Loki: A System for Serving ML Inference Pipelines with Hardware and Accuracy Scaling

Sohaib Ahmad
University of Massachusetts Amherst
sohaib@cs.umass.edu

Hui Guan
University of Massachusetts Amherst
huiguan@cs.umass.edu

Ramesh K. Sitaraman
University of Massachusetts Amherst
ramesh@cs.umass.edu

ABSTRACT

The rapid adoption of machine learning (ML) has underscored the importance of serving ML models with high throughput and resource efficiency. Traditional approaches to managing increasing query demands have predominantly focused on hardware scaling, which involves increasing server count or computing power. However, this strategy can often be impractical due to limitations in the available budget or compute resources. As an alternative, accuracy scaling offers a promising solution by adjusting the accuracy of ML models to accommodate fluctuating query demands. Yet, existing accuracy scaling techniques target independent ML models and tend to underperform while managing inference pipelines. Furthermore, they lack integration with hardware scaling, leading to potential resource inefficiencies during low-demand periods. To address the limitations, this paper introduces Loki, a system designed for serving inference pipelines effectively with both hardware and accuracy scaling. Loki incorporates an innovative theoretical framework for optimal resource allocation and an effective query routing algorithm, aimed at improving system accuracy and minimizing latency deadline violations. Our empirical evaluation demonstrates that through accuracy scaling, the effective capacity of a fixed-size cluster can be enhanced by more than 2.7× compared to relying solely on hardware scaling. When compared with state-of-the-art inference-serving systems, Loki achieves up to a 10× reduction in Service Level Objective (SLO) violations, with minimal compromises on accuracy and while fulfilling throughput demands.

CCS CONCEPTS

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

KEYWORDS

Inference Serving, Model Serving, Inference Pipelines, Machine Learning, Autoscaling

ACM Reference Format:

Sohaib Ahmad, Hui Guan, and Ramesh K. Sitaraman. 2024. Loki: A System for Serving ML Inference Pipelines with Hardware and Accuracy Scaling. In *The 33rd International Symposium on High-Performance Parallel and Distributed Computing (HPDC '24), June 3–7, 2024, Pisa, Italy.* ACM, New York, NY, USA, 14 pages. https://doi.org/10.1145/3625549.3658688

HPDC '24, June 3–7, 2024, Pisa, Italy

ACM ISBN 979-8-4007-0413-0/24/06
https://doi.org/10.1145/3625549.3658688

1 INTRODUCTION

The growing popularity of machine learning (ML) has led to the development of model serving systems[1], where pre-trained ML models are hosted on a cluster of servers to provide fast and accurate responses to inference queries. Model serving systems typically guarantee certain Service Level Objectives (SLOs) to users in terms of latency deadlines and, at the same time, strive to achieve high throughput and high resource efficiency in order to serve as many queries as possible in a given amount of time.

As query demand (measured by queries per second or QPS) usually changes over time, model serving systems need to handle demand variations gracefully. To accommodate increasing query demands, conventional methods primarily rely on hardware scaling, i.e., adding more devices or using more powerful accelerators, to improve system throughput [10, 32]. However, hardware scaling may not be feasible due to budget constraints or the limited availability of hardware resources in edge clusters or private clouds.

Accuracy scaling. Accuracy scaling has recently been proposed as an alternative to hardware scaling [5, 17]. A model serving system that uses accuracy scaling adapts model accuracy instead of hardware resources to gracefully handle query demand variations. Accuracy scaling is motivated by the fact that ML models can offer different levels of accuracy depending on the time spent computing the answer: less time spent on computation leads to less accurate results but higher throughput. ML models with different accuracy profiles are called "model variants", which can be created from model compression techniques [7, 37], or as part of neural network architecture designs. When the query demand is high, a model serving system with accuracy scaling serves queries using less accurate model variants to avoid SLO violations. When the demand drops, the system serves queries using more accurate model variants to improve accuracy.

Accuracy scaling is a particularly effective strategy in scenarios where query demands are so high that they risk overwhelming the available servers. In such cases, accuracy scaling ensures that the system continues to provide timely responses. These prompt responses that are less accurate are often more critical than slower more accurate responses, or worse, queries that fail to be processed at all. Accuracy scaling is especially beneficial for real-time ML applications, including interactive and cloud-based editing services, where quick feedback is essential [13, 22].

Accuracy scaling can be strategically combined with hardware scaling to optimize query handling. During periods of low query demand, the model serving system can employ hardware scaling to reduce the number of active servers, thereby optimizing resource usage. As query demand escalates, the system can shift to accuracy

[1]We use the terms "model serving" and "inference serving" interchangeably.

scaling. This transition enables the system to enhance its throughput capacity (in QPS), accommodating the surge in demand while still adhering to the SLOs set for user satisfaction.

Limitations of existing approaches. The current approach to accuracy scaling is primarily tailored for serving individual, independent ML models. However, as ML becomes more integrated into practical applications, the use of *inference pipelines* is increasingly common. These pipelines combine multiple ML models to tackle complex inference tasks and are becoming a standard part of ML inference workloads. For instance, an image generation pipeline such as Adobe Firefly [4] might sequentially employ a text embedding model, a diffusion model, and an image super-resolution model to produce high-resolution images from text prompts. The end-to-end inference latency of these pipelines must adhere to the specific SLOs set by the application (e.g., 200ms).

When the existing accuracy scaling method [5] is applied to inference pipelines, it often leads to suboptimal resource allocation, resulting in high SLO violation rates and poor response quality. The core issue with the current accuracy scaling approach lies in its *pipeline-agnostic* perspective on resource allocation. It adjusts the accuracy of ML models and allocates computing resources without considering the dependencies between the models in the different tasks of a pipeline. This lack of consideration for the inter-model relationships can impair the overall effectiveness of resource use in complex, multi-model inference tasks.

Another limitation of the existing accuracy scaling method is its lack of integration with hardware scaling. This shortcoming becomes evident particularly during periods of low query demand. In such scenarios, instead of scaling down the hardware resources, current methods continue to utilize all available servers to handle queries. This approach leads to inefficiencies, as it does not dynamically adjust the server usage based on the actual demand, resulting in unnecessary resource expenditure and under-utilization of the server infrastructure.

The Loki system. To address the problem, this paper introduces Loki[2], a model serving system designed to handle inference pipelines effectively using both hardware and accuracy scaling. The primary objectives of Loki are to maximize the overall system accuracy and minimize the active server count, while adapting to fluctuating query demands. System accuracy is defined as the average accuracy across all queries processed by the system. Loki operates under the assumption that, given sufficient resources, every query would prefer the most accurate model variant. However, it also recognizes that in situations where resources are constrained, a timely response with slightly lower accuracy is acceptable.

When query demand is relatively low compared to available server capacity, Loki optimizes resource usage by reducing the number of active servers (and thus hardware costs) required to process queries with the most accurate model variants. Under these conditions, the system consistently achieves maximum accuracy. In contrast, as the query demand increases, Loki smoothly transitions to a pipeline-aware accuracy scaling mode. This mode focuses on maximizing system accuracy while accommodating the increasing volume of queries, resulting in all servers actively participating in query processing.

[2]Loki is named after a Norse mythology figure who possessed the ability to change form and appearance, much like our system transitions between hardware and accuracy scaling.

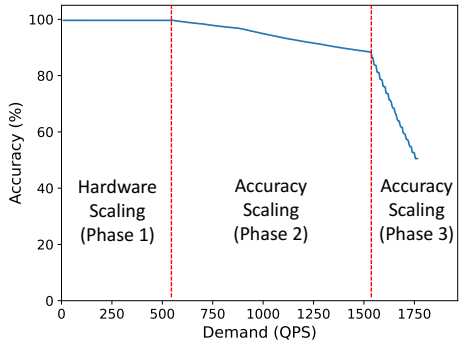


Figure 1: For a traffic analysis pipeline that consists of two sequential tasks, Loki first accomodates the increase in query demand by using hardware scaling. When demand increases further, Loki successively decreases the accuracy of each task of the pipeline to increase throughput to meet demand. Phase 2 decreases the 2^{nd} task's accuracy as it causes smaller end-to-end accuracy drop.

Functioning of Loki. To illustrate the functioning of Loki, we hosted a simple two-task traffic analysis ML pipeline on a cluster of 20 servers. The first task of the pipeline consists of an object detection model for identifying cars in an image and the second task classifies the identified car according to its make and model. Loki managed the resources in the server cluster using both hardware and accuracy scaling to serve user queries for this pipeline.

Figure 1 illustrates the functioning of Loki as it varied the system throughput to meet the user demand, ensuring that no queries were dropped. In phase 1, as demand increased, Loki accommodated the demand by hardware scaling, while accuracy remained unchanged. In this phase, Loki increased the number of available servers to meet the increasing query demand, but continued to use model variants that yield the highest accuracy. In the second phase at around 560 QPS, it is no longer possible to scale the hardware since the server limits of the cluster have been reached. In response, Loki decreases accuracy to serve the increasing demand. Loki recognized that it is possible to get a larger increase in throughput for a given loss in end-to-end accuracy of the pipeline by using a less accurate model for the second task of the pipeline. Consequently, it decreased the accuracy of the second task to increase throughput, while keeping the accuracy of the first task at the highest level. As query demand continued to increase to about 1550 QPS, the system could no longer decrease the accuracy of task 2 to meet the demand. Therefore, it entered phase 3 where it starts to decrease the accuracy of task 1 of the pipeline. This allows Loki to support upto 1765 QPS which is maximum throughput the system can support without dropping any request.

