KAIROS: Stateful, Context-Aware Power-Efficient Agentic Inference Serving

Yichao Yuan
University of Illinois
Urbana-Champaign
Urbana, IL, USA
yichaoy2@illinois.edu

Mosharaf Chowdhury
University of Michigan
Ann Arbor, MI, USA
mosharaf@umich.edu

Nishil Talati
University of Illinois
Urbana-Champaign
Urbana, IL, USA
nishil@illinois.edu

Abstract

Power has become a central bottleneck for AI inference. This problem is becoming more urgent as agentic AI emerges as a major workload class, yet prior power-management techniques focus almost entirely on single-turn LLM serving. Our analysis shows that agentic serving behaves fundamentally differently: each request carries long-lived context that evolves across tool-interleaved turns, and lowering GPU frequency can push the system into a thrashing regime where memory pressure sharply worsens both performance and power efficiency. These observations show that power optimization for agentic serving requires rethinking.

We present KAIROS, a context-aware power optimization system for agentic AI serving. KAIROS uses *agent context as a first-class control signal* to jointly manage GPU frequency, per-instance concurrency, and multi-instance request placement. This enables KAIROS to save power when memory headroom exists while avoiding thrashing and preserving performance targets. At a high level, KAIROS tracks requests at agent granularity, adapts local control to context growth and agent progress, and routes agents across instances to jointly improve power efficiency and memory stability. Evaluated across diverse software and data engineering agentic tasks, KAIROS achieves an average of 27% (up to 39.8%) power reduction while meeting the performance targets.

CCS Concepts: • **Hardware → Platform power issues**; • **Software and its engineering → Power management**; *Memory management*; • **Computing methodologies →** *Intelligent agents.*

Keywords: Agentic AI, LLM inference serving, power efficiency, DVFS, GPU frequency scaling, KV cache management

ACM Reference Format:
Yichao Yuan, Mosharaf Chowdhury, and Nishil Talati. 2027. KAIROS: Stateful, Context-Aware Power-Efficient Agentic Inference Serving. In *Proceedings of the 32nd ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 1 (ASPLOS '27), April 11–15, 2027, Heraklion, Greece.* ACM, New York, NY, USA, 21 pages. https://doi.org/10.1145/3845814.3850050



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

ASPLOS '27, Heraklion, Greece
© 2027 Copyright held by the owner/author(s).
ACM ISBN 979-8-4007-3012-2/2027/04
https://doi.org/10.1145/3845814.3850050

1 Introduction

Power is a *first-order constraint* for AI systems because AI data center demand has far outpaced the growth of available energy and grid capacity [22, 23, 42]. Reducing power is therefore critical not only for sustainability, but also for improving throughput within a fixed facility power envelope by enabling more usable compute capacity [48, 57, 58]. This urgency is likely to grow because recent trends [5, 60] show *agentic AI* as a major emerging inference workload, and serving agentic AI requires orders of magnitude higher power than traditional LLM serving [28]. This is because agentic AI replaces a single inference with many tool-interleaved, stateful LLM invocations, often requiring dozens of model calls per request and thereby driving much higher computational cost. Therefore, *improving the power consumption of agentic AI serving is both urgent and critical.*

A large body of prior work has explored reducing power and energy cost of *single-turn LLM serving*. One major class of techniques uses dynamic frequency and power control. For example, DVFS-based runtimes that tune GPU frequency online using signals such as workload slack, phase behavior, or workload state [6, 26, 36, 48, 55, 58, 61, 72]. A second class improves efficiency through system-level scheduling and resource management, such as hardware-aware placement, heterogeneous scheduling, and power-aware cluster orchestration [33, 57, 63]. However, these techniques are designed for *stateless* LLM serving, where requests are short-lived and largely independent. This raises a fundamental question: *can power saving techniques designed for single-turn LLM serving be applied directly to agentic AI serving?*

To answer this question, we conduct a detailed characterization of agentic AI workloads from the perspective of an LLM serving system. Specifically, we focus on ReAct-style agents [69] because prior work has shown that they lie near/on the intelligence–cost Pareto frontier [28]. We find that, unlike single-turn serving, agentic workloads maintain *long-lived state* for each request through a *context cache* [30] that grows across tool-interleaved turns and varies widely

across agents. Our study further shows that this context growth is *highly dynamic*: it varies substantially over time within and across agents, and the distributions of agent lifetimes, turn counts, and growth rates all exhibit long tails. This makes memory pressure difficult to predict and control, and creates a fundamental tension for power management: lowering GPU frequency saves power, but it also slows execution, and in agentic workloads that slowdown accumulates across turns and can directly hurt per-agent progress.

Our analysis also uncovers a new power-management phenomenon: lowering GPU frequency beyond a certain value can drive the system into a *thrashing regime*, where accumulated context exceeds GPU memory capacity. As a result, the context needs to be recomputed [27, 32] or loaded from slower memory tiers [9] that degrade throughput, latency, and power efficiency. This strong coupling between power and memory stability shows that agentic serving is not merely a scaled-up version of single-turn LLM serving, but a fundamentally different workload whose stateful and dynamic behavior reshapes the power-performance trade-off. As a result, *we must rethink the problem of power optimization for agentic AI* around context growth, memory stability, and thrashing-aware control.

In this paper, we present KAIROS, a context-aware power management system for agentic AI serving that reduces power consumption while maintaining a target performance SLO. Our goal is to minimize system power without significantly sacrificing end-to-end agent progress, despite the highly dynamic and stateful nature of agentic workloads. To achieve this, KAIROS is designed around three key principles: (1) dynamically adjusting GPU frequency based on context pressure to safely trade performance for power savings, (2) regulating concurrency to prevent excessive context growth that can destabilize the system, and (3) scaling and routing requests across multiple serving instances in a power-aware manner to consolidate load at low demand and spread load under high context pressure.

To realize these goals, KAIROS introduces a unified runtime that combines lightweight request tracking, per-instance control, and global routing. First, it *tracks requests at the agent granularity*, enabling the system to reason about long-lived, multi-turn workflows rather than isolated LLM calls. Each serving instance is then equipped with a *context-aware controller* that monitors context growth and system load to adapt GPU frequency and admission decisions. This allows KAIROS to save power when capacity exists while avoid thrashing and preserving performance. Finally, a *global router* places agents across serving instances based on context and load, consolidating under light demand and spreading under high context pressure. Together, these components use *agent context as a first-class signal* for power-efficient and stable agentic AI serving.

We evaluate KAIROS on NVIDIA H100 GPUs that run vLLM [30] across three datasets, including SWE-bench Verified [11], DABStep [17], and Terminal-Bench 2.0 [40], and two agent types: mini-swe-agent [67] and Terminus-2 [40]. Unlike single-turn LLM serving that relies on TTFT/TBT, we adopt a throughput-based SLO since agentic execution is best measured by the rate of task progress (§4). KAIROS reduces average GPU power by **27%** (up to **39.8%**) for a single serving instance and **46.3%** for multi-instance serving while maintaining per-agent SLO of P5 throughput 20 tokens/s. KAIROS show effectiveness in avoiding thrashing, demonstrating stable operation under dynamic context growth.

To the best of our knowledge, **KAIROS is the first work to optimize power for agentic inference serving**. Our key contributions are:

- A detailed characterization of agentic inference serving that reveals why power optimization is fundamentally different and harder compared to single-turn LLM serving.
- A context-aware control mechanism that jointly manages GPU frequency and concurrency using context growth, memory pressure, and agent progress to reduce power while avoiding thrashing and preserving SLO attainment.
- KAIROS: an end-to-end system for agentic AI serving that reduces power by 27%, maintaining performance targets.

2 Background: Agentic Inference Serving and Power/Energy Efficiency

This section provides brief background on the agentic inference serving workload, how is it different than single-turn LLM inference, and the need for power efficiency.

2.1 Agentic AI Inference Serving

Agentic inference differs from single-turn LLM inference in that the model is invoked *repeatedly in a closed-loop process* that alternates between reasoning, acting, and observing feedback, rather than a single prompt-response exchange. A few examples of existing agent designs include ReAct [69], Reflexion [52], and LATS [75], which differ mainly in how they organize control flow around the model. A recent characterization study [28] found that ReAct often lies on or near the accuracy–cost Pareto frontier. Therefore, we focus our discussion on **agents following ReAct paradigm** because they are widely adopted and effective across diverse tasks including multi-hop question answering [53], web shopping [68], data engineering [17], and software engineering [25]. As we will demonstrate in evaluation (§ 8.2), our techniques are generally applicable to advanced production-grade agents like Claude Code [3].

As shown in Figure 1, a ReAct agent repeatedly sends prompts and tool outputs to an LLM server, such as vLLM [30], executes the returned action in an execution environment, and feeds the resulting observation into the next turn. When many agents run in parallel, the serving system must handle

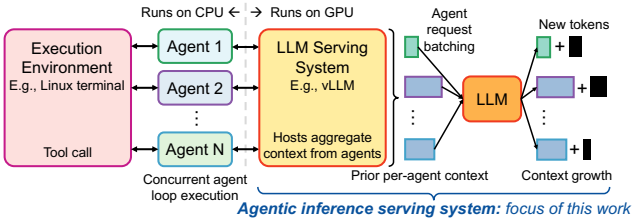


Figure 1. Example of ReAct agent workflow: multiple concurrent agents perform multi-turn conversations between execution environment and the LLM serving system.

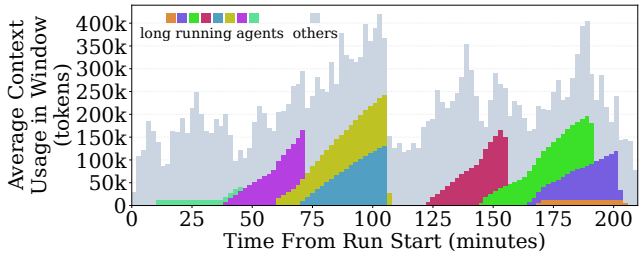


Figure 2. Agent context growth over time for concurrent workloads served by a single vLLM instance; the eight longest contexts are highlighted in distinct colors, with others in gray.

many interleaved prefill and decode phases whose *per-agent contexts persist and grows* across turns rather than disappearing after one response. In systems such as vLLM, previously processed tokens are retained in the *prefix/context cache* [30] so later turns can resume from accumulated history instead of recomputing the entire conversation from scratch. Once the agent completes a task, the entire context becomes useless, freeing up a large GPU memory capacity. This makes agentic inference serving fundamentally ***stateful***, because each new turn depends on prior context.

A concrete example of a ReAct agent is SWE-Bench [25] that resolves GitHub issues, where an agent iteratively inspects code, runs commands and tests, revises its hypothesis, edits files, and re-queries the model until it produces a valid patch. Tokens from each turn are retained in a context cache, making serving inherently *stateful*, while actions in the environment are guided by the language model.

2.2 An Urgent Need for Power/Energy Efficiency

AI data centers face severe energy and power pressure, with inference as a major contributor and modern deployments operating at hundreds of megawatts to gigawatts [28, 38, 58]. This paper focuses on *power* because data centers are increasingly constrained by available watts; reducing per-server power can increase deployable compute capacity within a fixed facility power envelope [7]. This issue is becoming more urgent as recent work [5, 60] points to agentic AI as an emerging workload class, and prior characterization [28] shows that agentic inference consumes two to three orders of magnitude higher energy than single-turn LLM. This gap arises because agentic serving repeatedly alternates prefill, decode, and tool phases over many turns while the serving system retain prior tokens in the context cache, causing GPU memory usage to grow unevenly and intensify both capacity and bandwidth pressure. Although agentic execution uses CPUs for orchestration and tool interaction, the GPU-side LLM serving path remains the dominant power-hungry component, which is the focus of our work (Figure 1).

GPU power can be reduced through static power caps, dynamic frequency scaling, or broader thermal/power-aware scheduling [26, 48, 55, 57]. In our workload setting, agent context evolves dynamically across turns, making static policies suboptimal fit, motivating DVFS as an online control mechanism. Because lowering frequency also slows inference down, any such approach must balance power savings against a performance target SLOs. SLOs in LLM serving are defined using latency metrics: TTFT and TBT, which are designed to capture user-perceived responsiveness in interactive applications [1]. In contrast, we argue for a throughput-based SLO for agentic serving with more details in §4.

3 Characterization of Agentic Inference Serving

3.1 Experimental setup

While §7 presents a detailed methodology used throughout the paper, this subsection briefly discusses the experimental setup used for our characterization study. We run agentic AI inference using the Harbor [19] framework on a single NVIDIA H100 GPU with context caching enabled [30]. Unless otherwise specified, the GPU runs a vLLM server that with a Qwen3-Coder-30B model with SWE-bench Verified [11] dataset and mini-swe-agent [67]. We use a fixed input request rate of 0.08 agent jobs per second using a Poisson distribution to represent real-world traffic patterns [41]. Therefore, in the steady state, vLLM serves requests from multiple agents concurrently. Each agent job will perform multiple conversation turns, resulting in a large batch size.

3.2 Dynamic Context Growth

Figure 2 illustrates how the context cache capacity of serving concurrent agents evolves over time. The eight longest contexts are distinguished using different colors, while all remaining contexts are shown in gray to improve the readability of the figure. Unlike single-turn LLM serving, where workload variability arises mainly from input/output sequence lengths [1, 30, 71], agentic serving introduces an additional source of variation: long-lived context capacity evolves ***dynamically*** in three unique ways.

