
# Accordion: A malleable pipeline scheduling approach for adaptive SLO-aware inference serving

# Accordion: A malleable pipeline scheduling approach for adaptive SLO-aware inference serving

Pirah Noor Soomro
Chalmers University of Technology
and University of Gothenburg
Gothenburg, Sweden
pirah@chalmers.se

Nikela Papadopoulou
University of Glasgow
Glasgow, Scotland UK
nikela.papadopoulou@glasgow.ac.uk

Miquel Pericàs
Chalmers University of Technology
and University of Gothenburg
Gothenburg, Sweden
miquelp@chalmers.se

## Abstract

With the rising demand for machine learning-based applications, efficient and cost-effective inference serving systems have become imperative. These systems are tasked with meeting customer requirements outlined by Service Level Objectives (SLOs), encompassing model accuracy, response time, and cost considerations. Despite the adoption of proactive scheduling techniques by modern inference serving systems, dynamic factors such as fluctuating query patterns still pose challenges such as delayed response time. To address these, we propose an adaptive solution leveraging SLO-aware scheduling techniques to optimize resource allocation. Our approach aims to minimize the need for additional resources per inference service. By introducing malleable inference pipelines, we enhance flexibility in resource allocation during peak loads by readjusting the resource assignment to processing pipelines to accommodate maximum possible queries dynamically. Our findings indicate that the proposed scheduler effectively utilizes system resources throughout execution while meeting most SLOs (4.2× less SLO violations). We observe an average reduction of 1.6× in the end-to-end latency of query processing, compared to baseline methods. We also demonstrate the impact of dynamically reducing the resources per inference query to accommodate more inference queries in the system. Our solution accommodates 1.4× more queries on average compared to the baselines and achieves 1.6× higher system throughput in terms of queries per second on average.

## CCS Concepts

• **Computing methodologies** → *Machine learning*; • **Software and its engineering** → **Scheduling**.

## Keywords

Inference Serving System, Parallel Pipelines, Malleable Task Scheduling, SLO-Aware scheduling techniques

**ACM Reference Format:**
Pirah Noor Soomro, Nikela Papadopoulou, and Miquel Pericàs. 2025. Accordion: A malleable pipeline scheduling approach for adaptive SLO-aware inference serving. In *22nd ACM International Conference on Computing Frontiers (CF '25), May 28–30, 2025, Cagliari, Italy.* ACM, New York, NY, USA, 9 pages. https://doi.org/10.1145/3719276.3725190


*CF '25, Cagliari, Italy*

ACM ISBN 979-8-4007-1528-0/25/05
https://doi.org/10.1145/3719276.3725190

## 1 Introduction

Applications relying on machine learning inference are growing in number and expected to keep increasing with time [22, 23, 29, 34, 41]. Indicatively, Meta (formerly Facebook) reported that it serves over trillions of inference requests each day [17], and the estimates are, up to 90% of the AI resources in production datacenters are consumed by inference queries [8, 19]. Businesses that leverage machine learning services for their products not only require quick and accurate inference responses but also at a minimal cost. Inference serving systems provide dedicated infrastructure for inference demand, usually operating under strict Service Level Objectives (SLOs) that define acceptable response times (e.g., tail latency or throughput) [15]. Achieving these SLOs can be costly, as specialized ML hardware [24] is often required to meet the requirements under bursts. However, static solutions for bursty workloads necessitate provisioning additional resources, driving up operating costs. A more effective alternative is to dynamically adjust inference resources to make optimal use of the existing hardware, thereby reducing costs without compromising performance [32].

Modern inference serving systems implement a wide variety of techniques to assign resources proactively, in order to satisfy SLOs and maintain high resource utilization. Popular techniques are multi-tenancy [15, 28], i.e. co-locating inference models on shared resources; adaptive batching [9, 15, 33] - dynamically adjusting the batch size of the inference model; model selection [18, 33] which refers to selecting amongst different variants of a model; and accuracy scaling [7], where model variants are selected based on the accuracy of the model.

Many factors can lead to SLO violations, resulting in a deviation from targeted performance. These factors include (i) a variable pattern of incoming inference queries in terms of batch size and arrival times [4], and (ii) a change in the execution environment due to co-located applications, which requires a reassessment of the available resources [39, 40]. In case of SLO violations, modern inference serving systems may revise the scheduling decisions to find a new resource allocation configuration, and migrate the application to a new set of resources, which can result in adding migration overhead and an increase in the amount of resources to satisfy SLOs under new conditions [25, 36]. Alternatively, they may drop/defer the query if the available resources do not suffice [42]. These techniques lead to resource overprovisioning and/or increased operational costs. For example, the inference query load decreases after certain peak hours. To avoid resource overprovisioning, Proteus [7] performs accuracy scaling, thereby reducing the resources of an individual model during peak time, sacrificing some accuracy. In this paper, we adopt an alternative approach by dynamically readjusting

the layer-to-resource assignment of a model while maintaining performance as required by the SLO. Our method mitigates the need to increase resources during peak load and improves the overall throughput of the inference serving system. We assume pipelined parallelism, treating the model as an inference pipeline composed of stages that encapsulate consecutive model layers. Layers within a stage are implemented in a data-parallel manner, allowing each stage to be treated as a unique schedulable and malleable task, i.e. a set of resources can be assigned/reassigned to a task, and layers can be added or removed from a task. To the best of our knowledge, this strategy is not used in state-of-the-art inference serving systems [7, 15, 33], and is orthogonal to existing inference serving schedulers [7, 15, 18, 33], therefore, it can be used to prevent unnecessary resource overprovisioning and better utilize existing resources.

Our proposed solution adapts to fluctuating loads by leveraging malleable resource allocation to models in the form of malleable inference pipelines. Additionally, we implement an adaptive SLO-aware scheduling strategy for these malleable pipelines. If a new query arrives and there are no resources to allocate, the scheduler searches for opportunities to downscale existing pipelines without violating the SLO requirements of any inference request being served. We quantify the impact of using malleability in terms of system throughput, and assess our solution in terms of resource utilization, system occupancy, and end-to-end latency. Finally, we compare our solution against three baseline resource allocation strategies. The first allocates the maximum possible resources per inference query statically, showcasing gains in resource utilization. The second mirrors existing inference-serving systems, assigning SLO-aware resources [25, 36]. The third utilizes query buffers to process similar queries together, favoring peak workloads by forwarding incoming queries to resources serving similar queries [10].

In summary, we make the following contributions:

(1) We propose a scheduler for inference serving systems that leverages malleable pipelines to effectively utilize system resources throughout execution, resulting in 4.2× fewer SLO violations compared to baselines and enhancing the overall processing time for the query stream. Additionally, we observe an average improvement of 1.6× in the end-to-end latency compared to baseline methods.
(2) We demonstrate the impact of dynamically reducing resources per inference query to accommodate more queries, achieving 1.4× more queries on average compared to baselines, and 1.6× higher system throughput in terms of queries per second on average.

