CAN AGENT HARNESSES AND INFERENCE ENGINES HEAR EACH OTHER? THE HEAR PROTOCOL FOR AGENTIC LLM SERVING

Anonymous authors
Paper under double-blind review

ABSTRACT

LLM agents increasingly execute complex workflows involving multi-turn reasoning, tool use, and parallel agents. Efficient serving requires decisions that span two layers with complementary information: the agent harness understands workflow dependencies, context lifecycles, and execution objectives, whereas the inference engine observes request queues, KV-cache state, resource pressure, and execution capabilities. Existing interfaces do not systematically connect these views, limiting workflow-aware execution. We propose **HEAR**, a bidirectional **H**arness–**E**ngine p**A**i**R**ing protocol for agentic LLM serving. HEAR standardizes how the harness communicates workflow intent and execution requirements and how the engine returns runtime state, capabilities, and outcomes. By separating protocol semantics from optimization policies, HEAR supports diverse coordination strategies without changing workflow or model semantics. We instantiate HEAR for online cache-aware runtime coordination and workload-aware execution-mode selection for agent roles. Across four conversational and research-agent benchmarks under memory-constrained, concurrent serving, HEAR achieves a **1.61**× batch speedup and reduces median time-to-first-token by **2.23**× on SCBench. Mooncake shows that workflow intent and live engine state provide complementary benefits across load regimes. On BrowseComp-Plus and DeepResearchBench, workload-specific configurations yield **1.23**× and **2.45**× end-to-end speedups, respectively, without observed task-quality degradation. These results establish HEAR as a reusable coordination substrate for efficient agentic LLM serving. Our code is available at `https://anonymous.4open.science/r/hear-3D1E/`.

1 INTRODUCTION

LLM agents tackle complex tasks through multi-round planning, tool use, and collaboration (Yao et al., 2023; Wu et al., 2024). Unlike a single generation, these applications execute workflows of interdependent model calls with evolving contexts. An agent harness orchestrates the workflow, maintains context, and manages tool interactions, whereas an inference engine, such as vLLM (Kwon et al., 2023) or SGLang (Zheng et al., 2024), performs model execution, request batching, and KV-cache management. Thus, end-to-end efficiency depends not only on accelerating individual calls, but also on coordinating requests and reusable state across the workflow.

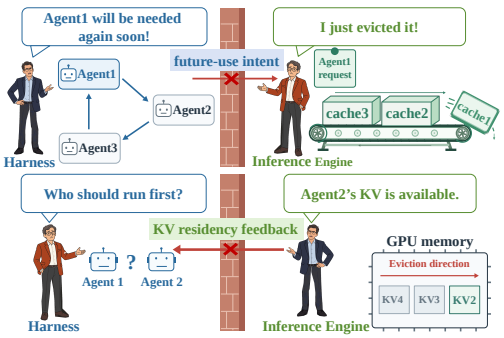

Figure 1: **Motivating example.** Without workflow intent, the engine may evict KV state shortly before reuse; without KV-residency feedback, the harness may schedule a cold request before an equivalent cache-resident one.

This coordination is *difficult* because the information needed for execution decisions is split across the two layers. The harness knows request dependencies, context lifecycles, and application objectives, while the engine observes cache residency, request queues, resource pressure, and supported

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.

execution modes (Pan et al., 2025; Chen et al., 2026a). Neither view is sufficient in isolation. As illustrated in Figure 1, an engine unaware of future context use may evict KV state immediately before it is needed, forcing costly re-prefilling. Conversely, a harness unaware of cache residency may schedule a cold request before a cache-reusable one, even when both are ready and can be legally reordered. The same information gap affects execution configuration: selecting an appropriate inference mode requires matching harness-visible workload characteristics and objectives with engine-side efficiency, capacity, and quality trade-offs. Effective coordination thus requires a bidirectional interface through which both layers can exchange information and carry out joint decisions.

Prior work demonstrates the value of workflow information and runtime feedback for agentic serving (Lin et al., 2024; Pan et al., 2025; Chen et al., 2026a; Yu et al., 2026; Gim et al., 2025; Zhang et al., 2026). These efforts reveal a more fundamental systems gap: **agentic LLM serving lacks a shared control-plane abstraction that connects workflow semantics with runtime execution state.** Consequently, individual optimizations must define their own cross-layer signals, control semantics, and operation lifecycles, tightly coupling policies to particular harnesses and inference engines and limiting their portability and composition.

To address this missing abstraction, we introduce **HEAR**, a bidirectional **Harness-Engine pAiRing protocol** for agentic LLM serving. HEAR associates each exchange with the relevant request, context version, or serving instance and organizes communication into four semantic categories. The harness provides *Execution Description and Intent* and *Execution Requirements and Control*; the engine returns *State and Capabilities* and *Execution Outcomes*. HEAR further defines message force and operation lifecycle by distinguishing description from instruction, preference from requirement, observation from guarantee, and acceptance from completion. Coordination policies determine what information to exchange and what actions to take, whereas HEAR defines what those exchanges mean and how they should be handled. The protocol coordinates how requests from an existing workflow are served without changing the workflow's task-selection logic or the model's semantics.

We demonstrate the generality of HEAR through two complementary policies operating at different timescales. *Cache-Aware Runtime Coordination* is an online policy that combines harness-provided waiting constraints and future-reuse intent with engine-reported KV residency and queue state to coordinate request ordering and cache retention. *Role-specific inference configuration* is a coarser-grained policy that maps agent roles and their input–output and concurrency characteristics to engine-supported inference modes using measured efficiency, capacity, and quality profiles. The two policies use the same interaction contract to answer distinct execution questions: *which ready request should run next* and *how requests from each workflow role should be executed*.

We systematically evaluate HEAR across four conversational and research-agent benchmarks under memory-constrained, concurrent serving. HEAR exposes complementary information that neither layer can exploit alone. On SCBench (Li et al., 2025b), waiting-protected cache-aware scheduling achieves a **1.61**× batch speedup while reducing median and P95 time-to-first-token by **2.23**× and **1.45**×, respectively. Mooncake (Qin et al., 2026) shows that future-reuse intent improves cache retention, but must be conditioned on live engine state to remain beneficial under overload. On BrowseComp-Plus (Chen et al., 2026b) and DeepResearchBench (Du et al., 2026), workload-specific role configurations deliver **1.23**× and **2.45**× end-to-end speedups, respectively, without observed degradation in task quality. Together, these results show that harness intent and engine feedback are complementary, workload-dependent signals for execution coordination.

Our contributions are summarized as follows:

- We introduce **HEAR**, a bidirectional Harness-Engine protocol that associates cross-layer exchanges with requests and contexts, organizes them into four semantic categories, and defines their message force and operation lifecycle.
- We instantiate two **execution-level coordination policies**, online cache-aware runtime coordination and role-specific inference configuration, showing how a common interaction contract supports both runtime scheduling and cache decisions and coarser-grained execution-mode selection.
- Through **controlled evaluations** on SCBench, Mooncake, BrowseComp-Plus, and DeepResearchBench, we show that combining harness intent with engine state improves conversational-serving efficiency and yields **1.23**×**–2.45**× end-to-end speedups for research-agent workflows without observed degradation in task quality.

2 Related Work

Agent Harnesses and Workflow Orchestration. Agent harnesses such as AutoGen (Wu et al., 2024), AgentScope (Gao et al., 2025), LangGraph (The LangChain Team, 2024), and Deep Agents (Chase, 2025) support multi-agent communication, stateful control flow, model invocation, context management, and tool use. LangGraph represents dynamic workflows as stateful graphs, while Deep Agents adds planning and subagent delegation. These frameworks determine *what* agents execute and *when* model calls become ready, but lack direct access to runtime conditions independently managed by inference engines, including request queues, batching, KV-cache residency, and resource pressure. **HEAR** bridges this boundary by conveying workflow intent and requirements to the engine and returning engine state and execution outcomes to the harness.

LLM Inference Systems. LLM inference systems improve request-level efficiency through continuous batching (Yu et al., 2022), chunked prefill (Agrawal et al., 2024), and prefill-decode disaggregation (Zhong et al., 2024). KV-cache optimizations include PagedAttention for flexible allocation and sharing in vLLM (Kwon et al., 2023), RadixAttention for prefix reuse in SGLang (Zheng et al., 2024), asynchronous multi-tier loading and prefetching in LMCache (Liu et al., 2025; vLLM Contributors, n.d.), and selective retention, dynamic context selection, or compression in H_2O, SnapKV, OmniKV, and KIVI (Zhang et al., 2023; Li et al., 2024; Hao et al., 2025; Liu et al., 2024).

These systems provide engine-side mechanisms but do not define how workflow semantics reach the engine or how runtime state and outcomes return to the harness. Rather than replacing their schedulers, cache managers, or execution modes, **HEAR** provides a bidirectional contract for communicating context lifecycles, future reuse, and execution requirements; discovering engine capabilities; requesting supported operations; and observing their outcomes.

Cross-Layer Coordination for Agentic Serving. Recent systems combine application structure with runtime feedback for targeted optimizations. Parrot exposes inter-request dependencies, while Agentix prioritizes calls from program history (Lin et al., 2024; Luo et al., 2026). KVFlow uses workflow structure for retention and prefetching, whereas PBKV predicts dynamic agent invocations (Pan et al., 2025; Zheng et al., 2026). InferCept and Continuum preserve KV state across external-operation pauses (Abhyankar et al., 2024; Li et al., 2025a), while CONCUR controls concurrency using cache pressure (Chen et al., 2026a). Broader systems include Helium's query-plan-based caching and scheduling (Wadlom et al., 2026), Pythia's predictability-driven caching, scheduling, and autoscaling (Yu et al., 2026), Pie's application-defined inferlets for generation and KV management (Gim et al., 2025), and an intermediate policy runtime providing shared identities and observation, scoring, prediction, and action primitives (Zhang et al., 2026). These designs couple cross-layer signals to particular policies or execution substrates.

