TURBOSERVE: A SERVING FRAMEWORK FOR STREAMING VIDEO GENERATION

Anonymous authors
Paper under double-blind review

ABSTRACT

Streaming video generation is emerging as a new serving workload in which users interact with long-lived sessions that generate video progressively, chunk by chunk. Unlike offline video generation or typical LLM serving, streaming video generation must preserve session state across active and idle periods, repeatedly schedule ongoing sessions, and deliver each chunk under a tight latency target. This creates two key serving challenges in multi-user, multi-GPU environments: session duration heterogeneity, where long-running sessions make placement decisions suboptimal over time, and temporal user-demand heterogeneity, where the number of active sessions fluctuates sharply across bursts and idle periods. We present TURBOSERVE, the *first* serving system designed specifically for streaming video generation workloads. TURBOSERVE formulates serving as an online scheduling problem that jointly coordinates session placement and GPU provisioning. Its *closed-loop scheduling algorithm* combines a migration-aware placement controller, which rebalances sessions across GPUs to reduce the maximum per-chunk latency, with a load-driven autoscaling controller, which adapts the GPU budget to workload variation for improved cost efficiency. To support these decisions at runtime, TURBOSERVE implements coalesced chunk processing for batching concurrent active sessions on the same GPU, GPU-CPU offloading for session suspension and resumption, and GPU-GPU migration for online rebalancing. We evaluate TURBOSERVE on real-world production traces from COMPANYX[1] across multiple model sizes and GPU clusters with up to 64 NVIDIA B300 GPUs. Compared with baseline serving configurations, TURBOSERVE reduces worst-case per-chunk latency by 37.5% and total GPU operating cost by 37.2% on average. We provide an anonymized code artifact.

1 INTRODUCTION

Video generation models, such as Sora Brooks et al. (2024), HunyuanVideo Kong et al. (2024), and Wan Wan et al. (2025), have achieved remarkable progress in synthesizing high-fidelity, temporally consistent videos from natural language prompts. These advances have unlocked a broad range of applications spanning film production, advertising, education, gaming, and personalized content creation. Traditional video generation pipelines operate in an offline, one-shot fashion: the user submits a prompt and waits seconds to minutes for a complete video to be returned, with no opportunity to steer the output mid-generation. This offline generation paradigm is now being supplanted by a new wave of streaming video generation pipelines, such as StreamDiffusionV2 Feng et al. (2025), Self-Forcing Huang et al. (2026), HY-

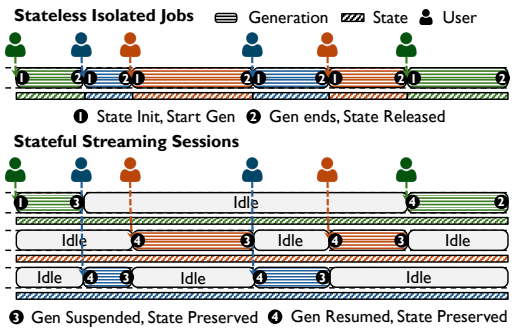


Figure 1: Motivation for streaming video generation serving. Stateless isolated generation jobs versus stateful streaming generation sessions.

[1]The company name is anonymized for double-blind review.

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

WorldPlay HunyuanWorld (2025), and LongLive Yang et al. (2025), which generate video progressively chunk by chunk, accept user prompts and interaction signals on-the-fly, and emit each chunk under a tight per-chunk latency target so that users observe the video as it is being generated. *This paradigm shift motivates a serving system purpose-built for streaming video generation.*

Current diffusion-based video generation systems, such as FastVideo FastVideo Team (2025), xDiT Fang et al. (2024), and vLLM-Omni Yin et al. (2026), are primarily optimized for stateless, one-shot generation requests, where each request can be scheduled, resourced, and completed in isolation. Streaming video generation violates this stateless serving assumption: serving is organized around persistent sessions that preserve prompt context and cached temporal states while continuously accepting new prompts or interaction signals. As a result, the system must coordinate evolving session states over time rather than treating each generation as an isolated job, as illustrated in Figure 1. Building an efficient serving system for streaming video generation in a multi-user, multi-GPU environment introduces two primary dimensions of heterogeneity:

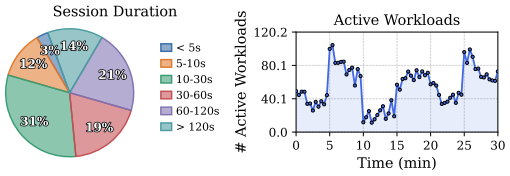


Figure 2: Streaming workload characteristics from COMPANYX's production trace. **Left:** Session duration distribution. **Right:** Number of active sessions over a 30-minute period.

- **Challenge 1: Session duration heterogeneity.** Different sessions persist for vastly different durations, ranging from a single short clip to tens of minutes of iterative prompting and refinement, as illustrated in Figure 2 (Left). Systems that treat sessions as fungible short-lived requests make placement decisions that become suboptimal as long-running sessions accumulate. Moreover, session activity evolves over time, so a placement that is initially balanced can become highly imbalanced as users transition between active and idle states.
- **Challenge 2: Temporal user-demand heterogeneity.** Within and across sessions, demand fluctuates sharply over time: users alternate between bursts of activity and idle periods, and the number of active sessions rises and falls as new sessions arrive and existing ones complete, as illustrated in Figure 2 (Right). Systems with static provisioning face a fundamental trade-off: provisioning for peak demand over-provisions GPUs and wastes resources during low-demand periods, while provisioning for average demand under-provisions the system during bursts and prevents sessions from sustaining real-time generation.

Resolving these challenges requires coordinated control over both session placement and GPU provisioning. Long-lived sessions need to be dynamically placed and migrated across GPUs so that load remains balanced as sessions arrive, depart, and transition between active and idle, while the provisioned GPU budget needs to adapt to workload demand. Critically, these two controls are tightly coupled: the available GPU budget bounds which placements are feasible, while the current placement determines whether that budget is sufficient. This motivates a closed-loop design that jointly manages session placement and GPU provisioning for streaming video generation.

To address these challenges, we propose TURBOSERVE, to the best of our knowledge, the first serving system designed specifically for streaming video generation workloads in multi-user, multi-GPU environments. TURBOSERVE jointly coordinates session placement and GPU provisioning under heterogeneous session durations and dynamic user demand. Our contributions are summarized as follows:

- **Contribution 1:** We formulate streaming video generation serving as an online scheduling problem that jointly determines session placement and GPU provisioning under per-chunk latency constraints. We design a closed-loop scheduling algorithm combining a migration-aware placement controller with a load-driven autoscaling controller. The placement controller performs event-driven min-max rebalancing to reduce worst-case per-chunk latency, while the autoscaling controller adjusts the GPU budget based on runtime load feedback to improve cost efficiency.
- **Contribution 2:** We implement TURBOSERVE with runtime support for multi-session chunk execution and session mobility. TURBOSERVE uses coalesced chunk processing for concurrent active sessions, GPU-CPU offloading for idle-session suspension/resumption, and direct GPU-GPU state migration for online rebalancing.

- **Contribution 3:** We evaluate TURBOSERVE on real-world production traces from COMPANYX across multiple video generation model sizes. Experimental results show that TURBOSERVE reduces worst-case per-chunk latency by 37.5% and total GPU operating cost by 37.2% on average.

2 BACKGROUND AND MOTIVATION

2.1 STREAMING VIDEO GENERATION AND SERVING

Streaming video generation serves a user session rather than a one-shot generation request (Figure 1). In the workflow adopted by autoregressive streaming video models such as Self-Forcing Huang et al. (2026), the model first produces an initial video chunk from the user prompt, then generates subsequent chunks conditioned on the prompt together with previously generated chunks and their cached state. During the session, the user may provide new prompts or interaction signals, or may become temporarily idle, causing the session to transition between active and idle periods. The serving system must therefore repeatedly schedule the same long-lived session, preserve its generation state, and deliver each video chunk within a tight latency target.

Recent systems such as StreamDiffusionV2 Feng et al. (2025), StreamingT2V Henschel et al. (2025), LongLive Yang et al. (2025), Self-Forcing Huang et al. (2026), and HYWorldPlay Hunyuan-World (2025) demonstrate the shift from offline video synthesis toward long-horizon, interactive, and real-time streaming generation. These model-level advances create a new class of workloads in which video chunks must be generated continuously and delivered with low latency. However, most existing work focuses on model architecture, inference pipeline design, or generation quality, rather than multi-user serving efficiency at the cluster level.

Recent LLM serving systems have substantially improved online inference efficiency Zheng et al. (2024); Jiang et al. (2023); Zhang et al. (2025); Yu et al. (2022); Zhong et al. (2024); Wu et al. (2024); Mao et al. (2025); Wu et al. (2023); Ye et al. (2025). However, streaming video generation differs from typical request-centric serving in two important aspects. First, *state lifetime*: streaming video generation must preserve session state across multiple active and idle periods. Second, *latency target*: each active session must repeatedly deliver video chunks within a tight per-chunk latency target. Existing autoscaling systems Sun et al. (2024); Xiang et al. (2025); Chen (2025); Li et al. (2025); Fu et al. (2024) dynamically adjust GPU resources under fluctuating demand, but streaming generation additionally requires scaling decisions to coordinate with the suspension, resumption, migration, and redistribution of long-lived session states. We provide a more detailed discussion of related work in Appendix A.

2.2 BASIC STREAMING SERVING RUNTIME

Given these distinct serving characteristics, we build on $\text{TURBOSERVE}_{\text{base}}$, the production runtime framework tailored for streaming video generation in COMPANYX. It provides the basic runtime support required by streaming video generation: (**i**) concurrent chunk execution under per-chunk latency constraints, (**ii**) persistent session-state and lifecycle management, and (**iii**) GPU-CPU state offloading for suspension and resumption.

