SHEPHERD: Serving DNNs in the Wild

Hong Zhang
University of Waterloo

Yupeng Tang
Yale University

Anurag Khandelwal
Yale University

Ion Stoica
UC Berkeley

Abstract

Model serving systems observe massive volumes of inference requests for many emerging interactive web services. These systems need to be scalable, guarantee high system goodput and maximize resource utilization across compute units. However, achieving all three goals simultaneously is challenging since inference requests have very tight latency constraints (10–500ms), and production workloads can be extremely unpredictable at such small time granularities.

We present SHEPHERD, a model serving system that achieves all three goals in the face of workload unpredictability. SHEPHERD uses a two-level design that decouples model serving into planning and serving modules. For planning, SHEPHERD exploits the insight that while individual request streams can be highly unpredictable, aggregating request streams into *moderately-sized groups* greatly improves predictability, permitting high resource utilization as well as scalability. For serving, SHEPHERD employs a novel online algorithm that provides guaranteed goodput under workload unpredictability by carefully leveraging preemptions and model-specific batching properties. Evaluation results over production workloads show that SHEPHERD achieves up to 18.1× higher goodput and 1.8× better utilization compared to prior state-of-the-art, while scaling to hundreds of workers.

1 Introduction

Model inference has grown to become a critical component of many interactive applications [1–11]. Facebook, for instance, serves tens of trillions of inference requests per day [12]. Compared to model training, model inference dominates production costs: on AWS, inference accounts for over 90% of the machine learning infrastructure cost [13]. This has driven significant effort in the design of model serving systems to serve inference requests from several applications with deep neural network (DNN) architectures, often using hardware accelerators like graphics processing units (GPUs) to meet tight per-request latency service-level objectives (SLOs), *e.g.*, 50–500ms. These systems typically group requests with the same SLO and target model into separate request streams, and must make two types of scheduling decisions across them to meet system goals. First, they make request serving decisions to maximize *system goodput*, i.e., the number of requests that meet their SLO deadlines per unit time. Second, they make resource provisioning decisions in order to *scale* to a massive number of request streams using large pools of GPUs, while ensuring *high utilization* for the GPU pool for cost-efficiency.

We find that meeting these goals is challenging due to *short-term workload unpredictablity*: our analysis of both synthetic and production workloads (§2.2) indicates that while the average request arrival rates are predictable over longer timescales (i.e., hours), they are bursty and unpredictable at smaller time granularities (i.e., millisconds) that must be considered when scheduling requests to meet their SLO deadlines. As such, existing solutions [3–11] fail to meet one or more of the above goals due to two key reasons.

First, existing systems expose a hard tradeoff between resource utilization and scalability under short-term unpredictability, as they typically employ one of two classes of scheduling policies: (1) *periodic per-stream* policies [3–9], which make scheduling decisions (i.e., resource provisioning, batch sizing, load balancing, etc.) for each stream of requests separately in a periodic manner, and (2) *online global* policies, which make scheduling decisions in an online manner by time-multiplexing the entire pool of resources (*e.g.*, GPUs) across all request streams [10, 11]. On one hand, while the periodic and per-stream nature of scheduling for the former permit scaling to many request streams and compute resources, these systems must over-provision resources to handle unpredictable bursts of requests during each period, resulting in poor resource utilization. On the other hand, online global policies can achieve higher resource utilization by adapting the amount of resources allocated to each stream in an online fashion, but scale poorly with the number of request streams and size of the resource pool due to the increased complexity of online scheduling decisions.

Second, existing approaches are fundamentally unable to provide any *guarantees on system goodput* under unpredictable workloads. We establish several important theoretical results to show why this is fundamental (§5). First, making the optimal scheduling decisions (*e.g.*, executing, deferring or dropping a request) requires future knowledge of request arrival patterns, and even with perfect knowledge, the problem is NP-complete. Second, no online algorithm can achieve goodput that is even within a constant factor of the optimal with perfect knowledge *without using preemption*. Since existing approaches [10] employ simple heuristics without considering preemption, their performance can be arbitrarily worse than the optimal under unpredictable workloads (§2.2).

This raises the question: *Is it possible to design a model*

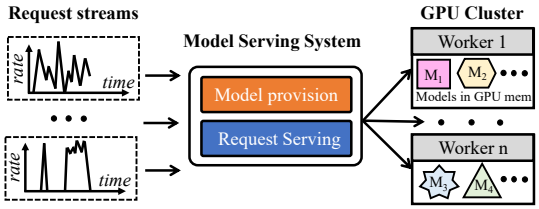


Figure 1: High-level architecture of model serving system

serving system that is scalable, achieves high utilization, and provides guaranteed high goodput under unpredictable serving workloads? In this paper, we answer the above question in affirmative with SHEPHERD, a model serving system resilient to workload unpredictability.

To break the utilization-scalability trade-off exposed by existing solutions, we make an important observation: while individual request streams can be highly unpredictable, aggregating them into *groups* permits accurate resource provisioning. Moreover, our analysis show that even moderately-sized groups comprising hundreds to thousands of streams can already offer reasonable predictability (§3.1). SHEPHERD realizes this insight into a *two-level* design that decouples model serving into a *periodic planning* phase and an *online serving* phase. For the planning phase, we introduce HERD, a planner that periodically classifies inference request streams, DNN models, and GPUs into several *serving groups*. Then based on the planning results, the serving phase employs an online algorithm FLEX to serve requests across streams within each serving group *independently*. HERD solves an ILP to efficiently balance utilization and scalability (§4): on one hand, HERD limits the size of each group, restricting the online scheduling algorithm's decision space to a limited number of streams and GPUs within each group. On the other hand, HERD provisions a sufficient number of streams and GPU workers for each serving group to maximize utilization.

To achieve guaranteed high goodput, we design FLEX (§5), an online scheduling algorithm that leverages preemption and model-specific batching properties. First, we note that while preemption permits correcting for sub-optimal scheduling decisions in the online setting, preempting too often can result in significant amount of wasted work. As such, FLEX carefully weighs the utility of the currently running batch of requests against pending candidate requests to decide whether or not the running batch should be preempted. Second, FLEX leverages a model-specific relationship between the batch size in batched inference execution and its execution latency to determine appropriate batch sizes and the order of execution across request streams. We show that both techniques work in concert to achieve SHEPHERD's goodput guarantee.

We implement SHEPHERD (§6) and evaluate it using a combination of testbed experiments and large-scale emulations with both production and synthetic workloads (§7). Our results show that (1) SHEPHERD achieves up to 18.1× higher goodput and 1.8× higher utilization than periodic per-stream solutions, (2) SHEPHERD achieves up to 5.2× higher goodput compared to heuristic-based online approaches, and (3) SHEPHERD's goodput scales linearly with the number of workers.

2 Background and Motivation

We begin with an overview of model serving systems (§2.1) and short-term workload unpredictability (§2.2).

2.1 System Model and Goals

We focus on Deep Neural Network (DNN) model serving systems [3–11] deployed on GPU clusters (Figure 1). Users issue inference requests, which the system must serve using a specific DNN model on one of its GPU workers within a latency SLO specified for the request, typically 10–500ms [6]. Requests for the same model and with the same latency SLO are typically grouped into a *request stream*, with arbitrary request arrival patterns within each stream. In serving these streams, serving systems can benefit significantly by batching requests on GPUs — on an NVIDIA GTX1080, batching together 32 inference requests improves model serving throughput by 4.7–13.3× for VGG, ResNet and Inception models relative to serving them individually [6]. Taking the above constraints into account, the system makes two scheduling decisions: *model provisioning* decisions to determine which models should be loaded on which and how many GPUs, and *request serving* decisions to determine:

- *batch size:* how many requests to be executed in a batch,
- *batch priority:* which batch should be executed first, and,
- *target GPU:* which GPU to execute the batch on.

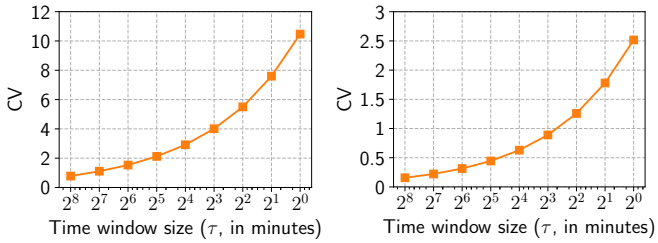
Note that although multiple batches can be executed on one GPU worker concurrently, their execution time becomes nondeterministic due to poor performance isolation on GPUs. As such, most model serving systems [3–11] execute one batch at a time for performance predictability.

The key performance goal for a model serving system is to maximize the system goodput, or the number of served requests that meet their SLO requirements per unit time; requests that fail to meet them often hold no utility for the user. Since serving systems must cater to thousands of requests streams [1, 12], the system should also scale to large clusters with thousands of GPUs in order to serve them. Finally, since inference pipelines comprise the majority of the machine learning infrastructure costs in production settings [13], serving systems should target high resource utilization of the GPU clusters to maximize cost-efficiency.

2.2 Short-term Workload Unpredictability

We find that a key challenge in achieving all three of the goals outlined above is *short-term workload unpredictability*[1] — while the average request arrival rates are predictable over longer timescales (*e.g.*, hours), they can be quite unpredictable at smaller time granularities (*e.g.*, milliseconds) that must be

[1] Unpredictability in request arrival patterns is orthogonal to *performance predictability* demonstrated in prior works [6,10], where the execution latency for inference requests on GPUs is often quite predictable.



(a) Production workload (b) Synthetic workload

Figure 2: **The coefficient of variance (CV) over the number of request in each time window vs window size (τ, in minutes).** The CV value increases dramatically as time window size decreases.

Solutions	Utilization	Scalability	Goodput
Periodic, per-stream policies [3–9]	✗	✓	✗
Online, global policies [10,11]	✓	✗	✗
SHEPHERD	✓	✓	✓

Table 1: Existing solutions under short-term unpredictability

considered to meet per-request SLO deadlines. Next, we show the presence of short-term unpredictability and its impact for both production and synthetic application workloads. Since we are unaware of any publicly available production traces for inference workloads, we use Microsoft's recently released traces for Azure Functions [14] for our production workload, which is noted by recent work to be representative of real-world inference workloads in terms of both diurnal patterns and short-term burstiness [4, 10]. The trace contains the number of function invocations performed at minute granularities across $\sim 46k$ applications over a two-week period. Our synthetic workload simulates $1k$ user request streams as Poisson processes with average arrival rate following an exponential distribution, a commonly-used approach in approximating human-generated invocations [3, 4, 14].

To study workload unpredictability, we divide the entire time period into non-overlapping time windows of size τ, and compute the number of requests $r_{t,s}$ in each time window t for every stream s. We quantify unpredictability in each stream using the coefficient of variance — the ratio of the standard deviation to the mean across $r_{t,s}$. Note that meeting 10–500ms request SLOs requires optimizing scheduling decisions in time window sizes (τ) of hundreds of milliseconds. Figure 2 shows the average coefficient of variance across all streams for different values of τ: for both synthetic and production workloads, coefficient of variance increases drastically as τ decreases. Clearly, while statistical models may be able to estimate average arrival patterns at hours time-scales, the high coefficient of variance at even minute-granularity makes sub-second request arrival patterns nearly impossible to predict.

Under short-term workload unpredictability, existing solutions [3–11] are unable to meet one or more of the three performance goals outlined in §2.1 (Table 1):

Poor resource utilization. Many existing approaches [3–9] make *periodic* provisioning and serving decisions for each user stream *independently*. Within each period (typically a few minutes to hours), inference requests are served following a fixed schedule determined at the beginning of the period. Since scheduling decisions are computed per-stream and updated only periodically, such approaches can scale to many streams over massive pools of GPUs. However, these approaches also tend to over-provisioning GPUs in order to maximize the number of request SLOs met in the presence of short-term burstiness, resulting in poor resource utilization.

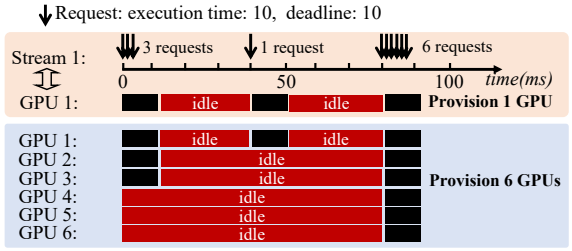
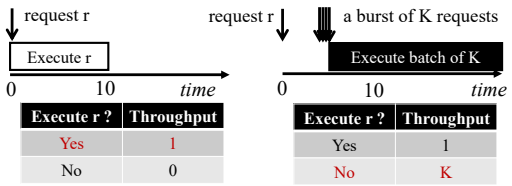


Figure 3: **Periodic per-stream policies observe poor utilization.** Request arrival pattern is shown at the top, with each request's execution time as well as latency SLO being 10ms. Provisioning one GPU (top) based on average load causes 70% of the requests to miss their deadline. Provisioning six GPUs (bottom) allows all requests to meet their SLOs, but reduces resource utilization to 17%.

As a concrete example, Figure 3 shows a user stream with average arrival rate of 1 request every 10ms, with each request's execution time and latency SLO being 10ms as well. The bursty nature of the workload causes three requests to arrive at t=0ms, one at t=40ms and six at t=80ms. Provisioning one GPU for the stream based on the average load would cause 7 out of 10 requests to miss their SLO deadlines — two from the first burst and five from the last. Provisioning six GPUs permits all request latency SLOs to be met, but reduces the resource utilization to 17%, since the GPUs are collectively idle for 500ms out of 600ms cumulative runtime.

Poor scalability. An alternate approach employed by other serving systems is to time-multiplex the GPU cluster across different user streams to achieve better resource utilization [10, 11]. Instead of provisioning and scheduling request for each stream independently and periodically, the system scales the number of GPUs allocated to each stream in an online manner in response to workload fluctuations. While this results in better resource utilization, it also limits system scalability — scheduling decisions to maximize system goodput grow super-linearly in computational complexity with both the number of request streams as well as the number of GPUs they are served over. Our scalability evaluation of Clockwork [10], a recent model serving system that employs such an approach, shows that its goodput does not scale beyond a hundred GPU workers, saturating at $\sim 50k$ requests/second (§7.1). In contrast, real-world inference serving load at Facebook can be as high as 2.3 billion requests/second [12].

