yang Lin, Jing Lin, Qingsen Han, Danning Ke, Shikuan Hong, Yiwu Yao, and Gongyi Wang. Razorattention: Efficient kv cache compression through retrieval heads. In *International Conference on Learning Representations*, volume 2025, pp. 16632–16646, 2025.

Zhenting Wang, Huancheng Chen, Jiayun Wang, and Wei Wei. Memex (rl): Scaling long-horizon llm agents via indexed experience memory. *arXiv preprint arXiv:2603.04257*, 2026.

Di Wu, Hongwei Wang, Wenhao Yu, Yuwei Zhang, Kai-Wei Chang, and Dong Yu. Longmemeval: Benchmarking chat assistants on long-term interactive memory. *arXiv preprint arXiv:2410.10813*, 2024.

Guangxuan Xiao, Yuandong Tian, Beidi Chen, Song Han, and Mike Lewis. Efficient streaming language models with attention sinks. In *International Conference on Learning Representations*, volume 2024, pp. 21875–21895, 2024.

Guangxuan Xiao, Jiaming Tang, Jingwei Zuo, Junxian Guo, Shang Yang, Haotian Tang, Yao Fu, and Song Han. Duoattention: Efficient long-context llm inference with retrieval and streaming heads. In *International Conference on Learning Representations*, volume 2025, pp. 37228–37253, 2025.

An Yang, Anfeng Li, Baosong Yang, Beichen Zhang, Binyuan Hui, Bo Zheng, Bowen Yu, Chang Gao, Chengen Huang, Chenxu Lv, et al. Qwen3 technical report. *arXiv preprint arXiv:2505.09388*, 2025.

John Yang, Carlos Jimenez, Alexander Wettig, Kilian Lieret, Shunyu Yao, Karthik Narasimhan, and Ofir Press. Swe-agent: Agent-computer interfaces enable automated software engineering. *Advances in Neural Information Processing Systems*, 37:50528–50652, 2024.

Jiayi Yao, Hanchen Li, Yuhan Liu, Siddhant Ray, Yihua Cheng, Qizheng Zhang, Kuntai Du, Shan Lu, and Junchen Jiang. Cacheblend: Fast large language model serving for rag with cached knowledge fusion. In *Proceedings of the twentieth European conference on computer systems*, pp. 94–109, 2025.

Shunyu Yao, Howard Chen, John Yang, and Karthik Narasimhan. Webshop: Towards scalable real-world web interaction with grounded language agents. *Advances in Neural Information Processing Systems*, 35:20744–20757, 2022a.

Shunyu Yao, Jeffrey Zhao, Dian Yu, Nan Du, Izhak Shafran, Karthik Narasimhan, and Yuan Cao. React: Synergizing reasoning and acting in language models. *arXiv preprint arXiv:2210.03629*, 2022b.

Rui Ye, Zhongwang Zhang, Kuan Li, Huifeng Yin, Zhengwei Tao, Yida Zhao, Liangcai Su, Liwen Zhang, Zile Qiao, Xinyu Wang, et al. Agentfold: Long-horizon web agents with proactive context management. *arXiv preprint arXiv:2510.24699*, 2025a.

Zihao Ye, Lequn Chen, Ruihang Lai, Wuwei Lin, Yineng Zhang, Stephanie Wang, Tianqi Chen, Baris Kasikci, Vinod Grover, Arvind Krishnamurthy, et al. Flashinfer: Efficient and customizable attention engine for llm inference serving. *Proceedings of Machine Learning and Systems*, 7, 2025b.

Zhenyu Zhang, Ying Sheng, Tianyi Zhou, Tianlong Chen, Lianmin Zheng, Ruisi Cai, Zhao Song, Yuandong Tian, Christopher Ré, Clark Barrett, et al. H2o: Heavy-hitter oracle for efficient generative inference of large language models. *Advances in neural information processing systems*, 36:34661–34710, 2023.

Lianmin Zheng, Liangsheng Yin, Zhiqiang Xie, Chuyue Sun, Jeff Huang, Cody H Yu, Shiyi Cao, Christos Kozyrakis, Ion Stoica, Joseph E Gonzalez, et al. Sglang: Efficient execution of structured language model programs. *Advances in neural information processing systems*, 37:62557–62583, 2024.

Yinmin Zhong, Shengyu Liu, Junda Chen, Jianbo Hu, Yibo Zhu, Xuanzhe Liu, Xin Jin, and Hao Zhang. {DistServe}: Disaggregating prefill and decoding for goodput-optimized large language model serving. In *18th USENIX symposium on operating systems design and implementation (OSDI 24)*, pp. 193–210, 2024.

A Appendix Outline

In this appendix, we report the following sections as supplementary materials for the main paper.

- Appendix B describes the benchmarks, models, compression baselines, recovery settings, and serving environment used in our experiments.
- Appendix C gives the decision loop and the admission and lifetime of recovered evidence.
- Appendix D defines the task, KV-compression, recovery, detector, and efficiency metrics reported throughout the paper.
- Appendix E describes compressor training data, optimization and checkpoint selection, and the recovery detector's data splits, label construction, and fitting procedure.
- Appendix F reports cross-backend budget comparisons, task-success–memory trade-offs, text-level comparisons across budgets, component ablations, risk detection, and tool-definition allocation and routing ablations.
- Appendix G describes the concurrent-serving study and reports its per-condition throughput, latency, KV occupancy, memory accounting, and recovery overhead.

B Experimental Setup

B.1 Benchmarks and Evaluation

Table 4 specifies the closed-loop task cohorts and official scorers. BFCL budgeted native arms use greedy model decoding (temperature 0, seed 0), at most 4,096 completion tokens per call, and at most 96 generation calls per task, including drafts and regenerations. Each reported paired condition is run once; tasks ending in a method failure count as unsuccessful.

Table 4: **Closed-loop evaluation cohorts.** The ToolSandbox cohort comprises 129 scenarios with three distractor tools, selected from 1,032 available scenarios.

Benchmark	Evaluated tasks	Simulated user	Official score
BFCL Base	200 multi-turn Base tasks	None	Task success from state and response checks
BFCL Long Context	200 multi-turn Long Context tasks	None	Task success from state and response checks
ToolSandbox	129 three-distractor-tool scenarios	Official user role	Mean scenario similarity

ToolSandbox includes a simulated user, served by the same SGLang server. Its requests go directly to that server and bypass history management, so its own context is never budgeted or compressed. We use the official simulated-user role, configured to access our SGLang endpoint and handle empty tool-call lists. This role sends no sampling parameters, so SGLang applies the model's default chat sampling: temperature 0.7, top-p 0.8, and top-k 20. The server seed is 42, but requests carry no per-request seed, so simulated-user turns can differ across runs.

B.2 Model and Runtime

All methods use Qwen3-4B-Instruct-2507. Closed-loop experiments run through SGLang. Racer and C^{2}KV use the history-compression checkpoint saved after 1,000 optimizer steps.

B.3 History Capacity and Baselines

The main paired comparison uses a 256-token history capacity. For KV methods, the cap counts retained historical KV positions, including compressed representations and restored native evidence during regeneration; for text methods, it counts the complete rendered historical text, including summaries and retrieved trajectories. System instructions, raw tool schemas, and the current request

are outside the history cap. Full has no history cap. Table 5 gives the backend settings used in the main comparison.

