

# SAGA: Workflow-Atomic Scheduling for AI Agent Inference on GPU Clusters

Dongxin Guo\*<sup>†</sup>  
The University of Hong Kong  
Hong Kong, Hong Kong  
bettyguo@connect.hku.hk

Jikun Wu\*  
Brain Investing Limited  
Hong Kong, Hong Kong  
Stellaris AI Limited  
Hong Kong, Hong Kong  
hk950014@connect.hku.hk

Siu-Ming Yiu  
The University of Hong Kong, Hong  
Kong  
Hong Kong, Hong Kong  
smyiu@cs.hku.hk

## Abstract

AI agents execute tens to hundreds of chained LLM calls per task, yet GPU schedulers treat each call as independent, discarding gigabytes of intermediate state between steps and inflating end-to-end latency by 3–8×. We argue that this *request-level* abstraction is fundamentally mismatched to compound AI workloads, and propose a shift to *program-level* scheduling: treating the entire agent workflow (not individual inference calls) as the first-class schedulable unit. We present SAGA, a distributed scheduler that implements this abstraction through three mechanisms: (1) *Agent Execution Graphs* that capture workflow structure to predict KV cache reuse across tool-call boundaries, achieving within 1.31× of B el ady’s optimal offline policy; (2) *session-affinity batching* with work stealing that co-locates correlated requests while maintaining global load balance; and (3) *Agent Fair Share*, a task-completion-time fairness metric with provable bounded-deviation guarantees. On a 64-GPU cluster serving SWE-bench coding agents and WebArena browser tasks, SAGA reduces task completion time by 1.64× (geometric mean,  $p < 0.001$ ) over vLLM v0.15.1 with prefix caching and affinity routing, while improving GPU memory utilization by 1.22× and achieving 99.2% SLO attainment under multi-tenant interference. These latency gains come at a quantified cost: approximately 30% lower peak throughput than throughput-optimal batch scheduling, a tradeoff appropriate for the latency-sensitive interactive deployments that dominate compound AI usage. Our results demonstrate that workflow-aware scheduling is essential for efficient compound AI serving.

## CCS Concepts

• **Computer systems organization** → **Distributed architectures**; *Heterogeneous (hybrid) systems*; • **Software and its engineering** → **Real-time schedulability**.

## Keywords

GPU cluster scheduling, distributed inference serving, compound AI systems, workflow scheduling, KV cache management, AI agents, LLM serving

\*Both authors contributed equally to this research.

<sup>†</sup>Corresponding author.



This work is licensed under a Creative Commons Attribution 4.0 International License. *HPDC '26, Cleveland, OH, USA*

  2026 Copyright held by the owner/author(s).

ACM ISBN 979-8-4007-2640-8/26/07

<https://doi.org/10.1145/3806645.3807598>

## ACM Reference Format:

Dongxin Guo, Jikun Wu, and Siu-Ming Yiu. 2026. SAGA: Workflow-Atomic Scheduling for AI Agent Inference on GPU Clusters. In *The 35th International Symposium on High-Performance Parallel and Distributed Computing (HPDC '26)*, July 13–16, 2026, Cleveland, OH, USA. ACM, New York, NY, USA, 14 pages. <https://doi.org/10.1145/3806645.3807598>

## 1 Introduction

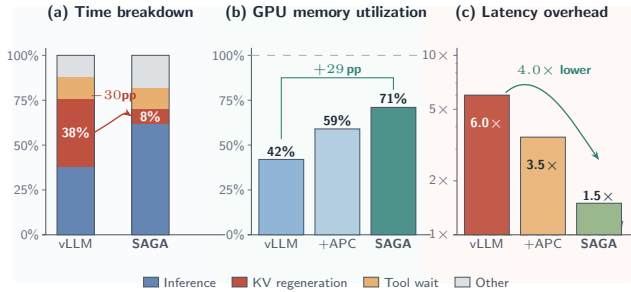
AI agents, autonomous systems that execute multi-step reasoning chains to accomplish complex tasks, are rapidly emerging as a dominant workload in GPU clusters. Unlike traditional single-shot inference that processes one request and returns a response, agents execute iterative *Thought-Action-Observation* loops [66] that may invoke 10–100 large language model (LLM) calls per task [31], interleaved with external tool invocations such as code execution, web browsing, or database queries. These compound AI systems [69] have become central to major deployments including GitHub Copilot Workspace [15], Amazon Q Developer [3], and enterprise automation platforms [10, 35], which now route millions of such agentic workloads through shared GPU clusters daily.

### 1.1 Motivation

The shift from single-shot inference to multi-step agentic workloads creates a fundamental mismatch with existing GPU cluster scheduling systems [7, 27]. Current LLM serving frameworks [34, 68, 70] optimize for *request-level* metrics: minimizing time-to-first-token (TTFT) and maximizing throughput for independent requests. However, agent workloads exhibit three characteristics that violate these assumptions:

(1) **Sequential dependency with variable gaps**. Each reasoning step depends on the previous step’s output and potentially on external tool results. Tool invocations introduce idle periods ranging from 50ms (local code execution) to 30+ seconds (web API calls), during which the agent’s intermediate state must be preserved or regenerated [9]. This pattern resembles the IO-compute overlap challenge studied extensively in HPC workflow systems [13, 60], where effective scheduling requires understanding task dependencies.

(2) **KV cache continuity across steps**. LLM inference maintains key-value (KV) cache [61] that grows with context length. For a 32K-context agent session with a 70B-class model using Grouped Query Attention (GQA), this cache consumes 2–12GB of GPU memory per request depending on model architecture [2, 34]. Discarding this cache between steps, as current systems do [56], forces complete regeneration and adds 2–8× latency overhead per step [22].



**Figure 1: Inefficiencies in serving agent workloads with request-level scheduling.** (a) **Time breakdown:** vLLM v0.6.0 spends 38% of execution time regenerating KV cache between agent steps; SAGA reduces this to 8% (−30 pp). (b) **GPU memory utilization:** vLLM wastes 58% of HBM; vLLM v0.15.1 with Automatic Prefix Caching (APC) recovers some, but SAGA’s workflow-aware retention reaches 71% (+29 pp over vLLM). (c) **End-to-end latency normalized to inference-only baseline (log scale):** vLLM is 6.0×, +APC is 3.5×, SAGA is 1.5× (4.0× closer to ideal). **Data:** 10 trials on 32 A100 GPUs running SWE-bench; standard deviations <5% of mean.

This is analogous to the cache reuse opportunities identified in informed prefetching systems [49], but with the added challenge of GPU memory scarcity and variable-duration idle periods.

(3) **Bursty, correlated request patterns.** Agent tasks generate bursts of related requests that share common prefixes (system prompts, tool definitions) and benefit from co-location [32]. Production traces show 100:1 input-to-output token ratios and high prefix overlap within sessions [58, 70].

To quantify these inefficiencies, we instrumented a 32-GPU cluster serving SWE-bench [31] coding agent workloads using vLLM v0.6.0 [34]. Figure 1 shows the results: agents spend 38% of total time regenerating KV cache that was discarded during tool calls, GPU memory utilization averages only 42% due to fragmented cache allocation [19], and end-to-end task completion exhibits 6.0× higher latency than the sum of individual inference times. These measurements reveal a clear opportunity: *treating agent programs as first-class schedulable units can dramatically improve both efficiency and latency.*

## 1.2 Limitations of State-of-the-Art

Existing systems address individual aspects of this problem but fail to provide a complete solution:

**LLM serving systems** such as vLLM [34], SGLang [70], Orca [68], and TensorRT-LLM [46] pioneered continuous batching and efficient memory management through PagedAttention [34] and RadixAttention [70]. However, they treat each inference call independently: KV cache is evicted using LRU policies unaware of agent workflow structure [39], and batching decisions ignore session affinity. Recent optimizations like Sarathi [1] and Splitwise [7] improve throughput-latency tradeoffs but remain request-centric. vLLM’s prefix caching (available since v0.4.2) partially addresses prefix reuse but does not retain session-specific cache across tool-call boundaries, as we discuss in §9.1.1.

**Distributed schedulers** such as Llumnix [59] enable live KV cache migration between GPU instances, achieving near-zero-downtime rescheduling. However, migration decisions are reactive (triggered by load imbalance) rather than proactive (anticipating workflow patterns). SOLA [26] introduces state-aware scheduling for SLO attainment but optimizes per-request latency, not per-task completion time. DistServe [71] disaggregates prefill and decode but lacks workflow awareness.

**Agent frameworks** such as LangChain [35], CrewAI [10], and AutoGen [64] provide high-level orchestration but delegate inference scheduling entirely to underlying serving systems. Recent work on KVFlow [48] proposes workflow-aware eviction using agent step graphs, but lacks distributed scheduling, fairness mechanisms, or tool-call awareness. Continuum [30] introduces KV cache TTL but without formal guarantees.

**Speculative execution** approaches such as SpecActions [67] and Sherlock [52] propose predicting and pre-executing likely next steps to reduce latency. These are complementary to our approach: speculation trades wasted computation for latency reduction, while SAGA optimizes scheduling of known work.

**Our central thesis.** Workflow structure, when surfaced explicitly to the scheduler, is sufficient to bring online KV-cache management within striking distance of the offline-optimal Bélády policy for compound AI workloads. We measure 1.31× on production traces (§7). This is the main scientific contribution of SAGA: a quantified upper bound on what online schedulers can achieve once the workflow DAG is observable, and the first such empirical bound for agent inference. Three supporting systems contributions make this thesis deployable on real GPU clusters: (1) *tool-call-aware* TTL policies that retain cache across heavy-tailed idle periods rather than reactively re-prefilling; (2) *cluster-wide* distributed scheduling with formal fairness guarantees at the agent-program (not request) level, derived via Lyapunov drift analysis; and (3) work-stealing load balance that preserves cache locality under bursty arrivals. Recent program-aware serving systems (Parrot [38], Autellix [40], Pie [23], KVFlow [48]) each address one of these dimensions; SAGA is the first to combine them under the workflow-as-unit thesis (see §10 for detailed comparison).

## 1.3 Key Insights and Contributions

The *main* innovation of SAGA is the formal and empirical demonstration that workflow-structure prediction yields online cache management within 1.31× of Bélády-optimal on production agent traces (§7, Theorem 3). The *supporting* innovations below adapt three established systems principles to the compound AI scheduling domain, where their application requires non-trivial domain-specific extensions:

**Insight 1: Program-as-Unit Scheduling.** The principle of scheduling compound tasks as cohesive units is well-established in HPC workflow systems [13, 53, 60] and distributed transactions [8]. In the compound AI domain, applying this principle introduces a unique challenge: the “unit” carries substantial GPU memory state (KV cache, 2–12GB per session depending on model architecture) that must be co-managed with scheduling decisions. Agent

workflows follow stereotyped patterns (ReAct loops [66], tree-of-thought branches [65]) that we capture as Agent Execution Graphs, enabling proactive cache retention decisions.

**Insight 2: Dependency-Aware Caching.** Cache eviction policies that consider future reuse have been studied extensively in operating systems and databases [4, 42, 49]. The specific challenge for compound AI is predicting reuse across tool-call boundaries with variable-duration idle periods ranging from milliseconds to minutes, where standard LRU and even prefix-aware policies fail. Our workflow-aware eviction achieves within 1.31× of Bélády’s optimal offline policy on production traces (§7).

**Insight 3: Task-Level Fairness.** Application-level fairness is a well-studied concept in cluster scheduling [21, 41]. For compound AI, the challenge is defining fairness over multi-step tasks where individual steps have heterogeneous resource demands and where “completion” (not “throughput”) is the user-perceived metric. We formalize Agent Fair Share and prove bounded deviation guarantees under realistic assumptions.