Lack of goodput guarantees. Maximizing goodput is challenging under short-term unpredictability. To see why, consider the example in Figure 4, where a request r with an exe-



(a) Scenario A (b) Scenario B

Figure 4: **Example highlighting challenges in online scheduling with short-term unpredictability.** The optimal scheduling decision for request r at time $t = 0$ depends on future arrivals: the performance can be far from optimal depending on the scenario and the scheduling decision to either execute request r or to drop it.

cution duration of 10ms arrives at time $t = 0$. The request has a tight SLO deadline that necessitates its immediate execution for the deadline to be satisfied. The scheduling algorithm has two choices: to schedule the request, or drop it. Unfortunately, the optimal decision to maximize system goodput depends on the future arrival pattern. Specifically, in Scenario A, since no other request arrives during r's execution, the optimal choice is to serve the request. In scenario B, however, where a large burst of K requests with equally tight deadlines arrive at time $t = 5$, the optimal decision is drop r, since it would prevent K request SLOs from being satisfied in favor of one. Note that if the SLO deadline for r was not as tight, the scheduler would have yet another choice to consider — whether or not to defer r's execution so that it may be batched with future requests.

Since future arrival patterns cannot be accurately predicted in the short-term, making the right scheduling choice is inherently hard. Existing solutions rely on simple heuristics, which provides no guarantees on how far the performance could be from the optimal. While they perform well on certain workloads, their performance can be arbitrarily worse than the optimal under unpredictable workloads, similar to the above example. We validate this observation experimentally in §7.2.

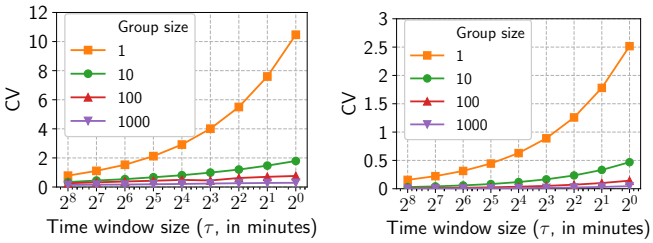
3 SHEPHERD Design

We now outline SHEPHERD's key design elements.

3.1 Overcoming Short-term Unpredictability

We leverage three key observations to overcome the challenges introduced by short-term unpredictability (§2.2):

Group-level predictability and group multiplexing. We observe that while the short-term arrival pattern for individual request streams are hard to predict, the aggregated arrival pattern across a group of request streams tends to be much more predictable. We validate this observation by considering the same workloads in Figure 2, but randomly classifying the request streams into *serving groups* of different sizes and measuring the coefficient of variance *per-group* instead of *per-stream*. Figure 5 shows that increasing the group sizes drastically reduces the coefficient of variance even at smaller window sizes. Note that the networking community has long made similar observations for bursty network flows, where statistical multiplexing can drastically improve utilization



(a) Production workload (b) Synthetic workload

Figure 5: **Coefficient of variance (CV) *for groups of streams* vs window size (τ, in minutes).** The CV increase with decreasing window sizes is much slower for larger group sizes.

by dynamically sharing a network link across network flows based on their instantaneous demands [15–18].

However, unlike multiplexing a network link across a few flows, multiplexing thousands of GPU workers across tens of thousands of SLO-bound request streams in real time presents a significant scalability challenge. To address it, we observe that even at moderate group sizes (100–1000), the per-group coefficient of variance is small enough to make its arrival pattern highly predictable (Figure 5). This motives a *group multiplexing* approach that first partitions the GPU cluster and request streams into moderately-sized serving groups (§4), then applies statistical multiplexing per-group to perform online scheduling (§5). This approach offers a means to break the tradeoff between resource utilization and scalability faced by existing systems: moderately sized groups are predictable enough even in the short-term to accurately provision their resources for high resource utilization. At the same time, restricting the online scheduling algorithm's decision space to streams and GPUs assigned to each group drastically limits its computational complexity, allowing the system to scale to much larger number of request streams and GPUs.

Preemption to correct for scheduling errors. As noted in the example from Figure 4, the optimal scheduling decision often depends on future arrival patterns, which can be hard to predict. As such, any non-clairvoyant online algorithm is bound to occasionally make sub-optimal scheduling decisions. We find that the ability to correct such decisions when its sub-optimality becomes apparent via preemptions is *necessary* for achieving performance guarantees for an online scheduling algorithm. For instance, in the example of Figure 4, a solution can correct for a sub-optimal scheduling decision in both scenarios by simply preempting r if a burst of requests arrives later. Preemptions in online scheduling algorithms are not a new concept; they have been used in a variety of scheduling contexts [19, 20] to achieve bounds on the algorithm's *competitive ratio* — the ratio between its performance and that of an optimal offline algorithm. Leveraging insights from recent work on context switching for DNN training on GPUs [21] allows us to realize preemptions efficiently for inference workloads (§6), and combining it with model-specific batch-latency relationships (described next) permits bounding

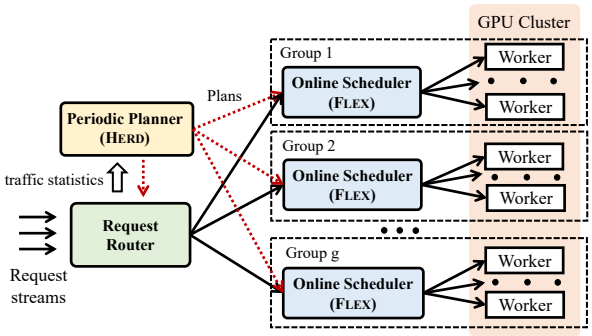


Figure 6: Overview of SHEPHERD design

the competitive ratio for online model serving.

Model-specific batch-latency relationships. Empirical measurements in prior work [6] indicate that a simple linear model can accurately describe the execution latency for varying request batch sizes in model serving workloads. In particular, for a batch B of size $|B|$ being executed on a model m, the execution latency $\ell_m(B)$ is given by:

$$\ell_m(B) = \alpha_m \cdot |B| + \beta_m \tag{1}$$

where β_m is the baseline execution latency for executing an empty batch on the model, while α_m is the latency for each additional request in the batch.

We find exploiting this relationship helps make better scheduling *and* stream grouping decisions. First, larger batches help amortize the fixed cost β_m and achieve higher throughput, but too large a batch may miss the SLO deadline altogether. As such, making scheduling and preemption decisions that leverage the batch-latency relationship to prioritize appropriately large batches that are likely to meet their deadline, permit better performance guarantees for the online scheduling algorithm. Second, when scheduling requests across streams in a serving group of certain models, we find that the online algorithm can achieve better performance guarantees if the models have similar α and β values (§5.2).

We next describe how SHEPHERD incorporates all of these insights into an end-to-end design.

3.2 Design Overview

SHEPHERD leverages group-level predictability in a two-level design that comprises a periodic planning and an online serving component. At a high-level, the periodic planning component leverages long-term load statistics to partition the entire GPU cluster into several serving groups, and determines how models and request streams querying them are mapped to these groups to optimize both resource utilization and system scalability. The online serving component, on the other hand, schedules requests from streams in each serving group across the group's allocated GPUs, and ensures that its goodput is always within a constant factor of the optimal schedule.

SHEPHERD's architecture (Figure 6) comprises four key components: a planner (HERD), a request router, a scheduler (FLEX) per serving group and multiple GPU workers. HERD executes periodic planning, and informs each GPU worker which serving group it belongs to and which models it must serve. HERD also assigns a group-level scheduler to each serving group — the total number of group-level schedulers can be scaled based on the number of models being served by the system and the aggregate load across them. The request router forwards client inference requests to group-level schedulers based on their target model, and collects statistics regarding their arrival patterns that HERD employs to compute group-level mappings. The group-level schedulers, in turn, execute our online scheduling algorithm, FLEX, to schedule inference requests across GPU workers in their own serving group.

HERD (§4). While even random assignment of models and GPU workers to serving groups can achieve decent workload predictability (§3.1), achieving high utilization and guaranteed goodput requires considering a number of additional constraints. To this end, HERD formulates this assignment problem as an Integer Linear Program (ILP) incorporating all such constraints. In particular, as noted in §3.1, colocating models with similar α, β values (Eq. 1) in the same serving group yields better goodput guarantees in FLEX. Consequently, HERD also incorporates model-affinity — a measure of similarity across α, β values — in its ILP.

FLEX (§5). FLEX's goal is to provide guaranteed high goodput for each group under short-term unpredictability. To this end, we answer three key theoretical and practical questions:

- **What performance guarantees are possible?** We first establish two impossibility results. We show that determining an optimal solution is NP-hard, even in the offline setting. In the online setting, we show that no online algorithm can achieve performance *competitive* with the optimal offline solution without using preemption. Since prior model serving systems do not employ preemption, they are fundamentally unable to provide any performance guarantees.
- **What performance guarantees can FLEX provide?** FLEX ensures that for each serving group, the aggregated goodput achieved is guaranteed to be at most $12.62 \cdot K\times$ worse than the optimal offline schedule with complete knowledge of the future. K is a model-affinity parameter that reduces to one if all models in the serving group have the same α and β, and increases if they diverge (§5.2).
- **How does FLEX achieve this guarantee?** FLEX leverages two key insights outlined in §3.1: preemption to correct for scheduling errors, and model-specific batch-latency relationships. First, preempting a scheduled batch requires carefully weighing the utility brought by the scheduled batch of requests against the utility of the new batch to be scheduled — the threshold beyond which preemption is performed significantly impacts the performance bound FLEX can achieve. Second, FLEX leverages the model-specific relationship in Eq. 1 to determine appropriate batch sizes

Decision variables	Definition
$x_{ij} \in \{0,1\}$	Is stream i mapped to group j?
$y_{cj} \in \{0,1\}$	Is affinity-set c mapped to group j?
$z_{kj} \in \{0,1\}$	Is model k mapped to group j?
$size_j \in \mathcal{N}+$	# of GPUs allocated to group j
Input parameters	**Definition**
mem	GPU memory capacity
$\overline{G}$	Scalability limit for # of GPUs per group
N	# of GPUs in cluster
$h_{ki} \in \{0,1\}$	Does stream i use model k?
$q_{ck} \in \{0,1\}$	Does affinity-set c include model k?
Optimization goal	**Definition**
$bt(i)$	The burst tolerance metric for stream i

Table 2: Variables used in HERD's ILP.

and their order of execution across request streams.

4 Periodic Planner: HERD

HERD operates in two steps. It first determines the number of GPUs n_i that would be needed to sustain the average load $rate_i$ for each request stream separately. To do so, HERD empirically measures the maximum goodput T_i each stream i can achieve on a single GPU, and uses it to compute n_i as $\frac{rate_i}{T_i}$. It uses n_i to define a new *burst tolerance* metric (bt) that captures the increase in load that the stream can tolerate if assigned to a particular serving group relative to the average-load based assignment of GPUs. More formally,

$$bt(i) = \frac{\text{\# GPUs } i \text{ can use for its peak load}}{\text{\# GPUs } i \text{ needs for its average load}} = \sum_j \frac{size_j \cdot x_{ij}}{n_i}$$

where x_{ij} is 1 if stream i is assigned to group j (0 otherwise), and $size_j$ is the number of GPUs assigned to group j.

Second, HERD uses an Integer Linear Program (ILP) to combine streams into serving groups to *maximize the minimum burst tolerance across all streams*; this captures the goal of ensuring every stream can tolerate as heavy a burst as possible, subject to a certain set of constraints:

(a) **Cluster-size limit** ensures that the total number of GPUs assigned across all serving groups is no larger than the cluster-size N (in number of GPUs).

(b) **Group-worker limit** ensures that the total number of GPUs $size_j$ assigned to each group j does not exceed the maximum scalability limit $\overline{G}$ of the online algorithm.

(c) **GPU-memory limit** ensures that the sum of model sizes assigned a serving group j does not exceed the GPU memory capacity mem.

(d) **Group surjectivity** ensures that every stream i is assigned to a single group j, and only if its associated model is also assigned to group j.

(e) **Affinity-set surjectivty** ensures that models assigned to the same group j have similar α, β values (as defined in Eq. 1) to ensure better performance guarantees in FLEX.

We capture the divergence in model α, β values as K (defined in §5), and pre-compute affinity-sets $c_1, c_2, \ldots$ as a partitioning of models such that K between any two models in an affinity set is $\leq \overline{K}$; this simplifies our ILP constraint to only picking models from the same cluster.

Our ILP is presented below, with variables listed in Table 2:

$$\textbf{maximize} \quad \min_i \{bt(i)\} \tag{2}$$

$$\textbf{s.t.} \quad \sum_j size_j \leq N, \qquad \text{((a) Cluster-size limit)}$$

$$size_j \leq \overline{G}, \quad \forall j \qquad \text{((b) Group-worker limit)}$$

$$\sum_k z_{kj} \cdot |m_k| \leq mem, \quad \forall j \qquad \text{((c) Memory limit)}$$

$$\left.\begin{aligned} &\sum_j x_{ij} = 1, \quad \forall i \\ &h_{ki} \cdot x_{ij} \leq z_{kj}, \quad \forall i,j,k \end{aligned}\right\} \qquad \text{((d) Group surjectivity)}$$

$$\left.\begin{aligned} &\sum_c y_{cj} = 1, \quad \forall j \\ &q_{ck} \cdot z_{kj} \leq y_{cj}, \quad \forall i,j,k \end{aligned}\right\} \qquad \text{((e) Affinity-set surjectivity)}$$

Note that the above formulation is not linear due to the non-linear optimization goal, which contains: (1) a max-min term, and, (2) a product between binary and non-negative variables ($x_{ij} \cdot size_j$). However, both can be linearized using standard techniques [22] — we omit the linearized ILP for brevity. Similar to prior work [6], HERD ensures that all models to be served by a worker in the subsequent online serving phase are present in GPU memory, with some memory set aside for the operation of the online algorithm, FLEX. We discuss additional challenges due to memory constraints in §8.

HERD complexity and periodicity. Since solving HERD's ILP is NP-hard, and we must scale to millions of streams and thousands of workers, we first aggregate streams using the same model into a single "model-stream", then apply the ILP to optimize the burst tolerance metric across the model-streams. The burst tolerance metric of the model-stream the lower bound of the burst tolerance metric for each stream in it. Note that different streams in the model-stream may have different SLOs, but this will not affect the correctness of our ILP, since none of the constraints (a) – (e) depend on per-stream SLO. Instead, FLEX incorporates the impact of SLOs across different streams during online serving.

Also, note that we only need to ensure that the ILP solver is much faster than HERD's periodicity, which, in turn, depends on how frequently the workload characteristics change enough to require recomputing group assignments. Fortunately, our analysis of Microsoft's Azure Function trace [14] shows that the workloads within moderately-sized serving groups remain stable for tens of minutes or more, while our solver can compute a plan within a few seconds (§7.3).

Input variables	Definition
$S = \{r_1, r_2, ...\}$	A request stream from one application.
a_r, d_r, m_r	Arrival time, deadline, model for request r.
$a(B), d(B), m(B)$	Arrival time, deadline and model for batch B.
$\mathbb{B} = \{B_1, B_2, ...\}$	Set of all possible batches.
Decision variables	**Definition**
$I(B,t,n) \in \{0,1\}$	Is batch B is executed at time t on GPU n?

