PARD: Enhancing Goodput for Inference Pipeline via ProActive Request Dropping

Zhixin Zhao
Tianjin University
zhao612@tju.edu.cn

Yitao Hu*
Tianjin University
yitao@tju.edu.cn

Simin Chen
University of Texas at Dallas
simin.chen@utdallas.edu

Mingfang Ji
Tianjin University
jimingfang@tju.edu.cn

Wei Yang
University of Texas at Dallas
wei.yang@utdallas.edu

Yuhao Zhang
Tianjin University
yuhaozhang@tju.edu.cn

Laiping Zhao
Tianjin University
laiping@tju.edu.cn

Wenxin Li
Tianjin University
toliwenxin@tju.edu.cn

Xiulong Liu
Tianjin University
xiulong_liu@tju.edu.cn

Wenyu Qu
Tianjin University
wenyu.qu@tju.edu.cn

Hao Wang
Stevens Institute of Technology
hwang9@stevens.edu

Abstract

Modern deep neural network (DNN) and large language model (LLM) applications integrate multiple models into inference pipelines with stringent latency requirements for customized tasks. To mitigate extensive request timeouts caused by accumulation, systems for inference pipelines commonly drop a subset of requests so the remaining ones can satisfy latency constraints. Since it is commonly believed that request dropping adversely affects goodput, existing systems only drop requests when they have to, which we call *reactive* dropping. However, this reactive policy can *not* maintain high goodput, as it neither makes timely dropping decisions nor identifies the proper set of requests to drop, leading to issues of dropping requests too late or dropping the wrong set of requests.

We propose that the inference system should *proactively* drop certain requests in advance to enhance the goodput across the entire workload. To achieve this, we design an inference system PARD. It enhances goodput with timely and precise dropping decisions by integrating a proactive dropping method that decides when to drop requests using runtime information of the inference pipeline, and an adaptive request priority mechanism that selects which specific requests to drop based on remaining latency budgets and workload intensity. Evaluation on a cluster of 64 GPUs over real-world workloads shows that PARD achieves 16%–176% higher goodput than the state of the art while reducing the drop rate and wasted computation resources by 1.6×–17× and 1.5×–62× respectively.

CCS Concepts: • **Computer systems organization** → **Cloud computing**; • **Computing methodologies** → **Machine learning**.

Keywords: Machine learning systems; Inference serving; Request scheduling; Overload control

ACM Reference Format:
Zhixin Zhao, Yitao Hu, Simin Chen, Mingfang Ji, Wei Yang, Yuhao Zhang, Laiping Zhao, Wenxin Li, Xiulong Liu, Wenyu Qu, and Hao Wang. 2026. PARD: Enhancing Goodput for Inference Pipeline via ProActive Request Dropping. In *The 21st European Conference on Computer Systems (EuroSys '26), April 27–30, 2026, Edinburgh, UK*. ACM, New York, NY, USA, 16 pages. https://doi.org/10.1145/nnnnnnn.nnnnnnn

1 Introduction

Increasing task complexity and proliferation of open-source models have made DNN inference pipelines crucial and cost-efficient for emerging real-time applications [1–7]. These pipelines consist of multiple models, where requests pass through each model to complete customized tasks under a strict latency objective. For example, the avatar generation pipeline [1, 2] in Figure 1 transforms live video of a person into a virtual character for real-time interaction. Existing inference systems define goodput [8–10], which is the number of requests that can meet latency objective per unit of time, to measure how well the system provides services for latency-sensitive inference pipelines. To guarantee high goodput, existing systems primarily focus on techniques such as resource scaling [8, 11–13], dynamic batching [1, 2, 14, 15] and GPU scheduling [16–19].

*Corresponding author.

EuroSys '26, Edinburgh, UK.

ACM ISBN 979-8-4007-1196-1/2026/04
https://doi.org/10.1145/nnnnnnn.nnnnnnn

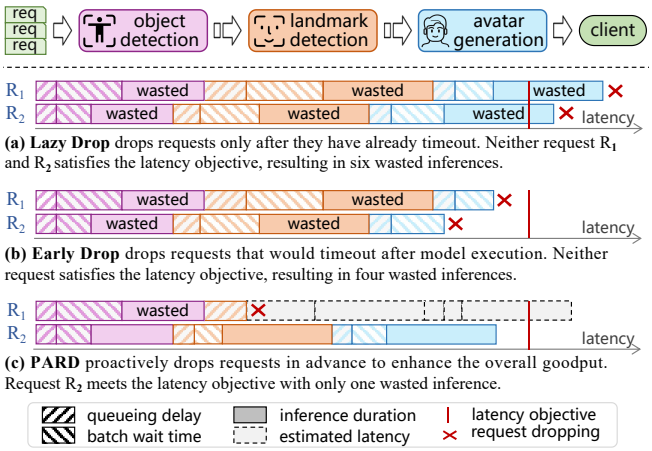


Figure 1. Request latency composition under various dropping policies in a three-model inference pipeline.

Despite these efforts, certain requests inevitably experience large latencies due to unpredictable events such as workload bursts [8, 10] or machine failure [20]. In real-time inference serving (*e.g.,* video analytics), requests that fail to meet their latency service-level objectives (SLOs) are useless [1, 5, 21]. Without careful handling, these requests prolong the latency of subsequent requests due to the queueing effect, eventually leading to latency violations and poor goodput. Therefore, existing systems employ request dropping policies to drop a portion of requests [1, 2, 4, 22–24], ensuring the remaining ones meet the latency objective and avoid goodput degradation.

However, existing dropping policies can *not* maintain high goodput since they share a reactive design. Since it is commonly believed that dropping requests adversely affects goodput, existing systems *drop requests only when they have to be dropped*, *i.e.,* when it has already timed out before model execution [22] (Figure 1a) or will be timed out after its execution [1] (Figure 1b). These reactive designs fail to make proper dropping decisions for two primary reasons. (1) Drop requests too late. When making decisions at a given module[1] in the pipeline, the reactive policy only considers the latency up to the current module. It ignores the latency budget for subsequent modules, leading to inopportune dropping decisions and wasted computation resources. (2) Drop the wrong set of requests. The reactive policy makes dropping decisions in request's arrival order. It ignores the impact of the difference in requests' remaining latency budget and the workload variation, which cannot determine the proper set of requests to drop for goodput enhancement.

In this paper, we argue that **inference system should proactively drop certain requests in advance to enhance the goodput for the entire workload**. However, three characteristics of the inference pipeline make this challenging. (1) *Model Cascading.* Requests traverse multiple cascaded models with only end-to-end latency objectives specified. While the latency budget of each model must be considered when making dropping decisions. (2) *Latency Uncertainty.* Batch wait time from the batching process [1] and queueing cause high uncertainty in request latency. This uncertainty is further amplified by model cascading of the pipeline. (3) *Workload Variation.* Inference workloads exhibit dynamic and bursty characteristics, demanding that dropping policy effectively adapts to varying workload intensity. Building on our argument, we design and implement an inference system called PARD, which adapts the characteristics and makes proactive dropping decisions to enhance the goodput of the inference pipeline as follows.

First, PARD designs a novel proactive request dropping method to address *when to drop requests* for each module in the pipeline. Existing reactive policies rely solely on latency information from preceding modules and overlook the latency budget requirements of subsequent modules, causing most drops to concentrate in the last few modules of the pipeline. In contrast, PARD proactively estimates request latency across the entire pipeline with bi-directional runtime information from the preceding and subsequent modules, enabling timely dropping decisions in the early stages of the pipeline, reducing wasted computation (Figure 1c).

Second, PARD introduces an adaptive request priority method to decide *which specific requests to drop* in each module. Existing reactive policies drop requests strictly by arrival order, ignoring the difference in their remaining latency budget. In contrast, PARD designs a hybrid priority mechanism that determines the decision-making order across requests based on each request's remaining latency budget and workload intensity. This mechanism ensures a smooth transition between dropping priorities, so as to determine a proper set of requests to drop under dynamic workloads, which avoids unnecessary request dropping.

When enhancing the goodput for the inference pipeline, PARD makes the following contributions:

- PARD highlights the critical role of request dropping in enhancing goodput and reveals why existing reactive design fails to maintain high goodput for inference pipeline through a systematic analysis.
- PARD designs a proactive dropping policy and adaptive request priority, which is orthogonal to existing scheduling methods like resource scaling and dynamic batching. It could further enhance the goodput of existing systems.
- Evaluation shows that PARD improves goodput by 16%–176% over state-of-the-art systems, while reducing the drop rate and wasted computation by 1.6×–17× and 1.5×–62× respectively. The ablation study (§5.3) and Retrieval-Augmented Generation (RAG) case study (§7) quantify the impact of PARD's design choices and its generalization to emerging LLM-powered generative workloads.

[1]Each module serves a specific model of the inference pipeline using the assigned computation resources.

2 Background

Real-time Inference Pipeline. A typical DNN inference pipeline aims to deliver real-time response to the clients, where any request missing its latency SLO is invalid. Existing systems adopt techniques such as dynamic batching, resource scaling and GPU scheduling [13, 15, 16, 25–27], to satisfy the latency objectives. However, even with such optimizations, it is impossible to guarantee that *all* requests can satisfy the latency SLO. Unforeseen events such as workload bursts or machine failures [8, 10, 20] pose challenges. For instance, during workload bursts, resources cannot scale up instantly due to model cold starts [8, 28, 29], causing request accumulation and latency violations.

Request Dropping Policies. To avoid the negative impact of timeout requests on goodput, recent systems adopt request dropping [1, 4, 9, 22, 28, 30, 31]. The idea is to drop requests that would miss the latency objective, especially when incoming workload exceeds system capacity. Request dropping aims to reduce the queueing delay for the remaining requests, so the remaining ones are more likely to achieve lower latency and meet the latency objective. This, in turn, enhances the overall goodput for the entire workload.

Dropping policies in existing systems fall into two types. When making dropping decisions at a given module in the pipeline, the first type drops requests that already exceed the latency objective [4, 22], while the second type considers both the accumulated latency in preceding modules and the inference duration of current module, and then drops requests that cannot complete within the latency objective [1, 28, 30]. Depending on specific policy, the system drops requests that have exceeded the latency objective or have a remaining latency budget less than inference duration before inference [1, 2, 4, 5, 22, 28]. Since these policies only drop requests when they have to be dropped, we define them as *reactive* policies. We argue that such reactive policies cannot make proper dropping decisions, leading to sub-optimal goodput, as we discuss next.

3 Motivation

This section reveals the limitations of existing dropping policies (§3.1) and provides in-depth analysis from two dimensions: timing of decision and criteria for selection (§3.2). Finally, we identify two key questions and corresponding challenges to design an optimal dropping policy (§3.3).

3.1 Limitation of Reactive Dropping Policy

To highlight the limitations of current inference systems for multi-DNN inference pipelines, we evaluate Nexus [1] and Clipper++, a modified version of Clipper [22] for inference pipeline workload, using the `lv-tweet` workload as described in §5.1. We compare them with a naive baseline that serves the inference pipeline without dropping any requests. The primary evaluation metric is *goodput* and *drop rate* (*i.e.,* the percentage of dropped requests to all requests).

We calculate the minimum goodput and corresponding drop rate over the entire runtime across various time window sizes in Figure 2a and Figure 2b for comparison. From the results, it is notable that the goodput of Nexus and Clipper++ may decrease to as low as 30% and 21% of the input request rate, with drop rates of 70% and 79%, which performs worse than the naive baseline at several window sizes.