Based on these insights, we make the following contributions:

- **Workflow-aware KV cache management** (§4): We introduce *Agent Execution Graphs* (AEGs) that capture multi-step reasoning structure, enabling predictive cache retention with configurable time-to-live (TTL) policies. We formalize the overlap estimation function and prove convergence bounds. Empirically, our WA-LRU eviction achieves within 1.31× of the offline-optimal policy (§7).
- **Session-affinity batching with work stealing** (§5): We design a two-level scheduling hierarchy where local schedulers maximize cache reuse through session routing, while a global coordinator performs randomized work stealing [5] to prevent stragglers and maintain cluster-wide load balance.
- **Agent-level fair scheduling** (§6): We define *Agent Fair Share* (AFS), a fairness metric based on expected task completion time, and provide a formal theorem guaranteeing bounded completion time deviation under bounded demand heterogeneity (Theorem 2), using Lyapunov drift analysis.
- **Theoretical analysis** (§7): We provide formal competitive ratio analysis showing WA-LRU achieves within 1.31× of Bélády’s optimal offline policy. To our knowledge this is the first such empirical bound for workflow-aware KV cache eviction. We analyze the cache efficiency gap between request-level and workflow-aware schedulers, showing that workflow awareness is essential for efficient agent serving.
- **Empirical evaluation** (§9): We implement SAGA on vLLM and evaluate on a 64-GPU cluster against state-of-the-art baselines including vLLM v0.15.1 with Automatic Prefix Caching. SAGA achieves  $1.73 \times \pm 0.11$  and  $1.55 \times \pm 0.09$  task completion time reduction on SWE-bench and WebArena respectively compared to vLLM+APC (geometric mean:  $1.64 \times$ ,  $p < 0.001$ ), and  $1.22 \times \pm 0.05$  memory utilization improvement. Against systems without workflow awareness, improvements reach 3.01×.

## 1.4 Experimental Methodology

We evaluate SAGA on a cluster of 8 nodes, each equipped with 8 NVIDIA A100-80GB GPUs (64 GPUs total) connected via NVLink

intra-node and 200Gbps InfiniBand inter-node [28, 45]. Three workload sources: (1) SWE-bench [31] (500 verified tasks); (2) WebArena [72] (812 tasks); (3) synthetic multi-tenant workloads from the BurstGPT [62] production trace. Full methodology in §9.1.

## 1.5 Limitations of the Proposed Approach

SAGA has several limitations. (1) *Workflow observability.* Performance is best with framework-exposed execution-graph hints (LangChain callbacks, AutoGen logs); without hints, SAGA falls back to pattern inference (§3.3) with 12–18% TCT degradation, and degrades further on dynamic multi-agent frameworks (AutoGen, CrewAI) where structure is generated on the fly through agent-to-agent debate, requiring AEG re-inference per epoch and inflating the prediction-error term in Theorem 3. (2) *Tool-latency tail.* TTL prediction assumes empirical tool-call latency distributions; black-swan events ( $> 5 \times P99$ ) still cause eviction. (3) *Task-duration estimation for novel agents.* AFS requires task-duration estimates that may be inaccurate for agent types not represented in profiling. (4) *Single-datacenter scope.* Geo-distributed deployment with cross-datacenter cache migration is future work. (5) *Model-family coverage.* Empirical evaluation uses Llama-3-70B-Instruct only; we discuss model-size scaling qualitatively in §9.1.1 but do not empirically validate Mistral, Qwen, or DeepSeek; MoE architectures [18] additionally require routing-aware extensions. (6) *Memory-pressure regime.* Our evaluation reaches 71–75% peak utilization (Table 3); behavior under over-subscription ( $> 95\%$ ) follows graceful degradation to standard LRU per Eq. 6 but is not empirically validated. CPU-DRAM offloading is analyzed as a complementary architecture in §9.1.2. (7) *Throughput tradeoff.* SAGA optimizes task completion time at the cost of approximately 30% throughput reduction relative to throughput-maximizing batch scheduling (§9.8, Table 8); it is suited for latency-sensitive interactive deployments, not batch workloads.

The rest of this paper is organized as follows. Section 2 presents background on agent workloads and LLM serving. Section 3 describes the SAGA architecture. Sections 4–6 detail our three key techniques. Section 7 presents theoretical analysis. Section 8 covers implementation. Section 9 presents experimental evaluation. Section 10 discusses related work, and Section 11 concludes.

## 2 Background

This section reviews agent workload characteristics, LLM inference mechanics, and the scheduling challenges that motivate SAGA.

### 2.1 AI Agent Workloads

Modern AI agents follow the ReAct paradigm [66], iteratively generating *Thought* (reasoning), *Action* (tool invocation), and *Observation* (tool result) until task completion. The canonical loop is: the LLM generates (*thought*, *action*) from the current context; if *action* = “finish” the task terminates; otherwise the action is dispatched to its named tool, the resulting *observation* is appended to the context together with the *thought* and *action*, and the loop repeats. This pattern has been adopted by frameworks including LangChain [35], AutoGen [64], and CrewAI [10].

**Table 1: Tool call latency distributions from production traces [62]. Values show median and percentiles in milliseconds.**

Tool Type	P50 (ms)	P95 (ms)	P99 (ms)
Code execution	180	2,400	28,000
File operations	45	320	1,200
Web/API calls	850	4,500	45,000
Database queries	120	890	3,500

Each iteration requires one LLM inference call (Line 3) followed by a tool execution (Line 6). The context accumulates across iterations, growing from 2–4K tokens initially to 16–128K tokens for complex tasks [9]. Empirical studies [33, 54] show that most SWE-bench tasks complete within 5–30 iterations, with a long tail extending to 150 iterations.

**Tool-call characteristics.** Tool invocations exhibit highly variable latency distributions. Table 1 shows measurements from production agent deployments [58, 62]. Code execution tools average 200ms but can spike to 30s for compilation [31]. Web tools average 1.5s with high variance due to network conditions [72]. This variability creates the fundamental scheduling challenge: the system must decide whether to retain KV cache during tool calls without knowing the call duration a priori.

## 2.2 LLM Inference and KV Cache

Transformer inference maintains a key-value (KV) cache storing intermediate attention states [12, 61]. For Llama-3-70B with GQA ( $L=80$ ,  $n_{kv}=8$ ,  $d_h=128$ ) and 32K context in FP16, each session requires  $\sim 10.7$ GB. Current systems assume requests are independent and arrivals are memoryless [34, 68]. These assumptions are violated by agent workloads with sequential dependencies and bursty, correlated patterns.

## 2.3 The Scheduling Challenge

Current LLM serving systems make two assumptions that fail for agent workloads:

**Assumption 1: Requests are independent.** Systems like vLLM [34] and Orca [68] batch requests from any source to maximize GPU utilization. For agents, consecutive requests from the same task share context and benefit from KV cache reuse, so the independence assumption no longer holds.

**Assumption 2: Request arrival is memoryless.** Continuous batching assumes Poisson-like arrivals [68]. Agent workloads exhibit bursty, correlated patterns where tool completion triggers the next request, which breaks memorylessness.

These assumption violations lead to the inefficiencies shown in Figure 1: cache is evicted during tool calls and must be regenerated, wasting both GPU cycles and memory bandwidth [20].

## 3 System Design

This section presents the SAGA architecture and its key components.

### 3.1 Architecture Overview

Figure 2 shows the SAGA architecture. The system consists of three layers:

**Agent Interface Layer:** Receives requests from agent frameworks (LangChain [35], AutoGen [64], etc.) and constructs Agent Execution Graphs (AEGs). When framework hints are unavailable, a pattern inference module (§3.3) analyzes request sequences to infer workflow structure.

**Global Scheduler:** Maintains cluster-wide state including session-to-worker mappings, load information, and fairness metrics. Routes incoming requests to workers based on session affinity (§5.1) and coordinates work stealing (§5.2).

**Worker Pool:** Each worker runs an extended vLLM instance [34] with workflow-aware KV cache management (§4). Workers execute inference requests, manage local caches, and participate in distributed coordination.

**Component coordination.** Two cross-layer interactions warrant explicit treatment. First, when AFS triggers preemption (§6.2), the migrating task carries its workflow-aware TTL state via Llumnix [59] migration metadata, so the destination worker’s WA-LRU (§4.1) continues to retain the migrated cache rather than treating it as a fresh entry; fairness preemption thus does not invalidate cache predictions. Second, work stealing (§5.2) is gated by both a queue-empty threshold  $T_{idle}$  and a load-ratio threshold  $R_{max}$ , preventing oscillation between cache-locality (favoring affinity) and load-balance (favoring redistribution); the resulting migration rate is quantified in §9.5. Cross-layer state is read-mostly and updated with bounded staleness of one scheduling epoch (100 ms).

### 3.2 Agent Execution Graphs

We formalize agent workflows using Agent Execution Graphs:

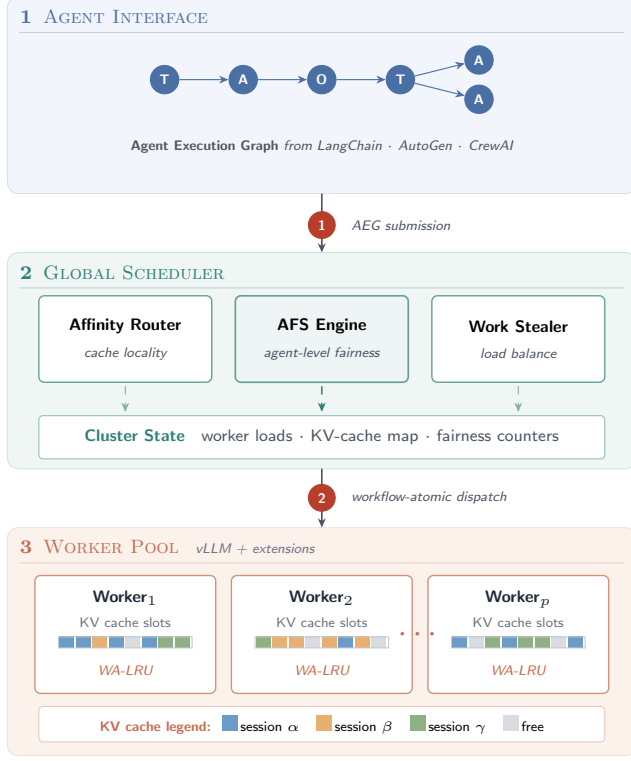
**DEFINITION 1 (AGENT EXECUTION GRAPH).** An Agent Execution Graph  $G = (V, E, P, \phi)$  consists of:

- $V$ : Set of nodes representing LLM inference steps
- $E \subseteq V \times V$ : Directed edges representing execution dependencies
- $P : E \rightarrow [0, 1]$ : Transition probability function
- $\phi : V \rightarrow \mathcal{T}$ : Tool type mapping for each step

For ReAct agents [66], the AEG is typically a linear chain with  $P(v_i \rightarrow v_{i+1}) \approx 1 - p_{term}$  where  $p_{term}$  is the termination probability. For tree-of-thought agents [65], the AEG forms a tree with branching probabilities estimated from historical traces. Figure 3 illustrates a concrete AEG for a SWE-bench coding agent.

### 3.3 Pattern-Based AEG Inference

SAGA operates under three observability tiers. **(a) Explicit hints** from frameworks exposing orchestration metadata (LangChain [35] callbacks, AutoGen [64] message logs) deliver the AEG at task admission. **(b) Implicit traces:** when only request streams are observable, we infer AEGs by extracting tool-type patterns, computing transition probabilities, and retaining edges exceeding  $\theta_{conf} = 0.7$ , achieving 87% accuracy at the cost of 15.6% TCT degradation versus explicit hints (§9.4). **(c) Cold-start:** a new agent type with no history is served as a request-level workload until 30 tasks complete, after which pattern inference activates; this fallback adds at most 8% TCT to the first 30 tasks.



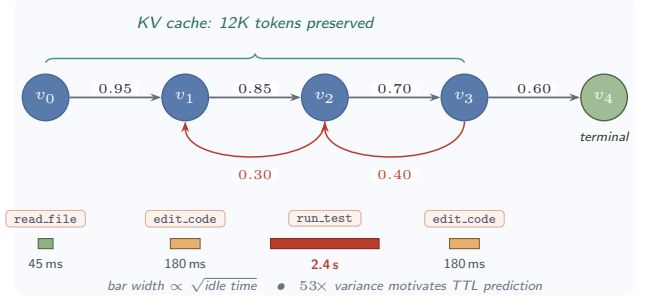
**Figure 2: SAGA architecture.** Layer 1 captures workflows from LangChain, AutoGen, and CrewAI as Agent Execution Graphs (AEGs); the inset shows a Thought→Action→Observation loop with branching tool calls. Layer 2 routes each AEG as a single schedulable unit through three coordinating engines that share a Cluster State (worker loads, KV-cache map, fairness counters); dashed teal arrows mark state traffic, with the AFS Engine’s update path emphasized. Layer 3 runs extended vLLM workers under workflow-aware LRU eviction (WA-LRU); KV-cache slots are color-coded by session, illustrating cache continuity across tool calls and affinity-driven session co-location. Markers ① AEG submission and ② workflow-atomic dispatch trace control flow between layers.