Each GPU worker hosts a model replica and serves multiple active sessions through coalesced chunk execution: ready sessions on the same GPU are grouped into a chunk batch, and generated chunks and updated states are written back to their corresponding sessions. The runtime maintains sessions in *execution*, *suspend*, or *terminate* states. When a session becomes idle, its persistent state is offloaded from GPU to host memory and its GPU slot is released; when it becomes active again, the state is restored before generation resumes. This mechanism decouples session lifetime from GPU residency while preserving the context required for later resumption. More details about basic streaming serving runtime can be found in Appendix B.

2.3 CHARACTERIZATION AND MOTIVATION

We next characterize how session placement and GPU provisioning affect the latency and cost of streaming video generation serving.

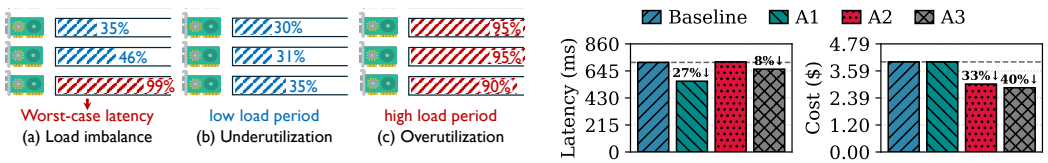


Figure 3: Characterization of session and GPU management in streaming video serving. **Left:** Evolving session activity causes load imbalance across GPUs, where the most heavily loaded GPU determines the worst-case per-chunk latency; a fixed GPU allocation further leads to underutilization during low-demand periods and overutilization during high-demand periods. **Right:** Case study comparing the baseline with session migration (A1), GPU autoscaling (A2), and their joint use (A3). *Latency* denotes the worst-case per-chunk latency, and *Cost* denotes the total GPU operating cost.

Characterization setup. We evaluate on 8 NVIDIA H100 GPUs and replay production traces from CompanyX with time-varying arrivals, departures, and active/idle session transitions. We report the worst-case per-chunk latency across active sessions and the total GPU operating cost over the trace, using the same model, trace, and latency target for all configurations. For this characterization, we use a static least-loaded placement policy on top of TurboServe$_{\text{base}}$ and a fixed GPU allocation, reaching a worst-case per-chunk latency of 0.71 s with a total GPU cost of \$3.99 [2]. Detailed workload statistics and policy configurations are provided in Appendix C.

Observation: latency imbalance and resource inefficiency. The baseline exposes two limitations. First, session activity evolves over time, so a placement that is initially balanced can become highly imbalanced as users transition between active and idle states. Since the worst-case per-chunk latency is determined by the most heavily loaded GPU, such imbalance directly increases system latency (Figure 3, left). Second, a fixed GPU allocation leads to resource underutilization during low-demand periods while still incurring high latency during bursty periods because the GPU budget cannot adapt to workload variation.

A1: Session migration. A1 isolates the benefit of session rebalancing by periodically migrating active sessions while keeping the same fixed GPU allocation as the baseline. As shown in Figure 3 (right), A1 reduces the worst-case per-chunk latency by **26.53%** at the same GPU cost, showing that migration reduces bottleneck latency by correcting stale session placements.

A2: GPU autoscaling. A2 isolates the benefit of elastic provisioning by dynamically adjusting the GPU budget according to workload demand. Compared with the baseline, A2 reduces total GPU cost by **32.57%** while maintaining the worst-case per-chunk latency within **0.71 s**, showing that autoscaling improves resource efficiency under time-varying demand.

A3: Joint migration and autoscaling. A3 combines session migration with GPU autoscaling, including rebalancing around scaling decisions. Compared with the baseline, A3 reduces the worst-case per-chunk latency by **8.17%** and total GPU cost by **40.25%**. Migration and autoscaling therefore address complementary bottlenecks: migration reduces load imbalance across GPUs, while autoscaling adapts the resource budget to workload demand. Their coordination is also important during resource changes: scale-out benefits from redistributing sessions onto newly provisioned GPUs, whereas scale-in requires consolidation before GPUs can be released.

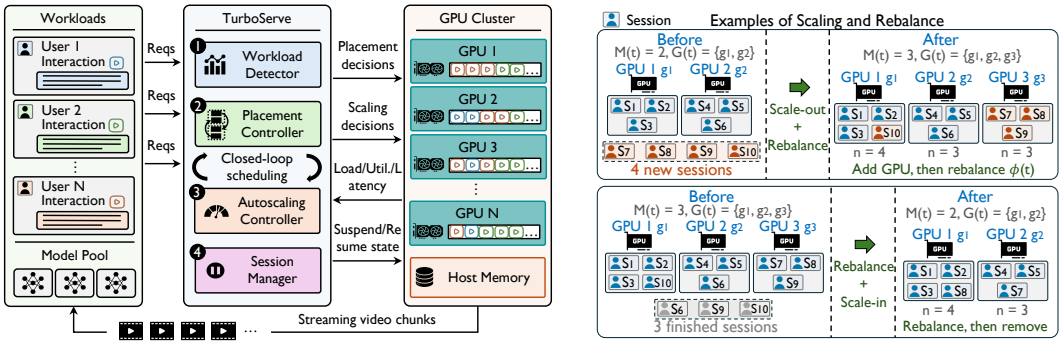
These observations motivate TurboServe to integrate migration-aware placement and GPU autoscaling into a unified closed-loop scheduling framework.

3 TurboServe Design

3.1 System Overview and Problem Formulation

As shown in Figure 4(a), TurboServe consists of four major components: a *workload detector*, *placement controller*, *autoscaling controller*, and *session manager*. The workload detector monitors session arrivals, departures, and active/idle transitions and extracts recent workload signals. The placement controller assigns and migrates sessions across GPU workers, while the autoscaling controller adjusts the provisioned GPU budget using runtime load feedback. The session manager

[2] We run experiments on an in-house CompanyX cluster, but report GPU operating cost in cloud-price-equivalent Amazon Web Services (2026) dollars for readability.



(a) System architecture **(b) Closed-loop scaling and rebalancing**

Figure 4: TURBOSERVE system overview and closed-loop scheduling. **(a)** TURBOSERVE processes streaming requests and session events through a workload detector, placement controller, autoscaling controller, and session manager. Placement and scaling decisions control GPU workers, while runtime load, utilization, and latency are fed back to the scheduler. **(b)** The interaction between autoscaling and session rebalancing: scale-out first expands the GPU set and then redistributes sessions, whereas scale-in first consolidates sessions and then releases the freed GPUs.

maintains session lifecycle states and coordinates execution, suspension, resumption, and migration. Together, these components form the closed loop described in §3.2.

Online scheduling model. We formulate streaming video serving as an event-driven online scheduling problem. At each scheduling event t, let $\mathcal{S}(t)$ denote the current session set and $\mathcal{G}(t) = \{g_1, \ldots, g_{M(t)}\}$ the set of provisioned GPUs, where $M(t)$ is the current GPU budget. Let $\alpha_i(t) \in \{0, 1\}$ indicate whether session s_i is currently active. The scheduler jointly determines (**i**) the GPU budget $M(t)$ and (**ii**) a placement $\phi_i(t)$ that assigns each session to a GPU $g_j \in \mathcal{G}(t)$ or to $\emptyset$ when its state is suspended. Each GPU can serve at most K concurrent sessions under the per-chunk latency constraint.

We define the GPU operating cost as $\mathcal{C}(t) = c_{\text{gpu}} M(t)$ where c_{gpu} is the per-GPU rental cost and the worst-case per-chunk latency as $\mathcal{L}(t) = \max_{s_i \in \mathcal{S}(t):\, \phi_i(t) \neq \emptyset} \ell_i(t)$ where $\ell_i(t)$ denotes the per-chunk latency of session s_i. At each event, the scheduling objective is

$$\begin{aligned} \min_{M(t),\, \phi(t)} \quad & \mathcal{C}(t) + \lambda(t)\mathcal{L}(t) \\ \text{s.t.} \quad & |\{i : \phi_i(t) = g_j\}| \leq K, \ \forall g_j \in \mathcal{G}(t), \qquad \alpha_i(t) = 1 \Rightarrow \phi_i(t) \neq \emptyset, \ \forall s_i \in \mathcal{S}(t). \end{aligned} \tag{1}$$

The first constraint bounds per-GPU concurrency, while the second ensures that active sessions remain executable. The weight $\lambda(t)$ controls the latency–cost trade-off and is adapted together with the target utilization by the autoscaling controller (§3.2.2). Additional notation and formulation details are provided in Appendix D.

3.2 CLOSED-LOOP SCHEDULING

We design a closed-loop scheduling algorithm that jointly coordinates session placement and cluster autoscaling. It consists of two tightly coupled components: (**i**) a *placement controller* that updates the session assignment $\phi(t)$ under the current GPU budget and provides load feedback, and (**ii**) an *autoscaling controller* that adjusts the GPU budget $M(t)$ based on this feedback. The resulting budget change in turn modifies the feasible placement space, closing the control loop between session rebalancing and resource provisioning.

3.2.1 MIGRATION-AWARE PLACEMENT CONTROLLER

The placement controller operates at each scheduling event t and determines the assignment $\phi(t)$ of sessions to the currently provisioned GPUs $\mathcal{G}(t)$. Given a fixed GPU budget $M(t)$, it seeks a feasible placement that minimizes the worst-case per-chunk latency:

$$\mathcal{L}^*(M, t) = \min_{\phi(t) \text{ feasible under } M(t)} \mathcal{L}(t). \tag{2}$$

Session assignment. The controller starts from the previous placement $\phi(t^-)$, removing terminated sessions while retaining existing assignments to avoid unnecessary migration. It then handles sessions whose placement must be updated, such as newly arrived sessions or suspended sessions that become active. For each such session, the controller evaluates feasible GPUs in $\mathcal{G}(t)$ and selects the assignment that minimizes the resulting bottleneck latency $\mathcal{L}(t)$. When multiple assignments result in similar latency, a fixed tie-breaking rule, such as preferring a less-loaded GPU, is used. The formal definition of the session-assignment set is provided in Appendix D.2.