Table 5: History backend settings in the 256-token cross-backend comparison. The native KV backends use persistent sessions; the selector settings govern retention within the shared capacity.

Backend	Retained-history rule	Run configuration
CommitKV	Event-page retention with pending-page protection	Reference-attention backend
StreamingLLM	Recent-history retention	Physical eviction; recent window 64
H2O	Heavy-hitter and recent-history retention	Physical eviction; recent window 64; recent fraction 0.5
SnapKV	Pooled attention over old positions plus recent history	Physical eviction; recent window 64; average-pooling kernel 5
PyramidKV	Layer/head-specific retention using pooled attention	Reference-attention backend; recent window 64; average-pooling kernel 5
C^2KV	Compressed event gists and selected native history	Nominal source-to-gist ratio 8:1

The component study crosses protection and recovery at 256 tokens (Appendix F.4). Additional comparisons use 128 and 512 tokens on BFCL Base (Appendix F.1).

B.4 Text-Level History Compression

The comparison in Section 4.2 uses budget-constrained HiAgent and ACON adaptations at 256 tokens; Appendix F.3 additionally evaluates BFCL Base at 128 and 512 tokens. Before each model generation, the SGLang renderer checks the rendered history against the selected cap. Summarization uses additional model calls.

HiAgent retains subgoal summaries and exposes an internal retrieval tool for completed trajectories. The adapter evicts the oldest history records to meet the cap. If a requested trajectory cannot fit, the model receives a budget-unavailable response and continues. ACON reuses the published UT-to-CO compression guidelines, transferred to BFCL without task-specific guideline optimization. It maintains a rolling summary and adds an explicit output-token constraint, allowing up to three compression attempts to fit the rendered history. Tasks that cannot satisfy the history cap are counted as unsuccessful.

B.5 Recovery Settings

The risk threshold is $s > 0.5$, with at most one regeneration per decision. Appendix C specifies the recovery operations; Appendix E.5 gives the detector features and fitting procedure.

B.6 Main Cross-backend Study

Table 1 evaluates Racer on BFCL Base, BFCL Long Context, and ToolSandbox. Each pair uses the same benchmark tasks, model checkpoint, decoding settings, and GPU model per task. The detector and its calibration are transferred unchanged from the learned-compression setting.

B.7 History-Budget Sweep

The sweep evaluates C^2KV with and without Racer on BFCL Base at history capacities of 128, 160, 192, 224, 256, 384, 512, and 1,024 token-equivalent positions. Both conditions use the same history-compression checkpoint and a nominal source-to-gist ratio of eight. Every budget evaluates the same task IDs under both conditions, with the GPU model matched within each pair.

B.8 Tool-definition Study

Table 2 evaluates recorded next-action predictions on Agent-LLM-Traces by compressing only tool definitions. We compare uniform compression with selective allocation, which retains the top-3

query-relevant tool definitions in full and compresses the remaining definitions. Each backend pair is evaluated on a common subset of 91 valid decisions under the same experimental configuration.

The C2KV experiments use per-tool document compression, where individual tool definitions are processed as separate documents. The uniform and selective variants use requested compression ratios of 2.43 and 4, respectively. The SnapKV-based experiments use full-document compression with a requested ratio of 8, treating the compressible tool definitions as a single block. Table 2 reports the measured rather than requested compression ratios.

For CacheBlend + SnapKV, we combine SnapKV compression with value-difference-based KV fusion when incorporating the compressed tool block into the preceding context, following CacheBlend (Yao et al., 2025). For EPIC + SnapKV, the tool definitions are independently prefilled and compressed with SnapKV. When linking the compressed block to the preceding native context, leading retained KV positions are recomputed following EPIC's static recomputation strategy (Hu et al., 2024). Under selective allocation, the top-3 definitions remain uncompressed, while the remaining definitions form the block passed through the corresponding compression and context-reuse procedure.

The C2KV and SnapKV-based experiments use different compression checkpoints and document organization settings. Both checkpoints are trained from the same Qwen3-4B-Instruct-2507 backbone, with the backbone frozen and only the additional compression modules updated. Since Full bypasses these compression modules, both checkpoints share the same underlying full-context model. Comparisons between uniform compression and selective allocation are conducted within each experimental configuration.

C Runtime Details

C.1 Decision Loop

Algorithm 1 gives the Racer decision loop. Tool allocation follows the setting of each experiment. Each decision permits one regeneration; drafts and regenerations both count toward the generation limit.

Algorithm 1 One decision of Racer

Require: tool catalog T, archive $\mathcal{A}_t$, live input U_t, history capacity B_{hist}, risk threshold τ

Form T_t with native KV for the selected complete tool definitions and compressed KV for all remaining definitions; in history-only experiments, retain all tool definitions as native KV
Allocate history H_t using shared protection and the backend representation within B_{hist}
$\widehat{C}_t \leftarrow \text{Compose}(T_t, H_t, U_t)$
Prefill $\widehat{C}_t$; decode held draft $\tilde{a}_t$ and extract risk features z_t
$a_t \leftarrow \tilde{a}_t$; $\widehat{C}'_t \leftarrow \emptyset$
if a generation call remains **then**
    $s_t \leftarrow \sigma(w^\top \widehat{z}_t + b_{\text{det}})$
    if $s_t > \tau$ **then**
        Rank complete archived events using the request, draft, and latest observation
        $\widehat{C}'_t \leftarrow$ the first feasible, changed view from $\text{Repack}(\widehat{C}_t, e, B_{\text{hist}})$
    end if
    if $\widehat{C}'_t \neq \emptyset$ **then**
        Attempt one regeneration from $\widehat{C}'_t$ to obtain a_t
        Set $a_t \leftarrow \tilde{a}_t$ if capacity rejects recovery or the regenerated action is malformed
    else
        Apply source-verified binding corrections to eligible arguments of a_t
    end if
end if
Commit a_t and archive the resulting interaction

C.2 Initial History Allocation

For C^{2}KV, initial allocation first protects the latest complete tool event when it fits, then admits request-relevant history. Its lexical query combines the current user request with argument values of tool calls since that request. From the most recent eligible complete tool events, it ranks at most twelve indexed candidates using the lexical score defined below and requests the first two; each native admission is measured against the remaining history capacity. The tool-event bonus in that score is constant within this initial candidate pool. If protection of the latest event prevents the required representation from fitting, the policy removes that optional protection and recomputes the request ranking. A value cue links an argument value to its matching observation value; a failure cue records a recent failed operation relevant to the request. Admitted cues and retained history share B_{hist}. The live input contains system instructions and the suffix following the assistant message preceding the latest user or tool input.

The C^{2}KV allocator first uses complete events and applies its capacity fallbacks before switching to source-message allocation if they remain infeasible. It retries at the requested compression ratio, removes gist copies made redundant by native sources, tries a smaller native rendering of complete tool-event narrative, then tries projected tool-call arguments alone and with that narrative projection. Projected raw content retains complete-event gist backing. At source-message granularity, C^{2}KV assigns retained messages to native or compressed representations. Token-selecting backends retain historical KV by their serving-engine rule; when source protection is enabled, the engine admits the requested source units within its capacity. Mandatory live input is preserved throughout, and KV occupancy is measured from the serving engine.