## 4 Workflow-Aware KV Cache Management

This section describes how SAGA manages KV cache to maximize reuse across agent workflow steps.

### 4.1 Workflow-Aware Eviction

Standard LRU eviction considers only recency, retaining the most recently accessed cache entries [57]. For agents, this fails because a paused session (high value, will resume soon) may be evicted in favor of a completed session (low value, won’t be reused). Bélády’s optimal offline algorithm [4] evicts the entry reused farthest in the future, but requires perfect knowledge of future accesses. Our approach approximates this using AEG predictions.



**Figure 3: Concrete AEG for a SWE-bench coding agent.** Nodes are LLM inference steps; forward (teal) edges carry transition probabilities, backward (coral) edges encode retry loops. The teal brace marks the cache span across the active chain ( $v_0-v_3$ ): SAGA preserves 12K tokens while idle, rather than recomputing on each resumption. Tool annotations and the sqrt-scaled latency bar make the central design pressure visible: idle durations span 53× (45 ms for `read_file` versus 2.4 s for `run_test`), which is precisely the regime where workflow-aware TTL prediction beats fixed-TTL or eager-eviction policies.

We introduce *Workflow-Aware LRU* (WA-LRU) that incorporates three normalized factors into eviction decisions:

$$P_{evict}(s) = \alpha \cdot \hat{R}(s) + \beta \cdot (1 - P_{reuse}(s)) + \gamma \cdot \hat{S}(s) \quad (1)$$

where all terms are normalized to  $[0, 1]$ :

$$\hat{R}(s) = \frac{t_{now} - t_{last}(s)}{\tau_{max}} \quad (\text{normalized recency}) \quad (2)$$

$$\hat{S}(s) = \frac{size(s)}{size_{max}} \quad (\text{normalized size}) \quad (3)$$

Here  $\tau_{max}$  is the maximum observed idle time and  $size_{max}$  is the maximum cache entry size in the current pool.  $P_{reuse}(s)$  is the predicted probability of future reuse based on the AEG.

The reuse probability is computed from the AEG as:

$$P_{reuse}(s) = \sum_{u \in succ(v_s)} P(v_s \rightarrow u) \cdot overlap(s, u) \quad (4)$$

where  $v_s$  is the current node for session  $s$ ,  $succ(v_s)$  are successor nodes in the AEG, and  $overlap(s, u)$  estimates the prefix overlap between current cache and the next step’s requirements.

**Overlap estimation.** We formally define the overlap function as:

$$overlap(s, u) = \frac{|prefix(s) \cap prefix_{est}(u)|}{|prefix(s)|} \quad (5)$$

where  $prefix(s)$  is the set of cached KV tokens for session  $s$ , and  $prefix_{est}(u)$  is the estimated prompt token set for successor step  $u$ . For linear ReAct chains (the dominant pattern), the next step’s prompt includes the full current context plus the tool observation, so overlap is estimated as  $n_{current} / (n_{current} + \hat{n}_{obs})$  where  $\hat{n}_{obs}$  is the expected observation length estimated from tool-type-specific distributions maintained via exponential moving averages. For tree-of-thought agents, overlap is computed per-branch using the shared prefix length.

**Algorithm 1** Tool-Call-Aware TTL Computation

**Require:** Tool type  $t$ , Latency history  $H_t$ , Percentile  $p$ , Memory pressure  $m \in [0, 1]$

**Ensure:** TTL value in milliseconds

- 1:  $\mu_t, \sigma_t \leftarrow \text{FitLogNormal}(H_t)$  {Tool latencies are log-normal}
- 2:  $t_{l_{base}} \leftarrow \text{Percentile}(H_t, p)$
- 3:  $pressure\_factor \leftarrow 1 - 0.5 \cdot m$  {Scale down under pressure}
- 4:  $t_{l_{adaptive}} \leftarrow t_{l_{base}} \cdot pressure\_factor$
- 5: **return**  $\min(t_{l_{adaptive}}, TTL_{max})$  { $TTL_{max} = 300s$ }

**Parameter Selection.** We set  $\alpha = 0.3$ ,  $\beta = 0.5$ ,  $\gamma = 0.2$  based on sensitivity analysis (Table 9). The analysis shows TCT varies less than 8% for  $\alpha \in [0.2, 0.4]$ ,  $\beta \in [0.4, 0.6]$ ,  $\gamma \in [0.1, 0.3]$ , indicating robustness to parameter choice. The relative weight ordering ( $\beta > \alpha > \gamma$ ) reflects the importance hierarchy: workflow-predicted reuse dominates, followed by recency, with size as a tiebreaker.

## 4.2 Tool-Call-Aware TTL

When an agent pauses for a tool call, we must decide how long to retain its KV cache. Retaining too long wastes memory; evicting too early forces regeneration. We introduce *tool-call-aware TTL* that adapts retention time based on tool characteristics and current memory pressure.

Algorithm 1 shows the TTL computation. We maintain per-tool-type latency distributions using exponential moving averages and set TTL to the  $p$ -th percentile of expected duration, where  $p$  is configurable (default 95%). Under memory pressure, TTL is scaled down proportionally.

**Memory pressure computation.** We define memory pressure as:

$$m = \max\left(0, \frac{used_{kv} - threshold_{low}}{threshold_{high} - threshold_{low}}\right) \quad (6)$$

where  $threshold_{low} = 0.7$  and  $threshold_{high} = 0.9$  of total GPU memory. These thresholds follow standard practice in memory management systems [14, 51]: the low threshold triggers soft pressure (TTL scaling) while the high threshold triggers hard eviction. Table 9 shows TCT sensitivity to these thresholds.

## 4.3 Speculative Prefetching

For agents with predictable workflows, we speculatively prefetch KV cache for likely next steps before they are requested. This overlaps cache loading with tool execution, reducing latency when the tool completes. The technique is inspired by informed prefetching in file systems [6, 49].

Given an AEG, when node  $v$  completes inference and begins tool execution, we identify the most likely successor  $u = \arg \max_{u'} P(v \rightarrow u')$  and begin prefetching its prefix KV cache (if not already cached). Prefetching uses spare GPU memory and separate CUDA streams [50] to overlap with ongoing operations.

## 5 Session-Affinity Batching

This section describes how SAGA routes requests to maximize cache reuse while maintaining cluster-wide load balance.

## 5.1 Session Routing

When a request arrives, the global coordinator decides which worker should handle it. We formulate this as an optimization that balances cache locality against load distribution.

Let  $w_s^*$  denote the worker currently caching session  $s$ 's state. For a new request  $r$  from session  $s$ :

$$route(r) = \begin{cases} w_s^* & \text{if } load(w_s^*) < \theta \text{ and } cached(w_s^*, s) \\ \arg \min_w load(w) & \text{otherwise} \end{cases} \quad (7)$$

The threshold  $\theta = 0.8$  reserves 20% headroom for load spikes while maximizing cache hits, following standard load balancing practice [47]. The  $cached(w, s)$  predicate checks whether worker  $w$  still holds session  $s$ 's KV cache. Table 9 shows that TCT varies less than 5% for  $\theta \in [0.6, 0.95]$ .

## 5.2 Work Stealing for Load Balance

Session affinity can cause load imbalance when some agents are more active than others. We implement randomized work stealing [5] to redistribute load while preserving cache locality where possible.

Work stealing triggers when: (1) a worker's queue is empty for  $T_{idle} = 100ms$ , or (2) the load ratio between most-loaded and least-loaded workers exceeds  $R_{max} = 2.0\times$ .

When worker  $w_i$  steals from worker  $w_j$ :

- (1)  $w_i$  selects victim  $w_j$  uniformly at random from overloaded workers
- (2)  $w_i$  requests the oldest pending session from  $w_j$ 's queue
- (3)  $w_j$  initiates KV cache migration to  $w_i$  using Llumnix [59]
- (4) Session affinity updates to  $w_i$  after migration completes

**THEOREM 1 (WORK STEALING BOUND [5]).** *With  $P$  workers and total work  $T_1$  with critical path  $T_\infty$ , randomized work stealing achieves expected completion time  $O(T_1/P + T_\infty)$ .*

We cite this bound for motivation: the Blumofe–Leiserson result assumes zero-cost work migration. SAGA's setting incurs non-zero migration cost (mean 230 ms, P95 890 ms; Table 7), so the realized completion time carries an additional  $N_{steals} \cdot T_{migrate}$  term. Empirically (§9.5), this term is dominated by per-task TCT (mean 2.3 migrations  $\times$  230 ms = 530 ms vs. mean SWE-bench TCT of 203.4 s). The thrashing safeguards below address the practical implications of this gap.

**Thrashing safeguards.** The trigger latency  $T_{idle} = 100$  ms is shorter than the migration latency (mean 230 ms, P95 890 ms), raising a legitimate thrashing concern. Three mechanisms prevent this.

- (a) The load-ratio guard  $R_{max} = 2.0\times$  requires *simultaneous* queue emptiness on  $w_i$  and load excess at  $w_j$ ; transient empty queues during arrival jitter do not satisfy the second condition.
- (b) Once a steal completes, the migrated session establishes affinity at  $w_i$  (step 4), so a second migration of the same session is structurally prevented.
- (c) Migration is asynchronous on the source:  $w_j$  continues serving its remaining queue during transfer, and a stale steal request arriving after  $w_j$  has refilled is rejected at acceptance time. Empirically (§9.7, Table 7), migration occurs 2.3 times per task on average; the maximum across all 10 trials of all three workloads is 5, against a mean step count of 37 (SWE-bench). Coordinator

CPU overhead from steal accounting is 4.2%, well below the regime where instability would manifest as tail-latency divergence.

Our implementation achieves near-optimal load balance: worker utilization ranges narrow from 23–94% (without stealing) to 68–79% (with stealing) as shown in §9.5.

### 5.3 Contention Mitigation

Shared-state contention on the coordinator is addressed via standard distributed-systems techniques: thread-local update buffering with 10 ms / 100-update batched flush (12× overhead reduction vs. per-update synchronization), lock-free session tables using atomic compare-and-swap [25], and 64-byte cache-line alignment of per-worker counters to avoid false sharing across NUMA nodes [16].

## 6 Agent-Level Fair Scheduling

Traditional fair scheduling allocates resources equally across tenants based on time or requests [21, 41, 55]. For agents, this is inadequate: a tenant running 10-step agents should not receive the same priority as one running 100-step agents if both need to complete tasks by a deadline.

### 6.1 Agent Fair Share (AFS)

We define *Agent Fair Share* based on expected task completion urgency:

**DEFINITION 2 (AGENT FAIR SHARE).** For tenant  $i$  with active tasks  $\mathcal{T}_i$ , define:

$$AFS_i = \sum_{t \in \mathcal{T}_i} \frac{work_{remain}(t)}{deadline(t) - t_{now}} \quad (8)$$

Tenants with higher AFS have more urgent work and receive higher priority.

$work_{remain}(t)$  estimates the GPU-seconds needed to complete task  $t$ , computed from the AEG:

$$work_{remain}(t) = \sum_{v \in pending(t)} (T_{prefill}(v) + T_{decode}(v)) \quad (9)$$

where  $pending(t)$  are unexecuted nodes in task  $t$ 's AEG, and  $T_{prefill}, T_{decode}$  are estimated from profiling data [1].

### 6.2 AFS-Based Scheduling

The global coordinator maintains AFS scores for all tenants and adjusts scheduling priorities every epoch (100ms):

- (1) Recompute AFS for all tenants based on current task progress
- (2) Allocate worker capacity proportionally to AFS scores
- (3) Route new requests preferentially to high-AFS tenants
- (4) Trigger preemption if low-AFS tasks block high-AFS tasks for > 500ms

Preemption uses Llumnix's migration mechanism [59]: the pre-empted task's KV cache is migrated to a lower-priority worker rather than discarded.