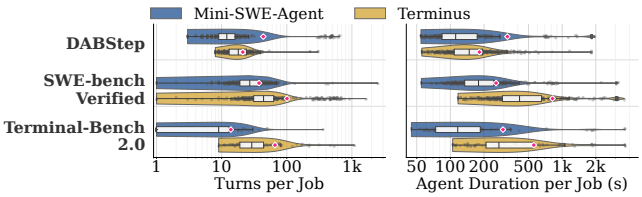


Figure 3. Conversation turn count (left) and agent duration (right) log distribution across two agent types and three datasets, showing a high degree of variation.

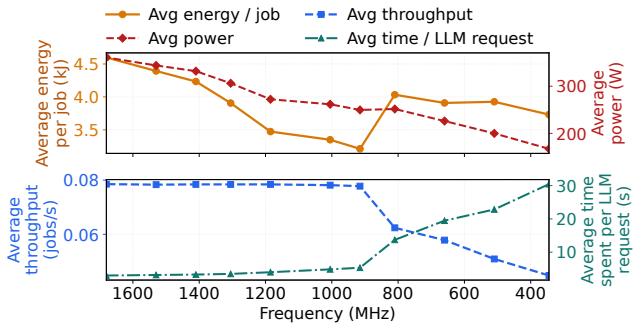


Figure 4. Effect of reducing GPU frequency on energy and power consumption (top), throughput, and average time per LLM request (bottom) for agentic serving system.

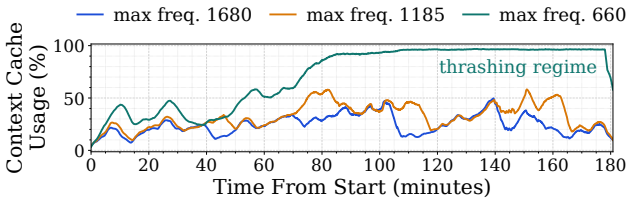


Figure 5. Aggregate context cache growth over time for concurrent agents running at three maximum GPU core frequencies: 1680 MHz, 1185 MHz, and 660 MHz.

First, the capacity and longevity of each context depends on the *number of conversation turns*. For example, our setup serves an average of 17 agents in parallel. The number of conversation turns vary from 1 to 2518, with an average of 37 turns, that depends on the complexity of agent task. This makes the length of each agent context and its longevity in the serving system a function of the input task, which cannot be determined statically.

To understand this behavior further, Figure 3 shows the distribution of number of conversation turns and total time spent by each agent in the LLM serving system. Due to space limitation, the distribution of maximum context length per agent is shown in Figure 14 (§A.3). Results reveal significant variability in turn counts and agent duration. Specifically, the heavy-tailed distribution highlights that while many jobs conclude quickly, a non-trivial portion of *long-running* tasks requires thousands of turns.

Because agents have different lifetimes, **second**, context accumulation across all agents running concurrently is highly uneven over time. When an agent completes, its entire context cache is released at once, causing an abrupt drop in the total accumulated context footprint. **Third**, the rate of context growth varies largely across multiple agents. This rate is a function of runtime properties including input and output sequence lengths for each conversation, and the amounts of time spent on LLM and tool calls. This analysis clearly shows that agentic serving creating highly dynamic and unpredictable memory demands for the serving system.

3.3 Effect of GPU Frequency Scaling

Reducing GPU core frequency is a common way to lower power, but it also creates a trade-off with inference performance. Prior work has only studied this trade-off in single-turn LLM serving [14, 26, 36, 39, 57, 58, 72]. Figure 4 shows how GPU frequency affects agentic serving efficiency. Two observations stand out: (1) reducing frequency from 1680 MHz to 900 MHz lowers energy and power by about 30% with little impact on throughput or latency, but (2) below 900 MHz this trend reverses, with sharply higher energy, lower throughput, and longer time for each LLM request.

This reversal occurs because lowering frequency slows execution while the request arrival rate remains fixed at 0.08 agents/s (§3.1). As service slows, pending work accumulates and the total context footprint of concurrent agents grows until it exceeds GPU memory capacity, pushing the system into a ***thrashing regime*** where cached context is evicted and previously processed tokens must be *recomputed*. This extra work raises energy consumption, reduces throughput, and increases latency. Although memory pressure can also appear in single-shot LLM serving, agentic workloads amplify this effect over multi-turn serving.

3.4 Deep Dive Into Context Thrashing

Based on these observations, we classify the serving system into two regimes: **non-thrashing**, where the aggregate context of concurrent agents fits within available GPU memory, and **thrashing**, where it exceeds that capacity. Figure 5 shows the transition by plotting aggregate context growth at 1680, 1185, and 660 MHz under a fixed request rate of 0.08 agents/s. At 1185 MHz, execution slows only slightly relative to 1680 MHz, so tasks still complete fast enough to keep memory pressure stable. At 660 MHz, however, compute throughput drops enough that accumulated context grows faster than it can be drained, exceeds available memory, pushing the system into a thrashing regime. This result shows that *frequency-induced compute bottlenecks are a root*

cause of both energy and performance instability in agentic serving. Appendix §A.4 further examines thrashing performance under two common fallbacks: recomputation [27, 32] and LMCache-based offloading [9].

3.5 Summary of Unique Challenges

Based on our characterization, we identify three unique Challenges for power saving in agentic inference serving.

- **C1: Tension between power saving and performance.** Lowering GPU frequency reduces power, but also slows agent execution, creating a fundamental trade-off. The unique challenge is that the slowdown is *context-dependent and accumulates across turns*, making it difficult to reduce power without violating a performance target.
- **C2: Highly dynamic context demand.** In agentic serving, both the size and the lifetime of each agent's context evolve online as a function of task complexity and tool interaction, making memory demand challenging to predict or provision statically.
- **C3: Strong coupling between power and memory stability.** Reducing GPU frequency can save power, but slower execution prolongs agent lifetime and increases resident context, which can push the system into a *thrashing regime* that sharply degrades throughput, latency, and power efficiency.

These challenges differ fundamentally from single-turn LLM serving, motivating a rethinking of power optimization.

4 Problem Formulation

We model an agentic serving system $\mathcal{S}$ in Figure 1 as a collection of underlying serving instances $\mathcal{M}$. A serving instance $m \in \mathcal{M}$ is defined as a single unit of an LLM server that either a single or multiple GPUs, depending on the size of the model. The system processes an incoming stream of agents using a collection of instances. Let $\mathcal{A}$ denote the set of agents, where each agent $a \in \mathcal{A}$ generates a sequence of requests:

$$a = (r_1, r_2, \dots), \quad r_i = (\Delta p_i, \Delta d_i),\ r_i \in \mathcal{R}, \tag{1}$$

arriving at a target rate (e.g., agent requests per second), where Δp_i is the number of newly appended prefill tokens (e.g., initial prompt or environment observation) and Δd_i is the number of generated decode tokens.

The context (history) after step i is the concatenation of all prior tokens:

$$H_i = (\Delta p_1, \Delta d_1, \Delta p_2, \Delta d_2, \dots, \Delta p_i), \quad C_i = |H_i|. \tag{2}$$

Each request r_i is processed with end-to-end latency t_i, which depends on system conditions and the current context size C_i. We define the per-agent throughput as:

$$\text{Throughput}(a) = \frac{\sum_i \Delta d_i}{\sum_i t_i}, \tag{3}$$

and say that an agent a satisfies a throughput-based SLO with target τ (tokens/s) if $\text{Throughput}(a) \geq \tau$.

While prior work uses latency-based SLOs such as TTFT, TBT, or TTLT [73] (time to last token), we argue that *per-agent throughput* is a more *user-centric* metric that better captures end-to-end agent progress over its full lifecycle. TTFT and TBT are designed for fine-grained, human-facing generation, whereas agentic serving operates at the task level with multi-turn, autonomous execution. TTLT is less actionable for online control because it depends on an isolated request-specific latency that is not available in online serving at runtime. In contrast, throughput directly reflects runtime agent progress under shared execution, making it a more *actionable* SLO for agentic serving.

Let $\mathcal{A}(T)$ denote the set of agents that complete within a time interval $[0, T]$, and let $|\mathcal{A}(T)|$ be its cardinality. We define the SLO attainment rate over $[0, T]$ as:

$$\text{SLO-Attainment}(T) = \frac{1}{|\mathcal{A}(T)|} \sum_{a \in \mathcal{A}(T)} \mathbf{1}\big[\text{Throughput}(a) \geq \tau\big]. \tag{4}$$

We define the total instantaneous system power and the average system power over $[0, T]$ as:

$$P(t) = \sum_{m \in \mathcal{M}} P_m(t), \quad \bar{P}(T) = \frac{1}{T} \int_0^T P(t)\, dt, \tag{5}$$

where $P_m(t)$ denotes the instantaneous power of instance m at time t. Given these definitions, we define an optimization objective as follows.

Optimization objective: Minimize the power consumption $\bar{P}(T)$ while maintaining SLO-Attainment(T) comparable to a non power-optimized baseline.

5 Design Goals and Overview

Motivated by the unique challenges of agentic inference serving (§3.5), this section presents the key design goals and a high-level overview of KAIROS toward achieving the optimization objective above.

5.1 KAIROS Design Goals

KAIROS defines three Design Goals.

- **DG1: Context-aware frequency control.** Control GPU frequency using context as a first-class signal so the system can trade speed for power savings when memory headroom exists. The key goal is to *reduce frequency* only up to the point that the *system remains in the non-thrashing regime* while *maintaining SLO attainment*.
- **DG2: Context-aware concurrency control.** Regulate how many agents are admitted to each serving instance so that aggregate context demand remains within a safe memory envelope. The goal is to *avoid* overloading any instance with excessive long-lived context that would *trigger thrashing* and hurt both performance and power.
- **DG3: Power-aware multi-instance scaling and routing.** For multiple serving instances, scale resources and

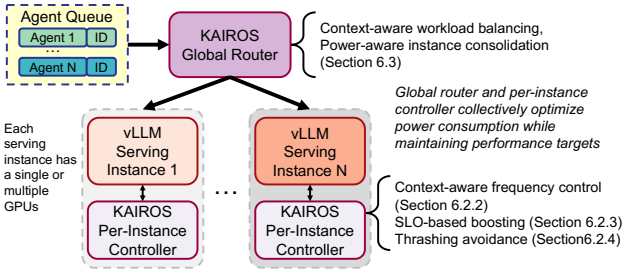


Figure 6. Design overview of KAIROS: it tracks serving requests for each agent, a global router assigns requests to different vLLM serving instances, and per-instance controller that adjusts GPU frequency to optimize power.

route requests across them to *jointly minimize system power* and *maintain SLO attainment*. The goal is to consolidate load onto fewer instances under light demand to exploit low idle-power states, while spreading load under higher context pressure to avoid local memory instability.

5.2 KAIROS Design Overview

The design goals motivate a unified context-aware runtime that jointly controls frequency, concurrency, and multi-GPU scaling to reduce power while preserving SLO attainment. Figure 6 shows the high-level design of KAIROS. KAIROS is designed around the three goals in §5.1 to address the challenges (§3.5), and has the following components.

- **Agent ID Tracker.** KAIROS first annotates incoming LLM requests with an agent identifier so that requests belonging to the same multi turn workflow can be tracked across turns. Unlike a semantic variable [34], which encode application-level data dependencies and request semantics across LLM calls, our tracking is purely identifier-based and carries no information about request content or relationships. This is necessary because, from the viewpoint of the LLM serving system, requests from different agents otherwise appear as ordinary LLM requests without explicit agent-level structure.
- **Context-Aware Per-Instance Controller.** Each serving instance is paired with a local controller that monitors context growth, memory pressure, and agent progress, and then manages the instance accordingly. At a high level, this controller adjusts GPU frequency and regulates per-instance concurrency to save power, avoid thrashing, and preserve SLO attainment, directly supporting *DG1* and *DG2* while addressing *C1–C3*. There is a single controller per serving instance that spans either a single or multiple GPUs, depending on the model size.
- **Context-Aware Multi-instance Router.** Across multiple serving instances, KAIROS employs a global, context-aware router that dynamically adapts instance utilization to workload intensity. Under low load, the router *consolidates* agents onto a subset of instances to keep other GPUs in low-power idle states, while under high load it *spreads* agents to balance context pressure and avoid thrashing. By jointly optimizing placement for both memory stability and power efficiency, the router improves system-wide SLO attainment while reducing unnecessary power overhead, supporting *DG3* to address *C2* and *C3*.

Together, these components form a unified runtime that uses context as the central control signal across both local instance management and global request routing.

6 KAIROS Design Details

In this section, we present the detailed design of KAIROS.

6.1 Agent ID Tracker

A key challenge in agentic LLM inference is that serving backends are not agent-aware: requests are issued independently without agent identifiers, obscuring provenance and preventing agent-level scheduling and control. Modifying agent programs to add such annotations is impractical due to their complexity, heterogeneity, and rapid evolution.

To address this, KAIROS introduces a non-intrusive *Agent ID Generator* that enables agent-level tracking without modifying user programs. A lightweight wrapper launches each agent as a child process, assigns a unique ID, emits lifecycle signals (`agent_start`/`agent_end`), and transparently encodes the ID into each request (via the API key), ensuring end-to-end propagation. An example interface is shown in Appendix A.6.

At the backend, the router extracts agent IDs from requests to enable agent-level scheduling, context tracking, and control. This design is fully compatible with existing interfaces, incurs minimal overhead by reusing metadata channels, and provides the abstraction needed for KAIROS 's context-aware optimizations.