C.3 Evidence Admission and Lifetime

Racer forms a recovery query from the current request, held draft text, latest complete tool event (call and observations), and the names and argument values of valid parsed draft calls. Let Q be its non-stopword token set, T_e the tokens of an event e, and h_e the number of case-folded query anchors found in that event's text. Anchors are quoted spans, identifiers containing a digit or separator, and normalized parsed argument values. The deterministic rank is

$$s(e) = 100h_e + 4|Q \cap T_e| + \frac{|Q \cap T_e|}{|Q \cup T_e|} + 8\mathbf{1}[e \text{ is a tool event}].$$

Only complete, eligible events with an anchor or token match are ranked; instruction events, the current user event, already native or protected events, and revision-cancelled events are excluded. Ties prefer the more recent source message, then the lexicographically smaller event ID. The runtime tries candidates in this order and restores the first one whose repacked view both fits the history and model-context limits and differs from the draft view. That complete event becomes native evidence protected during regeneration. Evidence admission also reserves the KV that the backend must retain, including any still-protected CommitKV pending pages.

Repacking first tries the existing allocation, then the finer source-message allocation if needed, while keeping the complete recovery target native. For C^{2}KV, optional native history is demoted to its compressed representation before optional compressed history is released from the active view; demotion retains the source in compressed KV, whereas release removes its active KV. Admission removes the target's compressed source copies when the complete native copy replaces them. For token-selecting backends, the serving engine applies the backend's retention rule within the residual capacity. If evidence cannot be admitted, the held draft is retained; a malformed regenerated action also falls back to that draft.

The C^{2}KV runtime assembles cached compressed blocks with native context before prefill. Risk features are extracted from the held draft and the last prompt-token hidden state. When recovery changes the context, the runtime prefills the revised view and generates the action once more.

When no recovered view is selected and generation allowance remains, source-binding rules inspect parsed tool calls. They can replace a string argument with a unique credential or quoted-text literal from the current request, or resolve a first/last ordinal against a unique observed list result. An ambiguous literal, competing list result, or intervening mutation prevents that correction. The runtime verifies the source span or observation version and preserves the tool names, call structure, and all

unproved arguments; if no binding verifies, it commits the held draft unchanged. This correction makes no additional model call. At the next decision, the shared policy allocates history again.

C.4 Efficient Serving with C^{2}KV

KV reuse. The runtime caches compressed blocks by source token IDs, compression ratio, and encoder type. It looks up the blocks selected for a context in a batch and encodes missing blocks before generation. The engine separately stores the initial native prefix preceding the first compressed block in SGLang's RadixCache. A request matches this cache by its leading native tokens and prefills the remaining prefix before injecting compressed KV.

Background history compression. During model generation, the runtime prepares additional chunks from completed, observed history for future context assembly. It first reserves the compression calls required by the selected context, then admits uncached candidate events whose compressed representation fits the history capacity and remaining call allowance. After the foreground request is admitted, the engine schedules their encoding on a dedicated CUDA stream alongside its decode stream. A single background worker computes the compressed KV, and the scheduler publishes completed blocks into the shared cache. Subsequent decisions retrieve these blocks through the same lookup used for foreground compression. Cache capacity bounds the background work admitted by the engine.

Incremental context preparation. A session-local token cache reuses complete chat-template encodings and rendered segments, including the system and tool prologue. Cache entries depend on message content, tool definitions, generation-prompt settings, and tokenizer configuration. Repacking a context reuses unchanged segments and tokenizes new or modified segments. A CPU worker also prepares history chunks from a snapshot of the observed conversation while generation proceeds. The runtime consumes a completed chunk plan when its source remains an unchanged prefix of the current session.

Concurrent serving. Persistent runtime workers share a single SGLang engine, which batches requests across sessions and uses CUDA graphs for decoding. Tasks are dispatched to the next available worker. Each worker retains its loaded modules and tokenizer across tasks, while constructing a fresh controller and session state for each task.

Execution costs. Compression of missing blocks selected for the current context precedes its prefill; background compression prepares blocks that later decisions can reuse. Recovery adds evidence selection, context preparation, and another model generation to the current decision. Section 4.3 measures the end-to-end throughput of this implementation, including tool execution and scoring. Appendix G defines the timing window and KV accounting and reports the post-draft time spent on recovery.

D Metrics

Table 6 summarizes the aggregation and coverage of the history, tool-definition, and serving measurements.

Task success. For N tasks with binary official outcomes y_i, task success is $100 \sum_i y_i / N$. BFCL uses its state and response checks. ToolSandbox reports mean official scenario similarity, $\sum_i s_i / N$, where $s_i \in [0, 1]$. Table 1 reports BFCL differences in percentage points and ToolSandbox differences in similarity score.

Budget-sweep memory measurements. For task i, let V_{ig} be the logical active model-KV bytes recorded by the serving engine for completed generation g. We take the larger of its post-prefill and final-generation receipts, then report $\overline{P} = N^{-1} \sum_i \max_g V_{ig}$ in MiB. This covers system instructions, tool definitions, live input, retained history, restored evidence, and generated tokens. It measures the active request's KV rather than shared prefix-cache storage, model weights, or total device memory. Drafts and regenerations both contribute to the task maximum.

Let F_g be the uncompressed history length for the same generation prefix and A_g its active historical KV, including raw history, compressed positions, and restored evidence. Effective history compression is $C_{\text{hist}} = \sum_g F_g / \sum_g A_g$, pooled over all completed generations of the evaluated

Table 6: **Compression and KV measurement scopes.** Ratios are dimensionless; memory measurements use the units shown.

Measurement	Aggregation	Coverage and location
Full-history length / budget	Mean history length over measured Full-run model inputs, divided by the fixed history budget	Historical input only; Table 1 column headings.
Realized history compression	Total uncompressed history positions divided by total active history positions, over completed generations	Native history, compressed history, and restored evidence along each method's own trajectory; Appendix D.
Tool-KV compression	Mean of per-decision full-to-retained tool-definition ratios	Native schemas and compressed tool KV; Table 2 and Appendices F.6–F.7.
Logical peak model KV (MiB)	Maximum recorded active model KV per task, then mean over tasks	Whole active request, including drafts and regenerations; Appendix D.
Time-weighted in-use KV (GiB)	Integral of shared-engine in-use KV over the batch, divided by batch duration	Active main-pool KV, pinned gist KV, and pending compression KV; reusable gist cache is reported separately. Appendix G.2.

tasks. Controller receipts provide both lengths in token-equivalent positions. System instructions, tool definitions, and live input are excluded from this ratio; empty-history inputs contribute zero to both sums. This is a ratio of totals, not a mean of per-generation compression ratios. Each condition follows its own closed-loop trajectory, so the two conditions can have different generation counts and uncompressed-history totals.

Tool-definition metrics. Token-F1 measures token overlap between the prediction and recorded response; ROUGE-L is the F1 score of their longest common token subsequence. Both are averaged over decisions and reported as percentages. Tool-name accuracy checks the first predicted tool name. Tool-KV compression is calculated as the arithmetic mean of per-decision compression ratios, where each ratio divides the original tool-definition token count by the total retained native and compressed tool-KV positions. All decisions receive equal weight, regardless of their original tool-catalog lengths.