It should be noted that Loki can support 2.7× more throughput at the end of phase 2 than a system that does hardware scaling alone, albeit with a modest drop in accuracy of 13%. Further, Loki can support up to 3× more throughput at the end of phase 2 than

hardware scaling alone, albeit with a more significant accuracy drop. In practice, there is usually a minimum level of acceptable accuracy required for queries, which limits the amount of accuracy scaling that can be performed.

Our contributions. Our specific contributions follow:

- We design Loki, the first model serving system that integrates hardware scaling with accuracy scaling to effectively serve inference pipelines.
- We present a MILP-based theoretical framework for optimal allocation of resources in a cluster that incorporates performance models for the accuracy and throughput of inference pipelines. Using this framework, Loki periodically decides which model variants are hosted on which servers to meet throughput, accuracy, and latency requirements.
- When queries arrive, they need to be routed to the right sequence of model variants in the pipeline. For this task, we propose an efficient routing algorithm that intelligently routes the queries in real-time to maximize system accuracy and minimize SLO violations.
- We evaluate Loki against two state-of-the-art model serving systems, one system that performs hardware scaling but not accuracy scaling [10] and another that performs accuracy scaling but is pipeline-agnostic [5]. Using query workloads from synthetic and production traces, we show that Loki reduces SLO violations by more than 10× compared to pipeline-agnostic accuracy scaling systems while using 2.7× fewer servers during off-peak times where the demand is low. Further, Loki increases the effective capacity of the cluster by more than 2.7× compared to systems that do not perform accuracy scaling.

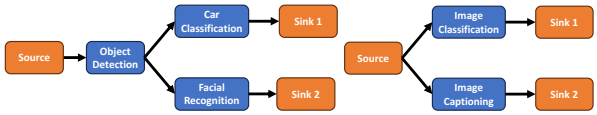
2 BACKGROUND AND CHALLENGES

This work is motivated by the importance of serving inference pipelines and the benefits of accuracy scaling in model serving. We provide the background and then outline the challenges in building a pipeline-aware inference serving system.

2.1 Background

Inference pipelines. Inference pipelines integrate multiple ML tasks together in a dataflow graph to address more complex tasks. These pipelines can be represented as *directed rooted trees*, where each node (or vertex) represents a task, the input, or the output, and each directed edge of two tasks denotes the flow of data between them[3]. The root of the tree is referred to as the source (i.e., the input) and the leaves of the trees are the sinks (i.e., the outputs). Thus, the rooted tree consists of multiple source-to-sink paths, where each of these paths has its own end-to-end accuracy. The end-to-end accuracy of the pipeline graph is the average of the end-to-end accuracy of all the source-to-sink paths.

In the execution of an inference pipeline for serving a query (also called a request[4]), the ML model for one task generates intermediate outputs that serve as inputs (termed *intermediate queries*) for the ML model in the subsequent tasks. Figure 2 illustrates two representative inference pipelines studied in this paper. The traffic analysis pipeline can be used to generate traffic analytics on the video feed from cameras at intersections. The social media pipeline can be used by platforms such as Twitter and Facebook to detect objects in the image and generate suggested captions.



(a) Traffic analysis pipeline **(b) Social media pipeline**

Figure 2: Examples of inference pipelines

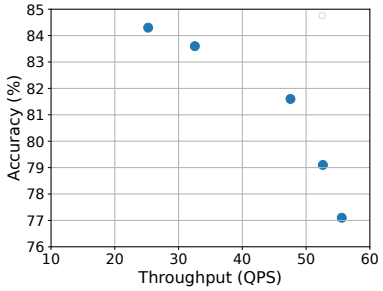


Figure 3: Accuracy-throughput tradeoff for EfficientNet model variants as profiled on an NVIDIA V100 GPU

Accuracy scaling. Accuracy scaling leverages the fact that model variants with different compute complexities (e.g., models from the EfficientNet family [39]) can be used to serve the same inference task. A model variant that is more lightweight is usually less accurate, but can be executed faster, resulting in higher throughput on the same hardware, as shown in Figure 3. The concept of accuracy scaling was first introduced in Proteus [5], a model serving system designed for handling independent ML models on a cluster with a fixed number of servers. Accuracy scaling is particularly effective in managing high query demands with a limited number of servers. In scenarios where the volume of queries exceeds the server capacity, accuracy scaling strategically reduces the accuracy of the models. This reduction is done to ensure that the system meets the latency deadlines of the queries, thus balancing the trade-off between accuracy and timely response under heavy load conditions.

2.2 Challenges

Despite the promise of accuracy scaling, applying it to serve inference pipelines is challenging due to the complexities introduced by the inter-dependencies of ML models.

2.2.1 Optimal resource allocation. In this work, a resource allocation plan includes three key specifications: (1) the choice of model variants for each task of an inference pipeline, (2) the number of replicas for each model variant (termed *replication factor*), and (3) the maximum batch size that can be used for each model variant.

[3]Loki does not support general directed acyclic graphs where an ML model derives input from multiple models. This paper uses the terms "inference pipeline" and "pipeline graph" interchangeably.
[4]We use the terms query and request interchangeably in this work.

The maximum batch size corresponds to the maximum time budget assigned to a task.

In the context of accuracy scaling, an optimal resource allocation plan maximizes system accuracy while satisfying a target query demand given a fixed cluster size. The accuracy scaling approach introduced in Proteus [5] is pipeline-agnostic, meaning it adjusts the accuracy of ML models individually without considering the interdependencies between them. When applied to inference pipelines, this approach can lead to suboptimal resource allocation, resulting in poor query response quality and high rates of SLO violations. These interdependencies present three major issues:

1. Impact of the accuracy of individual models on the end-to-end accuracy of the pipeline. Choosing model variants for each task must be made with the knowledge of its impact on the end-to-end accuracy on the pipeline. When facing increased query demands, the system should reduce the accuracy of models that minimally affect the end-to-end pipeline accuracy. For instance, Figure 1 shows that decreasing accuracy of the second task in the pipeline causes smaller end-to-end accuracy drop compared to the first task. This consideration is absent in the existing accuracy scaling methods, which do not consider the influence of individual models on end-to-end pipeline accuracy.

2. Throughput bottlenecks. The optimal batch size and replication factor for each selected model variant depend on the throughput bottleneck of the inference pipeline. If a given task is not the bottleneck, increasing the batch size or assigning more resources to a model variant of that task enhances throughput for that task but does not necessarily improve overall system throughput. Moreover, allocating more resources to the bottleneck task may create resource shortages for other tasks, potentially creating new bottlenecks. Using a larger batch size at a given task also introduces longer processing delays for that task, reducing the time available for other tasks. This is a departure from the scenario addressed by Proteus, where ML models are independent and throughput improvements in any model enhance overall system throughput.

3. Workload multiplication effects. The workload of each task can be influenced by the model variant used in preceding task. For example, in a pipeline with a face detection model followed by a face recognition model, the input demand for the recognition model depends on the output of the detection model. A more accurate detection model might detect more faces, thereby increasing the workload for the recognition task. The existing accuracy scaling approach fails to account for such workload dependencies between models in resource management.

Our approach. We design performance models that assess how a resource allocation plan influences system accuracy, latency, and throughput capacity. These performance models are particularly crafted to consider the intricate relationships between the models in a inference pipeline. Utilizing these performance models, we can frame the resource allocation problem within a Mixed-Integer Linear Programming (MILP) framework and leverage MILP solvers to determine the optimal resource allocation plan. Loki periodically invokes the solver to re-allocate resources to accomondate macro-scale query demand changes.

Additionally, the performance models enable us to integrate hardware scaling into the same MILP framework used for accuracy scaling, albeit with a distinct optimization goal. In terms of hardware scaling, the ideal resource allocation plan is defined as the one that minimizes the number of active servers required to process queries while meeting a target query demand. The optimization formula for this objective is based on our performance model, which delineates the connection between the system's latency, throughput capacity, and the specifics of a resource allocation plan. This unified approach under the MILP framework allows for a cohesive treatment of both hardware and accuracy scaling, each with its unique optimization targets, while maintaining a consistent underlying methodology. We present the performance models and our optimization in Section 4.

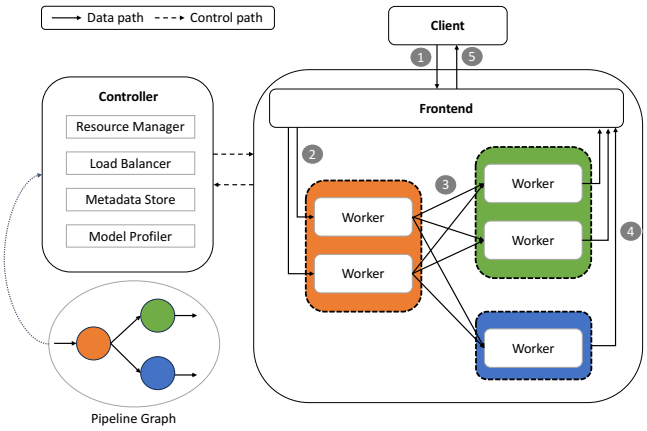


Figure 4: System Architecture of Loki

2.2.2 Query execution with accuracy scaling. Another challenge in utilizing accuracy scaling for serving inference pipelines is deciding the optimal execution path for each incoming query. This decision aims to enhance system accuracy while minimizing violations of SLO. The MILP formulation used for system accuracy estimation operates under the assumption that each deployed ML model functions at its maximum throughput to meet the target query demand. However, this assumption may not always be valid due to the dynamic nature of query arrivals during runtime. The method by which queries are routed and executed can significantly affect both the quality of their responses and their capacity to adhere to predetermined latency deadlines.