6.2 Context-Aware Per-Instance Controller

The goal of the KAIROS *per-instance context controller* is to manage each serving instance so that it reduces power while preserving SLO attainment and avoiding the non-ideal thrashing regime. Its key insight is that, in agentic serving, control decisions such as GPU frequency scaling and concurrency regulation must be driven by *context dynamics*, since each agent's state persists across turns and directly shapes future memory pressure and performance. This makes per-instance control fundamentally different from conventional stateless inference, where policies can focus primarily on the latency of the current request.

Accordingly, this subsection presents the controller through five components. *Control Policy Formulation* defines the local optimization objective. *Context-Aware Frequency Control* addresses *DG1* and tackles *C1/C3* by trading power for performance based on context pressure. *SLO-based Frequency*

Boosting preserves throughput targets under slowdown, directly addressing *C1*. *Thrashing Avoidance* provides the safety mechanism that keeps the instance within a stable non-thrashing region and performs context-aware concurrency control (*DG2*). Together, these components realize the local control plane of KAIROS, while *DG3* is handled globally by the multi-instance router (§6.3).

6.2.1 Control Policy Formulation. We model the controller as a discrete-time policy that operates at control epochs $k = 0, 1, 2, \ldots$. At each epoch, agents are divided into two sets: the set of ongoing agents $\mathcal{A}_k^{\text{on}}$ that are currently executing, and the set of pending agents $\mathcal{A}_k^{\text{pend}}$ that have arrived but have not yet been admitted.

We define the system context usage at epoch k as the aggregate context footprint of all ongoing agents:

$$U_k = \sum_{a \in \mathcal{A}_k^{\text{on}}} C^a_{j_a(k)}. \tag{6}$$

At each control epoch, the controller selects a GPU frequency setting f_k and updates the set of ongoing agents by admitting a subset of $\mathcal{A}_k^{\text{pend}}$ into execution. The control policy is therefore defined as

$$\pi : U_k \mapsto (f_k, \mathcal{A}_k^{\text{on}}). \tag{7}$$

Here, f_k governs the power–performance trade-off, while the evolution of $\mathcal{A}_k^{\text{on}}$ regulates concurrency to keep the system within a stable non-thrashing regime.

6.2.2 Context-Aware Frequency Control. We first describe the frequency control, which addresses *DG1* and directly targets *C1* and *C3* by adapting GPU frequency to the current context pressure of each serving instance.

At instance startup, KAIROS obtains two pieces of static information: (i) the set of available GPU frequency levels

$$\mathcal{F} = \{f^{(1)}, f^{(2)}, \ldots, f^{(L)}\},$$

sorted in ascending order, and (ii) the token capacity of the instance, denoted by $U_{\max}$.

At control epoch k, the controller observes the aggregate context usage U_k and selects the frequency level as

$$f_k = \begin{cases} f^{(L)}, & U_k \geq \alpha U_{\max}, \\ f^{(\ell_k)}, & \text{otherwise}, \end{cases} \tag{8}$$

where $\alpha \in (0, 1]$ is the context-pressure threshold for entering the protective high-frequency regime, and

$$\ell_k = \left\lfloor \frac{U_k}{\alpha U_{\max}} \cdot (L - 1) \right\rfloor + 1. \tag{9}$$

This policy linearly partitions the safe region $[0, \alpha U_{\max})$ into $L - 1$ intervals and increases the frequency level monotonically with context usage. In practice, we set $\alpha = 0.75$.

The design choice is guided by three principles. **First**, when context usage is low, the instance has substantial unused memory capacity. In this regime, reducing frequency slows individual requests but can safely trade excess capacity for lower power, since the additional context residency still fits within the safe region. In other words, the controller exploits available context headroom to reduce power.

Second, as context usage rises, the same slowdown becomes increasingly risky. Lower frequency prolongs agent lifetime, increases resident context, and can accelerate context accumulation, eventually pushing the instance into the thrashing regime. To address this risk, the controller progressively reduces the aggressiveness of frequency downscaling as U_k grows, using discrete levels to provide a smooth transition from low-pressure to high-pressure operation.

Third, low context usage often coincides with low memory pressure and short per-iteration execution time, which means the instance may be operating well above the performance required to meet the SLO. In this region, the controller can safely exploit available SLO slack for more aggressive power savings. Taken together, this policy uses aggregate context as the key runtime signal for balancing power reduction against performance and memory stability.

6.2.3 SLO-Aware Frequency Boosting. While the context-aware frequency policy improves power efficiency by exploiting available context headroom, it does not by itself guarantee per-agent performance. To address *C1* and further support *DG1*, KAIROS adds a *lightweight runtime safeguard* that boosts frequency when observed agent progress falls below the target SLO.

Specifically, for each agent a, we track its achieved throughput up to epoch k:

$$\text{Throughput}_k(a) = \frac{\sum_{i \leq j_a(k)} \Delta d_i}{\sum_{i \leq j_a(k)} t_i}, \tag{10}$$

where $j_a(k)$ denotes the latest completed step of agent a at epoch k.

At each control epoch, the controller checks the minimum throughput across all in-process agents:

$$\text{Throughput}_k^{\min} = \min_{a \in \mathcal{A}_k^{\text{on}} \cup \mathcal{A}_k^{\text{pend}}} \text{Throughput}_k(a). \tag{11}$$

If $\text{Throughput}_k^{\min}$ falls below the SLO target τ, the controller overrides the context-aware policy and sets the frequency to the maximum level:

$$f_k \leftarrow f^{(L)}. \tag{12}$$

This mechanism serves as a corrective boost when the system begins to under-perform. It temporarily increases serving capacity to recover throughput. Under normal conditions, the controller continues to benefit from context-aware power savings. Under stress, however, it prioritizes performance targets and reduces the likelihood of SLO violations.

6.2.4 Thrashing Avoidance via Concurrency Control. While frequency control helps manage the power–performance trade-off under changing context pressure, it is not sufficient

to keep the system out of the context-thrashing regime. This is challenging because thrashing creates a negative feedback loop: once throughput drops, requests accumulate, resident context grows further, and the resulting memory pressure drives the system even deeper into instability. Consequently, even a short overload can push the instance into a persistently inefficient operating region. To address *DG2*, *C2* and *C3*, KAIROS employs a concurrency control mechanism that bounds the aggregate context footprint of ongoing agents.

Specifically, at each control epoch k, the controller enforces an upper bound on total context usage:

$$U_k \leq \beta U_{\max}, \tag{13}$$

where $\beta \in (0, 1)$ is a safety margin (e.g., $\beta = 0.95$ in practice) that reserves headroom for transient decode growth, which is not directly observable before execution.

If admitting a new agent would violate this constraint, the agent remains in the pending set. If the constraint is already violated (e.g., due to bursty arrivals), newly arrived agents are deferred to avoid further escalation. To improve utilization without inducing oscillation, the controller uses a second threshold $\gamma < \beta$ (e.g., $\gamma = 0.9$) for admission. When pending agents exist and $U_k < \gamma U_{\max}$, the controller incrementally admits agents from $\mathcal{A}_k^{\text{pend}}$ in arrival order into $\mathcal{A}_k^{\text{on}}$ until the threshold is approached. This two-threshold design keeps concurrency near the safe capacity boundary, preserves headroom for short-term growth, and maintains operation in the non-thrashing regime while still utilizing available memory efficiently.

6.2.5 Putting It Altogether. Algorithm 1 in §A.1 summarizes the per-instance control loop in KAIROS. At each control epoch, the controller first updates the current context usage, per-agent throughput, and average power, and then selects a baseline frequency using the context-aware policy. It next applies two corrective mechanisms: SLO-based boosting to recover performance when throughput falls below target, and thrashing avoidance to regulate admissions and keep context usage within a safe memory region. Together, these steps provide a unified control loop that balances power saving, SLO attainment, and memory stability.

6.3 KAIROS Context-Aware Multi-Instance Router

Modern GPUs exhibit a significant difference in power consumption between idle and active states, rather than a smooth scaling with workload intensity. In particular, an idle GPU can operate at a very low power state (e.g., 50W on NVIDIA H100), while even a small amount of incoming work can trigger a substantial increase in power (e.g., an additional 100W), even under light load and at the lowest frequency setting. This creates a significant gap between idle and lightly-loaded operating regimes, making naive policies like round robin less efficient due to activates all instances.

To this end, KAIROS introduces a context-aware router that adapts instance utilization to the input load; its pseudocode is shown in Algorithm 2 and 3 in §A.2. Under peak demand, the router tends to utilize all available instances, while under lighter load it consolidates agents onto a smaller subset of GPUs and leaves the remaining instances idle to avoid unnecessary power overhead.[1] The router achieves this through two components: an assignment policy for initial placement and a reassignment policy for dynamic load balancing.

Assignment policy. The router uses a threshold-based assignment policy based on per-instance ongoing agent context usage. When there exist instances whose ongoing context usage remains below a threshold (set to 50% of the maximum context capacity), KAIROS prioritizes consolidation by assigning new agents in increasing instance ID order, effectively filling lower-index instances first. Once all active instances exceed this threshold, the router switches to a load-spreading mode and assigns each incoming agent to the instance with the lowest current context usage.

Reassignment policy. KAIROS maintains, for each agent, a step counter since its last (re)assignment and allows reassignment only after the counter reaches a fixed threshold (8 conversation turns in our implementation). At that point, the router compares the context usage of the agent's current instance with that of the active instance having the lowest context usage. If the current instance's context usage is at least 2× higher, the agent is reassigned to that lower-usage instance, and its counter is reset.

7 Evaluation Methodology

We evaluate KAIROS on an NVIDIA H100 NVL server, where each GPU has a 400 W TDP, paired with an Intel Xeon Platinum 8592+ CPU and 2 TB of DRAM. We use vLLM [30] v0.14.0 to host the LLM serving instance. Our primary model is *Qwen3-Coder-30B*,[2] quantized to FP8, with each vLLM instance using a single H100 GPU with 80 GB of memory. We evaluate KAIROS under varying input request arrival rates and SLO targets. We do not use prefill–decode (PD) disaggregated inference, since context pressure and context-driven serving behavior remain fundamental regardless of disaggregation; §B.1 provides a detailed justification.

Our evaluation uses three widely used datasets spanning diverse domains. (1) *SWE-Bench Verified* [25], a human validated benchmark for whether agents can generate test-passing fixes for real GitHub issues; (2) *DABStep* [17], a benchmark of real-world data analysis tasks for evaluating multi-step reasoning over structured and unstructured data; and (3) *Terminal-Bench 2.0* [40], a benchmark of realistic,

[1] Even when an instance is idle, it continues running the vLLM server and consumes power close to the GPU's idle level, which allows it to be brought back into service quickly without incurring a long restart delay [58].
[2] We additionally use *Ministral-3-14B* for model diversity.

human-verified command-line tasks for evaluating aegnt's ability to autonomously complete complex, long-horizon tasks. We use Zeus [13, 70] for all the frequency and power related control and measurement.

We use the two different agents scaffolding, whose implementation in Harbor [19], for the evaluation: (1) *mini-swe-agent [67]*: a lightweight open-source LLM agent that achieves strong performance despite its minimal design. (2) *terimus-2 [40]*: Harbor's autonomous reference agent for evaluating LLM capabilities.

To evaluate the broad applicability of KAIROS across different models and agent harness, we further include experiments based on Ministral-3-14B [35] as well as Claude Code [3] as agent harness, in addition to the main experiments.

For a fair evaluation, we first record the requests and timing from a real agent execution, then replay those requests with exactly the same token distribution, relative timing, and agent issuing order. As a result, our evaluation guarantees identical workload across different runs, eliminating any runtime variations. The evaluation runs over a 3-hour window to capture context dynamics and reflect real-world long-running workloads.

8 Evaluation Results

This section presents a detailed evaluation of how KAIROS design choices impact both performance and power.

8.1 Single-Instance Agentic Serving

Performance and power comparison. Figure 7 compares performance, power, and energy across different agent jobs/s arrival rates, with a per-agent SLO target of 20 tokens/s. We present additional results under varying SLO targets in the following discussion. In this section, we use P5 throughput as the performance metric, defined as the throughput of the slowest 5% of agents; analogous to P95 latency. The study evaluates three baselines: no frequency control, a fixed GPU frequency of 810 MHz, and KAIROS. While both fixed frequency baseline and KAIROS reduce power relative to no control, the fixed-frequency baseline achieves slightly larger power savings at the cost of significantly degraded SLO attainment. In contrast, KAIROS strikes a balance by achieving an average power saving of 27% (up to 39.8%) while meeting the per-agent SLO target of 20 tokens/second.

This result *underscores the effectiveness of a dynamic solution* over a static approach. While statically reducing GPU frequency can lower power, it is insufficient on its own because it often causes significant SLO violations. Even from a power-only standpoint, it is impractical for deployment since it requires expensive per-workload grid search and frequency tuning for finding an optimal operating conditions, which keeps changing rapidly under a highly dynamic workload (§3). Instead, KAIROS achieves a careful balance between power reduction and performance, maintaining SLO attainment while reducing power. This balance comes from dynamically adapting GPU frequency and concurrency based on context pressure and agent progress through context-aware frequency control, SLO-driven boosting, and thrashing-aware concurrency control.