Detector metrics. The detector predicts whether the current user turn fails following a held draft under the no-recovery continuation defined in Appendix E.4. We report AUROC, average precision, and thresholded precision, recall, and balanced accuracy on labeled calibration decisions.

E Training Data and Configurations

E.1 C^{2}KV Compressor Training

We use Qwen3-4B-Instruct-2507 as the frozen language-model backbone and train separate encoders for tool definitions and history. Training updates the gist embeddings and compression-specific QKV projections. The decoder weights remain fixed.

Each training example consists of compressible sources X, an uncompressed native context U, and a target continuation y. The encoder produces compressed KV states $E_\theta(X)$, which are composed with U before target generation. For tool definitions, X contains the tool schemas; for history, it contains completed interaction events preceding the target action. We optimize target-only cross-entropy:

$$\mathcal{L}_{\text{comp}}(\theta) = -\mathbb{E}_{(X,U,y)\sim\mathcal{D}_{\text{comp}}} \sum_j \log p_\phi \left(y_j \mid y_{<j}, \text{Compose}(E_\theta(X), U) \right), \tag{12}$$

where ϕ denotes the frozen decoder parameters. Native-context tokens are masked from the loss.

E.2 Compressor Training Data

History encoder. The history corpus contains 17,549 ratio-conditioned examples derived from 8,775 assistant decision points in 6,844 sessions (Table 7). It combines multi-turn Toucan trajectories, resolved Open-SWE trajectories, document question answering from HotpotQA, 2Wiki, and LongMagpie, and the training split of Agent-LLM-Traces. The Open-SWE subset contains OpenHands and SWE-agent trajectories generated by MiniMax-M2.5 and Qwen3.5.

The adapters preserve message roles and associate tool observations with their calls. At each eligible assistant decision, completed historical events form the compressible input, while the current workspace and target continuation remain native. Question-answering examples place the source documents in history and supervise the answer to the current question. We sample decision points with seed 42 and construct examples at nominal compression ratios of four and eight. Preparation uses a 48-million-token budget, at most 64 decision points per session, and a QA sampling target of 15%. Table 7 reports the resulting corpus after packing and token-budget admission. Each epoch shuffles sessions while preserving decision order within a session.

Table 7: **History-compressor training corpus.** A decision point is an assistant continuation within a session; a training example additionally specifies its compression ratio.

Source	Sessions	Decision points	Training examples
Toucan multi-turn	3,378	4,294	8,588
Open-SWE resolved trajectories	1,563	2,571	5,141
HotpotQA / 2Wiki / LongMagpie	1,894	1,894	3,788
Agent-LLM-Traces	9	16	32
Total	6,844	8,775	17,549

History is packed into chunks of at most 768 tokens with a 64-token overlap and at most 48 chunks per example. The encoder-input cap is 36,864 tokens; the system, workspace, and target caps are 8,192, 4,096, and 4,096 tokens, respectively. The main sequence is capped at 16,384 tokens. The workspace retains the most recent complete tool event. Agent-LLM-Traces sessions are drawn from its task-proxy training partition; the source adapters exclude rows marked as development, validation, evaluation, or test data.

Tool-definition encoder. The repository tool-encoder recipe uses Agent-LLM-Traces (Bandel et al., 2026), whose source collection contains 1,781 execution traces. It shuffles session IDs with seed 42 and assigns 90% of sessions to training and 10% to evaluation before extracting decision examples. An example requires tool definitions, input messages, and a nonempty tool-call target. The recipe samples at most four eligible decisions per session. Tool catalogs exceeding 10,000 tokens are excluded; retained schemas are represented by at most ten chunks of up to 1,024 tokens each. The native prompt and target share a 2,048-token sequence limit. Prompt truncation reserves 128 target tokens, or the full target when shorter. The tool-definition comparison in Table 2 uses a common evaluation set of 91 decisions from 27 sessions.

Relationship to the evaluation tasks. BFCL is used for checkpoint selection and detector development. The main cross-backend evaluation covers the complete BFCL Base and Long Context sets, including development families. The BFCL-trained detector is transferred to ToolSandbox without additional fitting.

E.3 Optimization and Checkpoint Selection

The history encoder optimizes only compression parameters with AdamW, linear warmup, and cosine learning-rate decay (Table 8). History training uses one GPU with an effective batch size of 32. Its two-epoch run comprises 1,098 optimizer updates. The repository tool-encoder recipe also specifies AdamW, warmup, cosine decay, eight Ascend NPUs, and DeepSpeed ZeRO-3.

We select the history-encoder checkpoint by maximizing the total number of correct tasks across compression ratios four and eight on a 128-task BFCL development set. The selected checkpoint at step 1,000 is used throughout the history experiments; the main configuration uses ratio eight. The

Table 8: **History-encoder settings and repository tool-encoder recipe.** Effective batch size includes all devices and gradient accumulation.

Setting	History encoder	Tool-definition recipe
Optimizer	AdamW	AdamW
Learning rate	5×10^{-5}	5×10^{-7}
Weight decay	0.1	0.1
Learning-rate schedule	Cosine	Cosine
Warmup	6% of updates	5 updates
Epochs	2	5
Devices	1 GPU	8 Ascend NPUs
Batch size per device	2	1
Gradient accumulation	16	4
Effective batch size	32	32
Maximum main-sequence length	16,384	2,048
Compute precision	BF16	BF16
Checkpoint or output	Step 1,000	Final-save directory

tool-definition experiments use two independently trained compression checkpoints based on the same frozen Qwen3-4B-Instruct-2507 backbone. The per-tool C2KV experiments use the checkpoint saved after 1,088 optimizer steps, while the SnapKV-based experiments use the checkpoint saved after 1,000 steps.

E.4 Detector Data and Labels

Detector examples are collected with the language model and history compressor frozen. The source pool contains 104 BFCL tasks from 26 families, each comprising the Base, Long Context, Missing Function, and Missing Parameter variants of a source task. Sampling uses seed 0, at most eight pending states per family, and at most two per task variant. States are grouped by family before assignment to training or calibration.

The resulting collection contains 80 training states from 17 families and 38 calibration states from nine other families. A state contributes a binary training target when its preceding user-turn prefix passes the official checker and the continuation yields a defined current-turn outcome. This produces 62 training labels from 15 families and 36 calibration labels from nine families (Table 9). Other sampled states are omitted from fitting and detector-metric computation.

Table 9: **Detector data.** Family counts refer to states with binary labels. A positive label denotes failure of the current user turn following the held draft.

Split	Families	Labeled states	Pass	Fail
Training	15	62	27	35
Calibration	9	36	25	11

Operational target. At a pending decision t, the compressed-context draft $\tilde{a}_t$ is held before execution. To label it, we restore the corresponding environment state, submit that draft without recovery, and continue the frozen policy to task termination. We then apply BFCL's official state and response checks to the completed prefix ending at the user turn that contains t. The label in Eq. 11 is one when this turn fails and zero when it passes. The preceding prefix refers to earlier user turns; an individual tool step within the current turn is not scored as a completed turn. The supervision therefore includes the continuation after the held draft. The official end-state and response-path checks determine correctness, including their treatment of alternative execution paths.