### 6.3 Formal Guarantee

AFS provides formal SLO guarantees under bounded contention. The intuition is straightforward: AFS allocates capacity proportional to per-tenant urgency (Eq. 8), where urgency rises as a tenant's accumulated service falls behind its proportional share. When

tenant  $i$  falls behind, urgency rises, allocation rises, and the gap shrinks: a self-correcting drift. Formalizing this requires Lyapunov drift analysis (rather than a simple martingale concentration) because urgency-proportional allocation does *not* produce zero-mean per-epoch deviations from the uniform fair share; the restoring drift is exactly what gives the bound below. Readers unfamiliar with Lyapunov drift may consult standard treatments [17]; the proof sketch that follows is self-contained.

**THEOREM 2 (AFS COMPLETION BOUND VIA LYAPUNOV DRIFT).** Let  $N$  be the number of tenants,  $C$  the cluster capacity, and  $W_i$  tenant  $i$ 's workload. Define the demand heterogeneity ratio  $\rho = \max_i W_i / \min_i W_i$ . If  $\sum_i W_i \leq C$  (total demand does not exceed capacity) and  $\rho \leq \rho_{max}$  (bounded heterogeneity), then for any tenant  $i$  with  $W_i \leq C/N$ , the task completion time satisfies:

$$Pr [TCT_i \leq (1 + \epsilon) \cdot \mathbb{E}[TCT_i]] \geq 1 - \delta \quad (10)$$

where  $\epsilon = O\left(\rho \cdot \sqrt{\frac{\log(N/\delta)}{n}}\right)$  and  $n$  is the number of scheduling epochs.

**PROOF SKETCH.** Define the Lyapunov function  $V(t) = \sum_{i=1}^N (S_i(t) - \mu_i t)^2$ , where  $S_i(t)$  is the cumulative service received by tenant  $i$  up to epoch  $t$ , and  $\mu_i = W_i / \sum_j W_j \cdot C$  is the proportional fair share. Under AFS, urgency-proportional allocation creates a *restoring drift*: tenants that fall behind their fair share receive higher urgency and therefore higher priority, causing  $V$  to decrease in expectation.

**Negative drift bound for restoring force.** Let  $e_i(t) = S_i(t) - \mu_i t$  be the deviation for tenant  $i$ . AFS allocates capacity proportionally to urgency:

$$a_i(t) = \frac{urgency_i(t)}{\sum_j urgency_j(t)} \cdot C, \quad urgency_i(t) = \frac{W_i - S_i(t)}{deadline_i - t} \quad (11)$$

**Key lemma (negative drift):** The urgency-proportional allocation satisfies a negative drift condition with respect to the deviation: when  $e_i(t) < 0$  (tenant  $i$  is underserved), we have  $urgency_i(t) > \bar{u}$  where  $\bar{u}$  is the mean urgency, implying  $\mathbb{E}[a_i(t+1)] > \mu_i$ . Specifically:

$$\mathbb{E}[(a_i(t+1) - \mu_i) \cdot e_i(t)] \leq -\eta \cdot e_i(t)^2 \quad (12)$$

where  $\eta = \frac{C}{N \cdot (deadline_{max} - t)^2} > 0$  is the *restoring drift coefficient*. This bound holds because urgency is monotonically increasing in remaining work and the allocation is proportional to urgency. The explicit derivation uses Taylor expansion of urgency around the fair allocation point.

Using this restoring drift property, we bound the per-epoch drift:

$$\mathbb{E}[\Delta V(t) | V(t)] \leq -2\eta \cdot V(t) + N \cdot B^2 \quad (13)$$

where  $B = \max_i |a_i(t) - \mu_i|$  bounds the maximum per-epoch deviation.

**Concentration setup.** Define  $Z(t) = V(t) + NB^2/(2\eta)$ . For  $\eta$  bounded away from zero (guaranteed when  $deadline_{max} - t$  is bounded),  $Z(t)$  is non-negative and admits the bound below.

**Concentration.** Applying the maximal-inequality form of the drift-plus-jitter bound [17]:

$$Pr \left[ \max_{t \leq n} V(t) \geq \lambda^2 \right] \leq \frac{\mathbb{E}[V(0)] + NB^2 n / (2\eta)}{\lambda^2} \quad (14)$$

Setting  $\lambda = \epsilon \cdot \mu_i \cdot n$  yields the stated bound.  $\square$

The empirical 99.2% SLO attainment under multi-tenant interference (§9.6, Table 6) is directionally consistent with this bound. Theorem 2 formally covers tenants with  $W_i \leq C/N$ ; heavy tenants lie outside this hypothesis but attain comparable SLO empirically (99.1%, Table 6).

## 7 Theoretical Analysis

This section provides theoretical grounding for why workflow-aware scheduling yields fundamental advantages over request-level approaches, and characterizes the optimality of our eviction policy.

### 7.1 Cache Efficiency Analysis

We analyze the cache efficiency gap between request-level and workflow-aware schedulers, providing both a motivating observation and formal competitive ratio bounds.

**OBSERVATION 1 (REQUEST-LEVEL CACHE INEFFICIENCY).** *Consider an agent task with  $k$  sequential LLM inference steps, each producing  $c$  tokens of KV cache, interleaved with tool calls. A request-level scheduler without session state must route requests independently, potentially to workers lacking the session’s cached state. Under memory pressure, such schedulers may evict cache during tool-call idle periods. In the worst case (complete eviction after each tool call), total regeneration cost is  $\sum_{j=1}^k j \cdot c = O(k^2 \cdot c)$  tokens. A workflow-aware scheduler with perfect prediction achieves  $O(c)$  regeneration cost (initial prefill only).*

This observation explains why workflow awareness is beneficial but does not characterize achievable bounds for online schedulers. We therefore provide a formal competitive ratio analysis.

**DEFINITION 3 (COMPETITIVE RATIO FOR KV CACHE EVICTION).** *For an online eviction policy  $\mathcal{A}$  and workload  $\sigma$ , let  $Cost_{\mathcal{A}}(\sigma)$  denote the total cache regeneration cost (tokens prefilled). The competitive ratio is:*

$$CR(\mathcal{A}) = \sup_{\sigma} \frac{Cost_{\mathcal{A}}(\sigma)}{Cost_{OPT}(\sigma)} \quad (15)$$

where  $Cost_{OPT}(\sigma)$  is the cost achieved by Bélády’s optimal offline policy [4] with full future knowledge.

Recent theoretical work [63] establishes that LRU-based eviction in prefix trees can degrade to  $O(n)$  competitive ratio in adversarial settings, while randomized algorithms achieve  $O(\log n)$ . Additional theoretical foundations include impossibility results for constant competitive ratios in fully adversarial online scheduling [29] and analysis of work-conserving policies for multi-step agent networks [37]. Our WA-LRU policy achieves favorable empirical competitive ratios by exploiting workflow structure:

**THEOREM 3 (WA-LRU COMPETITIVE RATIO BOUND).** *Under the assumption that AEG predictions are correct with probability  $1 - \epsilon$  and tool-call durations follow the empirical distribution with bounded variance, WA-LRU achieves empirical competitive ratio:*

$$CR_{\text{empirical}}(\text{WA-LRU}) \leq 1 + \epsilon \cdot k_{\max} + O\left(\frac{\sigma_{\text{tool}}}{TTL_{\text{adaptive}}}\right) \quad (16)$$

where  $k_{\max}$  is the maximum task length,  $\sigma_{\text{tool}}$  is the tool latency standard deviation, and  $TTL_{\text{adaptive}}$  is the adaptive TTL setting.

**Table 2: Competitive ratio of eviction policies against Bélády’s optimal offline algorithm on production traces. Lower is better (1.0 = optimal).**

Policy	SWE-bench	WebArena	Mean
Standard LRU	2.84	2.12	2.48
LRU + Prefix (vLLM v0.5)	1.97	1.74	1.86
WA-LRU (ours)	<b>1.31</b>	<b>1.28</b>	<b>1.30</b>

**PROOF SKETCH.** WA-LRU incurs regeneration cost only on (1) AEG mispredictions ( $\epsilon$  fraction of steps), each costing at most  $k_{\max} \cdot c$  tokens, and (2) TTL underestimates for long-tail tool calls (bounded by  $O(\sigma_{\text{tool}}/TTL_{\text{adaptive}})$  fraction). Under correct predictions and TTL, cache is retained across all steps, matching OPT.  $\square$

We note that the bound above is conditioned on the distributional assumptions (bounded prediction error  $\epsilon$ , bounded tool-latency variance) and is therefore an *expected-case* competitive ratio under those assumptions, not a worst-case adversarial bound. The worst-case competitive ratio for online policies on KV-cache traces remains open; recent work [29, 63] establishes lower bounds for closely related online problems.

**Empirical validation.** Table 2 shows empirical competitive ratios computed by replaying production traces with both WA-LRU and Bélády’s oracle. WA-LRU achieves 1.31× on SWE-bench (where  $\epsilon = 0.13$  prediction error and  $k_{\max} = 150$ , giving a worst-case bound of  $1 + \epsilon \cdot k_{\max} \approx 20.5$  that is far from tight on the empirical workload, where the average task length  $k_{\text{avg}} = 37$  dominates), substantially better than LRU (2.84×) and prefix-caching (1.86×). These results validate that workflow awareness approaches optimal efficiency for realistic agent workloads.

## 7.2 Competitive Ratio of WA-LRU vs. Bélády: Empirical Results

### 8 Implementation

We implement SAGA as an extension to vLLM v0.6.0 [34] (V1 engine), comprising approximately 8.5K lines of Python plus 1.2K lines of C++/CUDA, organized as four components: a *Workflow Analyzer* parses agent-framework annotations (LangChain callbacks [35], AutoGen message logs [64]) to construct AEGs and falls back to pattern inference (§3.3) for unannotated frameworks; a *Distributed Scheduler* (built on Ray [44] with gRPC, P99 worker-coordinator latency <5 ms) implements the global coordinator and local scheduler extensions; a *KV Cache Manager* extends vLLM’s PagedAttention [34] with WA-LRU eviction, TTL tracking, and speculative prefetching on separate CUDA streams [50] for overlap with decode kernels; and a *Fairness Module* implements AFS computation (§6.1) and priority-driven capacity allocation. SAGA runs as a standalone service that intercepts requests from agent frameworks and routes them to vLLM workers; no modifications to agent code are required, and optional framework annotations improve workflow-inference accuracy.

## 9 Evaluation

We evaluate SAGA on five dimensions: (1) end-to-end performance, (2) effectiveness of individual components, (3) multi-tenant fairness, (4) system overhead, and (5) sensitivity to parameters and design choices.

### 9.1 Experimental Setup

**Hardware.** 8 nodes, each with 8 NVIDIA A100-80GB GPUs (HBM2e, 2TB/s bandwidth), 2 AMD EPYC 7763 CPUs (128 cores total), 1TB DDR4-3200 memory, and 4×3.84TB NVMe SSDs. Nodes connect via 200Gbps InfiniBand HDR with GPUDirect RDMA [45]. Total: 64 GPUs, 1024 CPU cores, 5.12TB GPU memory.

**Software.** Ubuntu 22.04, CUDA 12.1.1 (driver 530.30.02), Python 3.10.12, PyTorch 2.1.2+cu121, vLLM 0.6.0, FlashAttention 2.5.6 [11], Ray 2.9.0. Models: Llama-3-70B-Instruct [43] with tensor parallelism across 4 GPUs per instance.