HEAR instead standardizes the bidirectional *interaction contract* between an existing harness and engine without prescribing policies, relocating application logic, or adding a runtime tier. It associates workflow descriptions and controls with requests and contexts, returns engine state, capabilities, and outcomes, and defines message force and operation lifecycle. Policies thus retain their decision rules while sharing reusable semantics for cross-layer coordination.

3 The HEAR Protocol

HEAR is a bidirectional execution-coordination protocol between an agent harness and an inference engine. It defines what cross-layer messages mean and how they are handled, while leaving individual policies to decide what information to exchange and what actions to take. Figure 2 summarizes the protocol and illustrates how repeated exchanges can coordinate a dynamic workflow as both workflow progress and engine conditions evolve.

3.1 Scope and Execution Model

The harness and engine retain distinct responsibilities: the harness manages workflow semantics, i.e., agent roles, dependencies, readiness, and context lifecycles, while the engine manages model execution, request queues, KV caches, and serving resources. HEAR connects these views to coordinate request ordering and admission, cache preparation and retention, and supported execution configurations. It governs *how* workflow-selected requests are served without changing task selection,

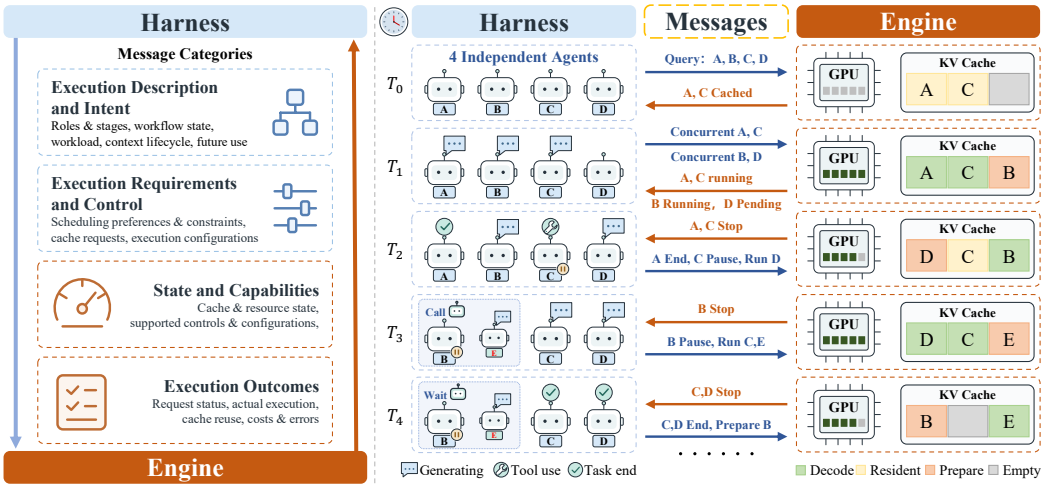

Figure 2: **Overview of HEAR and an illustrative dynamic-agent trace.** *Left:* HEAR organizes bidirectional harness-engine communication into four semantic message categories. *Right:* As the workflow evolves, the harness sends workflow descriptions and execution controls (blue), while the engine returns runtime state and execution outcomes (orange), forming a feedback loop for scheduling and KV-cache management. KV-cache colors denote decoding, resident, preparing, and empty states. The trace illustrates one policy instantiation; HEAR does not prescribe this sequence of interactions.

relaxing dependencies, or modifying model semantics. Workflows may evolve dynamically without being known in advance, and each exchange identifies its associated request, context version, or serving instance, linking workflow requirements to the corresponding engine state and outcomes.

Illustrative Trace. Figure 2 (right) tracks four agents across decision points T_0–T_4. At T_0, the harness queries the current requests of agents A–D, and the engine reports resident KV state for A and C. As agents execute, pause, finish, or create new work, the harness enforces request validity and dependencies, while engine feedback informs admission, ordering, resumption, and KV preparation. This trace illustrates one HEAR-enabled policy rather than a protocol-mandated sequence.

3.2 Message Categories

Cross-layer coordination requires both the conditions motivating a decision and the information or controls needed to carry it out. HEAR organizes these exchanges into four semantic categories, summarized in Figure 2 (left) and Table 1.

Harness-to-Engine Messages. *Execution Description and Intent* conveys execution-relevant workflow facts or predictions, including agent roles, readiness, workload characteristics, context lifecycles,

Table 1: **Bidirectional message categories and representative information.**

Category	Representative Contents
Harness → Engine	
Execution Description and Intent	Agent roles and stages; workflow dependencies and readiness; request sizes and output limits; context lifecycles; known future uses and explicitly marked predictions.
Execution Requirements and Control	Scheduling preferences and waiting-time constraints; cache preparation, retention, and release requests; allowed or requested execution configurations and serving targets.
Engine → Harness	
State and Capabilities	Reusable prefix ranges and residency tiers; allocatable capacity, queue state, and resource pressure; supported controls and configurations; calibrated or estimated execution costs.
Execution Outcomes	Status updates for inference and control requests (accepted, completed, rejected, unsupported, or failed); actual configuration, permitted fallback, and cache reuse; latency, resource use, retries, and errors.

Table 2: **Key semantic distinctions in bidirectional protocol interactions.**

Distinction	Protocol Interpretation
Intent vs. Control	Descriptions expose current or anticipated execution conditions; only explicit control requests ask the receiver to act. *For example, reporting that context c will be reused is distinct from requesting that its KV cache be retained or prepared.*
Preference vs. Requirement	Preferences are best-effort, whereas requirements constrain valid execution. An unsatisfied requirement must be rejected or handled by an explicitly permitted fallback. *For example, a priority hint may affect request ordering, but it cannot override a mandatory protection condition.*
Observation vs. Guarantee	State reports describe observed conditions, and capability reports describe supported operations. Neither reserves resources nor guarantees that a future request will be admitted. *For example, GPU-residency feedback may guide scheduling without reserving the prefix.*
Acceptance vs. Completion	Acceptance means that an operation has been admitted for processing; completion confirms that its stated effect has been realized. Rejected, unsupported, and failed operations remain distinct outcomes. *For example, an accepted cache-preparation request may still be pending.*

and expected reuse. *Execution Requirements and Control* requests engine actions, such as scheduling preferences, waiting-time constraints, cache operations, or execution configurations. The former explains the conditions surrounding a request; the latter asks the engine to act.

Engine-to-Harness Messages. *State and Capabilities* reports the conditions under which decisions must be made, including cache residency, queue state, resource pressure, supported operations, and available configurations. *Execution Outcomes* reports how an inference or control request was handled, including its status, realized configuration, cache reuse, resource cost, or failure. State and Capabilities support decision making, whereas outcomes describe realized effects.

The categories specify semantic roles rather than packet formats. A single exchange may contain information from multiple categories, and an implementation may expose only the categories and fields it supports. Capability reports allow the harness to discover this supported subset instead of assuming that every engine implements every control.

3.3 Interaction Semantics

Message categories identify the purpose of an exchange but do not determine its force or lifecycle. HEAR therefore defines four additional distinctions, summarized in Table 2.

Intent vs. Control. Descriptions and predictions provide decision context; only explicit control requests ask the receiver to act. For example, reporting that a context is likely to be reused does not itself request that the engine retain or prepare its KV state.

Preference vs. Requirement. Preferences are best-effort signals, whereas requirements constrain valid execution. Thus, a cache-aware priority may reorder ready requests but cannot violate workflow dependencies or waiting-time protection; an unsatisfied requirement must be rejected or handled through an explicitly permitted fallback.

Observation vs. Guarantee. State and capability reports describe current conditions and supported operations without reserving resources or guaranteeing future availability. For example, A and C's T_0 residency may guide scheduling but does not reserve their prefixes.

Acceptance vs. Completion. Acceptance indicates that an operation has been admitted for processing, whereas completion confirms that its intended effect has been realized. In Figure 2 (right), requesting preparation of B does not imply that B's KV state is immediately available. The corresponding outcome must distinguish acceptance, completion, rejection, lack of support, and execution failure.

Together, these distinctions prevent policies from treating descriptions as commands, observations as reservations, or accepted operations as completed ones. Later decisions can therefore depend on realized engine effects rather than assumptions about previously issued controls.

3.4 POLICY INTEGRATION

Joint-Information Policy. HEAR separates the protocol from the policies that use it. Orchestration methods such as Conductor and Puppeteer derive decisions from task and agent state (Nielsen et al., 2026; Dang et al., 2025), but application-side information alone does not reveal runtime conditions such as request-level cache residency or queue pressure. HEAR makes these engine-side signals available alongside workflow information, enabling policies to condition execution decisions on both views. We express this joint-information decision as:

$$d_i = \pi(h_i, s_j), \tag{1}$$

where h_i denotes the workflow and request information available at decision point T_i, s_j is the latest relevant engine update, and π selects an execution arrangement or configuration d_i. As the two layers evolve independently, the latest applicable engine update s_j need not have been produced at T_i.

Illustrative Mapping. In the illustrative trace, h_i captures which agents are ready, running, paused, or finished, while s_j captures the latest cache-residency and execution feedback. At T_0, for example, the workflow view identifies agents A–D, whereas the engine view reports resident KV state for A and C. A policy combines these views to choose an execution decision; subsequent outcomes update the information available to later decisions.

A policy may execute at either layer. It can apply a decision locally or communicate it through the controls defined in Section 3.3. Without engine feedback, Equation 1 reduces to the harness-local decision $d_i = \pi(h_i)$; without workflow information, the engine remains limited to request-local state.