## 2 Background

Inference serving systems are instrumental in managing the burgeoning demand for machine learning inference across diverse sectors like healthcare, security, and analytics [22, 23, 29, 34, 41].

**Query load:** The query load in inference serving exhibits several distinct characteristics. The query size fluctuates dynamically according to type of applications, spanning from a single inference query to larger batch sizes [18]. Moreover, inference serving has been reported to follow a diurnal load pattern in arrival queries per second, across datacenters and services [20]. This pattern leads to significant fluctuations in the load between peak and off-peak times, posing challenges in resource provisioning and utilization.

**Allocation of resources to inference queries:** Resource allocation by schedulers in inference serving systems is a pivotal aspect, directly influencing system performance and efficiency. These systems operate under hard cost constraints. Specialized ML hardware is used to achieve interactive latencies [24] which is comparatively expensive to procure and operate, and must thus be used efficiently. Since many inference queries need to be processed simultaneously, inference serving systems co-locate inference models on the shared computing resources, to avoid resource under-utilization, as long as Quality of Service (QoS) is maintained. Various techniques have been proposed for efficient resource allocation, such as multi-tenant inference serving [28], adaptive batching, and model caching [15, 37]. *SLO-aware scheduling* prioritizes tasks based on their latency or throughput requirements, as specified by service level objectives (SLOs) [12–14, 18]. Overall, efficient resource allocation by schedulers is essential for optimizing inference serving systems' performance and ensuring reliable operation under varying workload conditions. *Dynamic workload adjustment* is another key feature, allowing schedulers to adapt resource allocation based on workload characteristics and system conditions [18]. For example, during periods of high demand, more resources may be allocated to critical tasks to prevent latency spikes, while non-critical tasks may receive fewer resources to ensure overall system stability.

**Inference parallelism:** Schedulers allocate computational resources to optimize throughput and meet SLOs, defined as latency or throughput objectives. Techniques such as *parallelism exploration* determine optimal configurations for resource utilization [31], while *model partitioning and pipelining* enable parallel execution by dividing complex models into smaller components [21, 30]. Parallel inference pipelines enhance throughput by leveraging layer-wise parallelism, reducing communication overhead, and minimizing weight copying [11]. The "bind-to-stage" method [26] assigns each pipeline stage to a distinct set of compute resources, hereby referred to as *execution places* avoiding resource sharing. Achieving optimal throughput necessitates balanced pipeline stages; otherwise, throughput may be impeded by pipeline stalls resulting from dependencies among stages. Inference pipelines can be made malleable by readjusting the network layers to form a new pipeline with a new length. This feature can be used to adjust resource allocation during varied workload and SLO requirements.

### 2.1 Motivation

Adapting resource allocation dynamically to handle increasing inference queries while avoiding SLO violations is crucial for optimizing resource utilization. To motivate our work, we conduct a scheduling experiment on an inference server, hosted on a single node with 64 cores, serving queries for predictions with the VGG16 network. We assume the notion of an *Execution Place (EP)* as a set of compute/memory resources, on which we can bind and execute a single stage of a pipeline. In this system, we define 8 execution places, each featuring 8 cores, avoiding overprovisioning of resources. We examine a window of 20 seconds, during which we issue 100 queries of batch size 10, with random arrival times. [4, 16, 33]. We set an SLO of 10*s* as a challenging target under co-location, given the the measured end-to-end latency for VGG16, shown in Table 1. We

evaluate three schedulers: 1) *Malleable Pipelines* dynamically adjust resources, balancing throughput and SLO compliance. 2)*Maximum Pipelines* maximize resource allocation but increase query waiting times. 3) *SLO-Compliant Pipeline*s allocate just enough resources to meet SLOs, leaving resources underutilized during light workloads. Figure1 shows that Malleable Pipelines achieve the best balance, with higher throughput and fewer SLO violations (12%) compared to Maximum (38%) and SLO-Compliant Pipelines (45%). The Maximum pipelines scheduler tends to use more resources than needed, resulting into longer waiting times for the queries, which could be accommodated earlier during the execution. On the other hand, the SLO-compliant scheduler cause some EPs to remain idle when there is less workload. A malleable pipeline approach is about finding the balance between the other two types of scheduling, reducing the SLO violations and efficiently utilizing resources.

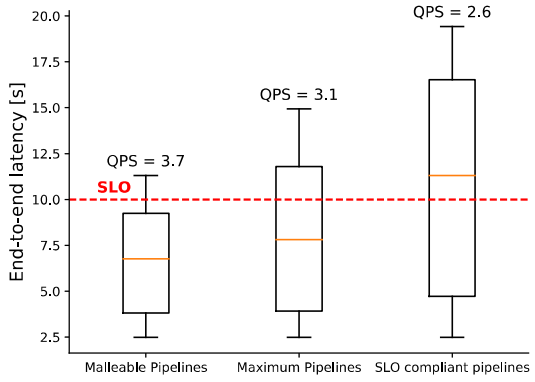



**Figure 1: End-to-end latency and throughput (Queries Per Second, QPS) for VGG16 pipelines under a 100-query load on a 64-core system, evaluated using three scheduling policies: *Malleable Pipelines, Maximum Pipelines* (with 8 cores per EP), and *SLO-Compliant Pipelines.***

## 3 Methodology

### 3.1 System Overview

To simulate a real-world inference serving system and test our proposed approach, we design a system to perform scheduling of inference pipelines, supporting malleability. We present an overview of the system in operation in Figure 2.

Inference serving systems consist of multiple compute units organized across several large servers. In our evaluation, we specifically focus on a single server containing multiple compute units called execution places (EPs). Each EP is a multi-core processing unit, and several EPs together form a multi-chip architecture. We evaluate all scheduling scenarios under the assumption that all EPs have uniform, non-contented memory access, and that EPs do not share any cache level.

Within this system, all inference models are executed using pipeline parallelism. Each workload assigned to an EP represents a pipeline stage of an inference model. For example, an inference model with eight pipeline stages would require eight EPs. Each pipeline stage is composed of several layers of the model. Since each EP includes multiple cores (8 cores per EP in this configuration), layers within a pipeline stage execute in a data-parallel manner. Consequently, a group of consecutive inference layers is allocated to an EP by the runtime system's task assignment mechanism. Depending on how the runtime manages parallel tasks, these allocations are executed as threads or processes.