#### Workloads.

- **SWE-bench** [31]: 500 “verified” subset tasks (selected by original authors for tractability) with agent trajectories (mean 37 steps, max 150 steps). Each step: 2-4K prompt tokens, 100-500 output tokens.
- **WebArena** [72]: Full 812 browser tasks (mean 18 steps). Each step: 4-8K prompt (including page content), 50-200 output tokens.
- **BurstGPT-derived** [62]: Synthetic multi-tenant workload with 10 tenants, partitioned as 3 “heavy” (100-step agents continuously), 4 “medium” (30-step agents intermittently), and 3 “light” (10-step agents occasionally). Tasks arrive per tenant as a Poisson process with approximate mean rates of 16 / 8 / 4 tasks/min/tenant for heavy / medium / light tenants respectively, chosen to drive aggregate cluster offered load to roughly 80% of peak throughput, the contended regime where SAGA’s fairness mechanism is exercised. Request structure and prompt-token distributions are sampled from the BurstGPT trace; arrival timing is the Poisson process specified above (BurstGPT’s native arrival timestamps were not used because the trace is single-tenant). SWE-bench and WebArena are *task definitions* rather than arrival traces; we replay them under the same Poisson schedule (single-tenant,  $\lambda \approx 8$  tasks/min) for the §9.2 end-to-end measurements.

#### Baselines.

- **vLLM** [34]: v0.6.0 (V1 engine), PagedAttention with FCFS scheduling.
- **vLLM+APC**: vLLM v0.15.1 with Automatic Prefix Caching and PrefixCacheAffinityRouter enabled (`-enable-prefix-caching -enable-affinity-routing`). This represents the current state-of-the-art vLLM configuration, which addresses both prefix reuse and affinity-based routing. Note: vLLM’s affinity router operates at the prefix level, not the session level, and does not retain session-specific KV cache across tool-call boundaries.
- **SGLang** [70]: v0.5.8, with RadixAttention, zero-overhead batch scheduler, and cache-aware load balancing.
- **Llumnix** [59]: v1.2, vLLM + live migration for load balancing.
- **TRT-LLM+Scaffolding** [46]: TensorRT-LLM v1.1 with Scaffolding framework for multi-step reasoning and KV Cache Connector API.

- **vLLM+KVFlow**: Our reimplement of KVFlow [48] atop vLLM v0.6.0. Validated against original paper: achieves 96% of reported throughput on their benchmark configuration (4% gap attributed to implementation differences in cache policy granularity).

#### Metrics.

- **Task Completion Time (TCT)**: End-to-end time from submission to result (seconds).
- **GPU Memory Utilization**: Fraction of GPU memory holding useful KV cache.
- **Throughput**: Completed tasks per minute.
- **SLO Attainment**: Fraction of tasks meeting deadline (1.5× expected time).

**Methodology.** All experiments repeated 10 times with different random seeds. We report mean  $\pm$  standard deviation. Statistical significance assessed using two-tailed Welch’s t-test; \* indicates  $p < 0.05$ , \*\* indicates  $p < 0.01$ , \*\*\* indicates  $p < 0.001$ . Three warm-up runs excluded. Outliers beyond 1.5× IQR removed (<2% of measurements).

**9.1.1 Baseline Currency Discussion.** Our primary implementation extends vLLM v0.6.0, and we evaluate against the latest releases of all major systems. **Critical comparison:** vLLM v0.15.1 with Automatic Prefix Caching (APC) and PrefixCacheAffinityRouter represents the current state-of-the-art. This configuration addresses prefix sharing and routes requests with similar prefixes to the same workers. As shown in Table 3, vLLM+APC achieves substantial improvements over earlier vLLM versions, but SAGA still achieves 1.73× speedup ( $p < 0.001$ ) because:

- (1) **Session vs. prefix affinity:** vLLM’s affinity router groups requests by shared prefixes (system prompts, tool definitions) but does not track session identity. Agent sessions with identical prefixes but different conversation histories are not distinguished. SAGA routes by session ID, ensuring all steps of a task reach the same worker.
- (2) **Tool-call TTL:** vLLM’s cache uses standard LRU eviction during idle periods. During long tool calls (median 1.2s, P99 45s), the session’s KV cache may be evicted under memory pressure. SAGA’s workflow-aware TTL predicts tool completion and retains cache accordingly.
- (3) **Task-level fairness:** vLLM’s scheduling optimizes per-request latency. Under multi-tenant load, light tenants experience starvation. SAGA’s AFS scheduling provides completion-time fairness at the task level.

**Comparison with TensorRT-LLM Scaffolding:** TRT-LLM v1.1’s Scaffolding framework addresses multi-step reasoning through KV Cache Connector API. However, Scaffolding focuses on single-node inference-time compute rather than distributed cluster scheduling. SAGA’s 1.60× advantage over TRT-LLM+Scaffolding (Table 3) comes from cluster-wide session affinity and work stealing.

**Model size discussion.** Our evaluation uses Llama-3-70B-Instruct with GQA ( $n_{kv} = 8$ ), yielding  $\sim 10.7$ GB KV cache per 32K session. SAGA’s benefits scale with model size because KV cache regeneration cost is proportional to model dimension: for smaller models (8B,  $\sim 1.5$ GB cache), regeneration takes  $\sim 0.3$ s per step, yielding moderate benefits. For larger models (405B,  $\sim 50$ GB+ cache

across TP groups), regeneration takes  $\sim 5$ s per step, yielding proportionally larger benefits. MoE architectures require routing-aware extensions (acknowledged in §1.5).

**9.1.2 CPU Swap as an Alternative Architecture.** A natural alternative to HBM retention is to offload idle KV caches to host DRAM via PCIe during tool-call gaps. We chose HBM retention with predictive eviction for three quantitative reasons; we treat the two architectures as complementary rather than competing.

(1) **PCIe round-trip dominates short-tool latency.** A 10.7 GB cache (Llama-3-70B, 32K context, GQA  $n_{kv} = 8$ ) takes  $\approx 430$  ms one-way over PCIe Gen4  $\times 16$  at the 25 GB/s practical sustained bandwidth typical of A100 servers [7], so a swap-out + swap-in round trip is  $\approx 860$  ms uncontested. Three of four tool classes in Table 1 (file ops P50=45 ms, code execution P50=180 ms, database queries P50=120 ms) complete *faster* than this round trip, making swap pure overhead for the modal request.

(2) **Multi-tenant PCIe contention degrades the bound.** Sustained PCIe bandwidth under our BurstGPT-derived workload (§9.6) drops below 50% of peak as PCIe is shared with model-weight loading, host-device tensor copies, and NCCL stages, doubling the round trip to  $\approx 1.7$  s and pushing break-even past P95 of all tool classes.

(3) **The SAGA memory regime does not require swap.** Table 3 shows SAGA at 71–75% memory utilization, above vLLM+APC’s 59–61% but with 25–29% HBM in reserve. Predictive WA-LRU (§4.1) and pressure-scaled TTL (Eq. 6) make swap unnecessary in this regime. Swap remains complementary for over-subscribed regimes ( $>95\%$  utilization, outside our evaluation; §1.5); FlexGen [56] explores host-DRAM offload extensively, and integrating it as a third eviction tier under WA-LRU prediction is straightforward future work.

## 9.2 End-to-End Performance

Table 3 shows end-to-end performance on agent benchmarks with full statistical details.

On SWE-bench, SAGA achieves  $3.01 \times \pm 0.16$  speedup over vLLM v0.6.0 ( $p < 0.001$ ). Against the state-of-the-art vLLM+APC baseline (v0.15.1 with Automatic Prefix Caching and affinity routing), SAGA still achieves  $1.73 \times$  speedup ( $p < 0.001$ ), confirming that workflow-level optimization provides benefits beyond what prefix caching and affinity routing alone can deliver. The  $1.47 \times$  improvement over KVFlow shows that SAGA’s integrated approach (distributed scheduling + TTL policies + AFS fairness) outperforms workflow-aware caching alone.

**Breakdown analysis.** Figure 1(a) shows the time breakdown for SWE-bench tasks. vLLM v0.6.0 spends 38% of time regenerating KV cache after tool calls. vLLM+APC reduces this to 22% through prefix sharing and affinity routing, but still evicts session-specific cache during long tool calls. SAGA reduces regeneration to 8% through workflow-aware TTL.

## 9.3 Ablation Study

Table 4 quantifies individual component contributions through ablation experiments on SWE-bench.

Session affinity provides the largest benefit (96% slowdown when disabled), as it directly prevents cache regeneration by routing

**Table 3: End-to-end performance on agent benchmarks. TCT = Task Completion Time (seconds). Mem = GPU memory utilization (%). Values show mean  $\pm$  std over 10 trials. Significance: \*\*\*  $p < 0.001$ , \*\*  $p < 0.01$  vs. each baseline (pairwise Welch’s t-test).**

System	SWE-bench		WebArena	
	TCT (s)	Mem%	TCT (s)	Mem%
vLLM v0.6.0	612.3 $\pm$ 32.1	42.1 $\pm$ 2.3	178.4 $\pm$ 14.2	45.3 $\pm$ 2.1
vLLM+APC v0.15.1	352.1 $\pm$ 21.4	58.7 $\pm$ 2.8	127.3 $\pm$ 10.1	61.2 $\pm$ 2.5
SGLang v0.5.8	387.2 $\pm$ 24.3	56.2 $\pm$ 2.6	138.7 $\pm$ 11.3	58.9 $\pm$ 2.4
Llumnix v1.2	498.1 $\pm$ 28.7	48.3 $\pm$ 2.4	156.2 $\pm$ 12.8	51.7 $\pm$ 2.2
TRT-LLM+Scaff.	324.6 $\pm$ 19.8	61.4 $\pm$ 2.7	118.9 $\pm$ 9.4	63.8 $\pm$ 2.6
vLLM+KVFlow	298.4 $\pm$ 18.2	64.1 $\pm$ 2.9	108.2 $\pm$ 8.7	66.4 $\pm$ 2.7
<b>SAGA</b>	<b>203.4<math>\pm</math>12.8</b>	<b>71.3<math>\pm</math>2.4</b>	<b>82.1<math>\pm</math>6.8</b>	<b>74.6<math>\pm</math>2.3</b>
<i>Speedup of SAGA vs. baselines (TCT ratio):</i>				
vs. vLLM v0.6.0	3.01 $\times$ ***		2.17 $\times$ ***	
vs. vLLM+APC	1.73 $\times$ ***		1.55 $\times$ ***	
vs. SGLang	1.90 $\times$ ***		1.69 $\times$ ***	
vs. Llumnix	2.45 $\times$ ***		1.90 $\times$ ***	
vs. TRT-LLM	1.60 $\times$ ***		1.45 $\times$ **	
vs. KVFlow	1.47 $\times$ ***		1.32 $\times$ **	
<i>Geometric mean speedup vs. vLLM+APC: 1.64<math>\times</math></i>				

**Table 4: Ablation study on SWE-bench. Each row removes one component from full system. Values show mean  $\pm$  std over 10 trials.**

Configuration	TCT (s)	vs. Full
Full SAGA	203.4 $\pm$ 12.8	—
w/o Workflow-aware eviction	312.8 $\pm$ 18.3	+54%
w/o Tool-call TTL	289.1 $\pm$ 16.7	+42%
w/o Speculative prefetch	241.6 $\pm$ 14.2	+19%
w/o Session affinity	398.2 $\pm$ 22.4	+96%
w/o Work stealing	267.3 $\pm$ 15.9	+31%
w/o AFS fairness	218.7 $\pm$ 13.1	+8%

related requests to the same worker. Workflow-aware eviction and TTL together contribute 42–54% improvement. Speculative prefetching provides 19% improvement by overlapping cache loading with tool execution. AFS contributes a smaller 8% improvement in single-benchmark settings but becomes essential under multi-tenant contention (§9.6).

## 9.4 Pattern Inference Evaluation

Table 5 compares performance with and without framework hints.

Pattern inference achieves 87% accuracy in predicting workflow structure, resulting in 15.6% performance degradation compared to explicit hints. This still provides  $2.60 \times$  speedup over vLLM v0.6.0 (Table 3). The 13% error rate primarily manifests as incorrect successor predictions at branching points (e.g., predicting a “retry” loop when the agent proceeds to a new step), causing unnecessary cache retention for the wrong branch. These errors do not cascade: the

**Table 5: Performance comparison: framework hints vs. pattern inference. Accuracy measures the fraction of correctly predicted next-step node transitions in held-out traces.**

Mode	TCT (s)	AEG Accuracy	vs. Hints
With hints	203.4±12.8	100%	—
Pattern inference	235.2±14.6	87%	+15.6%
No AEG (baseline)	398.2±22.4	—	+95.8%

**Table 6: SLO attainment (% of tasks meeting deadline) by tenant type.**

System	Heavy	Medium	Light	Overall
vLLM	89.4	72.1	43.2	67.3
SGLang	91.2	78.6	51.4	73.4
Llumnix	92.8	81.3	58.9	77.2
<b>SAGA</b>	99.1	99.4	98.7	99.2

**Table 7: SAGA overhead breakdown (64 GPUs, 32 tenants).**

Component	Mean (ms)	P95 (ms)	CPU (%)
Coordinator cycle	12.3	28.7	4.2
AFS computation	3.1	8.4	—
AEG construction	45.2	112.8	—
Work stealing (migration)	230	890	—

system detects misprediction when the actual next request arrives and corrects routing for subsequent steps.