Migration-aware min-max rebalancing. After session assignment, the controller performs migration-aware min-max rebalancing to reduce the bottleneck latency. Let $n_j(t) = |\{i : \phi_i(t) = g_j\}|$ denote the number of sessions assigned to GPU g_j, and let $g_{\max}(t)$ denote a GPU attaining the current maximum per-chunk latency. The controller considers migrating a session s_i on $g_{\max}(t)$ to a feasible target GPU $g_{j'} \in \mathcal{G}(t) \setminus \{g_{\max}(t)\}$ and evaluates the resulting bottleneck latency $\mathcal{L}'(t)$.

The migration cost $\kappa_i(t)$ captures the overhead of reassigning s_i and is modeled using the α-β communication model Hockney & Eastwood (2021). The benefit of a candidate migration is quantified by

$$\Gamma_{i,j'}(t) = \mathcal{L}(t) - \mathcal{L}'(t) - \eta \cdot \kappa_i(t), \tag{3}$$

where $\eta > 0$ controls the trade-off between latency reduction and migration overhead. At each iteration, the controller selects $(i^*, j'^*) = \arg\max_{i,\, j'} \Gamma_{i,j'}(t)$ (i.e. the candidate with the largest gain) to move and applies the migration if the gain is positive. Rebalancing terminates when no candidate migration yields $\Gamma_{i,j'}(t) > 0$. This event-driven local search therefore progressively reduces bottleneck latency while avoiding migrations whose expected benefit does not outweigh their transfer cost.

Scheduling complexity. Let $N(t) = |\mathcal{S}(t)|$ and $M(t) = |\mathcal{G}(t)|$. The session assignment step evaluates up to $M(t)$ GPUs per session in $\mathcal{U}(t)$, yielding $O(|\mathcal{U}(t)| \cdot M(t))$ time. In rebalancing, each iteration considers at most K sessions on the bottleneck GPU and evaluates their migration to up to $M(t)$ targets, resulting in $O(K \cdot M(t))$ time per iteration. The overall complexity is linear in $M(t)$ per event and negligible relative to the per-chunk generation latency.

3.2.2 Load-Driven Autoscaling Controller

The autoscaling controller determines the GPU budget $M(t)$ using load feedback from the placement controller. Let $\rho_{\max}(t) = \max_{g_j \in \mathcal{G}(t)} \frac{n_j(t)}{K}$ denote the maximum normalized GPU load after placement, where K is the maximum number of concurrent sessions that can be served by one GPU under the target per-chunk latency. Rather than explicitly solving the joint latency–cost objective in Equation 1 at every scheduling event, the controller regulates the system toward a target utilization $\hat{\rho}(t)$.

Hysteresis-based scaling trigger. Scaling decisions are driven by the load signal $\rho_{\max}(t)$ provided by the placement controller, relative to the target utilization $\hat{\rho}(t)$, which represents the desired fraction of the maximum capacity K (e.g., $\hat{\rho}(t) = 0.7$ corresponds to 70% utilization). To avoid oscillations under fluctuating workloads, the controller employs a *hysteresis mechanism* Qu et al. (2018) with a tolerance parameter $\delta > 0$ (e.g., $\delta = 0.1$), which defines the stability margin around the target utilization: scale-out is triggered when $\rho_{\max}(t) > \hat{\rho}(t) + \delta$, and scale-in is triggered when $\rho_{\max}(t) < \hat{\rho}(t) - \delta$.

Proportional scaling adjustment. The scaling magnitude follows a *proportional (target-tracking) policy* Kubernetes (2024), which adjusts the GPU budget in proportion to the deviation between the current load and the target utilization, enabling rapid convergence to the target utilization. Let $N_{\text{req}}(t) = |\{i : \phi_i(t) \neq \emptyset\}|$ denote the total number of sessions requiring GPU execution. The target GPU budget is then computed as $M_{\text{tar}}(t) = \left\lceil \frac{N_{\text{req}}(t)}{K\hat{\rho}(t)} \right\rceil$, and the scaling adjustment is $\Delta M(t) = M_{\text{tar}}(t) - M(t)$. After each scaling action, the updated load signal $\rho_{\max}(t)$ is obtained from the placement controller after recomputing $\phi(t)$ under the new budget, and is then used for subsequent decisions.

Adaptive control parameters. The autoscaling controller regulates the trade-off between GPU cost and worst-case per-chunk latency through the control parameters $(\lambda(t), \hat{\rho}(t))$, where $\lambda(t)$ weights latency in Equation 1 and $\hat{\rho}(t)$ specifies the target per-GPU utilization. To adapt to workload dynamics, the controller adjusts these parameters according to recent session arrivals and activation

patterns. Highly fluctuating workloads use a larger $\lambda(t)$ and a smaller $\hat{\rho}(t)$ to preserve additional latency headroom, whereas stable workloads use a smaller $\lambda(t)$ and a larger $\hat{\rho}(t)$ to improve resource efficiency. We realize this adaptation through a workload-volatility-based parameter mapping constructed offline and queried online; the full mapping procedure and comparison against an offline trace-tuned oracle are detailed in Appendix E.

Closing the loop. The interaction between the placement controller and autoscaling controller constitutes a closed control loop (as demonstrated in Algorithm 1 in Appendix D and illustrated in Figure 4(b)). In this loop, the placement controller provides the load feedback $\rho_{\max}(t)$, which informs autoscaling decisions, while the autoscaling controller updates the GPU budget $M(t)$, thereby regulating system load and worst-case per-chunk latency under dynamic workloads.

3.3 TurboServe Deployment

Building on the basic streaming serving runtime in §2.2, this section describes the deployment-level mechanisms that make TurboServe efficient under dynamic multi-session workloads. Since the scheduling framework continuously updates session placement and GPU provisioning, the runtime must efficiently manage session states and cluster resources during these updates. We therefore focus on *session mobility* and *elastic GPU management*; additional deployment details are provided in Appendix F.

Session mobility. Each GPU worker maintains a shared model replica together with isolated per-session states. Consequently, when the placement controller migrates an active session, TurboServe transfers only the corresponding session state rather than the model replica or completed outputs. GPU-to-GPU migration uses one-sided GPU memory access to directly transfer per-session state between workers without CPU staging. To preserve consistency, migration occurs only at chunk boundaries: the source first completes the current chunk, the session state is transferred to the target worker, and session ownership is updated only after the transfer completes. This design preserves session correctness while enabling fast rebalancing of active sessions to reduce bottleneck per-chunk latency.

Elastic GPU management. TurboServe is deployed on top of CompanyX's internal GPU cluster management system and maintains a logical serving pool of GPU workers, while the underlying cluster manager handles physical resource allocation and cleanup. During scale-out, TurboServe requests additional workers, launches the worker runtime, loads the model replica from locally pre-staged checkpoints, and marks the worker ready for session assignment. During scale-in, selected workers are first marked as draining; active sessions are migrated to remaining GPUs and idle sessions are suspended before the workers are released back to the cluster manager.

4 System Evaluation

4.1 Experimental Setup

Hardware, models, and workloads. We evaluate TurboServe on CompanyX's internal GPU serving clusters consisting of 16 NVIDIA H20 GPUs (Cluster 1) and 64 NVIDIA B300 GPUs (Cluster 2). Each server contains 8 GPUs connected through NVLink, while cross-node communication uses RDMA-enabled InfiniBand. We evaluate LongLive-style streaming video generation models at multiple scales, including 1.3B and 7B variants, and replay six production traces with heterogeneous session durations, bursty activation patterns, and time-varying numbers of active sessions. Detailed hardware configurations and workload statistics are provided in Appendix G.

Metrics and baselines. We use two primary metrics: (**i**) worst-case per-chunk latency across active sessions and (**ii**) total GPU operating cost during serving. For latency comparisons, methods are evaluated under matched GPU cost; for cost comparisons, they are evaluated under matched worst-case per-chunk latency.

We compare TurboServe against three representative serving strategies built on TurboServe$_{\text{base}}$: (**i**) TurboServe$_{\text{base}}$, which uses round-robin placement without migration or autoscaling; (**ii**) TurboServe$_{\text{base}}$+LAG, which assigns newly activated sessions to the currently least-loaded GPU; and (**iii**) TurboServe$_{\text{base}}$+MAG, which assigns sessions to the GPU with the lowest memory utilization. Detailed baseline configurations are provided in Appendix G.

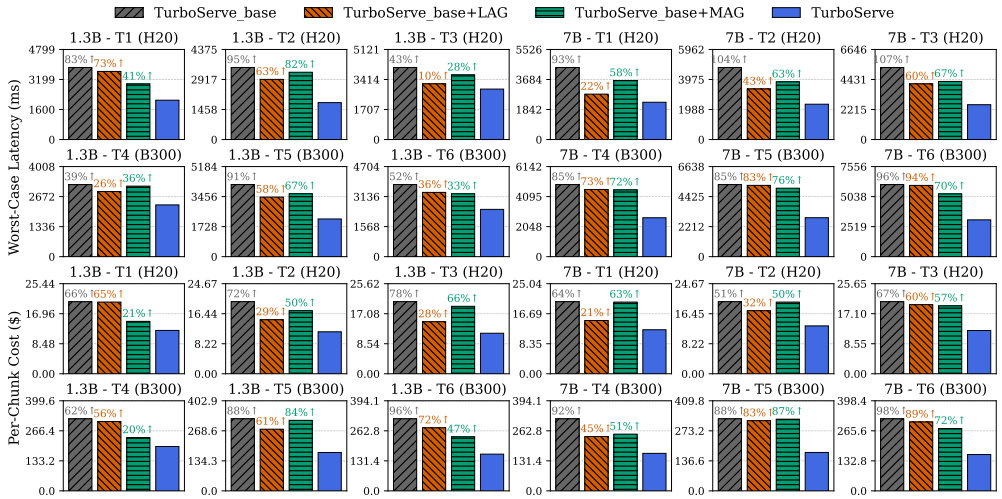


Figure 5: End-to-end experimental results comparing TURBOSERVE against baseline serving strategies across different models, traces, and clusters. The first two rows report worst-case per-chunk latency under matched GPU operating cost, while the last two rows report GPU operating cost under matched worst-case per-chunk latency.