3.2 Understand Why Reactive Policy Fails

Observation #1: Reactive policy drops requests too late in the inference pipeline. At first glance, request dropping seems to adversely affect goodput. So existing systems drop requests only when they have already violated or are about to violate the latency objective. This reactive approach can *not* make timely dropping decisions in the inference pipeline, as it only considers the accumulated latency for preceding and current modules, which drops requests too late.

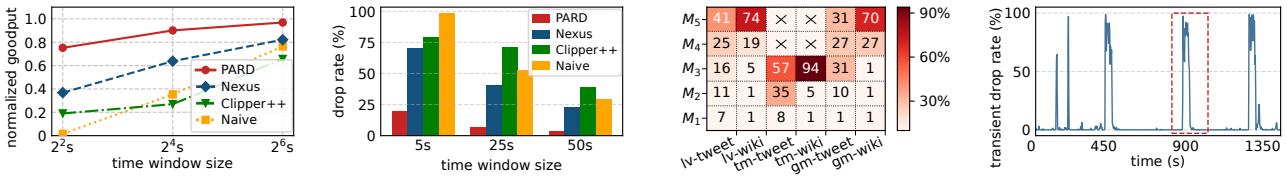
Figure 2c shows the percentage of dropped requests at each module for various workloads as described in §5.1 under reactive dropping policy, where 57.1% to 97.2% of dropped requests are dropped in the latter half of the pipeline. To understand why, we take the traffic monitoring application `tm` with 3 modules cascaded (the third and fourth column of Figure 2c) as an example, where 57.1% and 94.4% of dropped requests are dropped in the last module under two traces. As shown in Figure 1a and Figure 1b, if a request R_1 experiences long wait time (including queueing delay and batch wait time) in the early modules, it will over-consume the end-to-end latency budget. Existing reactive policy will *not* drop R_1 until after the last module's inference, when the remaining budget is already insufficient. Once dropped, the computation resources consumed by R_1 are wasted, causing the **drop-too-late** issue and raising the invalid rate[3]. These wasted computations bring backpressure for preceding modules, increasing queuing delay and ultimately lowering goodput.

Observation #2: Reactive policy drops the wrong set of requests. Existing systems maintain a FIFO request queue for each worker, making dropping decisions based on arrival order. This arrival-order-based method can *not* identify the proper set of requests to drop and over-consume latency budget at the early stages of the pipeline, leaving insufficient budget for later stages and ultimately reducing goodput.

Figure 2d shows the drop rate of the reactive dropping under the `lv-tweet` workload with five cascaded modules: the transient drop rate exceeds 95% around t=850s even though the input request rate only doubles at that moment (Figure 10e). Figure 3a explains why: within a window of

[2]Figure 2c and Figure 2d present the results of Clipper++; Nexus exhibits similar trends (§5.2).

[3]Invalid rate measures the ratio of GPU time consumed by dropped requests (*i.e.,* wasted computation resources), and it will be formally defined in §5.1.



(a) Goodput comparison **(b)** Drop rate comparison **(c)** Percent of dropped requests **(d)** Transient drop rate

Figure 2. (a) and (b) The minimum goodput and corresponding drop rate across various time window sizes of existing inference systems, naive baseline, and PARD under `lv-tweet` workload. (c) The percentage of dropped requests at each module under different workloads from §5.1 with the reactive dropping policy. (d) Transient drop rate of the reactive dropping policy[2].

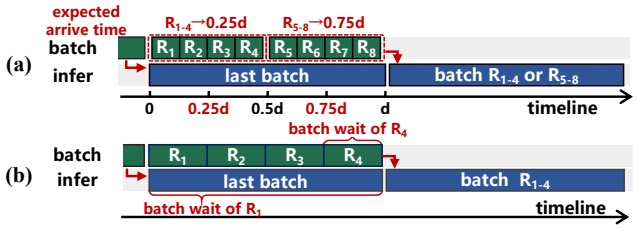


Figure 3. (a) The reactive dropping policy makes decisions based on request arrival order, leading to the drop-wrong-set issue. (b) Batched requests have different batch wait times W, ranging from 0 to the batch execution duration d.

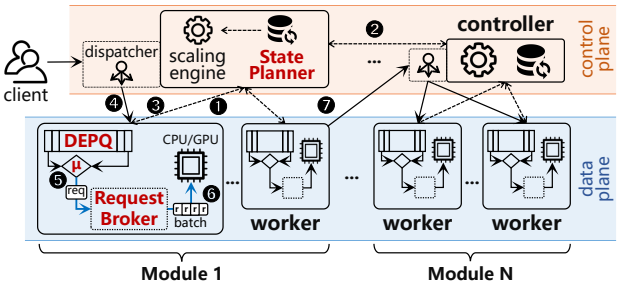


Figure 4. PARD overview

duration d (the inference time of one batch) with batch size 4, there are 8 arrivals but the system can serve 4 within the latency constraint, so exactly half must be dropped to prevent accumulation. The reactive policy serves requests in arrival order, keeping the earliest arrived $R_{1\to4}$ and dropping $R_{5\to8}$. In the depicted window, requests $R_{1\to4}$ fall in $[0, 0.5d]$ with a mean arrival time of $0.25d$, so their expected batch wait time to the next batch start is $d - 0.25d = 0.75d$, whereas the expected batch wait time of $R_{5\to8}$ is only $d - 0.75d = 0.25d$. Therefore, the reactive policy's FIFO request queue wrongly keeps $R_{1\to4}$ with a higher batch wait time, which over-consumes the request's latency budget. This **drop-wrong-set** issue eventually causes requests to violate latency objectives and increases the drop rate.

3.3 Implications

The experimental results highlight that applying a reactive dropping policy cannot yield high goodput. To overcome the limitations of existing inference systems and ensure high goodput for inference pipelines, PARD should adopt a *proactive* dropping policy, addressing two key questions: (1) *When to drop requests.* PARD should proactively identify which requests can not be completed within the latency objective, enabling timely dropping decisions. (2) *Which specific requests to drop.* PARD should proactively select the set of requests to be dropped that enhance goodput.

However, designing such a system presents several challenges: (1) *Estimating the end-to-end latency of each request:* Addressing the first question requires accurately estimating the latency of each request at each module. This is challenging due to the latency uncertainty caused by batching (Figure 3b) and queueing. (2) *Identifying main reasons for request dropping:* Addressing the second question requires the system to identify the main reasons for dropping. Given the variability and burstiness of inference workloads, these reasons may differ significantly. Next, we will introduce how PARD addresses these two critical challenges through proactive request dropping and adaptive request priority.

4 System Design

We start with an overview of PARD (§4.1), followed by its two key designs, proactive request dropping (§4.2) and adaptive request priority (§4.3).

4.1 System Overview

We design PARD, a pipeline inference system that ensures high goodput by *proactively identifying requests with insufficient latency budgets and dynamically selecting proper set of requests to drop under varying workloads.* Figure 4 illustrates system architecture with two key components: (1) distributed controllers for request dispatching, state synchronization, and latency estimation at each module, and (2) parallel workers that execute dropping decisions for individual requests.

PARD allocates each module in the inference pipeline one controller and multiple workers. Each controller's State Planner monitors the runtime state of each worker, including queueing delay, batch size, and throughput ①, and synchronizes these states across modules ②. Given these states, the State Planner calculates the module's required latency budget and sends it to its assigned workers ③. At runtime, requests

will be dispatched among workers by dispatcher ④. Then, requests will enter a request queue called DEPQ to determine its dropping priority based on latency and workload information ⑤. Request Broker in each worker fetches requests from DEPQ and decides whether to drop or infer each request ⑥. Finally, for those inferred requests, workers redirect them to subsequent modules in the inference pipeline ⑦.

4.2 Proactive Request Dropping

Insight and Approach. *To timely drop requests for goodput enhancement, PARD proactively drops requests by estimating their latency with bi-directional runtime information.* Specifically, when making dropping decisions at workers of each module, PARD estimates the end-to-end request latency based on the request cumulative latency information from the preceding modules, as well as the batch wait time distribution and queueing latency from the subsequent modules.

Latency Decoupling. As shown in Figure 5, for a request at module $M_k,\ k \in [1, N]$ of an N-module inference pipeline, its end-to-end latency $\mathbb{L}$ can be decoupled into three parts: cumulative latency from preceding modules $\mathbb{L}_{pre}$, current module's latency budget $\mathbb{L}_{cur}$, and latency budget for subsequent modules $\mathbb{L}_{sub}$. At this stage, only $\mathbb{L}_{pre}$ is determined. If latency in module k is $Lat_{(k)}$, then $\mathbb{L}$ can be expressed as:

$$\mathbb{L} = \mathbb{L}_{pre} + \mathbb{L}_{cur} + \mathbb{L}_{sub} \tag{1a}$$

$$= \sum_{i=1}^{k-1} Lat_{(i)} + Lat_{(k)} + \sum_{i=k+1}^{N} Lat_{(i)} \tag{1b}$$

This decoupling clarifies why reactive policy fails and how PARD should make dropping decisions. Reactive approaches considers only $\mathbb{L}_{pre}$ and $\mathbb{L}_{cur}$ in Equation 1a, ignoring the latency budget for subsequent modules ($\mathbb{L}_{sub}$). This omission prevents timely detection of latency violations, causing incorrect decisions. A proper policy should proactively drop requests based on end-to-end latency for timely decisions.

However, making correct dropping decisions is challenging due to the uncertainty of request latency $Lat_{(k)}$ in subsequent modules. As shown in Figure 3b, when a request arrives at module M_k, it enters the request queue and waits. The scheduler collects the next batch right after the previous one begins execution to avoid GPU idling. Thus, $Lat_{(k)}$ consists of three components: (1) queueing delay Q_k, the time before a request is added to a batch; (2) batch wait time W_k, the time between being added to a batch and the start of inference; and (3) execution duration D_k. Formally:

$$Lat_{(k)} = Q_k + W_k + D_k,\ k \in [1, N] \tag{2}$$

As shown in Figure 3b, batch wait time is uncertain and varies between 0 and the execution duration d_k, depending on when the request is added to the batch. This makes the aggregated batch wait time $\sum_{i=k+1}^{N} W_i$, a component of $\mathbb{L}_{sub}$ in Equation 1a, highly unpredictable and ranges from 0 to $\sum_{i=k+1}^{N} d_i$. Similarly, aggregated queueing delay $\sum_{i=k+1}^{N} Q_i$ fluctuates with workload dynamics. Overestimating these values leads to excessive dropping, while underestimating them increases invalid requests, both ultimately reducing goodput (§5.3).

In summary, to make timely dropping decisions, it's critical to properly estimate the latency for subsequent modules and the one for the preceding and current modules. Next, we introduce how PARD estimates the end-to-end latency $\mathbb{L}$ via its novel State Planner and Request Broker to decide when to drop with bi-directional runtime information.