Compatibility with Existing Policies. HEAR supports policies that already combine application and engine information. For example, PBKV integrates predicted agent invocations with memory and bandwidth constraints for KV prefetching (Zheng et al., 2026), while Pythia combines workflow progress with queue and capacity state for scheduling (Yu et al., 2026). Their predictors and optimization rules remain unchanged; adapters expose the required workflow descriptions, engine observations, and controls through HEAR. Thus, HEAR provides a common semantic substrate rather than a fixed algorithm. Section 4 demonstrates this separation at two timescales: online cache-aware coordination on SCBench and Mooncake Bench, and role-specific execution configuration on BrowseComp-Plus and DeepResearchBench.

4 EXPERIMENTS

4.1 EXPERIMENTAL SETUP

We evaluate **HEAR** on two execution decisions that combine harness workflow information with engine state. **Cache-aware runtime coordination** uses rapidly changing cache and queue state to control request ordering and KV retention, evaluated on a controlled closed-loop SCBench workload (Li et al., 2025b) and a production-derived Mooncake trace (Qin et al., 2026). **Capability-aware execution configuration** combines workload descriptions with engine capability profiles to select role-specific inference modes, evaluated on BrowseComp-Plus (Chen et al., 2026b) and DeepResearchBench (Du et al., 2026) using benchmark-specific workflows built on a common LangGraph harness. All policies preserve the original task-selection logic and workflow dependencies.

Within each setting, all methods use identical models, inputs, prompts, workflow and tool budgets, and hardware. Policies and method-specific parameters are frozen before evaluation, each arm starts from an independently initialized serving state, and reported results include all failures and retries. We assess end-to-end efficiency and task quality, using request-, token-, and cache-level measurements for attribution. Implementation details and complete results are provided in Appendices A and B.

4.2 CACHE-AWARE RUNTIME COORDINATION

HEAR Instantiation and Runtime Policies. We instantiate Equation 1 as an online coordination policy. At decision point T_i, the harness describes the ready requests $\mathcal{R}_i$, their contexts $\mathcal{C}_i$, and applicable execution requirements $\mathcal{U}_i$, including waiting protection. The engine returns the latest relevant cache availability K_j and queue state Q_j:

$$h_i = (\mathcal{R}_i, \mathcal{C}_i, \mathcal{U}_i), \qquad s_j = (K_j, Q_j). \tag{2}$$

Table 3: **Cache-aware runtime coordination results on SCBench.** All times are in seconds. Batch completion is the elapsed time to complete all users' multi-turn conversations. Gray subscripts report speedups relative to **FCFS**. Reuse denotes the KV-cache reuse rate for follow-up prompt tokens.

Configuration	TTFT ↓			Batch Completion ↓	Reuse ↑
	P50	P95	MAX		
FCFS	$63.1_{1.00\times}$	$74.2_{1.00\times}$	$79.0_{1.00\times}$	$481_{1.00\times}$	19.2
Cache-Aware	$4.6_{13.6\times}$	$75.8_{0.98\times}$	$93.9_{0.84\times}$	$\mathbf{224}_{\mathbf{2.15\times}}$	**87.6**
Cache-Aware + Guard-40	$28.3_{2.23\times}$	$\mathbf{51.1}_{\mathbf{1.45\times}}$	$\mathbf{65.0}_{\mathbf{1.22\times}}$	$299_{1.61\times}$	66.1
Cache-Aware + Guard-60	$\mathbf{4.5}_{\mathbf{13.9\times}}$	$64.0_{1.16\times}$	$80.4_{0.98\times}$	$230_{2.09\times}$	84.7

The index j allows engine feedback to update asynchronously from harness decision point i. The scheduling policy produces:

$$d_i = \pi_{\text{sched}}(h_i, s_j) = (\boldsymbol{p}_i, \mathcal{U}_i), \tag{3}$$

where $\boldsymbol{p}_i$ assigns priorities to ready requests. Under HEAR, cache and queue reports provide *State and Capabilities*; priorities are scheduling preferences under *Execution Requirements and Control*; and waiting protection is a mandatory requirement. The engine applies each preference subject to these requirements and returns the realized execution outcome.

We compare three execution-order policies. **FCFS** uses the default engine scheduler (Kwon et al., 2023). Inspired by KVFlow (Pan et al., 2025), **Cache-Aware** converts engine-reported KV residency into harness-side scheduling preferences; **Cache-Aware+Guard-**W additionally prevents bypass or preemption after a request waits W seconds. We evaluate $W \in \{40, 60\}$ on SCBench and $W = 40$ on Mooncake. For Mooncake, we also evaluate **Session-Aware**, a KV-retention policy inspired by prediction-guided methods (Zheng et al., 2026; Yu et al., 2026). After each request, the harness estimates session continuation from its depth and prior inter-turn intervals, explicitly marking the estimate as descriptive intent rather than a retention requirement. The engine combines this estimate with access recency, retaining recent contexts with higher continuation likelihood while gradually deprioritizing long-idle ones. We evaluate **Session-Aware** alone and with **Cache-Aware+Guard**. The appendix provides implementation details and policy parameters.

Workloads. SCBench (Li et al., 2025b) lacks request timestamps, so we construct a closed-loop multi-turn workload with Poisson session arrivals and sampled inter-turn think times, serving Qwen3-8B on one RTX PRO 6000 with a 96-GiB L2 cache. The production-derived Mooncake trace (Qin et al., 2026) records request timestamps, input/output lengths, and remapped block hashes, preserving real arrivals and inter-turn gaps while exposing intra- and inter-session prefix sharing. We replay 50%, 75%, and 100% of sessions beginning within a 15-minute window, preserving their start times and complete turn sequences. We serve Qwen3-8B on one H100 with a 96-GiB L2 cache.

SCBench Results. Table 3 shows trade-off among cache reuse, responsiveness, and batch completion. Compared with **FCFS**, which achieves only 19.2% reuse due to prefix recomputation, **Cache-Aware** uses engine residency feedback to raise reuse to 87.6%, reduce median TTFT from 63.1 to 4.6 seconds, and shorten batch completion from 481 to 224 seconds. Yet, repeatedly deferring cold requests increases maximum TTFT. Waiting protection bounds this behavior. **Cache-Aware+Guard-40** improves P50, P95, and maximum TTFT over **FCFS** while retaining a 1.61× batch speedup. Increasing the threshold to 60 seconds favors reuse and batch completion but weakens tail-latency protection. These results exercise both directions of HEAR: engine feedback identifies reuse opportunities, while harness requirements constrain how aggressively the scheduler exploits them.

Mooncake Results. Figure 3 and Table 4 show that the preferred policy varies with load while the HEAR interface remains unchanged. At 50% load, limited queuing makes idle-period eviction the primary source of reuse loss; **Session-Aware** therefore outperforms execution-order control, raising reuse from 19.5% to 31.9%. At 75%, high concurrency and remaining reuse opportunities make eviction- and execution-order control complementary: combining **Session-Aware** with **Cache-Aware + Guard** yields the shortest session time, highest throughput, and highest reuse. At 100%, **Cache-Aware** achieves better P50, session latency, and throughput, whereas **Session-Aware** improves tail latency and reuse. Appendix B.1 provides execution timelines. Thus, no policy dominates across loads; HEAR enables workload-dependent use of workflow intent and engine feedback through unchanged message categories and interaction semantics.

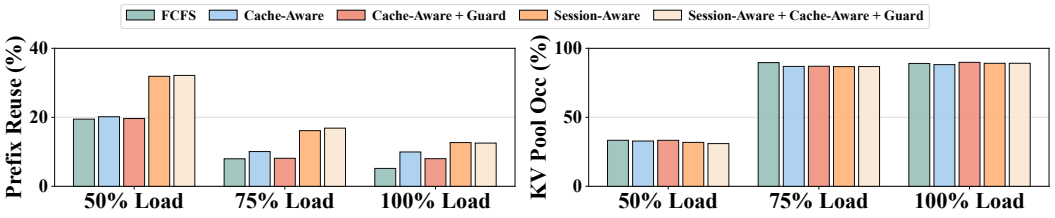

Figure 3: **KV reuse and GPU KV occupancy on Mooncake Bench under different load levels.**

Table 4: **Cache-aware runtime coordination results on Mooncake Bench.** Session is the total response-waiting time per multi-turn conversation; throughput is measured in turns per minute.