Algorithm 2 Constructing detector labels

Require: Frozen language model and compressor; sampled pending states grouped into training and calibration sets

for each sampled state s_t in its assigned split **do**
  if the preceding user-turn prefix does not pass the official checker **then**
    Skip s_t
  else
    Restore the environment and compressed context at s_t
    Obtain the held draft $\tilde{a}_t$ and its features z_t
    Commit $\tilde{a}_t$ without recovery
    Continue the frozen policy to task termination
    $v_t \leftarrow$ official pass/fail result for the user turn containing t
    if v_t is defined **then**
      Add $(z_t,\ 1 - v_t)$ to the state's assigned split
    end if
  end if
end for

E.5 Detector Fitting

The feature vector in Eq. 8 contains eight PCA coordinates of the last prompt-token hidden state, the draft's mean token negative log-likelihood, a turn-stop indicator, and an action-parsing indicator. We read the hidden state from decoder layer 34 using zero-based indexing. The turn-stop indicator is one when the draft parses successfully and contains no tool call; the parsing indicator is one when action parsing succeeds. These features are computed before the held draft is executed.

We fit PCA on the training hidden states and standardize the resulting 11-dimensional vector using training means μ and standard deviations ν:

$$\widehat{z}_i = (z_i - \mu)/\nu, \qquad s_i = \sigma(w^\top \widehat{z}_i + b_{\mathrm{det}}).$$

The linear detector minimizes regularized binary cross-entropy:

$$\mathcal{L}_{\mathrm{det}} = \frac{1}{|\mathcal{I}|} \sum_{i \in \mathcal{I}} [-y_i \log s_i - (1 - y_i) \log(1 - s_i)] + \lambda \|w\|_2^2. \tag{13}$$

We select the inverse regularization strength from $C \in \{0.01, 0.1, 1.0\}$ by mean log loss in three-fold cross-validation grouped by source family. PCA and standardization are fitted separately within each training fold. The selected value is $C = 1.0$; we refit the preprocessing and classifier on all 62 labeled training states using L-BFGS with a maximum of 2,000 iterations. The 36 calibration labels are excluded from PCA, standardization, classifier fitting, and C selection; they provide the detector metrics in Table 16. The deployment threshold is fixed at $s > 0.5$.

F Additional Experimental Results

F.1 Generality across History Budgets

We extend the cross-backend comparison on BFCL Base to history budgets of 128 and 512 tokens. Table 10 uses the same tasks, model checkpoint, and decoding settings as the 256-token study, with the GPU model matched within each pair. Retained history and recovered evidence share the cap.

On BFCL Base, Racer improves task success for all six backends at both budgets, with gains of 1.0–6.0 percentage points at 128 tokens and 2.0–11.0 points at 512 tokens.

Table 11 extends the comparison to BFCL Long Context for all six backends, using 200 paired tasks per backend at each budget. At 512 tokens, all six backends improve, by 0.5–13.0 percentage points, and mean success rises from 15.8% to 19.7%. At 128 tokens, H2O, SnapKV, CommitKV, and C^2KV improve while StreamingLLM and PyramidKV decline; mean success increases from 7.6% to 9.6%. Thus, gains across backends are consistent at 512 tokens on this benchmark but mixed at the tighter budget.

Table 10: **Recovery across KV backends and history budgets on BFCL Base.** Task success (%) ↑ over 200 multi-turn task episodes at 128- and 512-token history budgets. Only uses the listed backend without Racer; Δ is the change in percentage points after adding Racer. Positive gains are shown in **bold**.

	128 tokens			512 tokens		
Backend	Only	+Racer	Δ	Only	+Racer	Δ
CommitKV	12.5	15.0	**+2.5**	29.5	31.5	**+2.0**
StreamingLLM	15.0	17.0	**+2.0**	28.5	33.0	**+4.5**
H2O	9.5	15.5	**+6.0**	28.5	31.5	**+3.0**
SnapKV	11.0	13.0	**+2.0**	28.0	30.0	**+2.0**
PyramidKV	11.5	13.0	**+1.5**	27.5	32.5	**+5.0**
C^2KV	11.0	12.0	**+1.0**	19.0	30.0	**+11.0**

Table 11: **Recovery across KV backends and history budgets on BFCL Long Context.** Task success (%) ↑ over 200 multi-turn task episodes at 128- and 512-token history budgets. Only uses the listed backend without Racer; Δ is the change in percentage points after adding Racer. Positive gains are shown in **bold**.

	128 tokens			512 tokens		
Backend	Only	+Racer	Δ	Only	+Racer	Δ
CommitKV	8.5	9.5	**+1.0**	16.0	16.5	**+0.5**
StreamingLLM	12.0	11.5	-0.5	15.5	19.5	**+4.0**
H2O	7.5	9.5	**+2.0**	17.5	18.0	**+0.5**
SnapKV	6.0	9.0	**+3.0**	21.0	24.0	**+3.0**
PyramidKV	9.0	8.0	-1.0	20.0	22.0	**+2.0**
C^2KV	2.5	10.0	**+7.5**	5.0	18.0	**+13.0**

On ToolSandbox, the gains vary across backends (Table 12). C^2KV has the largest increase at both budgets: mean similarity rises from 0.426 to 0.512 at 128 tokens and from 0.507 to 0.550 at 512 tokens. H2O also improves at both budgets, whereas PyramidKV and CommitKV decrease. StreamingLLM improves at 128 tokens and is unchanged at the displayed precision at 512 tokens; SnapKV decreases at 128 tokens and improves at 512 tokens.

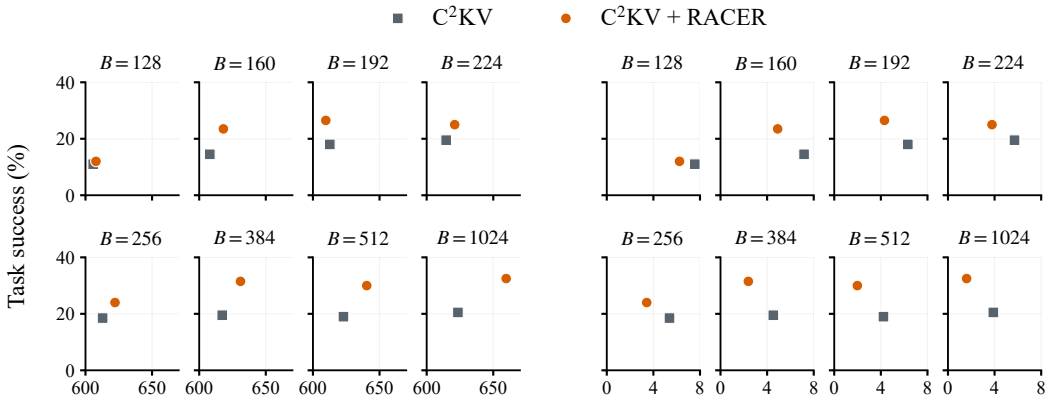
Table 12: **Recovery across KV backends and history budgets on ToolSandbox.** Mean official scenario similarity ↑ over 129 scenarios at 128- and 512-token history budgets. Only uses the listed backend without Racer; Δ is the change in similarity score after adding Racer. Scores and differences are shown to three decimal places. Positive gains are shown in **bold**.