State Planner. In PARD, each module's State Planner estimates the unpredictable request latency of subsequent modules $\mathbb{L}_{sub}$. As discussed, $\sum_{i=k+1}^{N} Q_i$, $\sum_{i=k+1}^{N} W_i$, and $\sum_{i=k+1}^{N} D_i$ have distinct physical implications and characteristics, and any of them can dominate $\mathbb{L}$ under certain conditions (*e.g.,* a surge in $\sum_{i=k+1}^{N} Q_i$ due to bursts). Thus, PARD estimates these three components independently:

- $\sum_{i=k+1}^{N} Q_i$: Each State Planner monitors the recent average queueing delay q_i using a sliding window[4] and synchronizes with other modules. The cumulative queueing delay is then calculated as $\sum_{i=k+1}^{N} Q_i = \sum_{i=k+1}^{N} q_i$.
- $\sum_{i=k+1}^{N} D_i$: The State Planner periodically synchronizes batch sizes across modules and computes cumulative execution duration using offline model profiling[5], *i.e.,* $\sum_{i=k+1}^{N} D_i = \sum_{i=k+1}^{N} d_i$, where d_i is the profiled duration.
- $\sum_{i=k+1}^{N} W_i$: As shown in Figure 3b, the batch wait time W_i at module M_i varies between 0 and d_i, depending on when a request enters the batch. This uncertainty accumulates across cascaded modules, making $\sum_{i=k+1}^{N} W_i$ unpredictable, ranging from 0 to $\sum_{i=k+1}^{N} d_i$ at module M_k. For timely and accurate dropping decisions, this aggregated wait time must be carefully estimated. We next explain how PARD derives it.

Batch wait estimation. The exact value of $\sum_{i=k+1}^{N} W_i$ for each request is highly variable and unpredictable, leading to under- or over-estimation issues. Nevertheless, its runtime distribution can be observed, allowing PARD to derive a balanced estimate, denoted as w_k, when making dropping decisions at module M_k.

First, both under- and over-estimation reduce goodput, but for opposite reasons. Under-estimating $\sum_{i=k+1}^{N} W_i$ (*e.g.,* $w_k = 0$) causes fewer requests to be dropped at the current module[6]. These *mis-kept* requests are likely to be dropped later in subsequent modules when the actual batch wait time exceeds the estimate, increasing the invalid rate and backpressure, which ultimately raises the overall drop rate. Conversely, over-estimating $\sum_{i=k+1}^{N} W_i$ (*e.g.,* $w_k = \sum_{i=k+1}^{N} d_i$)

[4]Default to a 5s linear weighted window, with sensitivity analysis in §5.4.
[5]DNN inference follows a standard batching process, thus offline profiling yields accurate and stable per-stage latency estimates [1, 32], ensuring PARD's robustness across different models and hardware.
[6]Requests may still be dropped due to other factors, such as long queueing delay in preceding modules.

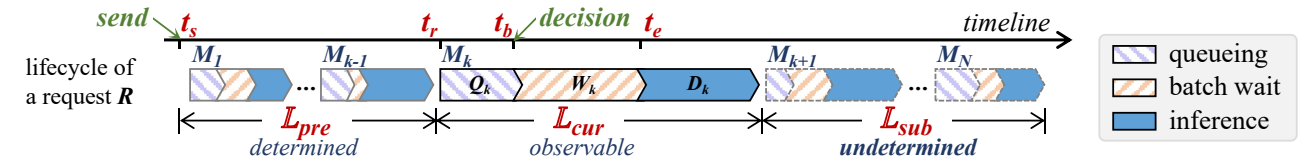


Figure 5. Lifecycle of a request R sent at t_s by the client in an N-module pipeline, where t_r, t_b, t_e represent the moments when the request is received by module M_k, put into a batch, and when the batch execution starts, respectively. At t_b, PARD could get all bi-directional runtime information for each request and make a timely dropping decision before it enters a batch.

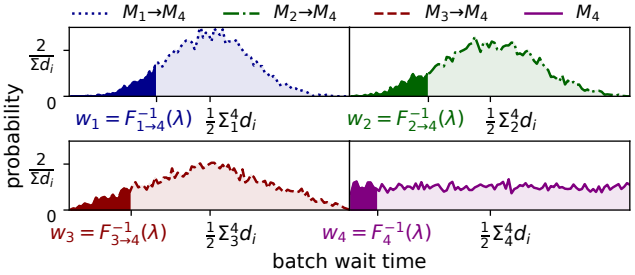


Figure 6. Probability density of requests' total batch wait time at each module in a 4-modules inference pipeline.

leads to premature drops at the current module, since the assumed wait time constraint for downstream modules is overly strict. These *mis-dropped* requests directly increase the drop rate. To avoid both extremes, PARD seeks a *sweet spot* w_k within $[0, \sum_{i=k+1}^{N} d_i]$ that balances invalid and drop rates, thereby improving goodput.

Second, the optimal w_k depends on the module's position in the pipeline. Figure 6 shows the probability density of aggregated batch wait time in a four-module pipeline with fixed batch sizes, based on $10k$ sampled requests from the `wiki` trace (§5.1). As more modules are cascaded, the aggregated batch wait time of weakly correlated W_i becomes concentrated around $\frac{1}{2}\sum_{i=k+1}^{N} d_i$, following the central limit theorem. For example, the total wait time from M_1 to M_4 is centered near $\frac{1}{2}\sum_{i=1}^{4} d_i$, whereas M_4's own wait time is uniformly distributed over $[0, d_4]$. Hence, for modules earlier in the pipeline, w_k should be set closer to $\frac{1}{2}\sum_{i=k+1}^{N} d_i$ to provide effective constraints across the pipeline.

Based on the above observations, the State Planner applies a three-round heuristic to determine w_k for each module. First, it performs random sampling[7] on recent arrivals to derive the probability density function (PDF) $F_{k+1\to N}$ for aggregated batch wait time from M_{k+1} to M_N. Second, it selects a quantile $\lambda \in [0, 1]$ such that $F_{k+1\to N} = \lambda$, meaning that λ proportion of requests have aggregated wait times no greater than w_k. Third, it estimates $w_k = F^{-1}_{k+1\to N}(\lambda)$, which is then used by each module's Request Broker for dropping decisions via $\sum_{i=k+1}^{N} W_i = w_k$.

[7]The runtime complexity is $O(M(N-k+1))$, where M is the length of the collected arrival process, default M=10,000.

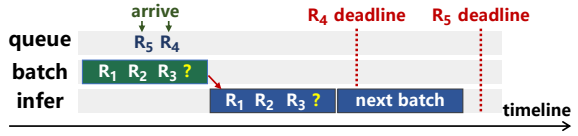


Figure 7. Example of dropping under steady workload.

Intuitively, λ serves as a control knob on how aggressively PARD estimates $\sum_{i=k+1}^{N} W_i$ for subsequent modules. When $\lambda = 0$, $w_k = F^{-1}(0) = 0$ (lower bound); when $\lambda = 1$, $w_k = F^{-1}(1) = \sum_{i=k+1}^{N} d_i$ (upper bound). To balance the invalid computation and drop rate, we set $\lambda = 0.1$ as an empirical default. This serves as a relatively loose estimation since mis-kept requests can be remedied by subsequent modules, whereas mis-dropped requests are irreversible[8]. §5.2 shows that this choice improves goodput across different traces and applications, and Figure 14c shows that the default 0.1 is close to the optimal λ with at most a 0.3% drop rate gap. With this quantile, PARD estimates w_k with two desirable properties: (1) w_k approaches $\frac{1}{2}\sum_{i=k+1}^{N} d_i$ as the number of subsequent modules increases; and (2) w_k varies in real time as batch sizes changes. For example, in a four-module pipeline (Figure 6) with equal duration d, $\lambda = 0.1$ yields:

$$w_1 = \mathbf{0.31}\sum_1^4 d_i = 1.24d \quad w_2 = \mathbf{0.28}\sum_2^4 d_i = 0.84d$$
$$w_3 = \mathbf{0.22}\sum_3^4 d_i = 0.44d \quad w_4 = \mathbf{0.10}\sum_4^4 d_i = 0.10d$$

Although some requests may still be mis-kept or mis-dropped, this heuristic balances drop and invalid rates to improve goodput (§5.3).

Request Broker. PARD 's Request Broker estimates the end-to-end latency $\mathbb{L}$ using Equation 1 and Equation 2:

Backward. As shown in Figure 5, when a request reaches module M_k at timestamp t_r, its cumulative latency $\mathbb{L}_{pre}$ from preceding modules is already determined. Specifically, it equals the elapsed time from when the request was sent (t_s) to when it arrives at M_k (t_r): $\mathbb{L}_{pre} = t_r - t_s$.

Current. Request Broker leverages Equation 2 to calculate the latency $\mathbb{L}_{cur}$ of module M_k in three steps.

- Q_k: When the request is dequeued at timestamp t_b, queueing delay is $Q_k = t_b - t_r$.

[8]Therefore, although $\lambda = 0.5$ yields lower latency estimation error, it does not lead to higher goodput (§5.4).

- W_k: Workers collect the next batch only after the previous batch begins execution to avoid GPU idling. Thus, Request Broker can derive the expected start time t_e of the current batch, equal to the expected end of the previous batch. The batch wait time is then $W_k = t_e - t_b$.
- D_k: Execution duration depends on the DNN model and batch size, obtained from offline profiling, *i.e.,* $D_k = d_k$, where d_k is the profiled duration at the current batch size.

Forward. As discussed, Request Broker collects the latency information for subsequent modules $\mathbb{L}_{sub} = \sum_{i=k+1}^{N} Q_i + \sum_{i=k+1}^{N} D_i + \sum_{i=k+1}^{N} W_i$ from State Planner.

In summary, as shown in Figure 4, before a request enters a batch via Request Broker (*i.e.,* at t_b in Figure 5), Request Broker and State Planner could obtain all bi-directional runtime information in Equation 1 and Equation 2. Request Broker then calculates end-to-end latency $\mathbb{L}$ as in Equation 3. For directed acyclic graph (DAG) pipelines, where each vertex is a module, PARD estimates the latency $\mathbb{L}'$ of a request along each subsequent DAG path and takes the maximum as the end-to-end latency estimation.

$$\mathbb{L} = \mathbb{L}_{pre} + \mathbb{L}_{cur} + \mathbb{L}_{sub} \tag{3a}$$

$$= \underbrace{t_e - t_s + d_k}_{\text{Request Broker}} + \underbrace{\sum_{i=k+1}^{N} q_i + \sum_{i=k+1}^{N} d_i + F^{-1}_{k+1\to N}(\lambda)}_{\text{State Planner}} \tag{3b}$$

4.3 Adaptive Request Priority

Insight and Approach. *To select the set of requests to drop that enhances goodput, PARD dynamically reorders requests based on their remaining latency budget and workload intensity.* PARD employs a double-ended priority queue (DEPQ) to alter the request decision order and designs adaptive request prioritization to handle bursty workloads as follows.

First, requests should be reordered by remaining latency budgets, since arrival order does not reflect them. Batching introduces uncertainty: consecutively arriving requests may experience different wait times in preceding modules, making arrival order ineffective for estimating remaining budgets. Therefore, PARD should use an efficient reordering strategy based on remaining latency, as illustrated in Figure 3a, to select the proper set of requests to drop.

Second, PARD should reorder requests dynamically based on workload, since the causes of request dropping vary with workloads. During bursts, drops mainly arise from queueing delays caused by request accumulation. In this case, PARD's reordering should reduce the queueing delay, thereby preserving larger latency budget for subsequent modules. Conversely, under steady workloads (*i.e.,* input workload does not exceed worker throughput), the main reason for dropping is latency uncertainty. For example, in Figure 7, suppose a worker needs one more request to form a batch, with R_4 and R_5 waiting. R_4 has experienced longer latency in preceding modules and arrives later than R_5. If R_5 is chosen, R_4 will suffer a long batch wait and miss the deadline; if R_4 is chosen, both meet the latency objective. In such cases, reordering should prioritize requests with lower remaining budgets, reducing queueing and wait time to complete more requests within the latency objective.