Another crucial component of the inference serving system is the query load and its associated metadata (see Step 1 in Figure 2). Our simulated input streams consist of various parameters, including the type of inference model to be executed, the batch size of the query, the priority of the query (e.g., Low or High), SLO requirements specified in terms of latency, and the arrival time of the query. In a practical scenario, the query streams would be assembled by a stream pre-processor. The query load, characterized by batch sizes and arrival times, aligns with real-world workload patterns, as detailed in [4, 16, 33], exhibiting diurnal peaks and arrival rates following a Poisson distribution [18]. In this context, the batch size refers to the number of inference instances per request. Our proposed solutions can be used in real-world inference systems by replacing the Query Stream with the actual stream of queries.
Due to its design, the system presents challenges such as real-time query arrivals with varying inference sizes, query priorities, and SLO requirements. As each query constitutes a distinct workload, it requires the allocation of separate resources. The arrival times and batch sizes keep changing across the query stream. We consider that these factors are randomized and unpredictable; this hinders the pre-scheduling of resources, underscoring the need for a dynamic resource allocator to efficiently handle incoming workloads.
Before the scheduler comes into action, the following components are activated. First, an input stream is created, providing details of the query workload and associated metadata. Second, an offline-generated mapping table is constructed to link inference models to pipelines with various configurations. Each pipeline configuration specifies the depth of the pipeline and how layers are distributed across its stages. These configurations are derived using the state-of-the-art layer-to-pipeline mapping method proposed in [38].

A pipeline configuration details the sequential assignment of network layers to pipeline stages. For example, the configuration [2, 4, 4, 3] corresponds to a network with 13 layers, assigning the first 2 layers to stage 1, the next 4 layers to stage 2, and so on. Configurations are ranked based on performance, enabling the scheduler to select optimal pipeline arrangements that meet the SLO requirements for the specific query. After these preprocessing steps, the online scheduler dynamically allocates resources to queries by using these ranked configurations while considering the current system occupancy.

### 3.2 Resource Allocation and Priority-Based Inference Queue Management

To evaluate different resource allocation strategies for inference queries, we devise a scheduler within our simulated environment. Incoming inference queries are categorized into two queues: High and Low priority (see Step 1 in Figure 2), inspired by production-level model serving systems such as NVIDIA Triton Inference Server [3] defining model configurations. The queues assist in a) prioritizing resource allocation and b) avoiding dropping high-priority

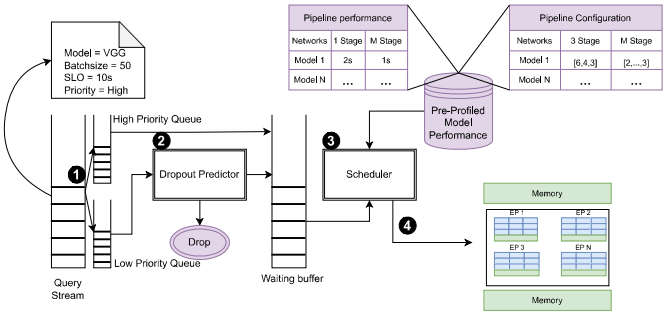



**Figure 2: System Overview. Steps 1 and 2 generate a workload for a simulated inference system. Step 3 performs the resource assignment and scheduling. Finally, the query is executed in step 4**

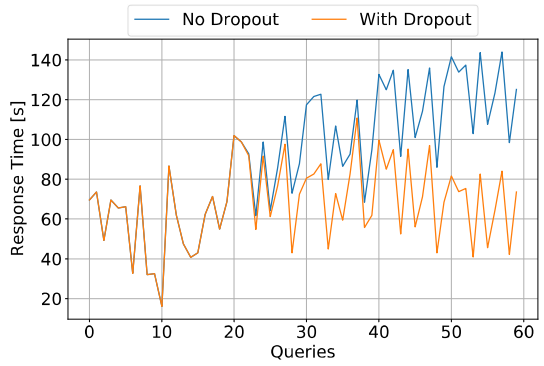



**Figure 3: Impact of dropout predictor on the processing time of high priority queries**

queries. The scheduler periodically checks for new queries and allocates the resources. Resources are first allocated to high-priority queries, and then assigned to low-priority queries. To mitigate waiting times for high-priority queues, the scheduler integrates a predictor (see Step 2 in Figure 2). Similar to [7, 37], we drop low-priority queries based on the outcome of the predictor. The predictor estimates waiting and processing times based on pre-profiled performance data and current system conditions. If the predicted end-to-end latency (i.e waiting time + processing time) for a low-priority query exceeds a specified time window, the scheduler discards the query or directs it to other available servers, indicating that the inference server cannot process it within the required timeframe.

**Dropout Predictor:** Algorithm 1 outlines the process of predicting the execution time of inference queries. The algorithm takes two inputs. The first input is the inference query $Q$, i.e. the model type, batch size, SLO target, and query priority. The second input is $EP\_times$, a list indicating the finish times of queries executing on individual EPs. These times accumulate with new finish times, as new queries are projected to execute on respective execution places. Initially, the algorithm calculates the required resources and processing time for the query, based on SLO requirements, utilizing pre-profiled execution times of models on specific resource sizes, such as the latency of the model on a pipeline of length $N$. The formation of pipelines for a given length is performed offline using Shisha [38], a layer-to-pipeline mapping algorithm. Lines 4 and 5 identify the $N$ EPs with the earlierst finish times. The EP with the latest finish time among the candidate EPs determines the waiting time of the current query. Line 6 calculates the total processing time ($E_t$) of the query. If the processing time of a low-priority query exceeds the allowed time window, the query is dropped; otherwise, the finish times of candidate EPs are updated by accumulating the processing time of the current query.

**Algorithm 1** Dropout Predictor

1: **Input:** $Q$ (inference query), $EP\_times$ (execution times of EPs)
2: $N \leftarrow$ calculate_required_EPs($Q$)
3: $P_t \leftarrow$ calculate_execution_time($Q, N$)
4: $candidate\_EPs \leftarrow$ get_min_finish_time_EPs($N$)
5: $W_t \leftarrow$ max_finish_time($candidate\_EPs$)
6: $E_t \leftarrow W_t + P_t$
7: **if** ($E_t >$ TIME_WINDOW and priority($Q$) == LOW) **then**
8: Drop_query()
9: **else**
10: update_EP_times(candidate_EPs, $E_t$)
11: **end if**

We use the dropout predictor with our scheduling technique, Accordion, and all baselines. Figure 3 shows the processing time of high-priority queries from two periodically bursty workloads and scheduling with Accordion. The results show that $\sim$ 82% of the queries on average have a better processing time compared to when the dropout predictor is not used.