	128 tokens			512 tokens		
Backend	Only	+Racer	Δ	Only	+Racer	Δ
CommitKV	0.520	0.508	-0.012	0.566	0.555	-0.011
StreamingLLM	0.530	0.537	**+0.007**	0.553	0.553	0.000
H2O	0.538	0.548	**+0.010**	0.538	0.552	**+0.014**
SnapKV	0.521	0.517	-0.004	0.510	0.541	**+0.031**
PyramidKV	0.545	0.533	-0.012	0.568	0.545	-0.023
C^2KV	0.426	0.512	**+0.086**	0.507	0.550	**+0.043**

F.2 Budget Sensitivity and Realized KV Usage

We examine how the history capacity changes the task-success–memory trade-off on BFCL Base. We compare C^2KV alone with C^2KV plus Racer at eight history budgets from 128 to 1,024 token-equivalent positions (Appendix B.7), keeping the model, tool definitions, and decoding settings fixed. At each budget, the two conditions use the same tasks and GPU model. Retained history and recovered evidence share the cap, following the matched-capacity comparison used by REPAIRKV (Rusli et al., 2026).

Figure 4 compares task success against peak active actor KV (left) and effective history compression (right). Following the utility–memory view of SideQuest (Kariyappa & Suh, 2026), each panel



(a) Mean task-peak KV (MiB) (b) History compression (×)

Figure 4: **Task success–memory trade-offs on BFCL Base.** Each panel fixes a history budget B on 200 paired tasks. (a) Peak KV, with a zoomed horizontal axis; upper-left is better. (b) History compression; upper-right is better.

fixes one history budget and shows one operating point per method. The compression view isolates history from uncompressed instructions, tool definitions, and live input. Appendix D defines both measurements.

The success gain depends on the history budget. At $B_{\text{hist}} = 128$, adding Racer raises task success from 11.0% to 12.0%. The gain is larger at 160 tokens (14.5% to 23.5%), 384 tokens (19.5% to 31.5%), and 512 tokens (19.0% to 30.0%). At 1,024 tokens, success improves from 20.5% to 32.5%.

Shared budgets yield different realized memory use. At 384 tokens, mean task-peak KV changes from 617.3 to 631.0 MiB, while effective history compression changes from 4.54× to 2.39×. At 192 tokens, higher success (18.0% to 26.5%) coincides with a lower mean task-peak KV (612.7 to 609.6 MiB). The two agents follow their own closed-loop trajectories, and peak KV also includes uncompressed live inputs and generated tokens.

F.3 Comparison with Text-Level Compression across Budgets

We compare budget-constrained HiAgent and ACON with C^{2}KV plus Racer on BFCL Base at history budgets of 128, 256, and 512 tokens (Table 13). The text methods use the adaptations described in Appendix B.4; their cap counts rendered history tokens, while the KV cap counts active historical KV positions, including restored evidence.

Table 13: **BFCL Base success across history budgets.** Task success (%) ↑ over 200 multi-turn task episodes at each history budget in tokens. HiAgent and ACON use text-level compression; C^{2}KV+Racer uses KV-level compression and pre-action recovery.

Method	128	256	512
HiAgent	5.0	21.5	33.5
ACON	2.5	11.0	29.0
C^{2}KV+Racer	12.0	24.0	30.0

The relative ordering depends on the history budget. C^{2}KV plus Racer achieves 12.0% success at 128 tokens, compared with 5.0% for HiAgent and 2.5% for ACON. At 512 tokens, HiAgent achieves 33.5%, followed by C^{2}KV plus Racer at 30.0% and ACON at 29.0%. Thus, the advantage over both text baselines at the tightest budget does not persist at the larger budget.

F.4 Intervention Component Ablations

All variants of a backend use the same GPU model for each task, model checkpoint, greedy decoding, and full tool definitions. Protection selects resident source records or sentences before eviction under the backend's mandatory retention rules. Pre-action interventions comprise event recovery using the fixed detector with one regeneration per decision, and source-verified argument correction when event recovery is not performed. Disabling pre-action interventions commits the first draft directly. Recovered evidence shares the 256-token history capacity.

Event-only evaluation. Event-only evaluation disables argument correction while retaining risk-triggered retrieval and regeneration. Scores use full-system trajectories when no argument was changed and matched rollouts with correction disabled otherwise. For the two task–backend pairs without a valid matched outcome, we report the lower-bound score: observed successes divided by the full task count.

Table 14: **Intervention component combinations.** BFCL Base task success (%) ↑ on 200 multi-turn task episodes per backend at $B_{\text{hist}} = 256$. Event only enables risk-triggered retrieval and regeneration without argument correction. Mean averages the four compressors; dashes denote unevaluated combinations.

Protection	Pre-action intervention	SnapKV	H2O	StreamingLLM	CommitKV	Mean	C^{2}KV
Off	Off	19.5	20.0	24.0	25.0	22.1	18.5
Off	Event only	20.5	21.0	25.5	26.0	23.2	—
Off	Event + correction	21.0	21.0	26.0	26.5	23.6	—
On	Off	22.5	25.5	23.5	25.0	24.1	21.0
On	Event only	22.5	25.0	24.0	26.0	24.4	22.5
On	Event + correction	23.0	25.5	25.0	26.5	25.0	24.0

The contribution depends on the compressor. Protection alone raises mean success from 22.1% to 24.1%. It contributes most on H2O and SnapKV, adding 5.5 and 3.0 percentage points over compression alone. On CommitKV, pre-action interventions add 1.5 points with either protection setting.

Table 15: **Task-level component contributions.** Changes in BFCL Base success (pp) after enabling each component, with newly solved / newly failed tasks in parentheses. $B_{\text{hist}} = 256$, with 200 multi-turn task episodes per backend.

	Add protection		Add pre-action interventions	
Backend	Interventions off	Interventions on	Protection off	Protection on
SnapKV	+3.0 (12/6)	+2.0 (10/6)	+1.5 (4/1)	+0.5 (2/1)
H2O	+5.5 (15/4)	+4.5 (15/6)	+1.0 (3/1)	+0.0 (2/2)
StreamingLLM	-0.5 (6/7)	-1.0 (6/8)	+2.0 (4/0)	+1.5 (4/1)
CommitKV	+0.0 (7/7)	+0.0 (7/7)	+1.5 (4/1)	+1.5 (3/0)
C^{2}KV	+2.5 (17/12)	—	—	+3.0 (9/3)

Learned KV compression. With C^{2}KV, protection uses the native event allocator with $8\times$ history encoding. Task success is 18.5% for compression alone, 21.0% with protection, and 24.0% with protection and pre-action interventions. All conditions use the complete task set, with capacity failures counted as unsuccessful tasks.

F.5 Risk Detection

We evaluate the detector on 36 labeled calibration decisions from nine source task families disjoint from detector training. A positive label denotes failure of the current user turn following the held draft under a no-recovery continuation; the set contains 11 positive and 25 negative labels. Table 16

reports discrimination metrics and precision, recall, and balanced accuracy at the fixed threshold $s > 0.5$. Appendix E describes the detector fitting procedure.

Table 16: **Risk-detector performance** on 36 calibration decisions at threshold $s > 0.5$.