In summary, PARD should adapt to bursty and variable inference workloads by selectively dropping requests to improve goodput. To achieve this, it employs two request prioritization mechanisms and a delayed adaptive priority transition policy, reordering requests in real-time based on remaining latency budget and workload intensity.

Request Priority with DEPQ. PARD prioritizes requests using a double-ended priority queue (DEPQ), deciding which to drop based on remaining latency budgets and workload intensity. This ensures that the correct set of requests is dropped. We define the module load factor $\mu = \boldsymbol{T_{in}}/\boldsymbol{T_m}$, where $\boldsymbol{T_{in}}$ is the input workload and $\boldsymbol{T_m}$ is the module throughput determined by batch size and execution duration. μ thus indicates whether the system is under-provisioned. Depending on μ, PARD applies two policies:

- $\mu > 1$: The system is under-provisioned. To avoid excessive queueing that consumes latency budgets, PARD applies the High Budget First (HBF) policy, prioritizing requests with the largest remaining budgets. This preserves more budget for subsequent modules, thereby reducing the probability of requests being dropped, lowering drop rate and invalid rate (§5.3).
- $\mu \leq 1$: The workload is within module's processing capacity. To reduce unnecessary drops caused by latency uncertainty, PARD applies the Low Budget First (LBF) policy, prioritizing requests with the smallest remaining budgets. This reduces the probability of dropping requests with smaller latency budgets, leading to a lower drop rate and higher goodput (§5.3).

PARD implements both prioritization strategies using a DEPQ in each worker, with the remaining latency budget as the priority. The DEPQ can simultaneously pop requests with the largest and smallest remaining latency budgets using a min-max heap. PARD uses DEPQ to enable more effective dropping decisions based on remaining latency budget and workload burstiness rather than arrival order.

Adaptive Priority with Delayed Transition. To ensure smooth switching between HBF and LBF, PARD adapts prioritization based on two thresholds, $Th_{HBF} = 1.0 + \epsilon$ and $Th_{LBF} = 1.0 - \epsilon$. When $\mu > Th_{HBF}$, it switches to HBF; when $\mu < Th_{LBF}$, it switches to LBF. For μ within $[1.0 - \epsilon, 1.0 + \epsilon]$, the priority remains unchanged, avoiding frequent changes from workload fluctuations and sustaining higher goodput (§5.3). The boundary ϵ is computed dynamically as $\epsilon = \frac{\sum|\boldsymbol{T_{in}} - \boldsymbol{T_s}|}{\sum \boldsymbol{T_{in}}}$, where $\boldsymbol{T_s}$ is the workload smoothed by a sliding-window average. This allows ϵ to expand under bursty workloads, suppressing frequent switches and ensuring stable priority transitions with consistently high goodput (§5.2).

5 Evaluation

We compare PARD against existing inference systems in terms of its goodput and request drop rate. Then, we perform an ablation study to quantify the importance of PARD's design decisions on goodput enhancement.

5.1 Methodology

Implementation. We implement PARD in roughly 15k lines of Python: 6.5k for system and tests, 5k for the application library, and the rest for benchmarks. PyTorch [33] serves as the default inference backend. Before startup, PARD performs an offline profiling to obtain per-model execution duration and throughput under various batch sizes for online latency estimation. PARD also adopts dynamic batching and resource scaling similar to [1, 5]: yields feasible batch sizes and per-worker throughput based on offline profiling, and adjusts the number of workers per module based on request rate and per-worker throughput at runtime. Controllers and workers run in separate containers, communicating via gRPC for low-overhead request routing and state exchange.

To support widely deployed DAG-style pipeline applications [2, 34], PARD defines a pipeline via a JSON file composed of module configurations (`name`, `id`, `pres`, `subs`), where `name` is the module registered in the application library, `pres` and `subs` specify the preceding and subsequent module IDs. PARD automatically splits requests when `subs` contains multiple modules and merges sub-requests when `pres` has multiple inputs. In such cases, the State Planner estimates a request's end-to-end latency as the maximum across all DAG paths, as described in §4.2.

Testbed. Experiments run on a 16-machine cluster, each with 4 NVIDIA 2080Ti GPUs and 48 CPU cores. We virtualize the cluster into 64 worker containers (one GPU per container). NTP [35] keeps clocks synchronized to millisecond-level accuracy so that cross-container timestamps are correct when computing cross-module latencies (*e.g.*, $t_e - t_s$ in Equation 3) for dropping decisions.

Workload. Following prior works [1, 2, 5, 6], we build three real-world pipelines: (1) `tm` (traffic monitoring), which uses three models (object detection, face recognition, and text recognition) to monitor vehicle and pedestrian information. (2) `lv` (live video analysis), which analyzes live video using five models (person detection, face recognition, expression recognition, eye tracking, and pose recognition). (3) `ga` (game analysis), which analyzes game streaming using five models (object detection, kill count detection, alive player recognition, health value recognition, and icon recognition). We also build a DAG-style pipeline based on `lv`: (4) `da` (DAG-style live video analysis), where requests from person detection module are simultaneously forwarded to the pose recognition and face recognition modules, and their outputs are subsequently merged in the expression recognition module.

Table 1. Ablation baselines considered in §5.3.

Ablation	Source	Key characteristic
PARD-back	[2, 28]	Considers preceding modules only
PARD-sf	[21]	Ignores Q, W of subsequent modules
PARD-oc	[44]	Overload control based on Q
PARD-split	[22]	Fixed per-module SLO split
PARD-WCL	–	Split latency budget dynamically
PARD-lower	–	Assumes batch wait as 0
PARD-upper	–	Assumes batch wait as Σd_i
PARD-FCFS	[1]	Drops by arrival order
PARD-HBF	–	High-Budget-First only
PARD-LBF	[9]	Low-Budget-First only

Following prior work [1, 2, 5], we set latency SLOs for the pipelines to 400ms, 500ms, 600ms, and 420ms, respectively, enabling flexibility for dynamic batching and resource efficiency. We also conduct a sensitivity study in §5.4 to evaluate how PARD performs under tighter and looser SLOs. Input video streams are collected from public datasets [36, 37] and video streaming platforms [38, 39]. As shown on the left of Figure 10, we replay three representative real-world traces as request rates: the Wikipedia access trace [40], the Twitter access trace [41], and the Azure Function trace [42]. These traces capture typical periodic and bursty patterns of DNN inference workloads and are widely used in evaluating inference serving systems [10, 31, 43].

Baseline. We use Nexus [1] and Clipper [22] as primary baselines because they are open-source and implement dropping policies. Nexus adopts a reactive policy that scans the queue in arrival order with a sliding window equal to the batch size, stopping at the first position where all requests in the window can meet the current module's latency budget and dropping all earlier ones. Clipper, designed for single-module applications, drops requests only if they already exceed the latency objective before inference. Following [1], we extend Clipper to Clipper++ for pipelines by proportionally dividing the end-to-end SLO across modules, *i.e.*, $SLO_k = SLO * d_k / \sum_{i=1}^{N} d_i$, and making per-module drop decisions accordingly. We also include a naive baseline, which applies no dropping policy.

Several recent serving systems also incorporate request dropping [2, 9, 21, 28, 44]. To isolate each design in PARD, we construct ablation baselines by disabling or replacing components with those from these systems. Table 1 summarizes these baselines, with detailed explanations in §5.3.

Metrics. We focused on three metrics: (1) *Goodput*: the number of requests completed within the latency objective per unit time, indicating quality of service. We focus on periods of the workload that require request dropping and provide the goodput during these periods. (2) *Drop rate*: the ratio of dropped requests to total requests. Requests that have completed inference but violate the SLO are also considered

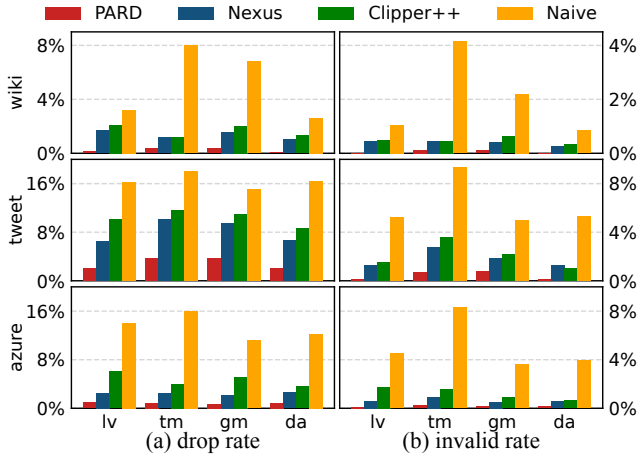


Figure 8. Average drop rate and invalid rate of PARD and baseline systems under 12 workloads.

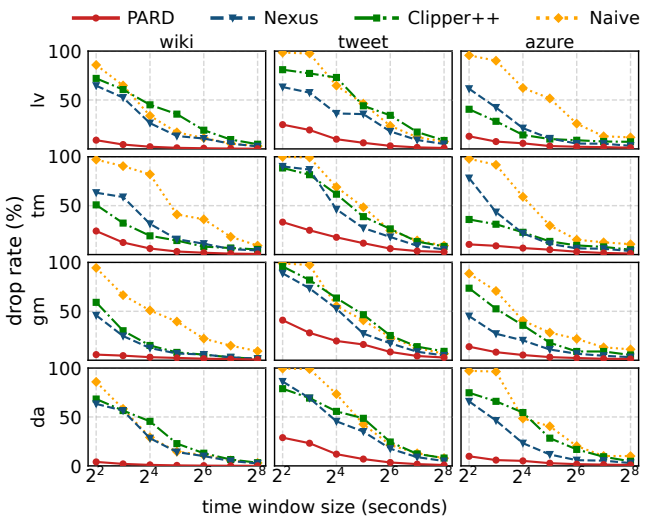


Figure 9. The maximum average drop rate over the entire runtime in different time window sizes.

dropped requests. (3) *Invalid rate*: the ratio of invalid computation to total computation, defined as the ratio of GPU time consumed by dropped requests to the total GPU time consumed by all requests. An optimal inference system should enhance goodput while minimizing drop and invalid rates.

5.2 Comparison Results

Overall Results. Figure 10 presents the normalized goodput of PARD compared with three alternatives, where PARD maintains higher goodput across nearly the entire period. Specifically, as shown in Figure 8, PARD drops an average of only 0.12%-3.6% of requests under various workloads, which reduces the drop rate and wasted computation resources by 1.6×-16.7× and 1.5×-61.9× than Nexus and Clipper++. Therefore, as shown in Figure 10, PARD increases the goodput by 16%-176% compared to Nexus and Clipper++, indicating the effectiveness of its proactive dropping policy. Besides, the naive baseline has the worst goodput under all six workloads, with average drop and invalid rates up to 35× and 129× those of PARD, highlighting the importance of request dropping for goodput enhancement.