Our approach. We present a greedy request routing algorithm in Section 5 that routes requests in a way that maximizes system accuracy. To minimize SLO violations, we perform a runtime optimization to drop requests that are unlikely to meet their SLOs, in order to free up resources for requests with higher chances of meeting their SLOs.

3 SYSTEM ARCHITECTURE OF LOKI

We now present an overview of Loki's system architecture and provide more details in Sections 4 and 5. Figure 4 shows the three key components: Controller, Frontend, and Workers.

Controller. The Controller is responsible for managing the resources in the system and for routing the queries. It uses the following sub-components to achieve this.

Resource Manager. The Resource Manager performs resource allocation periodically in response to the incoming demand to indicate which model variants to host, as well as their replication factors and maximum batch sizes. It consults the Metadata Store to get the historical query demand, the pipeline graph, and the profile of model variants for each task in the graph to perform the allocation. Once the Resource Manager obtains an allocation plan, it adjusts the allocation of workers to model variant instances in the system to reflect the new allocation plan. The Resource Manager assumes a finite-size cluster for allocation. As long as it can meet demand using the highest accuracy model variants for each task in the pipeline, it tries to scale the hardware needed to serve the demand. If the demand cannot be met even using the entire cluster, it drops accuracy to meet the increased demand. We explain the details of the resource allocation algorithm in Section 4.

Load Balancer. The Load Balancer is tasked with routing the queries through the hosted instances to maximize system accuracy. It uses the resource allocation plan set up by the Resource Manager as well as the pipeline graph and real-time demand from the Metadata Store to perform the routing. It sets up routes from the Frontend to the first-task workers of the pipeline, as well as the routes between intermediate workers.

Model Profiler. The Controller uses the Model Profiler to profile the expected processing times of each model variant in the pipeline with different batch sizes during the initial setup. The profiles are then stored in the Metadata Store and used by the Resource Manager every time it performs resource allocation.

Metadata Store. The Metadata Store holds the information required by the Resource Manager and Load Balancer. It stores the representation of the pipeline as a graph, the profiled throughput and accuracy of each model variant, and the profiled end-to-end accuracy of each source-to-sink path through the graph. During the initial setup, a pipeline graph, the model variants for each node in the graph, and the end-to-end pipeline latency requirement are registered in the Metadata Store.

Frontend. The Frontend accepts queries from the client and forwards them to the respective workers. The query is then forwarded by those workers to intermediate workers in the pipeline, and the workers at the last task of the pipeline return the results to the Frontend which then returns the results to the client. The Frontend also records the incoming demand into the system and reports it to the Controller which stores it in the Metadata Store.

Workers. The workers host the model variants and execute inference queries. Each worker has a queue that it uses to form batches. As the worker executes queries, it records the number of subsequent requests generated for downstream tasks in terms of a multiplicative factor on the incoming number of requests and reports it to the Controller using heartbeat messages.

Query Processing. Clients interact with Loki in the following way. Client sends a query to the Frontend of Loki (①). The Frontend routes the query to one of first-task workers (②). The first-task worker passes the intermediate query (or queries) to one (or more) second-task worker (③) and so on. The last-task worker (or workers) pass the inference results to the Frontend (④). The Frontend aggregates the results and returns them to the client (⑤).

We now provide details of the two core modules: the Resource Manager and the Load Balancer in Sections 4 and 5 respectively.

4 RESOURCE MANAGER

The Resource Manager is tasked with allocating resources in the system to meet the incoming demand. It uses the incoming demand as input and outputs the resource allocation plan that describes the model variants to host as well as the replication factor and the maximum batch size that can be used for each model variant by performing the following two steps.

(1) **Hardware scaling.** The Resource Manager first tries to serve the incoming demand with the minimum number of workers by using the most accurate model variants for each task in the pipeline. If this is not possible, it executes the accuracy scaling step below.
(2) **Accuracy scaling.** If the Resource Manager is unable to meet demand by using the entire cluster with the most accurate model variants, it tries to determine the minimum amount of system accuracy to sacrifice in order to meet the demand. This enables the Resource Manager to increase the throughput capacity of the cluster, allowing it to serve a greater demand compared to using hardware scaling alone.

Each of the above steps is modeled as a mixed-integer linear program (MILP) as described below. The MILPs are solved by the Resource Manager to get the resource allocation plan.

4.1 MILPs for hardware and accuracy scaling

We now formulate the resource allocation problem as a mixed integer linear programming (MILP) optimization. We first elaborate the input and the output of the optimization problem and then introduce the performance models that quantify the relationship between a resource allocation plan and system accuracy, latency, and throughput. Table 1 summarizes the notation used in the optimization.

Inputs. We are given as input the pipeline graph consisting of a set of tasks T and a set of directed edges E where an edge $e = (i, j) \in E$ denotes an edge from the i^{th} task t_i to the j^{th} task t_j. The pipeline graph is a directed rooted tree with the source node (t_1) as the root which does not have any incoming edges. We are also given the incoming demand of the system, D, that arrives at the root. Let $r(i, k)$ represent the multiplicative factor of the k^{th} model variant of task $t_i \in T$.

Output. The output is the resource allocation plan, defined by the two optimization variables: $x(i, k)$ and $y(i, k)$, representing the number of instances to host for the k^{th} model variant of task $t_i \in T$ along with the maximum batch size to use for it, respectively.

Meeting the system throughput demand. To model how an allocation plan affects the system throughput, we need to introduce two concepts: augmented graph, and intermediate query demand.

Augmented graph. The augmented graph aims to represent all possible materializations of a pipeline graph using different combinations of model variants for each task. We construct an augmented graph from the given pipeline graph in the following way: For every vertex i in the pipeline graph that represents the i^{th} task t_i, we create vertices (i, k) in the augmented graph representing the k^{th} model variant of task t_i. We add a directed edge from a vertex (i, k) to (j, k') in the augmented graph if (i, j) is an edge in the pipeline graph, for all k, k'.

Subscripts	
T	the set of tasks
t_i	the i^{th} task in the pipeline, $t_i \in T$
V_i	the set of model variants for the i^{th} task
$v_{i,k}$	the k^{th} model variant for the i^{th} task, $v_{i,k} \in V_i$
E	the set of edges between tasks in the pipeline graph
P	the set of all root-to-sink paths in the augmented graph
B	the set of allowed batch sizes
b	batch size, $b \in B$
Inputs	
D	incoming demand (QPS)
S	number of workers in the cluster
L	latency SLO of the pipeline
$r(i,k)$	multiplicative factor for the k^{th} model variant of i^{th} task
$q(i,k,b)$	profiled throughput (QPS) for the k^{th} model variant of the i^{th} task with batch size b
$A(v_{i,k})$	profiled accuracy of the k^{th} model variant of the i^{th} task
$\hat{A}(p)$	end-to-end profiled accuracy of path p
Optimization variables	
$x(i,k)$	number of instances for the k^{th} model variant of the i^{th} task
$y(i,k)$	maximum batch size to use for the k^{th} model variant of the i^{th} task
Intermediate variables	
$c(p)$	ratio of queries supported through path p
$I(p)$	1 if there is any traffic through the path p; 0 otherwise
$l(i,k)$	the processing latency of the k^{th} model variant of the i^{th} task with the configured batch size
$\hat{l}(p)$	end-to-end latency through path p

Table 1: Notation for MILP

Intermediate query demand. The Resource Manager not only needs to host enough model instances to serve the incoming queries at the first task of the pipeline, but it also needs to consider the intermediate queries generated by the initial tasks to host model instances for the downstream tasks in the pipeline. For example, an object detection model in the traffic analysis pipeline may detect 10 cars in an image, creating 10 intermediate queries to be served by the car classification model. Therefore, it needs estimates of the multiplicative factor for each model variant. The Resource Manager uses the estimate of incoming demand into the system as well as the profiled multiplicative factor of each model variant to estimate the intermediate query demand.