## 3.3 Accordion: An SLO-Aware Malleable Pipeline Resource Allocator

This paper presents a novel approach that capitalizes on the malleability feature of inference pipelines, namely dynamically allocating resources to adapt to fluctuating workloads while preserving the SLO requirements of high-priority queues. Our proposed resource allocation strategy, outlined in Algorithm 2, starts by computing the best possible resource allocation ($Best_{assign}$) for a given query, $Q$. Initially, the algorithm checks for available free EPs to accommodate this allocation. If there are insufficient free EPs, it evaluates whether the available resource capacity can still meet the SLO. As a fallback, it reviews the assignments of currently active pipelines, retrieving EPs from them without violating the SLO of their respective queries ($Q_s$), as depicted in lines 16 through 34. If no EPs can be retrieved from existing queries, the query must wait until resources become available. The procedure of fetching EPs from active queries utilizes pre-profiled data to check the latency of a model w.r.t. to pipeline depths under consideration. Once a decision is made to fetch EPs from a currently active query, the pipeline is rescheduled on the new set of EPs and the query processing is resumed. We do not re-allocate EPs from scratch to active queries, as this would induce additional cost to migrate the model to a new set of EPs. Instead, we remove the number of EPs that need to be fetched. This incurs the cost of reconfiguring the pipeline of the active query on fewer EPs, i.e. moving a few layers between EPs, according to the new pipeline configuration. For a ResNet layer, the weights are typically in the order of $100KB$ (potentially slightly more or

less). Assuming a NUMA-like link, we estimate the time to move the weights to $5ms$. Because this latency is very small compared to the end-to-end latency of the inference pipeline (which ranges from 0.2 to 4.8 seconds as per Table 1), its impact is negligible and therefore not taken into account. Furthermore, we observed in our experiments that the maximum frequency of readjustments is only 0.9% of the total number of inference passes within a query stream. Upon readjusting the pipeline, the query is resumed from the inference instance which was being processed. For example, a query pipeline with an inference size of B is readjusted after processing X (X < B) inference instances. Then the new configuration pipeline will resume from inference instance number X+1.

**Algorithm 2** Adaptive SLO aware scheduler

**Input:** $Q$ (inference query), $EP\_status$
$M$ = get_Model_type($Q$);
$Best_{assign}$ = get_best_assignment($Q$)
$EP_{set}$ = get_optimal_EP_requirement($Best_{assign}$)
$Free_{EPs}$ = get_Free_EPs($EP\_status$)
**if** $Free_{EPs} \neq 0$ **and** SATISFY_SLO($Q$, $Free_{EPs}$) **then**
    $EP_{set} = Free_{EPs}$ ▷ Assign less but enough EPs
    Assign($EP_{set}$, $Q$)
    Add($Q$, active_queries)
**else**
    **if** $Free_{EPs} == 0$ **then**
        $EP_{set}$ = get_minimal_EP_requirement($Q$)
        Fetch_EPs($EP_{set}$)
    **end if**
**end if**
**procedure** FETCH_EPs($EP_{set}$)
    **for** $Q_s$ in active_queries **do**
        $depth$ = get_pipeline_depth($Q_s$)
        **while** ($depth > 1$) **do**
            **if** (confirm_SLO($Q_s$, $depth$)) **then**
                reduce_pipeline($Q_s$, $depth - 1$)
                Free_EP($EP\_status$, 1);
                $depth - -$;
            **else**
                break;
            **end if**
        **end while**
    **end for**
    **if** (get_Free_EPs($EP\_status$) == $EP_{set}$) **then**
        Assign($EP_{set}$, $Q$)
        Add($Q$, active_queries)
    **else**
        wait($Q$)
    **end if**
**end procedure**

# 4 Evaluation

## 4.1 Experimental setup

We simulate the inference serving system and query patterns by generating a database of execution times for individual neural network layers. To construct this database, we execute neural network layers on an Intel Alder Lake [35] which contains 8 Performance cores (P-cores). We record the execution times for all the layers on a single EP (8 P-cores). The database is later used by the simulator to calculate execution times for pipeline stages. We consider layers executed one after another within a pipeline stage so the execution time of a stage is calculated as the sum of execution times of each layer. The end-to-end latency and speedups of the networks recorded on EPs for 1 to 8 stage pipelines are shown in Table 1. Since the measured performance on Alder Lake is on the order of seconds, our evaluation Service Level Objectives (SLOs) are also expressed in seconds. However, applying advanced optimization techniques—such as vectorization, matrix units, or GPUs—could reduce latencies to more acceptable levels.

Our simulated inference serving system comprises 64 EPs, as discussed in Section 3.1, with each set of 8 P-cores representing a single EP within the system (a total of 512 cores). To simplify experimentation, we cap the highest possible allocation per query at a set of 8 EPs (64 P-cores), a decision based on the observation that representative models do not scale beyond 8 P-cores.

For the inference queries, we employ the Keras [2] implementation of three representative convolutional neural networks: VGG16, ResNet50, and ResNet152. Our simulated query patterns are inspired by previous literature [4, 16, 33], , employing periodic arrival patterns typical of real-world workloads observed over intervals up to 10 minutes. Additionally, the inference query sizes follow a log-normal distribution.

## 4.2 Baseline Approaches

The scheduler periodically monitors the incoming query stream, and allocates execution places (EPs) to the pending inference queries awaiting processing. We examine various resource assignment approaches commonly used in state-of-the-art inference serving systems, together with our proposed approach, Accordion.

(1) **SLO-Conforming:** Schedulers such as those in Hercules [25] and DeepRecSys [18] allocate resources strictly to meet query SLOs. When the system load increases dynamically, resources are reassessed and scaled accordingly, potentially adding servers to maintain the required service levels. We adopt this baseline by selecting a pipeline configuration that satisfies the given SLO and allocating EPs accordingly.
(2) **Fastest resource allocation:** Inspired by the method described in [36], this strategy allocates the maximum feasible resources to incoming queries, achieving the fastest possible query completion, enhancing overall system throughput by rapidly processing queries. We implement this by assigning the fastest pipeline configuration and corresponding EPs to each query, which typically leads to using more EPs per query compared to the SLO-Conforming approach.
(3) **Buffer-Based Solution - BBS :** As presented in [10], this approach buffers incoming queries when resources are occupied by prior inference requests. If an incoming query matches a query (model type and SLO) already processing, it is queued and processed sequentially on the existing EP set, optimizing resource reuse.
(4) **1-Stage:** This conservative strategy allocates exactly one EP per query, resulting in a simple, single-stage inference pipeline.

## 4.3 Evaluation Metrics

To analyze SLO conformance and resource utilization, we evaluate the end-to-end latency of the inference pipelines scaled to batch size and SLO violations. System occupancy is measured as the percentage of EPs busy at any given time. System throughput is quantified in terms of Queries Per Second (QPS) delivered by the scheduling algorithms for the entire query stream.

**Query Streams:** We generate six sample query patterns to mimic real-world/real-time scenarios discussed in [4, 16, 33]. To simulate

the periodic behavior of incoming queries, we introduce 2 and 5 peaks representing periodically bursty workloads. We present the arrival rate for the two patterns in Figure 4a. The batch sizes of the queries follow a log-normal distribution. The type of models (i.e. VGG16, ResNet50 or ResNet152) and associated priority (i.e. High or Low) are assigned randomly. Query streams 1,2 and 3 contain 2 peaks and average inference sizes (also referred to as batch sizes) of 1,20, and 100 and query streams 4,5,6 contain 5 peaks with average batch sizes of 20, 50, and 70 respectively. These values are selected through multiple trials to select values that saturate the simulated system to its maximum capacity. Query streams 1 and 4 have the lowest workload relative to the system's capacity, indicating periods of lighter traffic. Figure 4b illustrates the execution times of each query, when the inference is executed as a single-stage pipeline. The query streams contain three model types, as mentioned in Table 1.