To understand the performance gap, we attribute the advantage of PARD to two factors, further validated by the ablation study in §5.3. First, existing systems fail to decide *when* to drop requests across modules. Nexus makes reactive dropping decisions without considering the latency budget for subsequent modules, causing 51%–97% of drops to cluster in the latter half of the pipeline. Clipper++ identifies requests with insufficient latency budget earlier by splitting the end-to-end latency SLO into per-module budgets. However, 31%–89% of drops still occur in later modules, and the system suffers higher invalid rates than both Nexus and PARD since splitting restricts requests' latency budget flexibility (§5.3). In contrast, PARD's proactive dropping enables early decisions based on end-to-end latency estimation. Thus PARD achieves lower invalid rates and drop rates, leading to higher goodput.

Second, existing systems fail to select *which* requests to drop properly. Both Nexus and Clipper++ drop requests purely in request arrival order, ignoring remaining latency budgets and workload intensity. This design wastes latency budgets during bursts and adds unnecessary queueing delays for requests with tighter deadlines, yielding transient drop rates up to 90% and 96% (Figure 9). Instead, PARD dynamically reorders requests using adaptive priority for selective dropping decisions, cutting transient drop rates by 41%–98% across all timescales compared with baselines and consistently improving goodput (Figure 9).

Applicable to DAG workloads. In DAG-style pipelines, PARD estimates the maximum end-to-end latency across all branches and drops requests accordingly. As shown in Figure 8a, this yields 2.1×–12.6× and 3.1×–16.7× lower drop rates than Nexus and Clipper++ under application da. Unlike lv, da executes pose and face recognition modules in parallel; dropping in one module invalidates computation in the other one for da. This raises PARD 's invalid rate to 1.21×–1.36× that of lv, though still 3.2×–15.9× lower than baselines.

Recent DAG pipelines further complicate request dropping with request-specific dynamic paths [43, 45, 46], where the chosen branch depends on intermediate results. This variability amplifies latency uncertainty and reduces the accuracy of PARD 's dropping decisions. To evaluate this, we adapt da so each request probabilistically takes either the pose or face branch. In this setting, PARD 's drop rate rises by 0.05×, 0.21×, and 0.10× across three traces due to mis-estimation. Request-path prediction techniques [47, 48] could be incorporated into PARD to yield more accurate latency estimation, which we leave for future work.

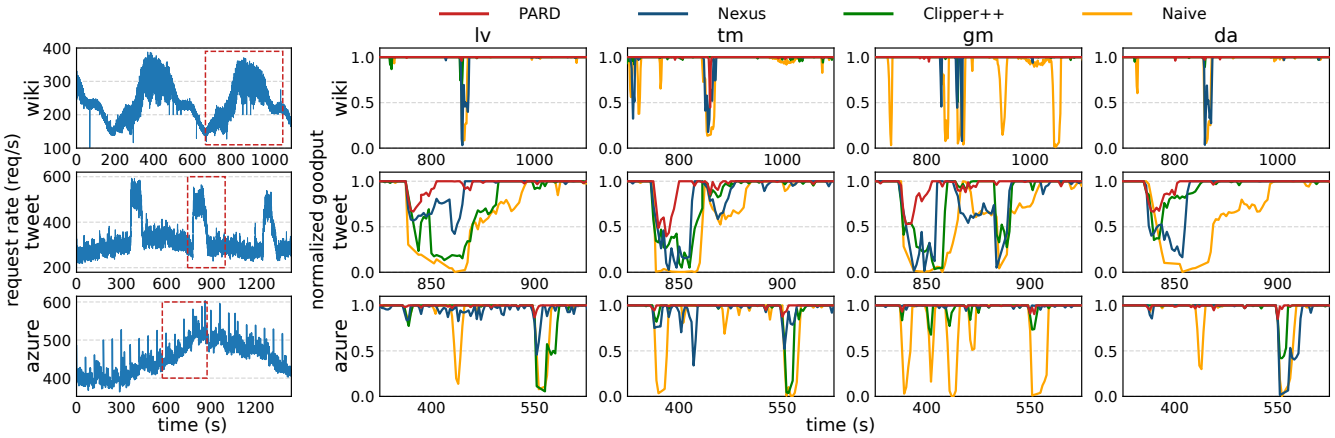
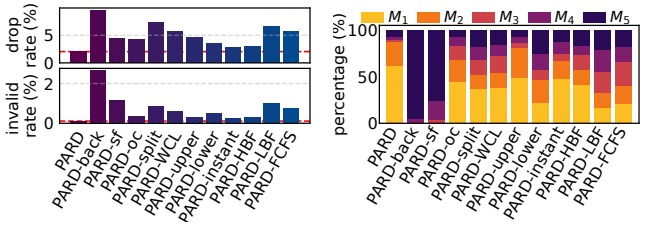


Figure 10. Left: Three real-world traces [40–42]. Right: The normalized real-time goodput of PARD and baseline systems across 12 workloads, corresponding to the red-boxed regions in each trace.



(a) Drop rate and invalid rate **(b)** Drops at each module

Figure 11. Comparison results of PARD and its alternatives: (a) Average drop rate and invalid rate. (b) Percentage of dropped requests at each module.

5.3 Ablation Study

In this section, we disable each of PARD's features to demonstrate its contribution to guaranteeing goodput[9].

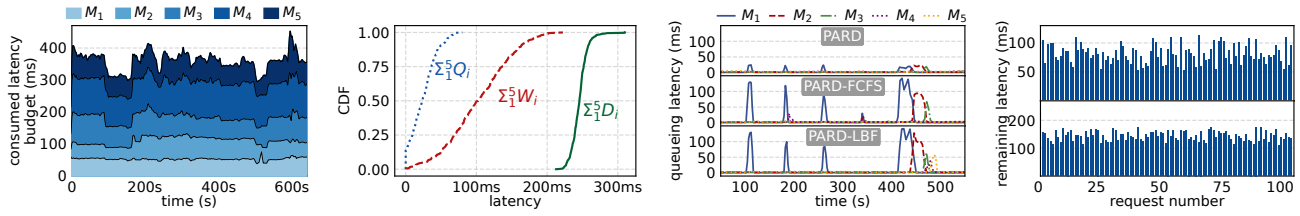
How important is proactive latency estimation? PARD outperforms alternatives by making timely drop decisions using end-to-end latency estimates derived from *bi-directional runtime information* (§5.2). To verify that, we compare against (1) PARD-back, which considers only preceding and current modules (*i.e.,* $\mathbb{L}_{sub} = 0$), similar to Clockwork [28], Nexus [1], and Scrooge [2]; (2) PARD-sf, which accounts for execution durations of subsequent modules (*i.e.,* $\mathbb{L}_{sub} = \sum_{i=k+1}^{N} d_i$), similar to DREAM [21]; and (3) PARD-oc, which adopts DAGOR's overload control strategy [44]. In PARD-oc, requests are dropped when the average queueing delay of a module exceeds a threshold T; the module then notifies preceding modules and admits requests at a rate of $(1 - \alpha) \times$ input_rate simultaneously[10].

As shown in Figure 11a, PARD-back, PARD-sf, and PARD-oc yield drop rates 1.1×-3.6× higher than PARD, with invalid rates 2.1×-24× higher. PARD-back ignores downstream budgets, causing an average drop rate of 9.5%, and 95% of drops concentrated in the last module, producing the highest invalid rate (Figure 11b). PARD-sf improves to a 4.5% drop rate with 76% of drops in the last module, but still suffers late drops by neglecting subsequent queueing and batch wait ($\sum_{i=k+1}^{N} Q_i$ and $\sum_{i=k+1}^{N} W_i$), highlighting their importance. PARD-oc achieves fewer late drops by jointly throttling admission across modules, with only 7.2% requests being dropped in the last module. However, its microservice-oriented design overlooks the large latency uncertainty from batching. This coarse-grained design still yields drop and invalid rates 2.1× and 3.0× higher than PARD, highlighting the need for proactive latency estimation in DNN pipelines.

In contrast, PARD, by accurately estimating $\mathbb{L}_{sub}$, concentrates 87% of drops in the first two modules, reducing invalid rates by 2.1× to 24× than three baselines. The reduced computation benefits other requests, reducing the average queueing delay by 4.7% to 12%. Consequently, PARD's proactive policy with bi-directional runtime information reduces drop rates by up to 52% to 78%.

Why not split the latency objective? PARD drops requests by comparing estimated end-to-end latency $\mathbb{L}$ with the overall SLO. An alternative is to split the SLO into per-module budgets and drop requests when their latency exceeds a module's budget. We argue this approach cannot drop requests properly. To validate, we compare: (1) PARD-split, which splits the SLO into fixed per-module budgets, similar to Clipper++; and (2) PARD-WCL, which dynamically allocates budgets based on modules' runtime worst-case latency (WCL), including queueing, batch wait, and execution.

[9]Note that we use `lv-tweet` workload for ablation study and sensitivity analysis. Other workloads show similar results and are omitted for brevity.
[10]We tune T and α for each traces, and obtain the best performance with $\alpha = 0.4$, $T = 20$ms for the `wiki`, and $T = 25$ms for `tweet` and `azure`.



(a) Consumed latency budget **(b)** Latency distribution **(c)** Queueing delay **(d)** Remaining latency budget

Figure 12. (a) Consumed latency budget of 100k SLO-compliant requests at each module. (b) CDF of end-to-end queueing delay $\sum_{i=k+1}^{N} Q_i$, batch wait time $\sum_{i=k+1}^{N} W_i$, and inference duration $\sum_{i=k+1}^{N} D_i$. (c) Queueing delay of each module in the pipeline during workload burst. (d) The remaining latency budget of 100 consecutive requests at M_2 (lower) and M_3 (upper).

As shown in Figure 11a, PARD-split incurs drop and invalid rates 2.6× and 6.7× higher than PARD. Splitting prevents early modules from over-consuming budgets, concentrating 51% of drops in the first two modules (Figure 11b) and keeping the invalid rate low (0.82%). However, splitting limits flexibility: when a module suffers queueing delays from cold starts (*e.g.,* around 200s and 600s in Figure 12a), the system cannot reallocate budgets to mitigate the bottleneck. PARD-WCL reduces drop and invalid rates by 30% and 27% relative to PARD-split by dynamically allocating budgets according to WCL. While even with tuned allocate frequency, it still underperforms due to budget demands fluctuating rapidly across modules (Figure 12a), and batching introduces large variation in per-request latency (Figure 12d). As a result, its drop and invalid rates remain 2.8× and 5.4× higher than PARD. These results show that neither static nor dynamic splitting can handle the latency uncertainty of batching in DNN pipelines. In contrast, by treating the pipeline as a whole and estimating end-to-end per-request latency, PARD achieves 64% and 72% lower drop rates than PARD-split and PARD-WCL.

Why use a sweet spot w_k? Because the aggregated batch wait time $\sum_{i=k+1}^{N} W_i$ is unpredictable and highly variable, PARD derives a sweet spot w_k to estimate it for dropping decisions. To validate this design, we compare PARD with: (1) PARD-lower, which assumes the lower bound $\sum_{i=k+1}^{N} W_i = 0$ (*i.e.,* $\mathbb{L}_{sub} = \sum_{i=k+1}^{N} Q_i + \sum_{i=k+1}^{N} d_i$); and (2) PARD-upper, which assumes the upper bound $\sum_{i=k+1}^{N} W_i = \sum_{i=k+1}^{N} d_i$ (*i.e.,* $\mathbb{L}_{sub} = \sum_{i=k+1}^{N} Q_i + 2\sum_{i=k+1}^{N} d_i$).