We next model the requirement that model variants chosen for a task need to meet the task's intermediate query demand in Constraint 2. For every model variant $v_{i,k}$, we want to ensure that it has enough resources to serve all requests arriving at it. To get the number of requests arriving at $v_{i,k}$, we need to consider all paths that contain it. Let $P'_{i,k}$ be the set of all paths p that start at a vertex that corresponds to the root and end in vertex (i,k) that represents model variant $v_{i,k}$. For $p \in P'_{i,k}$, let $m(p,i,k)$ represent the number of requests derived from a single request entering path p that reach $v_{i,k}$. Thus, the following holds.

$$m(p,i,k) = \prod_{(i',k') \in p} r(i',k') \tag{1}$$

We now ensure that $v_{i,k}$ has enough resources to serve all requests going through it. Let P be the set of all paths in the augmented graph that start at a vertex that corresponds to the root and end at a vertex that corresponds to a sink. Let $P_{i,k}$ be the set of all paths $p \in P$ that include vertex (i,k) that represents model variant $v_{i,k}$. As the total number of requests per second entering the pipeline is D and $c(p)$ is the ratio of these requests that we route through the path p, the number of requests that arrive at $v_{i,k}$ after multiplication are $\sum_{p \in P_{i,k}} D \cdot c(p) \cdot m(p,i,k)$. Then, to ensure $v_{i,k}$ has enough resources to serve all the requests going through it, we add the following constraint.

$$\sum_{p \in P_{i,k}} D \cdot c(p) \cdot m(p,i,k) \leq x(i,k) \cdot q(i,k,y(i,k)) \quad \forall v_{i,k} \in V_i, \forall t_i \in T \tag{2}$$

We require that the number of workers used may not exceed the total available workers in the cluster.

$$\sum_{i,k} x(i,k) \leq S \tag{3}$$

Meeting the latency SLO. We now model how a resource allocation plan affects the end-to-end pipeline latency, which is bounded by the SLO requirements of queries. The maximum batch size is bounded by the latency requirements of requests and also the largest batch size feasible on a specific device. We configure the maximum batch size using one of the batch sizes from the set of allowed batch sizes (Constraint 4). The processing latency of a model variant depends on the maximum batch size configured for it and the throughput of the model using that batch size (Constraint 5).

$$y(i,k) \in B \qquad \forall v_{i,k} \in V_i, \forall t_i \in T \tag{4}$$

$$l(i,k) = \frac{y(i,k)}{q(i,k,y(i,k))} \qquad \forall v_{i,k} \in V_i, \forall t_i \in T \tag{5}$$

We define the end-to-end processing latency through a path p as following.

$$\hat{l}(p) = \sum_{(i,k) \in p} l(i,k) \qquad \forall p \in P \tag{6}$$

As we need to meet the end-to-end latency SLO of the pipeline, we need to ensure that the processing latency through each path serving any query is less than the SLO.

$$\hat{l}(p) \cdot I(p) \leq L \qquad \forall p \in P \tag{7}$$

To account for the waiting time of queries in the queue, we divide the SLO by two. This is motivated by an observation from prior work [5, 36]: a query that arrives right after a batch starts executing needs to wait for the current batch to finish, before starting execution with the next batch and thus may have to wait twice the amount of processing time of a batch. Based on this, we divide the SLO by two, assuming that the query's waiting time in the queue is as long as the query's execution time.

Modeling the system accuracy. As mentioned in Section 2.2.1, the model variants chosen at each task of the pipeline affect the

end-to-end accuracy. Therefore, to capture this accuracy, we profile the end-to-end accuracy of every path $p \in P$ as $\hat{A}(p)$. Given that the optimization configures the ratio of requests through the path p to be $c(p)$, the system accuracy is $\sum_{p \in P} c(p) \cdot \hat{A}(p)$.

The MILP optimization. We now present the MILP optimization for both hardware and accuracy scaling based on the above-mentioned performance models.

Step 1: Hardware scaling. We first try to serve demand using the most accurate model variant for each task. To achieve this, we constrain the number of hosted instances for all other model variants to be 0. Let us denote the most accurate model variant for the task t_i as:

$$v_i^{max} = \arg\max_{v_{i,k} \in V_i} A(v_{i,k}) \qquad \forall t_i \in T \tag{8}$$

Then we can denote the set of all other model variants as:

$$\overline{V_i} = \{v_{i,k} \in V_i | A(v_{i,k}) < A(v_i^{max})\} \qquad \forall t_i \in T \tag{9}$$

We can now define the constraint to disallow less accurate model variants as following.

$$x(i,k) = 0 \qquad \forall v_{i,k} \in \overline{V_i}, \forall t_i \in T \tag{10}$$

In this case, the optimization objective is to minimize the number of workers used to serve the demand.

$$min \sum_{i,k} x(i,k) \quad \text{s.t. Constraints 1-10 hold} \tag{11}$$

It is important to note that it may not be possible to serve demand with the highest accuracy model variants by using even the entire cluster. In this case, the above optimization will immediately detect the constraints to be infeasible, and we resort to accuracy scaling.

Step 2: Accuracy scaling. The optimization objective for accuracy scaling is to maximize the system accuracy, which is the average accuracy experienced by all queries served by the system. The system accuracy is measured by multiplying the end-to-end accuracy of each path by the ratio of queries flowing through it.

$$max \sum_{p \in P} c(p) \cdot \hat{A}(p) \quad \text{s.t. Constraints 1-7 hold} \tag{12}$$

4.2 Solving the MILP

As the Resource Manager is invoked periodically to respond to long-term changes in demand, it can tolerate a higher runtime from considering a large number of paths through the pipeline, as long as it yields an optimal solution at the end. For the purpose of our experiments, we use a 10-second invocation interval for the Resource Manager. We show in Section 6.5 that the runtime overhead of the MILP is low enough to allow it to adapt reasonably quickly to this invocation frequency. Additionally, the Resource Manager may reallocate resources if it detects a significant change in the demand between its periodic invocations. To estimate the demand to serve, we use an exponentially weighted moving average on the recent demand history.

Latency budgets for tasks. The batch sizes set by the MILP not only serve as guidance for the workers to form batches during execution, they also allow us to set latency budgets for each task. Since the optimization ensures that the execution latency through every path falls under the SLO using the configured batch sizes, we use the execution time of a model variant with the configured batch size as the latency budget for its task. These latency budgets are useful during query execution to make sure requests are on track to meet their SLOs as they move through the tasks in the pipeline. In case they fall behind, we can use the latency budgets to perform early dropping of requests as we detail in Section 5.2.

Communication latency. It is important to consider the communication latency between workers since the end-to-end execution latency of a query depends on the communication latency between the workers that serve that particular query. As we consider all servers to be in the same cluster, we assume communication latency between any pair of servers to be homogeneous. Therefore, during resource allocation, we subtract the product of the total number of servers in the path with this communication latency from the latency SLO of the pipeline.

Estimating multiplicative factors. As mentioned before, each request generates multiple requests for downstream tasks in the pipeline. We refer to the number of outgoing requests generated for each incoming request as the multiplicative factor. We note that every model variant can have a different multiplicative factor, for example, extending the example from above, a lower accuracy object detection model such as YOLOv5n may detect fewer cars in an image compared to a higher accuracy model variant such as YOLOv5x. Each model variant hosted at a worker records the multiplicative factors it observes when serving queries and reports them to the Controller through heartbeat messages. The Controller aggregates these for each model variant to be used by the Resource Manager.

5 LOAD BALANCER

The Load Balancer produces the routing tables that enable each query to be routed through a sequence of model variants in real-time to maximize system accuracy and reduce SLO violations. To achieve this, it takes as input the resource allocation plan produced by the Resource Manager, the pipeline graph, and the recent demand history from the Metadata Store, and outputs routing tables for both the Frontend and workers.

The Load Balancer is a centralized component and periodically updates the routing tables of workers, while the workers use their respective routing tables during real-time execution to find downstream workers for intermediate requests. We present our algorithm in this section and explore the overhead of the Load Balancer in Section 6.5.

5.1 Request Routing

We now present our request routing algorithm, MostAccurateFirst (Algorithm 1). The algorithm works in the following way: Starting from the root node of the pipeline graph, it takes the incoming QPS of the node and assigns model variants to it in non-increasing order of their profiled single-model accuracies. As each

model variant can have a different multiplicative factor, the outgoing requests for this node are calculated by multiplying the requests assigned to each model variant by the multiplicative factor of that variant. The outgoing requests are sent to the children nodes, and we recursively repeat the same procedure on each of the children. MostAccurateFirst generates routing tables based on estimated demand and updates the routing tables of all the workers and the Frontend.

Algorithm 1

```
1:  procedure MostAccurateFirst(pipelineGraph, worker metadata)
2:    sortedGraph ← TopologicalSort(pipelineGraph)
3:    routingTables ← ϕ
4:    for task in sortedGraph do
5:      workers ← sort(task.workers)  ▷ By single-model accuracy
6:      for worker in workers do
7:        workerTable ← ϕ
8:        for child in task.children do
9:          outgoing ← worker.incoming * task.multFactor * child.branchRatio
10:         totalChildDemand ← outgoing
11:         childWorkers ← sort(child.workers)
12:         for cWorker in childWorkers do
13:           if cWorker.capacity > 0 & outgoing > 0 then
14:             routed ← min(outgoing, cWorker.capacity)
15:             routingProbability ← routed / totalChildDemand
16:             workerTable.addEntry(cWorker, routingProbability)
17:             outgoing ← outgoing − routed
18:             cWorker.capacity ← cWorker.capacity − routed
19:             cWorker.incoming ← cWorker.incoming + routed
20:       routingTables[worker] ← workerTable
21:   return routingTables
```

The Load Balancer runs the MostAccurateFirst algorithm every time the Resource Manager changes the resource allocation plan. It also runs periodically between successive invocations of the Resource Manager. On each execution, the MostAccurateFirst algorithm produces routing tables for every worker and pushes the routing tables to the respective workers. The workers then use their routing tables during real-time query execution to find downstream workers to forward intermediate queries. Since we saturate workers for each node in non-increasing order of their single-model accuracies, we may have some workers for each node with leftover capacity. We make a list of these workers along with their leftover capacities and propagate this list to their upstream workers. The upstream workers can use this list to perform Opportunistic Rerouting, a technique we describe in Section 5.2.