Load	Configuration	TTFT ↓			Session ↓	Throughput ↑	Reuse ↑
		P50	P95	MAX			
50%	**FCFS**	0.7 ×1.00	2.8 ×1.00	6.2 ×1.00	13.7 ×1.00	86.1 ×1.00	19.5
	Cache-Aware	0.6 ×1.21	2.4 ×1.17	**5.9 ×1.06**	13.3 ×1.02	86.0 ×1.00	20.2
	Cache-Aware + Guard	0.6 ×1.10	2.8 ×1.01	7.4 ×0.84	13.6 ×1.01	86.2 ×1.00	19.6
	Session-Aware	0.6 ×1.16	2.5 ×1.14	6.3 ×0.99	13.0 ×1.05	86.6 ×1.01	31.9
	Session-Aware + Cache-Aware + Guard	**0.5 ×1.35**	**2.3 ×1.21**	6.1 ×1.02	**12.7 ×1.08**	**86.8 ×1.01**	**32.1**
75%	**FCFS**	10.9 ×1.00	51.9 ×1.00	56.7 ×1.00	50.2 ×1.00	116.4 ×1.00	8.0
	Cache-Aware	8.7 ×1.26	42.9 ×1.21	49.6 ×1.14	42.6 ×1.18	119.2 ×1.02	10.1
	Cache-Aware + Guard	9.1 ×1.20	42.3 ×1.23	47.9 ×1.18	42.5 ×1.18	119.3 ×1.03	8.1
	Session-Aware	10.5 ×1.04	**39.0 ×1.33**	**45.1 ×1.26**	43.2 ×1.16	119.5 ×1.03	16.1
	Session-Aware + Cache-Aware + Guard	**8.2 ×1.33**	39.8 ×1.30	45.4 ×1.25	**39.8 ×1.26**	**121.0 ×1.04**	**16.9**
100%	**FCFS**	163.9 ×1.00	277.0 ×1.00	281.5 ×1.00	256.9 ×1.00	122.7 ×1.00	5.2
	Cache-Aware	**86.8 ×1.89**	209.9 ×1.32	220.9 ×1.27	**156.1 ×1.65**	**128.3 ×1.05**	10.0
	Cache-Aware + Guard	97.1 ×1.69	209.2 ×1.32	217.6 ×1.29	178.7 ×1.44	122.8 ×1.00	8.0
	Session-Aware	93.0 ×1.76	**199.2 ×1.39**	**207.8 ×1.35**	171.8 ×1.50	124.6 ×1.02	**12.7**
	Session-Aware + Cache-Aware + Guard	161.7 ×1.01	267.4 ×1.04	277.7 ×1.01	243.2 ×1.06	108.9 ×0.89	12.5

4.3 Capability-Aware Execution Configuration

HEAR Instantiation and Candidate Space. We instantiate Equation 1 as an engine-side configuration selector. The harness description h_i specifies the agent role, workflow stage, workload shape, execution objective, and quality requirements. The engine view s_j provides compatible execution modes and their measured latency, throughput, KV-capacity, and stability profiles. These inputs correspond to HEAR's *Execution Description and Intent*, *Execution Requirements and Control*, and *State and Capabilities*. We consider the shared candidate space:

$$\mathcal{C} = \{\text{vLLM}, \text{OmniKV}\} \times \{\text{vLLM}, \text{H}_2\text{O}, \text{SnapKV}\}, \tag{4}$$

where the first and second sets denote the Main-side and Reader/sub-agent-side modes. Among configurations passing development-stage quality and stability checks, the selector minimizes makespan:

$$d_i^{\text{config}} = \pi_{\text{E}}(h_i, s_j), \qquad d^*_{\text{BCP}} = (\text{vLLM}, \text{H}_2\text{O}), \qquad d^*_{\text{DRB}} = (\text{OmniKV}, \text{H}_2\text{O}). \tag{5}$$

The mappings and method-specific parameters are verified and frozen before formal evaluation, and are loaded by the Engine at workflow initialization.

Workloads and Evaluation Design. We evaluate all six configurations on 208 BrowseComp-Plus questions (Chen et al., 2026b) and 100 DeepResearchBench instances (Du et al., 2026) per arm, using identical within-benchmark budgets and independently initialized serving states. Both retain their native orchestration and retrieval environments within a common LangGraph-based Main–Retriever–Reader/sub-agent harness, with GLM-4.7-Flash serving Main and Reader/sub-agent inference on separate H100 GPUs. Appendices A and B detail selector calibration, execution-mode parameters, per-arm results, and quality and stability analyses.

End-to-End Results. Table 5 reports end-to-end time, P95 latency, and task quality for all six configurations. Main entries are measured values, subscripts give ratios relative to vLLM–vLLM, and bold entries denote the frozen selections. On BrowseComp-Plus, vLLM–H_2O reduces wall time from 5.69 to 4.62 hours, yielding a 1.23× speedup while preserving Judge accuracy (46.63% vs. 46.15%). On DeepResearchBench, OmniKV–H_2O reduces makespan from 5.93 to 2.42 hours, yielding a 2.45× speedup while preserving RACE (41.1% vs. 40.6%). Aggregate efficiency and

Table 5: **Capability-aware execution configuration results.**

Main	Reader	Browscomp-plus			Deep-research-bench		
		Wall (h) ↓	P95 E2E (min) ↓	Acc. (%) ↑	Makesp (h) ↓	P95 E2E (min) ↓	RACE (%) ↑
vLLM	**vLLM**	$5.69_{1.00\times}$	$40.0_{1.00\times}$	46.15	$5.93_{1.00\times}$	$39.8_{1.00\times}$	40.6
OmniKV	**vLLM**	$5.37_{1.06\times}$	$40.3_{0.99\times}$	47.12	$6.17_{0.96\times}$	$39.9_{1.00\times}$	41.1
vLLM	**H_2O**	$\mathbf{4.62_{1.23\times}}$	$39.6_{1.01\times}$	46.63	$2.76_{2.15\times}$	$15.6_{2.55\times}$	40.5
OmniKV	**H_2O**	$4.98_{1.14\times}$	$37.2_{1.08\times}$	43.27	$\mathbf{2.42_{2.45\times}}$	$15.8_{2.52\times}$	41.1
vLLM	**SnapKV**	$4.83_{1.18\times}$	$40.4_{0.99\times}$	44.23	$2.85_{2.08\times}$	$17.7_{2.25\times}$	40.0
OmniKV	**SnapKV**	$5.29_{1.07\times}$	$40.6_{0.99\times}$	43.75	$2.76_{2.15\times}$	$18.3_{2.17\times}$	40.5

Table 6: **Role-level execution attribution.**

Benchmark	Main Output Tok./Req.	Main Latency Alt. → Selected	Reader/Sub Latency Dense → Selected	Cache Evidence
BrowseComp-Plus	∼ 0.4K	OmniKV → vLLM 17.0 → 14.9 s $_{1.14\times}$	vLLM → H_2O 53.4 → 9.5 s $_{5.60\times}$	Prefill −76.2% Reuse 6.2% → 80.3%
DeepResearchBench	∼ 4.4K	vLLM → OmniKV 69.4 → 64.2 s $_{1.08\times}$	vLLM → H_2O 421.5 → 58.4 s $_{7.22\times}$	Chain reuse 49.8%

P95 latency occasionally favor different mappings, confirming that selection must account for the execution objective communicated in h_i. Overall, both selected mappings improve end-to-end efficiency without an observed degradation in task quality.

Role-Level Attribution. Table 6 compares configurations that change one role's mode while holding the other fixed. Reader/sub-agent execution provides the largest consistent gain. Selecting H_2O reduces mean latency by 5.60× on BrowseComp-Plus and 7.22× on DeepResearchBench. On BrowseComp-Plus, this gain accompanies a 76.2% reduction in prefilled tokens and an increase in cache reuse from 6.2% to 80.3%; on DeepResearchBench, it accompanies 49.8% chain reuse. The preferred Main-side mode is workload-dependent. BrowseComp-Plus Main requests generate approximately 0.4K output tokens, and vLLM is $1.14\times$ faster than OmniKV. DeepResearchBench Main requests generate approximately 4.4K tokens in a round-synchronized workflow, and OmniKV is $1.08\times$ faster than vLLM. The longer decoding workload better amortizes mode-specific overhead.

Fixed-Mapping Cost. Neither selected mapping is optimal for both workloads. Applying OmniKV–H_2O universally would increase BrowseComp-Plus wall time by 7.8%, whereas applying vLLM–H_2O universally would increase DeepResearchBench makespan by 14.0%. Additional throughput–capacity and makespan–tail-latency trade-offs are reported in Appendix B.

4.4 Cross-Policy Takeaway

The two policy families instantiate **HEAR** at different timescales: cache-aware coordination combines dynamic cache and queue state with harness intent and requirements to control request ordering and KV retention, while capability-aware configuration combines stable engine profiles with workload descriptions and objectives to select inference modes. Both follow Equation 1, since neither layer alone has sufficient information. Across conversational and research-agent workloads, decisions adapt to the workload while the HEAR interface and workflow semantics remain unchanged, demonstrating a common bidirectional contract for workload-specific policies.

5 Conclusions

We introduce **HEAR**, a bidirectional Harness–Engine protocol that connects workflow intent and requirements with engine state, capabilities, and outcomes while preserving workflow and model semantics. We instantiate HEAR for cache-aware runtime coordination and role-specific inference configuration. On SCBench, waiting-protected cache-aware scheduling achieves a **1.61**× batch speedup while improving median and tail TTFT. Mooncake further shows that future-use-aware retention must account for live queue state under overload. On BrowseComp-Plus and DeepResearchBench, workload-specific role configurations yield **1.23**× and **2.45**× end-to-end speedups, respectively, without observed degradation in task quality. These results establish HEAR as a reusable, policy-independent substrate for workload-adaptive coordination across the Harness-Engine boundary, with its benefits determined by the information and controls exposed by the serving stack.

AI USE STATEMENT

We used generative AI tools to assist with refining the conceptual framework and experimental methodology, modifying and debugging code, interpreting experimental results, revising parts of the manuscript, Chinese–English translation, literature review, LaTeX formatting, and the preparation of scientific figures. The research questions, system design, experimental decisions, and conclusions were determined by the authors. Generative AI was not used to generate datasets or experimental results. All AI-assisted content, technical claims, figures, and citations were manually reviewed and verified by the authors. The authors take full responsibility for the final content of this work.

ETHICS STATEMENT

This work involves no human-subject experiments or collection of new personal or sensitive data. Since the proposed protocol exchanges execution and resource metadata, practical deployments should enforce access control, tenant isolation, and bounded waiting. Our method improves inference efficiency but does not address the safety or reliability of the underlying model outputs.

REPRODUCIBILITY STATEMENT

The main text specifies the proposed protocol, policy instantiations, workloads, baselines, evaluation metrics, and experimental settings. Appendix A provides the implementation details and execution-mode configurations needed to reproduce the reported experiments.