As shown in Figure 11a, PARD-lower under-estimates $\sum_{i=k+1}^{N} W_i$, mis-keeping requests that lack sufficient budgets for later modules. Its invalid rate is 3.5× higher than PARD, with 3.2× more drops concentrated in the last three modules (Figure 11b), lowering goodput by 6.8%. In contrast, PARD-upper over-estimates $\sum_{i=k+1}^{N} W_i$, mis-dropping requests with adequate budgets. It yields a drop rate 1.3× higher, leading to 6.5% lower goodput. Figure 12b further shows that $\sum_{i=k+1}^{N} W_i$ exhibits far greater variance than $\sum_{i=k+1}^{N} Q_i$ or $\sum_{i=k+1}^{N} D_i$. Hence, PARD derives w_k within $[0, \sum_{i=k+1}^{N} d_i]$ to estimate $\sum_{i=k+1}^{N} W_i$. While some requests are still mis-kept or mis-dropped, this sweet-spot estimation balances drop and invalid rates, thereby improving goodput.

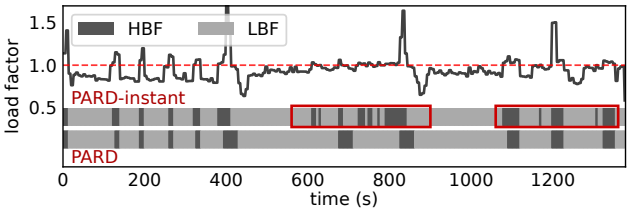
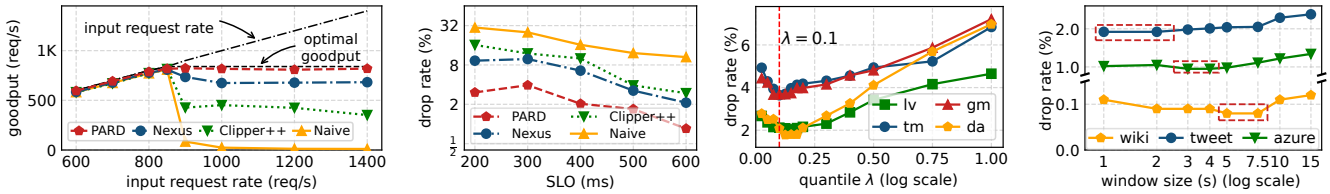


Figure 13. Load factor and prioritization mechanism transition in PARD and PARD-instant.

Why does the remaining latency budget matter? Reactive policies drop requests in arrival order, leading to suboptimal decisions. In contrast, PARD selects the dropping set based on remaining latency budgets and workload intensity, sustaining goodput under diverse workloads. To verify this, we compare with: (1) PARD-FCFS, which drops by arrival order like Nexus [1] and Clipper++; (2) PARD-HBF, which always applies HBF; (3) PARD-LBF, which always applies LBF, similar to SHEPHERD [9]; and (4) PARD-instant, which applies HBF and LBF without delayed transition.

As shown in Figure 11a, the drop rates for PARD-FCFS, PARD-LBF, and PARD-HBF are 1.8×, 2.2×, and 0.5× higher than PARD's due to several factors. (1) During workload bursts, PARD-FCFS always prioritizes the earliest arrived requests, over-consuming subsequent requests' latency budgets and causing accumulation (Figure 12c). This increases queueing delay by 34% and reduces goodput by 24% compared to PARD. Similarly, PARD-LBF suffers from request accumulation, resulting in a goodput 29% lower than PARD's. (2) During steady workloads, PARD-FCFS fails to make proper dropping decisions due to latency uncertainty. Figure 12d shows highly variable, time-independent remaining latency budgets for 100 requests in M_2 and M_3. Ignoring this variability, PARD-FCFS and PARD-HBF achieve 24% and 6% lower goodput than PARD. Although PARD-instant achieves the lowest drop rate among the baselines, it still drops 25% more requests than PARD due to frequent transition between HBF



(a) Stress-testing **(b)** SLO sensitivity **(c)** Sensitivity of λ **(d)** Sensitivity of window size

Figure 14. Experimental results for sensitivity analysis: (a) Average goodput across different request rates. (b) Average drop rate under various SLO settings. (c) Drop rate at different quantile λ. (d) Drop rate at different moving window sizes[11].

and LBF during fluctuations (Figure 13). Conversely, PARD 's delayed transition provides smooth switching and achieves the highest goodput.

5.4 Sensitivity and Overhead Analysis

Stress-testing. PARD makes timely, precise dropping decisions using bi-directional runtime information and workload intensity. To evaluate its behavior under heavy load, we fix the instance number of PARD and three baselines, and increase the request rate until goodput stabilizes. As shown in Figure 14a, Naive quickly fails to meet the SLO because it cannot clear accumulated requests. Nexus and Clipper++ also degrade since they rely on unidirectional information and over-consume latency budgets. When request rates exceed the testbed's capability, PARD achieves 11.9%–132.9% higher goodput and the gap between PARD 's goodput and the optimal goodput (*i.e.,* the minimum of request rate and system throughput) is 3.4×–23.4× smaller than that of baselines.

Sensitivity to SLO. We set each application's SLO following prior works [1, 2, 5] and evaluate sensitivity by varying SLOs from 200ms to 600ms. To meet different constraints, all systems adjust expected batch sizes for each module. As shown in Figure 14b, PARD sustains lowest drop rates (0.85%–3.04%) across SLOs, which is 1.9×–5.3× lower than baselines, demonstrating the robustness of PARD 's design.

Sensitivity analysis of λ. We evaluate the sensitivity of quantile λ, which determines w_k. As shown in Figure 14c, the optimal λ is not always 0.1 but consistently lies between 0.075 and 0.15, with drop rates showing little variation within this range. Moreover, while $\lambda = 0.5$ yields lower batch wait estimation error by approximating the mean batch wait time (Figure 6), the default $\lambda = 0.1$ achieves a 1.1%–2.0% lower drop rate by mitigating irreversible mis-drops (§4.2). Therefore, we set $\lambda = 0.1$ by default. For new applications, we also recommend 0.1 as the default setting, while users can also tune λ for the best performance.

Sensitivity to window size. PARD applies a 5s sliding window to smooth recent queueing delays. As in Figure 14d, the optimal window is trace-dependent: bursty traces such as `tweet` (coefficient of variation, CV $\approx$ 1.0) and `azure` (CV $\approx$ 1.3) favor shorter windows due to rapidly varying queueing delays, while the stable `wiki` trace (CV $\approx$ 0.47) benefits from longer ones. The drop-rate gap between each trace's optimum and the 5s default is only 3.2%–6.3%, confirming that the default is sufficient for our main results. For new workloads, a practical guideline is: 5–7s for stable traces (CV $<$ 0.5), 3–5s for moderately bursty traces (0.5 $\leq$ CV $<$ 1.0), and 1–3s for highly bursty traces (CV $\geq$ 1.0).

Overheads. PARD introduces three sources of overhead: (1) *Batch-wait distribution updates*, executed asynchronously on a separate thread with complexity $O(MN)$ (M = 10,000 samples over N modules, typically $N \leq 6$ [2, 4, 5]). This process is independent of the worker count and adds no extra request latency. (2) *State synchronization*, which exchanges compact module states (queueing delay, batch size, throughput, drop rate, and batch-wait distribution) bundled into one protobuf package per second in a separate thread. This results in $<$ 3.2 Kbps traffic per worker, which is negligible compared to the 240-640 Mbps data plane bandwidth. Even scaling to 1000 workers, the aggregate control traffic remains small ($\approx$ 0.64 Mbps), ensuring scalability. (3) *Request reordering in DEPQ*, with `put()` and `get()` operations of $O(\log n)$ for average queue length n. Since this overhead is local to each worker, it does not increase with cluster scale. Experiments show these operations add less than 0.16% request latency.

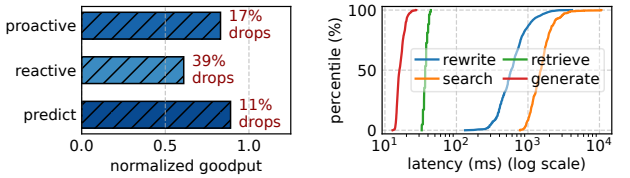
6 Related Work

Request Dropping. Numerous inference systems adopt dropping to ensure service quality [1, 2, 4, 21, 22, 28, 30]. Nexus [1], Scrooge [2], and IPA [4] drop requests without considering downstream latency budgets, similar to PARD-back, which wastes resources by executing requests already close to timeout. DREAM [21] assumes requests will run with the smallest batch size and no queueing in later modules, similar to PARD-sf, leading to drop-too-late issue. Orloj [30] and Clockwork [28] target single-model workloads, dropping only when deadlines are missed or remaining budget is insufficient. Extending them to pipelines through latency splitting (*e.g.,* PARD-split or PARD-WCL) still yields reactive, stage-local designs that lack pipeline-wide awareness.

[11] We analyze λ sensitivity on `tweet` trace and window size sensitivity on 1v application, omitting other workloads as they exhibit similar trends.

Table 2. Evaluation setup of RAG workflow.

Setup	Description
Testbed	2 × A100-80GB GPUs, vLLM v0.9.0 [58], LangChain [59]
Input	10k queries from HotpotQA [60] (Azure trace).
`Rewrite`	Rewrite query with Llama-3-8B [61] (*continuous batching*).
`Retrieve`	Retrieve relevant context from FAISS [62] database, which contains 483k items from HotpotQA (*batching execution*)
`Search`	Search online with Tavily API [63] (*multithreading*).
`Generate`	Generate answer with Llama-3-8B (*continuous batching*).



(a) Normalized Googput **(b)** Module Latency Distribution

Figure 15. Proactive drop demo in RAG workflow.

Beyond inference, several overload control techniques for microservices and DAG workloads [44, 49–51] can also be viewed as request dropping. These methods perform reactive admission control based on queueing delays or instantaneous load. However, like PARD-oc, they neither handle latency budgets across modules nor address batching-induced uncertainty, limiting their effectiveness for inference pipelines.

Latency Profiling. PARD estimates per-request end-to-end latency by decoupling and profiling latency distributions to enable proactive dropping. Prior works [9, 28, 30, 52, 53] also profile request latency online but with different challenges and approaches. Orloj [30] captures empirical execution latency distributions in dynamic DNNs for request scheduling and batching to improve finish rates. Protego [53] monitors contention queues to estimate per-request queueing delays and drop requests that would miss SLOs, thereby reducing tail latency. MittOS [52] predicts I/O request latency in the OS via queueing and device-efficiency profiling, rejecting requests unlikely to meet SLOs for early failover. While PARD shares these insights on latency profiling and early rejection, it faces the unique challenge of highly uncertain batch wait times, amplified by model cascades. This motivates its pipeline-specific design, which balances over- and under-estimation to achieve high goodput.

Other works [54–56] use profiled latency distributions to locate bottlenecks or manage resources rather than making per-request decisions. Nonetheless, their approaches inspire PARD 's batch-wait estimation.

Request Priority. Several inference systems [9, 57] assign priorities to requests similar to PARD-HBF and PARD-LBF. For example, Shepherd [9] prioritizes requests with the closest deadlines, while GrandSLAm [57] reorders requests in descending order of remaining latency budget. However, these systems adopt fixed priority schemes, limiting their ability to sustain high goodput under diverse workloads.