Table 3: Notations for online batch scheduling.

5 Online Serving Algorithm: FLEX

We first formulate the online serving problem (§5.1) and then present the FLEX algorithm to provide guaranteed goodput under short-term unpredictability (§5.2).

5.1 Problem Formulation

Our online serving setting focuses on scheduling inference requests across models and GPUs assigned to a single serving group. Requests within each stream query the same model with the same latency SLO. Each request r has an arrival time a_r, deadline d_r and queries model m_r. Requests are served in batches; for a batch B, arrival time $a(B)$ is the arrival time of the most recent request in B, and deadline $d(B)$ is the earliest deadline of all requests in B. Let $\mathbb{B}$ be the set of all possible batches of requests; the online serving algorithm decides whether to execute batch $B \in \mathbb{B}$ at time t on GPU n, which we capture as the decision variable $I(B,t,n) \in \{0,1\}$. The goal of online serving is to maximize the overall goodput: the number of requests that meet their SLOs per second. Table 3 summarizes the notations for our problem formulation.

Optimal offline serving algorithm. We find that the offline serving problem where the scheduler has access to the complete future can be formulated as the following Zero-one Integer Linear Program (ZILP):

$$\textbf{maximize} \sum_t \sum_n \sum_{B\in\mathbb{B}} |B| \cdot I(B,t,n) \tag{3}$$

$$\textbf{s.t.} \sum_t \sum_n \sum_{\{B|r\in B\}} I(B,t,n) \le 1, \quad \forall r \tag{a}$$

$$\sum_{B\in\mathbb{B}} \sum_{\{t'|t'\le t\le t'+\ell_{m_B}(B)\}} I(B,t',n) \le 1, \quad \forall t,n \tag{b}$$

$$a(B)\cdot I(B,t,n) \le t, \quad \forall B,t,n \tag{c}$$

$$(\ell_{m_B}(B)+t)\cdot I(B,t,n) \le d(B), \quad \forall B,t,n \tag{d}$$

$$I(B,t,n) \in \{0,1\}, \quad \forall B,t,n \tag{e}$$

Intuitively, the ZILP maximizes the total number of requests that meet their latency SLOs across all selected batches ($I(B,t,n) = 1$), which in turn maximizes the total goodput. The ZILP constraints correspond to:

(a) Each request can be executed in at most one batch,

(b) A GPU can only execute one batch at a time,

(c) No selected batch can start before its arrival time,

Algorithm 1 FLEX Algorithm

1: **Initialize:**
2: **for** each model m **do**
3: $\quad Q_m \leftarrow$ Priority queue of m's requests sorted by deadlines.
4: **Event: On completion of a batch on any GPU** n**:**
5: $B_{g,n} \leftarrow$BATCHGEN(n) # Largest feasible batch across all Q_m
6: Execute $B_{g,n}$ and dequeue requests in $B_{g,n}$ from model queue
7: **for** each GPU n **do**
8: $\quad B_{g,n} \leftarrow$BATCHGEN(n) # Update candidate batch
9: **Event: On arrival of request** r**:**
10: Enqueue r to corresponding queue
11: **for** each GPU n **do**
12: $\quad B_{c,n} \leftarrow$ The batch currently being executed on GPU n
13: $\quad B_{g,n} \leftarrow$BATCHGEN(n)
14: $\quad$ **if** $B_c = \emptyset$ **then**
15: $\quad\quad$ Execute $B_{g,n}$ and dequeue requests in $B_{g,n}$
16: $\quad$ **else if** $|B_{g,n}| \ge \lambda \times |B_{c,n}|$ **then** # Preemption rule
17: $\quad\quad$ Preempt $B_{c,n}$
18: $\quad\quad$ Execute $B_{g,n}$ and dequeue requests in $B_{g,n}$
19: $\quad\quad$ Treat requests in $B_{c,n}$ as new arrivals (go to Line 11)

(d) Every selected batch must finish before its deadline, and

(e) The decision variable $I(B,t,n)$ must either be 1 or 0.

Clearly, the optimal solution to the above ZILP is also the optimal offline schedule. Obtaining such an optimal is unrealistic — not only is it impractical to have access to the complete future (or even a reasonable prediction of it, §2), computing the optimal solution to the ZILP is NP-hard [23].

Achievable guarantees. However, the optimal offline schedule provides us with a baseline of the best schedule possible, and permits us to reason about how close an online algorithm can get to such a solution. More formally, the performance guarantee an online algorithm can achieve is typically captured by the *competitive ratio*: the worst-case ratio of the ZILP's goodput to the online algorithm's goodput over all possible inputs. Note that our focus is on online request serving decisions, so we assume both algorithms have the same resources provisioned to them. We establish the following important result regarding the competitive ratio:

Theorem 5.1 *No non-preemptive, deterministic algorithm can achieve a bounded competitive ratio for online serving.*

We defer the proof to Appendix A, but note that since existing online serving algorithms [6, 10, 11] are non-preemptive, they are incapable of achieving a bounded competitive ratio.

5.2 FLEX Algorithm

Algorithm 1 presents our FLEX algorithm that achieves a bounded competitive ratio for online serving. During initialization, FLEX creates a priority queue Q_m for each model m, which holds requests sorted by tightness of their deadlines. The algorithm reacts to two key events: (1) *completion event* of a batch on any GPU, and, (2) *arrival event* of a new request.

For a completion event, FLEX simply generates a new batch B_g and executes it; all requests in B_g are dequeued from corresponding model queue Q_m. To generate the new batch, FLEX finds the largest feasible batch across all queues, such that all requests in the batch can meet their latency SLOs.

For an arrival event, FLEX generates a candidate batch for each GPU as outlined above, and compares it with the currently running batch. If the generated batch is λ times larger than currently running batch, the current batch is preempted. If preemption occurs, requests in preempted batch that can still meet their SLOs are re-enqueued to their corresponding priority queues. The re-enqueued requests will be treated as newly arrived requests so they can be scheduled again.

We now dive deeper into salient features of the algorithm.

Choice of λ. The preemption threshold λ plays a crucial role in bounding FLEX's competitive ratio. A conservative preemption policy with larger λ can result in a poor competitive ratio, while an aggressive preemption policy with smaller λ can waste GPU resources, since the preempted work does not contribute to system goodput. As such, we express the competitive ratio in terms of λ, and formulate the problem of finding the optimal competitive ratio as an optimization problem. Solving this problem yields the optimal value of $\lambda \approx 3.03$ (Theorem 5.2). Note that while a worker may experience cascading preemptions if batches keep arriving with sizes $\lambda\times$ than the currently executing batch, our choice of λ ensures that the total wasted work is always much less than the additional useful work performed post-preemption. In practice, the effect of cascading preemptions is bounded due to our maximum batch size limit (128 by default). We defer the description of our preemption implementation to §6.

Prioritizing batches for a single model. Online job scheduling algorithms [19, 20, 24–27] tend to consider one of two key metrics as optimization goals: a job's *value*, and its *value density*. In the online model serving context, the value of a job (batch) corresponds to the number of requests it contains (i.e., its batch size), while the value density corresponds to its contribution to system goodput (i.e., $\frac{\text{batch size}}{\text{batch latency}}$). Traditional online job scheduling algorithms often fail to achieve a bounded competitive ratio since optimizing these two goals are often at odds with each other, i.e., optimizing total value density comes at the cost of optimizing total value across jobs, and vice versa. Fortunately, Eq. 1 establishes a linear relationship between value density and value for batches of inference requests: for a single model, *larger batches always contribute more to system goodput*. As such, our preemption and batch generation criteria always favor larger batches to maximize total value and value density simultaneously, enabling FLEX to achieve a bounded competitive ratio. In contrast, prior slack-based prioritization schemes (*e.g.*, tightest deadline first [10]) are unable to provide such guarantees. In fact, our evaluation (§7) shows that prioritizing larger batches over those with tigher deadlines leads to higher goodput under high load.

Extending FLEX to multiple models. While the above prioritization scheme is straightforward when a single model is involved, extending FLEX's competitive ratio analysis to a multi-model scenario is challenging, since the linear relationship between batch value and value density no longer holds *across* models. However, the batch-latency relationship in Eq. 1 still allows us to bound the batch value and value density across models using the model-specific parameters α and β. More precisely, we define an affinity metric $A(m_i, m_j)$ between two models m_i and m_j as:

$$A(m_i,m_j) = \begin{cases} \frac{\alpha_i+\beta_i}{\alpha_j}, & \text{if } \alpha_j+\beta_j-\beta_i \le 0 \\ \min(\frac{\alpha_i+\beta_i}{\alpha_j}, \max(\frac{\alpha_i}{\alpha_j+\beta_j-\beta_i}, \frac{\alpha_i}{\alpha_j})), & \text{otherwise} \end{cases}$$

where $\alpha_i, \alpha_j, \beta_i$ and β_i are the model-specific parameters for models m_i and m_j respectively. While its specific formulation is devised to establish FLEX's competitive ratio (Theorem 5.2), we note that $A(m_i, m_j)$ is close to 1 if m_i and m_j have similar α and β, and deviates from 1 as the α and β values for the models diverge. For a set of models $\mathbf{M}$, we show that the competitive ratio is a multiple of K, the largest affinity value $A(m_i, m_j)$ across all pairs of models (m_i, m_j) in $\mathbf{M}$, i.e.,

$$K = \max_{i,j \in \mathbf{M}} A(m_i, m_j) \tag{4}$$

FLEX properties. Our analysis in Appendix B shows that:

Theorem 5.2 *Algorithm 1 is* $12.62 \cdot K$*-competitive with preemption threshold* $\lambda \approx 3.03$*, with K defined in Eq. 4.*

We note that FLEX is the first algorithm that achieves guaranteed performance for online model serving to the best of our knowledge. We validate FLEX's performance empirically over a wide range of representative workloads in §7. Finally, while we defer the complexity analysis to Appendix C the following result establishes FLEX's complexity:

Theorem 5.3 FLEX *has a worst-case complexity of O(G), where G is the number of GPUs in the serving group.*

6 SHEPHERD Implementation

Our SHEPHERD implementation follows the architecture described in Figure 6. The periodic planner (HERD), request router and online scheduler are implemented as C++ processes, while the GPU workers support configurable model execution runtimes like PyTorch [28] and Apache TVM [29].

Supporting preemptions. While recent hardware-based preemptions on newer GPUs [31] may enable better performance, we opt for software-based preemptions adapted from Pipeswitch [21] in SHEPHERD due to its general applicability to commodity GPUs. Pipeswitch supports preemption of DNN training tasks by inserting exit points between the training phases of different DNN layers: when a preemption is requested, the execution of the current training task can be terminated at the next exit point. Since PipeSwitch currently supports preemptions for PyTorch only, we use the PyTorch

Model	α (ms)	β (ms)	# Exit points
ResNet18 (RN18)	0.22	3.74	40
ResNet34 (RN34)	0.38	5.78	46
ResNet50 (RN50)	0.75	7.96	46
ResNet101 (RN101)	1.25	13.57	39
ResNet152 (RN152)	1.77	18.98	84
ResNeSt50 (RS50)	1.18	15.39	78
ResNeSt101 (RS101)	1.91	29.21	57
ResNeSt200 (RS200)	3.35	45.43	96
ResNeSt269 (RS269)	4.37	74.20	128
DenseNet121 (DN121)	0.69	19.96	129
DenseNet161 (DN161)	1.74	23.10	171
DenseNet169 (DN169)	0.83	27.47	120
DenseNet201 (DN201)	1.12	32.33	142
GoogLeNet (GN)	0.25	8.41	44
Inception v3 (I3)	0.96	11.77	122
R-CNN (RCNN)	2.59	14.90	51
BERT (BERT)	40.98	5.67	43

Table 4: DNN Models evaluated in SHEPHERD. BERT [30] is a popular NLP model, while the rest are popular CV models from six different model families.

runtime by default in our implementation, although the same approach could be implemented for Apache TVM as well.

Adapting the preemption approach from training to inference pipelines introduces a key challenge: while the overhead for preemption is not a major concern for long-running model training tasks, it is quite crucial to minimize preemption overheads for model inference. On one hand, adding too few exit points to an inference task introduces unacceptable *preemption delay* — the time between from the preemption being requested and actually being completed — since the preempted task may still execute for tens of milliseconds before the reaching next exit point. On the other hand, adding too many exit points slows down the normal execution of inference tasks, as each exit point introduces non-negligible *execution delay*. To better navigate the trade-off, we evaluate the preemption and execution delay overheads with different number of exit points for different DNN models via comprehensive profiling, and determine the optimal number of exit points for each individual model (§7.3). Table 4 shows the DNN models used in SHEPHERD, with their α, β values (Eq. 1) and the number of exit points. Note that adding exit points incurs a one-time offline profiling cost during model registration; this can be implemented as a part of the DNN framework, making it completely transparent to users.

7 Evaluation

We evaluate SHEPHERD to answer the following questions:

- How does SHEPHERD compare against state-of-the-art schemes for real-world workloads? (§7.1)
- How does each design component in SHEPHERD contribute to its performance gains? (§7.2)
- What overheads do SHEPHERD's preemption and periodic planning components introduce? (§7.3)

Setup. All our experiments were run on Amazon EC2. For GPU workers, our testbed experiments use 12 p3.2xlarge instances each with 8 vCPUs, 61GB RAM, and one NVIDIA Tesla v100 GPU with 16GB memory, while our large-scale emulations use m4.16xlarge instances with 64 vCPUs and 256GB RAM. The request router, periodic planner, and online schedulers are deployed on separate m4.16xlarge instances.

Metrics. We focus on goodput, utilization and scalability as our key metrics. Goodput and utilization values are averaged over 5 runs, while scalability is measured as the increase in system goodput on increasing the number of workers.

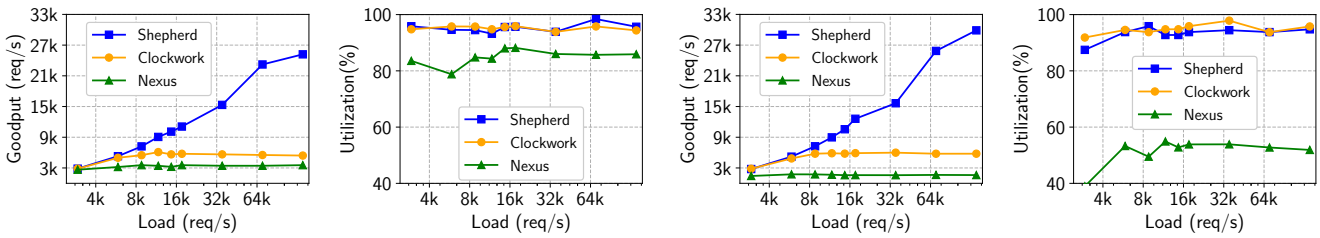
Compared schemes. We compare SHEPHERD against Clockwork [10] and Nexus [6]. Clockwork is representative of online global scheduling policies, while Nexus is a representative of the periodic per-stream approach (§1). We implement all evaluated policies in our SHEPHERD prototype and use a PyTorch-based runtime to ensure that the performance differences are solely due to the scheduling decisions rather than choices in system implementation or the underlying runtime.