Effectiveness of SLO boosting with different targets. Figure 8 shows per-agent P5 throughput (left) and average power (right) under different SLO targets: 20, 35, and 45 tokens/s at a fixed request arrival rate of 0.03 agent jobs/s. This data is collected by running mini-swe-bench with DABstep. It compares a no frequency control baseline with KAIROS configured for different SLO targets. Three key observations emerge. *First*, KAIROS adapts effectively to different SLO targets, meeting the required throughput while significantly reducing power. This demonstrates the effectiveness of its context-aware frequency control and SLO-driven boosting mechanisms of KAIROS.

Second, tighter SLO targets require higher GPU frequencies, which reduces power savings: as the SLO increases from 20 to 45 tokens/s, power savings drop from 32.9% to 6.3% relative to the baseline. The 45 tokens/s target is close to the full-frequency hardware limit. At this aggressive SLO, KAIROS barely meets the target, leaving limited opportunity for frequency reduction and therefore yielding a less favorable power–performance trade-off. Appendix §A.5 further shows per-agent throughput distribution.

Third, the figure reveals a hardware limit, where 45 tokens/s represents the maximum achievable per-agent SLO for this setup. Pushing beyond this point would significantly degrade SLO attainment. This indicates a tipping point governed by hardware capacity, suggesting that further improvements in SLO would require additional compute resources rather than better control alone.

Figure 9 provides a deeper look into KAIROS 's runtime behavior under a request arrival rate of 0.03 jobs/s[3] and an SLO target of 35 tokens/s by showing instantaneous power, GPU frequency, and context size over time. As context size (subfigure (a)) grows, KAIROS gradually increases GPU frequency (subfigure (b)) to accelerate agent progress and prevent excessive accumulation, while reducing frequency when context pressure subsides to save power (subfigure (c)). This adaptive behavior is reflected in the corresponding changes in power consumption, which closely track frequency adjustments. When the system detects potential SLO violations, it temporarily boosts the GPU to maximum frequency (shown by blue points) to recover performance. Overall, the figure illustrates how KAIROS dynamically responds to changing workload conditions to balance power savings and SLO attainment using context as a first-class control signal.

[3]Due to space limitation, we carefully select representative request rates and SLO targets to demonstrate the effectiveness of different techniques proposed in KAIROS in Figures 9, 10, and 11.

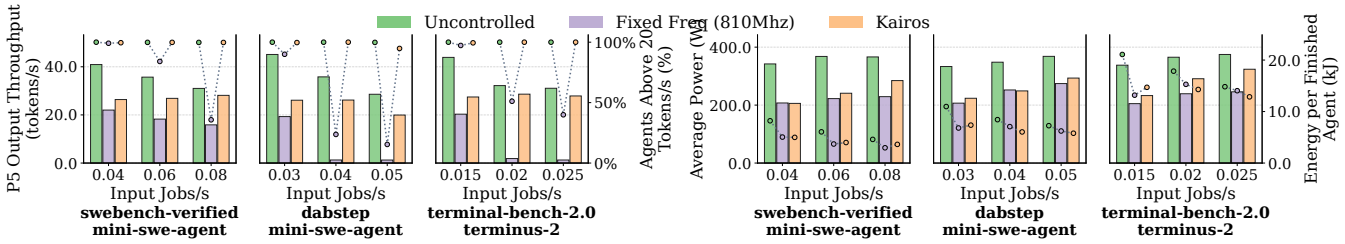


Figure 7. Comparison of throughput, SLO attainment (20 tokens/s per-agent target), energy, and power across request rates for no frequency control, fixed 810 MHz, and KAIROS. KAIROS reduces power by 27% on average without sacrificing SLO.

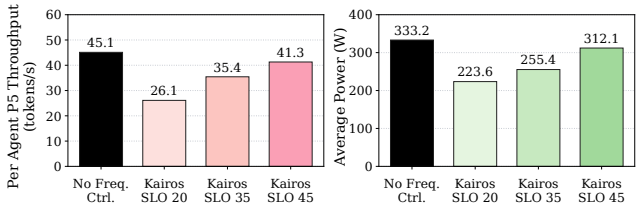


Figure 8. Effectiveness of SLO boosting. P5 throughput and average power of KAIROS with different SLO targets of 20, 35, and 45 tokens/s. This experiment runs mini-swe-agent on DABStep with an arrival rate of 0.03 jobs/s.

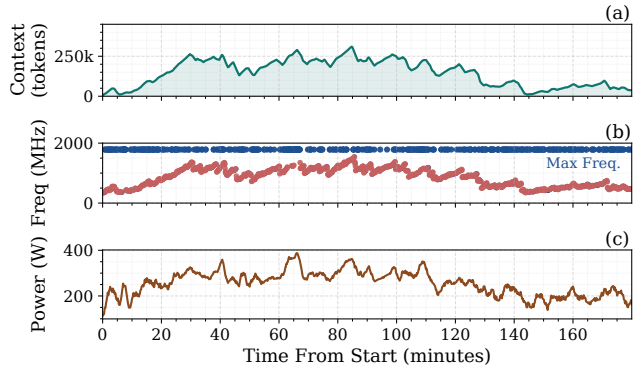


Figure 9. Change in instantaneous power, GPU frequency, and context size of KAIROS with respect to time for mini-swe-agent running on DABStep with arrival rate of 0.03 jobs/s and SLO target of 35 tokens/s.

Effectiveness of thrashing avoidance (concurrency control). Figure 10 evaluates the effectiveness of the thrashing avoidance mechanism under a higher load (0.05 jobs/s) with an SLO target of 20 tokens/s using mini-swe-agent on DABStep. It compares three configurations: no frequency control, KAIROS without thrashing avoidance, and KAIROS with thrashing avoidance. While the no-control baseline avoids thrashing by running at high frequency, KAIROS without thrashing avoidance reduces frequency to save power, pushing the system into a thrashing regime. This leads to a severe drop in per-agent P5 throughput (left figure) to 2.3 tokens/s

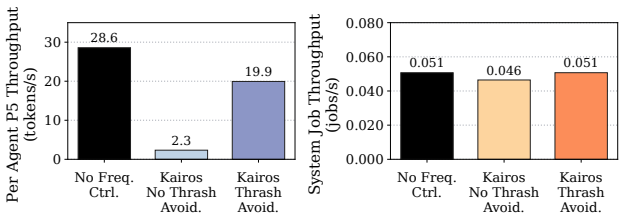


Figure 10. Effectiveness of thrashing avoidance. P5 throughput and overall job throughput for mini-swe-agent on DABStep with request rate of 0.05 jobs/s and an SLO target of 20 tokens/s.

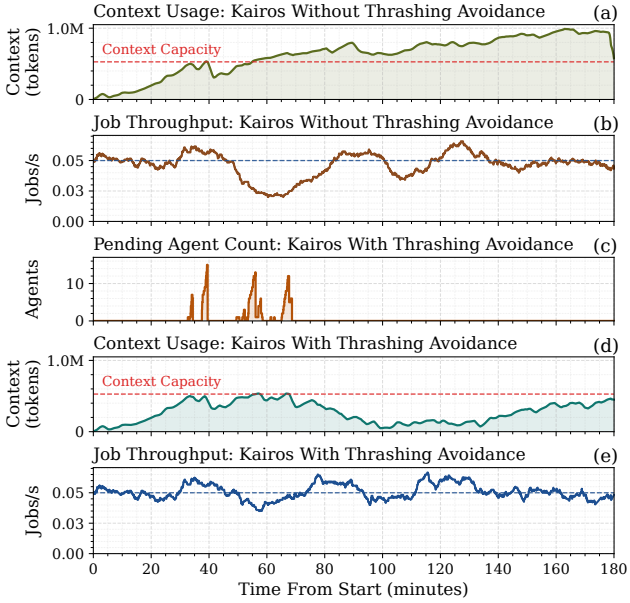


Figure 11. Context size, agent processing throughput and pending agents with and without employing thrashing avoidance technique in KAIROS. This experiment runs mini-swe-agent with DABStep with a request rate of 0.05 jobs/s and a target SLO of 20 tokens/s.

Table 1. Effectiveness of KAIROS across multiple models.

	Qwen3-Coder-30B (0.06 jobs/s)		Ministral-3-14B (0.06 jobs/s)	
	No Freq. Control	KAIROS	No Freq. Control	KAIROS
Power (W)	367.9	240.9	370.8	263.8

and overall throughput (right figure) of 0.046 jobs/s, failing to keep up with the arrival rate. In contrast, enabling thrashing avoidance mechanism allows KAIROS to maintain P5 throughput close to the 20 tokens/s target and sustain system throughput at 0.05 jobs/s, demonstrating stable and efficient operation.

Figure 11 provides a time-series view explaining this behavior. Without thrashing avoidance, context usage (subfigure (a)) exceeds capacity and remains high, pushing the system into a persistent thrashing regime where throughput drops (subfigure (b)) and queued agents accumulate (subfigure (c)), making recovery difficult. This leads to unstable operation as the system cannot drain incoming work fast enough. With thrashing avoidance enabled, KAIROS prevents this buildup by controlling context growth (subfigure (d)) and, when necessary, triggering corrective frequency boosts to reduce accumulated context. As a result, context stays within capacity, job throughput remains stable (subfigure (e)), and avoids queue buildup. These results highlight how thrashing avoidance and SLO-aware boosting work together to maintain system stability under high load.

Effectiveness across multiple models. Table 1 shows that the benefits of KAIROS generalize across models, both running SWE-Bench Verified. On *Qwen3-Coder-30B*, KAIROS reduces power by 34.5%, and on *Ministral-3-14B*, it reduces power by 28.9%. These results indicate that the benefit of KAIROS is not tied to a single model, but is broadly effective across different LLMs.

8.2 Generality Across Agent Harnesses and Models

We further broaden the evaluation using a production agent harness, Claude Code, and another model, Ministral-3-14B [35]. Table 2 evaluates KAIROS with Claude Code under different request loads. Across the evaluated loads, KAIROS reduces average GPU power by 9.0–36.7%, showing that its benefits extend to a production agent harness beyond simple ReAct-style agents. Table 3 evaluates KAIROS with Ministral-3-14B on DABStep and Terminal-Bench 2.0. KAIROS reduces average GPU power by 26.1% on DABStep and 21.9% on Terminal-Bench 2.0, demonstrating that its effectiveness is not tied to a specific model. Together, these results show that using context usage signal for reducing power generalizes beyond simple ReAct. Because KAIROS controls relative to available KV-cache capacity, it is also largely decoupled from a specific model or hardware platform.

Table 2. Power reduction with Claude Code under different request loads (JPS: jobs per second).

Load	Baseline	KAIROS	Power Reduction
0.001 JPS	354.71 W	224.45 W	36.72%
0.002 JPS	385.74 W	292.13 W	24.27%
0.005 JPS	390.12 W	354.94 W	9.02%

Table 3. Power reduction with Ministral-3-14B across additional benchmarks (JPS: jobs per second).

Benchmark / Load	Baseline	KAIROS	Power Reduction
DABStep, 0.03 JPS	341.57 W	252.46 W	26.09%
Terminal-Bench 2.0, 0.015 JPS	334.56 W	261.23 W	21.92%

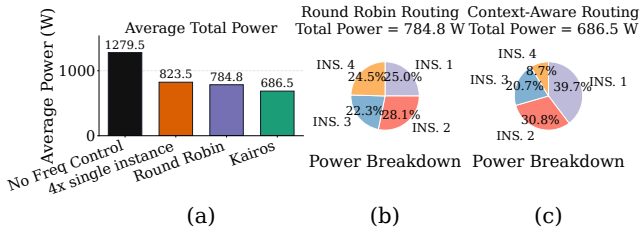


Figure 12. (a) Power comparison across four serving instances for no frequency control, a replicated single-instance baseline, a round-robin routing policy, and KAIROS with context-aware routing. (b,c) Per-instance power breakdown for round robin and KAIROS routing.

8.3 Multi-Instance Agentic Serving

Figure 12 demonstrates the effectiveness of KAIROS under multi-instance scaling at an input rate of 0.16 jobs/s. This experiment runs mini-swe-agent using SWE-Bench Verified dataset. Compared to the baseline without frequency control, KAIROS with round-robin routing reduces average power from 1279.5 W to 784.8 W, while its context-aware routing further lowers it to 686.5 W, achieving a total power reduction of 46.3%. The key difference lies in how work is distributed across instances: round-robin assigns load evenly, preventing GPUs from entering low-power idle states, whereas KAIROS 's context-aware routing consolidates work onto fewer GPUs when possible. This consolidation avoids the significant power overhead of lightly-loaded GPUs and enables other instances to remain near idle power. As shown in Figures 12(b,c), this results in a more efficient power distribution across instances, highlighting the benefit of coordinated routing and control. In addition, KAIROS's context-aware router achieves per agent p5 throughput of 26.9 tokens/s, surpassing the 25.4 tokens/s when using round robin policy.

Table 4. Runtime overhead of KAIROS.

Benchmark	Overhead	Percentage
DABStep	0.164 s	3.28%
SWE-bench Verified	0.142 s	3.29%
Terminal-Bench 2.0	0.180 s	2.90%

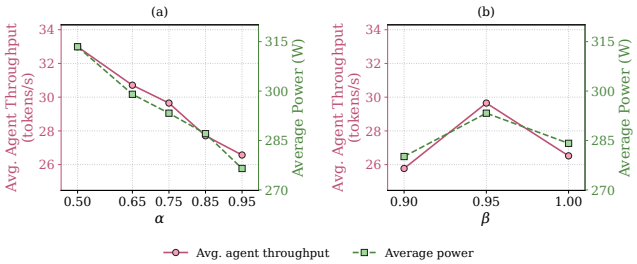


Figure 13. Impact of control parameters α and β in § 6.2.1 on agent throughput and power consumption. (a) α: frequency-scaling threshold, balancing power savings and throughput. (b) β: admission threshold, balancing utilization and thrashing avoidance.