As the end-to-end pipeline accuracy is a monotonic function of single-model accuracies, and MostAccurateFirst ensures that each node in every source-to-sink path in the pipeline graph gets the highest single-model accuracy for a given QPS, this means MostAccurateFirst maximizes the end-to-end pipeline accuracy.

5.2 Early dropping with opportunistic rerouting

The Resource Manager and Load Balancer assess the recent demand history of the system to allocate resources and set up routing for future requests. However, real-time demand can deviate from these estimates. Moreover, the estimates are made at the granularity of seconds, while request arrivals and multiplicative factors may fluctuate at smaller timescales between these estimates. Due to these reasons, there is a possibility that some requests may exceed their SLOs despite provisioning the system to prevent SLO violations. In such instances, it may be more effective to preemptively drop a request if we can anticipate that it is likely to miss its SLO. This decision can only be made at runtime during query execution at individual workers. We refer to this process as *early dropping*, and it can mitigate SLO violations by freeing up resources for requests that are expected to meet their SLOs.

Since requests undergo execution sequentially through the tasks within the pipeline, we use the latency budget of each task to estimate whether a request is on track to meet its SLO. Recall that we set the latency budget of each task by using the batch sizes set by the Resource Manager for each hosted model variant.

We consider two naïve mechanisms to perform early dropping using the allocated latency budgets for the pipeline tasks:

(1) **Per-task dropping.** When a request finishes execution at a given task, we note the total time spent by the request at the task, i.e., the processing time of the request as well as the time it spent waiting in the queue. If this time exceeds the latency budget assigned for the given task, we estimate that the request is likely to miss its end-to-end SLO since the SLO is divided into latency budgets for individual tasks by the Resource Manager. Therefore, one possible mechanism is to drop the request early on in order to free up resources for requests that are more likely to meet their end-to-end SLOs. Per-task dropping tracks the request at every task during its execution and drops it if it misses the latency budget for any task along the path. However, it is important to note that this approach might be overly aggressive, as a request that exceeds its latency budget for an earlier task may still have the potential to meet the end-to-end SLO by compensating at a subsequent task.

(2) **Last-task dropping.** This mechanism does not drop any request up until the last task, even if it exceeds its per-task latency budget at earlier tasks. When the request reaches the last task and its leftover latency budget is smaller than the expected processing time, the request is then dropped. While this approach is more conservative than per-task dropping, it carries the risk of tying up resources at upstream tasks for requests that may ultimately be dropped later.

Opportunistic rerouting. To strike a balance between the above-mentioned extreme approaches, we introduce a novel mechanism for early dropping termed *Opportunistic Rerouting*. This approach involves intelligently redirecting requests that are running behind if there is a chance for them to meet their latency SLOs.

Opportunistic rerouting navigates the tradeoff between being overly aggressive or conservative. The key idea is that if a request exceeds its latency budget at any given task, we try to find a faster alternative path for the subsequent task in order to make up for it.

We accomplish this as follows. Suppose a request exceeds the latency budget for the given task by x amount of time, indicating the time we need to make up. Once the request completes its execution at the given worker, we identify the downstream worker to forward the request using the routing table, following our standard procedure. Let us denote the profiled execution time of this downstream worker as y. To compensate for the x time deficit, we need to find a downstream worker capable of executing within $y - x$ time to offset the exceeded budget for this task. As mentioned in Section 5.1, the Load Balancer propagates a backup table to all workers, listing downstream workers with leftover capacities. We scan this table for a worker whose profiled execution time is at most $y - x$. If there are multiple such workers, we select the one with the highest accuracy. If there is still a tie, we break it randomly. If no such worker can be found, we drop the request. Note that this entire process takes place in real-time at individual workers during query execution.

Opportunistic rerouting reduces SLO violations by preemptively saving requests from missing their SLOs and trading off accuracy for SLO fulfillment. We compare opportunistic rerouting with the naïve early dropping techniques mentioned above to study its performance benefits in Section 6.3.

6 EMPIRICAL EVALUATION

We now present our prototype implementation, experimental setup, and our empirical results.

6.1 Experimental setup

Implementation: We implement Loki in ~8K lines of Python code[5]. We use ONNX runtime [12] with the CUDA execution provider to host the host the models on GPUs for efficient inference. We use Gurobi [18] to solve our MILP optimization. Our cluster consists of 20 NVIDIA GTX 1080 Ti GPU workers.

We extend the discrete-event simulator from [5] to evaluate our system on a wide range of system parameters. This approach aligns with established practices in the field, as DNN inference is known for its high determinism [14, 44]. Previous works (e.g., [5, 29]), typically conduct a core set of experiments on an actual cluster and then compare the results obtained obtained from the cluster with those from simulation to demonstrate the quantitative differences. They then utilize simulation to investigate the impact of a wide range of parameters on the performance of the system. In line with this methodology, we also use our simulator to explore a broad range of parameters and their effects on our system's performance.

Pipelines: We consider two types of pipelines in our evaluation, both shown in Figure 2:

- *Traffic analysis.* It first detects the objects in the video frames and then runs fine-tuned car classification or facial recognition models on the detected car and person objects, respectively. We use YOLOv5 [24] as the object detection model, EfficientNet [38] for car classification, and VGG [8] for facial recognition.
- *Social media.* The social media pipeline detects objects in images and generates suggested captions for the images. It uses ResNet [20] for image classification and CLIP-ViT [31] for image captioning.

We use a total of 32 model variants in our evaluation across the two pipelines. We normalize the accuracy of each model variant in a model family by the accuracy of its most accurate variant.

Datasets: We use two input datasets.

- *Traffic data.* We use a single day from the Microsoft Azure functions trace [35] for query arrivals to drive load for the traffic analysis pipeline. We use shape-preserving transformations to scale the trace in a way that it matches the capacity of our cluster. Since this trace only contains aggregated information of request arrivals but no request content, we use images from the Bellevue traffic dataset [6] as the request content to perform inference and generate intermediate requests for subsequent tasks.
- *Social media.* We use the Twitter trace [1] used by prior inference serving systems [5, 33] to drive load for the social media pipeline. However, as the Twitter trace also contains only aggregated information about request arrivals but not request content, we use images from the MS-COCO captions dataset [9] as the content for the requests.

Evaluation metrics: We define metrics to evaluate our system.

(1) *System accuracy* is the average accuracy experienced by all requests served by the system.
(2) *Cluster utilization* indicates the ratio of workers used at any given time to the total number of workers in the cluster.
(3) *SLO violation ratio* indicates the ratio of requests that miss their SLOs.

Note that a request could miss its SLO in two ways: (i) it finishes past its SLO, (ii) it gets dropped preemptively by the system. In both cases, the system is unable to fulfill the request

Baselines for comparison: We compare Loki, the first system capable of performing pipeline-aware hardware and accuracy scaling, against two approaches.

- *InferLine* [10] is a pipeline-aware, but accuracy-agnostic inference serving system. It can perform hardware scaling but requires the clients to specify a single model variant to use for each task in the pipeline and does not support switching between model variants.
- *Proteus* [5] is an inference serving system that can scale accuracy for single models but is pipeline-agnostic. We set it up to serve inference pipelines by letting it handle each task in the pipeline independently, i.e., it scales accuracy for each task independently since it is unaware of the dependencies between them.

6.2 Performance comparison

We present an end-to-end comparison of the system performance of Loki against the baselines on the two representative pipelines.

Traffic analysis pipeline. We first study the end-to-end performance of the traffic analysis pipeline. Figure 5 shows the timeseries of the trace demand, the system accuracy for each approach, the percentage of workers used in the cluster, and the SLO violation ratio. For this experiment, we use an end-to-end pipeline latency

[5]Our code is available at https://github.com/UMass-LIDS/Loki

SLO of 250ms. We explore the sensitivity of the system to different SLO values in Section 6.4.

We show the point when Loki shifts between hardware scaling and accuracy scaling with the help of the dotted vertical lines. InferLine offers low SLO violations during the hardware scaling phase, but since it is not capable of performing accuracy scaling, its SLO violations shoot up during that time and it is not able to meet the increased demand. Therefore, compared to InferLine which performs hardware scaling alone, Loki effectively increases the capacity of the cluster by 2.5×.

Proteus consistently suffers from high SLO violations due to the fact that it is not pipeline-aware and manages each task in the pipeline graph independently. Therefore, Proteus is not able to identify the dependencies between the tasks to match the throughput of different tasks, leading to the creation of throughput bottlenecks. Therefore, Loki reduces SLO violations by 10× compared to a pipeline-unaware accuracy scaling approach such as Proteus.

As Loki performs accuracy scaling in a pipeline-aware manner, it is also able to achieve higher system accuracy than Proteus since the latter may drop accuracy for a task that may lead to a higher drop in end-to-end accuracy, while Loki uses the knowledge of end-to-end accuracies to drop minimal accuracy.