For Nexus, we set the reconfiguration period to 60 seconds as recommended in [6]. Moreover, since Nexus is designed for predictable workloads, we adapt their algorithm to provision for the peak demand in every 60-second window of the workload to ensure it can sustain the provided load. For SHEPHERD, we set the GPU group-worker limit to 12, since we found it to be large enough to ensure workload predictability (due to a large enough group size) while being well within our scheduler's scalability limit. The GPU memory limit for p3.2xlarge instances is large enough to fit all 13 DNN models. Finally, we place all the models in a single affinity-set.

DNN Models. We evaluate SHEPHERD with 17 DNN models widely used for model inference (Table 4), taken from PyTorch Hub [32]. For Clockwork and Nexus, we use models without any exit points (needed for preemption in SHEPHERD, §6) to ensure they do not suffer any performance penalties for execution delays. We ensure the models remain in GPU memory for the duration of all our experiments to eliminate performance impacts of loading models into GPU memory.

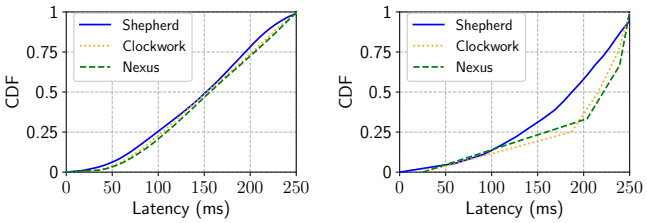
Workloads. Similar to prior work [10], we use the Microsoft's publicly-released production traces from Azure Functions (MAF) [14] as a representative production model serving workload. MAF interleaves a wide range of workloads, including heavy-sustained, low-utilization, bursty and fluctuating workloads. For our 13 profiled DNN models, we assign the 46,000 streams from MAF to models in a round-robin manner, and configure all streams with a default SLO of 250ms[2], unless otherwise specified. The MAF trace only contains the aggregated number of requests per one-minute interval for each request stream. Therefore, we generate two request ar-

[2]We use a relatively relaxed SLO compared to [10] since the PyTorch runtime used in our implementation observes longer inference latencies compared to the TVM runtime used in [10].



(a) Goodput (MAF-stable) (b) Utilization (MAF-stable) (c) Goodput (MAF-bursty) (d) Utilization (MAF-bursty)

Figure 7: **Performance variation with load.** Under high load, SHEPHERD achieves 4.6 (5.2)× and 7.1 (18.1)× higher goodput than Clockwork and Nexus under the MAF-stable (MAF-bursty) workload, respectively. SHEPHERD and Clockwork achieve high system utilization while Nexus's utilization remains under 89% (55%) across different arrival rates under the MAF-stable (MAF-bursty) workload.



(a) Low load (3*k* req/s) (b) High load (144*k* req/s)

Figure 8: **Latency CDFs for MAF-stable workload with 250ms SLO.** The latency CDF is presented for the set of requests admitted by each approach. At high load, large portions of Clockwork and Nexus request latencies are close to the SLO, while SHEPHERD's request latencies are distributed more evenly. See §7.1 for details.

rival patterns within each one-minute interval: (1) a Poisson process to model stable workloads, similar to [10] ("MAF-stable"), and (2) a more bursty Markov-modulated Poisson process (MMPP) similar to [9] ("MAF-bursty").

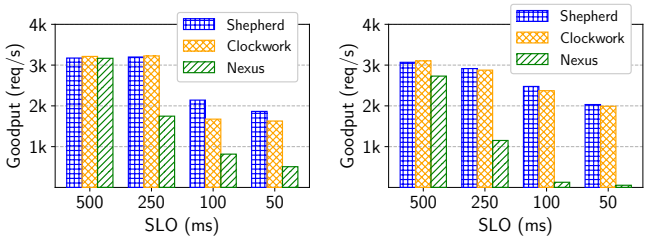
7.1 SHEPHERD in the Wild

We first evaluate the compared systems for real-world workloads on a testbed comprising 12 GPU workers and large-scale emulations that mimic work done by a GPU on CPU cores.

Performance variation with load (Figure 7). For the MAF-stable workload, with a low request arrival rate (*e.g.*, at $\sim 3k$ requests/second), all systems can meet the SLO deadlines for most requests in the workload. As such, both SHEPHERD and Clockwork achieve high system utilization (over 95%) and high goodput. At higher loads, while both systems are consistently busy serving requests (resulting in high utilization) neither SHEPHERD nor Clockwork can satisfy all request deadlines; however, since Clockwork prioritizes requests based on how close their deadline is, it greedily schedules many small batches of requests with tight deadlines, resulting in a reduced goodput. In contrast, SHEPHERD always prioritizes execution of larger batches, while the use of preemption ensures that large batches never get blocked by small batches scheduled before them. SHEPHERD can therefore efficiently utilize limited GPU resources to maximize goodput under high load, and while Clockwork's goodput starts to saturate beyond a load of 6*k* requests/second, SHEPHERD's goodput keeps increasing, outperforming Clockwork by up to 4.6× at 144*k* requests/second. We confirm that SHEPHERD's gains stem from its preemption and prioritization design choices in §7.2. We observe similar trends for Clockwork and SHEPHERD under the MAF-bursty workload.

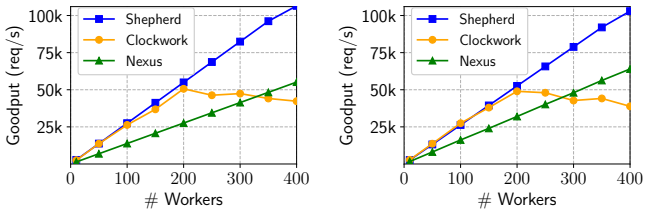
For Nexus, we find that the goodput largely remains the same as we increase the load under both MAF-stable and MAF-bursty workloads, with a goodput that is up to 7.1× and 18.1× lower than SHEPHERD. Moreover, Nexus's utilization remains under 89% for the MAF-stable workload and 55% for the MAF-bursty workload — even under high load. These observations can largely be attributed to Nexus's offline approach — during its periodic planning phase, Nexus takes the arrival rate as input and calculates the number of GPU workers required along with an offline schedule for each worker. With a fixed number of workers, Nexus can only make its planning decision assuming a specific arrival rate that it can completely satisfy, which ends up being much lower than the applied load. Moreover, during online serving phase, Nexus is unable to adjust its planning decisions dynamically based on the increased arrival rates. This impact is even more severe for the MAF-bursty workload, where predetermined execution plan is unable to adapt to periodic bursts of requests, resulting in even lower utilization (1.8× worse than SHEPHERD) and goodput relative to the MAF-stable workload.

Figure 8 plots per-request latency CDFs for SHEPHERD, Clockwork, and Nexus at low (3*k* requests/second) and high load (144*k* requests/second) for the MAF-stable workload. Note that while Figure 7(a) shows the proportion of requests admitted by each system, the CDF only depicts the latency of requests admitted by each solution. All systems observe similar latency distributions at low load (Figure 8(a)). At high load, however, a large portion of requests in Clockwork observe latency close to the SLO, since Clockwork prioritizes serving requests closer to their deadlines. Nexus also shares a similar CDF pattern, as its periodic scheduler tries to batch together as many requests as it can based on request deadlines. In contrast, SHEPHERD's request latencies are distributed more evenly; this is because SHEPHERD priortizes requests based on their batch sizes rather than their deadlines, and the evaluated workload results in batches of widely varying sizes at different times. We observe similar trends under the



(a) MAF-stable (b) MAF-bursty

Figure 9: **Goodput with varying request SLOs.** SHEPHERD outperforms Nexus and Clockwork by up to 38× and 1.3×, respectively, under tight SLOs. We omit SLOs ≤ 10ms since some of our evaluated models have higher execution latencies (Table 4).



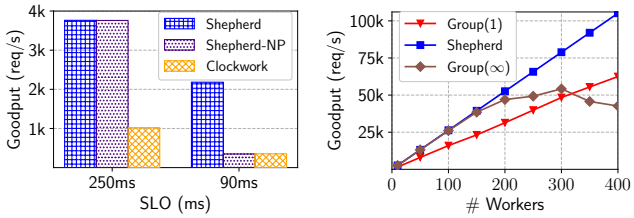
(a) MAF-stable (b) MAF-bursty

Figure 10: **Scheduler scalability with emulated workers.** For both workloads, Clockwork does not scale beyond 200 workers; Nexus scales linearly but observes 40–50% lower goodput than SHEPHERD. SHEPHERD observes both high goodput and linear scaling.

MAF-bursty workload.

Goodput with varying request SLOs (Figure 9). To understand the impact of request SLOs, we fix the arrival rate to ∼ 3*k* requests/second and measure the goodput for the compared approaches with varying SLO values. All approaches achieve high goodput with 500ms SLO, since almost all request deadlines can be met with a relaxed SLO. On reducing SLO from 500ms to 50ms, all approaches observe reduced goodput; Clockwork's reduction is smaller due to its online algorithm that prioritizes requests with tighter deadlines, while Nexus observes higher reduction, especially for the MAF-bursty workload. This is because its periodically computed static execution plan is unable to adapt to small bursts of requests, resulting in even fewer requests meeting their deadlines. However, SHEPHERD's online FLEX algorithm is able leverage prioritization and preemption to maximize the number requests that meet the stringent SLOs, outperforming both Clockwork and Nexus by up to 1.3× and 38× respectively.

Scheduler scalability (Figure 10). Due to the limited number of GPUs in our testbed, we were unable to evaluate the scalability of SHEPHERD and existing systems beyond a point. We therefore complement our testbed experiments with large-scale emulations with up to 400 *emulated* workers. As in prior work [10], an emulated worker is identical to a real SHEPHERD worker, except an inference request triggers no meaningful work; instead, they wait for a period of time determined by the corresponding model's batch-latency characteristics (Table 4), before returning a response. We run the



(a) Benefits of FLEX (b) Benefits of HERD

Figure 11: **Understanding SHEPHERD benefits.** (a) Prioritization and preemption in SHEPHERD results in 3.7× and 6.2× improvement in goodput, respectively; SHEPHERD-NP refers to a non-preemptive variant of SHEPHERD. (b) SHEPHERD achieves both high goodput and scalability with group-worker limit $\overline{G} = 12$.

MAF-stable and bursty workloads with varying number of emulated workers (N), scaling up the total load applied to the system with the number of workers. We apply a low enough load per worker to ensure any requests dropped in SHEPHERD and Clockwork are solely due to the scheduler's failure to scale to large number of workers.

Clockwork's goodput scales linearly with smaller N, slows down around $N = 150$, and saturates at 50*k* request/second around $N = 200$ since its centralized scheduler becomes the bottleneck[3] (Figure 10(a)). Nexus goodput, on the other hand, scales almost linearly with N; this is expected since Nexus's scheduling decisions are computed per-stream and updated only periodically. However, its periodically computed schedule results in ∼ 40% lower goodput than SHEPHERD. This is because Nexus's computed schedule conservatively provisions for a load that a given number of workers can sustain without adapting to any changes due to workload unpredictability, as discussed in the results for Figure 7. Finally, SHEPHERD observes both consistently high goodput and linear scaling. The linear scaling is attributed to SHEPHERD dividing its workers into groups, each with a group-worker count of 12, which is below the scalability limit of our online scheduler. The high goodput, on the other hand, is attributed to each group being large enough for efficient multiplexing across request streams. As such, SHEPHERD outperforms Clockwork and Nexus by 2.5× and 1.8× respectively in terms of goodput at $N = 400$ workers. We note, however, that SHEPHERD employs multiple schedulers — specifically, $\lceil \frac{N}{12} \rceil$ schedulers for N workers — in contrast to Clockwork's single centralized scheduler to achieve its linear scaling. We observe similar trends with the MAF-bursty workload in Figure 10(b).

7.2 Understanding SHEPHERD Benefits

We now dig deeper into how each design component in SHEPHERD contributes to its overall performance gains.

Benefits of FLEX (Figure 11(a)). To demonstrate the effec-

[3]This trend is consistent with the scalability results reported in the Clockwork paper [10] albeit with a higher peak goodput due to differences in the system implementation and execution runtime.

tiveness of batch prioritization and preemption in FLEX, we create a synthetic workload with two streams. Stream A is bursty and issues requests to the low-latency model ResNet18 ($\sim$ 4ms for batch size $= 1$, $\sim$ 32ms for batch size $= 128$). Requests in stream A arrive periodically in bursts of 1024 requests at $t = 5$ms, 125ms, 245ms, ..., i.e., with a period of 120ms. Stream B is stable and issues requests to the high-latency model ResNet269 (79ms for batch size $= 1$), and has individual requests arriving at $t = 0$ms, 1ms, 2ms ...; note that in the absence of other queued requests from stream B, any approach would schedule batches of size 1 for it every 1ms.

We provision one GPU worker for both streams, and compare the performance for SHEPHERD and Clockwork for this workload. To decouple the contributions of preemption from prioritization, we also evaluate a non-preemptive variant of SHEPHERD that retains all the properties of FLEX except preemption. We run the experiments under two different SLOs (250ms and 90ms) to separate the contributions of prioritization and preemption in SHEPHERD, as described next.

For 250ms SLO, both SHEPHERD and non-preemptive SHEPHERD outperform Clockwork by 3.7×. Since Clockwork prioritizes requests with tighter deadlines, it always ends up prioritizing high-latency requests of stream B over low-latency requests of stream A. In contrast, SHEPHERD's batch generation prioritizes larger batches — since stream A's low-latency requests can accumulate much larger batches under the 250ms SLO (*e.g.*, 128 sized batches with 32ms latency) and achieve much higher goodput. Prioritizing stream A's requests allows SHEPHERD to leverage the limited GPU resource to complete more requests in the same time span. In more detail, after a batch of stream B (comprising a single request) scheduled at time $t = 1ms$ completes after 79ms, SHEPHERD prioritizes stream A's queued requests over the remaining requests of stream B. With an SLO of 250ms, most requests in stream A can meet their SLO deadlines, permitting SHEPHERD to achieve high goodput even without preemption.