8.4 Runtime Overhead

KAIROS only tracks lightweight per-agent metadata and routes agents across instances, so the added work is limited to bookkeeping and routing operations. Therefore, the tracking, placement, and routing overhead introduced by KAIROS is small. Table 4 reports the measured overhead across the three benchmarks. The overhead remains around 3% in all cases, confirming that KAIROS introduces only modest runtime cost.

8.5 Parameter Sensitivity

We evaluate the sensitivity of KAIROS to its two control policy parameters in §6.2.1, α and β, which govern frequency scaling and KV-cache headroom, respectively. The parameter α controls the power–performance trade-off: a higher α saves more power but can trigger more admission control and reduce throughput, while a lower α keeps the system at higher frequency and saves less power. Figure 13 (a) reports the sensitivity to α. As α increases, KAIROS achieves lower average power at the cost of lower agent throughput. We use $\alpha = 0.75$ as a balanced setting between power saving and performance. The parameter β controls how conservatively KAIROS reserves KV-cache headroom. A value that is too high can lead to oscillation as agent states grow and amplify overhead, while a value that is too low underutilizes the serving instance. Figure 13 (b) reports the sensitivity to β. Among the evaluated settings, $\beta = 0.95$ achieves the highest average agent throughput while maintaining moderate power consumption. We therefore use $\beta = 0.95$ as the default setting.

Table 5. Timeout rate and average power on SWE-bench Verified.

	No SLO		SLO = 25 tokens/s	
Jobs/s	Timeout	Power	Timeout	Power
0.04	0.00% (0/454)	206.64 W	0.00% (0/454)	235.33 W
0.08	0.12% (1/861)	287.51 W	0.00% (0/861)	310.32 W

8.6 KAIROS's Impacts on Task Failure Rate

Because KAIROS only adjusts execution frequency without modifying model outputs, its impact on task success can arise only indirectly through execution-time constraints such as task timeouts. We therefore evaluate the excessive tasks failure on SWE-bench Verified introduced by KAIROS. As shown in Table 5, at arrival rates of 0.04 and 0.08 jobs/s, KAIROS without an SLO incurs additional task failure rates due to timeout of 0.00% (0/454) and 0.12% (1/861), respectively, whereas enforcing a 25 tokens/s SLO results in no additional failures in either setting. These results indicate that the throughput-based SLO effectively prevents timeout-induced degradation in task completion.

9 Discussion

We discuss several aspects of KAIROS that clarify its scope and relationship to existing serving techniques, including its interaction with KV-cache offloading, the use of throughput as an online SLO for agent progress, and its complementarity with fine-grained DVFS.

Relationship to KV-cache offloading. KAIROS's context-aware DVFS is compatible with systems with KV-cache offloading enabled. While offloading allows the system to handle higher request loads, our LMCache measurements in Appendix §A.4 show that it still introduces noticeable overhead, which ongoing/recent works such as MORI [66], Continuum [32] and more [27, 59] seek to reduce. Therefore, DVFS with offloading enabled still needs to be context-aware: lowering GPU frequency can prolong agent lifetimes, increase aggregate context pressure, and induce additional offload/reload overhead that may offset the benefits from power saving. Joint optimization of DVFS and KV-cache offloading is an interesting direction for future work.

The choice of throughput-based SLO. Agentic workloads are goal-driven: users specify a high-level task, the agent autonomously explores toward completion, and the user-visible latency is end-to-end task-completion time. KAIROS uses per-agent generation throughput as an online proxy for end-to-end latency and sustained task progress, since higher token/s reduces end-to-end execution time while task-completion latency is unavailable at runtime. Fine-grained token-level latency such as per-turn TTFT/TBT is less visible to users during agentic execution.

Relationship to fine-grained DVFS. KAIROS complements fine-grained DVFS policies rather than replacing them. Existing approaches [6, 26, 36, 48, 55, 58, 61, 72] use local metrics such as TPOT or operator latency, while KAIROS provides a longer-term context-pressure signal that captures their impact on KV-cache residency.

10 Related Work

To the best of our knowledge, ***KAIROS is the first work to optimize power for agentic inference serving***. Below, we compare KAIROS with closely related works.

Power optimization in LLM serving. Prior work on power-efficient LLM serving mainly falls into two categories: DVFS [6, 26, 36, 48, 55, 58, 61, 72], which adjusts GPU frequency online using runtime signals such as slack, batch behavior, KV-cache growth, or phase-specific latency predictions, and system-level scheduling [33, 57, 63], which reduces power through hardware-aware placement and resource allocation. These works target mostly *stateless* LLM serving, where requests are treated as independent. In contrast, agentic serving is fundamentally ***stateful***: multi-turn conversations continuously grow and revisit context, so power optimization must account for the evolution of that state over time, which is the key focus of our work. How to generalize and integrate the idea of KAIROS to energy-first system designs as visioned in [10] is an interesting future work.

Power optimization in LLM training. Parallel to inference-side power optimization, another line of work focuses on reducing the power and energy consumption of LLM training [12, 55, 65]. Although some of the underlying mechanisms overlap with KAIROS, these techniques are tailored to the forward and backward passes of training and do not account for the evolving state inherent in agentic serving like KAIROS.

Energy/power characterization. Several recent works [4, 21, 39, 43, 44, 56, 62, 64] characterize, model, or benchmark the energy and power behavior of LLM inference rather than directly optimizing it. They study inference energy tradeoffs and control knobs, build workload-based energy/runtime models, profile fine-grained power use, quantify energy and carbon emissions, and propose benchmarking or reporting frameworks such as energy-per-token. A closely related work [28] on agents characterizes the infrastructure cost of dynamic reasoning and shows that agentic workflows introduce substantially higher resource and energy demands than conventional single-shot inference. In contrast, our work takes a step beyond characterization: we design a runtime mechanism that uses context growth as a control signal to actively reduce energy while maintaining safe, non-thrashing performance for agentic serving.

Agentic inference and stateful LLM serving. Recent systems optimize agentic workflows through workflow-aware runtimes and program semantics [8, 27, 34, 49–51], while others address multi-round behavior through cache retention and adaptive prefill/decode placement [20, 32]. In parallel, LLM serving work improves memory efficiency through KV-cache management, paging, offloading, scheduling, and disaggregation [2, 29, 30, 45, 47, 74]. Together, these works show stateful and memory-bound workload behavior, but focus mainly on performance. In contrast, our work targets *power optimization* in agentic workloads and explicitly accounts for the feedback loop between slower execution, increased agent lifetime, and growing context footprint, using this coupling to drive global control decisions.

Autoscaling and resource management for AI serving. Prior work on autoscaling and datacenter resource management provisions resources using signals such as queue length, utilization, token rate, and SLO violations [15, 16, 18, 24, 31, 37, 46, 54]. These methods match capacity to demand, but do not account for the stateful, highly dynamic nature of agentic workloads. In our setting, scaling must also control multi-turn context growth and keep per-instance demand within a safe non-thrashing regime.

11 Conclusion

This paper showed that agentic AI serving is fundamentally different from single-turn LLM serving due to its long-lived, dynamically evolving context and the emergence of a thrashing regime under power scaling. We found that reducing GPU frequency can unexpectedly worsen both performance and power efficiency by increasing context residency and triggering memory instability. Based on these insights, this paper designed KAIROS, a dynamic system that jointly manages frequency, concurrency, and routing using *context as a first-class signal*. Our results demonstrated that KAIROS enables significant average power savings of 27% while maintaining performance targets, highlighting the need to rethink power reduction for stateful agentic AI serving. This work opens the door to a broader class of context-aware, power-efficient runtimes for agentic AI. It further suggests that rethinking serving around state, memory stability, and dynamic control will be essential as agentic workloads continue to scale.

Acknowledgments

We used AI tools for code development, figure plotting, and text editing. We thank the anonymous reviewers for their constructive feedback and valuable suggestions. This work was supported in part by NSF grants CCF-2450085 and CNS-2106184 and DARPA ML2P Award HR0011-26-9-E190. This work was also supported in part by the Semiconductor Research Corporation (SRC) under the GRC AIHW grant, the AMD AI & HPC Cluster Program, and Advanced Micro Devices, Inc. through the Funding Academic Research program.

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 *Proceedings of the 18th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*. USENIX Association.

[2] Amey Agrawal, Haoran Qiu, Junda Chen, Íñigo Goiri, Chaojie Zhang, Rayyan Shahid, Ramachandran Ramjee, Alexey Tumanov, and Esha Choukse. 2024. No Request Left Behind: Tackling Heterogeneity in Long-Context LLM Inference with Medha. *arXiv preprint arXiv:2409.17264* (2024). doi:10.48550/arXiv.2409.17264

[3] Anthropic. 2025. Claude 3.7 Sonnet and Claude Code. https://www.anthropic.com/news/claude-3-7-sonnet. Accessed: 2026-09-08.

[4] Mauricio Fadel Argerich and Marta Patiño-Martínez. 2024. Measuring and Improving the Energy Efficiency of Large Language Models Inference. *IEEE Access* 12 (2024), 80187–80200. doi:10.1109/ACCESS.2024.3409745

[5] Zain Asgar, Michelle Nguyen, and Sachin Katti. 2025. Efficient and Scalable Agentic AI with Heterogeneous Systems. *arXiv preprint arXiv:2507.19635* (2025). doi:10.48550/arXiv.2507.19635

[6] Omar Basit, Yunzhao Liu, Z. Jonny Kong, and Y. Charlie Hu. 2026. BiScale: Energy-Efficient Disaggregated LLM Serving via Phase-Aware Placement and DVFS. *arXiv preprint arXiv:2602.18755* (2026). doi:10.48550/arXiv.2602.18755

[7] Belfer Center for Science and International Affairs. 2025. AI Data Centers and the U.S. Electric Grid. https://www.belfercenter.org/research-analysis/ai-data-centers-us-electric-grid. Accessed: 2026-04-14.

[8] Gohar Irfan Chaudhry, Esha Choukse, Haoran Qiu, Íñigo Goiri, Rodrigo Fonseca, Adam Belay, and Ricardo Bianchini. 2025. Murakkab: Resource-Efficient Agentic Workflow Orchestration in Cloud Platforms. *arXiv preprint arXiv:2508.18298* (2025). doi:10.48550/arXiv.2508.18298

[9] Yihua Cheng, Yuhan Liu, Jiayi Yao, Yuwei An, Xiaokun Chen, Shaoting Feng, Yuyang Huang, Samuel Shen, Kuntai Du, and Junchen Jiang. 2025. LMCache: An Efficient KV Cache Layer for Enterprise-Scale LLM Inference. *arXiv preprint arXiv:2510.09665* (2025). doi:10.48550/arXiv.2510.09665

[10] Mosharaf Chowdhury, Jae-Won Chung, Jeff J. Ma, Nishil Talati, and Ruofan Wu. 2026. Energy Calculus: A Compositional Algebra of Energy in Computational Systems. arXiv:2607.11087 [cs.DC] https://arxiv.org/abs/2607.11087

[11] Neil Chowdhury, James Aung, Chan Jun Shern, Oliver Jaffe, Dane Sherburn, Giulio Starace, Evan Mays, Rachel Dias, Marwan Aljubeh, Mia Glaese, Carlos E. Jimenez, John Yang, Leyton Ho, Tejal Patwardhan, Kevin Liu, and Aleksander Madry. 2024. Introducing SWE-bench Verified. https://openai.com/index/introducing-swe-bench-verified/

[12] Jae-Won Chung, Yile Gu, Insu Jang, Luoxi Meng, Nikhil Bansal, and Mosharaf Chowdhury. 2024. Reducing Energy Bloat in Large Model Training. In *Proceedings of the ACM SIGOPS 30th Symposium on Operating Systems Principles (SOSP '24)*. ACM, 144–159. doi:10.1145/3694715.3695970

[13] Jae-Won Chung, Jeff J. Ma, Ruofan Wu, Jiachen Liu, Oh Jun Kweon, Yuxuan Xia, Zhiyu Wu, and Mosharaf Chowdhury. 2025. The ML.ENERGY Benchmark: Toward Automated Inference Energy Measurement and Optimization. arXiv:2505.06371 [cs.LG] https://arxiv.org/abs/2505.06371

[14] Jae-Won Chung, Ruofan Wu, Jeff J. Ma, and Mosharaf Chowdhury. 2026. Where Do the Joules Go? Diagnosing Inference Energy Consumption. arXiv:2601.22076 [cs.LG] https://arxiv.org/abs/2601.22076

[15] Daniel Crankshaw, Gur-Eyal Sela, Corey Zumar, Xiangxi Mo, Joseph E. Gonzalez, Ion Stoica, and Alexey Tumanov. 2020. InferLine: Latency-Aware Provisioning and Scaling for Prediction Serving Pipelines. In *Proceedings of the 11th ACM Symposium on Cloud Computing (SoCC '20)*. 477–491. doi:10.1145/3419111.3421285

