electronics

Article

HetSev: Exploiting Heterogeneity-Aware Autoscaling and Resource-Efficient Scheduling for Cost-Effective Machine-Learning Model Serving

Hao Mo [1], Ligu Zhu [1,2,*], Lei Shi [1,*], Songfu Tan [1] and Suping Wang [1]

[1] State Key Laboratory of Media Convergence and Communication, Communication University of China, Beijing 100024, China
[2] Beijng Key Laboratory of Big Data in Security & Protection Industry, Beijing 100024, China
* Correspondence: zhuligu@cuc.edu.cn (L.Z.); leiky_shi@cuc.edu.cn (L.S.)

Abstract: To accelerate the inference of machine-learning (ML) model serving, clusters of machines require the use of expensive hardware accelerators (e.g., GPUs) to reduce execution time. Advanced inference serving systems are needed to satisfy latency service-level objectives (SLOs) in a cost-effective manner. Novel autoscaling mechanisms that greedily minimize the number of service instances while ensuring SLO compliance are helpful. However, we find that it is not adequate to guarantee cost effectiveness across heterogeneous GPU hardware, and this does not maximize resource utilization. In this paper, we propose HetSev to address these challenges by incorporating heterogeneity-aware autoscaling and resource-efficient scheduling to achieve cost effectiveness. We develop an autoscaling mechanism which accounts for SLO compliance and GPU heterogeneity, thus provisioning the appropriate type and number of instances to guarantee cost effectiveness. We leverage multi-tenant inference to improve GPU resource utilization, while alleviating inter-tenant interference by avoiding the co-location of identical ML instances on the same GPU during placement decisions. HetSev is integrated into Kubernetes and deployed onto a heterogeneous GPU cluster. We evaluated the performance of HetSev using several representative ML models. Compared with default Kubernetes, HetSev reduces resource cost by up to 2.15× while meeting SLO requirements.

Keywords: inference serving; autoscaling; cost effectiveness; multi-tenant inference

Citation: Mo, H.; Zhu, L.; Shi, L.; Tan, S.; Wang, S. HetSev: Exploiting Heterogeneity-Aware Autoscaling and Resource-Efficient Scheduling for Cost-Effective Machine-Learning Model Serving. *Electronics* **2023**, *12*, 240. https://doi.org/10.3390/electronics12010240

Academic Editor: Yue Wu

Received: 25 November 2022
Revised: 16 December 2022
Accepted: 26 December 2022
Published: 3 January 2023

1. Introduction

Machine-learning (ML)-based solutions have been permeating almost every field, and their deployment is wide and deep, leading to an emerging demand for ML as a service (MLaaS) platforms such as Amazon ML, Google Cloud ML, and Microsoft Azure ML [1–3]. The typical workflow of these MLaaS platforms consists of two phases: training and inference. In the training phase, developers create and train ML models based on the training dataset and publish them as online cloud services. In the inference phase, end users and client applications send inference requests to these services with supplied inputs, and the trained models perform inference (prediction) [4]. The prediction is typically the application of sub-second latency service-level objectives (SLOs) [5] and must be performed in real-time, resulting in the increased adoption of powerful hardware accelerators such as GPUs in clusters [6,7].

The conventional approach to turning ML models into prediction services is to host the models on a serving system. An important goal for such a serving system under dynamic workload is the ability to meet latency service-level objectives (SLOs) in a cost-effective manner. Achieving such cost effectiveness can be challenging due to the heterogeneity of resources. This is because MLaaS providers offer various GPU instance options (e.g., Nvidia V100, P100 and T4) coupled with different pricing models, and there is no general guidance on choosing the appropriate instance type across heterogeneous GPU hardware when autoscaling dynamic workloads.

Prior autoscaling approaches to achieve cost effectiveness for ML model serving [5,6,8] typically autoscale service instances based on monitored or predicted metrics. For example, works [5,6] make predictions about the request arrival rates over short time periods, based on which they proactively autoscale service instances to avoid over-provisioning. However, these approaches do not work well for ML model serving in heterogeneous GPU clusters. For one, they mainly focus on minimizing the number of service instances without considering the heterogeneous execution environments, or considering only heterogeneous hardware resources such as CPUs and GPUs, rather than heterogeneous GPU types such as Nvidia V100, P100, and T4. Further, they assume that each service instance is exclusively bound to a GPU device, since existing popular cluster managers such as Kubernetes [9] prohibit the explicit use of GPU sharing (i.e., only allowing one ML instance to be assigned to each GPU). This is problematic because their GPU under-utilization decreases serving throughput and resource efficiency, requiring additional GPU devices to meet demand.

The ability to co-locate ML instances on the same GPU (i.e., multi-tenant inference) has proven to be a solution to address under-utilization [10–13]. MLaaS providers also tend to adopt such multi-tenant inference due to practical cost considerations (e.g., power consumption, cost of hardware). While co-locating ML instances on the same GPU can improve GPU utilization, it also creates the problem of latency performance degradation (which we refer to as inter-tenant interference) [10,14–16]. Inter-tenant interference can be more challenging to deal with when workloads are not static, but dynamic [17]. Therefore, attention should be paid to addressing inter-tenant interference between ML instances sharing the same GPU during co-location placement. In this paper, we propose HetSev: a heterogeneity-aware and resource-efficient inference serving system. Given the heterogeneous GPU devices used to host instances of each ML service, HetSev guarantees the appropriate type and number of GPU instances according to monitored metrics (e.g., throughput, latency), adding new co-located instances as needed, and removing unnecessary instances. HetSev takes advantage of multi-tenant inference to achieve high GPU utilization while avoiding latency performance degradation. Our approach enables scalable service instances co-running on expensive GPU devices to further cut costs for ML inference services, in contrast to current approaches [5,6,18] which simply focus on achieving novel autoscaling mechanisms. HetSev is designed as a set of extended components, including a scaling controller integrated with our custom autoscaler and an instance controller integrated with our co-location scheduler. We integrated HetSev into Kubernetes [9] with a plugged backend model server—Tensorflow Serving [19], a high-performance ML model server for production environments. For comparison, we simulated a serving system as the baseline which deploys a Tensorflow Serving model server with the default Kubernetes. We evaluated HetSev with several representative ML models for image classification, object detection, and language translation: ResNet50 [20], Inception-v3 [21], SSD-ResNet50 [22], and Transformer [23]. The results show that HetSev decreases resource cost by up to 2.15$\times$ the baseline, while meeting the predefined SLO requirements.

This article is organized as follows: Sections 2 and 3 discuss the background and related work, and multi-tenant inference characterization study, respectively. Sections 4 and 5 outline the design and implementation details of the HetSev system. Section 6 presents the experiment setup and evaluation results. Section 7 provides the conclusions.

2. Background and Related Work

In this section, we provide a brief overview of ML inference serving and autoscaling techniques and investigate the cause of poor resource efficiency when autoscaling ML inference services. We also provide background information on multi-tenant inference optimization, which is used in our approach to help deploy ML inference services more efficiently.

2.1. Machine-Learning Inference Serving

Machine-learning inference serving is an online prediction service for low-latency deep neural network (DNN) model inference. End users of services send inference requests to these trained models using mobile applications or interactive webs through REST APIs. Based on the input data, these trained models infer the results and return them to end users. As an example, an end-to-end vocabulary speech-recognition application transcribes spoken words into text [24].

The conventional approach to deploying an ML prediction service is to provision a container, and within the container, host the ML model on a serving system. For example, serving systems such as Seldon Core [25], MXNet Model Server [26], TensorFlow Serving [19] and Clipper [4] host the model in a Docker [27] container to enable process isolation. The serving system is analogous to a webserver and exposes services with interfaces via HTTP or gRPC protocols. To provide low-latency inference services, the serving system employs some model-agnostic optimizations such as caching and batching [4].