However, with a reduced SLO of 90ms, non-preemptive SHEPHERD cannot complete executing larger batches of stream A's requests within their SLO deadline since it *waits* for stream B's batch to finish (i.e., at $t = 80$ms). Thus, the performance for non-preemptive SHEPHERD is similar to Clockwork — most of stream A's requests fail to meet their deadline. With preemption, a large batch of stream A's requests preempts the scheduled (much smaller) batch of stream B's requests, allowing most requests of stream A to finish within the deadline. As such, SHEPHERD outperforms both its non-preemptive variant and Clockwork by 6.2×. Note that the performance for heuristic-driven and non-preemptive approaches can be made arbitrarily worse than SHEPHERD by increasing stream A's burst size and reducing its request execution latency, as discussed in §2.2 and Theorem 5.1, respectively.

Benefits of HERD (Figure 11(b)). We use the same setting as the large-scale emulation in §7.1 and vary the number of group-worker limit $\overline{G}$ for HERD (§4). With a group-worker limit of $\overline{G} = \infty$, SHEPHERD always chooses a group size equal to the number of workers. As such, it reduces to the online global approach, observing the same scalability limit as Clockwork (Figure 10). With a group-worker limit of $\overline{G} = 1$, SHEPHERD cannot efficiently multiplex across streams, leading to constantly lower goodput compared to SHEPHERD with multiple workers. As such, HERD allows SHEPHERD to achieve a goodput that is 2.5× and 1.7× higher than the two grouping alternatives, respectively, at 400 workers.

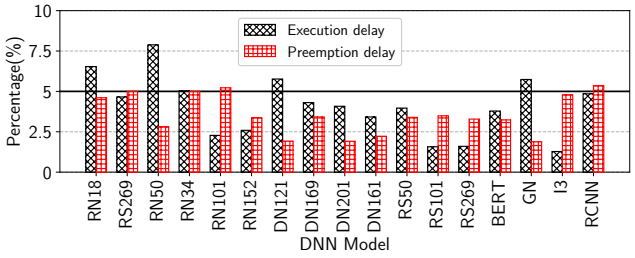


Figure 12: **Preemption overheads.** The preemption delay and additional execution delay relative to the normal batch executions for most of our evaluated models remains below 5%.

# streams	# models	# workers	solver	network	loading
200,000	200	200	0.55	0.19	0.71
400,000	400	400	2.51	0.35	1.23
600,000	600	600	4.28	0.51	1.84
800,000	800	800	8.53	0.62	2.41
1,000,000	1,000	1,000	13.26	0.90	3.14

Table 5: Components of the periodic planning latency (in seconds).

7.3 Understanding SHEPHERD Overheads

Finally, we evaluate the preemption and periodic planning overheads in SHEPHERD to show that neither impact SHEPHERD's performance benefits in any significant manner.

Preemption overheads (Figure 12). As discussed in §6, efficient preemption should minimize two overheads: (1) *preemption delay*, or the time between from the preemption being requested and actually being completed, and (2) *execution delay*, the additional latency introduced by exit points for normal batch execution. We achieve a reasonable trade-off between these two overheads by specifically tailoring appropriate number of exit points for each model listed in Table 4.

We measure the *relative* preemption overheads introduced by SHEPHERD, i.e., the preemption and execution delay relative to normal batch execution time, averaged over batch sizes 1–128. For most models, both the preemption delay and the extra execution delay are well below 5%.

Periodic planning overheads in HERD (Table 5). The periodic planning latency in HERD consists of three parts: (1) the solver latency for solving the ILP (Eq. 2), (2) the network latency for broadcasting the plan to schedulers and workers, and, (3) the loading latency for loading the models from CPU memory to GPU on each worker. We run large-scale emula-

tion to divide the system into 10 serving groups, and measure these latencies with different number of streams, models, and emulated workers. The solver latency accounts for most of the planning time, taking 13.26 seconds for 1 million streams and $1k$ workers. Network latency is always less than a second, while model loading time increases with the number of models. Even so, the total planning latency is always much smaller than HERD's scheduling period, which is tens of minutes. Moreover, the solver and network latency for the next planning phase can be pipelined with the current online serving phase, ensuring that planning is never a bottleneck.

8 Discussion and Caveat

We now outline avenues of future research in SHEPHERD.

Group predictablility under different workloads. Although we have demonstrated group predictability using two representative workloads, we note that the number of streams (i.e., group size) to achieve sufficient group predictability may be different for real-world workloads. With insufficient predictability, HERD may under or overprovision resources for some groups, although FLEX would still provide the same performance guarrantee within each group since it does not rely on predictability. Moreover, group predictability itself is rooted in statistical multiplexing theory, and holds when a large enough number of request streams in the workload have statistical independence [33–35]. While well-exploited in the networking community to achieve high utilization under bursty network traffic patterns [15–18], more in-depth quantitive analysis of group predictability for real-world inference serving workloads is important future work.

Model affinity *vs.* degree of multiplexing. Recall from §4 that HERD includes an affinity-set surjectivty constraint, which requires that models assigned to the same group j have divergence less than $\overline{K}$. With a small $\overline{K}$, HERD will break models into more groups, with each group containing fewer but more similar models, i.e., models with similar model affinity values. While this enables tighter performance guarantees in FLEX, it also reduces the degree of multiplexing within each group, since GPU workers in each group can serve streams across a smaller set of models. Although a single affinity group (i.e., $\overline{K}{=}\infty$) yields a looser competitive ratio, our evaluation shows that it still results in high empirical performance for the MAF workload. Finding an optimal value of $\overline{K}$ is promising future work.

Fairness across request streams. Similar to prior serving system designs [6, 10], we focus on the isolated GPU cluster settings where fairness across request streams and models is not a major concern. Fairness can be an important metric to extend our design to multi-user or cloud scenarios.

Dynamic model swapping. Similar to prior work [6], SHEPHERD only loads models onto GPU memory at the start of a planning period. An alternative solution is to dynamically swap models between GPU and CPU memory on-demand during online serving [10]. However, since such swaps are likely to take much longer than serving a request, its cost must be weighed against the potential performance gains from swapping in a new model. We leave incorporating this decision as a part of online serving as future work.

Large DNN models. If a DNN model is so large that it cannot be co-located with other models in GPU memory, HERD must place it in an isolated group with reduced degree of multiplexing. It is possible, however, to break such large models into smaller partitions [36] to group them with other models for better multiplexing.

9 Related Work

We discussed existing model serving systems in §2; we now discuss prior work related to SHEPHERD in other areas.

Preemption for ML workloads. PipeSwitch [21] allows preempting a training tasks to execute an inference task. Irina [11] applies preemption to improve average latency for inference tasks. LazyBatching [8] is an inference system that can preempt and stall the currently ongoing batch. SHEPHERD leverages preemption approaches outlined in these works to achieve guaranteed high goodput. Concurrent to our work, REEF [31] leverages ISA support for preemptions [37, 38] in recent AMD GPUs to enable μs-scale preemptions. While our current implementation still implements preemptions in software, it can readily incorporate hardware-based preemptions. Future improvements in this space will only improve SHEPHERD's performance further.

Online job scheduling. The theory community has long considered issues of prioritization and preemption in online job scheduling [19, 20, 24–27]. Its adaptation to model serving, however, has a few nuances — the scheduler for model serving must also decide how to optimally execute requests across batches while taking into account model-specific batch-latency relationships. Our scheduling algorithm exploits both to achieve strong performance guarantees.

10 Conclusion

We have presented SHEPHERD, a distributed a DNN model serving system. SHEPHERD employs a periodic planner that aggregates request streams into moderately-sized groups for high utilization and scalability, and an online scheduler that employs a novel online algorithm to provide guaranteed goodput. Evaluation over production workloads shows that SHEPHERD achieves 17.2× higher goodput and 1.8× higher utilization than prior approaches and scales to hundreds of workers.

Acknowledgement

We thank our shepherd Ravi Netravali and the anonymous NSDI reviewers for their insightful feedback. This research is supported by NSF Awards 2112562, 2047220, 1730628 and gifts from AWS, Ant Group, Ericsson, Futurewei, Google, Intel, Meta, Microsoft, NetApp, Scotiabank, and VMware.

References

[1] Kim Hazelwood, Sarah Bird, David Brooks, Soumith Chintala, Utku Diril, Dmytro Dzhulgakov, Mohamed Fawzy, Bill Jia, Yangqing Jia, Aditya Kalro, et al. Applied machine learning at facebook: A datacenter infrastructure perspective. In *HPCA*, 2018.

[2] Carole-Jean Wu, David Brooks, Kevin Chen, Douglas Chen, Sy Choudhury, Marat Dukhan, Kim Hazelwood, Eldad Isaac, Yangqing Jia, Bill Jia, et al. Machine learning at facebook: Understanding inference at the edge. In *2019 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, pages 331–344. IEEE, 2019.

[3] Daniel Crankshaw, Gur-Eyal Sela, Xiangxi Mo, Corey Zumar, Ion Stoica, Joseph Gonzalez, and Alexey Tumanov. InferLine: latency-aware provisioning and scaling for prediction serving pipelines. In *SoCC*, pages 477–491, 2020.

[4] Francisco Romero, Qian Li, Neeraja J Yadwadkar, and Christos Kozyrakis. INFaaS: Automated Model-less Inference Serving. In *ATC*, pages 397–411, 2021.

[5] Arpan Gujarati, Sameh Elnikety, Yuxiong He, Kathryn S McKinley, and Björn B Brandenburg. Swayam: distributed autoscaling to meet slas of machine learning inference services with resource efficiency. In *Middleware*, pages 109–120, 2017.

[6] Haichen Shen, Lequn Chen, Yuchen Jin, Liangyu Zhao, Bingyu Kong, Matthai Philipose, Arvind Krishnamurthy, and Ravi Sundaram. Nexus: a gpu cluster engine for accelerating dnn-based video analysis. In *Proceedings of the 27th ACM Symposium on Operating Systems Principles*, pages 322–337, 2019.

[7] Ahsan Ali, Riccardo Pinciroli, Feng Yan, and Evgenia Smirni. Batch: machine learning inference serving on serverless platforms with adaptive batching. In *SC: International Conference for High Performance Computing, Networking, Storage and Analysis*, pages 1–15. IEEE, 2020.

[8] Yujeong Choi, Yunseong Kim, and Minsoo Rhu. Lazy Batching: An SLA-aware Batching System for Cloud Machine Learning Inference. In *2021 IEEE International Symposium on High-Performance Computer Architecture (HPCA)*, pages 493–506. IEEE, 2021.

[9] Chengliang Zhang, Minchen Yu, Wei Wang, and Feng Yan. Mark: Exploiting cloud services for cost-effective, slo-aware machine learning inference serving. In *ATC*, pages 1049–1062, 2019.

[10] Arpan Gujarati, Reza Karimi, Safya Alzayat, Wei Hao, Antoine Kaufmann, Ymir Vigfusson, and Jonathan Mace. Serving dnns like clockwork: Performance predictability from the bottom up. In *14th {USENIX} Symposium on Operating Systems Design and Implementation ({OSDI} 20)*, pages 443–462, 2020.

[11] Xiaorui Wu, Hong Xu, and Yi Wang. Irina: Accelerating DNN Inference with Efficient Online Scheduling. In *4th Asia-Pacific Workshop on Networking*, pages 36–43, 2020.

[12] Peter Mattson, Vijay Janapa Reddi, Christine Cheng, Cody Coleman, Greg Diamos, David Kanter, Paulius Micikevicius, David Patterson, Guenther Schmuelling, Hanlin Tang, et al. Mlperf: An industry standard benchmark suite for machine learning performance. *IEEE Micro*, 40(2):8–16, 2020.

[13] AWS. Deliver high performance ML inference with AWS Inferentia. https://d1.awsstatic.com/events/reinvent/2019/REPEAT_1_Deliver_high_performance_ML_inference_with_AWS_Inferentia_CMP324-R1.pdf., 2019.

[14] Mohammad Shahrad, Rodrigo Fonseca, Íñigo Goiri, Gohar Chaudhry, Paul Batum, Jason Cooke, Eduardo Laureano, Colby Tresness, Mark Russinovich, and Ricardo Bianchini. Serverless in the wild: Characterizing and optimizing the serverless workload at a large cloud provider. In *ATC*, pages 205–218, 2020.

[15] Ravi R Mazumdar. Performance modeling, stochastic networks, and statistical multiplexing. *Synthesis Lectures on Communication Networks*, 6(2):1–211, 2013.

[16] Basil Maglaris, Dimitris Anastassiou, Prodip Sen, Gunnar Karlsson, and John D Robbins. Performance models of statistical multiplexing in packet video communications. *IEEE transactions on communications*, 36(7):834–844, 1988.

[17] Kavitha Chandra. Statistical multiplexing. *Wiley Encyclopedia of Telecommunications*, 5:2420–2432, 2003.

[18] Hiroshi Saito, Masatoshi Kawarasaki, and Hiroshi Yamada. An analysis of statistical multiplexing in an atm transport network. *IEEE Journal on Selected Areas in Communications*, 9(3):359–367, 1991.

[19] S. Goldman, Jyoti Parwatikar, and S. Suri. Online scheduling with hard deadlines. *J. Algorithms*, 34:370–389, 2000.

[20] Richard J. Lipton and Andrew Tomkins. Online interval scheduling. In *In Proceedings of the Fifth Annual ACM-SIAM Symposium on Discrete Algorithms*, pages 302–311, 1994.

[21] Zhihao Bai, Zhen Zhang, Yibo Zhu, and Xin Jin. Pipeswitch: Fast pipelined context switching for deep learning applications. In *14th {USENIX} Symposium on Operating Systems Design and Implementation ({OSDI} 20)*, pages 499–514, 2020.

[22] Boris N Pshenichnyj. *The linearization method for constrained optimization*, volume 22. Springer Science and Business Media, 2012.

[23] Christos H Papadimitriou and Kenneth Steiglitz. *Combinatorial optimization: algorithms and complexity*. Courier Corporation, 1998.

[24] Ran Canetti and Sandy Irani. Bounding the power of preemption in randomized scheduling. *SIAM Journal on Computing*, 27(4):993–1015, 1998.

[25] Xujin Chen, Xiaodong Hu, Tie-Yan Liu, Weidong Ma, Tao Qin, Pingzhong Tang, Changjun Wang, and Bo Zheng. Efficient mechanism design for online scheduling. *Journal of Artificial Intelligence Research*, 56:429–461, 2016.

[26] Sanjoy Baruah, Gilad Koren, Decao Mao, Bhubaneswar Mishra, Arvind Raghunathan, Louis Rosier, Dennis Shasha, and Fuxing Wang. On the competitiveness of on-line real-time task scheduling. *Real-Time Systems*, 4(2):125–144, 1992.

[27] Sally A Goldman, Jyoti Parwatikar, and Subhash Suri. Online scheduling with hard deadlines. *Journal of Algorithms*, 34(2):370–389, 2000.