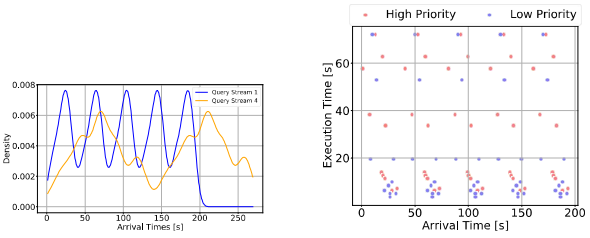
**SLO Targets:** For our analysis, we use the execution times of the 1-Stage pipeline models listed in Table 1. We define three Service Level Objectives (SLOs)—Strict (10%), Moderate (25%), and Relaxed (50%)—based on percentages of the maximum execution time observed across the entire query stream. These distinct SLO settings enable us to evaluate scheduler performance under both stringent and relaxed timing constraints. The rationale behind selecting these SLOs is to establish latency requirements relative to each specific query stream. This approach allows us to evaluate scheduler performance across a spectrum of latency constraints, rather than using a single fixed latency as employed in previous work [15]. We note that we determine the SLOs based on the performance of the 1-Stage pipeline configuration, which is an 8-core (1 EP) data-parallel implementation, however during simulation, the schedulers typically allocate multiple EPs, leading to improved latency. We also highlight that our SLO targets derive from performance data as shown in Figure 4, and do not consider query waiting times, which can introduce additional delays impacting end-to-end latency. The actual speedups achieved might be lower than those anticipated solely from the baseline 1-Stage pipeline latencies used for SLO calculations.

**Table 1: End-to-end Latency and speedup of pipelines comprising of stages 2 to 8 executed on 2 to 8 EPs respectively**

| Model | Latency (s) | Speedup | | |
|---|---|---|---|---|
| | | 2 EPs | 4 EPs | 8 EPs |
| ResNet50 | 1.40 | 1.33 | 2.19 | 3.29 |
| VGG16 | 0.69 | 1.43 | 2.71 | 3.47 |
| ResNet152 | 4.82 | 1.94 | 2.06 | 5.66 |

## 4.4 Evaluating SLO-aware malleability with Accordion

*4.4.1 End-to-End Latency.* To assess the impact of malleability, we first compare Accordion against all the baseline approaches, for all 6 query streams. The results, shown in Figure 5, underscore the impact of malleability in enhancing overall system performance, as all other approaches result in increased waiting times. *Accordion* demonstrates a substantial improvement in completing the query




(a) Query streams arrival rates (b) 1-Stage query stream 2

**Figure 4: Pattern of query streams**

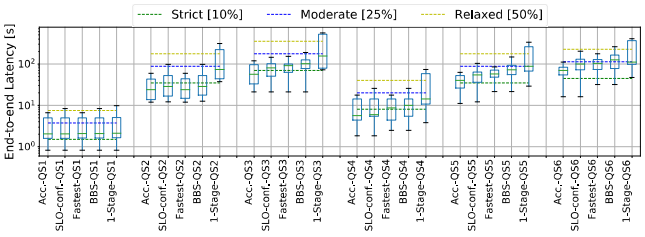



**Figure 5: Comparison of the end-to-end latency of Accordion and baselines, for all query streams. The horizontal lines represent different SLO targets.**

stream. Specifically, it is 1.3× faster on average compared to *Fastest*, *BBS* and *SLO-Conforming*, and 2.2× faster than the *1-Stage* baseline.

*4.4.2 Comparison of SLO-aware scheduling techniques.* The efficiency of Accordion is underscored by its ability to process inference queries with latencies that do not violate the target SLOs. To assess this ability, in Figure 5, we examine the performance of Accordion against the baselines detailed in Section 4.2, for the three different SLO targets. The *SLO-Conforming* technique allocates sufficient resources to queries to meet the SLO targets. However, because of longer waiting times for all queries, it leads to SLO violations. The *Fastest* technique, assisted by offline performance measurements, is designed to ensure SLO conformance and assigns the maximum numbers of EPs to deliver the fastest inference. Nevertheless, as it does not dynamically adjust resource allocation during peak periods, it also results in SLO violations due to longer waiting times. *BBS* also results in SLO violations due to longer waiting times, as queries arriving when no resources are available are buffered within a selected set of EPs, already processing a similar type of queries.

Query stream 1 contains queries with smaller batch sizes and low arrival rate relative to execution times of the queries, which means that enough EPs are free at the arrival time of queries and most queries are processed immediately. This is a best-case scenario and all schedulers perform well with this type of workload. With query streams 2 and 4, all schedulers except *BBS* are able to achieve moderate and strict SLO targets. The increased workload in these streams increases the waiting times with *BBS*. The benefit of malleability is visible in query streams 3, 5, and 6. These streams contain a heavier workload, resulting in constantly busy EPs. In this case, the only way to free up some EPs is through dynamic readjustment of

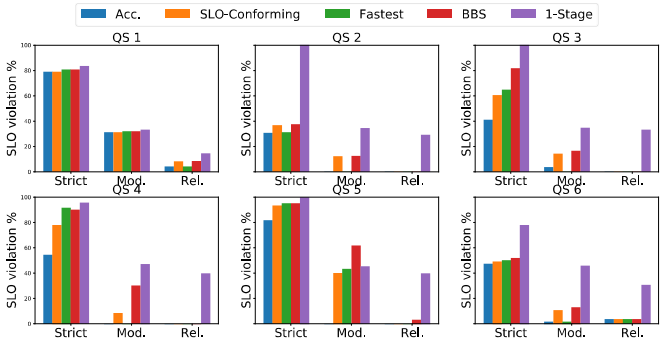



**Figure 6: SLO violations per query stream for Accordion against baseline approaches.**

existing pipelines by assigning EPs to waiting queries, as quickly as EPs can be fetched. This decreases the waiting time of queries and makes it possible to sustain SLO targets. The results show that only *Accordion* is able to achieve the Strict (10%) end-to-end latency target on average, while always sustaining the Moderate (25%) SLO targets, compared to the other baselines.

*4.4.3 SLO Violations.* Although all schedulers except *1-Stage* prioritize SLO conformance, long waiting times due to unavailability of EPs still cause SLO violations. Figure 6 shows the SLO violations for all the scheduling techniques. *Accordion* achieves on average 1.13× fewer SLO violations compared to the baselines under strict SLO targets. Furthermore, it performs on average 4.2× fewer SLO violations under moderate SLO targets. Finally, *Accordion* results in approximately only 2% of SLO violations across all query streams under relaxed SLO targets. This corresponds to 1.7× fewer violations compared to the baseline techniques.