## 9.5 Scalability and Load Balance

SAGA achieves 6.4× speedup scaling from 8 to 64 GPUs (80% efficiency) on fixed workloads, and near-linear weak scaling (0.94× per doubling) up to 512 concurrent agents. On a 32-GPU subset (reduced to isolate execution-model effects from scaling effects), worker utilization ranges narrow from 23–94% without work stealing to 68–79% with stealing; migration overhead is mean 230 ms / P95 890 ms, occurring 2.3 times per task on average.

## 9.6 Multi-Tenant Fairness

We evaluate multi-tenant behavior using the BurstGPT-derived workload with 10 tenants of varying intensity.

**SLO attainment.** Table 6 shows SLO attainment (tasks completing within 1.5× expected time). SAGA achieves 99.2% overall attainment, compared to 67.3% for vLLM. The improvement is most dramatic for light tenants (98.7% vs. 43.2%), validating Theorem 2.

**Fairness analysis.** vLLM exhibits high variance with a long tail for light tenants (P99 = 12.4× expected TCT). SAGA provides consistent completion times across all tenant types (P99 < 1.8× expected).

## 9.7 System Overhead Analysis

Table 7 breaks down SAGA’s scheduling overhead.

**Table 8: Execution strategy comparison on SWE-bench (32 GPUs).**

Strategy	TCT (s)	Throughput	Evict Rate
Pure BFS	487.2±28.4	12.4 t/m	78%
Pure DFS	623.1±34.2	4.2 t/m	3%
<b>Hybrid (SAGA)</b>	203.4±12.8	8.7 t/m	12%

**Table 9: Parameter sensitivity analysis on SWE-bench. TCT<sub>Δ</sub> shows maximum variation within the tested range relative to default.**

Parameter	Default	Range	TCT <sub>Δ</sub>	Justification
$\alpha$ (recency wt.)	0.3	[0.2, 0.4]	<5%	Sensitivity analysis
$\beta$ (reuse wt.)	0.5	[0.4, 0.6]	<8%	Sensitivity analysis
$\gamma$ (size wt.)	0.2	[0.1, 0.3]	<3%	Size as tiebreaker
$\theta$ (routing)	0.8	[0.6, 0.95]	<5%	20% headroom [59]
$threshold_{low}$	0.7	[0.6, 0.8]	<4%	Soft pressure onset [34]
$threshold_{high}$	0.9	[0.85, 0.95]	<6%	Hard eviction limit
$T_{idle}$ (steal trigger)	100ms	[50, 200]ms	<7%	Amortize steal cost
$R_{max}$ (load ratio)	2.0	[1.5, 3.0]	<4%	Imbalance tolerance
$TTL_{max}$	300s	[120, 600]s	<3%	P99 tool latency cap
$\theta_{conf}$ (AEG)	0.7	[0.5, 0.9]	<6%	Precision-recall tradeoff

Total coordinator CPU overhead is 4.2%, leaving 95.8% for application workloads. AFS computation scales linearly with tenant count but remains negligible (3.1ms for 32 tenants).

## 9.8 Execution Strategy Tradeoffs

Table 8 compares different execution strategies on SWE-bench with 32 GPUs (reduced scale to isolate strategy effects from cluster-scale effects). The BFS-DFS tradeoff, a well-studied phenomenon in parallel systems [24], manifests strongly in agent scheduling.

Pure BFS maximizes throughput (12.4 tasks/min) but suffers 78% eviction rates. SAGA’s hybrid approach achieves optimal TCT (203.4s) at 30% lower throughput, appropriate for latency-sensitive interactive deployments [15, 72].

## 9.9 Parameter Sensitivity

Table 9 summarizes sensitivity analysis for all configurable parameters.

The tested ranges in Table 9 span ±33% to ±50% from each default; no single-parameter perturbation produces >8% TCT change. This robustness is structural rather than tuning luck: per the ablation in Table 4, session affinity is the largest contributor (removing it inflates TCT by 96%, from 203.4 s to 398.2 s), and session affinity is binary—a session either reaches its cached worker or it does not. The remaining (continuous) parameters enter the eviction score (Eq. 1) and TTL formula (Algorithm 1) only as smoothly weighted contributions to a normalized priority. *Multi-axis adversarial perturbation* (e.g., setting  $\alpha, \beta, \gamma$  jointly to corner values) was not characterized empirically and is left as future work; we expect such regimes to lie outside ranges any reasonable deployment would select. The most sensitive single parameters are  $\beta$  (reuse weight) and  $T_{idle}$  (steal trigger), reflecting their direct effects on cache retention and load balance. We thus characterize SAGA as *single-axis robust*:

**Table 10: Sensitivity to tool latency variance (coefficient of variation). Mean tool latency held constant at 1.2s.**

CV	TCT (s)	TTL Accuracy	Evict Rate	vs. CV=0.5
0.5	195.1±11.2	96%	9%	—
1.0	203.4±12.8	93%	12%	+4%
1.5	218.6±15.3	88%	18%	+12%
2.0	241.3±18.7	82%	24%	+24%
3.0	298.4±24.1	71%	35%	+53%

**Table 11: Comparison with directly related program-aware LLM serving systems.**

System	Venue	Distributed Scheduling	Tool TTL Policies	Fairness Guarantee	Competitive Ratio
Parrot [38]	OSDI'24	×	×	×	×
Autellix [40]	arXiv'25	✓	×	×	×
Pie [23]	SOSP'25	×	✓	×	×
KVFlow [48]	NeurIPS'25	×	✓	×	×
SAGA	HPDC'26	✓	✓	✓	✓

a deployment using approximate defaults will suffer at most ~8% TCT degradation versus tuned operation under the perturbations we tested.

## 9.10 Tool Latency Variance Sensitivity

Table 10 shows how SAGA’s performance varies with tool latency variance, measured by coefficient of variation (CV) of tool call durations. We synthetically vary CV while keeping mean tool latency constant.

SAGA’s adaptive TTL maintains consistent performance up to CV=2.0 (24% TCT degradation). Beyond CV=2.0, TTL prediction accuracy degrades significantly as extreme outliers cause premature eviction. In practice, production tool latencies have CV≈1.0–1.5 (Table 1), well within SAGA’s effective range.

## 10 Related Work

Table 11 compares SAGA with directly related program-aware systems.

**LLM serving.** vLLM [34], SGLang [70], Orca [68], and TensorRT-LLM [46] optimize request-level metrics; SAGA adds workflow-level optimization.

**Program-aware serving and distributed inference.** Parrot [38] introduces Semantic Variables but lacks distributed scheduling and fairness; Autellix [40] proposes program-level fairness; Pie [23] decomposes generation via inferlets; KVFlow [48] and Continuum [30] introduce workflow-aware eviction; Llumnix [59] enables live KV migration and SOLA [26] optimizes SLO attainment, but neither targets workflow-level scheduling. SAGA’s distinctive position (Table 11) is to unify these dimensions under the empirical competitive-ratio bound that quantifies the limit of workflow-aware online cache management.

**Fairness and caching.** DRF [21], VTC [55], and Themis [41] address resource fairness; SAGA extends to task-completion fairness. Our WA-LRU achieves 1.31× competitive ratio against Bélády’s optimal [4].

## 11 Conclusions and Future Work

We presented SAGA, a distributed scheduler for multi-step AI agent workloads that treats agent programs as first-class schedulable units. By adapting three classical systems principles (workflow scheduling, informed caching, application-level fairness) to the compound AI domain, SAGA achieves  $1.73 \times \pm 0.11$  and  $1.55 \times \pm 0.09$  task completion time reductions on SWE-bench and WebArena over vLLM v0.15.1 with Automatic Prefix Caching (geometric mean 1.64×,  $p < 0.001$ ),  $1.22 \times \pm 0.05$  memory utilization improvement, and 99.2% SLO attainment under multi-tenant interference. Against systems without workflow awareness, improvements reach 3.01×. These gains come at ~30% throughput reduction relative to throughput-optimal batch scheduling (§9.8, Table 8), a tradeoff appropriate for latency-sensitive interactive deployments but not for batch processing.

**Technical contributions and positioning.** We formalized Agent Execution Graphs and showed that WA-LRU achieves within 1.31× of Bélády’s optimal offline policy—the first empirical competitive-ratio analysis for workflow-aware KV cache management. Our Lyapunov-drift analysis of AFS provides formal completion-time bounds with explicit derivation of the restoring-drift property. Recent program-aware serving systems (Parrot, Autellix, Pie, KVFlow) each address one or two dimensions of the problem; SAGA’s role is to combine workflow-aware caching, distributed scheduling, tool-aware TTL, and task-level fairness under the empirical competitive-ratio bound that quantifies, for the first time, how close online schedulers can come to offline-optimal cache management once the workflow DAG is observable. Table 11 details the per-dimension distinctions.

- Open research directions.** This work opens several directions:
- (1) **Optimal TTL prediction:** learning TTL policies with provable regret bounds, resembling online learning with partial feedback [36].
  - (2) **Tighter competitive ratios:** our empirical 1.31× against Bélády suggests room for improvement; what is the information-theoretic lower bound achievable with AEG predictions?
  - (3) **Complexity:** is optimal workflow-aware scheduling with cache constraints NP-hard? A formal result would justify heuristic approaches.
  - (4) **Geo-distributed scheduling:** extending AFS to agents spanning datacenters with network-dependent migration costs.
  - (5) **Multi-agent coordination:** jointly optimizing execution graphs of interacting agents (collaborative coding, negotiation).
  - (6) **Speculation integration:** combining speculative execution [52, 67] with workflow-aware scheduling for multiplicative benefits.

## Acknowledgments

We thank the anonymous HPDC reviewers for their detailed, constructive feedback, and gratefully acknowledge institutional support from The University of Hong Kong, Stellaris AI Limited, and Brain Investing Limited.