[28] Pytorch. https://pytorch.org/.

[29] Apache tvm: An end to end machine learning compiler framework for cpus, gpus and accelerators. https://tvm.apache.org/.

[30] Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. Bert: Pre-training of deep bidirectional transformers for language understanding. *arXiv preprint arXiv:1810.04805*, 2018.

[31] Mingcong Han, Hanze Zhang, Rong Chen, and Haibo Chen. Microsecond-scale preemption for concurrent {GPU-accelerated}{DNN} inferences. In *16th USENIX Symposium on Operating Systems Design and Implementation (OSDI 22)*, pages 539–558, 2022.

[32] Pytorch hub. https://pytorch.org/hub/.

[33] Kavitha Chandra. Statistical multiplexing. *Wiley Encyclopedia of Telecommunications*, 5:2420–2432, 2003.

[34] Basil Maglaris, Dimitris Anastassiou, Prodip Sen, Gunnar Karlsson, and John D Robbins. Performance models of statistical multiplexing in packet video communications. *IEEE transactions on communications*, 36(7):834–844, 1988.

[35] Ward Whitt. Tail probabilities with statistical multiplexing and effective bandwidths in multi-class queues. *Telecommunication Systems*, 2(1):71–107, 1993.

[36] Lianmin Zheng, Zhuohan Li, Hao Zhang, Yonghao Zhuang, Zhifeng Chen, Yanping Huang, Yida Wang, Yuanzhong Xu, Danyang Zhuo, Joseph E Gonzalez, et al. Alpa: Automating inter-and intra-operator parallelism for distributed deep learning. *arXiv preprint arXiv:2201.12023 (OSDI)*, 2022.

[37] GPUOpen. Amd gpu isa documentation. https://gpuopen.com/documentation/amd-isa-documentation, 2021.

[38] Nathan Otterness and James H Anderson. Amd gpus as an alternative to nvidia for supporting real-time workloads. In *32nd Euromicro Conference on Real-Time Systems (ECRTS 2020)*. Schloss Dagstuhl-Leibniz-Zentrum für Informatik, 2020.

[39] Richard Hamming. *Numerical methods for scientists and engineers*. Courier Corporation, 2012.

A Competitive Ratio without Preemption

Theorem A.1 *No non-preemptive, deterministic algorithm can achieve a bounded competitive ratio for online serving.*

Proof Consider a batch X_1 with $|X_1| = 1$, $\ell_{X_1} = 1$ and $d_{X_1} = 1$ that arrives at time $t = 0$. A deterministic non-preemptive algorithm $\mathcal{B}$ serves X_1 with probability $p = \{0, 1\}$. We consider two scenarios after the scheduling decision at $t = 0$: (A) no request arrives afterwards, and therefore, the optimal solution has a total value of 1, and algorithm $\mathcal{B}$ has a total value of p; (B) another batch X_2 arrives at time $t = \varepsilon$ with $|X_2| = \omega \to \infty$, $\ell_{X_2} = 1$ and $d_{X_2} = 1 + \varepsilon$: the optimal solution can achieve a total value of ω by ignoring X_1, and algorithm $\mathcal{B}$ has a total value of $p + (1-p) \cdot \omega$, since it is non-preemptive.

Note that the competitive ratio should be no less than the ratio between the optimal solution over algorithm $\mathcal{B}$ in either scenario. As such, by combining both cases we show the competitive ratio c of algorithm $\mathcal{B}$ should be no less than:

$$c \geq \max_{p=\{0,1\}} \left(\frac{1}{p}, \frac{\omega}{p + (1-p) \cdot \omega}\right) \to \infty \tag{5}$$

which completes the proof ■

B Competitive Ratio Analysis for FLEX

We define a *schedule* σ to be a sequence of batch executions (B, t_B), where t_B is the start time of batch B in the schedule σ. Note that since we allow preemption, some batches may get preempted and never complete; we use $\sigma^c \subset \sigma$ to denote the set of completed batches in σ and $\sigma^p \subset \sigma$ to denote the set of preempted batches. We say a schedule σ is *feasible* if (1) at any time, at most one batch $B \in \sigma$ is executing, and, (2) a request is completed (i.e., executed without being preempted) in at most one batch $B \in \sigma^c$. Let $v(\sigma) = \sum_{B \in \sigma} v(B, t)$ denote the aggregated value of all batches in σ. We have,

$$v(\sigma) = v(\sigma^c) + v(\sigma^p) \tag{6}$$

We use standard competitive analysis to evaluate our algorithm. We denote the schedule due to an algorithm $\mathcal{A}$ as $\sigma_{\mathcal{A}}$, and the optimal schedule constructed by a computationally unbounded offline algorithm as σ_*. We say that algorithm $\mathcal{A}$ is *c-competitive* if for any request stream we have:

$$c \cdot v(\sigma^c_{\mathcal{A}}) \geq v(\sigma_*) \tag{7}$$

To better differentiate batch sequences (B, t) between schedule $\sigma_{\mathcal{A}}$ and σ_*, we denote the batch sequences in $\sigma_{\mathcal{A}}$ as (I, t_I) and batch sequences in σ_* as (J, t_J). Moreover, for a batch $I \in \sigma_{\mathcal{A}}$, we denote its (1) start time as t_I; (2) value (batch size) as $|I|$; and (3) duration as ℓ_I. The same notation rules apply to $J \in \sigma_*$.

We prove our main result in Theorem 5.2 in three steps. First, we consider a simplification of the online batch scheduling algorithm that only considers online batch scheduling for a single model running on a single GPU (§B.1). We then extend the setting to include multiple models deployed on a single GPU (§B.2), and finally consider the general case of multiple models deployed across multiple GPUs (§B.3).

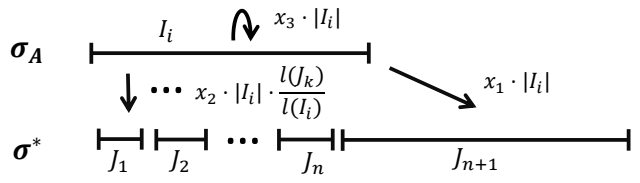
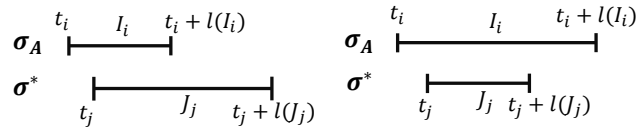


Figure 13: **The value assignment rule from one $I \in \sigma_{\mathcal{A}}$ to Js $\in \sigma_*$.**



(a) $J_j \in \sigma^*$ **blocked** by $I_i \in \sigma_A$ **(b)** $J_j \in \sigma^*$ **covered** by $I_i \in \sigma_A$

Figure 14: **The block and cover relationship between batches in $\sigma_{\mathcal{A}}$ and σ_*.** Note that if J is identical to I then J is covered by I (by our definition of blocking).

B.1 Single-GPU Single-Model Setting (sgsm)

For the single GPU, single model setting our key result is:

Theorem B.1 *Algorithm 1 is* 10.81*-competitive with a single model on a single GPU with preemption threshold* $\lambda \approx 2.38$.

Proof To prove the above theorem, we bound the value of batches in the optimal schedule (σ_*) by the value of completed batches in $\mathcal{A}$'s schedule ($\sigma^c_{\mathcal{A}}$). To this end, our analysis builds on the *value assignment approach* employed in [19, 20]. This approach operates in two steps:

1. Mapping. First, we *map* each batch in $\sigma_{\mathcal{A}}$ to a set of batches in σ_* in a manner that ensures each batch in σ_* is matched to at least one batch in $\sigma_{\mathcal{A}}$. This mapping identifies batches in σ_* that are related to batches in $\sigma_{\mathcal{A}}$, either because they overlap in their execution durations, or the batches contain common requests. More formally we define three relationships to compare a batch $J \in \sigma_*$ with a batch $I \in \sigma_{\mathcal{A}}$ (Figure 14):

M1. J is *blocked* by I if $t_I \leq t_J < t_I + \ell_I \leq t_J + \ell_J$.

M2. J is *covered* by I if $t_I < t_J$ and $t_I + \ell_I > t_J + \ell_J$.

M3. J is *intersected* by I if neither **R1** or **R2** hold, and $\exists r$ such that, $r \in I$ and $r \in J$.

We say J is temporally related to I if either **R1** or **R2** holds, and spatially related if **R3** holds.

2. Assignment. We assign values from each batch $I \in \sigma_{\mathcal{A}}$ to its mapped batches $J \in \sigma_*$, which we denote as $v_a(I, J)$. This assignment must satisfy two properties. First, it should be *feasible*, i.e., for any $I \in \sigma_{\mathcal{A}}$ its total assignment to all batches in σ_* should be equal to the value of I:

$$\sum_{J \in \sigma_*} v_a(I, J) = |I|, \quad \forall I \in \sigma_{\mathcal{A}} \tag{8}$$

Second, the assignment should be bounded, i.e., the total value assigned from all $I \in \sigma^c_{\mathcal{A}}$ to to all $J \in \sigma_*$ must be greater than or equal to a constant portion of the aggregated value of $J \in \sigma_*$:

$$\sum_{J \in \sigma_*} \sum_{I \in \sigma^c_{\mathcal{A}}} v_a(I,J) \geq r \cdot \sum_{J \in \sigma_*} |J| \tag{9}$$

where $r \in [0,1]$ is a constant. Note that for an assignment that is both feasible and bounded, we have:

$$\begin{aligned} v(\sigma^c_{\mathcal{A}}) &= \sum_{I \in \sigma^c_{\mathcal{A}}} |I| \\ &= \sum_{I \in \sigma^c_{\mathcal{A}}} \sum_{J \in \sigma_*} v_a(I,J) \\ &= \sum_{J \in \sigma_*} \sum_{I \in \sigma^c_{\mathcal{A}}} v_a(I,J) \\ &\geq \sum_{J \in \sigma_*} r \cdot |J| \\ &\geq r \cdot v(\sigma_*) \end{aligned} \tag{10}$$

Based on the definition of competitive ratio in Eq. 7, Eq. 10 suggests a competitive ratio of $c = \frac{1}{r}$.

The key tasks that remain are defining a feasible and bounded assignment v_a, and quantifying the bound r achieved by this assignment.

Defining the value assignment v_a: We are now ready to define our value assignment; as Figure 13 shows, a batch I in $\sigma_{\mathcal{A}}$ may cover n batches J_{c1} to J_{cn} (see **M2**), block at most one batch J_b (see **M1**) and intersect m batches J_{i1} to J_{im} (see **M3**). Our value assignment rules are as follows:

- **A1.** For a batch J_b that I blocks, $v_a(I,J_b) = x_1 \cdot |I|$.
- **A2.** For a batch J_c that I covers, $v_a(I,J_c) = x_2 \cdot |I| \cdot \frac{\ell_{J_c}}{\ell_I}$, i.e., the assigned value is proportional to the duration of J_c. Moreover, since the total duration of all covered batches J_{c1} to J_{cn} is no more than ℓ_I, the total assignment across J_{c1}, ..., J_{cn} is no more than $x_2 \cdot |I|$.
- **A3.** For a batch J_i that I intersects, we assign a value of x_3 to J_i for every request that is common between I and J_i, i.e., $v_a(I,J_i) = x_3 \cdot |I \cap J_i|$. Since each request will be executed at most once in σ_*, the total assigned value from I across all J_i is no larger than $x_3 \cdot |I|$.
- **A4.** If the total assigned value from I is less than $|I|$, we assign the residual value of I to any arbitrary $J \in \sigma_*$.

It is clear to see that the above assignment ensures that the total assignment from any batch $I \in \sigma_{\mathcal{A}}$ to all $J \in \sigma_*$ equals $|I|$, i.e., satisfies the feasibility constraint Eq. 8, if $(x_1 + x_2 + x_3) \leq 1$. Next, we quantify for each batch $J \in \sigma_*$, the lower-bound r to satisfy the boundedness constraint (Eq. 9).

3. Determining the bound r: A key challenge in determining the bound r for value $|J|$ relative to the value assigned to it, as per the boundedness constraint Eq. 9, is that a batch $I \in \sigma_{\mathcal{A}}$ and a batch $J \in \sigma_*$ can be related both temporally and spatially as outlined in our mapping step. As such, each such case requires analysis for the bound. As a concrete example, consider a batch J blocked by a batch I (as per **M1**). One possible reason J is not executed in $\sigma_{\mathcal{A}}$ is because a subset $J^E \subset J$ of requests may already have been dequeued from Q_m in $\sigma_{\mathcal{A}}$ and thus will not be executed again. Based on the dequeue condition in Algorithm 1, J^E is the subset of all requests in J that have already completed in $\sigma_{\mathcal{A}}$ at time t_J. On the other hand, it may be the case that J is not executed in $\sigma_{\mathcal{A}}$, because the value added by subset of requests $J^R \subset J$ that still remain to be executed (i.e., $J^R = J \setminus J^E$) is less than twice the value of the batch executed by $\sigma_{\mathcal{A}}$ in its place, namely I.

To accurately capture the impact of both of the above effects in determining the bound r, we define *virtual batches* J^R and J^E for each batch $J \in \sigma_*$ as above (see Figure 15). We denote the fraction of requests in J which belong to J^R as p, so that J^E contains the remaining $1-p$ fraction of requests. Note that p can take different values in $[0,1]$ for different J in σ_*. Since the value of a batch equals batch size, we have:

$$\begin{aligned} |J^R| &= p \cdot |J| \\ |J^E| &= (1-p) \cdot |J| \end{aligned} \tag{11}$$

Our next step is to determine the bound r based on J^R and J^E independently (r_R and r_E, respectively), and take the tighter of the two as our final lower bound, i.e., $r = max(r_R, r_E)$.

Determining r_R based on J^R: We first consider the lower-bound bound imposed on the value of J by only considering the virtual batches J^R. To this end, we confine ourselves to assignment rules **A1** and **A2** corresponding to blocked and covered batches, respectively.