[16] Christina Delimitrou and Christos Kozyrakis. 2014. Quasar: Resource-Efficient and QoS-Aware Cluster Management. In *Proceedings of the 19th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS '14)*. doi:10.1145/2541940.2541941

[17] Alex Egg, Martin Iglesias Goyanes, Friso Kingma, Andreu Mora, Leandro von Werra, and Thomas Wolf. 2025. DABstep: Data Agent Benchmark for Multi-step Reasoning. *arXiv preprint arXiv:2506.23719* (2025). doi:10.48550/arXiv.2506.23719

[18] Ori Hadary, Luke Marshall, Ishai Menache, Abhisek Pan, Esaias E. Greeff, David Dion, Star Dorminey, Shailesh Joshi, Yang Chen, Mark Russinovich, and Thomas Moscibroda. 2020. Protean: VM Allocation Service at Scale. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*. https://www.usenix.org/conference/osdi20/presentation/hadary

[19] Harbor Framework Team. 2026. *Harbor: A framework for evaluating and optimizing agents and models in container environments.* https://github.com/harbor-framework/harbor

[20] Wenhao He, Youhe Jiang, Penghao Zhao, Quanqing Xu, Eiko Yoneki, Bin Cui, and Fangcheng Fu. 2026. Efficient Multi-round LLM Inference over Disaggregated Serving. *arXiv preprint arXiv:2602.14516* (2026). doi:10.48550/arXiv.2602.14516

[21] Erik Johannes Husom, Arda Goknil, Lwin Khin Shar, and Sagar Sen. 2024. The Price of Prompting: Profiling Energy Use in Large Language Models Inference. *arXiv preprint arXiv:2407.16893* (2024). doi:10.48550/arXiv.2407.16893

[22] International Energy Agency. 2025. *Energy and AI.* Technical Report. International Energy Agency. https://www.iea.org/reports/energy-and-ai

[23] International Energy Agency. 2026. *Electricity 2026: Grids.* Technical Report. International Energy Agency. https://www.iea.org/reports/electricity-2026/grids

[24] Shashwat Jaiswal, Kunal Jain, Yogesh Simmhan, Anjaly Parayil, Ankur Mallick, Rujia Wang, Renee St. Amant, Chetan Bansal, Victor Rühle, Anoop Kulkarni, Steve Kofsky, and Saravan Rajmohan. 2025. Serving Models, Fast and Slow: Optimizing Heterogeneous LLM Inferencing Workloads at Scale. *arXiv preprint arXiv:2502.14617* (2025). doi:10.48550/arXiv.2502.14617

[25] Carlos E. Jimenez, John Yang, Alexander Wettig, Kilian Lieret, Shunyu Yao, Kexin Pei, Ofir Press, and Karthik Narasimhan. 2024. SWE-bench: Can Language Models Resolve Real-World GitHub Issues?. In *The Twelfth International Conference on Learning Representations (ICLR)*. https://openreview.net/forum?id=VTF8yNQM66

[26] Andreas Kosmas Kakolyris, Dimosthenis Masouros, Petros Vavaroutsos, Sotirios Xydis, and Dimitrios Soudris. 2024. SLO-aware GPU Frequency Scaling for Energy Efficient LLM Inference Serving. *arXiv preprint arXiv:2408.05235* (2024). doi:10.48550/arXiv.2408.05235

[27] Hao Kang, Ziyang Li, Xinyu Yang, Weili Xu, Yinfang Chen, Junxiong Wang, Beidi Chen, Tushar Krishna, Chenfeng Xu, and Simran Arora. 2026. ThunderAgent: A Simple, Fast and Program-Aware Agentic Inference System. *arXiv preprint arXiv:2602.13692* (2026). doi:10.48550/arXiv.2602.13692

[28] Jiin Kim, Byeongjun Shin, Jinha Chung, and Minsoo Rhu. 2026. The Cost of Dynamic Reasoning: Demystifying AI Agents and Test-Time Scaling from an AI Infrastructure Perspective. In *2026 IEEE International Symposium on High-Performance Computer Architecture (HPCA)*. 1–16.

[29] Kihyun Kim, Jinwoo Kim, Hyunsun Chung, Myung-Hoon Cha, Hong-Yeon Kim, and Youngjae Kim. 2025. Cost-Efficient LLM Serving in the Cloud: VM Selection with KV Cache Offloading. *arXiv preprint arXiv:2504.11816* (2025). doi:10.48550/arXiv.2504.11816

[30] Woosuk Kwon, Zhuohan Li, Siyuan Zhuang, Ying Sheng, Lianmin Zheng, Cody Hao Yu, Joseph E. 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 '23)*. 611–626. doi:10.1145/3600006.3613165

[31] Ruiqi Lai, Hongrui Liu, Chengzhi Lu, Zonghao Liu, Siyu Cao, Siyang Shao, Yixin Zhang, Luo Mai, and Dmitrii Ustiugov. 2025. TokenScale: Timely and Accurate Autoscaling for Disaggregated LLM Serving with Token Velocity. *arXiv preprint arXiv:2512.03416* (2025). doi:10.48550/arXiv.2512.03416

[32] Hanchen Li, Qiuyang Mang, Runyuan He, Qizheng Zhang, Huanzhi Mao, Xiaokun Chen, Hangrui Zhou, Alvin Cheung, Joseph Gonzalez, and Ion Stoica. 2025. Continuum: Efficient and Robust Multi-Turn LLM Agent Scheduling with KV Cache Time-to-Live. *arXiv preprint arXiv:2511.02230* (2025). doi:10.48550/arXiv.2511.02230

[33] Yueying Li, Zhanqiu Hu, Esha Choukse, Rodrigo Fonseca, G. Edward Suh, and Udit Gupta. 2025. EcoServe: Designing Carbon-Aware AI Inference Systems. *arXiv preprint arXiv:2502.05043* (2025). doi:10.48550/arXiv.2502.05043

[34] 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 24)*. 929–945. https://www.usenix.org/conference/osdi24/presentation/lin-chaofan

[35] Alexander H. Liu, Kartik Khandelwal, Sandeep Subramanian, Victor Jouault, Abhinav Rastogi, et al. 2026. Ministral 3. *arXiv preprint arXiv:2601.08584* (2026).

[36] Qunyou Liu, Darong Huang, Marina Zapater, and David Atienza. 2026. GreenLLM: SLO-Aware Dynamic Frequency Scaling for Energy-Efficient LLM Serving. In *Proceedings of 63rd ACM/IEEE Design Automation Conference [i.e. The Chips to Systems Conference] (DAC '26)*. ACM. https://infoscience.epfl.ch/handle/20.500.14299/261894

[37] David Lo, Liqun Cheng, Rama Govindaraju, Parthasarathy Ranganathan, and Christos Kozyrakis. 2015. Heracles: Improving Resource Efficiency at Scale. In *Proceedings of the 42nd Annual International Symposium on Computer Architecture (ISCA '15)*. doi:10.1145/2749469.2749475

[38] Alexandra Sasha Luccioni, Yacine Jernite, and Emma Strubell. 2024. Power Hungry Processing: Watts Driving the Cost of AI Deployment?. In *Proceedings of the 2024 ACM Conference on Fairness, Accountability, and Transparency (FAccT '24)*. Association for Computing Machinery, 85–99. doi:10.1145/3630106.3658542

[39] Paul Joe Maliakel, Shashikant Ilager, and Ivona Brandic. 2025. Characterizing LLM Inference Energy-Performance Tradeoffs across Workloads and GPU Scaling. *arXiv preprint arXiv:2501.08219* (2025). doi:10.48550/arXiv.2501.08219

[40] Mike A. Merrill, Alexander G. Shaw, Nicholas Carlini, Boxuan Li, Harsh Raj, Ivan Bercovich, Lin Shi, Jeong Yeon Shin, Thomas Walshe, E. Kelly Buchanan, Junhong Shen, Guanghao Ye, Haowei Lin, Jason Poulos, Maoyu Wang, Marianna Nezhurina, Jenia Jitsev, Di Lu, Orfeas Menis Mastromichalakis, Zhiwei Xu, Zizhao Chen, Yue Liu, Robert Zhang, Leon Liangyu Chen, Anurag Kashyap, Jan-Lucas Uslu, Jeffrey Li, Jianbo Wu, Minghao Yan, Song Bian, Vedang Sharma, Ke Sun, Steven Dillmann, Akshay Anand, Andrew Lanpouthakoun, Bardia Koopah, Changran Hu, Etash Guha, Gabriel H. S. Dreiman, Jiacheng Zhu, Karl Krauth, Li Zhong, Niklas Muennighoff, Robert Amanfu, Shangyin Tan, Shreyas Pimpalgaonkar, Tushar Aggarwal, Xiangning Lin, Xin Lan, Xuandong Zhao, Yiqing Liang, Yuanli Wang, Zilong Wang, Changzhi Zhou, David Heineman, Hange Liu, Harsh Trivedi, John Yang, Junhong Lin, Manish Shetty, Michael Yang, Nabil Omi, Negin Raoof, Shanda Li, Terry Yue Zhuo, Wuwei Lin, Yiwei Dai, Yuxin Wang, Wenhao Chai, Shang Zhou, Dariush Wahdany, Ziyu She, Jiaming Hu, Zhikang Dong, Yuxuan Zhu, Sasha Cui, Ahson Saiyed, Arinbjörn Kolbeinsson, Jesse Hu, Christopher Michael Rytting, Ryan Marten, Yixin Wang, Alex Dimakis, Andy Konwinski, and Ludwig Schmidt. 2026. Terminal-Bench: Benchmarking Agents on Hard, Realistic Tasks in Command Line Interfaces. *arXiv preprint arXiv:2601.11868* (2026).

[41] MLCommons. 2026. MLPerf Inference: Datacenter. https://mlcommons.org/benchmarks/inference-datacenter/. Accessed: 2026-04-09.

[42] Rohit Mural, Ryan Rafaty, Vennila Varadarajan, and Yitian Xu. 2026. *AI, Data Centers, and the U.S. Electric Grid.* Technical Report. Belfer Center for Science and International Affairs, Harvard Kennedy School. https://www.belfercenter.org/sites/default/files/2026-02/Mural%20et%20al_AI%20Data%20Centers%20Grid_20260206.pdf

[43] Chenxu Niu, Wei Zhang, Jie Li, Yongjian Zhao, Tongyang Wang, Xi Wang, and Yong Chen. 2026. TokenPowerBench: Benchmarking the Power Consumption of LLM Inference. In *Proceedings of the AAAI Conference on Artificial Intelligence*, Vol. 40. 32582–32590. doi:10.1609/aaai.v40i38.40535

[44] Miray Özcan, Philipp Wiesner, Philipp Weiß, and Odej Kao. 2025. Quantifying the Energy Consumption and Carbon Emissions of LLM Inference via Simulations. *arXiv preprint arXiv:2507.11417* (2025). doi:10.48550/arXiv.2507.11417

[45] Pratyush Patel, Esha Choukse, Chaojie Zhang, Aashaka Shah, Íñigo Goiri, Saeed Maleki, and Ricardo Bianchini. 2024. Splitwise: Efficient Generative LLM Inference Using Phase Splitting. In *2024 ACM/IEEE 51st Annual International Symposium on Computer Architecture (ISCA)*. doi:10.1109/ISCA59077.2024.00019

[46] Archit Patke, Dhemath Reddy, Saurabh Jha, Chandra Narayanaswami, Zbigniew Kalbarczyk, and Ravishankar Iyer. 2025. Hierarchical Autoscaling for Large Language Model Serving with Chiron. *arXiv preprint arXiv:2501.08090* (2025). doi:10.48550/arXiv.2501.08090

[47] Ramya Prabhu, Ajay Nayak, Jayashree Mohan, Ramachandran Ramjee, and Ashish Panwar. 2025. vAttention: Dynamic Memory Management for Serving LLMs without PagedAttention. In *Proceedings of the 30th ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS '25)*. doi:10.1145/3669940.3707256

[48] Haoran Qiu, Weichao Mao, Archit Patke, Shengkun Cui, Saurabh Jha, Chen Wang, Hubertus Franke, Zbigniew Kalbarczyk, Tamer Başar, and Ravishankar K. Iyer. 2024. Power-aware Deep Learning Model Serving with μ-Serve. In *2024 USENIX Annual Technical Conference (USENIX ATC 24)*. 75–93. https://www.usenix.org/conference/atc24/presentation/qiu

[49] Ritik Raj, Hong Wang, and Tushar Krishna. 2025. A CPU-Centric Perspective on Agentic AI. *arXiv preprint arXiv:2511.00739* (2025). doi:10.48550/arXiv.2511.00739

[50] 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). doi:10.48550/arXiv.2511.00330

[51] Keshav Santhanam, Deepti Raghavan, Muhammad Shahir Rahman, Thejas Venkatesh, Neha Kunjal, Pratiksha Thaker, Philip Levis, and Matei Zaharia. 2024. ALTO: An Efficient Network Orchestrator for Compound AI Systems. In *Proceedings of the 4th Workshop on Machine Learning and Systems (EuroMLSys '24)*. 117–125. doi:10.1145/3642970.3655844

[52] Noah Shinn, Federico Cassano, Bailin Labash, Ashwin Gopinath, Karthik Narasimhan, and Shunyu Yao. 2023. Reflexion: Language Agents with Verbal Reinforcement Learning. In *Advances in Neural Information Processing Systems (NeurIPS)*, Vol. 36. https://proceedings.neurips.cc/paper_files/paper/2023/hash/1b44b878bb782e6954cd888628510e90-Abstract-Conference.html