REFERENCES

Reyna Abhyankar, Zijian He, Vikranth Srivatsa, Hao Zhang, and Yiying Zhang. InferCept: Efficient intercept support for augmented large language model inference. In Ruslan Salakhutdinov, Zico Kolter, Katherine Heller, Adrian Weller, Nuria Oliver, Jonathan Scarlett, and Felix Berkenkamp (eds.), *Proceedings of the 41st International Conference on Machine Learning*, volume 235 of *Proceedings of Machine Learning Research*, pp. 81–95. PMLR, 21–27 Jul 2024. URL https://proceedings.mlr.press/v235/abhyankar24a.html.

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, Santa Clara, CA, July 2024. USENIX Association. ISBN 978-1-939133-40-3. URL https://www.usenix.org/conference/osdi24/presentation/agrawal.

Harrison Chase. Deep Agents. LangChain Blog, July 2025. URL https://www.langchain.com/blog/deep-agents. Accessed: 2026-09-24.

Qiaoling Chen, Zhisheng Ye, Tian Tang, Peng Sun, Boyu Tian, Guoteng Wang, Shenggui Li, Yonggang Wen, Zhenhua Han, and Tianwei Zhang. Concur: High-throughput agentic batch inference of llm via congestion-based concurrency control. *arXiv preprint arXiv:2601.22705*, 2026a. URL https://arxiv.org/abs/2601.22705.

Zijian Chen, Xueguang Ma, Shengyao Zhuang, Ping Nie, Kai Zou, Sahel Sharifymoghaddam, Andrew Liu, Joshua Green, Kshama Patel, Ruoxi Meng, Mingyi Su, Yanxi Li, Haoran Hong, Xinyu Shi, Xuye Liu, Hosna Oyarhoseini, Nandan Thakur, Crystina Zhang, Luyu Gao, Wenhu Chen, and Jimmy Lin. BrowseComp-Plus: A fair and disentangled evaluation benchmark for deep search agents. In *Proceedings of the 64th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pp. 22349–22370, 2026b. doi: 10.18653/v1/2026.acl-long.1023. URL https://aclanthology.org/2026.acl-long.1023/.

Yufan Dang, Chen Qian, Xueheng Luo, Jingru Fan, Zihao Xie, Ruijie Shi, Weize Chen, Cheng Yang, Xiaoyin Che, Ye Tian, Xuantang Xiong, Lei Han, Zhiyuan Liu, and Maosong Sun. Multi-agent collaboration via evolving orchestration. In *Advances in Neural Information Processing Systems*, volume 38, 2025. URL https://proceedings.neurips.cc/paper_files/paper/2025/file/f1320d2e2842169c6fc89dcbd80e94d0-Paper-Conference.pdf.

Mingxuan Du, Benfeng Xu, Chiwei Zhu, Licheng Zhang, Xiaorui Wang, and Zhendong Mao. Deepresearch bench: A comprehensive benchmark for deep research agents. In C. Vondrick, B. Hariharan, C. Raffel, L. Pinto, D. Yang, and A. Faust (eds.), *International Conference on Learning Representations*, volume 2026, pp. 42414–42448, 2026. URL https://proceedings.iclr.cc/paper_files/paper/2026/file/465f22be10e07b301c6ed58f0472f704-Paper-Conference.pdf.

Dawei Gao, Zitao Li, Yuexiang Xie, Weirui Kuang, Liuyi Yao, Bingchen Qian, Zhijian Ma, Yue Cui, Haohao Luo, Shen Li, et al. Agentscope 1.0: A developer-centric framework for building agentic applications. *arXiv preprint arXiv:2508.16279*, 2025.

In Gim, Zhiyao Ma, Seung-seob Lee, and Lin Zhong. Pie: A programmable serving system for emerging llm applications. In *Proceedings of the ACM SIGOPS 31st Symposium on Operating Systems Principles*, pp. 415–430, 2025. URL https://doi.org/10.1145/3731569.3764814.

Jitai Hao, Yuke Zhu, Tian Wang, Jun Yu, Xin Xin, Bo Zheng, Zhaochun Ren, and Sheng Guo. Omnikv: Dynamic context selection for efficient long-context llms. In *International Conference on Learning Representations*, volume 2025, pp. 87443–87464, 2025.

Woosuk Kwon, Zhuohan Li, Siyuan Zhuang, Ying Sheng, Lianmin Zheng, Cody Hao Yu, Joseph E. 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. Association for Computing Machinery, 2023. doi: 10.1145/3600006.3613165. URL https://doi.org/10.1145/3600006.3613165.

Hanchen Li, Runyuan He, Qiuyang Mang, Qizheng Zhang, Huanzhi Mao, Xiaokun Chen, Hangrui Zhou, Alvin Cheung, Joseph Gonzalez, and Ion Stoica. Continuum: Efficient and robust multi-turn llm agent scheduling with kv cache time-to-live. *arXiv preprint arXiv:2511.02230*, 2025a. URL https://arxiv.org/abs/2511.02230v6.

Yucheng Li, Huiqiang Jiang, Qianhui Wu, Xufang Luo, Surin Ahn, Chengruidong Zhang, Amir Abdi, Dongsheng Li, Jianfeng Gao, Yuqing Yang, and Lili Qiu. SCBench: A KV cache-centric analysis of long-context methods. In Y. Yue, A. Garg, N. Peng, F. Sha, and R. Yu (eds.), *International Conference on Learning Representations*, volume 2025, pp. 66063–66093, 2025b. URL https://proceedings.iclr.cc/paper_files/paper/2025/file/a540b17fb2295c736d5afd6c507acf66-Paper-Conference.pdf.

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. In A. Globerson, L. Mackey, D. Belgrave, A. Fan, U. Paquet, J. Tomczak, and C. Zhang (eds.), *Advances in Neural Information Processing Systems*, volume 37, pp. 22947–22970. Curran Associates, Inc., 2024. doi: 10.52202/079017-0722. URL https://proceedings.neurips.cc/paper_files/paper/2024/file/28ab418242603e0f7323e54185d19bde-Paper-Conference.pdf.

Chaofan Lin, Zhenhua Han, Chengruidong Zhang, Yuqing Yang, Fan Yang, Chen Chen, and Lili Qiu. Parrot: Efficient serving of LLM-based applications with semantic variable. In *18th USENIX Symposium on Operating Systems Design and Implementation (OSDI 24)*, pp. 929–945, Santa Clara, CA, July 2024. USENIX Association. ISBN 978-1-939133-40-3. URL https://www.usenix.org/conference/osdi24/presentation/lin-chaofan.

Yuhan Liu, Yihua Cheng, Jiayi Yao, Yuwei An, Xiaokun Chen, Shaoting Feng, Yuyang Huang, Samuel Shen, Rui Zhang, Kuntai Du, et al. Lmcache: An efficient kv cache layer for enterprise-scale llm inference. *arXiv preprint arXiv:2510.09665*, 2025.

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. In Ruslan Salakhutdinov, Zico Kolter, Katherine Heller, Adrian Weller, Nuria Oliver, Jonathan Scarlett, and Felix Berkenkamp (eds.), *Proceedings of the 41st International Conference on Machine Learning*, volume 235 of *Proceedings of Machine Learning Research*, pp. 32332–32344. PMLR, 21–27 Jul 2024. URL https://proceedings.mlr.press/v235/liu24bz.html.

Michael Luo, Xiaoxiang Shi, Colin Cai, Tianjun Zhang, Justin Wong, Yichuan Wang, Chi Wang, Yanping Huang, Zhifeng Chen, Joseph E. Gonzalez, and Ion Stoica. Agentix: An efficient serving engine for LLM agents as general programs. In *23rd USENIX Symposium on Networked Systems Design and Implementation (NSDI 26)*, pp. 2443–2459, Renton, WA, May 2026. USENIX Association. ISBN 978-1-939133-54-0. URL https://www.usenix.org/conference/nsdi26/presentation/luo.

Stefan Nielsen, Edoardo Cetin, Peter Schwendeman, Qi Sun, Jinglue Xu, and Yujin Tang. Learning to orchestrate agents in natural language with the conductor. In *International Conference on Learning Representations*, volume 2026, pp. 135686–135724, 2026. URL https://proceedings.iclr.cc/paper_files/paper/2026/file/dbb8193ad7e6fcbc7bb62ed9ee835110-Paper-Conference.pdf.

Zaifeng Pan, AJJKUMAR DAHYALAL PATEL, Yipeng Shen, Zhengding Hu, Yue Guan, Wan-Lu Li, Lianhui Qin, Yida Wang, and Yufei Ding. Kvflow: Efficient prefix caching for accelerating llm-based multi-agent workflows. In D. Belgrave, C. Zhang, H. Lin, R. Pascanu, P. Koniusz, M. Ghassemi, and N. Chen (eds.), *Advances in Neural Information Processing Systems*, volume 38, Main Conference, pp. 126246–126265. Curran Associates, Inc., 2025. doi: 10.52202/085713-4208. URL https://proceedings.neurips.cc/paper_files/paper/2025/file/b7971d31a7d5eb0f1eed2f8f6f368195-Paper-Conference.pdf.

Ruoyu Qin, Zheming Li, Weiran He, Jialei Cui, Heyi Tang, Feng Ren, Teng Ma, Shangming Cai, Yineng Zhang, Mingxing Zhang, et al. Mooncake: A kvcache-centric disaggregated architecture for llm serving. *ACM Transactions on Storage*, 22(4):1–38, 2026. URL https://doi.org/10.1145/3773772.

The LangChain Team. LangGraph. LangChain Blog, January 2024. URL https://www.langchain.com/blog/langgraph. Accessed: 2026-09-24.

vLLM Contributors. vLLM: KV offloading usage guide. Official documentation, n.d. URL https://docs.vllm.ai/en/latest/features/kv_offloading_usage/. Accessed: 2026-09-24.