- **Case 1: $I \in \sigma_{\mathcal{A}}$ blocks J^R.** Since J^R is blocked by I, it must be the case that $|J^R| \leq \lambda \cdot |I|$; otherwise J^R would have preempted I in $\sigma_{\mathcal{A}}$ at time t_J. Combined with the assignment rule **A1**, this gives us:

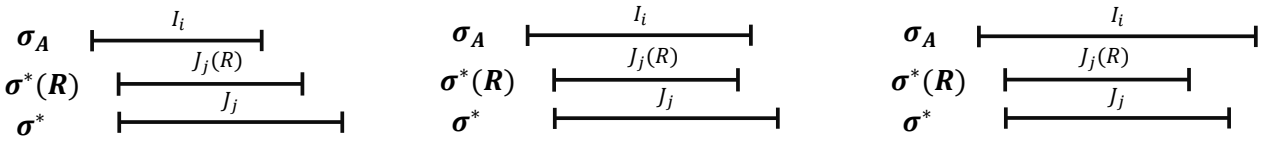
$$\begin{aligned} \sum_{I \in \sigma_{\mathcal{A}}} v_a(I,J) &\geq x_1 \cdot |I| \\ &\geq x_1 \cdot (\frac{1}{\lambda} \cdot |J^R|) \\ &= x_1 \cdot \frac{1}{\lambda} \cdot p \cdot |J| \end{aligned} \tag{12}$$

- **Case 2: $I \in \sigma_{\mathcal{A}}$ covers J^R.** To determine the lower bound in this case, we exploit two properties. First, since I covers J^R, $\ell_I > \ell_{J^R}$, i.e.,

$$\ell_I > \ell_{J^R} \tag{13}$$

Second, we exploit the property that a given model can always execute larger batches with smaller latency per record. Since I covers J^R,

$$\frac{|I|}{\ell_I} > \frac{|J^R|}{\ell_{J^R}} \tag{14}$$



(a) Case 1 (b) Case 2a (c) Case 2b

Figure 15: All possible conditions in $\sigma_{\mathcal{A}}$ for a batch $J \in \sigma_*$. Here schedule $\sigma_*(R)$ denotes the batch sequence of $(J(R), t_J)$.

Finally, as Figure 15(b)-(c) shows, this case can be further broken down into the following two sub-cases based on the relation between I and the real batch J:

– **Case 2a: I blocks J.** Assignment **A1** gives us:

$$\begin{aligned}\sum_{I\in\sigma_{\mathcal{A}}} v_a(I,J) &\geq x_1 \cdot |I| \\ &> x_1 \cdot \frac{|J^R|}{\ell_{J^R}} \cdot \ell_I \\ &= x_1 \cdot \frac{p\cdot|J|}{\ell_{J^R}} \cdot \ell_I \\ &> x_1 \cdot p \cdot |J|\end{aligned} \tag{15}$$

– **Case 2b: I covers J:** Assignment **A2** combined with Eqs. 13 and 14 applied to J gives us:

$$\begin{aligned}\sum_{I\in\sigma_{\mathcal{A}}} v_a(I,J) &\geq x_2 \cdot |I| \cdot \frac{\ell_J}{\ell_I} \\ &> x_2 \cdot \frac{|J|}{\ell_J} \cdot \ell_I \cdot \frac{\ell_J}{\ell_I} \\ &> x_2 \cdot |J|\end{aligned} \tag{16}$$

Note that in this case we do not need consider J^R to determine the bound since I directly covers J.

- **Case 3:** If neither of the above cases occur, then $\sigma_{\mathcal{A}}$ must be idle at time t_J. This implies that J^R must have been empty (i.e., $p = 0$), otherwise J^R would have been scheduled in $\sigma_{\mathcal{A}}$. Therefore, the following trivial bound holds:

$$\sum_{I\in\sigma_{\mathcal{A}}} v_a(I,J) \geq p\cdot|J| = 0 \tag{17}$$

Combining all the cases (Eq. 12, Eq. 15, Eq. 16 and Eq. 17), we have for any $J \in \sigma_*$:

$$\sum_{I\in\sigma_{\mathcal{A}}} v_a(I,J) \geq \min(\frac{p\cdot x_1}{\lambda}, x_2)\cdot|J| \tag{18}$$

Note that we omit the term from **Case 3**, since the corresponding inequality is dominated by $\frac{1}{2}\cdot x_1 \cdot p$ with $x_1 \leq 1$. Similarly, the term from **Case 2a** is also omitted since it is dominated by Case 1.

Aggregating both sides of Eq. 18 over all $J \in \sigma_*$, we get:

$$\begin{aligned}\min(\frac{p\cdot x_1}{\lambda}, x_2)\cdot\sum_{J\in\sigma_*}|J| &\leq \sum_{J\in\sigma_*}\sum_{I\in\sigma_{\mathcal{A}}} v_a(I,J) \\ &= \sum_{I\in\sigma_{\mathcal{A}}}\sum_{J\in\sigma_*} v_a(I,J) \\ &= \sum_{I\in\sigma_{\mathcal{A}}}|I| \\ &= v(\sigma_{\mathcal{A}})\end{aligned} \tag{19}$$

Next, we show how we can upper-bound $v(\sigma_{\mathcal{A}})$ by $v(\sigma^c_{\mathcal{A}})$. Note that batches in $\sigma_{\mathcal{A}}$ can form a chain based on the preemption relation. For each chain, the next batch on the chain preempts the previous one, and each chain must ended with a batch in $\sigma^c_{\mathcal{A}}$. We denote the chain which ends with batch $|I|$ as $chain(I)$. Denote $v(chain(I))$ as the value of all the batches in $chain(I)$, since each batch in $\sigma_{\mathcal{A}}$ will be covered by exactly one chain, we have

$$\sum_{I\in\sigma^c_{\mathcal{A}}} v(chain(I)) = \sum_{I\in\sigma_{\mathcal{A}}}|I| = v(\sigma_{\mathcal{A}}) \tag{20}$$

Moreover, based on the preemption rule we have that for each chain, the value of the ith batch in the chain must be no less than $\lambda\times$ the value of the $i-1$th batch. As such, $v(chain(I))$ must be no higher than $\frac{\lambda}{\lambda-1}\times|I|$, which indicates that:

$$\begin{aligned}v(\sigma^c_{\mathcal{A}}) &= \sum_{I\in\sigma^c_{\mathcal{A}}}|I| \\ &\geq \sum_{I\in\sigma^c_{\mathcal{A}}}\frac{\lambda-1}{\lambda}\cdot v(chain(I)) \\ &= \frac{\lambda-1}{\lambda}\cdot\sum_{I\in\sigma_{\mathcal{A}}}|I| \\ &= \frac{\lambda-1}{\lambda}\cdot v(\sigma_{\mathcal{A}})\end{aligned} \tag{21}$$

Combined with Eq. 6, we have,

$$\begin{aligned}v(\sigma_{\mathcal{A}}) &\leq \frac{\lambda}{\lambda-1}\cdot v(\sigma^c_{\mathcal{A}}) \\ &= \frac{\lambda}{\lambda-1}\cdot\sum_{I\in\sigma^c_{\mathcal{A}}}|I| \\ &= \frac{\lambda}{\lambda-1}\cdot\sum_{J\in\sigma_*}\sum_{I\in\sigma_{\mathcal{A}}} v_a(I,J)\end{aligned} \tag{22}$$

Combining Eqs. 19 and 22, we get:

$$\sum_{J\in\sigma_*}\sum_{I\in\sigma_{\mathcal{A}}} v_a(I,J) \geq \frac{\lambda-1}{\lambda}\cdot\min(\frac{p\cdot x_1}{\lambda},x_2)\cdot\sum_{J\in\sigma_*}|J| \quad (23)$$

This gives us a bound $r_R = \frac{\lambda-1}{\lambda}\cdot\min(\frac{p\cdot x_1}{\lambda},x_2)$.

Determining r_E based on J^E: Note that all requests in J^E must have been completed in $\sigma_{\mathcal{A}}$. So based on our assignment rule **A3**, any request in J^E must have been assigned a value of x_3 from one *completed* batch from $\sigma^C_{\mathcal{A}}$ (the set of completed batches in $\sigma_{\mathcal{A}}$). The above observation permits bounding $|J|$ based on J^E as follows:

$$\begin{aligned}\sum_{I\in\sigma^c_{\mathcal{A}}} v_a(I,J) &\geq \sum_{I\in\sigma^c_{\mathcal{A}}} x_3\cdot|I\cap J| \\ &\geq x_3\cdot|J^E| \\ &= (1-p)\cdot x_3\cdot|J|\end{aligned} \quad (24)$$

Aggregating both sides of Eq. 24 over all $J\in\sigma_*$, we get $r_E = (1-p)\cdot x_3$.

4. Determining the optimal competitive ratio: Combining the bounds r_R and r_E, we get the final competitive ratio:

$$\begin{aligned}c_{sgsm} &= \frac{1}{\max(r_R,r_E)} \\ &= \frac{1}{\min\limits_{p\in[0,1]}\{\max\{\frac{\lambda-1}{\lambda}\cdot\min\{\frac{p\cdot x_1}{\lambda},x_2\},(1-p)\cdot x_3\}\}}\end{aligned} \quad (25)$$

To minimize the value of c_{sgsm}, we can select appropriate values of x_1, x_2 and x_3 subject to the feasibility constraint $x_1+x_2+x_3\leq 1$, and select appropriate value of $\lambda\in(1,\infty)$. Note, however, that we cannot select p — it can take arbitrary values in $[0,1]$; as such, we have to consider the worst-case value for p to compute the competitive ratio. This provides us the following optimization problem:

$$\begin{aligned}&\min_{x_1,x_2,x_3,\lambda} c_{sgsm} \\ &\texttt{s.t. } x_1+x_2+x_3\leq 1 \\ &1<\lambda<\infty\end{aligned} \quad (26)$$

Solving the above optimization problem (via numerical methods [39]) yields the minimal value for $c_{sgsm} =\sim 10.81$ with preemption threshold $\lambda\approx 2.38$. ■

B.2 Single-GPU Multi-Model Setting (sgmm)

We now extend our analysis to the setting with k models $\{m_1,...,m_k\}$ deployed on a single GPU. The competitive analysis for the multi-model case also leverages the linear relationship between batch size and batch execution latency: for model m_i, the execution latency for a batch B is $\alpha_i\cdot|B|+\beta_i$, where α_i and β_i are model-specific constants

Theorem B.2 *Algorithm 1 is $10.81\cdot K$-competitive with multiple models on a single GPU, with preemption threshold $\lambda\approx 2.38$ and K defined in Eq. 4*

Proof The proof shares a similar structure with the single-GPU, single-model case, and is identical until we determine r_R based on J^R. Even so, the analysis for **Case 1** is still the same, since the preemption rule remains unchanged. For **Case 2**, however, Eq. 14 no longer holds, since with multiple models, a batch with larger length may have a smaller value density than a batch for a different model. However, with Eq. 1 we can replace Eq. 14 with:

$$K\cdot\frac{|I|}{\ell_I} > \frac{|J^R|}{\ell_{J^R}} \quad (27)$$

Now we show why Eq. 27 always holds. Assume I and J are batches for models m_1 and m_2 respectively. We have

$$\begin{aligned}\frac{|J^R|}{\ell_{J^R}}\cdot\frac{\ell_I}{|I|} &= \frac{|J^R|}{\alpha_2\cdot|J^R|+\beta_2}\cdot(\alpha_1+\frac{\beta_1}{|I|}) \\ &\leq \frac{|J^R|}{\alpha_2\cdot|J^R|+\beta_2}\cdot(\alpha_1+\beta_1) \\ &= \frac{|J^R|\cdot(\alpha_1+\beta_1)}{\alpha_2\cdot|J^R|+\beta_2} \\ &\leq K\end{aligned} \quad (28)$$

Note that Line 1 to Line 2 is based on the implicit constraint that $|I|\geq 1$ since it can only take integer values.

To further improve the bound, we notice that as $\ell_I > \ell_{J^R}$ always holds in Case 2, with Eq. 1 we have

$$\begin{aligned}&\alpha_1\cdot|I|+\beta_1 > \alpha_2\cdot|J^R|+\beta_2 \\ \rightarrow\ &|I| > \frac{\alpha_2\cdot|J^R|+\beta_2-\beta_1}{\alpha_1}\end{aligned} \quad (29)$$

On one hand, if $\alpha_2+\beta_2-\beta_1>0$, we have $\alpha_2\cdot|J^R|+\beta_2-\beta_1>0$. Then we can replace I in the first line of Eq. 28 with Eq. 29:

$$\begin{aligned}&\frac{|J^R|}{\ell_{J^R}}\cdot\frac{\ell_I}{|I|} = \frac{|J^R|}{\alpha_2\cdot|J^R|+\beta_2}\cdot(\alpha_1+\frac{\beta_1}{|I|}) \\ &< \frac{|J^R|}{\alpha_2\cdot|J^R|+\beta_2}\cdot(\alpha_1+\frac{\beta_1\cdot\alpha_1}{\alpha_2\cdot|J^R|+\beta_2-\beta_1}) \\ &= \frac{|J^R|}{\alpha_2\cdot|J^R|+\beta_2}\cdot(\frac{\alpha_1\cdot(\alpha_2\cdot|J^R|+\beta_2-\beta_1)+\beta_1\cdot\alpha_1}{\alpha_2\cdot|J^R|+\beta_2-\beta_1}) \\ &= \frac{|J^R|}{\alpha_2\cdot|J^R|+\beta_2}\cdot(\frac{\alpha_1\cdot\alpha_2\cdot|J^R|+\alpha_1\cdot\beta_2}{\alpha_2\cdot|J^R|+\beta_2-\beta_1}) \\ &= \frac{|J^R|}{\alpha_2\cdot|J^R|+\beta_2}\cdot(\frac{\alpha_1\cdot(\alpha_2\cdot|J^R|+\beta_2)}{\alpha_2\cdot|J^R|+\beta_2-\beta_1}) \\ &= \frac{\alpha_1\cdot|J^R|}{\alpha_2\cdot|J^R|+\beta_2-\beta_1} \\ &\leq K\end{aligned} \quad (30)$$

Then for Case 2a we will have:

$$\begin{aligned}\sum_{I\in\sigma_A} v_a(I,J) &\geq x_1\cdot|I| \\ &> \frac{x_1\cdot p\cdot|J|}{K}\end{aligned} \quad (31)$$

For Case 2b we have:

$$\sum_{I\in\sigma_A} v_a(I,J) \geq x_2\cdot|I|\cdot\frac{\ell_J}{\ell_I} > \frac{x_2\cdot|J|}{K} \tag{32}$$

The analysis for Case 3 and J^E is the same as the single model case. Similar to Eq. 25, by combining all cases we can calculate the competitive ratio c_{sgmm} for the multi-model case:

$$c_{sgmm} = \frac{1}{\min\limits_{p\in[0,1]}(\max(\frac{\lambda-1}{\lambda}\cdot\min(\frac{p\cdot x_1}{\lambda},\frac{p\cdot x_1}{K},\frac{x_2}{K}),(1-p)\cdot x_3))} \tag{33}$$

Since $K\geq 1$, combining Eqs. 33 and 25 gives us:

$$c_{sgmm}\leq K\cdot c_{sgsm} \quad \forall x_1,x_2,x_3,p,\lambda \tag{34}$$

As such, Algorithm 1 can always achieve a competitive ratio of $10.81\cdot K$ for the single-GPU, multi-model setting. ■

B.3 Multi-GPU Multi-Model Setting (mgmm)

Finally, we extend our analysis to the general case with k models $\{m_1,...,m_k\}$ and N GPUs. The major difference lies in the per-GPU preemption rule for the request arrival event — the new preemption rule ensures that at any time, no available batch will have a value $\lambda\times$ higher than the value of the currently running batches on *any* GPU. Moreover, the modified dequeue rule ensures that in the multi-GPU case, a request is completed in *at most one* batch in $\sigma_{\mathcal{A}}$.