However, existing serving systems mainly focus on the ease of model deployment and do not address the scalability and cost-effectiveness issues of serving ML models in clusters. Tensorflow serving [19] provides production environments for Tensorflow models as well as other types of models (e.g., ONNX models [28]), while relying on a cluster manager (e.g., Kubernetes [9]) for scaling. Swayam [5] is an autoscaling framework which focuses on resource efficiency and SLO compliance challenges for ML inference serving. Yet Swayam does not work well for heterogeneous GPU clusters due to not considering GPU heterogeneity. The objective of MArk [6] is to achieve low-latency, cost-effective inference among various public cloud-service options, while our goal is to reduce the GPU resource cost in heterogeneous GPU clusters. INFaaS [29] is an inference serving system which addresses resource scalability and cost efficiency on heterogeneous clusters through techniques including the combining of horizontal and vertical autoscaling and sharing hardware resources. However, it fails to consider performance interference when sharing resources across models. In contrast, HetSev meets latency SLO requirements at low cost by selecting the cheapest option among heterogeneous GPU instances and supporting multi-tenancy, while considering inter-tenant interference.

2.2. Autoscaling and Resource Efficiency

Autoscaling is a systematic technique to automatically adapt to workload changes by provisioning and de-provisioning resources [30]. A commonly used type is horizontal autoscaling, which involves adding and removing instances of resources. Another type is horizontal autoscaling, which refers to adding and removing resource capacity from existing instances and is out of the scope of this paper. Autoscaling has been studied extensively, producing a large body of work. Generally, there are two scaling methods to serve dynamic workloads.

Resource-based autoscaling. This method uses resource-based metrics (e.g., utilization, duty cycle) as its trigger to autoscale serving instances. Resource-based autoscaling is employed by many industrial MLaaS platforms, e.g., Kubernetes in Google Cloud [31], and SageMaker in AWS [32]. The autoscaling in these serving platforms follows some specified rules such as "increasing instances if CPU utilization reaches 80%" or "decreasing instances when GPU duty cycle stays below 60% for a certain amount of time". While resource-based autoscaling for CPU clusters has been intensively studied [33,34], there are few works on resource-based autoscaling for ML inference services hosted in GPU clusters. This is partly because most Kubernetes platforms, unlike industrial serving platforms such as Google Cloud, do not provide GPU resource-based metrics such as GPU duty cycle. Therefore, this method is mainly adopted in industrial serving platforms.

Request-based autoscaling. This method typically adjusts resource provisioning based on predicted or monitored request-based metrics (e.g., the request rate, latency). For example, works [5,6] implement an autoscaling mechanism which adjusts the resources based on the predicted request rate. They scale out serving instances when the predicted request rate exceeds the user-defined threshold, and scale in unused instances for resource efficiency. However, predicting workload over short time periods is intensive and takes longer to complete (seconds to minutes), which depends on the workload complexity and the metrics measured. The time taken to perform such predictions results in a negative impact on latency-sensitive ML inference services. In contrast, monitored request-based autoscaling makes no prediction and is easy to implement, and is, therefore, adopted in our approach.

However, obtaining high resource efficiency while autoscaling ML inference service in GPU clusters is a challenge task. Prior works to improve resource efficiency mainly focus on autoscaling mechanisms [5,6,18,35] or optimizing DNN inference [10,14,16,19,36–38] through scheduling GPU usage, model fragmentation and batching input data, ignoring the benefits of multi-tenant inference optimization. An important cause of such a situation is the reliance on the traditional cluster scheduler, which requires each serving instance to have exclusive access to the GPU device. This is problematic because it negatively affects resource efficiency and serving throughput. Existing approaches have demonstrated that multi-tenant inference has a positive improvement on GPU-cluster resource utilization [7,39–41], while its effectiveness can be affected by inter-tenant interference.

2.3. Multi-Tenant Inference

Multi-tenant inference is a multi-instance single-device computing paradigm wherein multiple ML instances co-run on one high-performance hardware, targeting improving hardware utilization and serving throughput. Multi-tenant inference optimization is introduced mainly due to the mismatch between the tremendous computing capability of recent GPUs (e.g., NVIDIA Tesla V100 with 130 TFLOPs/s) and the general ML models' inference requirements (e.g., ResNet50 model with 4 GFLOPs). Executing such a single ML model on modern GPU can result in the severe resource under-utilization of the cluster [42]. Therefore, cluster-level ML model serving also prefers multi-tenant inference optimization due to resource-efficiency considerations. Whilst model serving in clusters benefits a lot from the multi-tenant inference optimization, it also suffers from latency performance degradation from inter-tenant interference [15]. While ML job managers that allow for multi-tenancy now exist [7,39,40], researchers have paid less attention to actively solving interference between ML jobs which share hardware resources during placement decisions.

Interference is a systematic phenomenon which occurs when multiple processes compete for the same set of limited resources on the same machine [43–45]. GPU inter-tenant interference can also occur for the same reason. Specifically, ML models consist of many GPU kernels, and a limited set of computing units and memory cause queuing delays for model kernels [46–48]. These kernels are launched by the GPU kernel scheduler, which follows a policy similar to a round-robin fashion [49]. It has been proved that inter-tenant interference of co-located ML models inevitably leads to latency performance degradation [39,50,51]. Our experimental study in Section 3.2 also demonstrates that models experience latency increase when co-located with other models. Therefore, on the one hand, MLaaS providers and consumers will typically set a certain latency constraint (SLO) which require responses to requests within a given latency (e.g., less than 400 ms for image recognition for a smooth user experience), and, thus, multi-tenant inference with some latency increase in SLO range is considered acceptable. On the other hand, in order for a serving system to take full advantage of multi-tenant inference, maximizing resource utilization and serving throughput, the cluster scheduler should consider the effects of multi-tenant interference when performing ML instance co-location.

3. Multi-Tenant Inference Characterization Study

In this section, we conducted a set of experimental studies to address the following questions: (Q1) How effective is multi-tenant inference in improving resource utilization and throughput? (Q2) How severe is latency performance degradation due to inter-tenant interference, and how can we avoid it as much as possible?

Profiling Setup. We chose four representative ML models, ResNet50 [20], Inception-v3 [21], SSD-ResNet50 [22], and Transformer [23], for common prediction tasks such as image classification, object detection, and language translation. We leveraged TensorFlow Serving [19], a popular production ML model server, to host each model in a separate Docker [27] container. Inference jobs from simulated clients are submitted to the TensorFlow Serving. The TensorFlow Serving then runs each inference job on the hosted model (e.g., SSD model) using data provided by clients (e.g., images), and returns the results (e.g., the objects' classes and bounding boxes) to clients. Multiple clients instantiate multiple inference jobs concurrently, and TensorFlow Serving interleaves their execution. Each experiment was repeated multiple times to ensure metric consistency. The detailed descriptions of the ML models and hardware specifications are provided in Section 6.1.

3.1. Effectiveness of Multi-Tenant Inference

In response to (Q1), we performed an experimental study in which multiple ML-instance replicas co-run on a single GPU. We ran all four ML models in turn and observed each model's serving throughput and GPU utilization as the number of ML instance replicas increased.

As shown in Figure 1, the co-running instance replicas for each ML model achieve 1.85× higher serving throughput and 2.76× higher GPU utilization, on average, as they increase from 1 to 3. The rise in bars implies that the inference execution of increased instances can utilize the residual GPU resource to improve serving throughput. Moreover, we observe fewer improvements in serving throughput and GPU utilization when increasing ML instance replicas from 2 to 3 compared to increasing from 1 to 2. This is intuitive, as more co-located instances exhibit severe competition for available GPU resources, and it is reasonably predictable that there will be few or no performance improvements as instance replicas continue to increase. The correlation of the increase in the number of instances with the mitigation of performance improvements suggests that the number of co-located instances on a single GPU should be carefully configured considering multiple factors, such as the capacity of hardware, the ML model size, and inter-tenant interference.