[53] Mohit Shridhar, Xingdi Yuan, Marc-Alexandre Côté, Yonatan Bisk, Adam Trischler, and Matthew Hausknecht. 2021. ALFWorld: Aligning Text and Embodied Environments for Interactive Learning. In *The Ninth International Conference on Learning Representations (ICLR)*. https://arxiv.org/abs/2010.03768

[54] Gursimran Singh, Timothy Yu, Haley Li, Cheng Chen, Hanieh Sadri, Qintao Zhang, Yu Zhang, Ying Xiong, Yong Zhang, and Zhenan Fan. 2025. ElasticMoE: An Efficient Auto Scaling Method for Mixture-of-Experts Models. *arXiv preprint arXiv:2510.02613* (2025). doi:10.48550/arXiv.2510.02613

[55] Jeffrey Spaan, Kuan-Hsun Chen, and Ana-Lucia Varbanescu. 2026. Reducing Compute Waste in LLMs through Kernel-Level DVFS. *arXiv preprint arXiv:2601.08539* (2026). doi:10.48550/arXiv.2601.08539

[56] Jovan Stojkovic, Esha Choukse, Chaojie Zhang, Inigo Goiri, and Josep Torrellas. 2024. Towards Greener LLMs: Bringing Energy-Efficiency to the Forefront of LLM Inference. *arXiv preprint arXiv:2403.20306* (2024). doi:10.48550/arXiv.2403.20306

[57] Jovan Stojkovic, Chaojie Zhang, Íñigo Goiri, Esha Choukse, Haoran Qiu, Rodrigo Fonseca, Josep Torrellas, and Ricardo Bianchini. 2025. TAPAS: Thermal- and Power-Aware Scheduling for LLM Inference in Cloud Platforms. In *Proceedings of the 30th ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS '25)*. doi:10.1145/3676641.3716025

[58] Jovan Stojkovic, Chaojie Zhang, Íñigo Goiri, Josep Torrellas, and Esha Choukse. 2025. DynamoLLM: Designing LLM Inference Clusters for Performance and Energy Efficiency. In *Proceedings of the 2025 IEEE International Symposium on High-Performance Computer Architecture (HPCA)*. IEEE Computer Society, 1348–1362. doi:10.1109/HPCA61900.2025.00102

[59] Shubham Tiwari, Tapan Chugh, Nash Rickert, Simon Peter, Ratul Mahajan, and Haiying Shen. 2026. CacheWise: Understanding Workloads and Optimizing KVCache Management for Efficiently Serving LLM Coding Agents. arXiv:2606.16824 [cs.DC] https://arxiv.org/abs/2606.16824

[60] Noppanat Wadlom, Junyi Shen, and Yao Lu. 2026. Efficient LLM Serving for Agentic Workflows: A Data Systems Perspective. *arXiv preprint arXiv:2603.16104* (2026). doi:10.48550/arXiv.2603.16104

[61] Zibo Wang, Yijia Zhang, Fuchun Wei, Bingqiang Wang, Yanlin Liu, Zhiheng Hu, Jingyi Zhang, Xiaoxin Xu, Jian He, Xiaoliang Wang, Wanchun Dou, Guihai Chen, and Chen Tian. 2025. Using Analytical Performance/Power Model and Fine-Grained DVFS to Enhance AI Accelerator Energy Efficiency. In *Proceedings of the 30th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 1* (Rotterdam, Netherlands) *(ASPLOS '25)*. Association for Computing Machinery, New York, NY, USA, 1118–1132. doi:10.1145/3669940.3707231

[62] Patrick Wilhelm, Thorsten Wittkopp, and Odej Kao. 2025. Beyond Test-Time Compute Strategies: Advocating Energy-per-Token in LLM Inference. In *Proceedings of the 5th Workshop on Machine Learning and Systems (EuroMLSys '25)*. Association for Computing Machinery, Rotterdam, Netherlands, 1–8. doi:10.1145/3721146.3721953

[63] Grant Wilkins, Srinivasan Keshav, and Richard Mortier. 2024. Hybrid Heterogeneous Clusters Can Lower the Energy Consumption of LLM Inference Workloads. In *Proceedings of the 15th ACM International Conference on Future and Sustainable Energy Systems* (Singapore, Singapore) *(e-Energy '24)*. Association for Computing Machinery, New York, NY, USA, 506–513. doi:10.1145/3632775.3662830

[64] Grant Wilkins, Srinivasan Keshav, and Richard Mortier. 2024. Offline Energy-Optimal LLM Serving: Workload-Based Energy Models for LLM Inference on Heterogeneous Systems. In *Proceedings of the 3rd ACM HotCarbon Workshop on Sustainable Computer Systems (HotCarbon '24)*. Association for Computing Machinery, Santa Cruz, CA, USA, 1–7. doi:10.1145/3727200.3727217

[65] Ruofan Wu, Jae-Won Chung, and Mosharaf Chowdhury. 2026. Kareus: Joint Reduction of Dynamic and Static Energy in Large Model Training. arXiv:2601.17654 [cs.LG] https://arxiv.org/abs/2601.17654

[66] Tian Xia, Hanchen Li, Zhifei Li, Xiaokun Chen, Hao Kang, Yifan Qiao, Yi Xu, and Ion Stoica. 2026. Idleness is Relative: Exploiting Tool-Call Idle Windows for Offloading in Agentic Systems with MORI. arXiv:2606.00866 [cs.OS] https://arxiv.org/abs/2606.00866

[67] John Yang, Carlos E Jimenez, Alexander Wettig, Kilian Lieret, Shunyu Yao, Karthik R Narasimhan, and Ofir Press. 2024. SWE-agent: Agent-Computer Interfaces Enable Automated Software Engineering. In *The Thirty-eighth Annual Conference on Neural Information Processing Systems*. https://arxiv.org/abs/2405.15793

[68] Shunyu Yao, Howard Chen, John Yang, and Karthik Narasimhan. 2022. WebShop: Towards Scalable Real-World Web Interaction with Grounded Language Agents. In *Advances in Neural Information Processing Systems (NeurIPS)*, Vol. 35. https://proceedings.neurips.cc/paper_files/paper/2022/hash/82ad13ec01f9fe44c01cb91814fd7b8c-Abstract-Conference.html

[69] Shunyu Yao, Jeffrey Zhao, Dian Yu, Nan Du, Izhak Shafran, Karthik Narasimhan, and Yuan Cao. 2023. ReAct: Synergizing Reasoning and Acting in Language Models. In *The Eleventh International Conference on Learning Representations (ICLR)*.

[70] Jie You, Jae-Won Chung, and Mosharaf Chowdhury. 2023. Zeus: Understanding and Optimizing GPU Energy Consumption of DNN Training. In *USENIX NSDI*.

[71] 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 *Proceedings of the 16th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*. USENIX Association, 521–538.

[72] Jiahuan Yu, Aryan Taneja, Junfeng Lin, and Minjia Zhang. 2025. VoltanaLLM: Feedback-Driven Frequency Control and State-Space Routing for Energy-Efficient LLM Serving. arXiv:2509.04827 [cs.DC] https://arxiv.org/abs/2509.04827

[73] Wei Zhang, Zhiyu Wu, Yi Mu, Rui Ning, Banruo Liu, Nikhil Sarda, Myungjin Lee, and Fan Lai. 2026. JITServe: SLO-aware LLM Serving with Imprecise Request Information. In *23rd USENIX Symposium on Networked Systems Design and Implementation (NSDI 26)*. USENIX Association. https://www.usenix.org/conference/nsdi26/presentation/zhang-wei

[74] 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 24)*. 193–210. https://www.usenix.org/conference/osdi24/presentation/zhong-yinmin

[75] Andy Zhou, Kai Yan, Michal Shlapentokh-Rothman, Haohan Wang, and Yu-Xiong Wang. 2024. Language Agent Tree Search Unifies Reasoning, Acting, and Planning in Language Models. In *Proceedings of the 41st International Conference on Machine Learning (Proceedings of Machine Learning Research, Vol. 235)*. PMLR, 62138–62160. https://proceedings.mlr.press/v235/zhou24r.html

Algorithm 2 KAIROS Initial Assignment Algorithm

Require: Serving instances $\mathcal{I} = \{1, \ldots, N\}$; per-instance capacity $U_{\max}$; consolidation threshold θ_{cons}
Ensure: Assigned instance $\text{inst}(a)$ for each newly arrived agent a

Initialize $\mathcal{A}_i \leftarrow \emptyset$ for all $i \in \mathcal{I}$
function AssignAgent(a)
  Update ongoing context usage $U_i \leftarrow \sum_{b \in \mathcal{A}_i} C^b$ for all $i \in \mathcal{I}$
  // Initial assignment policy (§6.3)
  $\mathcal{I}_{\text{light}} \leftarrow \{i \in \mathcal{I} \mid U_i < \theta_{\text{cons}} U_{\max}\}$
  if $\mathcal{I}_{\text{light}} \neq \emptyset$ **then**
    $i^\star \leftarrow \min \mathcal{I}_{\text{light}}$
  else
    $i^\star \leftarrow \arg\min_{i \in \mathcal{I}} U_i$
  end if
  Assign agent a to instance $i^\star$
  $\mathcal{A}_{i^\star} \leftarrow \mathcal{A}_{i^\star} \cup \{a\}$
  $\text{inst}(a) \leftarrow i^\star$
  $s_a \leftarrow 0$
end function

Algorithm 3 KAIROS Reassignment Algorithm

Require: Serving instances $\mathcal{I} = \{1, \ldots, N\}$; reassignment interval T_{reassign}; imbalance threshold ρ
Ensure: Updated assignment $\text{inst}(a)$ when agent a issues a request

function MaybeReassign(a)
  $s_a \leftarrow s_a + 1$
  if $s_a < T_{\text{reassign}}$ **then**

    return
  end if
  Update ongoing context usage $U_i \leftarrow \sum_{b \in \mathcal{A}_i} C^b$ for all $i \in \mathcal{I}$
  // Reassignment policy (§6.3)
  $i \leftarrow \text{inst}(a)$
  $j \leftarrow \arg\min_{m \in \mathcal{I}} U_m$
  if $j \neq i$ **and** $U_i \geq \rho \cdot U_j$ **then**
    Reassign agent a from instance i to instance j
    $\mathcal{A}_i \leftarrow \mathcal{A}_i \setminus \{a\}$
    $\mathcal{A}_j \leftarrow \mathcal{A}_j \cup \{a\}$
    $\text{inst}(a) \leftarrow j$
  end if
  $s_a \leftarrow 0$
end function

Algorithm 1 KAIROS Per-Instance Controller Algorithm

Require: Frequency levels $\mathcal{F} = \{f^{(1)}, \ldots, f^{(L)}\}$; capacity $U_{\max}$; SLO target τ; power budget $P_{\max}$; thresholds α, β, γ
Ensure: Frequency level ℓ and active set $\mathcal{A}^{\text{on}}$ at each epoch

Initialize $\ell \leftarrow L$, $c \leftarrow 0$, $\mathcal{A}^{\text{on}} \leftarrow \emptyset$, $\mathcal{A}^{\text{pend}} \leftarrow \emptyset$
loop
  $\mathcal{A}^{\text{pend}} \leftarrow \mathcal{A}^{\text{pend}} \cup$ new tasks
  $U_k \leftarrow \sum_{a \in \mathcal{A}^{\text{on}}} C^a$
  Update per-agent throughput for all $a \in \mathcal{A}^{\text{on}} \cup \mathcal{A}^{\text{pend}}$
  Update average system power $\bar{P}$
  // Context-aware frequency control (§6.2.2)
  if $U_k \geq \alpha U_{\max}$ **then**
    $\ell \leftarrow L$
  else
    $\ell \leftarrow \left\lfloor \frac{U_k}{\alpha U_{\max}} \cdot (L-1) \right\rfloor + 1$
  end if
  // SLO-based boosting (§6.2.3)
  $\tau_{\min} \leftarrow \min_{a \in \mathcal{A}^{\text{on}} \cup \mathcal{A}^{\text{pend}}} \text{Throughput}(a)$
  if $\tau_{\min} < \tau$ **then**
    $\ell \leftarrow L$
  end if
  // Thrashing avoidance (§6.2.4)
  while $\mathcal{A}^{\text{pend}} \neq \emptyset$ **and** $U_k < \gamma U_{\max}$ **do**
    Move next agent from $\mathcal{A}^{\text{pend}}$ to $\mathcal{A}^{\text{on}}$
    Update U_k
  end while
  if $U_k > \beta U_{\max}$ **then**
    Defer new admissions
  end if
  Apply frequency level ℓ
end loop

A Appendix

A.1 KAIROS Per-Instance Controller Pseudo-code

Algorithm 1 presents the pseudo-code for algorithm that governs per-instance frequency controller in KAIROS that employs context-aware frequency control, SLO-based boosting, and thrashing avoidance.

A.2 KAIROS Global Router Pseudo-code

Algorithm 2 and Algorithm 3 together present the pseudo-code for KAIROS 's global context-aware router, which performs initial agent placement and dynamic reassignment across serving instances to enable consolidation under low load and balance context pressure under high load. The detailed explanation for the Algorithm 2 and 3 is presented in details in §6.3.