Noppanat Wadlom, Junyi Shen, and Yao Lu. Efficient llm serving for agentic workflows: A data systems perspective. *Proceedings of the ACM on Management of Data*, 4(3 (SIGMOD):1–29, 2026. URL https://doi.org/10.1145/3802046.

Qingyun Wu, Gagan Bansal, Jieyu Zhang, Yiran Wu, Beibin Li, Erkang Zhu, Li Jiang, Xiaoyun Zhang, Shaokun Zhang, Jiale Liu, Ahmed Hassan Awadallah, Ryen W White, Doug Burger, and Chi Wang. Autogen: Enabling next-gen LLM applications via multi-agent conversations. In *First Conference on Language Modeling*, 2024. URL https://openreview.net/forum?id=BAakY1hNKS.

Shunyu Yao, Jeffrey Zhao, Dian Yu, Nan Du, Izhak Shafran, Karthik R Narasimhan, and Yuan Cao. React: Synergizing reasoning and acting in language models. In *The Eleventh International Conference on Learning Representations*, 2023. URL https://openreview.net/forum?id=WE_vluYUL-X.

Gyeong-In Yu, Joo Seong Jeong, Geon-Woo Kim, Soojeong Kim, and Byung-Gon Chun. Orca: A distributed serving system for {Transformer-Based} generative models. In *16th USENIX symposium on operating systems design and implementation (OSDI 22)*, pp. 521–538, 2022.

Shan Yu, Junyi Shu, Yuanjiang Ni, Kun Qian, Xue Li, Yang Wang, Jinyuan Zhang, Ziyi Xu, Shuo Yang, Lingjun Zhu, et al. Pythia: Exploiting workflow predictability for efficient agent-native llm serving. *arXiv preprint arXiv:2604.25899*, 2026. URL https://arxiv.org/abs/2604.25899.

Rui Zhang, Chaeeun Kim, Cheng-Wei Ching, Yi Zhong, and Liting Hu. A policy-driven runtime layer for agentic llm serving. In *Proceedings of the 17th ACM SIGOPS Asia-Pacific Workshop on Systems*, pp. 31–37, 2026. URL https://doi.org/10.1145/3838177.3841725.

Zhenyu Zhang, Ying Sheng, Tianyi Zhou, Tianlong Chen, Lianmin Zheng, Ruisi Cai, Zhao Song, Yuandong Tian, Christopher Ré, Clark Barrett, Zhangyang "Atlas" Wang, and Beidi Chen. H2o: Heavy-hitter oracle for efficient generative inference of large language models. In A. Oh, T. Naumann, A. Globerson, K. Saenko, M. Hardt, and S. Levine (eds.), *Advances in Neural Information Processing Systems*, volume 36, pp. 34661–34710. Curran Associates, Inc., 2023. doi: 10.52202/075280-1506. URL https://proceedings.neurips.cc/paper_files/paper/2023/file/6ceefa7b15572587b78ecfcebb2827f8-Paper-Conference.pdf.

Haoyu Zheng, Fangcheng Fu, Jia Wu, Binhang Yuan, Yongqiang Zhang, Hao Wang, Yuanyuan Zhu, Xiao Yan, and Jiawei Jiang. Efficient serving for dynamic agent workflows with prediction-based kv-cache management. *arXiv preprint arXiv:2605.06472*, 2026.

Lianmin Zheng, Liangsheng Yin, Zhiqiang Xie, Chuyue Sun, Jeff Huang, Cody Hao Yu, Shiyi Cao, Christos Kozyrakis, Ion Stoica, Joseph E. Gonzalez, Clark Barrett, and Ying Sheng. Sglang: Efficient execution of structured language model programs. In A. Globerson, L. Mackey, D. Belgrave, A. Fan, U. Paquet, J. Tomczak, and C. Zhang (eds.), *Advances in Neural Information Processing Systems*, volume 37, pp. 62557–62583. Curran Associates, Inc., 2024. doi: 10.52202/079017-2000. URL https://proceedings.neurips.cc/paper_files/paper/2024/file/724be4472168f31ba1c9ac630f15dec8-Paper-Conference.pdf.

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, Santa Clara, CA, July 2024. USENIX Association. ISBN 978-1-939133-40-3. URL https://www.usenix.org/conference/osdi24/presentation/zhong-yinmin.

A Implementation Details

A.1 Protocol and Selector Implementation

Selector Inputs. For capability-aware configuration, the Harness instantiates h_i with the agent role, workflow stage, observed input and output workload shape, concurrency and synchronization pattern, execution objective, and applicable quality or feasibility requirements. The Engine instantiates s_j with the execution modes supported for that role and their development profiles, including request latency, throughput, KV-cache reuse and capacity behavior, recoverable capacity retries, and failure or fallback status. Each profile is associated with a concrete Engine configuration and source revision.

Development Calibration and Verification. A calibration script first filters configurations that are incompatible with the role or model, have incomplete or invalid runs, or fail the development quality and stability checks. Let $\mathcal{C}_{\text{valid}}$ denote the remaining candidates. The script ranks them by measured development makespan:

$$\widetilde{d} = \arg\min_{d \in \mathcal{C}_{\text{valid}}} T_{\text{dev}}(d), \tag{6}$$

where $\widetilde{d}$ is the programmatically recommended mapping. The recommendation is then verified against the execution logs and the quality and stability summaries. The verifier may select another candidate from $\mathcal{C}_{\text{valid}}$ only when these checks expose an invalid measurement, an unsupported execution path, or a feasibility or reliability issue not captured by makespan. The accepted mapping, denoted by d^*, is serialized before formal evaluation. Formal-evaluation results are not used during ranking or verification.

Freezing and Runtime Application.. The accepted role-to-mode mapping, method-specific parameters, and Engine revisions are serialized in a versioned configuration file. At workflow initialization, the Engine-side selector loads this file and resolves each role to the corresponding preconfigured Engine service; it does not search the six-arm space for every request. This procedure freezes

$$d^*_{\text{BCP}} = (\text{vLLM}, \text{H}_2\text{O}), \qquad d^*_{\text{DRB}} = (\text{OmniKV}, \text{H}_2\text{O}). \tag{7}$$

The runtime records the requested and actual execution modes together with completion, failure, or fallback outcomes. Neither the capability profiles, ranking rule, nor frozen mappings are updated using the formal six-arm results.

A.2 Cache-Aware Runtime Coordination Settings

Benchmark Construction and Fair Replay. For both benchmarks, the formal evaluation follows a trajectory recording and replay procedure. We first execute and record a complete trajectory, including request arrival times, input and output lengths, session/turn relationships, and prefix information. All configurations then replay the same trajectory. This ensures that different coordination policies do not observe different workloads due to changes in generated content, input/output lengths, or subsequent execution trajectories; configurations differ only in request scheduling and KV-cache management behavior.

For SCBench (Li et al., 2025b), the original data contain multi-turn interactions but do not provide real user request timestamps. We construct each sample as an independent session and use a Poisson process to simulate initial session arrivals. After each response completes, a user think time is sampled to determine when the next turn of the same session enters the system. Think time follows

$$\mathcal{N}(5, 1)\ \text{s}.$$

After the trajectory is recorded once, all configurations replay the same session-arrival and think-time sequences.

For Mooncake Bench (Qin et al., 2026), we use the request timestamps, input and output lengths, and block-hash information recorded in the production trace, while preserving the original session start times, inter-turn intervals, and complete turn sequences. Different load levels are constructed only by replaying 50%, 75%, and 100% of the sessions whose first requests fall within the same 15-minute window, without altering the original timing relationships between requests.

Experimental Parameters. SCBench uses Qwen3-8B on a single RTX PRO 6000 with a 96-GiB L2 KV cache. Mooncake Bench also uses Qwen3-8B, running on a single H100 with a 96-GiB L2 KV

cache. For SCBench, the Guard waiting-protection threshold is evaluated at $W = 40\,\mathrm{s}$ and $W = 60\,\mathrm{s}$; for Mooncake Bench, we use $W = 40\,\mathrm{s}$. Except for the coordination policy being compared, all experiments use the same replay trajectory, model configuration, cache capacity, and other serving parameters.

Coordination and Session-Aware KV Retention. CACHE-AWARE orders ready requests according to the KV-residency view maintained by the harness. For each request, the harness matches reusable prefix blocks sequentially from the beginning of the prompt. Matching stops at the first missing or evicted block, since subsequent blocks cannot independently form a reusable prefix. The harness separately counts reusable prefix tokens in the GPU and CPU tiers. A request is classified as *hot* if reusable prefixes cover at least 50% of the prompt, and *cold* otherwise.

Cold requests retain the default priority, while the priority of a hot request is defined as the negative number of reusable KV blocks:

$$\text{priority} = -N_{\text{reuse}},$$

where N_{reuse} denotes the number of reusable KV blocks. Because the engine schedules requests with smaller priority values first, hot requests are executed before cold requests, and hot requests are further ordered by decreasing reusable-prefix length.

The residency view reflects the cache state currently observed by the harness and does not constitute a residency guarantee; when a request actually executes, the engine rechecks the reusable content against the current cache state.

GUARD is submitted by the harness to the engine at fixed times and supports subsequent updates. Once a request has waited longer than the threshold W, the engine promotes it to a protected priority so that later cache-hot requests can no longer bypass it. Protected requests are still ordered according to their waiting order.

SESSION-AWARE is independent of the request-scheduling priority above and acts only on KV-cache eviction order. The harness maintains a session-return probability p by continuously summarizing completed conversations and their subsequent return behavior, and uses this statistic to provide a continuation estimate for the current session. The estimate is updated only from completed conversations and does not use future information from the current replay.