Detector	AUROC	AP	Precision	Recall	Balanced acc.
Linear risk detector	0.709	0.573	0.455	0.455	0.607

F.6 Tool-Definition Allocation Trade-offs

We study the accuracy–compression trade-off of top-k hybrid tool-definition compression. The top-3 hybrid retains the three tool definitions selected by the lexical router in full and compresses the remaining definitions with C^{2}KV. The top-1 variant retains only the highest-ranked definition, whereas the compact variant retains the name, description, and required parameters of the selected top-3 definitions. We also report drop-selected, top-K-only, and random-hybrid controls.

Table 17 summarizes prediction quality and actual tool-KV compression for these runs. The main-table and appendix experiments use different experimental configurations and common evaluation subsets. In particular, the main table includes separate per-tool and full-document settings, whereas the allocation ablations use their own compression configuration and evaluation subset. Consequently, the baseline scores and measured compression ratios are not directly comparable across these experiments. Comparisons should be made within each experimental setting.

Table 17: **Tool-definition allocation ablation.** The hybrid variants use lexical top-k selection; all other tool definitions use C^{2}KV where applicable. Quality metrics are percentages. Comp. is the measured tool-KV compression ratio.

Method	Token-F1 ↑	ROUGE-L ↑	Tool Name ↑	Comp. ↑
Full	53.65	52.95	47.56	1.00×
C^{2}KV hybrid (top-3)	**53.20**	**51.89**	**50.00**	2.64×
C^{2}KV hybrid (compact)	51.43	50.65	47.56	3.33×
Drop-selected	50.57	49.45	41.46	4.90×
Top-k only	49.05	48.20	45.12	6.84×
C^{2}KV hybrid (top-1)	50.12	48.99	42.68	3.37×
C^{2}KV	49.02	48.09	36.59	4.00×
Random hybrid	47.04	46.29	36.59	3.00×

The top-3 hybrid achieves 50.00% tool-name accuracy at 2.64× compression. Reducing selection to top-1 increases compression to 3.37× but lowers tool-name accuracy to 42.68%. The compact top-3 variant reaches 3.33× compression and 47.56% tool-name accuracy. These variants illustrate the trade-off between the size of selected tool definitions and downstream prediction quality. Since their measured compression ratios differ, these results do not establish an advantage at matched KV usage.

F.7 Tool-Definition Routing Strategies

We compare the lexical top-3 router with attention-based alternatives. The compressed-KV attention router ranks definitions by the mean attention from the current query to each compressed tool-KV segment. The full-KV attention router instead scores the complete tool-schema spans. The hybrid reranker starts from the lexical ranking and replaces its last top-3 candidate only when the full-KV attention signal provides sufficient support. Table 18 reports routing hit rate and downstream prediction quality for the reported runs.

Lexical routing obtains a 45.12% top-3 hit rate, compared with 10.98% for compressed-KV attention and 40.24% for full-KV attention. Full-KV reranking increases the hit rate to 51.22%, but its tool-name accuracy remains at 50.00%, matching lexical routing, while Token-F1 decreases slightly from 53.20 to 52.85. Thus, a higher routing hit rate does not necessarily yield higher downstream prediction quality in these runs.

Table 18: **Tool-definition routing ablation.** Router Hit denotes the percentage of decisions whose reference tool appears in the selected top-3. All quality metrics are percentages. Router Hit is not applicable to Full or uniform C^{2}KV, which do not perform top-3 selection.

Method	Token-F1 ↑	ROUGE-L ↑	Tool Name ↑	Router Hit ↑
Full	53.65	52.95	47.56	–
C^{2}KV	49.02	48.09	36.59	–
Lexical hybrid	**53.20**	**51.89**	**50.00**	45.12
Compressed-KV attention	50.67	49.86	47.56	10.98
Full-KV attention	48.78	47.78	46.34	40.24
Lexical + full-KV reranking	52.85	51.69	**50.00**	51.22

G Serving Efficiency

G.1 Setup

This study measures the completed-task throughput, per-task latency, and in-use KV of the C^{2}KV implementation of Racer as more agent tasks run concurrently on a shared serving engine. It uses BFCL Long Context, the BFCL multi-turn category with long tool outputs. We draw 64 of its 200 tasks by stratified sampling: tasks are grouped into 4 quartiles of their maximum prompt length under Full, and 16 tasks are drawn from each quartile with a fixed seed. Every batch submits these tasks in the same order.

We compare Full, which retains the complete history, with C^{2}KV under Racer at $B_{\text{hist}} = 256$, configured as in its row of Table 1. Both use the model checkpoint and decoding settings of Appendix B. Each batch uses a single SGLang engine on one GPU with 48 GB of memory. The engine configuration is shared: bfloat16 weights, FlashInfer attention (Ye et al., 2025b), a static memory fraction of 0.65, a main KV pool of 131,072 tokens (18.0 GiB), a gist pool of 34,491 tokens (4.7 GiB), chunked prefill of 512 tokens, RadixCache with LRU eviction, overlap scheduling, and CUDA-graph decoding. The gist pool is reserved in both runs but used only by Racer, which additionally returns the hidden states read by its detector. The engine admits at most c running requests, where c is the number of concurrent tasks. Each task runs the BFCL harness in a new process; Racer additionally keeps its runtime workers alive across tasks (Appendix C.4).

We evaluate $c \in \{1, 4, 8, 16\}$. For each method and c we run 3 batches of all 64 tasks, each on a freshly started engine, and report the mean and standard deviation over batches. Batches are distributed across two hosts in randomized order, with Full and Racer paired by concurrency and repetition on the same host. Other jobs use the hosts' remaining GPUs. Requests from concurrent tasks are batched together, with scheduling overlapped with computation.

For the context-length study, we use all 200 tasks and the same quartile partition based on maximum prompt length under Full. Each quartile is served separately at $c = 16$, with 50 tasks per batch and 3 batches per method. Both methods use the same task set within each quartile. The four groups contain different tasks; their prompt-length medians label Figure 5.

G.2 Measurements

Time. A task starts when a worker takes it and ends when its harness exits, both on the host monotonic clock. This interval includes task initialization, context preparation, engine queueing and generation, tool execution, and scoring. The batch window runs from the first task start to the last task end, excluding engine start-up. For N completed tasks and a window of T seconds, completed-task throughput is $3600N/T$ tasks per hour. Per-task latency is measured from dispatch to exit. We take its median and 95th percentile within each batch and average them over batches. Waiting for a free worker occurs before dispatch and is excluded from per-task latency.

In-use KV. The in-use occupancy $M(t)$ sums main-pool KV owned by active requests, excluding free slots and evictable RadixCache entries; gist-pool entries pinned by a running request; and registered pending compression K/V. Shared pool slots are counted once. Gist entries that no running request pins are retained for reuse, as Full's evictable prefix cache is; we report this gist cache separately rather than in $M(t)$. The engine updates each component at each scheduler iteration and whenever pending K/V is registered or released, and integrates it by holding each value until the

Table 19: Serving efficiency on BFCL Long Context (64 fixed tasks per batch). Values are mean±SD over 3 completed batches per condition. Metric definitions are in Appendix G.2.