*4.4.4 Malleable resource allocation and system throughput.* Accordion initially allocates higher numbers of resources to inference queries, resulting into higher performance. Subsequently, it dynamically adjusts the resource allocation width of the pipelines within the system whenever there is a need to accommodate additional queries but resources are insufficient. We analyze the effect of downsizing resources by observing the number of active queries in the system at any given time, in Figure 7, for all query streams. *Accordion* demonstrates the capability to accommodate a larger number of queries in the system for execution and manages to complete the workload earlier than the baselines. *Accordion* processes 1.3, 1.6, and 1.5× more queries compared to *SLO-Conforming*, *Fastest*, and *BBS*. *1-Stage* processes the highest number of queries (equal to the maximum number of EPs in the system, i.e 64) at any time interval as this scheme always assigns one EP per request, but sacrifices throughput because of longer processing times. Specifically, for query streams 2, 3, 4, and 6, *1-Stage* is ∼ 200% slower than *Accordion* on average. For query streams 1 and 4, the arrival rate of incoming queries is relative to the execution time of queries, and system resources are free at the time of query arrivals, therefore, we observe no significant difference between the different techniques, except for *1-Stage*. Figure 8 represents the system throughput achieved in

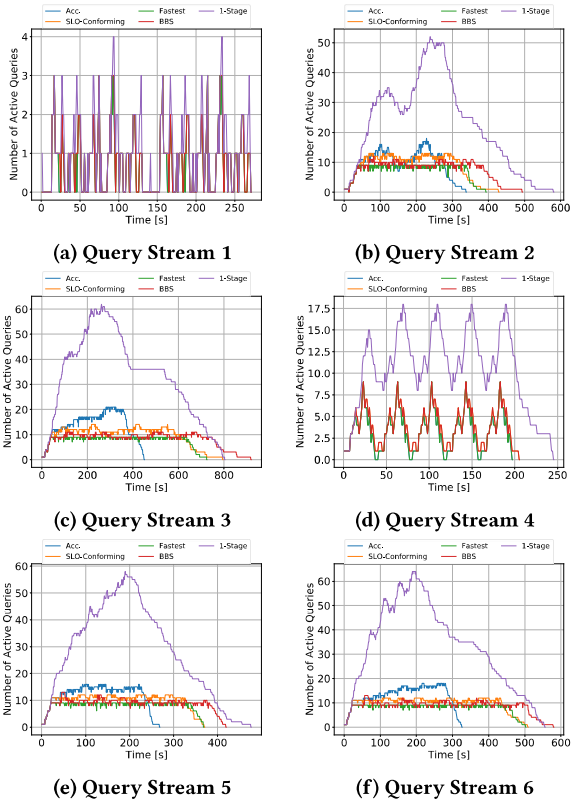



**Figure 7: Timeline of active queries in the system.**

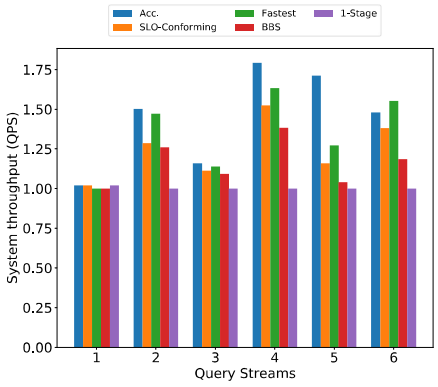
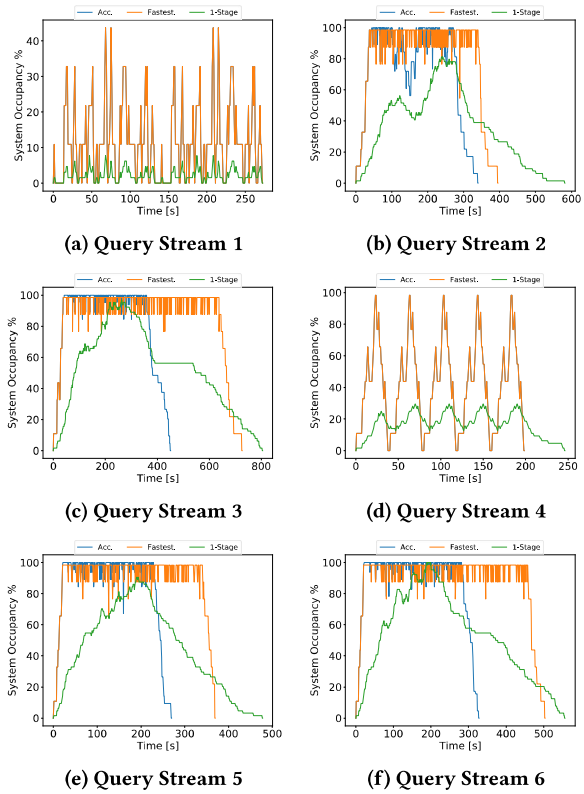



**Figure 8: Normalized system throughput in queries per second (QPS) of Accordion and baselines against the system throughput of 1-Stage baseline**

terms of queries per second (QPS). For query streams 5 and 6, *Accordion* achieves 2× higher throughput normalized to the minimum (*1-Stage*). *Accordion* generally performs better than the other techniques, offering 1.2× higher throughput than the baselines across all query streams.

*4.4.5 System occupancy.* The aim of malleable pipelines is to maximize resource utilization, thereby avoiding the need of adding




(a) Query Stream 1 (b) Query Stream 2

(c) Query Stream 3 (d) Query Stream 4

(e) Query Stream 5 (f) Query Stream 6

**Figure 9: System occupancy over time across query streams**

more compute resources during peak workload times. To assess the ability of Accordion to maintain high system occupancy, we look at the number of resources occupied by the queries during the course of execution of a whole query stream, in Figure 9. It is evident that for query streams 2, 3, 5, and 6, *Accordion* consistently maintains the system occupancy at 100% for ≈ 70% of the time and completes execution earlier than the *Fastest* and *1-Stage* schedulers. While the *Fastest* technique also sustains the system at full capacity for ∼ 82% of the time, it is plagued by longer waiting times and takes 1.6× longer time on average to finish the whole query stream. This is because *Fastest* does not dynamically adjust resources; even if there are slightly fewer available EPs than those required, they will not be used but the query will wait for more EPs to become available. Conversely, the *1-Stage* pipeline technique assigns exactly one EP per query, leaving EPs idle when no incoming or waiting queries are present, resulting in poor resource utilization. *Accordion* and *Fastest* show similar occupancy for query streams 1 and 4, which have low arrival rates and smaller inference batch sizes.