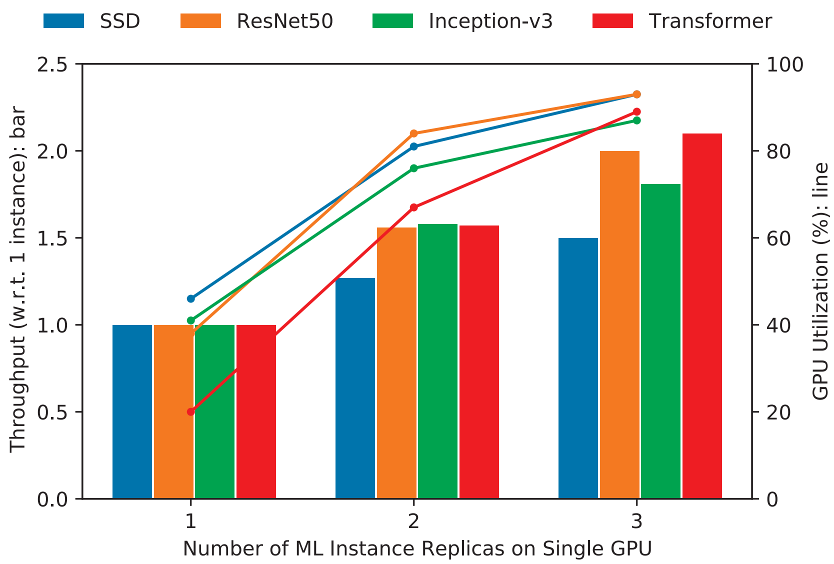

Figure 1. The serving throughput and GPU utilization with respect to the number of ML-instance replicas on a single GPU.

3.2. Interference of Multi-Tenant Inference

In response to (Q2), we performed an experimental study on 160 synthesized ML-instance co-location combinations on a single GPU. We built these instance combinations with the pairs of two ML models drawn from the four ML models, including identical

ML-instance co-location combinations and different ML-instance co-location combinations, and associated these instance combinations with four different batch sizes (i.e., 2, 4, 8, 16) to create a total of 160 pairs of instance combinations. We ran these instance combinations and observed the inference latency.

Figure 2 illustrates the cumulative distribution function (CDF) of latency increase relative to solo-running latency. As noted in the figure, for up to 90% of the combinations, the average latency increase is only 18%, indicating a huge potential for enhancing serving throughput and GPU utilization. Therefore, it is worthwhile to adopt multi-tenant inference optimization as long as the latency increase is kept in the SLO range.

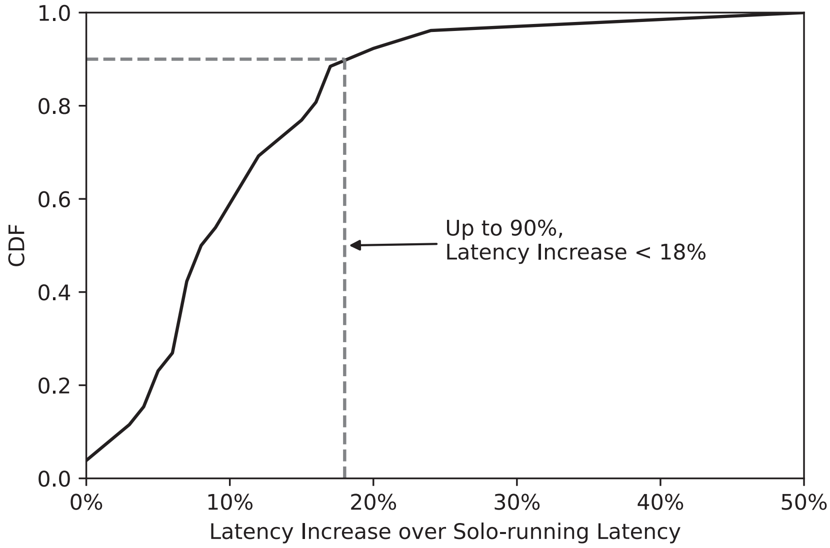

Figure 2. Cumulative distribution of latency increase relative to solo-running latency when bi-instance inference executes on a single GPU. Among all ML-instance co-location combination experiments, e.g., up to 90-percent of bi-instance execution show less than 18% latency increase. Models: ResNet50, Inception-v3, SSD-ResNet50, Transformer. Batch size: 2, 4, 8, 16. In total: 160 combinations, including identical ML-instance co-location combinations and different ML instance co-location combinations.

Additionally, we observed that the CDF exhibits a long tail in Figure 2, suggesting that the inter-tenant interference effect could be severe in some combinations. Thus, we further investigated the impact of the type of co-located ML instances on latency increase. Figure 3 shows latency increase compared to solo-running latency when bi-instance inference is executed on a single GPU. It reveals that co-running identical ML instances on a single GPU results in a larger latency increase than co-running different instances. This is because ML models contain many types of operators such as full-connection, convolution, activation, batch-normalization, etc.; different types of operators may be either computation intensive (e.g., full-connection, convolution) or memory intensive (e.g., activation, batch-normalization). Therefore, co-running identical ML models on the same GPU will be more likely to result in competition for computational or memory resources due to poor operator concurrency management than co-running different ML models (i.e., increased probability of co-running the same type of operators at the same time). As a result, the inference of both models slows down, and this will happen frequently as the number of tenants increases, thus reducing the overall serving throughput. Motivated by this insight, we devised a scheduling mechanism (Section 4.2), which avoids the co-location of identical ML instances to alleviate inter-tenant interference.

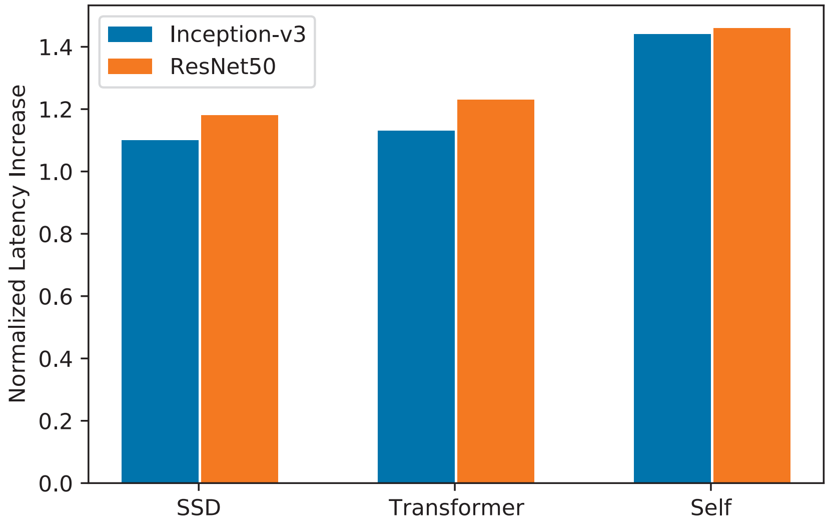

Figure 3. Latency increase compared to solo-running latency when bi-instance inference executes on a single GPU. Inception-v3 and ResNet50 instances experience the largest latency increase when co-located with selves.

4. Our Approach: HetSev

HetSev is a heterogeneity-aware and resource-efficient inference serving system designed as a set of components which can be deployed on existing cluster managers (e.g., Kubernetes) as extended operators. Figure 4 describes HetSev architecture, which consists of three main components: the scaling controller, the instance controller, and the metric repository. Upon the arrival of requests from clients, the scaling controller periodically updates request metrics (e.g., request arrival rate and latency) from the metric repository, and adopts the custom autoscaling mechanism to perform scaling actions based on monitored metrics. Once a scaling out action is performed, the scaling controller sends the adding new instances request to the instance controller, in which the scheduler assigns instances to GPUs following the principle—meeting the resource requirements while avoiding the co-location of identical ML instances on the same GPU.

HetSev employs a resource and workload monitor to obtain a cluster view and observe inference workloads. A cluster view is maintained through per-node monitoring agents to collect infrastructure information from each node, including CPU utilization, memory usage, and GPU usage. For each hosted service, inference workloads are tracked through the workload monitor to collect metrics such as throughput and latency SLO. Referring to Swayam [5], we set two common SLO requirements as our overall objectives. (1) Response-time threshold: a request is served within a specified time, denoted RT_{max}. (2) Service level: The service is considered satisfied only if at least SL_{min} percent of the requests are finished within the response-time threshold, where SL_{min} represents the desired service level. Eventually, this infrastructure and workload information is aggregated into the metric repository.