(a) Throughput and latency

c	Method	Tasks/h	p_{50} (s)	p_{95} (s)
1	Full	104.8±0.3	20±0	125±0
1	Racer	113.5±0.7	22±0	89±1
4	Full	274.1±6.9	30±1	140±18
4	Racer	277.6±12.9	32±2	158±35
8	Full	275.9±31.6	45±2	363±59
8	Racer	347.6±8.4	50±1	233±12
16	Full	211.9±11.4	138±24	725±61
16	Racer	353.0±23.5	93±8	348±28

(b) KV occupancy and scheduling

c	Method	In-use mean (GiB)	Sampled peak (GiB)	Gist cache (GiB)	KV-time (GiB s/task)	Running	Queued
1	Full	2.53±0.01	14.62±0.00	–	87±0	0.74±0.00	0.00±0.00
1	Racer	0.93±0.00	5.41±0.00	1.39±0.02	30±0	0.58±0.00	0.00±0.00
4	Full	8.33±0.28	15.89±1.21	–	109±4	2.97±0.09	0.12±0.12
4	Racer	4.29±0.52	13.05±1.63	1.63±0.15	56±7	2.75±0.08	0.05±0.01
8	Full	11.89±0.76	17.60±0.27	–	157±26	4.63±0.00	1.91±0.17
8	Racer	6.72±0.47	16.12±1.33	1.67±0.17	70±3	5.29±0.33	0.19±0.04
16	Full	13.04±0.21	17.52±0.41	–	222±9	6.13±0.37	7.31±0.51
16	Racer	9.87±0.32	17.10±0.11	1.82±0.14	101±9	9.37±0.22	1.22±0.37

Table 20: GPU memory accounting per condition (maximum over batches). Pools are reserved at engine start and are identical for both methods; the gist pool is allocated but unused under Full. Engine memory is the `nvidia-smi` per-process maximum polled every 5 s; no other process used the GPU in any batch (agent workers and tool execution run on the CPU).

c	Method	Main pool (GiB)	Gist pool (GiB)	Torch reserved (GiB)	Engine proc. (GiB)	Device max (GiB)
1	Full	18.00	4.74	33.4	33.9	33.9
1	Racer	18.00	4.74	33.6	34.1	34.1
4	Full	18.00	4.74	33.4	33.9	33.9
4	Racer	18.00	4.74	33.7	34.2	34.2
8	Full	18.00	4.74	33.5	33.9	34.0
8	Racer	18.00	4.74	33.7	34.2	34.2
16	Full	18.00	4.74	33.5	34.0	34.0
16	Racer	18.00	4.74	33.7	34.3	34.3

next update. We report the time-weighted mean $\frac{1}{T}\int M(t)\,dt$ and KV-time per task $\frac{1}{N}\int M(t)\,dt$ over the batch window, and the time-weighted mean of the gist cache. The sampled peak is the largest value of $M(t)$ among the engine's accumulator records, which are written at intervals of at least 0.5 s while the engine is active and at every request completion. The table also reports the time-weighted numbers of running and queued engine requests. Tokens are converted at 147,456 bytes per token. Model weights and activations are excluded from these KV measurements.

GPU memory. Pool sizes are fixed at engine start. We also poll device and per-process memory with `nvidia-smi` every 5 seconds and record the peak reserved PyTorch memory of the engine.

G.3 Results

Table 19 reports completed-task throughput, median and 95th-percentile latency, mean and sampled-peak in-use KV, and engine queueing for every condition of the concurrency sweep. Table 20 reports the reserved pools and total GPU memory alongside these in-use KV measurements.

Context-length groups. Figure 5 adds in-use KV and engine queueing to the per-quartile throughput of Figure 3(d). In the longest quartile, mean in-use KV is similar under the two methods (12.79 versus 12.63 GiB), while the mean engine queue is smaller under Racer (5.09 versus 11.15 requests).

Recovery overhead. The recovery path described in Appendix C.4 regenerates 18–22% of decisions at each concurrency level (Figure 6a). We measure its post-draft wall time from the end of

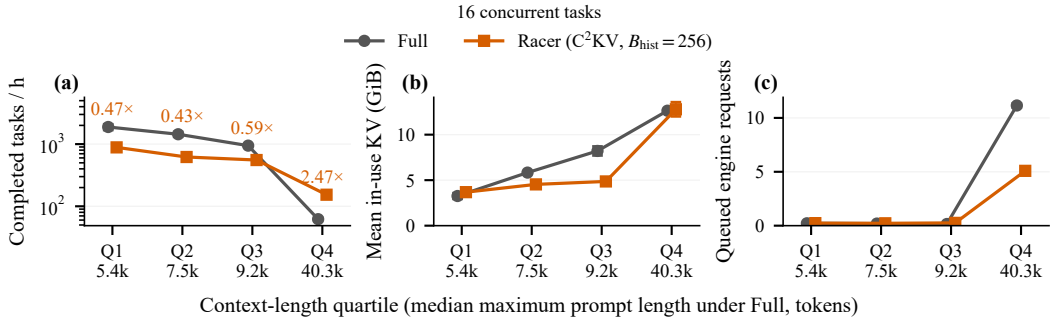


Figure 5: Serving performance across context-length groups at 16 concurrent tasks: (a) completed-task throughput; (b) time-averaged in-use KV; (c) time-averaged queued engine requests. The 200 tasks are divided into 4 quartiles of maximum prompt length under Full and served separately, with 50 tasks per group. Tick labels show group medians. Points are means over 3 batches per method and group; bars show one standard deviation. Panel (a) uses a logarithmic vertical axis, and annotations give the ratio of mean throughput under Racer to Full.

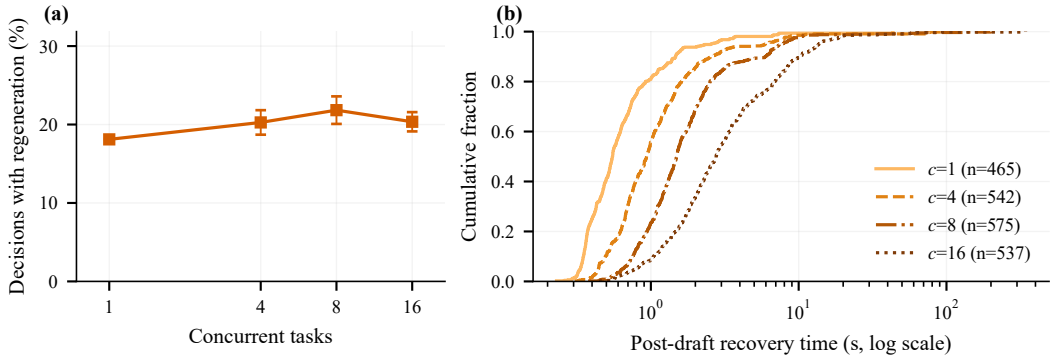


Figure 6: Recovery frequency and post-draft wall time in the serving study. (a) Fraction of decisions that regenerate (mean ± one standard deviation over 3 batches of the same 64 tasks). (b) Empirical cumulative distribution of elapsed time from the discarded draft's end to the regenerated action's end, pooled over the batches at each concurrency level; n is the number of regenerated decisions.

the discarded draft to the end of the regenerated action, including evidence selection, context preparation, engine queueing, and regeneration. Its median rises from 0.6 s with one task to 2.8 s with sixteen (Figure 6b).