4.2 END-TO-END PERFORMANCE

Latency under matched cost. As shown in Figure 5 (rows 1–2), TURBOSERVE consistently outperforms the baseline serving strategies across model sizes, production traces, and cluster configurations. Across all baseline comparisons, TURBOSERVE reduces worst-case per-chunk latency by 37.5% on average and by up to 51.6%. For example, on the 1.3B model with Trace 1 on Cluster 2, TURBOSERVE reduces latency by 28.2%, 20.5%, and 26.6% relative to TURBOSERVE$_{\text{base}}$, LAG, and MAG, respectively. The improvement comes from continuously correcting placement imbalance as session activity changes over time, preventing a small number of overloaded GPUs from determining the system-wide worst-case latency.

Cost under matched latency. As shown in Figure 5 (rows 3–4), TURBOSERVE also consistently achieves lower GPU operating cost under matched worst-case latency constraints, reducing cost by 37.2% on average and by up to 49.0%. On the same 1.3B/Trace 1/Cluster 2 configuration, TURBOSERVE reduces cost by 38.3%, 35.9%, and 16.7% relative to TURBOSERVE$_{\text{base}}$, LAG, and MAG, respectively. These improvements stem from TURBOSERVE's ability to dynamically adjust the GPU budget according to workload demand while simultaneously rebalancing sessions across available resources. By avoiding both resource underutilization during low-demand periods and excessive latency during bursts, TURBOSERVE achieves a more efficient latency-cost trade-off than static serving strategies.

Overall, the gains remain consistent across both model scales, all evaluated production traces, and H20/B300 clusters, showing that TURBOSERVE remains effective across model sizes, traces, and GPU clusters.

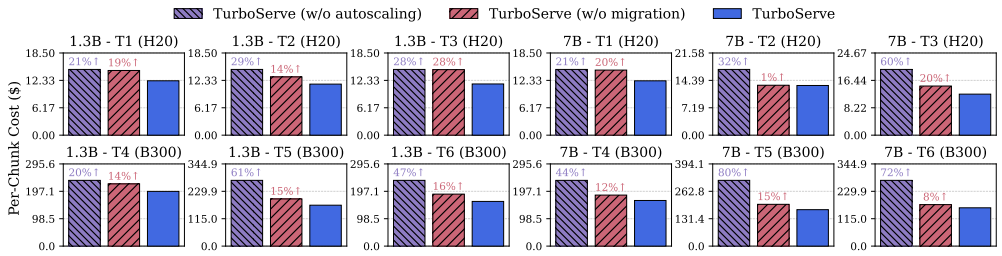


Figure 6: Cost-efficiency ablation of TURBOSERVE across model sizes, workload traces, and GPU types under matched latency constraints. We compare the full system against variants without session migration or without GPU autoscaling.

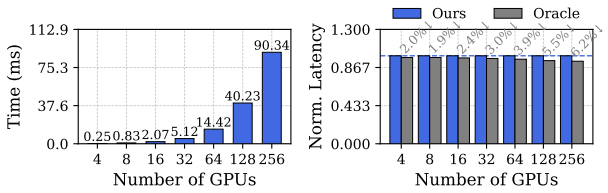


Figure 7: Efficiency and effectiveness of migration-aware min-max rebalancing. **Left:** scheduling efficiency. **Right:** effectiveness versus the oracle.

	T1	T2	T3
Oracle	$188.03	$158.55	$152.36
TURBOSERVE	$196.87 (+4.7%)	$171.71 (+8.3%)	$160.74 (+5.5%)

Table 1: Autoscaling cost on Cluster 2 compared with the offline oracle. Lower is better.

4.3 MECHANISM ABLATION AND SCHEDULING EFFECTIVENESS

We next evaluate the contribution of TURBOSERVE's two key mechanisms, session migration and GPU autoscaling, through ablation experiments across different model sizes, workload traces, and GPU types. As shown in Figure 6, removing either mechanism generally degrades cost efficiency compared with the full system. Disabling migration increases GPU operating cost by 15.0% on average and up to 28.0%, because the system can no longer effectively rebalance sessions across GPUs, resulting in persistent load imbalance under the same latency constraint. Disabling autoscaling increases GPU operating cost by 42.9% on average and up to 80.4%, by preventing the system from adapting the GPU budget to workload fluctuations and forcing resources to be provisioned more conservatively to maintain latency guarantees. Across all evaluated models and traces, the full TURBOSERVE configuration achieves the best cost efficiency.

Scheduling efficiency of the rebalancing algorithm. As shown in Figure 7 (Left), we evaluate the runtime of migration-aware min-max rebalancing from 4 to 256 GPUs. The results show that our algorithm is fast enough for online video generation tasks and scales efficiently to large clusters. On clusters with up to 64 GPUs, scheduling completes within 15 ms, less than 2% of the per-chunk generation time (on the order of hundreds of milliseconds). Even on larger clusters with 256 GPUs, scheduling completes within 0.1 s. These results show that the controller can react to online session events without becoming a bottleneck on the serving critical path.

Scheduling effectiveness of the rebalancing algorithm. To evaluate the effectiveness of our scheduling approach, we compare our migration-aware min-max rebalancing algorithm against an oracle that performs an exhaustive search for the best session placement across different cluster sizes at a much higher computational cost. We run each method on Trace 1 80 times and report the mean gap. As shown in Figure 7 (Right), TURBOSERVE reduces the gap to the oracle to 3.6% on average (6.5% maximum), while reducing the scheduling time by more than 10× on average.

Scheduling effectiveness of the autoscaling mechanism. To evaluate the effectiveness of our autoscaling mechanism, we compare its resource cost against an offline lower-bound oracle. Given the full trace in advance, including all future session arrivals, resumes, suspends, and departures, the oracle computes the minimum number of GPUs needed at each 1-minute slot to satisfy the target GPU utilization, then solves a dynamic program over GPU budgets to obtain the cost-optimal scaling schedule. We run both methods on three production traces on Cluster 2 and report the resulting cost in Table 1. As shown, TURBOSERVE stays within 6.1% of the oracle cost on average (8.3% maximum), despite operating online without any knowledge of future requests. Appendix G.2 further reports representative online scheduling traces and GPU-GPU session migration overhead at only 23–30 ms (2–3% of per-chunk latency).

5 CONCLUSION

This paper presents TURBOSERVE, the *first* serving system designed specifically for streaming video generation workloads in multi-session, multi-GPU environments. TURBOSERVE addresses heterogeneous session duration and time-varying user demand through a closed-loop scheduling framework that jointly coordinates migration-aware session placement and load-driven GPU autoscaling. Evaluations on real-world production traces from COMPANYX show that TURBOSERVE reduces worst-case per-chunk latency by 37.5% and total GPU operating cost by 37.2% on average compared with baseline serving configurations.

AI USE STATEMENT

In this work, we used generative AI tools to aid and polish writing, including language refinement, grammar checking, typo correction, and improving clarity and readability. We did not use generative AI tools to generate research ideas, design the methodology, or conduct experiments. All AI-assisted text was reviewed and revised by the authors. We take responsibility for the final content of this work, including text, claims, and artifacts produced with the aid of generative AI.

REPRODUCIBILITY STATEMENT

We provide an anonymized implementation of TURBOSERVE at `https://anonymous.4open.science/r/turboserve-anon-5FC0/`. Appendices C–G provide the workload characterization, scheduling formulation and algorithms, parameter profiling, runtime and deployment details, and experimental configurations needed to support reproducibility.

REFERENCES

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

Amazon Web Services. Amazon ec2 p5 instances. `https://aws.amazon.com/ec2/instance-types/p5/`, 2026.

Tim Brooks, Bill Peebles, Connor Holmes, Will DePue, Yufei Guo, Leo Jing, David Schnurr, Joe Taylor, Troy Luhman, Eric Luhman, et al. Video generation models as world simulators. *OpenAI Blog*, 1(8):1, 2024.

Bolin Chen. Flashserve: Cost-efficient serverless inference scheduling for large language models via tiered memory management and predictive autoscaling. In *Proceedings of the 2025 6th International Conference on Computer Science and Management Technology*, pp. 1291–1298, 2025.

Christina Delimitrou and Christos Kozyrakis. Quasar: Resource-efficient and qos-aware cluster management. *ACM Sigplan Notices*, 49(4):127–144, 2014.

Jiarui Fang, Jinzhe Pan, Xibo Sun, Aoyu Li, and Jiannan Wang. xdit: an inference engine for diffusion transformers (dits) with massive parallelism. *arXiv preprint arXiv:2411.01738*, 2024.

FastVideo Team. Fastvideo: A unified framework for accelerated video generation. `https://haoailab.com/blogs/fastvideo/`, 2025.

Tianrui Feng, Zhi Li, Shuo Yang, Haocheng Xi, Muyang Li, Xiuyu Li, Lvmin Zhang, Keting Yang, Kelly Peng, Song Han, et al. Streamdiffusionv2: A streaming system for dynamic and interactive video generation. *arXiv preprint arXiv:2511.07399*, 2025.

Yao Fu, Leyang Xue, Yeqi Huang, Andrei-Octavian Brabete, Dmitrii Ustiugov, Yuvraj Patel, and Luo Mai. {ServerlessLLM}:{Low-Latency} serverless inference for large language models. In *18th USENIX Symposium on Operating Systems Design and Implementation (OSDI 24)*, pp. 135–153, 2024.