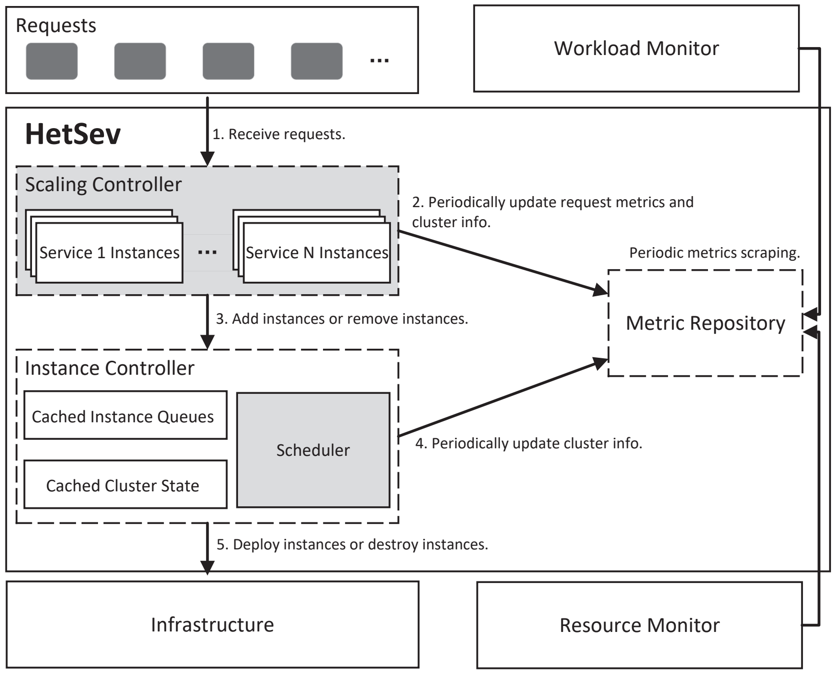

Figure 4. HetSev architecture. The boxes with dashed borders are the three main components of HetSev. The gray boxes are the controller-specific components which were re-implemented in our approach and deployed within a cluster manager (e.g., Kubernetes). The numbered arrows correspond to the typical workflow of HetSev.

4.1. Heterogeneity-Aware Instance Autoscaling

Given the diverse options for GPU instance types (e.g., Nvidia V100, T4, K80) in the cloud environment, HetSev essentially orchestrates a heterogeneous GPU cluster in response to changing demand. To react to dynamic workload, HetSev's scaling controller needs to decide the appropriate instance types and their numbers to use while minimizing the GPU resource cost of running instances.

Formulation.To figure out the type and number of instances required, we formulated the following integer linear programming (ILP) which determines a scaling action (scaling in, or scaling out). For an inference service, our ILP formulation's optimization variable (the outcome) is the optimal scaling action, X_i, for each instance type $i \in I$. X_i is an integer representing the scaling action as follows: (a) a negative value indicates scaling in instances of this type, (b) a positive value indicates scaling out instances of this type, (c) and a value of zero indicates no scaling needed. This ILP problem can, thus, be defined as follows:

$$\textit{minimize} \quad \sum_{i \in I} [C_i(X_i + \lambda O_i^{launch} max(X_i, 0))]$$

$$\textit{subject to} \quad \sum_{i \in I} T_i(N_i + X_i) \geq W, \forall i \in I \tag{1}$$

$$\sum_{i \in I} R_i^{type}(N_i + X_i) \leq R_{total}^{type}, \forall i \in I \tag{2}$$

$$N_i + X_i \geq 0, \forall i \in I \tag{3}$$

$$L_i^{inf} \leq S, \forall i \in I \tag{4}$$

where (a) I: the set of instance types available for model serving. (b) C_i: the hardware cost per unit time for running instance type i. (c) O_i^{launch}: the launching latency (i.e., the latency incurred during the instance-launching period, which spans instance launching and its readiness) of instance type i. (d) λ: a tunable parameter for the inference workload unpredictability. A large λ value places more weight on minimizing launching latency to meet SLO requirements when the inference workload is fluctuating. A small λ value places more weight on minimizing the hardware cost when the inference workload is stable. (e) T_i: the capacity of instance type i, measured by the maximum throughput of the service model with offline profiling. (f) N_i: the number of running instance type i. (g) W: the arrival rate of the inference workload. (h) R_i^{type}: the resource requirement of instance type i for a resource type (GPU memory, CPU memory, CPU cores), measured with offline profiling. (i) R_{total}^{type}: the summed amount of the residual resources of a type (GPUs, CPUs) on the underlying server machine, obtained from the metric repository. (j) L_i^{inf}: the inference latency of instance type i, measured by the execution time of the service model with offline profiling. (k) S: The SLO of the considered inference service.

The objective function that our ILP formulation minimizes is the total GPU resource cost of scaling actions for all instance types to satisfy the incoming inference workload. Multiple constraints must be satisfied. (1) With the selected scaling actions, HetSev can support the incoming inference workload. (2) The resource consumed by all running instances cannot exceed the systematic summed resources. (3) The number of running instances of each hosted service is non-negative. (4) The newly launched instances satisfy inference services' SLOs. However, since this ILP problem is NP-complete and due to the heterogeneity of instance types, we modified our cost-based algorithm with reference to [52] and turned to a heuristic algorithm to greedily solve this ILP problem.

A heuristic algorithm. We designed a heuristic algorithm which considers both the cost of running instances and the launching latency of new instances. Specifically, HetSev runs a scaling controller which approximates the ILP problem along with the instance selection policy as follows: (a) determine if there is a risk that the constraint will be violated, (b) consider two scaling actions, scaling out or scaling in, to meet the constraints, and (c) calculate the objective for each instance type with the chosen scaling action and select the one that minimizes the objective resource cost function.

Scaling-out algorithm. To decide if there is a need to scale out for an inference service (Constraint 1), the scaling controller computes the throughput saturation of all running instances, given the profiled values of their throughput capacity and the current workload they are serving. We calculate the current workload served by all running instances using the batch size and summed number of requests served per second. In practice, the saturation parameter is predefined and tunable for specific model service and SLO requirements. When the throughput saturation of all running instances exceeds the threshold for a period of time, the scaling controller concludes that we need to scale out. In this context, we set the threshold to 80% based on experience, leaving room to absorb sudden load spikes. We then proceed to the next stage: determining what instance to launch to meet the incoming inference workload.

To determine which instance to launch, the scaling controller uses an instance selection policy to select the cheapest option among all instance types. The selection policy works as follows. First, the selection policy filters the available instance types which can meet the SLO (Constraint 4) and resource requirements (Constraint 2). Second, the selection policy estimates the scaling cost of each available instance type by estimating the number of each available instance types that would be increased to satisfy the incoming request load (Constraint 1, 3). Finally, the selection policy calculates the cost function of our ILP formulation, by using the instance launching latency and the hardware cost to determine the instance type that supports higher throughput.

Scaling-in algorithm. To decide if there is a need to scale down and which instance to destroy, the scaling controller uses a selection policy which follows a similar algorithm for scaling out, as explained above. At regular intervals, the selection policy checks whether the incoming inference workload can be accommodated by removing a running instance. Note that we do not switch a running instance to the cheaper hardware even though they may not be the most cost-effective instance type in the next time step, since additional launching latency will be introduced. To ensure better service quality, the scaling controller should wait for a predefined cool-down period before executing the destroying request for an instance.

4.2. Resource-Efficient Instance Scheduling

At the core of our resource-efficient scheduling is to maximize GPU utilization via co-location placement decisions while minimizing latency performance degradation resulting from interference. We designed an algorithm, Algorithm 1, which can satisfy the constraints of resource requirements (e.g., CPU cores, CPU memory, GPU memory) of the ML instance and avoid the co-location of identical ML models on the same GPU when performing co-location placement. The resource requirements of each ML instance are obtained using offline profiling with the real workload. As co-locating identical ML models on the same GPU will result in severe inter-tenant interference, as described in Section 3.2, we introduce the locality constraints represented as a set of anti-affinity labels to avoid this in Algorithm 1. When allocating resources to instances, we borrowed some ideas from [53]. Unlike [53], which uses clustering and classifying to predict the load information for new cloud disks, we cluster instances into several groups according to their resource requirements to schedule different instances fairly. The algorithm has three steps.