## 5 Related Work

Various inference serving systems are currently employed in practice, offering a spectrum of solutions for SLO-aware resource allocations considering the query workload. Clipper [15], Pretzel[27], and DeepRecSys [18] statically optimize batching and hardware selection for inference serving systems, necessitating developers to specify a variant, manage, and scale model resources as the load varies. INFaaS [33], on the other hand, navigates the variant search space on developers' behalf and dynamically leverages model variants to meet applications' diverse requirements but does not support dynamic resource allocation based on variable query workload. Popular techniques in resource management for inference systems include multi-tenancy, as highlighted in works like [15, 28], which involves hosting multiple inference models on shared computational resources. This approach promotes resource efficiency and cost savings but can lead to challenges such as resource contention and interference between co-located models. Adaptive batching, as discussed in [15, 33], dynamically adjusts the batch size of inference models based on workload characteristics and system conditions. By optimizing resource utilization and reducing inference latency, adaptive batching enhances system responsiveness and efficiency. Model selection techniques [18, 33] involve selecting the most appropriate variant of a model based on application requirements and resource constraints. By selecting models with optimal computational complexity and accuracy levels, model selection optimizes resource allocation and enhances overall system performance. Accuracy scaling strategies, exemplified by [7], dynamically select model variants based on real-time accuracy needs. By leveraging simpler models when high accuracy is not critical, accuracy scaling ensures efficient resource allocation and performance optimization, adapting to changing accuracy requirements in inference tasks. Furthermore, resource allocation approaches typically adopted in [18, 25, 36] utilize offline performance data to match configurations with SLO requirements. To mitigate high query workload, proactive approaches are also adopted as proposed in [10] by leveraging query buffers to accommodate similar types of queries arriving in the future. SageMaker [1], AI Platform [6], and Azure ML [5] offer developers separate online and offline services that autoscale VMs based on query workload.

## 6 Conclusion

In this paper, we harness the inherent malleability feature of inference pipelines to address challenges encountered by inference serving systems during peak workload periods. By dynamically reallocating resources among pipelines, we demonstrate a reduction in the necessity to scale up system resources during peak workload times. Additionally, our proposed technique leads to improved resource utilization. Accordion serves as an additional feature atop a basic inference serving scheduler and can be implemented on servers utilizing any kind of processing unit. We abstract this functionality as execution places, making it adaptable to various hardware configurations.

## Acknowledgments

This work has received funding from the project PRIDE from the Swedish Foundation for Strategic Research with reference number CHI19-0048. This work has also received funding from the European High-Performance Computing Joint Undertaking (JU) under grant agreement No. 956702. The JU receives support from the European Union's Horizon 2020 research and innovation programme and Spain, Sweden, Greece, Italy, France, Germany. This work, in particular, has received funding from the Swedish Research Council under contract 2020-06735_3.

## References


[1] [n. d.]. Amazon SageMaker. https://aws.amazon.com/sagemaker/. Accessed: 2024.

[2] [n. d.]. Keras Documentation. https://keras.io/. Accessed: 2024.

[3] [n. d.]. NVIDIA Triton Inference Server Documentation. https://docs.nvidia.com/deeplearning/triton-inference-server/. Accessed: 2024.

[4] 2018. *Twitter Streaming Traces.* https://archive.org/details/archiveteam-twitter-stream-2018-04

[5] 2024. Azure Machine Learning. https://docs.microsoft.com/en-us/azure/machine-learning/.

[6] 2024. Google Cloud AI Platform. https://cloud.google.com/ai-platform/.

[7] Sohaib Ahmad, Hui Guan, Brian D Friedman, Thomas Williams, Ramesh K Sitaraman, and Thomas Woo. 2024. Proteus: A High-Throughput Inference-Serving System with Accuracy Scaling. (2024).

[8] Sherif Akoush, Andrei Paleyes, Arnaud Van Looveren, and Clive Cox. 2022. Desiderata for next generation of ML model serving. *arXiv preprint arXiv:2210.14665* (2022).

[9] Ahsan Ali, Riccardo Pinciroli, Feng Yan, and Evgenia Smirni. 2020. BATCH: Machine Learning Inference Serving on Serverless Platforms with Adaptive Batching. In *SC20: International Conference for High Performance Computing, Networking, Storage and Analysis*. 1–15.

[10] Ahsan Ali, Riccardo Pinciroli, Feng Yan, and Evgenia Smirni. 2022. Optimizing inference serving on serverless platforms. *Proceedings of the VLDB Endowment* 15, 10 (2022).

[11] Tal Ben-Nun and Torsten Hoefler. 2019. Demystifying parallel and distributed deep learning: An in-depth concurrency analysis. *ACM Computing Surveys (CSUR)* 52, 4 (2019), 1–43.

[12] Quan Chen, Hailong Yang, Minyi Guo, Ram Srivatsa Kannan, Jason Mars, and Lingjia Tang. 2017. Prophet: Precise qos prediction on non-preemptive accelerators to improve utilization in warehouse-scale computers. In *Proceedings of the Twenty-Second International Conference on Architectural Support for Programming Languages and Operating Systems*. 17–32.

[13] Quan Chen, Hailong Yang, Jason Mars, and Lingjia Tang. 2016. Baymax: Qos awareness and increased utilization for non-preemptive accelerators in warehouse scale computers. *ACM SIGPLAN Notices* 51, 4 (2016), 681–696.

[14] Yujeong Choi, Yunseong Kim, and Minsoo Rhu. 2021. Lazy batching: An SLA-aware batching system for cloud machine learning inference. In *2021 IEEE International Symposium on High-Performance Computer Architecture (HPCA)*. IEEE, 493–506.

[15] Daniel Crankshaw, Xin Wang, Guilio Zhou, Michael J Franklin, Joseph E Gonzalez, and Ion Stoica. 2017. Clipper: A {Low-Latency} online prediction serving system. In *14th USENIX Symposium on Networked Systems Design and Implementation (NSDI 17)*. 613–627.

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

[17] Facebook Engineering. [n. d.]. Building Meta's GenAI Infrastructure. https://engineering.fb.com/2024/03/12/data-center-engineering/building-metas-genai-infrastructure/. Accessed on April 22, 2024.

[18] Udit Gupta, Samuel Hsia, Vikram Saraph, Xiaodong Wang, Brandon Reagen, Gu-Yeon Wei, Hsien-Hsin S. Lee, David Brooks, and Carole-Jean Wu. 2020. DeepRecSys: A System for Optimizing End-To-End At-Scale Neural Recommendation Inference. In *2020 ACM/IEEE 47th Annual International Symposium on Computer Architecture (ISCA)*. 982–995. https://doi.org/10.1109/ISCA45697.2020.00084

[19] Udit Gupta, Carole-Jean Wu, Xiaodong Wang, Maxim Naumov, Brandon Reagen, David Brooks, Bradford Cottel, Kim Hazelwood, Mark Hempstead, Bill Jia, Hsien-Hsin S. Lee, Andrey Malevich, Dheevatsa Mudigere, Mikhail Smelyanskiy, Liang Xiong, and Xuan Zhang. 2020. The Architectural Implications of Facebook's DNN-Based Personalized Recommendation. In *2020 IEEE International Symposium on High Performance Computer Architecture (HPCA)*. 488–501. https://doi.org/10.1109/HPCA47549.2020.00047

[20] Kim Hazelwood, Sarah Bird, David Brooks, Soumith Chintala, Utku Diril, Dmytro Dzhulgakov, Mohamed Fawzy, Bill Jia, Yangqing Jia, Aditya Kalro, James Law, Kevin Lee, Jason Lu, Pieter Noordhuis, Misha Smelyanskiy, Liang Xiong, and Xiaodong Wang. 2018. Applied Machine Learning at Facebook: A Datacenter Infrastructure Perspective. In *2018 IEEE International Symposium on High Performance Computer Architecture (HPCA)*. 620–629. https://doi.org/10.1109/HPCA.2018.00059

[21] Zhihao Jia, Matei Zaharia, and Alex Aiken. 2019. Beyond Data and Model Parallelism for Deep Neural Networks. *Proceedings of Machine Learning and Systems* 1 (2019), 1–13.