Roberto Henschel, Levon Khachatryan, Hayk Poghosyan, Daniil Hayrapetyan, Vahram Tadevosyan, Zhangyang Wang, Shant Navasardyan, and Humphrey Shi. Streamingt2v: Consistent, dynamic, and extendable long video generation from text. In *Proceedings of the Computer Vision and Pattern Recognition Conference*, pp. 2568–2577, 2025.

Roger W Hockney and James W Eastwood. *Computer simulation using particles*. crc Press, 2021.

Xun Huang, Zhengqi Li, Guande He, Mingyuan Zhou, and Eli Shechtman. Self forcing: Bridging the train-test gap in autoregressive video diffusion. *Advances in Neural Information Processing Systems*, 38:167283–167308, 2026.

Team HunyuanWorld. Hy-world 1.5: A systematic framework for interactive world modeling with real-time latency and geometric consistency. *arXiv preprint*, 2025.

Youhe Jiang, Ran Yan, Xiaozhe Yao, Yang Zhou, Beidi Chen, and Binhang Yuan. Hexgen: Generative inference of large language model over heterogeneous environment. *arXiv preprint arXiv:2311.11514*, 2023.

Weijie Kong, Qi Tian, Zijian Zhang, Rox Min, Zuozhuo Dai, Jin Zhou, Jiangfeng Xiong, Xin Li, Bo Wu, Jianwei Zhang, et al. Hunyuanvideo: A systematic framework for large video generative models. *arXiv preprint arXiv:2412.03603*, 2024.

Kubernetes. Horizontal pod autoscaler. `https://kubernetes.io/docs/tasks/run-application/horizontal-pod-autoscale/`, 2024.

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

Rongzhi Li, Ruogu Du, Zefang Chu, Sida Zhao, Chunlei Han, Zuocheng Shi, Yiwen Shao, Huanle Han, Long Huang, Zherui Liu, et al. Taming the chaos: Coordinated autoscaling for heterogeneous and disaggregated llm inference. *arXiv preprint arXiv:2508.19559*, 2025.

Zhuohan Li, Lianmin Zheng, Yinmin Zhong, Vincent Liu, Ying Sheng, Xin Jin, Yanping Huang, Zhifeng Chen, Hao Zhang, Joseph E Gonzalez, et al. {AlpaServe}: Statistical multiplexing with model parallelism for deep learning serving. In *17th USENIX Symposium on Operating Systems Design and Implementation (OSDI 23)*, pp. 663–679, 2023.

Feng Liang, Akio Kodaira, Chenfeng Xu, Masayoshi Tomizuka, Kurt Keutzer, and Diana Marculescu. Looking backward: Streaming video-to-video translation with feature banks. In *International Conference on Learning Representations*, volume 2025, pp. 46425–46445, 2025.

Ziming Mao, Tian Xia, Zhanghao Wu, Wei-Lin Chiang, Tyler Griggs, Romil Bhardwaj, Zongheng Yang, Scott Shenker, and Ion Stoica. Skyserve: Serving ai models across regions and clouds with spot instances. In *Proceedings of the Twentieth European Conference on Computer Systems*, pp. 159–175, 2025.

Chenhao Qu, Rodrigo N Calheiros, and Rajkumar Buyya. Auto-scaling web applications in clouds: A taxonomy and survey. *ACM Computing Surveys (CSUR)*, 51(4):1–33, 2018.

Biao Sun, Ziming Huang, Hanyu Zhao, Wencong Xiao, Xinyi Zhang, Yong Li, and Wei Lin. Llumnix: Dynamic scheduling for large language model serving. In *18th USENIX symposium on operating systems design and implementation (OSDI 24)*, pp. 173–191, 2024.

Team Wan, Ang Wang, Baole Ai, Bin Wen, Chaojie Mao, Chen-Wei Xie, Di Chen, Feiwu Yu, Haiming Zhao, Jianxiao Yang, et al. Wan: Open and advanced large-scale video generative models. *arXiv preprint arXiv:2503.20314*, 2025.

Bingyang Wu, Yinmin Zhong, Zili Zhang, Shengyu Liu, Fangyue Liu, Yuanhang Sun, Gang Huang, Xuanzhe Liu, and Xin Jin. Fast distributed inference serving for large language models. *arXiv preprint arXiv:2305.05920*, 2023.

Bingyang Wu, Shengyu Liu, Yinmin Zhong, Peng Sun, Xuanzhe Liu, and Xin Jin. Loongserve: Efficiently serving long-context large language models with elastic sequence parallelism. In *Proceedings of the ACM SIGOPS 30th Symposium on Operating Systems Principles*, pp. 640–654, 2024.

Yuxing Xiang, Xue Li, Kun Qian, Yufan Yang, Diwen Zhu, Wenyuan Yu, Ennan Zhai, Xuanzhe Liu, Xin Jin, and Jingren Zhou. Aegaeon: Effective gpu pooling for concurrent llm serving on the market. In *Proceedings of the ACM SIGOPS 31st Symposium on Operating Systems Principles*, pp. 1030–1045, 2025.

Shuai Yang, Wei Huang, Ruihang Chu, Yicheng Xiao, Yuyang Zhao, Xianbang Wang, Muyang Li, Enze Xie, Yingcong Chen, Yao Lu, et al. Longlive: Real-time interactive long video generation. *arXiv preprint arXiv:2509.22622*, 2025.

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

Peiqi Yin, Jiangyun Zhu, Han Gao, Chenguang Zheng, Yongxiang Huang, Taichang Zhou, Ruirui Yang, Weizhi Liu, Weiqing Chen, Canlin Guo, et al. vllm-omni: Fully disaggregated serving for any-to-any multimodal models. *arXiv preprint arXiv:2602.02204*, 2026.

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

Li Zhang, Youhe Jiang, Guoliang He, Xin Chen, Han Lv, Qian Yao, Fangcheng Fu, and Kai Chen. Efficient mixed-precision large language model inference with turbomind. *arXiv preprint arXiv:2508.15601*, 2025.

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

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

A Additional Related Work

Streaming video generation. Recent streaming diffusion and video-generation models have begun to shift video generation from offline batch synthesis toward online and interactive streaming. Systems such as StreamDiffusionV2 Feng et al. (2025), StreamingT2V Henschel et al. (2025), and StreamV2V Liang et al. (2025) demonstrate that diffusion-based video generation can be organized around continuous or autoregressive generation, enabling long videos, dynamic inputs, and interactive use cases. Concurrent efforts such as LongLive Yang et al. (2025), Self-Forcing Huang et al. (2026), and HYWorldPlay HunyuanWorld (2025) further demonstrate this shift toward long-horizon, interactive, and real-time streaming video/world generation. These model-level advances create a new class of workloads in which video chunks must be generated continuously and delivered with low latency. However, most existing work focuses on model architecture, inference pipeline design, or frame-level generation quality, rather than multi-user serving efficiency at the cluster level. This leaves a gap between the capabilities of streaming video-generation models and the infrastructure needed to serve them efficiently at scale.

LLM serving. Recent LLM serving systems have significantly improved the efficiency of online inference Zheng et al. (2024); Jiang et al. (2023); Zhang et al. (2025); Yu et al. (2022); Zhong et al. (2024); Wu et al. (2024); Mao et al. (2025); Wu et al. (2023); Ye et al. (2025). Representative systems such as vLLM Kwon et al. (2023), Sarathi-Serve Agrawal et al. (2024), and AlpaServe Li et al. (2023) consider each request as a finite sequence of decoding iterations with dynamically growing KV-cache state. In contrast, streaming video generation must preserve session state across multiple active and idle periods and repeatedly schedule chunk generation under tight per-chunk latency targets. Existing LLM serving systems therefore do not explicitly optimize the session-based streaming video generation workload considered in this work.

Autoscaling for LLM serving. Autoscaling and elastic resource management have become increasingly important for cost-efficient LLM serving Sun et al. (2024); Xiang et al. (2025); Chen (2025); Li et al. (2025); Fu et al. (2024). Existing systems dynamically adjust GPU allocation, model placement, execution capacity, or checkpoint loading according to workload variation and latency objectives, substantially improving resource efficiency under fluctuating demand. However, these systems primarily target request-centric inference workloads. Streaming video generation instead introduces long-lived, stateful sessions that continuously alternate between active and idle periods while maintaining persistent generation states. Consequently, scaling decisions affect not only the amount of available GPU resources, but also how session states are suspended, resumed, migrated, and redistributed during resource changes.

B Basic Streaming Serving Runtime Details

Concurrent chunk execution. Streaming video generation executes at the granularity of video chunks, where each GPU worker hosts a model replica and serves multiple active sessions assigned to that worker. At each chunk step, the runtime follows a coalesced execution procedure: (**i**) it collects sessions whose next chunks are ready to be generated; (**ii**) it groups ready sessions on the same GPU into a coalesced chunk batch; and (**iii**) it invokes the model once for the batch and writes the generated chunks and updated states back to their corresponding sessions. Coalescing improves GPU utilization by amortizing model execution across concurrent sessions. However, streaming serving requires each active session to deliver chunks within the target per-chunk latency, so the number of co-located sessions on each GPU must be bounded.

Session state and lifecycle. Unlike stateless one-shot generation jobs, streaming video generation maintains persistent session state across multiple active and idle periods. Accordingly, the runtime distinguishes three session states: (**i**) *execution*, where the session is assigned to a GPU and generates chunks; (**ii**) *suspend*, where the session is idle and its GPU slot is released while its state is preserved; and (**iii**) *terminate*, where the session has completed and all associated resources are released. These states are updated through lifecycle operations such as initialization, suspension, resumption, and termination, triggered by session arrivals, departures, and active/idle user transitions.