Algorithm 1 Resource-efficient instance scheduling.

```
Input: I: pending instances awaiting scheduling.
    S: current cluster state.
    k, β: k queues and β instances to put into the buffer for each scheduling round.
Output: G: the assigned GPU device.
1: Q ⇐ Put pending instances into k queues via k-means (I, k)
2: while queues in Q is not empty do
3:   Ĩ ⇐ Pick β instances into scheduling buffer via weighted fairness
4:   for i in Ĩ do
5:     if the cluster has residual resources(S) then
6:       // resource capacity check (CPUcores, CPUMems, GPUMems)
7:       N ⇐ preselect all GPU devices passing resource capacity check (i, S)
8:       // anti-affinity labels check
9:       N ⇐ filter preselected GPU devices passing anti-affinity labels check (i, N)
10:      if N == Null then
11:        continue
12:      end if
13:      G ⇐ select the GPU device from N
14:      reture G
15:    end if
16:  end for
17: end while
```

Step 1 (line 1): We perform a clustering procedure on all instances before they are actually scheduled. We aggregate similar instances into several groups and manage the instances in a group separately. Specifically, we leverage the L1 Distance metric to identify similar instances based on the following features: (1) GPU memory; (2) ML model. These features characterize the resource requirements and model architecture for each instance and can be obtained through offline profiling. In practice, we use the k-means algorithm to identify similar instances among all instances awaiting scheduling and place them in the corresponding queues. The reason we use the K-means clustering algorithm is its fast convergence, which is very important for schedulers considering that the scheduling process is latency sensitive.

Step 2 (line 2–3): In each round of scheduling, we fairly select a certain number of instances from different queues as a batch, mainly considering queue length and waiting time. The core idea behind this is to avoid the starvation of any class of instances—whenever a class of instances begins to starve, it is expected that more instances will be picked up and processed from queues with larger queue lengths and longer waiting times. Here, instance starvation is the number of instances pending represented by the queue weight, which is the product of the queue length and the instance's median waiting time per queue.

Step 3 (line 4–17): We assign the most suitable GPU device to launch each instance in the cluster. Now all pending instances are sorted into $\tilde{I}$ in the weighted fairness order, the scheduler will try to allocate available resources to each instance in turn, while avoiding the interference. Specifically, for each instance, we first check the resource capacity and preselect all the GPU devices that satisfy all the requirements of the instance in terms of CPU cores, CPU memory, and GPU memory constraints, and then a candidate GPUs list is constructed (line 7). We then check the anti-affinity labels, filter out the GPU device that cannot satisfy the locality constraints of anti-affinity, and update the candidate GPUs list (Lines 9). Finally, we use a worst-fit algorithm to pick the most suitable GPU before the final resource allocation. The intuition behind this is to preferentially utilize the GPUs without any workload, thus avoiding inter-tenant interference as much as possible.

5. System Implementation

As illustrated in Figure 4, our HetSev inference serving system contains a scaling controller, instance controller, and metric repository. We implemented the scaling controller and instance controller in about 8k+ lines of Go code. These controllers communicate through remote procedure calls (RPC) and run as separate processes within the GPU cluster, i.e., in Kubernetes [9], our scaling controller and instance controller is a pod. We use the gRPC library as the underlying RPC implementation, allowing our scaling controller to send provision requests to the instance controller when a scaling decision is made. We also utilize a centralized time-series database for the metric repository. Below, we provide more details for a better understanding.

Scaling controller. The scaling controller is implemented based on a change to Horizontal Pod Autoscaling (HPA), a built-in Kubernetes resource which automatically increases or decreases the number of pod replicas to accommodate dynamic workloads. The scaling controller integrates a custom event-driven autoscaler which can both detect if the instance controller should be activated or deactivated, and poll custom metrics (e.g., throughput, latency, and resource utilization) from the metric repository periodically. As a result, the scaling controller will monitor those metrics and, based on the events that occur, it will automatically scale instances out or in accordingly.

Instance controller. The instance controller is implemented based on the build-in Kubernetes Deployment resource integrated with our co-location scheduler. The purpose of an instance controller is to maintain a stable set of replicated instances running at any given time. Thus, it guarantees the availability of a customized number of instances. The instance controller periodically polls cluster states (e.g., CPU memory usages, CPU utilization, and GPU usages), and stores them in ETCD, a built-in distributed key-value store in Kubernetes. When an adding-instances request is received, our scheduler directly reads cluster resource information stored in ETCD and determines the mapping between instances and GPUs according to the scheduling mechanism described in Section 4.2.

Monitoring. Monitoring is the key to instance scaling and scheduling in our approach. In order to collect requests' metrics information within the cluster, HetSev utilizes Istio [54], a popular tool to build service mesh, in our implementation. Istio directly attaches a sidecar proxy to each serving instance. Whenever passing data to a serving instance, the sidecar intercepts all network communication, and generates metrics for all service traffic in and out. These metrics provide information on behaviors such as the overall traffic volume and the response time for requests. To obtain a fine-grained view of GPU devices in the cluster, HetSev deploys the Nvidia DCGM-Exporter [55] container on each individual node reporting GPU usages. Finally, Prometheus [56] is used to collect all these metrics and stores them in a time-series database, which our scaling controller and scheduler can query and use to make decisions.

6. Evaluation

We first compare the overall performance of HetSev with Kubernetes [9] under the production workload from Twitter trace. To further demonstrate the effectiveness of HetSev's autoscaling mechanism, we then examine whether HetSev is able to scale instances to accommodate fluctuating workload. Finally, we compare the resource cost effectiveness of HetSev with Kubernetes [9].

Testbed. We deployed HetSev onto a cluster of eight GPU servers, with each node containing two Intel Xeon Silver 4210R × 10 Core Processors (2 threads per core) and 62 GB DDR4 memory. Among the eight GPU servers, four servers have two Nvidia Tesla V100 GPUs (32 GB GPU memory) each, and the other four servers have two Nvidia T4 GPUs (16 GB GPU memory) each. On each node was installed a Ubuntu Server 18.04 LTS and connected by 10 Gb Ethernet network. All GPUs were powered by Nvidia driver 470.103, CUDA 11.4, and cuDNN 8. The GPU cluster was managed by Kubernetes 1.22 and Docker version 1.18.3 was leveraged to provide containers.

ML models. Table 1 summarizes the four ML models used in our evaluation. These models with multiple sizes and coving diverse domains were deployed on an ML model server, i.e., Tensorflow Serving. To configure the batching of the ML models on the testbed cluster, we conducted lightweight profiling considering the model-specific response time, which is measured by the time span between the client sending the request and receiving the response. The response time contains the inference latency, which depends on the model-inherent computational complexities and batch size, as well as additional overhead due to client–server communication. Therefore, the larger the batch size we use, the longer the SLO response time becomes. As a result, the batch sizes we used were 8 for ResNet50 [20], 8 for Inception-v3 [21], 4 for SSD-ResNet50 [22], and 2 for Transformer [23], since using larger batch sizes results in unrealistically long response times.

Table 1. ML models used in the evaluation.

Model	Type	Size	Input Data
ResNet50	Image classification	90 MB	ImageNet
Inception-v3	Image classification	83 MB	ImageNet
SSD-ResNet50	Object detection	77 MB	COCO
Transformer	Language translation	168 MB	WMT 2014 English-to-German

SLO. Recall that SLO requirements refer to at least SLmin percent of requests being served within RT_{max} time. We set RT_{min} to 98% for all models, and set RT_{max} as 400 ms, 600 ms, 900 ms and 1200 ms for ResNet50, Inception-v3, SSD-ResNet50, and Transformer, respectively.

Baseline. HetSev extends the existing cluster manager (i.e., Kubernetes [9]) with an autoscaling mechanism and scheduling policy. To validate the effectiveness of HetSev's extension, we used the default Kubernetes (K8S) as the baseline for evaluation. For a fair comparison, we configured HetSev to closely resemble the autoscaling mechanism and resource management of Kubernetes, such as stabilization window of instance.