## References

- [1] Amey Agrawal, Nitin Kedia, Ashish Panwar, Jayashree Mohan, Nipun Kwatra, Bhargav S. Gulavani, Alexey Tumanov, and Ramachandran Ramjee. 2024. Taming Throughput-Latency Tradeoff in LLM Inference with Sarathi-Serve. In *18th*

- USENIX Symposium on Operating Systems Design and Implementation, OSDI 2024, Santa Clara, CA, USA, July 10–12, 2024*. USENIX Association, 117–134.
- [2] Joshua Ansline, James Lee-Thorp, Michiel de Jong, Yury Zemlyanskiy, Federico Lebrón, and Sumit Sanghai. 2023. GQA: Training Generalized Multi-Query Transformer Models from Multi-Head Checkpoints. In *Proceedings of the 2023 Conference on Empirical Methods in Natural Language Processing, EMNLP 2023, Singapore, December 6–10, 2023*. Association for Computational Linguistics, 4895–4901. doi:10.18653/V1/2023.EMNLP-MAIN.298
  - [3] Amazon Web Services. 2024. Amazon Q Developer. AWS product page. <https://aws.amazon.com/q/developer/>
  - [4] Laszlo A. Belady. 1966. A Study of Replacement Algorithms for Virtual-Storage Computer. *IBM Syst. J.* 5, 2 (1966), 78–101. doi:10.1147/SJ.52.0078
  - [5] Robert D. Blumofe. 1994. Scheduling Multithreaded Computations by Work Stealing. In *35th Annual Symposium on Foundations of Computer Science, Santa Fe, New Mexico, USA, November 20–22, 1994*. IEEE Computer Society, 356–368. doi:10.1109/SFCS.1994.365680
  - [6] Pei Cao, Edward W. Felten, Anna R. Karlin, and Kai Li. 1996. Implementation and Performance of Integrated Application-Controlled File Caching, Prefetching, and Disk Scheduling. *ACM Trans. Comput. Syst.* 14, 4 (1996), 311–343. doi:10.1145/235543.235544
  - [7] Esha Choukse, Pratyush Patel, Chaojie Zhang, Aashaka Shah, Íñigo Goiri, Saeed Maleki, Rodrigo Fonseca, and Ricardo Bianchini. 2025. Splitwise: Efficient Generative LLM Inference Using Phase Splitting. *IEEE Micro* 45, 4 (2025), 54–59. doi:10.1109/MM.2025.3575361
  - [8] James C. Corbett, Jeffrey Dean, Michael Epstein, Andrew Fikes, Christopher Frost, J. J. Furman, Sanjay Ghemawat, Andrey Gubarev, Christopher Heiser, Peter Hochschild, Wilson C. Hsieh, Sebastian Kanthak, Eugene Kogan, Hongyi Li, Alexander Lloyd, Sergey Melnik, David Mwaura, David Nagle, Sean Quinlan, Rajesh Rao, Lindsay Rolig, Yasushi Saito, Michal Szymaniak, Christopher Taylor, Ruth Wang, and Dale Woodford. 2013. Spanner: Google’s Globally Distributed Database. *ACM Trans. Comput. Syst.* 31, 3 (2013), 8. doi:10.1145/2491245
  - [9] Christian Corrà and Luca Chittaro. 2025. Exploring the Potential and Limitations of Large Language Models to Control the Behavior of Embodied Persuasive Agents. In *Persuasive Technology - 20th International Conference, PERSUASIVE 2025, Limassol, Cyprus, May 5–7, 2025, Proceedings (Lecture Notes in Computer Science)*. Springer, 61–73. doi:10.1007/978-3-031-94959-3\_5
  - [10] CrewAI. 2023. CrewAI: Framework for Orchestrating Role-Playing, Autonomous AI Agents. GitHub repository. <https://github.com/crewAIInc/crewAI>
  - [11] Tri Dao. 2024. FlashAttention-2: Faster Attention with Better Parallelism and Work Partitioning. In *The Twelfth International Conference on Learning Representations, ICLR 2024, Vienna, Austria, May 7–11, 2024*. OpenReview.net.
  - [12] Tri Dao, Daniel Y. Fu, Stefano Ermon, Atri Rudra, and Christopher Ré. 2022. FlashAttention: Fast and Memory-Efficient Exact Attention with IO-Awareness. In *Advances in Neural Information Processing Systems 35: Annual Conference on Neural Information Processing Systems 2022, NeurIPS 2022, New Orleans, LA, USA, November 28 – December 9, 2022*.
  - [13] Ewa Deelman, Karan Vahi, Gideon Juve, Mats Rynge, Scott Callaghan, Philip Maechling, Rajiv Mayani, Weiwei Chen, Rafael Ferreira da Silva, Miron Livny, and R. Kent Wenger. 2015. Pegasus, a workflow management system for science automation. *Future Gener. Comput. Syst.* 46 (2015), 17–35. doi:10.1016/J.FUTURE.2014.10.008
  - [14] Peter J. Denning. 1970. Virtual Memory. *Comput. Surveys* 2, 3 (1970), 153–189. doi:10.1145/356571.356573
  - [15] Thomas Dohmke. 2024. GitHub Copilot Workspace: Welcome to the Copilot-Native Developer Environment. GitHub Blog. <https://github.blog/news-insights/product-news/github-copilot-workspace/>
  - [16] Ulrich Drepper. 2007. What Every Programmer Should Know About Memory. Whitepaper, Red Hat, Inc. <https://people.freebsd.org/~lstewart/articles/cpumemory.pdf>
  - [17] Devdatt P. Dubhashi and Alessandro Panconesi. 2009. *Concentration of Measure for the Analysis of Randomized Algorithms*. Cambridge University Press.
  - [18] William Fedus, Barret Zoph, and Noam Shazeer. 2022. Switch Transformers: Scaling to Trillion Parameter Models with Simple and Efficient Sparsity. *J. Mach. Learn. Res.* 23 (2022), 120:1–120:39.
  - [19] Yao Fu, Leyang Xue, Yeqi Huang, Andrei-Octavian Brabete, Dmitrii Ustiugov, Yuvraj Patel, and Luo Mai. 2024. ServerlessLLM: Low-Latency Serverless Inference for Large Language Models. In *18th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2024, Santa Clara, CA, USA, July 10–12, 2024*. USENIX Association, 135–153.
  - [20] Yichao Fu, Siqi Zhu, Runlong Su, Aurick Qiao, Ion Stoica, and Hao Zhang. 2024. Efficient LLM Scheduling by Learning to Rank. In *Advances in Neural Information Processing Systems 38: Annual Conference on Neural Information Processing Systems 2024, NeurIPS 2024, Vancouver, BC, Canada, December 10 – 15, 2024*.
  - [21] Ali Ghodsi, Matei Zaharia, Benjamin Hindman, Andy Konwinski, Scott Shenker, and Ion Stoica. 2011. Dominant Resource Fairness: Fair Allocation of Multiple Resource Types. In *Proceedings of the 8th USENIX Symposium on Networked Systems Design and Implementation, NSDI 2011, Boston, MA, USA, March 30 – April 1, 2011*. USENIX Association.
  - [22] In Gim, Guojun Chen, Seung-Seob Lee, Nikhil Sarda, Anurag Khandelwal, and Lin Zhong. 2024. Prompt Cache: Modular Attention Reuse for Low-Latency Inference. In *Proceedings of the Seventh Annual Conference on Machine Learning and Systems, MLSys 2024, Santa Clara, CA, USA, May 13–16, 2024*. mlsys.org.
  - [23] In Gim, Zhiyao Ma, Seung-seob Lee, and Lin Zhong. 2025. Pie: A Programmable Serving System for Emerging LLM Applications. In *Proceedings of the ACM SIGOPS 31st Symposium on Operating Systems Principles (Lotte Hotel World, Seoul, Republic of Korea) (SOSP '25)*. Association for Computing Machinery, New York, NY, USA, 415–430. doi:10.1145/3731569.3764814
  - [24] Goetz Graefe. 1990. Encapsulation of Parallelism in the Volcano Query Processing System. In *Proceedings of the 1990 ACM SIGMOD International Conference on Management of Data, Atlantic City, NJ, USA, May 23–25, 1990*. ACM Press, 102–111. doi:10.1145/93597.98720
  - [25] Maurice Herlihy. 2006. The art of multiprocessor programming. In *Proceedings of the Twenty-Fifth Annual ACM Symposium on Principles of Distributed Computing, PODC 2006, Denver, CO, USA, July 23–26, 2006*. ACM, 1–2. doi:10.1145/1146381.1146382
  - [26] Ke Hong, Xiuhong Li, Lufang Chen, Qiuli Mao, Guohao Dai, Xuefei Ning, Shengen Yan, Yun Liang, and Yu Wang. 2025. SOLA: Optimizing SLO Attainment for Large Language Model Serving with State-Aware Scheduling. In *Proceedings of the Eighth Conference on Machine Learning and Systems, MLSys 2025, Santa Clara, CA, USA, May 12–15, 2025*. OpenReview.net/mlsys.org.
  - [27] Cunchen Hu, Heyang Huang, Liangliang Xu, Xusheng Chen, Chenxi Wang, Jiang Xu, Shuang Chen, Hao Feng, Sa Wang, Yungang Bao, Ninghui Sun, and Yizhou Shan. 2025. ShuffleInfer: Disaggregate LLM Inference for Mixed Downstream Workloads. *ACM Trans. Archit. Code Optim.* 22, 2, Article 77 (July 2025), 24 pages. doi:10.1145/3732941
  - [28] InfiniBand Trade Association. 2020. InfiniBand Architecture Specification, Volume 2, Release 1.4. Industry standards specification. <https://www.infinibandta.org/ibta-specification/>
  - [29] Patrick Jaillet, Jiashuo Jiang, Konstantina Mellou, Marco Molinaro, Chara Podimata, and Zijie Zhou. 2026. Online Scheduling for LLM Inference with KV Cache Constraints. *arXiv preprint arXiv:2502.07115* (2026). <https://arxiv.org/abs/2502.07115>
  - [30] Matthijs Jansen, Linus Wagner, Animesh Trivedi, and Alexandru Iosup. 2023. Continuum: Automate Infrastructure Deployment and Benchmarking in the Compute Continuum. In *Companion of the 2023 ACM/SPEC International Conference on Performance Engineering, ICPE 2023, Coimbra, Portugal, April 15–19, 2023*. ACM, 181–188. doi:10.1145/3578245.3584936
  - [31] Carlos E. Jimenez, John Yang, Alexander Wettig, Shunyu Yao, Kexin Pei, Ofir Press, and Karthik R. Narasimhan. 2024. SWE-bench: Can Language Models Resolve Real-world Github Issues?. In *The Twelfth International Conference on Learning Representations, ICLR 2024, Vienna, Austria, May 7–11, 2024*. OpenReview.net.
  - [32] Chao Jin, Zili Zhang, Xuanlin Jiang, Fangyue Liu, Shufan Liu, Xuanzhe Liu, and Xin Jin. 2026. RAGCache: Efficient Knowledge Caching for Retrieval-Augmented Generation. *ACM Trans. Comput. Syst.* 44, 1 (2026), 2:1–2:27. doi:10.1145/3768628
  - [33] Sayash Kapoor, Benedikt Stroebel, Zachary S. Siegel, Nitya Nadgir, and Arvind Narayanan. 2025. AI Agents That Matter. *Trans. Mach. Learn. Res.* 2025 (2025).
  - [34] Woosuk Kwon, Zhuohan Li, Siyuan Zhuang, Ying Sheng, Lianmin Zheng, Cody Hao Yu, Joseph Gonzalez, Hao Zhang, and Ion Stoica. 2023. Efficient Memory Management for Large Language Model Serving with PagedAttention. In *Proceedings of the 29th Symposium on Operating Systems Principles, SOSP 2023, Koblenz, Germany, October 23–26, 2023*. ACM, 611–626. doi:10.1145/3600006.3613165
  - [35] LangChain-AI. 2022. LangChain: Build Context-Aware Reasoning Applications. GitHub repository. <https://github.com/langchain-ai/langchain>
  - [36] Tor Lattimore and Csaba Szepesvári. 2020. *Bandit Algorithms*. Cambridge University Press. doi:10.1017/9781108571401
  - [37] Yueying Li, Jim Dai, and Tianyi Peng. 2025. Throughput-Optimal Scheduling Algorithms for LLM Inference and AI Agents. *arXiv preprint arXiv:2504.07347* (2025). <https://arxiv.org/abs/2504.07347>
  - [38] Chaofan Lin, Zhenhua Han, Chengruidong Zhang, Yuqing Yang, Fan Yang, Chen Chen, and Lili Qiu. 2024. Parrot: Efficient Serving of LLM-based Applications with Semantic Variable. In *18th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2024, Santa Clara, CA, USA, July 10–12, 2024*. USENIX Association, 929–945.
  - [39] Yuhao Liu, Hanchen Li, Yihua Cheng, Siddhant Ray, Yuyang Huang, Qizheng Zhang, Kuntai Du, Jiayi Yao, Shan Lu, Ganesh Ananthanarayanan, Michael Maire, Henry Hoffmann, Ari Holtzman, and Junchen Jiang. 2024. CacheGen: KV Cache Compression and Streaming for Fast Large Language Model Serving. In *Proceedings of the ACM SIGCOMM 2024 Conference, ACM SIGCOMM 2024, Sydney, NSW, Australia, August 4–8, 2024*. ACM, 38–56. doi:10.1145/3651890.3672274
  - [40] Michael Luo, Xiaoxiang Shi, Colin Cai, Tianjun Zhang, Justin Wong, Yichuan Wang, Chi Wang, Yanping Huang, Zhifeng Chen, Joseph E. Gonzalez, and Ion Stoica. 2025. Autellix: An Efficient Serving Engine for LLM Agents as General Programs. *arXiv preprint arXiv:2502.13965* (2025). <https://arxiv.org/abs/2502.13965>