GPU-CPU state offloading. To avoid occupying GPU memory and execution slots for idle sessions, TurboServe$_{\text{base}}$ supports GPU-CPU state offloading for suspension and resumption. When a session becomes idle or must release its GPU slot, the runtime copies its persistent session state

from GPU memory to host memory and releases the GPU slot. When the session becomes active again, the state is restored to the selected GPU before chunk generation resumes. This mechanism decouples session lifetime from GPU residency while preserving the context needed for later resumption.

C Characterization Details

C.1 Workload Statistics

Table 2: Workload trace characteristics. *Arrivals* is the number of new jobs entering the system in each span; *Departures* is the number of jobs completing in each span; *Avg. active* is the mean number of concurrently active jobs over the span.

Time (min)	$[0, 2)$	$[2, 4)$	$[4, 6)$	$[6, 8)$	$[8, 10]$
Arrivals	31	47	30	48	44
Departures	17	24	26	39	49
Avg. active	10.36	20.91	19.30	29.62	33.49

Table 2 demonstrates the workload trace characteristics for characterization and motivation.

Table 3 reports the detailed characteristics of the evaluated traces. Traces T1–T3 are evaluated on the H20 cluster (Cluster 1) and traces T4–T6 on the B300 cluster (Cluster 2)

Table 3: Workload trace characteristics. *Arrivals* is the number of new jobs entering the system in each span; *Departures* is the number of jobs completing in each span; *Avg. active* is the mean number of concurrently active jobs over the span.

Trace	Metric	$[0, 1)$	$[1, 2)$	$[2, 3)$	$[3, 4)$	$[4, 5)$	Total
T1	Arrivals	122	130	66	22	18	358
	Departures	44	86	88	64	38	320
	Avg. active	28.0	56.2	57.4	37.4	23.2	–
T2	Arrivals	218	214	248	192	204	1076
	Departures	60	124	204	196	204	788
	Avg. active	61.0	118.6	147.6	154.2	149.4	–
T3	Arrivals	74	148	156	264	156	798
	Departures	10	64	126	148	214	562
	Avg. active	13.2	49.4	112.8	121.4	148.4	–
T4	Arrivals	500	428	308	88	80	1404
	Departures	158	266	388	236	152	1200
	Avg. active	118.4	219.2	268.0	162.8	101.8	–
T5	Arrivals	874	862	998	762	814	4310
	Departures	238	504	810	784	814	3150
	Avg. active	245.8	475.2	589.2	616.8	598.4	–
T6	Arrivals	296	590	626	1062	618	3192
	Departures	36	260	502	608	846	2252
	Avg. active	54.4	198.4	451.4	487.2	592.0	–

C.2 Characterization Policies

Baseline configuration. Each newly activated session is assigned to the GPU with the least current load among the available GPUs. Once assigned, the session remains on the same GPU until it is suspended or terminated. The system uses a fixed GPU allocation for the entire trace.

A1: Session migration. To isolate the benefit of session rebalancing, A1 uses the same fixed GPU allocation as the baseline but periodically rebalances active sessions across GPUs every 10 seconds. At each rebalancing point, the system observes the current session distribution and migrates sessions to minimize the maximum per-GPU load. Since the GPU budget remains unchanged, any latency improvement comes from better session placement rather than additional resources.

A2: GPU autoscaling. To isolate the benefit of elastic GPU provisioning, A2 dynamically adjusts the GPU budget according to current workload demand instead of maintaining a fixed allocation throughout the trace. During high-demand periods, additional GPUs are provisioned to preserve latency headroom; during low-demand periods, unnecessary GPUs are released to reduce operating cost.

A3: Joint migration and autoscaling. A3 combines session migration with GPU autoscaling. Similar to A1, it periodically rebalances active sessions every 10 seconds while additionally rebalancing around each scaling decision according to the updated GPU set. Scale-out first adds GPUs and then redistributes active sessions to utilize the new capacity, whereas scale-in first consolidates sessions before releasing unnecessary GPUs.

D Additional Scheduling Details

D.1 Notation and Problem Formulation Details

Table 4: Frequently used notation in the scheduling framework.

Notation	Description
$\mathcal{S}(t)$	Session set at event t
$\mathcal{G}(t)$	Currently provisioned GPUs at event t
$M(t)$	Number of provisioned GPUs at event t
$\mathcal{C}(t)$	GPU operating cost at event t
K	Maximum concurrent sessions per GPU
$\phi_i(t)$	Placement of session s_i at event t
$\alpha_i(t)$	User-activity indicator for session s_i at event t
$\mathcal{L}(t)$	Worst-case per-chunk serving latency
$\mathcal{L}^*(M,t)$	Minimum worst-case latency under GPU budget $M(t)$
$\rho_{\max}(t)$	Maximum normalized GPU load after placement
$\hat{\rho}(t)$	Target per-GPU utilization
$\lambda(t)$	Latency weight in the optimization objective
$M_{\text{tar}}(t)$	Target GPU budget computed by autoscaling

Event-driven scheduling. The scheduler is invoked upon system events including session arrivals, departures, and active/idle state transitions. Each invocation corresponds to a decision epoch indexed by $t = 1, 2, \ldots$, at which the scheduler observes the current session and resource state and updates the GPU budget and session placement. Between consecutive events, the current scheduling decision remains in effect.

Session activity and placement. Let $\alpha_i(t) \in \{0, 1\}$ indicate whether the user is actively interacting with session s_i at event t. A placement $\phi_i(t) \in \mathcal{G}(t) \cup \{\emptyset\}$ assigns the session either to a currently provisioned GPU or to the suspended state. Following the runtime described in §2.2, suspended sessions preserve their state in host memory and can later be restored when they become active.

Cost and latency. The operating cost is $\mathcal{C}(t) = c_{\text{gpu}} M(t)$, where c_{gpu} is the per-GPU operating cost. $M(t)$ counts all provisioned GPUs, including GPUs that are being initialized during scale-out but are not yet ready to serve sessions. The serving latency is defined as

$$\mathcal{L}(t) = \max_{s_i \in \mathcal{S}(t):\, \phi_i(t) \neq \emptyset} \ell_i(t),$$

where $\ell_i(t)$ is the per-chunk latency of session s_i. The latency may increase with session co-location and can temporarily include migration-related overhead.

Latency–cost trade-off. The weight $\lambda(t)$ in Equation 1 balances GPU operating cost against worst-case per-chunk latency. A larger $\lambda(t)$ favors additional capacity and more aggressive rebalancing to reduce latency, whereas a smaller $\lambda(t)$ favors higher GPU utilization and lower provisioning cost. The autoscaling controller therefore adapts the operating point according to recent workload characteristics, as described in §3.2.2 and Appendix E.

Algorithm 1 Closed-Loop Scheduling Workflow

Require: Event stream $\{t\}$; session state $\mathcal{S}(t)$; placement $\phi(t)$; previous placement $\phi(t^-)$; GPU budget $M(t)$; previous GPU budget $M(t^-)$; load signal $\rho_{\max}(t)$; target GPU budget $M_{\text{tar}}(t)$; placement controller $\textsc{Place}(\cdot)$; autoscaling controller $\textsc{Scale}(\cdot)$
Ensure: Updated placement $\phi(t)$; updated budget $M(t)$

for each event t **do**
  /* Placement and GPU budget update */
  $(\phi(t), \rho_{\max}(t)) \leftarrow \textsc{Place}(\mathcal{S}(t), \phi(t^-), M(t^-))$
  $M_{\text{tar}}(t) \leftarrow \textsc{Scale}(\rho_{\max}(t), M(t^-))$
  if $M_{\text{tar}}(t) < M(t^-)$ **then**
    /* Scale-in: rebalancing precedes removal */
    $(\phi(t), \rho_{\max}(t)) \leftarrow \textsc{Place}(\mathcal{S}(t), \phi(t), M_{\text{tar}}(t))$
    $M(t) \leftarrow M_{\text{tar}}(t)$
  else if $M_{\text{tar}}(t) > M(t^-)$ **then**
    /* Scale-out: expansion precedes rebalancing */
    $M(t) \leftarrow M_{\text{tar}}(t)$
    $(\phi(t), \rho_{\max}(t)) \leftarrow \textsc{Place}(\mathcal{S}(t), \phi(t), M(t))$
  else
    /* No scaling: keep current budget */
    $M(t) \leftarrow M(t^-)$
  end if
end for

D.2 Placement Initialization and Session Assignment

At each scheduling event t, the placement controller initializes from the previous placement $\phi(t^-)$ by removing terminated sessions and retaining existing assignments. This incremental initialization avoids unnecessary migration and allows the controller to update only sessions affected by the current event.

The set of sessions requiring assignment is defined as

$$\mathcal{U}(t) = \left\{ s_i \in \mathcal{S}(t) : \alpha_i(t) = 1 \wedge \phi_i^-(t) = \emptyset \right\}, \tag{4}$$

which includes sessions that require GPU execution but are not currently assigned to a GPU. For each $s_i \in \mathcal{U}(t)$, the controller evaluates every feasible GPU $g_j \in \mathcal{G}(t)$ and selects the assignment that minimizes the resulting bottleneck latency $\mathcal{L}(t)$. When multiple assignments yield similar $\mathcal{L}(t)$, the controller uses a fixed tie-breaking rule, such as preferring the less-loaded GPU.

After these mandatory assignments are processed, the migration-aware min-max rebalancing procedure described in §3.2.1 considers additional migrations from the current bottleneck GPU and repeatedly applies positive-gain moves until no beneficial migration remains.

D.3 Closed-Loop Controller Workflow

Algorithm 1 summarizes the complete interaction between the placement and autoscaling controllers. Placement is first evaluated under the current GPU budget to obtain both a session assignment and the corresponding load signal. The autoscaling controller then computes a target GPU budget. Scale-in and scale-out use different execution orderings to ensure that all active sessions remain feasible throughout the resource transition.

During *scale-out*, the new GPU budget is applied before rebalancing so that the placement controller can immediately use the newly provisioned resources. During *scale-in*, the placement controller first computes a feasible placement under the smaller target budget; only after sessions have been consolidated are the excess GPUs removed. This ordering prevents resource removal from invalidating active-session placements.