Lastly, during off-peak times, Loki can leverage hardware scaling to reduce cost and energy by allowing the system to shut down servers that are not needed. Compared to Proteus which uses the entire cluster throughout since it does not perform hardware scaling, Loki reduces the number of servers needed to serve the demand, and consequently server cost, by up to 2.67×.

To summarize, Loki offers consistently lower SLO violations due to its pipeline-aware resource allocation. It increases the effective capacity of the cluster by 2.5× in this experiment, and can shut off servers to save cost and energy during off-peak times.

Validating the simulator. We conduct this experiment on our simulator as well to validate it and observe an average difference of 1.2% in accuracy, 1.8% in the SLO violation ratios, and 1.5% in the number of servers used. We note that the simulation results are close to the prototype results, and the differences are produced due to various factors such as small variances in model execution times and unexpected network delays. However, due to the deterministic nature of ML inference and this small difference, we use our simulator to conduct the remaining experiments in order to evaluate the system under a wide range of conditions and parameters. For the rest of this Section, we present results from our simulation unless otherwise noted.

Social media pipeline. We now present the end-to-end performance comparison on the social media pipeline in Figure 6. As before, we show the incoming demand into the system, the system accuracy offered by the different approaches, cluster utilization, and SLO violation ratio.

We observe similar trends as in the traffic analysis pipeline. When demand increases to the point where hardware scaling is not able to meet it, the SLO violations of InferLine shoot up to more than 5× of Loki. During this time, Loki is able to meet demand by sacrificing ~10% accuracy.

During off-peak times, Loki again uses about 2.67× less servers than Proteus which does not perform any hardware scaling. Loki also drops up to 20% less accuracy than Proteus due to the ability of the former to identify pipeline dependencies and their effect on end-to-end accuracy of the pipeline. Proteus continues to suffer from high violations again due to being pipeline-unaware. Loki increases the effective capacity of the cluster by 2.7× in this experiment.

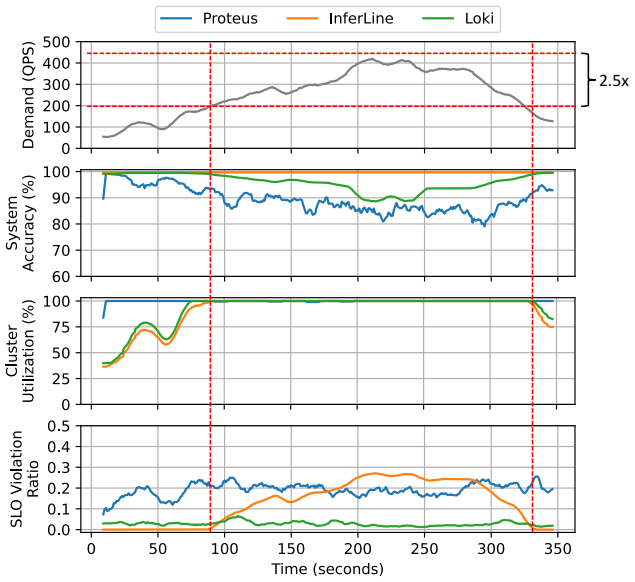


Figure 5: End-to-end comparison on the traffic analysis pipeline. Dotted vertical lines show transition between hardware and accuracy scaling. Loki achieves an increase of 2.5× in effective capacity compared to InferLine that performs hardware scaling alone and reduces SLO violations by up to 10× compared to Proteus that performs pipeline-unaware accuracy scaling.

6.3 Ablation study of the load balancer

We now take a deep dive into the request routing performed by the Load Balancer to understand where the performance benefits come from. Figure 7 shows the benefit achieved from the use of early dropping and opportunistic rerouting by comparing it against simpler versions as follows.

(1) *No early dropping:* This is the simplest version which does not perform any early dropping and follows the original routing plan.
(2) *Last-task dropping:* This version drops requests if they are expected to miss their SLOs, but only at the last task of the pipeline.
(3) *Per-task early dropping:* This version performs early dropping of requests at each task if they miss the assigned latency budget of that task.
(4) *Early dropping with opportunistic rerouting:* This is the full-fledged version of our approach that we use in our end-to-end implementation. It first tries to re-route requests through faster paths if they are expected to miss their SLO using the assigned path, and drops them if this is not possible.

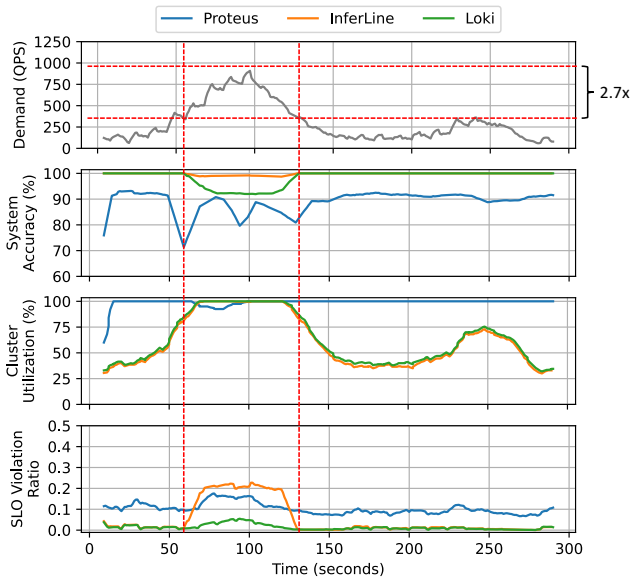


Figure 6: End-to-end comparison on the social media pipeline. Dotted vertical lines show transition between hardware and accuracy scaling. Loki achieves an increase of 2.7× in effective capacity compared to hardware scaling alone and reduces SLO violations by up to 10× compared to pipeline-unaware accuracy scaling.

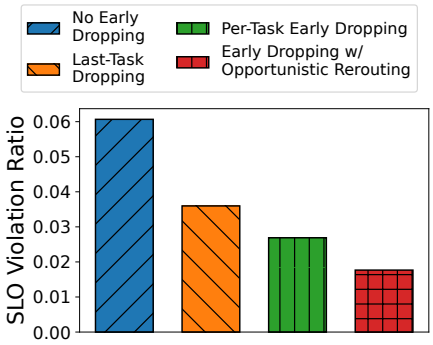


Figure 7: Ablation study of the load balancer shows that opportunistic rerouting has the most impact on SLO violations.

We observe that the version without any early dropping suffers from the highest SLO violations as it can waste resources on requests that are not on target to meet their SLOs, hence delaying and potentially timing out other requests as well. Last-task dropping improves SLO violations slightly by dropping requests if they are expected to miss SLOs, but since it only does this at the last task, it can be overly conservative in doing so and still suffers from high SLO violations. We observe that per-task early dropping improves performance further by dropping requests at each task if they are expected to miss the latency budget for the respective task.

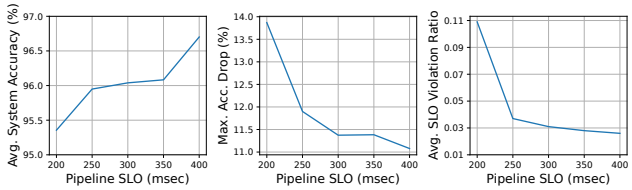


Figure 8: Effect of varying SLOs on Loki

However, this approach may drop requests too aggressively since a request that misses its latency budget for an earlier task may still potentially catch up at a later task.

Our approach, which opportunistically reroutes requests that are falling back to faster paths, minimizes SLO violations the most. If such rerouting is not possible, it means that the request has no way of meeting its SLO even if routed through the fastest path available. In this case, it drops the requests as a last resort in order to free up resources for other requests that may have a better chance of meeting their SLOs.

6.4 Effect of SLOs on system performance

We study the effect of varying the latency SLOs for the traffic analysis pipeline on the performance of Loki. To summarize the results from a large number of experiments, Figure 8 shows the following key metrics: (i) the average system accuracy across the entire experiment, (ii) the maximum accuracy drop, and (iii) the average SLO violation ratio. The maximum accuracy drop is the degradation in system accuracy from its highest possible value at peak demand.

We observe a general trend that performance improves sharply with initial increments of 50 milliseconds, but there are diminishing improvements in performance as we use larger values of SLO. This is because the optimization can use several knobs to meet tighter SLOs: (i) creating more replicas of model instances, (ii) decreasing the batch size of models in the path to lower end-to-end latency, and (iii) lowering accuracy by changing the model variant. Note that the Resource Manager can only increase the replication factor up to the point where the entire cluster is allocated, and the minimum batch size it can use is 1. Starting from 400 milliseconds, as the latency SLO gets tighter, the system can first respond by using these knobs without sacrificing any accuracy. However, when the system faces even tighter latency SLOs, once it exhausts these options, the system is compelled to resort to accuracy scaling to meet SLOs. This results in a decrease in overall system accuracy and leads to SLO violations due to the overhead associated with swapping model variants.

Below 200 milliseconds, the system cannot serve the demand even with the maximum degree of hardware and accuracy scaling because the sum of processing latencies across the entire pipeline of even the lowest accuracy model variants with a batch size of 1 exceeds this value of SLO.

6.5 Runtime performance

We now explore the runtime performance of both the core components of our system: the Resource Manager and Load Balancer.