We have the following theorem for the general case.

Theorem B.3 *For the multi-GPU, multi-model case, Algorithm 1 is* $12.62\cdot K$*-competitive with preemption threshold* $\lambda\approx 3.03$*, with K defined in Eq. 4.*

Proof The proof follows the same structure as the single-GPU, single-model setting as well. Define the schedule $\sigma_{\mathcal{A}}(u)$ as the schedule of Algorithm $\mathcal{A}$ on GPU $u\in[1,N]$ and $\sigma_*(v)$ as the optimal schedule on GPU $v\in[1,N]$. We have $\sigma_{\mathcal{A}}=\bigcup_u\sigma_{\mathcal{A}}(u)$ and $\sigma_*=\bigcup_v\sigma_*(v)$. Moreover, we define (u,v) as a GPU pair between the schedule $\sigma_{\mathcal{A}}(u)$ and $\sigma_*(v)$.

Value assignment rule between GPU pair (u,v) We apply a similar value assignment rule in the basic case *for each GPU pair* (u,v) in the general case. The major difference lies in assignment rules **A1** and **A2**, where we evenly spread the value for I from each $\sigma_{\mathcal{A}}(u)$ to all $\sigma_*(v)$ with different v.

- **A1.** For a batch $J_b\in\sigma_*(v)$ that $I\in\sigma_{\mathcal{A}}(u)$ blocks, $v_a(I,J_b)=\frac{x_1}{N}\cdot|I|$.
- **A2.** For a batch $J_c\in\sigma_*(v)$ that $I\in\sigma_{\mathcal{A}}(u)$ covers, $v_a(I,J_c)=\frac{x_2}{N}\cdot|I|\cdot\frac{\ell_{J_c}}{\ell_I}$.
- **A3.** For a batch $J_i\in\sigma_*(v)$ that $I\in\sigma_{\mathcal{A}}(u)$ intersects, we assign a value of x_3 to J_i for every request that is common between I and J_i, i.e., $v_a(I,J_i)=x_3\cdot|I\cap J_i|$.
- **A4.** If the total assigned value from $I\in\sigma_{\mathcal{A}}(u)$ is less than $|I|$, we assign the residual value of I to a random $J\in\sigma_*(v)$.

Similar to the basic case, the above pair-wise assignment rule ensures the following property:

Feasibility: For any GPU $u\in[1,N]$, with $(x_1+x_2+x_3)\leq 1$, the total assignment from any batch $I\in\sigma_{\mathcal{A}}(u)$ to all J in all $\sigma_*(v)$ equals $|I|$. That is:

$$\sum_{v\in[1,N]}\sum_{J\in\sigma_*(v)} v_a(I,J)=|I|, \quad \forall I\in\sigma_{\mathcal{A}}(u) \tag{35}$$

Boundedness: Similar to the basic case (Eq. 9), the assignment should be bounded. Here we want to show that the total value assigned from all I in all $\sigma^c_{\mathcal{A}}(u)$ to all batches J in all $\sigma_*(v)$ must be greater than or equal to a constant portion of the aggregated value of J in all $\sigma_*(v)$. That is:

$$\sum_{v\in[1,N]}\sum_{J\in\sigma_*(v)}\sum_{u\in[1,N]}\sum_{I\in\sigma^c_{\mathcal{A}}(u)} v_a(I,J)\geq r\sum_{v\in[1,N]}\sum_{J\in\sigma_*(v)}\cdot|J| \tag{36}$$

where $r\in[0,1]$ is a constant. Note that similar to the basic case (Eq. 10), for an assignment that is both feasible and bounded, we have:

$$\begin{aligned} v(\sigma^c_{\mathcal{A}}) &= \sum_{u\in[1,N]}\sum_{I\in\sigma^c_{\mathcal{A}}(u)}|I| \\ &= \sum_{v\in[1,N]}\sum_{J\in\sigma_*(v)}\sum_{u\in[1,N]}\sum_{I\in\sigma^c_{\mathcal{A}}(u)} v_a(I,J) \\ &\geq \sum_{v\in[1,N]}\sum_{J\in\sigma_*(v)} r\cdot|J| \\ &\geq r\cdot v(\sigma_*) \end{aligned} \tag{37}$$

Eq. 37 suggests a competitive ratio of $c=\frac{1}{r}$.

Determining the bound r**:** The key task that remains is to quantify the bound r achieved by the assignment. Similar to the basic case, this is done by bounding the values of J for each $\sigma_*(v)$ by values of I for each $\sigma_{\mathcal{A}}(u)$, based on both the J^E and J^R parts.

Determining r_R *based on* J^R*:* We can apply the same analysis as in the basic case for each GPU pair (u,v). Note that for **Case 1**, the modified preemption rule ensures that at any time, no available batch in $\sigma_{\mathcal{A}}$ will have a value $\lambda\times$ higher than the value of the currently running batches on *any* GPU. As such, the J^R from any GPU u must have a value no higher than $\lambda\times$ the value of the I blocks it in any u, which indicates:

$$\begin{aligned} \sum_{I\in\sigma_{\mathcal{A}}(u)} v_a(I,J) &\geq \frac{x_1}{N}\cdot|I| \\ &\geq \frac{x_1}{N}\cdot(\frac{1}{\lambda}\cdot|J^R|) \\ &= \frac{x_1}{N}\cdot\frac{1}{\lambda}\cdot p\cdot|J| \end{aligned} \tag{38}$$

Moreover, the analysis for **Case 2** and **Case 3** follows the same logic. Formally, for any u and $J \in \sigma_*(v)$ we have:

$$\sum_{I \in \sigma_{\mathcal{A}}(u)} v_a(I,J) \geq \begin{cases} \frac{x_1 \cdot p}{\lambda} \cdot \frac{|J|}{N}, & \text{Case 1} \\ \frac{x_1 \cdot p \cdot |J|}{K \cdot N}, & \text{Case 2a} \\ \frac{x_2 \cdot |J|}{K \cdot N}, & \text{Case 2b} \\ p \cdot |J|, & \text{Case 3} \end{cases} \tag{39}$$

Since the above equation holds for each $\sigma_{\mathcal{A}}(u)$, we have:

$$\sum_{u \in [1,N]} \sum_{I \in \sigma_{\mathcal{A}}(u)} v_a(I,J) \geq \min(\frac{x_1 \cdot p}{\lambda}, \frac{x_1 \cdot p}{K}, \frac{x_2}{K}) \cdot |J| \tag{40}$$

Aggregating both sides of Eq. 39 over all J in all $\sigma_*(v)$, we get:

$$\begin{aligned} & \min(\frac{x_1 \cdot p}{\lambda}, \frac{x_1 \cdot p}{K}, \frac{x_2}{K}) \cdot \sum_{v \in [1,N]} \sum_{J \in \sigma_*(v)} |J| \\ \leq & \sum_{v \in [1,N]} \sum_{J \in \sigma_*(v)} \sum_{u \in [1,N]} \sum_{I \in \sigma_{\mathcal{A}}(u)} v_a(I,J) \\ = & \sum_{u \in [1,N]} \sum_{I \in \sigma_{\mathcal{A}}(u)} |I| \\ = & v(\sigma_{\mathcal{A}}) \end{aligned} \tag{41}$$

Next, we bound $v(\sigma_{\mathcal{A}})$ by $v(\sigma^c_{\mathcal{A}})$. We apply the same chain analysis as we did for the basic case for each $\sigma_{\mathcal{A}}(u)$. More specifically we have:

$$\sum_{I \in \sigma^c_{\mathcal{A}}(u)} v(chain(I)) = \sum_{I \in \sigma_{\mathcal{A}}(u)} |I| \qquad \forall u \in [1,N] \tag{42}$$

Then based on the preemption rule we have:

$$\begin{aligned} v(\sigma^c_{\mathcal{A}}) &= \sum_{u \in [1,N]} \sum_{I \in \sigma^c_{\mathcal{A}}(u)} |I| \\ &\geq \sum_{u \in [1,N]} \sum_{I \in \sigma^c_{\mathcal{A}}(u)} \frac{\lambda - 1}{\lambda} \cdot v(chain(I)) \\ &= \frac{\lambda - 1}{\lambda} \cdot \sum_{u \in [1,N]} \sum_{I \in \sigma_{\mathcal{A}}(u)} |I| \\ &= \frac{\lambda - 1}{\lambda} \cdot v(\sigma_{\mathcal{A}}) \end{aligned} \tag{43}$$

Combining Eqs. 41 and 43, we get:

$$\begin{aligned} & \sum_{v \in [1,N]} \sum_{J \in \sigma_*(v)} \sum_{u \in [1,N]} \sum_{I \in \sigma^c_{\mathcal{A}}(u)} v_a(I,J) \\ \geq & \frac{\lambda - 1}{\lambda} \min(\frac{x_1 \cdot p}{\lambda}, \frac{x_1 \cdot p}{K}, \frac{x_2}{K}) \sum_{v \in [1,N]} \sum_{J \in \sigma_*(v)} \cdot |J| \end{aligned} \tag{44}$$

This gives us a bound $r_R = \frac{\lambda - 1}{\lambda} \min(\frac{x_1 \cdot p}{\lambda}, \frac{x_1 \cdot p}{K}, \frac{x_2}{K})$.

Determining r_E based on J^E: Note that based on the dequeue and preemption rule in Algorithm 1, some request in J^E may not have been completed in $\sigma_{\mathcal{A}}$. Instead, it only ensures that for any $J \in \sigma_*(v)$, all requests in the corresponding J^E must *have been (or being) executed* in some $\sigma_{\mathcal{A}}(u)$. Since each of the requests in J^E gets assigned a value of x_3 (based on **A3**), we have the following bound:

$$\begin{aligned} \sum_{u \in [1,N]} \sum_{I \in \sigma_{\mathcal{A}}(u)} v_a(I,J) &\geq \sum_{u \in [1,N]} \sum_{I \in \sigma_{\mathcal{A}}(u)} |I \cap J_i| \\ &\geq x_3 \cdot |J^E| \\ &= (1-p) \cdot x_3 \cdot |J| \end{aligned} \tag{45}$$

Note that Eq. 45 is in the exact same form as Eq. 40. So following the same procedure from Eq. 41 to Eq. 44 we can get:

$$\begin{aligned} & \sum_{v \in [1,N]} \sum_{J \in \sigma_*(v)} \sum_{u \in [1,N]} \sum_{I \in \sigma^c_{\mathcal{A}}(u)} v_a(I,J) \\ \geq & \frac{\lambda - 1}{\lambda} (1-p) \cdot x_3 \sum_{v \in [1,N]} \sum_{J \in \sigma_*(v)} \cdot |J| \end{aligned} \tag{46}$$

This gives us a bound $r_E = \frac{\lambda - 1}{\lambda}(1-p) \cdot x_3$.

Determining the optimal competitive ratio c_{mgmm}: Combining the bounds r_R and r_E, we get the final competitive ratio:

$$\begin{aligned} c_{sgsm} &= \frac{1}{\max(r_R, r_E)} \\ &= \frac{1}{\frac{\lambda - 1}{\lambda} \cdot \min\limits_{p \in [0,1]} \{\max\{\min(\frac{p \cdot x_1}{\lambda}, \frac{p \cdot x_1}{K}, \frac{x_2}{K}), (1-p) \cdot x_3\}\}} \end{aligned} \tag{47}$$

Similar to the basic case, we can select appropriate values of x_1, x_2, x_3 and λ to minimize c_{mgmm}.

$$\begin{aligned} & \min_{x_1, x_2, x_3, \lambda} c_{mgmm} \\ & \texttt{s.t.} \;\; x_1 + x_2 + x_3 \leq 1 \\ & 1 < \lambda < \infty \end{aligned}$$

Solving the above optimization problem yields the maximal value for $c_{mgmm} =\sim 12.62 \times K$ with preemption threshold $\lambda \approx 3.03$. ■

C Complexity Analysis for FLEX

Theorem C.1 FLEX *has a worst-case complexity of O(G), where G is the number of GPUs in the serving group.*

Proof Batch generation (Algorithm 2) has a complexity of $O(|\mathbb{M}| \cdot |Q|)$ where $|\mathbb{M}|$ is the number of models queues with newly enqueued requests since last update, and $|Q|$ is the largest queue size among these model queues. For each request arrival event, batch generation is triggered $O(G)$ times. Moreover, between every two invocations, at most one model queue changes. Therefore, $|\mathbb{M}|$ is at most 2 for each invocation (Line 2 in Algorithm 2). In addition, since each preemption will increase the size for the running batch by at least $\lambda\times$, each GPU can only be preempted by at most $O(log_{\lambda}(|Q|))$ times.

Algorithm 2 Batch generation algorithm in FLEX

1: **procedure** BATCHGEN(n)
2: $\mathbb{M} \leftarrow$ models with newly enqueued requests or currently running on GPU n
3: **for** each model $m \in \mathbb{M}$ **do**
4: Dequeue requests passing their deadlines from $Q(m)$
Line 5-13: Find largest feasible batch $B_g(n,m)$ for model m
5: Candidate request set $\mathbb{S} \leftarrow Q(m)$
6: **if** $B_c(n)$ uses model m **then**
7: $\mathbb{S} \leftarrow Q(m) \bigcup B_c(n)$
8: $B_g(n,m) \leftarrow \emptyset$
9: **for** request r in $\mathbb{S}$ with ascending deadline **do**
10: **if** r can meet SLO with batch size $|B_g(n,m))|$ **then**
11: Add r to $B_g(n,m)$
12: **else**
13: Break
14: $B_g(n) \leftarrow B_g(n,m)$ with largest batch size among all models
15: Return $B_g(n)$

As such, the re-enqueue event (Line 19 and 11) will be triggered by at most $O(log_\lambda(|Q|))$ times for each GPU. The complexity of enqueue operation is $O(log(|Q|))$ (Line 10), and the complexity of re-enqueue operation is $O(|Q| + |B_c(n)|)$ (Line 19). Note that $|B_c(n)|$ can never be larger than $|Q|$ by definition. Note that $|Q|$ and λ are constants. Based on the above analysis, the total complexity for each request arrival event and batch completion event is $O(G)$. Similar analysis applies for each batch completion event. Taken together, FLEX has an overall complexity of $O(G)$. ■