Inference Optimizations. Besides, numerous systems propose other inference optimization techniques for DNN serving, including resource scaling [3, 64], batching-aware scheduling [32, 65], GPU scheduling [16, 17], ensembling [10, 66], pipelining [67, 68], and spot instances [8, 69]. PARD's proactive request dropping is orthogonal to these approaches and can be combined to further enhance the quality of service.

7 Applicable to Other Workload

The core insight of PARD, that proactively dropping certain requests can enhance overall goodput, generalizes beyond DNN inference pipelines. To evaluate PARD under a stress test scenario where its assumptions are challenged, we apply it to Retrieval-Augmented Generation (RAG) [70]. RAG shares multi-stage execution and strict latency SLOs with DNN pipelines but introduces autoregressive generation and continuous batching, which break PARD's assumption of fixed execution duration and batch wait.

As a case study, we implement a four-module RAG pipeline as detailed in Table 2, where `retrieve` and `search` run in parallel as a DAG-style workflow. We set a time-to-first-token (TTFT) SLO of 5s and compare two dropping policies: (1) *proactive*, a customized version of PARD's proactive dropping that estimates `rewrite` and `search` latencies by recent averages and estimates `generate`'s prefill latency using offline profiling and input length, while `retrieve` latency is estimated as in PARD; and (2) *reactive*, which drops requests only after exceeding the TTFT SLO.

As in Figure 15a, proactive dropping reduces drop rate by 22%, confirming its applicability. However, Figure 15b highlights key differences from DNN inference: `rewrite` latency varies with output length; `rewrite` and `generate` use continuous batching, eliminating batch wait; and `search` suffers long-tail latency from network delays. Consequently, even proactive dropping leaves 17% of requests dropped. With oracle knowledge of `rewrite` output length (from offline runs with temperature 0), the drop rate falls to 11% (policy *predict* in Figure 15a). While unrealistic in practice, this can be approximated using orthogonal prediction strategies.

Limitation. First, for autoregressive models (*e.g.,* in RAG), the execution duration depends on the output token length, which is unknown beforehand. PARD currently relies on offline profiling, leading to estimation errors. Orthogonal output-length prediction techniques [71, 72] could be integrated to improve PARD's accuracy. Second, for DAG workflows with request-specific dynamic paths (e.g., conditional execution), PARD conservatively estimates the maximum latency across all potential branches. This may lead to overestimation and unnecessary drops. Future work can incorporate request-path prediction [47, 48] to identify the likely path for more accurate proactive dropping.

8 Conclusion

In this paper, we present PARD, a DNN inference system designed to enhance goodput by proactively dropping certain requests to meet latency objectives under real-world workloads. PARD leverages novel proactive request dropping and adaptive request priority methods to determine when to drop requests and which requests to drop at each module, optimizing goodput for the entire workload. Unlike existing systems' *reactive* dropping policies, which suffer from drop-too-late and drop-wrong-set issues, PARD's *proactive* approach significantly improves the goodput. The evaluation shows that PARD achieves 16%-176% higher goodput than the state of the art, while reducing the drop rate and invalid rate by 1.6×-17× and 1.5×-62× respectively.

9 Acknowledgments

We appreciate the insightful feedback from the anonymous reviewers and our shepherd, Shungeng Zhang. This work is supported by the National Natural Science Foundation of China under grants No. 62572341.

References

[1] 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.

[2] Yitao Hu, Rajrup Ghosh, and Ramesh Govindan. Scrooge: A cost-effective deep learning inference system. In *Proceedings of the ACM Symposium on Cloud Computing*, pages 624–638, 2021.

[3] Kamran Razavi, Manisha Luthra, Boris Koldehofe, Max Mühlhäuser, and Lin Wang. Fa2: Fast, accurate autoscaling for serving deep learning inference with sla guarantees. In *2022 IEEE 28th Real-Time and Embedded Technology and Applications Symposium (RTAS)*, pages 146–159. IEEE, 2022.

[4] Saeid Ghafouri, Kamran Razavi, Mehran Salmani, Alireza Sanaee, Tania Lorido-Botran, Lin Wang, Joseph Doyle, and Pooyan Jamshidi. Ipa: Inference pipeline adaptation to achieve high accuracy and cost-efficiency. *arXiv preprint arXiv:2308.12871*, 2023.

[5] 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 *Proceedings of the 11th ACM Symposium on Cloud Computing*, pages 477–491, 2020.

[6] Kevin Hsieh, Ganesh Ananthanarayanan, Peter Bodik, Shivaram Venkataraman, Paramvir Bahl, Matthai Philipose, Phillip B Gibbons, and Onur Mutlu. Focus: Querying large video datasets with low latency and low cost. In *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18)*, pages 269–286, 2018.

[7] Yecheng Xiang and Hyoseung Kim. Pipelined data-parallel cpu/gpu scheduling for multi-dnn real-time inference. In *2019 IEEE Real-Time Systems Symposium (RTSS)*, pages 392–405. IEEE, 2019.

[8] Chengliang Zhang, Minchen Yu, Wei Wang, and Feng Yan. MArk: Exploiting cloud services for cost-effective, slo-aware machine learning inference serving. In *USENIX Annual Technical Conference*, pages 1049–1062, 2019.

[9] Hong Zhang, Yupeng Tang, Anurag Khandelwal, and Ion Stoica. SHEPHERD: Serving DNNs in the wild. In *20th USENIX Symposium on Networked Systems Design and Implementation (NSDI 23)*, pages 787–808, 2023.

[10] Jashwant Raj Gunasekaran, Cyan Subhra Mishra, Prashanth Thinakaran, Bikash Sharma, Mahmut Taylan Kandemir, and Chita R Das. Cocktail: A multidimensional optimization for model serving in cloud. In *USENIX NSDI*, pages 1041–1057, 2022.

[11] Sohaib Ahmad, Hui Guan, Brian D Friedman, Thomas Williams, Ramesh K Sitaraman, and Thomas Woo. Proteus: A high-throughput inference-serving system with accuracy scaling. In *Proceedings of the 29th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 1*, pages 318–334, 2024.

[12] Fei Xu, Jianian Xu, Jiabin Chen, Li Chen, Ruitao Shang, Zhi Zhou, and Fangming Liu. igniter: Interference-aware gpu resource provisioning for predictable dnn inference in the cloud. *IEEE Transactions on Parallel and Distributed Systems*, 34(3):812–827, 2022.

[13] Chengliang Zhang, Minchen Yu, Wei Wang, and Feng Yan. Enabling cost-effective, slo-aware machine learning inference serving on public cloud. *IEEE Transactions on Cloud Computing*, 10(3):1765–1779, 2020.

[14] Pin Gao, Lingfan Yu, Yongwei Wu, and Jinyang Li. Low latency rnn inference with cellular batching. In *Proceedings of the Thirteenth EuroSys Conference*, pages 1–15, 2018.

[15] Ziyang Zhang, Yang Zhao, and Jie Liu. Octopus: Slo-aware progressive inference serving via deep reinforcement learning in multi-tenant edge cluster. In *International Conference on Service-Oriented Computing*, pages 242–258. Springer, 2023.

[16] Zhengxu Xia, Yitian Hao, Jun Duan, Chen Wang, and Junchen Jiang. Towards optimal preemptive gpu time-sharing for edge model serving. In *Proceedings of the 9th International Workshop on Container Technologies and Container Clouds*, pages 13–18, 2023.

[17] Weiguang Pang, Xiantong Luo, Kailun Chen, Dong Ji, Lei Qiao, and Wang Yi. Efficient cuda stream management for multi-dnn real-time inference on embedded gpus. *Journal of Systems Architecture*, 139:102888, 2023.

[18] Tongxuan Liu, Tao Peng, Peijun Yang, Xiaoyang Zhao, Xiusheng Lu, Weizhe Huang, Zirui Liu, Xiaoyu Chen, Zhiwei Liang, Jun Xiong, Donghe Jin, Minchao Zhang, Jinrong Guo, Yingxu Deng, Xu Zhang, Xianzhe Dong, Siqi Wang, Siyu Wu, Yu Wu, Zihan Tang, Yuting Zeng, Yanshu Wang, Jinguang Liu, Meng Kang, Menxin Li, Yunlong Wang, Yiming Liu, Xiaolong Ma, Yifan Wang, Yichen Zhang, Jinrun Yin, Keyang Zheng, Jiawei Yin, Jun Zhang, Ziyue Wang, Xiaobo Lin, Liangyu Liu, Liwei Lan, Yang Liu, Chunhua Peng, Han Liu, Songcheng Ren, Xuezhu Wang, Yunheng Shen, Yi Wang, Guyue Liu, Yitao Hu, Hui Chen, Tong Yang, Hailong Yang, Jing Li, Guiguang Ding, and Ke Zhang. xllm technical report, 2026.

[19] Fuxun Yu, Shawn Bray, Di Wang, Longfei Shangguan, Xulong Tang, Chenchen Liu, and Xiang Chen. Automated runtime-aware scheduling for multi-tenant dnn inference on gpu. In *2021 IEEE/ACM International Conference On Computer Aided Design (ICCAD)*, pages 1–9. IEEE, 2021.

[20] Jeffrey Dean and Luiz André Barroso. The tail at scale. *Communications of the ACM*, 56(2):74–80, 2013.

[21] Seah Kim, Hyoukjun Kwon, Jinook Song, Jihyuck Jo, Yu-Hsin Chen, Liangzhen Lai, and Vikas Chandra. Dream: A dynamic scheduler for dynamic real-time multi-model ml workloads. In *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 4*, pages 73–86, 2023.

[22] Daniel Crankshaw, Xin Wang, Giulio Zhou, Michael J Franklin, Joseph E Gonzalez, and Ion Stoica. Clipper: A low-latency online prediction serving system. In *NSDI*, volume 17, pages 613–627, 2017.

[23] 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 *Proceedings of the 18th ACM/IFIP/USENIX middleware conference*, pages 109–120, 2017.

[24] Zhixin Zhao, Yitao Hu, Guotao Yang, Ziqi Gong, Chen Shen, Laiping Zhao, Wenxin Li, Xiulong Liu, and Wenyu Qu. Slopt: Serving real-time inference pipeline with strict latency constraint. *IEEE Transactions on Computers*, 74(4):1431–1445, 2025.

[25] Zhixin Zhao, Yitao Hu, Ziqi Gong, Guotao Yang, Wenxin Li, Xiulong Liu, Keqiu Li, and Hao Wang. Harpagon: Minimizing dnn serving cost via efficient dispatching, scheduling and splitting. In *IEEE INFOCOM 2025-IEEE Conference on Computer Communications*, pages 1–10. IEEE, 2025.

[26] Yaosheng Fu, Evgeny Bolotin, Aamer Jaleel, Gal Dalal, Shie Mannor, Jacob Subag, Noam Korem, Michael Behar, and David Nellans. Autoscratch: Ml-optimized cache management for inference-oriented gpus. *Proceedings of Machine Learning and Systems*, 5:495–512, 2023.

[27] Christopher Olston, Noah Fiedel, Kiril Gorovoy, Jeremiah Harmsen, Li Lao, Fangwei Li, Vinu Rajashekhar, Sukriti Ramesh, and Jordan Soyke. Tensorflow-serving: Flexible, high-performance ml serving. *arXiv preprint arXiv:1712.06139*, 2017.