Resource Manager. Given that the Resource Manager considers all paths through the pipeline and yields an optimal solution by solving an MILP, it is expected to run orders of magnitude slower than the Load Balancer. We measure the average runtime of the MILP to be ∼500 milliseconds. As the Resource Manager is invoked periodically to adapt to long-term fluctuations in demand and does not lie on the critical path of query execution, the observed runtime allows for a reasonably swift adaptation of resource allocation in response to changing demands.

Load Balancer As the Load Balancer reacts to run-time changes in demand, it needs to respond much faster than the Resource Manager. In our experiments, we measure the average runtime of the load balancer to be ∼0.15 milliseconds. We attribute the fast runtime of the Load Balancer to the efficiency of our request routing algorithm presented in Section 5.1.

7 RELATED WORK

Inference serving is quickly becoming a hot topic of research. Representative production systems include TensorFlow-Serving [30], NVIDIA Triton Inference Server [3] and Amazon SageMaker [2]. Inference serving has also been extensively studied through research prototypes as well, such as Clipper [11], INFless [45], and PRETZEL [25]. These systems aim to provide a unified abstraction to the user to hide details of the underlying ML frameworks, data pre-processing, and performance optimization. Unlike these systems that require users to manage DNN models, Loki automatically configures the suitable DNN models to execute on GPU clusters.

The closest works to Loki are Proteus [5] and InferLine [10]. Proteus presents an inference serving system that can scale accuracy for single models but is pipeline-agnostic. It scales accuracy for each task in the pipeline independently since it is unaware of the dependencies between them. InferLine is a pipeline-aware inference serving system that minimizes the cost of inference serving by scaling the hardware in response to changes in demand.

INFaaS [33], Sommelier [17], and Tolerance Tiers [19] are also related as they also consider model variants with different accuracy-latency profiles in model serving systems. INFaaS [33] presents a model-less inference serving system that automates the selection of model variants for each query to minimize cost while meeting accuracy and latency requirements. Unlike Loki that explicitly optimizes accuracy as an objective, it treats accuracy as a constraint and focuses on hardware scaling to handle variable demands. Sommelier [17] is a model repository that can interface with inference serving systems to suggest model variants with lower accuracy to handle increases in load. Tolerance Tiers [19] allows developers to tradeoff accuracy for latency through programming APIs. However, it imposes a fundamental limitation on applications, compelling them to adhere to a single accuracy tier statically throughout the entire inference serving process, lacking the flexibility to dynamically adjust accuracy as part of a scaling approach.

Many inference serving systems try to optimize the cost of serving while meeting certain performance constraints. Kairos [28] is one such system that aims to minimize the cost of inference serving using heterogeneous cloud resources. MArk [46] and Scrooge [21] also try to minimize the cost of inference serving while trying to meet latency SLOs. iGniter [43] is an interference-aware inference serving system that minimizes serving cost. Cocktail [16] uses model ensembling to improve accuracy and meet latency requirements using minimal cost. Loki instead optimizes both cost and accuracy by unifying accuracy scaling and hardware scaling.

Some model serving systems propose techniques that can be combined with accuracy and hardware scaling to improve system throughput. Rafiki [41] is an analytics serving system that uses model ensembling during inference to improve accuracy at the cost of latency. PERSEUS [26] studies the performance and cost tradeoffs associated with multi-tenant model serving. Morphling [40] presents an algorithmic framework to minimize the cost of searching through possible configurations when setting up inference services. Clover [27] is an inference serving system that explores the tradeoff between carbon emissions and accuracy. DeepPlan [23] minimizes inference latency by exploiting recent advances in GPU technology to reduce the model loading latencies. SHEPHERD [48] and Clockwork [15] aim to minimize the tail latency of model serving by eliminating sources of unpredictability in the system.

There has been a lot of work specifically related to video analytics pipelines. VideoStorm [47] was the first work to explore the latency-accuracy tradeoff for the resource provisioning of video analytics applications that use DNNs. Llama [34] is a serverless framework for auto-tuning video analytics pipelines. Nexus [36] is another framework for serving video analytics pipelines on GPU clusters. In comparison, Loki is a system that is applicable to generic inference pipelines that can be represented as directed rooted trees (defined in Section 2.1).

Recent work also explores serving large language models (LLMs), such as AlpaServe [29] and Tabi [42]. LLM serving is different from traditional inference serving in the sense that it often requires partitioning the model to be served by multiple servers. Loki does not feature optimizations tailored to LLMs but can cater to inference pipelines with LLMs.

8 CONCLUSION

In conclusion, our work addresses the pressing need for efficient and cost-effective deployment of machine learning (ML) inference at the edge. By recognizing the challenge posed by limited edge resources and the computational demands of ML models, we introduce Loki, a system for resource provisioning of ML inference pipelines. Central to Loki is the concept of hardware and accuracy scaling, which dynamically adjusts accuracy levels to manage resource constraints when needed, thereby enhancing the effective capacity of edge clusters and minimizing resource usage during the off-peak. Our experimental results demonstrate that Loki significantly outperforms existing inference serving systems by reducing Service Level Objective (SLO) violations by up to 10× and increasing the effective capacity by up to 2.7× while sacrificing minimal accuracy and meeting throughput targets.

ACKNOWLEDGMENTS

This work is supported in part by the National Science Foundation under grants CNS-1763617, CNS-1901137, CNS-2106463, CNS-2312396, CNS-2338512, CNS-2224054, and DMS-2220211.

REFERENCES

[1] 2018. Twitter Streaming Traces. https://archive.org/details/archiveteam-twitter-stream-2018-04.

[2] 2020. Amazon SageMaker. Build, train, and deploy machine learning models at scale. https://aws.amazon.com/sagemaker/. Accessed: 2021-06-23.

[3] 2022. Triton Inference Server. https://developer.nvidia.com/nvidia-triton-inference-server.

[4] Adobe. 2024. *Adobe Firefly*. https://www.adobe.com/products/firefly.html

[5] 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 *(ASPLOS '24)*. Association for Computing Machinery, New York, NY, USA, 318–334. https://doi.org/10.1145/3617232.3624849

[6] Romil Bhardwaj, Zhengxu Xia, Ganesh Ananthanarayanan, Junchen Jiang, Yuanchao Shu, Nikolaos Karianakis, Kevin Hsieh, Paramvir Bahl, and Ion Stoica. 2022. Ekya: Continuous Learning of Video Analytics Models on Edge Compute Servers. In *19th USENIX Symposium on Networked Systems Design and Implementation (NSDI 22)*. USENIX Association, Renton, WA, 119–135. https://www.usenix.org/conference/nsdi22/presentation/bhardwaj

[7] Davis Blalock, Jose Javier Gonzalez Ortiz, Jonathan Frankle, and John Guttag. 2020. What is the state of neural network pruning? *Proceedings of machine learning and systems* 2 (2020), 129–146.

[8] Ken Chatfield, Karen Simonyan, Andrea Vedaldi, and Andrew Zisserman. 2014. Return of the devil in the details: Delving deep into convolutional nets. *arXiv preprint arXiv:1405.3531* (2014).

[9] Xinlei Chen, Hao Fang, Tsung-Yi Lin, Ramakrishna Vedantam, Saurabh Gupta, Piotr Dollár, and C. Lawrence Zitnick. 2015. Microsoft COCO Captions: Data Collection and Evaluation Server. *CoRR* abs/1504.00325 (2015). arXiv:1504.00325 http://arxiv.org/abs/1504.00325

[10] Daniel Crankshaw, Gur-Eyal Sela, Xiangxi Mo, Corey Zumar, Ion Stoica, Joseph Gonzalez, and Alexey Tumanov. 2020. InferLine: latency-aware provisioning and scaling for prediction serving pipelines. In *Proceedings of the 11th ACM Symposium on Cloud Computing* (Virtual Event, USA) *(SoCC '20)*. Association for Computing Machinery, New York, NY, USA, 477–491. https://doi.org/10.1145/3419111.3421285

[11] 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)*. USENIX Association, Boston, MA, 613–627. https://www.usenix.org/conference/nsdi17/technical-sessions/presentation/crankshaw

[12] ONNX Runtime developers. 2021. ONNX Runtime. https://onnxruntime.ai/. Version: 1.11.0.

[13] Biyi Fang, Xiao Zeng, and Mi Zhang. 2018. NestDNN: Resource-Aware Multi-Tenant On-Device Deep Learning for Continuous Mobile Vision. In *Proceedings of the 24th Annual International Conference on Mobile Computing and Networking* (New Delhi, India) *(MobiCom '18)*. Association for Computing Machinery, New York, NY, USA, 115–127. https://doi.org/10.1145/3241539.3241559

[14] Arpan Gujarati, Sameh Elnikety, Yuxiong He, Kathryn S. McKinley, and Björn B. Brandenburg. 2017. Swayam: distributed autoscaling to meet SLAs of machine learning inference services with resource efficiency. In *Proceedings of the 18th ACM/IFIP/USENIX Middleware Conference* (Las Vegas, Nevada) *(Middleware '17)*. Association for Computing Machinery, New York, NY, USA, 109–120. https://doi.org/10.1145/3135974.3135993

[15] Arpan Gujarati, Reza Karimi, Safya Alzayat, Wei Hao, Antoine Kaufmann, Ymir Vigfusson, and Jonathan Mace. 2020. Serving {DNNs} like Clockwork: Performance Predictability from the Bottom Up. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*. 443–462.