Workload. In the evaluation, we performed the request arrival process for ML workloads in two patterns: production workload and fluctuating workload. For production workload, we extracted information from the Twitter trace for a typical day out of the month [57], as there is no public production trace for an ML inference service. Furthermore, as noted in recent work [6], the trace typically represents the characteristics of real inference workloads, containing diurnal patterns and unexpected load spikes. Therefore, we synthesized the production workload to include low loads as well as a load spike, where the peak in the request rate is four times higher than the valley, referencing the results of burst demand growth common in industrial web applications. For fluctuating workload, we referred to common patterns [44] and synthesized an inference workload which indicates flat and waving rates, with a Poisson inter-arrival rate [58,59]. We generated requests by running offline clients, and simulated different request rates by adjusting the number of concurrent clients, where request rates are calculated using trace averages over a 120 s window in steps of 15 s. Then, clients submit requests sequentially (i.e., a client does not submit a request until the previous one is completed) with a configured batch size to an inference service. In summary, we utilize the production workload to evaluate how well HetSev performs compared to the baseline. Using fluctuating workload, we tested HetSev's performance under variable request rates.

6.1. HetSev with Production Workload

We first show HetSev improves throughput, cluster GPU utilization, and reduces response time compared to baseline.

Experimental setup. We used the production workload ranging from 200 to 1K QPS with a total of 6000 s of requests. We hosted an image-classification inference service using the ResNet-50 model and Inception-v3 model, which share 60% and 40% of workload, respectively. For instances co-location in HetSev, we configured multiple instances on a single GPU, whereas the baseline is one instance per GPU, as Kubernetes does not support the instance co-location HetSev introduces.

Results and discussion. Figure 5 shows that HetSev achieves up to 2.21× higher throughput, 2.56× higher cluster GPU utilization, and 2.3× lower response time compared to K8S. We observe that the curves of HetSev and K8S in terms of throughput and cluster GPU utilization are close during the low-load period (0 s–3500 s). This is because there are an adequate number of GPUs available for instances in K8S when dealing with low load, so HetSev's multi-tenant inference has no apparent advantage in improving throughput and cluster GPU utilization. In reaction to the load spike at 3500 s, HetSev adds instances immediately. Although K8S scales up instances similarly, it has lower throughput and cluster GPU utilization due to each instance holding exclusive access to a GPU and being unable to fully use GPU resources. HetSev also achieves a lower response time than K8S, thanks to HetSev enabling multiple instances to co-run on a GPU, making it possible to run more instances to spread the inference workload.

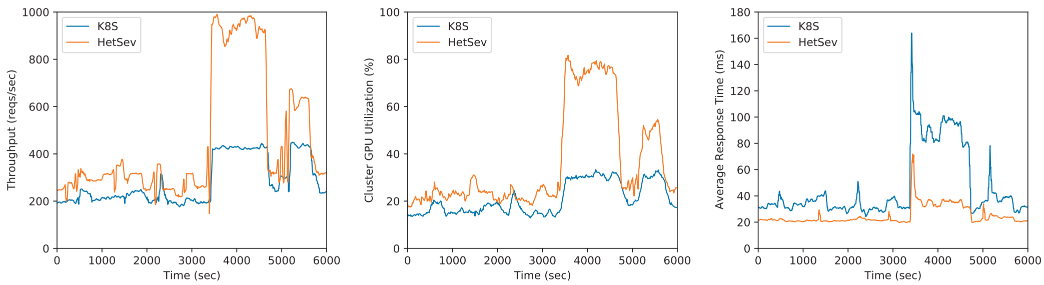

Figure 5. Throughput, cluster GPU utilization and response time under production workload.

6.2. HetSev with Fluctuating Workload

Next, we show the ability of HetSev's autoscaling mechanism to handle the fluctuating workload.

Experimental setup. To evaluate whether our prototype scaling controller can successfully scale co-located instances to accommodate fluctuating rates, we measured the performance of HetSev while submitting inference requests with the fluctuating workload for three computer vision models: ResNet-50, Inception-v3, and SSD-ResNet50. We limited the maximum number of instances for all models to 18 in this experiment.

Results and discussion. Figure 6 reports how our autoscaling mechanism performed for a 2500 s window. The top graph shows the serving throughput of each model service, the next graph number of serving instances, and the last one SLO violation for 5 s periods. Between 0 and 1000 s, the rate gradually increases and decreases to its initial rate. As the rate rises, HetSev successfully starts new instances, and spreads the workload across the instances to maintain SLOs. When the rate decreases, the number of instances decreases accordingly. The following wave rises from 1200 s to a higher peak before dropping, but HetSev also successfully adjusts the number of instances. While there are occasional SLO violations due to issues such as cold starts, note that the number of requests violated is only 1.12% of the total requests, keeping SLO violations under 4% on average.

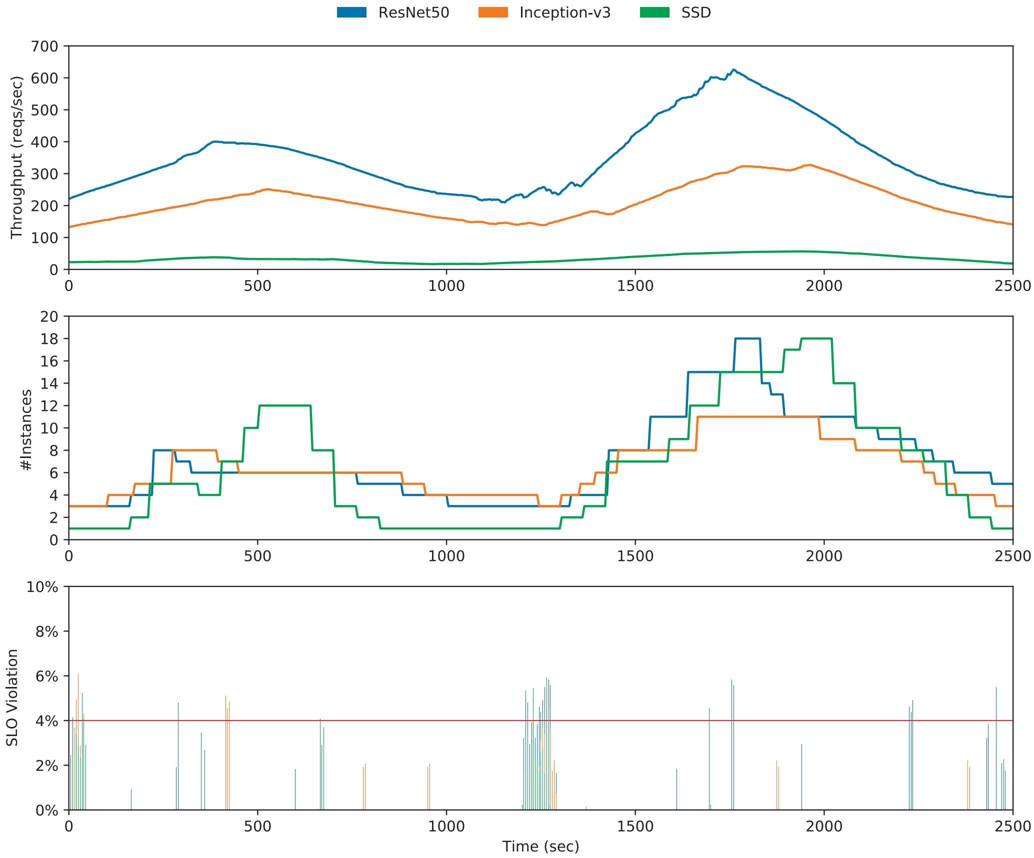

Figure 6. Throughput, number of instances and SLO violation of each model service under fluctuating workload.

6.3. Cost Effectiveness

We now show that HetSev saves resource costs compared to baseline through co-location and autoscaling mechanism.