[22] Fei Jiang, Yong Jiang, Hui Zhi, Yi Dong, Hao Li, Sufeng Ma, Yilong Wang, Qiang Dong, Haipeng Shen, and Yongjun Wang. 2017. Artificial intelligence in healthcare: past, present and future. *Stroke and vascular neurology* 2, 4 (2017).

[23] Junchen Jiang, Ganesh Ananthanarayanan, Peter Bodik, Siddhartha Sen, and Ion Stoica. 2018. Chameleon: scalable adaptation of video analytics. In *Proceedings of the 2018 conference of the ACM special interest group on data communication*. 253–266.

[24] Norman P Jouppi, Cliff Young, Nishant Patil, David Patterson, Gaurav Agrawal, Raminder Bajwa, Sarah Bates, Suresh Bhatia, Nan Boden, Al Borchers, et al. 2017. In-datacenter performance analysis of a tensor processing unit. In *Proceedings of the 44th annual international symposium on computer architecture*. 1–12.

[25] Liu Ke, Udit Gupta, Mark Hempstead, Carole-Jean Wu, Hsien-Hsin S Lee, and Xuan Zhang. 2022. Hercules: Heterogeneity-aware inference serving for at-scale personalized recommendation. In *2022 IEEE International Symposium on High-Performance Computer Architecture (HPCA)*. IEEE, 141–154.

[26] I-Ting Angelina Lee, Charles E Leiserson, Tao B Schardl, Zhunping Zhang, and Jim Sukha. 2015. On-the-fly pipeline parallelism. *ACM Transactions on Parallel Computing (TOPC)* 2, 3 (2015), 1–42.

[27] Yunseong Lee, Alberto Scolari, Byung-Gon Chun, Marco Domenico Santambrogio, Markus Weimer, and Matteo Interlandi. 2018. Pretzel: opening the black box of machine learning prediction serving systems. In *Proceedings of the 13th USENIX Conference on Operating Systems Design and Implementation* (Carlsbad, CA, USA) *(OSDI'18)*. USENIX Association, USA, 611–626.

[28] Matthew LeMay, Shijian Li, and Tian Guo. 2020. Perseus: Characterizing performance and cost of multi-tenant serving for cnn models. In *2020 IEEE International Conference on Cloud Engineering (IC2E)*. IEEE, 66–72.

[29] Fatemehsadat Mireshghallah, Mohammadkazem Taram, Prakash Ramrakhyani, Ali Jalali, Dean Tullsen, and Hadi Esmaeilzadeh. 2020. Shredder: Learning noise distributions to protect inference privacy. In *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*. 3–18.

[30] Deepak Narayanan, Aaron Harlap, Amar Phanishayee, Vivek Seshadri, Nikhil R. Devanur, Gregory R. Ganger, Phillip B. Gibbons, and Matei Zaharia. 2019. PipeDream: generalized pipeline parallelism for DNN training. In *Proceedings of the 27th ACM Symposium on Operating Systems Principles* (Huntsville, Ontario, Canada) *(SOSP '19)*. Association for Computing Machinery, New York, NY, USA, 1–15. https://doi.org/10.1145/3341301.3359646

[31] Nvidia. [n. d.]. *Multi-Process Service.* https://docs.nvidia.com/pdf/CUDA_Multi_Process_Service_Overview.pdf

[32] Miquel Pericàs. 2018. Elastic Places: An Adaptive Resource Manager for Scalable and Portable Performance. *ACM Trans. Archit. Code Optim.* 15, 2, Article 19 (May 2018), 26 pages. https://doi.org/10.1145/3185458

[33] Francisco Romero, Qian Li, Neeraja J Yadwadkar, and Christos Kozyrakis. 2019. INFaaS: A model-less and managed inference serving system. *arXiv preprint arXiv:1905.13348* (2019).

[34] Francisco Romero, Mark Zhao, Neeraja J Yadwadkar, and Christos Kozyrakis. 2021. Llama: A heterogeneous & serverless framework for auto-tuning video analytics pipelines. In *Proceedings of the ACM Symposium on Cloud Computing*. 1–17.

[35] Rotem. et al. 2022. Intel Alder Lake CPU Architectures. *IEEE Micro* 42, 3 (2022), 13–19. https://doi.org/10.1109/MM.2022.3164338

[36] Wonik Seo, Sanghoon Cha, Yeonjae Kim, Jaehyuk Huh, and Jongse Park. 2021. SLO-aware inference scheduler for heterogeneous processors in edge platforms. *ACM Transactions on Architecture and Code Optimization (TACO)* 18, 4 (2021), 1–26.

[37] Haichen Shen, Lequn Chen, Yuchen Jin, Liangyu Zhao, Bingyu Kong, Matthai Philipose, Arvind Krishnamurthy, and Ravi Sundaram. 2019. Nexus: a GPU cluster engine for accelerating DNN-based video analysis. In *Proceedings of the 27th ACM Symposium on Operating Systems Principles* (Huntsville, Ontario, Canada) *(SOSP '19)*. Association for Computing Machinery, New York, NY, USA, 322–337. https://doi.org/10.1145/3341301.3359658

[38] Pirah Noor Soomro, Mustafa Abduljabbar, Jeronimo Castrillon, and Miquel Pericàs. [n. d.]. Shisha: Online scheduling of CNN pipelines on heterogeneous architectures. In *PPAM 2022*.

[39] Pirah Noor Soomro, Nikela Papadopoulou, and Miquel Pericàs. 2023. ODIN: Overcoming Dynamic Interference in iNference Pipelines. In *European Conference on Parallel Processing*. Springer, 169–183.

[40] Neeraja J Yadwadkar, Francisco Romero, Qian Li, and Christos Kozyrakis. 2019. A case for managed and model-less inference serving. In *Proceedings of the Workshop on Hot Topics in Operating Systems*. 184–191.

[41] Haoyu Zhang, Ganesh Ananthanarayanan, Peter Bodik, Matthai Philipose, Paramvir Bahl, and Michael J Freedman. 2017. Live video analytics at scale with approximation and {Delay-Tolerance}. In *14th USENIX Symposium on Networked Systems Design and Implementation (NSDI 17)*. 377–392.

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