E Volatility-to-Parameter Mapping

The autoscaling controller adapts the control parameters $(\lambda(t), \hat{\rho}(t))$ in response to workload dynamics (§3.2.2). Following the workload-classification paradigm of Quasar Delimitrou & Kozyrakis

(2014), we classify workloads into a small set of levels based on a sliding-window volatility metric over session activations, profile offline the cost-optimal control parameters for each level, and at runtime apply the parameters associated with the level of the current workload. We further compare this online mapping against an offline trace-tuned oracle to validate its near-optimality.

Concretely, we measure workload variability through a single scalar metric, the *volatility* $\sigma(t)$ defined below, and realize the adaptation through a volatility-to-parameter mapping $\mathcal{T} : \mathcal{V} \to \mathbb{R}_{>0} \times (0, 1]$ that associates each volatility level $v_\ell \in \mathcal{V} = \{v_1, \ldots, v_L\}$ (e.g., $L = 10$) with profiled control parameters $(\lambda_\ell, \rho^*_\ell)$. The mapping is constructed offline and queried online to update the active parameters.

Volatility metric. Let a_τ denote the number of newly activated sessions at event τ. Given a sliding event window of size W ending at event t, the workload volatility is defined as:

$$\sigma(t) = \mathrm{std}\big(a_{t-W+1}, \ldots, a_t\big). \tag{5}$$

The volatility range observed across historical traces is partitioned into L ordered intervals, with v_ℓ representing the ℓ-th interval from low to high.

Offline profiling. We subsample representative segments from historical real-world workload traces and group them by volatility level. For each level v_ℓ, we perform a grid search over candidate control parameters (λ, ρ^*), replay the corresponding trace segments through the scheduler, and select the parameters $(\lambda_\ell, \rho^*_\ell)$ that minimizes GPU cost subject to the target worst-case per-chunk latency constraint. The resulting table $\{\mathcal{T}(v_\ell) = (\lambda_\ell, \rho^*_\ell)\}_{\ell=1}^{L}$ is persisted and loaded by the controller at startup.

Online replacement workflow. At each event t, the controller updates the active control parameters via a four-step *measure-quantize-look-up-replace* workflow:

1. *Measure.* Compute the current volatility $\sigma(t)$ over the sliding window of recent events.
2. *Quantize.* Map $\sigma(t)$ to the level $v_{\ell(t)} \in \mathcal{V}$ whose interval contains $\sigma(t)$.
3. *Look up.* Retrieve the profiled parameters $(\lambda_{\ell(t)}, \rho^*_{\ell(t)}) = \mathcal{T}(v_{\ell(t)})$.
4. *Replace.* Commit the update $(\lambda(t), \hat{\rho}(t)) \leftarrow (\lambda_{\ell(t)}, \rho^*_{\ell(t)})$, which takes effect in subsequent autoscaling decisions.

Profiling case study. We illustrate the offline profiling and the resulting volatility-to-parameter mapping on a representative trace family consisting of $L = 10$ segments of monotonically increasing volatility, generated by progressively scaling the burst magnitude of a base activation pattern. Table 5 summarizes the workload characteristics of each segment, and Table 6 reports the profiled control parameters together with the cost and latency observed during replay under the latency constraint $L_{\mathrm{SLO}} = 670\,\mathrm{ms}$. Three observations stand out from the profiled mapping:

Table 5: Workload characteristics of the ten profiling segments. *Volatility* is the standard deviation of newly activated session counts across 5 s bins; *arrivals* is the total number of new sessions over the segment; *peak active* is the maximum number of concurrent active sessions.

Level	Volatility	Arrivals	Peak active
1	0.86	7	23
2	1.32	11	27
3	1.92	15	31
4	2.66	19	35
5	3.15	23	39
6	3.77	27	43
7	4.39	31	47
8	5.14	35	51
9	5.51	39	55
10	6.38	43	59

O1: Monotonic reduction of ρ^* with volatility. The target utilization ρ^*_ℓ decreases monotonically as the volatility level increases, dropping from 0.80 at level 1 to 0.25 at level 10. This directly

Table 6: Profiled control parameters and replay results per volatility level under the latency constraint $L_{\text{SLO}} = 670\,\text{ms}$. *Valid* indicates whether the configuration satisfies the latency constraint; *pass rate* is the fraction of generated chunks whose per-chunk latency falls within L_{SLO}; *avg. cost* is the average GPU operating cost observed during replay.

Level	Volatility	λ	ρ^*	Valid	Pass rate	Avg. cost
1	0.86	0.2	0.80	✓	100%	338.66
2	1.32	0.2	0.80	✓	100%	369.35
3	1.92	0.2	0.65	✓	100%	498.73
4	2.66	0.2	0.65	✓	100%	526.14
5	3.15	0.2	0.65	✓	100%	547.34
6	3.77	0.2	0.50	✓	100%	700.77
7	4.39	0.2	0.50	✓	100%	702.44
8	5.14	0.2	0.50	✓	100%	705.05
9	5.51	0.2	0.25	✓	100%	770.44
10	6.38	0.2	0.25	✓	100%	769.46

realizes the design intuition stated in §3.2.2: at low volatility the controller can pack sessions densely without risking latency violations, while at high volatility it must reserve substantial per-GPU headroom to absorb activation bursts. Notably, the profiled ρ^*_ℓ values fall into four discrete bands $(0.80, 0.65, 0.50, 0.25)$, each spanning two to three adjacent volatility levels, suggesting that the cost-latency frontier exhibits a piecewise-flat structure rather than continuously varying with volatility.

O2: Stable λ across the entire range. The latency weight λ_ℓ remains fixed at 0.2 across all ten levels, indicating that for this trace family the cost-latency trade-off is governed primarily by ρ^*, with λ playing a secondary role. Conceptually, ρ^* adapts the system's safety margin in capacity space, while λ would primarily activate as a second axis under workloads where headroom adjustment via ρ^* alone proves insufficient (e.g., when ρ^* saturates at its lower bound and further headroom can only be obtained by penalizing latency more aggressively in the objective).

O3: Monotonic increase of cost with volatility. The average GPU cost increases monotonically as the volatility level rises, from 338.66 at level 1 to 770.44 at level 9, roughly $2.3\times$. This growth mirrors the reduction in ρ^* documented in O1: as ρ^*_ℓ shrinks, each GPU carries fewer concurrent sessions, forcing the controller to provision additional GPUs to serve the same workload. The mapping therefore makes explicit the cost of accommodating burstiness: each volatility level pays exactly the additional GPU cost required to handle its bursts. A non-adaptive controller using a fixed ρ^* would either incur level-9 cost at level-1 workloads (over-provisioning under stable load) or violate the latency constraint at high volatility (under-provisioning under bursts), whereas the volatility-keyed mapping recovers the cost-minimal feasible configuration at each level.

Results analysis. All ten levels achieve a 100% pass rate under $L_{\text{SLO}} = 670\,\text{ms}$, confirming that the constrained grid search consistently identifies feasible (λ, ρ^*) configurations across the entire volatility range. The mapping is therefore simultaneously *safe* (no per-chunk latency violations are observed), and *cost-aware* (each level pays only the cost required to absorb its own characteristic burstiness).

Comparison with an offline oracle. We further evaluate how close the online mapping is to an offline trace-tuned oracle. The goal is to isolate the gap caused by online, adaptive control-parameter selection, rather than differences in scheduling primitives, so both methods use the same migration and autoscaling mechanisms, the same maximum GPU budget, and the same scale-out delay and cost model.

Oracle. The oracle has access to the realized workload trace in each future control window. For every $30\,\text{s}$ window, it searches over $\rho^* \in \{0.50, 0.55, \ldots, 0.95\}$ and selects the cost-minimizing value that still satisfies the latency constraint, knowing the future arrivals exactly. This baseline is not deployable in practice, but provides an upper bound for evaluating the quality of the online mapping.

Table 7: Unseen fluctuating workload used for the oracle comparison. *Avg./max active sessions* characterize the per-window load.

Window	**w1**	**w2**	**w3**	**w4**	**w5**	**w6**	**w7**	**w8**	**w9**	**w10**
Avg. active sessions	32.00	17.17	7.67	23.47	51.23	72.43	12.43	56.90	22.30	53.17
Max active sessions	38	18	8	28	58	77	13	68	23	73

Ours. The online mapping does not observe the future trace. During serving, it computes the volatility from a recent sliding window, maps it to a profiled level, and applies the corresponding control parameters. In this experiment, λ is fixed at 0.20 and adaptation is performed through ρ^* alone.

Workload trace. We construct an unseen 5-minute workload comprising ten 30 s windows that alternate between low, medium, and high pressure. Each GPU hosts at most five concurrent sessions, and the maximum GPU budget is 16. Table 7 summarizes the per-window load characteristics.

Selected control parameters. Table 8 reports the target utilization selected by the oracle and by our online mapping. With full knowledge of future demand, the oracle picks the per-window cost-minimizing ρ^*, packing densely in low-pressure windows (ρ^* up to 0.95 in w3) and lowering ρ^* to 0.55 during high-pressure windows. Our method, observing only recent history, follows a qualitatively similar pattern in the early windows but settles into a steady $\rho^* = 0.60$ once volatility has risen, retaining this conservative setting until the volatility estimate decays. Overall, the oracle tends toward sharp, per-window-optimal swings (range 0.55 to 0.95), while our method tends toward smoother, more conservative selections (range 0.60 to 0.88), trading per-window optimality for stability against unforeseen bursts.

Table 8: Per-window target utilization ρ^*.