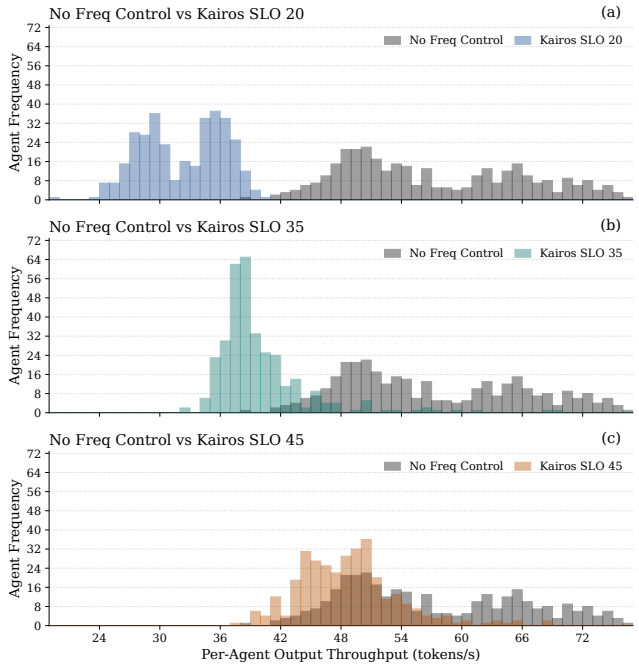


Figure 16. Per-agent throughput distribution comparing a no frequency control baseline and KAIROS with different SLO targets of 20, 35, and 45 tokens/s.

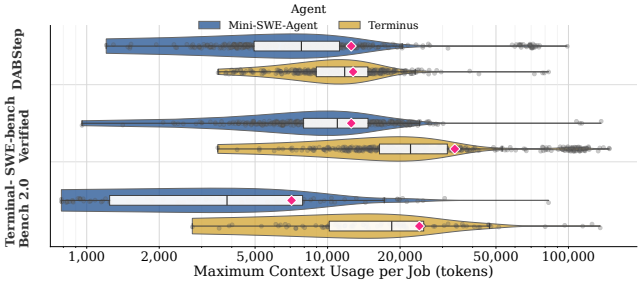


Figure 14. Distribution of maximum context length for different agents.

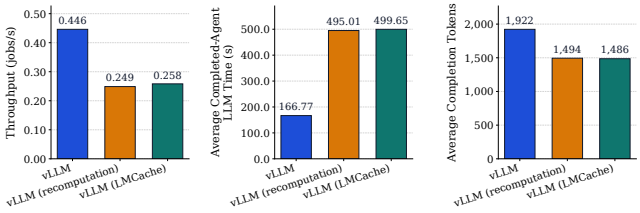


Figure 15. Average throughput (left), agent LLM time (middle), and average completion token thorughput (right) comparison of a non-thrashing vLLM baseline (0.5 jobs/s) with two thrashing baselines with recomputation and LMCache-based offloading (both 0.6 jobs/s). Across three latency and throughput metrics, the thrashing regime leads to severe performance degradation.

A.3 Context Usage Variation

Figure 14 uses the same workload setup as §3.1 and complements Figure 3 by showing the distribution of the *maximum context size* reached by each agent during execution. While Figure 3 characterizes variability in the number of conversation turns and total time spent in LLM execution, this figure captures the resulting peak memory footprint of each agent's context.

The key trend is consistent across all the benchmarks. Maximum context usage varies widely both within and across agent–dataset pairs, with a pronounced long tail of agents whose contexts grow to very large sizes. While many agents reach peak contexts on the order of only tens of thousands of tokens, a non-trivial subset grows to hundreds of thousands of tokens. This result reinforces that agentic serving creates highly heterogeneous and difficult-to-predict memory demand, where a small number of long-context agents can disproportionately stress the serving system.

A.4 System Performance in Thrashing Regime

While operating in the thrashing regime, systems typically employ either (1) recomputation of evicted tokens [27, 32] or (2) offloading context to CPU memory or lower-tier storage using frameworks like LMCache [9]. Figure 15 evaluates these two strategies (i.e., recomputation and offloading) against a baseline operating in the non-thrashing region. To understand the effect of thrashing, we construct a controlled experiment using a short-running mini-swe-agent request from SWE-bench Verified whose conversation length is at the 25th percentile (i.e., 18 turns), and replicate this request in the input stream. This setup simplifies regime control, while the resulting insight on the cost of thrashing is independent of the specific agent or request distribution. We replay the same request at two arrival rates: 0.5 and 0.6 requests/second, which place the system in the non-thrashing and thrashing regimes, respectively. We then compare performance across the two regimes. Our evaluation across three metrics reveals significant performance degradation for thrashing.

- **System Throughput:** Recomputation and offloading reduce average agent completion throughput by 44.2% and 42.1%, respectively.
- **LLM Latency:** The average time spent in LLM calls increases by 3× for both recomputation and offloading.
- **Effective Decode Throughput:** Excluding recomputation cycles, useful decode throughput drops by 22.3%.

These results quantify the sub-optimal nature of thrashing-heavy serving. Performance is limited by either *redundant compute cycles* in the recomputation baseline or *starvation of GPU compute resources* due to CPU-GPU data transfers.

Figure 17. Launching `id-tracker` with Harbor and a context-aware router

```
# `id-tracker` wraps an ordinary Harbor launch.
# It inherits the parent environment,
# reads the per-agent API token from env,
# generates an agent name internally,
# and forwards requests through the
# ctx-aware-router.

CTX_AWARE_ROUTER_URL="http://127.0.0.1:24157" \
OPENAI_BASE_URL="http://127.0.0.1:24157/v1" \
OPENAI_API_KEY=YOUR_KEY \
python -m id_tracker \
  --router-url "${CTX_AWARE_ROUTER_URL}" \
  -- \
  harbor run \
    --agent mini-swe-agent \
    --model \
        hosted_vllm/Qwen3-Coder-30B-A3B-Instruct \
    --dataset terminal-bench@2.0
```

A.5 Per Agent Throughput Distribution

To explain Figure 8 further, Figure 16 shows the distribution of per-agent throughput for the no frequency control baseline and KAIROS under different SLO targets of 20, 35, and 45 tokens/s. Across all settings, KAIROS shifts the throughput distribution left compared to the baseline, indicating that it deliberately slows down execution to reduce power. However, this shift is controlled: the bulk of the distribution is aligned with the respective SLO targets, showing that KAIROS adapts effectively to different performance requirements rather than applying a one-size-fits-all policy.

Importantly, the figure demonstrates that KAIROS slows agents just enough to meet the SLO without excessive overprovisioning. Under loose SLOs (e.g., 20 tokens/s), throughput is tightly clustered near the target, maximizing power savings by avoiding unnecessary high-frequency execution. As the SLO tightens (35 and 45 tokens/s), the distribution shifts right accordingly, reflecting higher frequencies to meet stricter performance demands. This behavior highlights that KAIROS achieves its core objective: it dynamically balances power and performance by reducing excess throughput while still maintaining SLO attainment across varying targets.

A.6 KAIROS ID Tracker Interface Example

We illustrate how KAIROS integrates with existing agent frameworks through a minimal, drop-in launch interface in Figure 17.

B Discussion

This section discusses two design and evaluation choices in detail.

B.1 PD Disaggregation

In this work, we characterize the complex interplay between frequency/power control, serving system performance, and context usage for serving agentic LLM workloads. This is a common new problem faced by different serving implementation like the PD aggregated/disaggregated instances, both need to carefully deal with the persistent agent context across different requests. In these settings, context becomes a first-class resource, and its dynamic usage directly shapes both system throughput and power-performance tradeoffs. While our design is instantiated on a PD-aggregated architecture, the underlying challenge is fundamentally driven by context dynamics rather than a specific execution model. We leave a full exploration of these techniques in PD-disaggregated settings to future work, where additional coordination overheads may arise.

B.2 Choice of Request Arrival Rate

In real-world deployments, a given hardware configuration can sustain only a limited input request arrival rate before performance degrades. As demand increases beyond this capacity, data center operators must scale up or scale out by using either more powerful hardware or higher number of hardware instances to maintain throughput and SLO. Therefore, in this work, we select input arrival rates that are commensurate with the capabilities of our experimental hardware, ensuring that the system operates in a realistic and meaningful regime. While higher arrival rates can be evaluated, doing so would require proportionally more GPU resources. Our goal is to study power–performance tradeoffs under representative operating conditions for a fixed hardware setup.

C Artifact Appendix

C.1 Abstract

This artifact accompanies our paper "KAIROS: Stateful, Context-Aware Power-Efficient Agentic Inference Serving." It contains the KAIROS implementation, post-processing tools, and plotting scripts. The supported reviewer workflow replays recorded Terminal-Bench 2.0 and DABStep agent conversations on Qwen3-Coder-30B FP8; Six experiments reproduce the Figure 7 comparison among no frequency control, fixed 810-MHz frequency, and KAIROS, and the Figures 10–11 comparison showing that context-aware admission prevents thrashing.

C.2 Artifact check-list (meta-information)

- **Algorithm:** Instance-level GPU DVFS; context-aware admission control; deterministic agent-conversation replay.
- **Program:** Python 3.12, Bash, Docker Compose, vLLM, KAIROS gateways and controllers, Zeus, and post-processing/plotting tools.
- **Compilation:** Not required; Python dependencies are locked by `uv.lock`.
- **Models:** Qwen/Qwen3-Coder-30B-A3B-Instruct-FP8 for the reviewer flow.
- **Data set:** Four recorded replay plans derived from SWE-Bench Verified, DABStep, and Terminal-Bench 2.0 agent runs.
- **Run-time environment:** Linux, Docker with Compose, NVIDIA Container Toolkit, an NVIDIA driver, and a running `zeusd`.
- **Hardware:** One NVIDIA H100 with sudo access for the six reviewer experiments
- **Run-time state:** Model weights, the published container image, verified replay plans, an available GPU/port profile, and a writable `/var/run/zeusd.sock`.
- **Execution:** Generate fixed experiment bundles, replay the conversations, post-process logs, report metrics, and render figures.
- **Metrics:** P5 per-agent output throughput, job throughput, GPU power, energy per completed agent, context occupancy, and latency.
- **Output:** Machine-readable summaries, tabular reports, and PDFs corresponding to the evaluated paper figures.
- **Experiments:** Six reviewer experiments for Figure 7 and Figures 10–11;
- **Disk space required:** 100GB
- **Time to prepare workflow:** Typically tens of minutes, depending on network and storage speed, excluding model and container downloads.
- **Time to complete experiments:** Approximately 18 H100 GPU hours of measured replay time for the six reviewer runs, plus setup and post-processing
- **Publicly available:** Yes; code is on GitHub and replay data is on Figshare.
- **Workflow automation:** Bash and Python scripts, `uv`, and Docker Compose.
- **Archived:** Code DOI: https://doi.org/10.6084/m9.figshare.34184400 ; data DOI: https://doi.org/10.6084/m9.figshare.33236611.v2.

C.3 Description

C.3.1 How to access. The source code is publicly available at https://doi.org/10.6084/m9.figshare.34184400, and the replay plans are available at https://doi.org/10.6084/m9.figshare.33236611.v2. The entry points are `README.data.md` for required data preparation, `README.ae.md` for the supported reviewer workflow.

C.3.2 Hardware dependencies. The reviewer workflow requires one NVIDIA H100 NVL with sudo access. GPU frequency control requires access to a `zeusd` daemon through `/var/run/zeusd.sock`.

C.3.3 Software dependencies. The host requires Linux, Python 3.12, `uv`, Docker with Compose, NVIDIA Container Toolkit, `curl`, `tar`, `md5sum`, and `sha256sum`. Python packages are pinned in `uv.lock`. Serving uses the published image `yichaoyuan/vllm-vllm-openai:v0.14.0-otel-lp-lmcache`, which includes vLLM 0.14.0 and the artifact's serving extensions. Hugging Face access is needed if the model is not already cached.

C.3.4 Data sets. The version-2 Figshare archive contains exactly four replay-plan files with a total logical size of approximately 29.8 GB. `README.data.md` records the immutable download, byte count, archive MD5, extraction paths, and per-file SHA-256 verification. The reviewer flow uses the DABStep and Terminal-Bench plans.

C.3.5 Models. All reviewer experiments use Qwen/Qwen3-Coder-30B-A3B-Instruct-FP8.

C.4 Installation

Clone the code repository and enter its root. First follow `README.data.md` to download, verify, and extract the replay plans. Then run `uv sync --frozen` and `uv run python ae/check_env.py --gpu 0 --port-profile 0`. The optional `--verify-checksums` flag performs the complete SHA-256 validation through the environment checker. If the administrator does not provide `zeusd`, start it as described in `README.ae.md`; set the Hugging Face cache and token variables only when model download is required.

C.5 Experiment workflow

1. Complete the download, extraction, and four SHA-256 checks in `README.data.md`.
2. Generate the six exact bundles with `uv run python ae/generate_experiments.py --gpu 0 --port-profile 0`.
3. Run the six keys listed in `README.ae.md` sequentially with `uv run python ae/run_experiment.py KEY`. Each command starts the matching vLLM/gateway stack, performs one three-hour replay, and tears the stack down.

4. Run `uv run python ae/postprocess.py` followed by `uv run python ae/report.py`. Materialize and plot Figures 10 and 11 using the commands in `README.ae.md`.

C.6 Evaluation and expected results

For the six-experiment AE workflow, the reproduced Terminal-Bench result from Figure 7 and the DABStep results from Figures 10 and 11 should match the corresponding results reported in the paper, allowing for small differences due to host and GPU measurement noise.