After a request completes, the engine records the corresponding context's p and completion time and computes its retention priority as

$$r = p \cdot e^{-\mathrm{idle}/400}.$$

Here, idle denotes the time elapsed since the context's most recent request completed. Contexts that are more likely to return and have been accessed more recently receive higher retention priority, while the priority gradually decays as idle time increases.

A.3 PROTOCOL COST ACCOUNTING

HEAR adds control-plane metadata processing rather than model-side computation. It does not introduce additional model layers or per-token GPU operators; its online work consists of state updates, object lookups, policy-input construction, control handling, and execution-outcome reporting. The computation introduced by OmniKV, H_2O, or SnapKV belongs to the corresponding execution mode rather than to the protocol itself.

For cache-aware coordination, Engine KV events update a Harness-maintained residency directory. At each scheduling point, the Harness queries reusable prefix blocks, constructs scheduling preferences, and attaches the applicable waiting requirement. The Engine applies these controls and reports the realized execution order, cache reuse, retries, and errors. For capability-aware configuration, candidate profiling, filtering, ranking, and verification occur during offline development. The online path loads the frozen role-to-mode mapping once at workflow initialization and performs a constant-size lookup when dispatching each model request.

Table 7 summarizes the frequency and cost attribution of these operations. All formal HEAR-enabled runs include the corresponding online control-plane path; the end-to-end results in the main text therefore reflect the net performance after these operations.

Table 7: **HEAR protocol cost accounting.**

Component	Frequency	Additional Online Work	Included in Formal E2E Time
KV-state update	Per cache event	Residency-directory metadata update	Yes
Scheduling decision	Per dispatch point	Prefix lookup, priority construction, and Guard check	Yes
Role-to-mode lookup	Per model request	Lookup in the frozen role mapping	Yes
Outcome reporting	Per inference/control outcome	Status, actual mode, reuse, retry, and error metadata	Yes
Profile calibration	Offline development stage	Candidate profiling, filtering, ranking, and verification	No

Table 8: **Execution-mode configurations for BrowseComp-Plus.**

Mode	Cache and Token Settings	Execution Capacity	Mode-specific Parameters
Main Role			
vLLM	**Prefix caching** Max. context 202,752; 32,768 batched tokens	16 sequences	**Dense execution**
OmniKV	**Radix cache** 32,768-token prefill chunks	4 batch / 4 decode / 16 GPU sequences	**Full-attention layers:** automatic; **Retention:** 8 sink / 256 recent / 4,096 decode tokens; **Prefill workspace:** 8 GiB
Reader Role			
vLLM	**Prefix caching** Max. context 202,752; 16,384 batched tokens	16 sequences	**Dense execution**
H_2O	**Chain cache** 16,384-token prefill chunks	12 batch / 12 decode / 36 GPU sequences	**Budgets:** 16,384 prefill / 8,192 decode tokens; **Recent ratio:** 0.2; **Scoring:** 128-token window with logits-based prefill scoring
SnapKV	**Chain cache** 16,384-token prefill chunks	12 batch / 12 decode / 36 GPU sequences	**Retention:** 64 sink / 512 recent / 4,096 decode tokens; **Observation window:** 32 tokens; **Other:** no full-attention layers or prefill sparsification

A.4 Role-Specific Models and Execution Modes

All configurations use GLM-4.7-Flash, with Main-side and Reader/sub-agent-side inference placed on separate H100 GPUs. Dense execution uses vLLM 0.26.0, while OmniKV, H_2O, and SnapKV are provided through SparseEngine. Tables 8 and 9 summarize the cache, capacity, and mode-specific settings used in the formal BrowseComp-Plus and DeepResearchBench runs, respectively. We report the behavior-affecting parameters here; complete launch commands, environment manifests, and source hashes are included in the supplementary artifact.

For BrowseComp-Plus, the Harness allows at most four Main and twelve Reader requests in flight; generation uses temperature zero, top-$p = 1$, and disables model thinking. OmniKV is based on SparseEngine revision `67174b1b`; the exact patched source snapshots are archived with the artifact.

For DeepResearchBench, Main requests use temperature 0.2, top-$p = 0.9$, a 4,096-token output limit, and model thinking. Sub-agent requests use the same sampling parameters with a 512-token output limit and disable thinking. All sparse modes use SparseEngine revision `469aa791`.

Table 9: **Execution-mode configurations for DeepResearchBench.**

Mode	Cache and Token Settings	Execution Capacity	Mode-specific Parameters
Main Role			
vLLM	**Prefix caching** Max. context 200,000; 32,768 batched tokens	10 sequences	**Dense execution**
OmniKV	**Radix cache** 32,768-token prefill chunks	10 batch / 10 decode / 16 GPU sequences	**Full-attention layers:** $\{0, 3, 9, 18, 22, 27, 43\}$
Sub-Agent Role			
vLLM	**Prefix caching** Max. context 200,000; 32,768 batched tokens	10 sequences	**Dense execution**
H_2O	**Chain cache** 32,768-token prefill chunks	10 batch / 10 decode / 44 GPU sequences	**Budgets:** 8,192 prefill / 4,096 decode tokens; **Recent ratio:** 0.2; **Scoring:** 128-token window with logits-based prefill scoring
SnapKV	**Chain cache** 32,768-token prefill chunks	10 batch / 10 decode / 44 GPU sequences	**Retention:** 64 sink / 512 recent / 4,096 decode tokens (4,672 total); **Observation window:** 32 tokens; **Other:** no full-attention layers or prefill sparsification

SnapKV Prefill Admission. SnapKV uses a compressed per-layer retention budget of 4,672 tokens, but does not sparsify the prefill phase. Consequently, a long request temporarily reserves space for its full prompt until prefill completes. For DeepResearchBench, the Harness therefore applies a FIFO prefill gate to SnapKV sub-agent requests, limiting the global in-flight prefill workload to 456,837 tokens. A grant is released upon the first streamed output token, when the Engine clears the request's full-prefill reservation. This Harness-side control is separate from SnapKV's Engine-side retention parameters.

B Additional Results

B.1 Runtime Behavior of Cache-Aware Runtime Coordination

Figure 4 shows the runtime timelines of the closed-loop SCBench workload. Each turn is decomposed into queue wait, model service, and user gap. Because all configurations replay the same session-arrival and think-time trajectory, the user-gap pattern is shared across methods, and the main differences arise from request execution order. Under FCFS, queueing accumulates and substantially extends the overall execution horizon. Cache-Aware prioritizes requests with reusable prefixes and concentrates more service earlier in the trace. Adding Guard limits prolonged waiting caused by repeated bypass while retaining the cache-aware execution-order policy.

Figure 5 shows the runtime behavior of the different scheduling policies under the Mooncake replay. GPU KV occupancy follows a similar trajectory across policies, while the main differences appear in KV restoration and queueing: session-aware variants restore substantially more KV, whereas cache-aware scheduling keeps restoration lower. Compared with FCFS, the coordinated policies also reduce the peak number of waiting requests, with Guard further limiting queue buildup without materially changing the overall KV-occupancy pattern.

B.2 Complete Six-Arm Results of Role-specific inference configuration

Table 10 reports the complete BrowseComp-Plus results over the shared 2×3 execution-mode space. Panel (a) includes aggregate efficiency and per-question latency, whereas Panel (b) reports task quality and completion outcomes. Throughput is measured in questions per hour, and all end-to-end latencies are reported in minutes. C, I, U, and F denote correct, incorrect, unresolved, and run-failure questions, respectively.

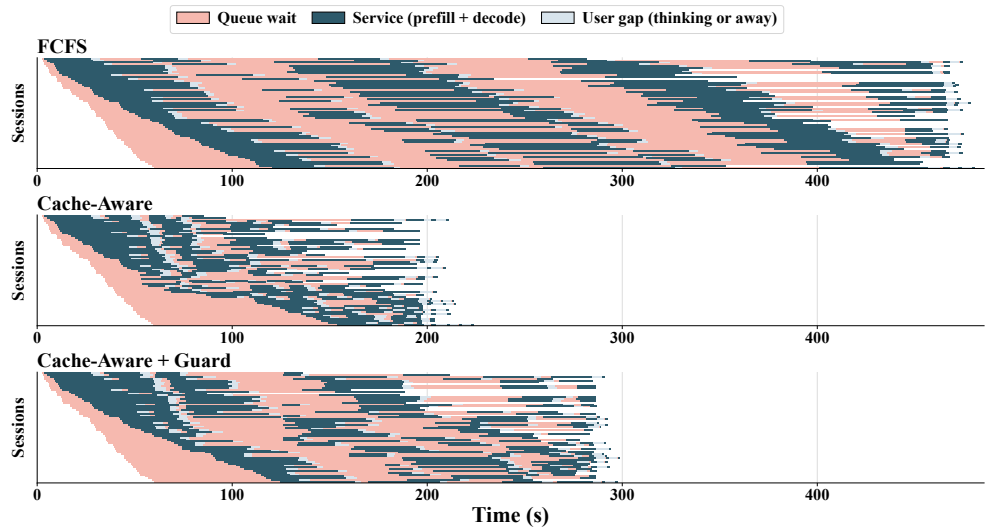

Figure 4: **Runtime execution timelines on SCBench.** Each session is decomposed into queue wait, model service (prefill + decode), and user gap.