Window	**w1**	**w2**	**w3**	**w4**	**w5**	**w6**	**w7**	**w8**	**w9**	**w10**
Offline oracle ρ^*	0.60	0.88	0.95	0.72	0.55	0.55	0.92	0.55	0.80	0.55
Ours ρ^*	0.66	0.72	0.88	0.78	0.60	0.60	0.60	0.60	0.60	0.60

Table 9: End-to-end comparison between the online mapping and the offline oracle. *Cost* is the total GPU operating cost; *avg./max lat.* are the per-chunk latency statistics across the entire trace.

Method	**Cost**	**Avg. lat.**	**Max lat.**
Offline oracle	3695.00	410.47 ms	668.05 ms
Ours	3721.93	421.57 ms	667.46 ms

Experimental results. Table 9 reports the headline simulation outcomes, and Table 10 shows the window-end GPU budgets. Both methods keep the served-chunk maximum latency at or below 668.05 ms, within the 670 ms target, with comparable average latencies (410.47 ms for the oracle and 421.57 ms for our method). The oracle uses fewer GPUs in low-pressure windows where it can safely raise ρ^* (e.g., w2, w3, w7, w9), while our method holds additional headroom in those windows because the recent volatility estimate has not yet decayed. On total operating cost, the online mapping (3721.93) comes within 0.73% of the offline oracle (3695.00).

Results analysis. This near-equality is striking given the substantial information asymmetry between the two methods: the oracle has access to every future arrival in every window and selects ρ^* per-window optimally (ranging from 0.55 to 0.95), whereas our method observes only a recent sliding window of activations and chooses ρ^* from the volatility mapping (ranging from 0.60 to 0.88). The result demonstrates that the volatility-based mapping is robust across diverse workload regimes: it handles low-pressure intervals, sustained bursts, and abrupt transitions alike without exceeding the latency target, while incurring almost the same total cost as the oracle. The mapping therefore captures essentially all of the cost-relevant structure that future knowledge could exploit, and remains effective under the inherent delay between observing rising volatility and provisioning additional GPUs. Since the oracle requires future knowledge and is not deployable in practice, it represents an unrealizable upper bound, and the online mapping effectively closes the gap to it under realistic causality constraints. We also conduct the same comparison on two additional traces subsampled

Table 10: Window-end GPU budgets selected by each method.

Window	w1	w2	w3	w4	w5	w6	w7	w8	w9	w10
Offline oracle	13	5	2	8	16	16	3	16	6	16
Ours	11	8	8	8	16	16	8	16	8	15

from real-world workloads, where the cost gap to the oracle is 2.25% and 3.99%, indicating that our volatility-based mapping generalizes beyond this single trace.

F Additional Runtime and Deployment Details

F.1 Session State Management

Session memory layout. Each GPU worker organizes runtime memory into three parts: (**i**) a shared model replica used by all sessions on the worker; (**ii**) isolated per-session state regions containing the session descriptor, prompt/control embeddings, temporal or KV-cache states, chunk-history features, intermediate latent buffers, and output metadata; and (**iii**) a session ownership table mapping each active session to its current worker and state buffers. This separation between static model resources and dynamic session state allows TurboServe to migrate or offload an individual session without moving the model replica or completed output chunks.

GPU-GPU state migration. When the placement controller moves an active session between GPUs, TurboServe migrates only its per-session state region. Each worker registers the memory used for session states with the cluster communication runtime, allowing the target worker to directly transfer the required state from the source worker without unnecessary CPU staging. After the transfer, the target installs the state into its local session table and resumes subsequent chunk generation.

Migration consistency. Migration is performed only at chunk boundaries to avoid interrupting an in-flight generation step. The source worker first completes the current chunk update and freezes the session state. The target worker then receives the required buffers and confirms their installation. Session ownership is updated only after the transfer completes, ensuring that subsequent chunks execute on the target worker and preventing duplicated execution.

F.2 Elastic GPU Management

CompanyX-managed GPU pool. TurboServe is deployed on top of CompanyX's internal GPU cluster management system. The cluster manager maintains a pool of available GPU workers whose container images, runtime dependencies, and model checkpoints are pre-staged on local storage. TurboServe maintains a logical serving pool consisting of the GPU workers currently admitted for streaming video generation, while the underlying cluster manager remains responsible for physical resource allocation, isolation, and cleanup.

Each active worker hosts a model replica and periodically reports its load, health status, and availability to the autoscaling controller. This separation allows TurboServe to perform serving-level autoscaling while reusing the existing cluster infrastructure for resource management.

Scale-out procedure. When the autoscaling controller triggers scale-out, TurboServe requests additional workers from the cluster manager. The cluster manager reserves available GPUs and attaches them to the logical serving pool, after which TurboServe launches the worker runtime, loads the model replica from the locally staged checkpoint, and marks the worker ready for session assignment. Once ready, the placement controller can assign new sessions or migrate existing sessions onto the added GPU. Pre-staged runtime environments and checkpoints avoid remote image pulling or checkpoint downloading on the critical scale-out path.

Scale-in procedure. When scale-in is triggered, TurboServe first marks selected workers as draining so that they no longer accept new sessions. The placement controller migrates their active sessions to the remaining GPU set, while idle sessions can be suspended using the GPU–CPU state-offloading mechanism described in §2.2. After all resident sessions have been migrated or suspended, TurboServe unloads the model replica, frees GPU memory, removes the worker from the serving pool, and returns the GPU to the cluster manager.

G Additional Evaluation Details and Results

G.1 Detailed Experimental Setup

Hardware environments. We evaluate TurboServe on CompanyX's internal GPU serving clusters consisting of 16 NVIDIA H20 GPUs (Cluster 1) and 64 NVIDIA B300 GPUs (Cluster 2). Each server contains 8 GPUs connected through high-bandwidth NVLink interconnects (956 GB/s), while cross-node communication is supported through RDMA-enabled InfiniBand networking (50 GB/s).

Models and workloads. We evaluate LongLive-style streaming video generation models at multiple scales. The workload is replayed from CompanyX's production traces containing heterogeneous session durations, bursty online activation patterns, and time-varying numbers of active sessions. Detailed statistics of Trace 1–6 are provided in Appendix C.1.

Baseline implementations. TurboServe$_{\text{base}}$ assigns newly activated sessions to GPUs in round-robin order and executes sessions in FCFS order without migration or autoscaling. TurboServe$_{\text{base}}$+LAG assigns newly activated sessions to the currently least-loaded GPU according to runtime GPU utilization. TurboServe$_{\text{base}}$+MAG assigns sessions to the GPU with the lowest memory utilization, aiming to balance memory pressure under long-lived sessions.

G.2 Runtime Breakdown and Overhead Analysis

Table 11: Twelve-minute snapshot of scheduling decisions for TurboServe in the Trace 1 experiment on Cluster 1. The autoscaling-decision column reports the actual GPU-budget changes in each two-minute window. The migration-decision column reports representative session movements, where s denotes a session ID and g denotes a GPU ID.

Time Window	Autoscaling Decisions	Migration Decisions
$(0, 2]$ min	$8 \to 1 \to 2 \to 3 \to 4 \to 5 \to 6$	$s1{:}g1 \to g0$, $s2{:}g1 \to g0$, ... $(+32)$
$(2, 4]$ min	$6 \to 7 \to 8 \to 9 \to 10$	$s49{:}g1 \to g2$, $s44{:}g1 \to g0$, ... $(+21)$
$(4, 6]$ min	$10 \to 11$	$s18{:}g2 \to g5$, $s52{:}g6 \to g8$, ... $(+22)$
$(6, 8]$ min	$11 \to 10 \to 9 \to 7 \to 8$	$s111{:}g7 \to g5$, $s16{:}g2 \to g7$, ... $(+27)$
$(8, 10]$ min	$8 \to 6 \to 7 \to 6 \to 4$	$s111{:}g5 \to g4$, $s45{:}g5 \to g2$, ... $(+23)$
$(10, 12]$ min	$4 \to 5 \to 7 \to 8 \to 7 \to 6 \to 7$	$s102{:}g1 \to g3$, $s102{:}g3 \to g2$, ... $(+68)$

Runtime scheduling behavior. Table 11 presents a representative 12-minute snapshot of TurboServe's scheduling decisions on Trace 1 running on Cluster 1. The autoscaling decisions show that TurboServe continuously adjusts the active GPU budget according to workload demand. For example, the budget first contracts from 8 GPUs and then gradually expands during the initial window, continues scaling out as the workload increases in $(2, 6]$ minutes, and later scales in during lower-demand periods. The migration decisions demonstrate how the placement controller reacts to these budget changes and runtime load imbalance by moving sessions across GPUs. Most windows involve tens of session movements, and the final window contains more movements because session placement becomes more imbalanced under the current GPU budget. Overall, the trace shows that autoscaling adjusts cluster capacity at a coarser timescale, while migration more frequently corrects transient placement imbalance, illustrating the complementary roles of the two controllers during online serving.

Table 12: Session migration overhead across different cluster and model configurations on Trace 1. Per-chunk latency and migration overhead report the average chunk-generation latency and average GPU-GPU session migration latency, respectively.

Metric	H20 (1.3B)	H20 (7B)	B300 (1.3B)	B300 (7B)
Per-chunk Latency	1054 ms	1201 ms	917 ms	1181 ms
Migration Overhead	23 ms (2%)	24 ms (2%)	24 ms (3%)	30 ms (3%)

Session migration overhead. Table 12 reports the average session-migration overhead across different cluster and model configurations. We measure the end-to-end migration latency during online GPU-GPU rebalancing, including buffer allocation/release, direct state transfer, and session

ownership updates. Across all settings, the migration overhead remains a small fraction of the per-chunk generation latency, ranging from 23-30 ms (2%-3% of per-chunk latency). Larger models incur higher absolute migration overhead due to larger per-session states, but the overhead remains a small and stable fraction of per-chunk latency across both H20 and B300 deployments. These measurements indicate that session migration incurs only a small runtime penalty relative to chunk generation, making it practical for online rebalancing when the expected latency reduction exceeds the transfer cost.