Experimental setup. For cost-effectiveness experiments, we used all four ML models with distinct request arrivals which follow the production workload pattern, including low load and a load spike. The models were each executed for about 3000 s. We configured both HetSev and baseline always satisfies the SLOs. We measured resource cost as the product of the GPU hardware cost per unit time and the cumulative duration of the used GPU memory footprint. In this context, the hardware cost for a running instance is estimated according to AWS EC2 pricing [60], and is proportional to its memory footprint. For example, we normalize cost to 0.191 per GB/s for Nvidia Tesla V100 GPU, and 0.024 per GB/s for Nvidia T4 GPU. It is worth mentioning that only the used memory of the GPU is considered when calculating the resource cost of an instance for HetSev, while for baseline all the memory of the GPU is taken into account, even though some of the memory is not used. This is due to the fact that the default scheduler in baseline only allows one instance to be assigned to each GPU, while HetSev supports co-running instances on the GPU.

Results and discussion. Figure 7 illustrates HetSev performs much better than K8S in terms of resource cost for all four model services. Overall, HetSev yields up to 2.15× (1.79× on average) resource cost reduction, while complying with SLOs all the time. HetSev's resource cost reduction comes from two contributions: (1) exploiting instance co-location cuts the cost during the high-demand periods due to the increased computing efficiency. (2) HetSev further reduces the cost by benefiting from the scaling controller's instance-selection policy, which favors the cheapest option among all available instance types (i.e., Nvidia Tesla V100 and Nvidia T4), while the baseline randomly selects the instance type since it does not consider GPU heterogeneity.

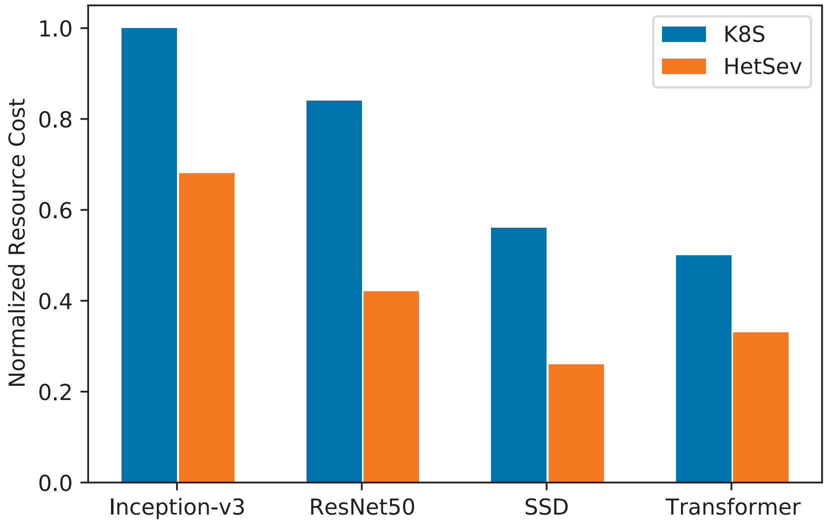

Figure 7. Total resource cost (normalized with respect to the maximum resource cost) for all four ML model services when run with default K8S and HetSev.

7. Conclusions

In this paper, we presented HetSev, a heterogeneity-aware and resource-efficient inference serving system which achieves cost effectiveness. Instead of simply minimizing the number of instances for serving inference workloads, which leads to sub-optimal cost effectiveness, our approach automatically provisions instances considering the heterogeneous GPU hardware coupled with different pricing models, and allows multiple instances to co-run on the same GPU to improve resource utilization and further cut costs. In our analysis, we showed that up to 90-percent of bi-instance execution exhibits a less than 18% latency increase among 160 ML instance co-location combinations, and that a latency increase can be further mitigated by avoiding the co-location of identical ML instances on the same GPU. HetSev is integrated into Kubernetes and can be integrated into other cluster managers. We demonstrated that HetSev can achieve up to 2.21× higher throughput, 2.56× higher cluster GPU utilization, 2.3× lower response time, and 2.15× cost reduction relative to default K8S. We also show that HetSev is capable of scaling co-located instances to accommodate fluctuating rates while keeping SLO violations under 4% on average.

To further reduce the resource cost, a promising direction of future work is to co-run even more ML instances on the same GPU. This will require more precise profiling of inter-tenant interference. To achieve this goal, we plan to use an ML-based predictive model (e.g., reinforcement learning, LSTM, etc.) to predict the potential latency increase caused by interference.

Author Contributions: All authors contributed to the study conception and design. Material preparation, data collection and analysis were performed by H.M. The first draft of the manuscript was written by H.M. Software, S.W.; Data curation, S.T.; Writing—review & editing, L.Z. and L.S. All authors commented on previous versions of the manuscript. All authors have read and agreed to the published version of the manuscript.

Funding: This work was supported in part by the National Key Research and Development Program in China (2022YFC3302103-01).

Informed Consent Statement: The participant has consented to the submission of the case report to the journal.

Data Availability Statement: All data generated or analysed during this study are included in this published article.

Conflicts of Interest: The authors have no relevant financial or non-financial interests to disclose.

References

1. Amazon Machine Learning. Available online: https://aws.amazon.com/machine-learning/ (accessed on 1 October 2022).
2. Google Cloud Prediction API Documentation. Available online: https://cloud.google.com/ai-platform/prediction/docs (accessed on 3 October 2022).
3. Microsoft Azure Machine Learning. Available online: https://azure.microsoft.com/en-us/svices/machine-learning/ (accessed on 22 September 2022).
4. Crankshaw, D.; Wang, X.; Zhou, G.; Franklin, M.J.; Gonzalez, J.E.; Stoica, I. Clipper: A {Low-Latency} Online Prediction Serving System. In Proceedings of the 14th USENIX Symposium on Networked Systems Design and Implementation (NSDI 17), Boston, MA, USA, 27–29 March 2017; pp. 613–627.
5. Gujarati, A.; Elnikety, S.; He, Y.; McKinley, K.S.; Brandenburg, B.B. 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, NV, USA, 11–15 December 2017; pp. 109–120.
6. Zhang, C.; Yu, M.; Wang, W.; Yan, F. Enabling cost-effective, slo-aware machine learning inference serving on public cloud. *IEEE Trans. Cloud Comput.* **2020**, *10*, 1765–1779. [CrossRef]
7. Shen, H.; Chen, L.; Jin, Y.; Zhao, L.; Kong, B.; Philipose, M.; Krishnamurthy, A.; Sundaram, R. Nexus: A GPU cluster engine for accelerating DNN-based video analysis. In Proceedings of the 27th ACM Symposium on Operating Systems Principles, Huntsville, ON, Canada, 27–30 October 2019; pp. 322–337.
8. Amazon SageMaker. Available online: https://aws.amazon.com/sagemaker/ (accessed on 11 October 2022).
9. Luksa, M. *Kubernetes in Action*; Simon and Schuster: New York, NY, USA, 2017.
10. Yu, F.; Bray, S.; Wang, D.; Shangguan, L.; Tang, X.; Liu, C.; Chen, X. Automated Runtime-Aware Scheduling for Multi-Tenant DNN Inference on GPU. In Proceedings of the 2021 IEEE/ACM International Conference On Computer Aided Design (ICCAD), Munich, Germany, 1–4 November 2021; pp. 1–9.
11. Dhakal, A.; Kulkarni, S.G.; Ramakrishnan, K. Gslice: Controlled spatial sharing of gpus for a scalable inference platform. In Proceedings of the 11th ACM Symposium on Cloud Computing, Virtual Event, 19–21 October 2020; pp. 492–506.
12. Choi, S.; Lee, S.; Kim, Y.; Park, J.; Kwon, Y.; Huh, J. Multi-model Machine Learning Inference Serving with GPU Spatial Partitioning. *arXiv* **2021**, arXiv:2109.01611.
13. Ghodrati, S.; Ahn, B.H.; Kim, J.K.; Kinzer, S.; Yatham, B.R.; Alla, N.; Sharma, H.; Alian, M.; Ebrahimi, E.; Kim, N.S.; et al. Planaria: Dynamic architecture fission for spatial multi-tenant acceleration of deep neural networks. In Proceedings of the 2020 53rd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO), Athens, Greece, 17–21 October 2020; pp. 681–697.
14. Choi, Y.; Rhu, M. Prema: A predictive multi-task scheduling algorithm for preemptible neural processing units. In Proceedings of the 2020 IEEE International Symposium on High Performance Computer Architecture (HPCA), San Diego, CA, USA, 22–26 February 2020; pp. 220–233.
15. Mendoza, D.; Romero, F.; Li, Q.; Yadwadkar, N.J.; Kozyrakis, C. Interference-aware scheduling for inference serving. In Proceedings of the 1st Workshop on Machine Learning and Systems, Online, 26 April 2021; pp. 80–88.
16. Wu, X.; Xu, H.; Wang, Y. Irina: Accelerating DNN Inference with Efficient Online Scheduling. In Proceedings of the 4th Asia-Pacific Workshop on Networking, Seoul, Republic of Korea, 3–4 August 2020; pp. 36–43.
17. Yu, F.; Wang, D.; Shangguan, L.; Zhang, M.; Tang, X.; Liu, C.; Chen, X. A Survey of Large-Scale Deep Learning Serving System Optimization: Challenges and Opportunities. *arXiv* **2021**, arXiv:2111.14247.
18. Gandhi, A.; Harchol-Balter, M.; Raghunathan, R.; Kozuch, M.A. Autoscale: Dynamic, robust capacity management for multi-tier data centers. *ACM Trans. Comput. Syst. (TOCS)* **2012**, *30*, 1–26. [CrossRef]
19. Olston, C.; Fiedel, N.; Gorovoy, K.; Harmsen, J.; Lao, L.; Li, F.; Rajashekhar, V.; Ramesh, S.; Soyke, J. Tensorflow-serving: Flexible, high-performance ml serving. *arXiv* **2017**, arXiv:1712.06139.
20. He, K.; Zhang, X.; Ren, S.; Sun, J. Deep residual learning for image recognition. In Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition, Las Vegas, NV, USA, 26 June–1 July 2016; pp. 770–778.
21. Szegedy, C.; Vanhoucke, V.; Ioffe, S.; Shlens, J.; Wojna, Z. Rethinking the inception architecture for computer vision. In Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition, Las Vegas, NV, USA, 26 June–1 July 2016; pp. 2818–2826.
22. Liu, W.; Anguelov, D.; Erhan, D.; Szegedy, C.; Reed, S.; Fu, C.Y.; Berg, A.C. Ssd: Single shot multibox detector. In Proceedings of the European Conference on Computer Vision, Amsterdam, The Netherlands, 11–14 October 2016; Springer: Berlin/Heidelberg, Germany, 2016; pp. 21–37.
23. Vaswani, A.; Shazeer, N.; Parmar, N.; Uszkoreit, J.; Jones, L.; Gomez, A.N.; Kaiser, Ł.; Polosukhin, I. Attention is all you need. In *Advances in Neural Information Processing Systems*; NIPS: Long Beach, CA, USA, 2017; Volume 30.