[16] Jashwant Raj Gunasekaran, Cyan Subhra Mishra, Prashanth Thinakaran, Bikash Sharma, Mahmut Taylan Kandemir, and Chita R. Das. 2022. Cocktail: A Multi-dimensional Optimization for Model Serving in Cloud. In *19th USENIX Symposium on Networked Systems Design and Implementation (NSDI 22)*. USENIX Association, Renton, WA, 1041–1057. https://www.usenix.org/conference/nsdi22/presentation/gunasekaran

[17] Peizhen Guo, Bo Hu, and Wenjun Hu. 2022. Sommelier: Curating DNN Models for the Masses. In *Proceedings of the 2022 International Conference on Management of Data*. 1876–1890.

[18] Gurobi Optimization, LLC. 2022. Gurobi Optimizer Reference Manual. https://www.gurobi.com

[19] Matthew Halpern, Behzad Boroujerdian, Todd Mummert, Evelyn Duesterwald, and Vijay Janapa Reddi. 2019. One size does not fit all: Quantifying and exposing the accuracy-latency trade-off in machine learning cloud service apis via tolerance tiers. *arXiv preprint arXiv:1906.11307* (2019).

[20] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. 2016. Deep residual learning for image recognition. In *Proceedings of the IEEE conference on computer vision and pattern recognition*. 770–778.

[21] Yitao Hu, Rajrup Ghosh, and Ramesh Govindan. 2021. Scrooge: A Cost-Effective Deep Learning Inference System. In *Proceedings of the ACM Symposium on Cloud Computing* (Seattle, WA, USA) *(SoCC '21)*. Association for Computing Machinery, New York, NY, USA, 624–638. https://doi.org/10.1145/3472883.3486993

[22] Yanxiang Huang, Bin Cui, Wenyu Zhang, Jie Jiang, and Ying Xu. 2015. TencentRec: Real-Time Stream Recommendation in Practice *(SIGMOD '15)*. Association for Computing Machinery, New York, NY, USA, 227–238. https://doi.org/10.1145/2723372.2742785

[23] Jinwoo Jeong, Seungsu Baek, and Jeongseob Ahn. 2023. Fast and Efficient Model Serving Using Multi-GPUs with Direct-Host-Access. In *Proceedings of the Eighteenth European Conference on Computer Systems* (Rome, Italy) *(EuroSys '23)*. Association for Computing Machinery, New York, NY, USA, 249–265. https://doi.org/10.1145/3552326.3567508

[24] Glenn Jocher. 2020. *YOLOv5 by Ultralytics*. https://doi.org/10.5281/zenodo.3908559

[25] 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 *13th {USENIX} Symposium on Operating Systems Design and Implementation ({OSDI} 18)*. 611–626.

[26] 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.

[27] Baolin Li, Siddharth Samsi, Vijay Gadepally, and Devesh Tiwari. 2023. Clover: Toward Sustainable AI with Carbon-Aware Machine Learning Inference Service *(SC '23)*. Association for Computing Machinery, New York, NY, USA, Article 20, 15 pages. https://doi.org/10.1145/3581784.3607034

[28] Baolin Li, Siddharth Samsi, Vijay Gadepally, and Devesh Tiwari. 2023. Kairos: Building Cost-Efficient Machine Learning Inference Systems with Heterogeneous Cloud Resources. In *Proceedings of the 32nd International Symposium on High-Performance Parallel and Distributed Computing* (Orlando, FL, USA) *(HPDC '23)*. Association for Computing Machinery, New York, NY, USA, 3–16. https://doi.org/10.1145/3588195.3592997

[29] Zhuohan Li, Lianmin Zheng, Yinmin Zhong, Vincent Liu, Ying Sheng, Xin Jin, Yanping Huang, Zhifeng Chen, Hao Zhang, Joseph E. Gonzalez, and Ion Stoica. 2023. AlpaServe: Statistical Multiplexing with Model Parallelism for Deep Learning Serving. In *17th USENIX Symposium on Operating Systems Design and Implementation (OSDI 23)*. USENIX Association, Boston, MA, 663–679. https://www.usenix.org/conference/osdi23/presentation/li-zhouhan

[30] Christopher Olston, Fangwei Li, Jeremiah Harmsen, Jordan Soyke, Kiril Gorovoy, Li Lao, Noah Fiedel, Sukriti Ramesh, and Vinu Rajashekhar. 2017. TensorFlow-Serving: Flexible, High-Performance ML Serving. In *Workshop on ML Systems at NIPS 2017*.

[31] Alec Radford, Jong Wook Kim, Chris Hallacy, Aditya Ramesh, Gabriel Goh, Sandhini Agarwal, Girish Sastry, Amanda Askell, Pamela Mishkin, Jack Clark, Gretchen Krueger, and Ilya Sutskever. 2021. Learning Transferable Visual Models From Natural Language Supervision. arXiv:2103.00020 [cs.CV]

[32] 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).

[33] Francisco Romero, Qian Li, Neeraja J Yadwadkar, and Christos Kozyrakis. 2021. {INFaaS}: Automated Model-less Inference Serving. In *2021 USENIX Annual Technical Conference (USENIX ATC 21)*. 397–411.

[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* (Seattle, WA, USA) *(SoCC '21)*. Association for Computing Machinery, New York, NY, USA, 1–17. https://doi.org/10.1145/3472883.3486972

[35] Mohammad Shahrad, Rodrigo Fonseca, Íñigo Goiri, Gohar Chaudhry, Paul Batum, Jason Cooke, Eduardo Laureano, Colby Tresness, Mark Russinovich, and Ricardo Bianchini. 2020. Serverless in the wild: Characterizing and optimizing the serverless workload at a large cloud provider. In *2020 USENIX Annual Technical Conference (USENIX ATC 20)*. 205–218.

[36] 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

[37] Vivienne Sze, Yu-Hsin Chen, Tien-Ju Yang, and Joel S Emer. 2017. Efficient processing of deep neural networks: A tutorial and survey. *Proc. IEEE* 105, 12 (2017), 2295–2329.

[38] Mingxing Tan and Quoc Le. 2019. Efficientnet: Rethinking model scaling for convolutional neural networks. In *International conference on machine learning*. PMLR, 6105–6114.

[39] Mingxing Tan and Quoc V. Le. 2020. EfficientNet: Rethinking Model Scaling for Convolutional Neural Networks. arXiv:1905.11946 [cs.LG]

[40] Luping Wang, Lingyun Yang, Yinghao Yu, Wei Wang, Bo Li, Xianchao Sun, Jian He, and Liping Zhang. 2021. Morphling: Fast, Near-Optimal Auto-Configuration for Cloud-Native Model Serving. In *Proceedings of the ACM Symposium on Cloud Computing* (Seattle, WA, USA) *(SoCC '21)*. Association for Computing Machinery,

New York, NY, USA, 639–653. https://doi.org/10.1145/3472883.3486987

[41] Wei Wang, Jinyang Gao, Meihui Zhang, Sheng Wang, Gang Chen, Teck Khim Ng, Beng Chin Ooi, Jie Shao, and Moaz Reyad. 2018. Rafiki: machine learning as an analytics service system. *Proc. VLDB Endow.* 12, 2 (oct 2018), 128–140. https://doi.org/10.14778/3282495.3282499

[42] Yiding Wang, Kai Chen, Haisheng Tan, and Kun Guo. 2023. Tabi: An Efficient Multi-Level Inference System for Large Language Models. In *Proceedings of the Eighteenth European Conference on Computer Systems* (Rome, Italy) *(EuroSys '23)*. Association for Computing Machinery, New York, NY, USA, 233–248. https://doi.org/10.1145/3552326.3587438

[43] Fei Xu, Jianian Xu, Jiabin Chen, Li Chen, Ruitao Shang, Zhi Zhou, and Fangming Liu. 2023. iGniter: Interference-Aware GPU Resource Provisioning for Predictable DNN Inference in the Cloud. *IEEE Transactions on Parallel and Distributed Systems* 34, 3 (2023), 812–827. https://doi.org/10.1109/TPDS.2022.3232715

[44] Feng Yan, Olatunji Ruwase, Yuxiong He, and Evgenia Smirni. 2016. SERF: Efficient Scheduling for Fast Deep Neural Network Serving via Judicious Parallelism. In *SC '16: Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*. 300–311. https://doi.org/10.1109/SC.2016.25

[45] Yanan Yang, Laiping Zhao, Yiming Li, Huanyu Zhang, Jie Li, Mingyang Zhao, Xingzhen Chen, and Keqiu Li. 2022. 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* (Lausanne, Switzerland) *(ASPLOS '22)*. Association for Computing Machinery, New York, NY, USA, 768–781. https://doi.org/10.1145/3503222.3507709

[46] Chengliang Zhang, Minchen Yu, Wei Wang, and Feng Yan. 2019. {MArk}: Exploiting Cloud Services for {Cost-Effective},{SLO-Aware} Machine Learning Inference Serving. In *2019 USENIX Annual Technical Conference (USENIX ATC 19)*. 1049–1062.

[47] 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)*. USENIX Association, Boston, MA, 377–392. https://www.usenix.org/conference/nsdi17/technical-sessions/presentation/zhang

[48] 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)*. USENIX Association, Boston, MA, 787–808. https://www.usenix.org/conference/nsdi23/presentation/zhang-hong