[28] 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.

[29] Francisco Romero, Qian Li, Neeraja J Yadwadkar, and Christos Kozyrakis. INFaaS: Automated model-less inference serving. In *USENIX Annual Technical Conference*, pages 397–411, 2021.

[30] Peifeng Yu, Yuqing Qiu, Xin Jin, and Mosharaf Chowdhury. Orloj: Predictably serving unpredictable dnns. *arXiv preprint arXiv:2209.00159*, 2022.

[31] Yanan Yang, Laiping Zhao, Yiming Li, Huanyu Zhang, Jie Li, Mingyang Zhao, Xingzhen Chen, and Keqiu Li. INFless: a native serverless system for low-latency, high-throughput inference. In *Proceedings of the 27th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 768–781, 2022.

[32] 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.

[33] Adam Paszke, Sam Gross, Francisco Massa, Adam Lerer, James Bradbury, Gregory Chanan, Trevor Killeen, Zeming Lin, Natalia Gimelshein, Luca Antiga, et al. PyTorch: An imperative style, high-performance deep learning library. *Advances in neural information processing systems*, 32, 2019.

[34] Yifan Zeng, Ruiting Zhou, Lei Jiao, and Renli Zhang. Online scheduling of edge multiple-model inference with dag structure and retraining. In *IEEE INFOCOM 2025-IEEE Conference on Computer Communications*, pages 1–10. IEEE, 2025.

[35] David Mills, Jim Martin, Jack Burbank, and William Kasch. Network time protocol version 4: Protocol and algorithms specification. Technical report, University of Delaware, 2010.

[36] Nabajeet Barman, Saman Zadtootaghaj, Steven Schmidt, Maria G Martini, and Sebastian Möller. Gamingvideoset: a dataset for gaming video streaming applications. In *2018 16th Annual Workshop on Network and Systems Support for Games (NetGames)*, pages 1–6. IEEE, 2018.

[37] I. O. de Oliveira, R. Laroca, D. Menotti, K. V. O. Fonseca, and R. Minetto. Vehicle-Rear: A new dataset to explore feature fusion for vehicle identification using convolutional neural networks. *IEEE Access*, 9:101065–101077, 2021.

[38] YouTube. Youtube live streaming. https://www.youtube.com/howyoutubeworks/product-features/live/, 2024.

[39] Twitch. Twitch. https://www.twitch.tv/, 2024.

[40] Guido Urdaneta, Guillaume Pierre, and Maarten Van Steen. Wikipedia workload analysis for decentralized hosting. *Computer Networks*, 53(11):1830–1845, 2009.

[41] Abeer Abdel Khaleq and Ilkyeun Ra. Cloud-based disaster management as a service: A microservice approach for hurricane twitter data analysis. In *2018 IEEE Global Humanitarian Technology Conference (GHTC)*, pages 1–8. IEEE, 2018.

[42] 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 *Proceedings of the 2020 USENIX Conference on Usenix Annual Technical Conference*, USENIX ATC'20, USA, 2020. USENIX Association.

[43] Foteini Strati, Xianzhe Ma, and Ana Klimovic. Orion: Interference-aware, fine-grained gpu sharing for ml applications. In *Proceedings of the Nineteenth European Conference on Computer Systems*, pages 1075–1092, 2024.

[44] Hao Zhou, Ming Chen, Qian Lin, Yong Wang, Xiaobin She, Sifan Liu, Rui Gu, Beng Chin Ooi, and Junfeng Yang. Overload control for scaling wechat microservices. In *Proceedings of the ACM Symposium on Cloud Computing*, pages 149–161, 2018.

[45] Chengzhi Lu, Huanle Xu, Yudan Li, Wenyan Chen, Kejiang Ye, and Chengzhong Xu. Smiless: Serving dag-based inference with dynamic invocations under serverless computing. In *SC24: International Conference for High Performance Computing, Networking, Storage and Analysis*, pages 1–17. IEEE, 2024.

[46] Vivek M Bhasi, Jashwant Raj Gunasekaran, Prashanth Thinakaran, Cyan Subhra Mishra, Mahmut Taylan Kandemir, and Chita Das. Kraken: Adaptive container provisioning for deploying dynamic dags in serverless platforms. In *Proceedings of the ACM Symposium on Cloud Computing*, pages 153–167, 2021.

[47] Nilanjan Daw, Umesh Bellur, and Purushottam Kulkarni. Xanadu: Mitigating cascading cold starts in serverless function chain deployments. In *Proceedings of the 21st International Middleware Conference*, pages 356–370, 2020.

[48] Jovan Stojkovic, Tianyin Xu, Hubertus Franke, and Josep Torrellas. Specfaas: Accelerating serverless applications with speculative function execution. In *2023 IEEE International Symposium on High-Performance Computer Architecture (HPCA)*, pages 814–827. IEEE, 2023.

[49] Inho Cho, Ahmed Saeed, Joshua Fried, Seo Jin Park, Mohammad Alizadeh, and Adam Belay. Overload control for {μs-scale}{RPCs} with breakwater. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*, pages 299–314, 2020.

[50] Matt Welsh and David Culler. Overload management as a fundamental service design primitive. In *Proceedings of the 10th workshop on ACM SIGOPS European workshop*, pages 63–69, 2002.

[51] Matt Welsh, David Culler, and Eric Brewer. Seda: An architecture for well-conditioned, scalable internet services. *ACM SIGOPS operating systems review*, 35(5):230–243, 2001.

[52] Mingzhe Hao, Huaicheng Li, Michael Hao Tong, Chrisma Pakha, Riza O Suminto, Cesar A Stuardo, Andrew A Chien, and Haryadi S Gunawi. Mittos: Supporting millisecond tail tolerance with fast rejecting slo-aware os interface. In *Proceedings of the 26th symposium on operating systems principles*, pages 168–183, 2017.

[53] Inho Cho, Ahmed Saeed, Seo Jin Park, Mohammad Alizadeh, and Adam Belay. Protego: Overload control for applications with unpredictable lock contention. In *20th USENIX Symposium on Networked Systems Design and Implementation (NSDI 23)*, pages 725–738, 2023.

[54] Yanqi Zhang, Weizhe Hua, Zhuangzhuang Zhou, G Edward Suh, and Christina Delimitrou. Sinan: Ml-based and qos-aware resource management for cloud microservices. In *Proceedings of the 26th ACM international conference on architectural support for programming languages and operating systems*, pages 167–181, 2021.

[55] Yu Gan, Mingyu Liang, Sundar Dev, David Lo, and Christina Delimitrou. Sage: practical and scalable ml-driven performance debugging in microservices. In *Proceedings of the 26th ACM International Conference on Architectural Support for Programming Languages and Operating*

Systems, pages 135–151, 2021.

[56] Zibo Wang, Pinghe Li, Chieh-Jan Mike Liang, Feng Wu, and Francis Y Yan. Autothrottle: A practical {Bi-Level} approach to resource management for {SLO-Targeted} microservices. In *21st USENIX Symposium on Networked Systems Design and Implementation (NSDI 24)*, pages 149–165, 2024.

[57] Ram Srivatsa Kannan, Lavanya Subramanian, Ashwin Raju, Jeongseob Ahn, Jason Mars, and Lingjia Tang. Grandslam: Guaranteeing slas for jobs in microservices execution frameworks. In *Proceedings of the Fourteenth EuroSys Conference 2019*, pages 1–16, 2019.

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

[59] LangChain AI. LangChain: A framework for building LLM-powered applications. https://github.com/langchain-ai/langchain, 2025. GitHub repository, supports chaining together models, embeddings, vector stores, and tools :contentReference[oaicite:0]index=0.

[60] Zhilin Yang, Peng Qi, Saizheng Zhang, Yoshua Bengio, William W Cohen, Ruslan Salakhutdinov, and Christopher D Manning. Hotpotqa: A dataset for diverse, explainable multi-hop question answering. *arXiv preprint arXiv:1809.09600*, 2018.

[61] Aaron Grattafiori, Abhimanyu Dubey, Abhinav Jauhri, Abhinav Pandey, Abhishek Kadian, Ahmad Al-Dahle, Aiesha Letman, Akhil Mathur, Alan Schelten, Alex Vaughan, et al. The llama 3 herd of models. *arXiv preprint arXiv:2407.21783*, 2024.

[62] Matthijs Douze, Alexandr Guzhva, Chengqi Deng, Jeff Johnson, Gergely Szilvasy, Pierre-Emmanuel Mazaré, Maria Lomeli, Lucas Hosseini, and Hervé Jégou. The faiss library. 2024.

[63] Tavily. Tavily: The Web Access Layer for AI Agents. https://www.tavily.com/, 2025. Official website, provides APIs for real-time search, extraction, crawling :contentReference[oaicite:0]index=0.

[64] Junguk Cho, Diman Zad Tootaghaj, Lianjie Cao, and Puneet Sharma. Sla-driven ml inference framework for clouds with heterogeneous accelerators. *Proceedings of Machine Learning and Systems*, 4:20–32, 2022.

[65] Jing Wu, Lin Wang, Qirui Jin, and Fangming Liu. Graft: Efficient inference serving for hybrid deep learning with slo guarantees via dnn re-alignment. *IEEE Transactions on Parallel and Distributed Systems*, 2023.

[66] Yang Bai, Lixing Chen, Mohamed Abdel-Mottaleb, and Jie Xu. Automated ensemble for deep learning inference on edge computing platforms. *IEEE internet of things journal*, 9(6):4202–4213, 2021.

[67] Jinwoo Jeong, Seungsu Baek, and Jeongseob Ahn. Fast and efficient model serving using multi-gpus with direct-host-access. In *Proceedings of the Eighteenth European Conference on Computer Systems*, pages 249–265, 2023.

[68] Tim Kaler, Nickolas Stathas, Anne Ouyang, Alexandros-Stavros Iliopoulos, Tao Schardl, Charles E Leiserson, and Jie Chen. Accelerating training and inference of graph neural networks with fast sampling and pipelining. *Proceedings of Machine Learning and Systems*, 4:172–189, 2022.

[69] Aaron Harlap, Andrew Chung, Alexey Tumanov, Gregory R. Ganger, and Phillip B. Gibbons. Tributary: Spot-dancing for elastic services with latency slos. In *Proceedings of the 2018 USENIX Conference on Usenix Annual Technical Conference*, USENIX ATC '18, page 1–13, USA, 2018.

[70] Yunfan Gao, Yun Xiong, Xinyu Gao, Kangxiang Jia, Jinliu Pan, Yuxi Bi, Yixin Dai, Jiawei Sun, Haofen Wang, and Haofen Wang. Retrieval-augmented generation for large language models: A survey. *arXiv preprint arXiv:2312.10997*, 2:1, 2023.

[71] Haoran Qiu, Weichao Mao, Archit Patke, Shengkun Cui, Saurabh Jha, Chen Wang, Hubertus Franke, Zbigniew Kalbarczyk, Tamer Başar, and Ravishankar K Iyer. Power-aware deep learning model serving with {μ-Serve}. In *2024 USENIX Annual Technical Conference (USENIX ATC 24)*, pages 75–93, 2024.

[72] Yunho Jin, Chun-Feng Wu, David Brooks, and Gu-Yeon Wei. s^3: Increasing gpu utilization during generative inference for higher throughput. *Advances in Neural Information Processing Systems*, 36:18015–18027, 2023.