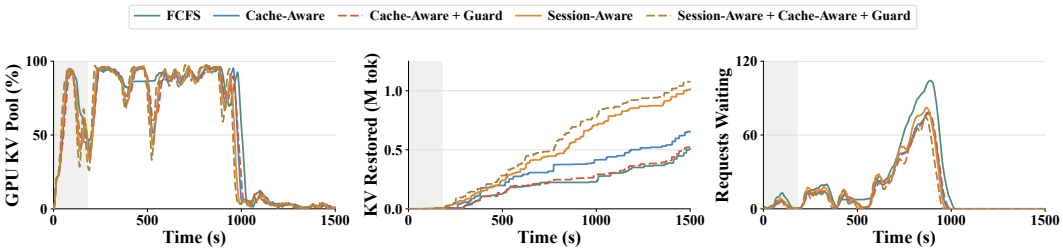

Figure 5: **Runtime behavior under the Mooncake replay.** GPU KV-pool occupancy, cumulative KV restoration, and the number of waiting requests over time for different scheduling policies.

Table 10: **Complete BrowseComp-Plus six-arm results.**

		Efficiency and latency						*Quality and completion outcomes*					
Main	**Reader**	**Wall (h) ↓**	**Speedup ↑**	**Throughput ↑**	**Median ↓**	**P95 ↓**	**Max ↓**	**Acc. (%) ↑**	**Retr. Recall (%) ↑**	**C**	**I**	**U**	**F**
	vLLM	5.69	1.00×	36.57	14.63	40.00	41.32	46.15	69.22	96	46	54	12
vLLM	**H_2O**	**4.62**	**1.23×**	**45.00**	**10.98**	**39.58**	**41.81**	**46.63**	**70.17**	**97**	**59**	**45**	**7**
	SnapKV	4.83	1.18×	43.03	10.75	40.38	43.26	44.23	70.56	92	63	40	13
	vLLM	5.37	1.06×	38.72	13.67	40.28	46.54	47.12	68.75	98	42	53	15
OmniKV	**H_2O**	4.98	1.14×	41.73	12.64	37.24	42.37	43.27	68.33	90	61	49	8
	SnapKV	5.29	1.07×	39.28	11.14	40.60	42.60	43.75	69.46	91	56	43	18

Table 11 reports the corresponding DeepResearchBench results. Panel (a) includes makespan, throughput, and per-instance latency, while Panel (b) reports the overall RACE score, its four quality dimensions, and completion outcomes. All quality scores are multiplied by 100. N denotes the number of reports evaluated by the RACE judge; Comp., Instr., and Read. denote comprehensiveness, instruction following, and readability, respectively.

As shown in Tables 10 and 11, the configurations selected during development achieve the lowest aggregate completion time within their respective benchmarks. BrowseComp-Plus selects vLLM–H_2O, while DeepResearchBench selects OmniKV–H_2O. In both cases, the efficiency improvement is obtained without an observed degradation in task quality. The complete results also expose secondary trade-offs: for example, OmniKV–H_2O yields a lower BrowseComp-Plus P95 latency despite not

Table 11: **Complete DeepResearchBench six-arm results.**

Main	Sub-agent	*Efficiency and latency*							*Quality and completion outcomes*							
		Makespan (h) ↓	Speedup ↑	Throughput ↑	E2E latency (min) ↓				N	RACE (%) ↑	Comp.	Insight	Instr.	Read.	Errors	Empty
					Mean	Median	P95	Max								
vLLM	**vLLM**	5.93	1.00×	16.7	29.46	30.64	39.81	41.18	97	40.65	39.93	38.68	42.98	42.91	1	4
	H_2O	2.76	2.15×	36.3	13.17	13.17	15.64	21.27	96	40.46	39.65	38.68	42.81	42.57	0	4
	SnapKV	2.85	2.08×	35.1	15.13	15.19	17.70	20.26	99	40.01	39.33	37.78	42.51	42.45	0	1
OmniKV	**vLLM**	6.17	0.96×	16.2	30.41	30.64	39.93	40.64	98	41.14	40.30	39.26	43.57	43.20	0	2
	H_2O	**2.42**	**2.45×**	**41.3**	**12.21**	**12.24**	**15.82**	**17.06**	**99**	**41.05**	**39.91**	**39.59**	**43.26**	**43.10**	**0**	**1**
	SnapKV	2.76	2.15×	36.3	14.70	14.66	18.33	19.63	99	40.50	39.44	38.62	42.99	42.83	0	1

Table 12: **Role-level workload shapes under the selected configurations.**

Benchmark	Role	Req./Item	Input/Req. (K)	Output/Req. (K)	Workflow Pattern
BrowseComp-Plus	Main	3.0	52.5	0.36	Dynamic coordination and answer synthesis
	Reader	13.3	59.1	0.36	Question-dependent document reads
DeepResearchBench	Main	5.4	32.5	4.42	Round-synchronized review and synthesis
	Sub-agent	12.0	135.3	0.51	Four parallel agents over three fixed rounds

Table 13: **Role-level execution behavior under arm-pair comparisons.**

Benchmark	Role	Alternative → Selected	Mean Latency (s)	Gain	Execution Evidence
BrowseComp-Plus	Main	OmniKV → vLLM	17.0 → 14.9	1.14×	Selected Main output: 0.36K tokens/request
	Reader	vLLM → H_2O	53.4 → 9.5	5.60×	Prefill -76.2%; reuse $6.2\% \rightarrow 80.3\%$
DeepResearchBench	Main	vLLM → OmniKV	69.4 → 64.2	1.08×	Selected Main output: 4.42K tokens/request
	Sub-agent	vLLM → H_2O	421.5 → 58.4	7.22×	Chain reuse: 49.8%

minimizing its wall time, while the DeepResearchBench sparse configurations differ substantially more in execution efficiency than in RACE.

B.3 Role-Level Workload and Execution Analysis

The two benchmarks share the same high-level role organization but expose substantially different workload shapes. Table 12 reports the role-level workload observed under the configuration selected for each benchmark. Values are normalized per benchmark item, i.e., per question for BrowseComp-Plus and per instance for DeepResearchBench.

B.4 Quality Evaluation

BrowseComp-Plus. We evaluate answer quality over all 208 questions in each arm. Unresolved questions and run failures remain in the accuracy denominator, and all formal Judge labels are available. Table 14 reports the complete outcome distribution and Wilson 95% confidence intervals, while Table 15 reports retrieval and citation quality.

Table 14: **BrowseComp-Plus answer-quality outcomes.**

Main	Reader	Outcome Counts				Answer Quality	
		Correct	Incorrect	Unresolved	Run Failure	Judge Acc. (%)	95% Wilson CI
vLLM	**vLLM**	96	46	54	12	46.15	[39.51, 52.94]
OmniKV	**vLLM**	98	42	53	15	47.12	[40.45, 53.89]
vLLM	**H_2O**	97	59	45	7	46.63	[39.98, 53.41]
OmniKV	**H_2O**	90	61	49	8	43.27	[36.72, 50.06]
vLLM	**SnapKV**	92	63	40	13	44.23	[37.65, 51.02]
OmniKV	**SnapKV**	91	56	43	18	43.75	[37.18, 50.54]

Table 15: **BrowseComp-Plus retrieval and citation quality.**

Main	Reader	Answers Produced	Evidence Quality (%)		
			Retrieval Recall	Citation Precision	Citation Recall
vLLM	**vLLM**	142	69.22	78.01	30.82
OmniKV	**vLLM**	140	68.75	77.71	33.90
vLLM	**H_2O**	156	70.17	75.44	28.62
OmniKV	**H_2O**	151	68.33	74.78	27.87
vLLM	**SnapKV**	155	70.56	72.63	28.10
OmniKV	**SnapKV**	147	69.46	75.03	28.47

Table 16: **DeepResearchBench RACE quality and evaluation coverage.**

Main	Sub-agent	Evaluation Coverage			RACE (%)				
		Judged	Empty	Task Errors	Overall	Comp.	Insight	Instr. Follow.	Read.
vLLM	**vLLM**	97	4	1	40.65	39.93	38.68	42.98	42.91
OmniKV	**vLLM**	98	2	0	41.14	40.30	39.26	43.57	43.20
vLLM	**H_2O**	96	4	0	40.46	39.65	38.68	42.81	42.57
OmniKV	**H_2O**	99	1	0	41.05	39.91	39.59	43.26	43.10
vLLM	**SnapKV**	99	1	0	40.01	39.33	37.78	42.51	42.45
OmniKV	**SnapKV**	99	1	0	40.50	39.44	38.62	42.99	42.83

Table 17: **Paired quality comparison against all-vLLM.**

Benchmark	Selected	Paired N	Quality Difference (%)	95% Bootstrap CI (%)	Paired Test
BrowseComp-Plus	vLLM–H_2O	208	$+0.48$	$[-5.29, 6.25]$	McNemar $p = 1.000$
DeepResearchBench	OmniKV–H_2O	89	$+0.51$	$[-0.36, 1.46]$	Paired t-test $p = .278$

DeepResearchBench. We evaluate generated reports using the official reference-free RACE evaluator. Because task errors and empty reports reduce the number of valid articles available to the Judge, Table 16 reports both evaluation coverage and the four RACE dimensions.

Paired Quality Comparisons. Table 17 compares each benchmark's selected configuration with its all-vLLM baseline. BrowseComp-Plus uses all 208 paired questions. For DeepResearchBench, we use the 89 instances that receive valid RACE scores in every arm, ensuring a common paired cohort for the six-arm analysis. Confidence intervals are percentile bootstrap intervals over paired questions or instances.

Judge Provenance. All BrowseComp-Plus outputs are evaluated with Qwen3-32B using a 4,096-token Judge output limit and the same intended evaluation settings. Arms A, B, and F are judged on an RTX PRO 6000, whereas Arms C, D, and E are judged on an H100. We retain this provenance and report per-question paired outcomes; re-evaluating all arms in a single Judge environment would further remove this environmental difference. DeepResearchBench uses the vendored official RACE evaluator with gpt-5.4-mini. Provider errors from the initial high-concurrency Judge pass are re-evaluated at a concurrency of four. The reported tables contain the final successful Judge outputs.