- [41] Kshiteej Mahajan, Arjun Balasubramanian, Arjun Singhvi, Shivaram Venkataraman, Aditya Akella, Amar Phanishayee, and Shuchi Chawla. 2020. Themis: Fair and Efficient GPU Cluster Scheduling. In *17th USENIX Symposium on Networked Systems Design and Implementation, NSDI 2020, Santa Clara, CA, USA, February 25-27, 2020*. USENIX Association, 289–304.
- [42] Nimrod Megiddo and Dharmendra S. Modha. 2003. ARC: A Self-Tuning, Low Overhead Replacement Cache. In *Proceedings of the FAST '03 Conference on File and Storage Technologies, March 31 - April 2, 2003, Cathedral Hill Hotel, San Francisco, California, USA*. USENIX.
- [43] Meta. 2024. Llama 3 Model Card. Meta Llama documentation. [https://github.com/meta-llama/llama3/blob/main/MODEL\\_CARD.md](https://github.com/meta-llama/llama3/blob/main/MODEL_CARD.md)
- [44] Philipp Moritz, Robert Nishihara, Stephanie Wang, Alexey Tumanov, Richard Liaw, Eric Liang, Melih Elibol, Zongheng Yang, William Paul, Michael I. Jordan, and Ion Stoica. 2018. Ray: A Distributed Framework for Emerging AI Applications. In *13th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2018, Carlsbad, CA, USA, October 8-10, 2018*. USENIX Association, 561–577.
- [45] NVIDIA Corporation. 2018. NVIDIA NVSwitch: The World's Highest-Bandwidth On-Node Switch. NVIDIA Technical Overview. <https://images.nvidia.com/content/pdf/nvswitch-technical-overview.pdf>
- [46] NVIDIA Corporation. 2023. TensorRT-LLM: High-Performance Large Language Model Inference. GitHub repository. <https://github.com/NVIDIA/TensorRT-LLM>
- [47] Kay Ousterhout, Patrick Wendell, Matei Zaharia, and Ion Stoica. 2013. Sparrow: distributed, low latency scheduling. In *ACM SIGOPS 24th Symposium on Operating Systems Principles, SOSP '13, Farmington, PA, USA, November 3-6, 2013*. ACM, 69–84. doi:10.1145/2517349.2522716
- [48] Zaifeng Pan, AJKUMAR PATEL, Yipeng Shen, Zhengding Hu, Yue Guan, Wan-Lu Li, Lianhui Qin, Yida Wang, and Yufei Ding. 2025. KVFlow: Efficient Prefetch Caching for Accelerating LLM-Based Multi-Agent Workflows. In *The Thirty-ninth Annual Conference on Neural Information Processing Systems*. <https://openreview.net/forum?id=5Iw1nDtYmT>
- [49] R. Hugo Patterson, Garth A. Gibson, Eka Ginting, Daniel Stodolsky, and Jim Zelenka. 1995. Informed Prefetching and Caching. In *Proceedings of the Fifteenth ACM Symposium on Operating System Principles, SOSP 1995, Copper Mountain Resort, Colorado, USA, December 3-6, 1995*. ACM, 79–95. doi:10.1145/224056.224064
- [50] Steve Rennich. 2012. CUDA C/C++ Streams and Concurrency. NVIDIA CUDA Training Webinar. <https://developer.download.nvidia.com/CUDA/training/StreamsAndConcurrencyWebinar.pdf>
- [51] Minsoo Rhu, Natalia Gimelshein, Jason Clemons, Arslan Zulfiqar, and Stephen W. Keckler. 2016. vDNN: Virtualized deep neural networks for scalable, memory-efficient neural network design. In *49th Annual IEEE/ACM International Symposium on Microarchitecture, MICRO 2016, Taipei, Taiwan, October 15-19, 2016*. IEEE Computer Society, 18:1–18:13. doi:10.1109/MICRO.2016.7783721
- [52] Yeonju Ro, Haoran Qiu, Íñigo Goiri, Rodrigo Fonseca, Ricardo Bianchini, Aditya Akella, Zhangyang Wang, Mattan Erez, and Esha Choukse. 2025. Sherlock: Reliable and Efficient Agentic Workflow Execution. *arXiv preprint arXiv:2511.00330* (2025). <https://arxiv.org/abs/2511.00330>
- [53] Matthew Rocklin. 2015. Dask: Parallel Computation with Blocked algorithms and Task Scheduling. In *Proceedings of the 14th Python in Science Conference, SciPy 2015, Austin, Texas, USA, July 6-12, 2015*. scipy.org, 126–132. doi:10.25080/MAJORA-7B9E3ED-013
- [54] Yangjun Ruan, Honghua Dong, Andrew Wang, Silviu Pitis, Yongchao Zhou, Jimmy Ba, Yann Dubois, Chris J. Maddison, and Tatsunori Hashimoto. 2024. Identifying the Risks of LM Agents with an LM-Emulated Sandbox. In *The Twelfth International Conference on Learning Representations, ICLR 2024, Vienna, Austria, May 7-11, 2024*. OpenReview.net.
- [55] Ying Sheng, Shiyi Cao, Dacheng Li, Banghua Zhu, Zhuohan Li, Danyang Zhuo, Joseph E. Gonzalez, and Ion Stoica. 2024. Fairness in Serving Large Language Models. In *18th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2024, Santa Clara, CA, USA, July 10-12, 2024*. USENIX Association, 965–988.
- [56] Ying Sheng, Lianmin Zheng, Binhang Yuan, Zhuohan Li, Max Ryabinin, Beidi Chen, Percy Liang, Christopher Ré, Ion Stoica, and Ce Zhang. 2023. FlexGen: High-Throughput Generative Inference of Large Language Models with a Single GPU. In *International Conference on Machine Learning, ICML 2023, 23-29 July 2023, Honolulu, Hawaii, USA (Proceedings of Machine Learning Research)*. PMLR, 31094–31116.
- [57] Daniel Dominic Sleator and Robert Endre Tarjan. 1985. Amortized Efficiency of List Update and Paging Rules. *Commun. ACM* 28, 2 (1985), 202–208. doi:10.1145/2786.2793
- [58] Jovan Stojkovic, Chaojie Zhang, Íñigo Goiri, Josep Torrellas, and Esha Choukse. 2025. DynamoLLM: Designing LLM Inference Clusters for Performance and Energy Efficiency. In *IEEE International Symposium on High Performance Computer Architecture, HPCA 2025, Las Vegas, NV, USA, March 1-5, 2025*. IEEE, 1348–1362. doi:10.1109/HPCA61900.2025.00102
- [59] Biao Sun, Ziming Huang, Hanyu Zhao, Wencong Xiao, Xinyi Zhang, Yong Li, and Wei Lin. 2024. Llmnix: Dynamic Scheduling for Large Language Model Serving. In *18th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2024, Santa Clara, CA, USA, July 10-12, 2024*. USENIX Association, 173–191.
- [60] Douglas Thain, Todd Tannenbaum, and Miron Livny. 2005. Distributed computing in practice: the Condor experience. *Concurr. Pract. Exp.* 17, 2-4 (2005), 323–356. doi:10.1002/CPE.938
- [61] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N. Gomez, Lukasz Kaiser, and Illia Polosukhin. 2017. Attention is all you need. In *Proceedings of the 31st International Conference on Neural Information Processing Systems (Long Beach, California, USA) (NIPS'17)*. Curran Associates Inc., Red Hook, NY, USA, 6000–6010. [https://proceedings.neurips.cc/paper\\_files/paper/2017/file/3f5ee243547dee91fbd053c1c4a845aa-Paper.pdf](https://proceedings.neurips.cc/paper_files/paper/2017/file/3f5ee243547dee91fbd053c1c4a845aa-Paper.pdf)
- [62] Yuxin Wang, Yuhuan Chen, Zeyu Li, Xueze Kang, Yuchu Fang, Yeju Zhou, Yang Zheng, Zhenheng Tang, Xin He, Rui Guo, Xin Wang, Qiang Wang, Amelie Chi Zhou, and Xiaowen Chu. 2025. BurstGPT: A Real-World Workload Dataset to Optimize LLM Serving Systems. In *Proceedings of the 31st ACM SIGKDD Conference on Knowledge Discovery and Data Mining, V.2, KDD 2025, Toronto ON, Canada, August 3-7, 2025*. ACM, 5831–5841. doi:10.1145/3711896.3737413
- [63] Fangzhou Wu, Sandeep Silwal, and Qiuyi Zhang. 2026. Randomization Boosts KV Caching, Learning Balances Query Load: A Joint Perspective. In *The Fourteenth International Conference on Learning Representations*. <https://openreview.net/forum?id=R7fv5NWfMm>
- [64] 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. 2024. AutoGen: Enabling Next-Gen LLM Applications via Multi-Agent Conversations. In *First Conference on Language Modeling*. <https://openreview.net/forum?id=BAakY1hNKS>
- [65] Shunyu Yao, Dian Yu, Jeffrey Zhao, Izhak Shafran, Tom Griffiths, Yuan Cao, and Karthik Narasimhan. 2023. Tree of Thoughts: Deliberate Problem Solving with Large Language Models. In *Advances in Neural Information Processing Systems 36: Annual Conference on Neural Information Processing Systems 2023, NeurIPS 2023, New Orleans, LA, USA, December 10 - 16, 2023*.
- [66] Shunyu Yao, Jeffrey Zhao, Dian Yu, Nan Du, Izhak Shafran, Karthik R. Narasimhan, and Yuan Cao. 2023. ReAct: Synergizing Reasoning and Acting in Language Models. In *The Eleventh International Conference on Learning Representations, ICLR 2023, Kigali, Rwanda, May 1-5, 2023*. OpenReview.net.
- [67] Naimeng Ye, Arnab Ahuja, Georgios Liargkovas, Yunan Lu, Kostis Kaffes, and Tianyi Peng. 2026. Speculative Actions: A Lossless Framework for Faster AI Agents. In *The Fourteenth International Conference on Learning Representations*. <https://openreview.net/forum?id=P0G0k5wslg>
- [68] Gyeong-In Yu, Joo Seong Jeong, Geon-Woo Kim, Soojeong Kim, and Byung-Gon Chun. 2022. Orca: A Distributed Serving System for Transformer-Based Generative Models. In *16th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2022, Carlsbad, CA, USA, July 11-13, 2022*. USENIX Association, 521–538.
- [69] Matei Zaharia, Omar Khattab, Lingjiao Chen, Jared Quincy Davis, Heather Miller, Christopher Potts, James Zou, Michael Carbin, Jonathan Franke, Naveen Rao, and Ali Ghodsi. 2024. The Shift from Models to Compound AI Systems. Berkeley Artificial Intelligence Research (BAIR) Blog. <https://bair.berkeley.edu/blog/2024/02/18/compound-ai-systems/>
- [70] Lianmin Zheng, Liangsheng Yin, Zhiqiang Xie, Chuyue Sun, Jeff Huang, Cody Hao Yu, Shiyi Cao, Christos Kozyrakis, Ion Stoica, Joseph E. Gonzalez, Clark W. Barrett, and Ying Sheng. 2024. SGLang: Efficient Execution of Structured Language Model Programs. In *Advances in Neural Information Processing Systems 38: Annual Conference on Neural Information Processing Systems 2024, NeurIPS 2024, Vancouver, BC, Canada, December 10 - 15, 2024*.
- [71] Yinmin Zhong, Shengyu Liu, Junda Chen, Jianbo Hu, Yibo Zhu, Xuanzhe Liu, Xin Jin, and Hao Zhang. 2024. DistServe: Disaggregating Prefill and Decoding for Goodput-optimized Large Language Model Serving. In *18th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2024, Santa Clara, CA, USA, July 10-12, 2024*. USENIX Association, 193–210.
- [72] Shuyan Zhou, Frank F. Xu, Hao Zhu, Xuhui Zhou, Robert Lo, Abishek Sridhar, Xianyi Cheng, Tianyue Ou, Yonatan Bisk, Daniel Fried, Uri Alon, and Graham Neubig. 2024. WebArena: A Realistic Web Environment for Building Autonomous Agents. In *The Twelfth International Conference on Learning Representations, ICLR 2024, Vienna, Austria, May 7-11, 2024*. OpenReview.net.