24. Bahdanau, D.; Chorowski, J.; Serdyuk, D.; Brakel, P.; Bengio, Y. End-to-end attention-based large vocabulary speech recognition. In Proceedings of the 2016 IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP), Shanghai, China, 20–25 March 2016; pp. 4945–4949.
25. SeldonIO. Seldon Core. Available online: https://github.com/SeldonIO/seldon-core (accessed on 21 October 2022).
26. AWSLABS. Multi Model Server. Available online: https://github.com/awslabs/multi-model-server (accessed on 24 October 2022).
27. Docker. Available online: https://www.docker.com/ (accessed on 14 September 2022).
28. Open Neural Network Exchange. Available online: https://github.com/onnx/onnx (accessed on 15 October 2022).
29. Romero, F.; Li, Q.; Yadwadkar, N.J.; Kozyrakis, C. {INFaaS}: Automated Model-less Inference Serving. In Proceedings of the 2021 USENIX Annual Technical Conference (USENIX ATC 21), Virtual Event, 14–16 July 2021; pp. 397–411.
30. Perri, D.; Simonetti, M.; Gervasi, O. Deploying Efficiently Modern Applications on Cloud. *Electronics* **2022**, *11*, 450. [CrossRef]
31. GOOGLE. Google Cloud Autoscaling. Available online: https://cloud.google.com/compute/docs/autoscaler (accessed on 23 October 2022).
32. AMAZON. AWS Autoscaling. Available online: https://aws.amazon.com/autoscaling/ (accessed on 27 September 2022).
33. Al-Haidari, F.; Sqalli, M.; Salah, K. Impact of cpu utilization thresholds and scaling size on autoscaling cloud resources. In Proceedings of the 2013 IEEE 5th International Conference on Cloud Computing Technology and Science, Bristol, UK, 2–5 December 2013; Volume 2, pp. 256–261.
34. Casalicchio, E. A study on performance measures for auto-scaling CPU-intensive containerized applications. *Clust. Comput.* **2019**, *22*, 995–1006. [CrossRef]
35. Zhu, J.; Yang, R.; Sun, X.; Wo, T.; Hu, C.; Peng, H.; Xiao, J.; Zomaya, A.Y.; Xu, J. QoS-aware co-scheduling for distributed long-running applications on shared clusters. *IEEE Trans. Parallel Distrib. Syst.* **2022**, *33*, 4818–4834. [CrossRef]
36. Hu, Y.; Rallapalli, S.; Ko, B.; Govindan, R. Olympian: Scheduling gpu usage in a deep neural network model serving system. In Proceedings of the 19th International Middleware Conference, Rennes, France, 10–14 December 2018; pp. 53–65.
37. Ding, Y.; Zhu, L.; Jia, Z.; Pekhimenko, G.; Han, S. Ios: Inter-operator scheduler for cnn acceleration. *Proc. Mach. Learn. Syst.* **2021**, *3*, 167–180.
38. Borowiec, D.; Yeung, G.; Friday, A.; Harper, R.H.; Garraghan, P. Trimmer: Cost-Efficient Deep Learning Auto-tuning for Cloud Datacenters. In Proceedings of the 2022 IEEE 15th International Conference on Cloud Computing (CLOUD), Barcelona, Spain, 10–16 July 2022; pp. 374–384.
39. Xiao, W.; Bhardwaj, R.; Ramjee, R.; Sivathanu, M.; Kwatra, N.; Han, Z.; Patel, P.; Peng, X.; Zhao, H.; Zhang, Q.; et al. Gandiva: Introspective cluster scheduling for deep learning. In Proceedings of the 13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18), Carlsbad, CA, USA, 8–10 October 2018; pp. 595–610.
40. Xiao, W.; Ren, S.; Li, Y.; Zhang, Y.; Hou, P.; Li, Z.; Feng, Y.; Lin, W.; Jia, Y. {AntMan}: Dynamic Scaling on {GPU} Clusters for Deep Learning. In Proceedings of the 14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20), Online, 4–6 November 2020; pp. 533–548.
41. Yeh, T.A.; Chen, H.H.; Chou, J. Kubeshare: A framework to manage gpus as first-class and shared resources in container cloud. In Proceedings of the 29th International Symposium on High-Performance Parallel and Distributed Computing, Stockholm, Sweden, 23–26 June 2020; pp. 173–184.
42. Yu, F.; Wang, D.; Shangguan, L.; Zhang, M.; Liu, C.; Chen, X. A Survey of Multi-Tenant Deep Learning Inference on GPU. *arXiv* **2022**, arXiv:2203.09040.
43. Delimitrou, C.; Kozyrakis, C. Paragon: QoS-aware scheduling for heterogeneous datacenters. *ACM SIGPLAN Not.* **2013**, *48*, 77–88. [CrossRef]
44. Delimitrou, C.; Kozyrakis, C. Quasar: Resource-efficient and qos-aware cluster management. *ACM SIGPLAN Not.* **2014**, *49*, 127–144. [CrossRef]
45. Novaković, D.; Vasić, N.; Novaković, S.; Kostić, D.; Bianchini, R. {DeepDive}: Transparently Identifying and Managing Performance Interference in Virtualized Environments. In Proceedings of the 2013 USENIX Annual Technical Conference (USENIX ATC 13), Berkeley, CA, USA, 26 June 2013; pp. 219–230.
46. Phull, R.; Li, C.H.; Rao, K.; Cadambi, H.; Chakradhar, S. Interference-driven resource management for GPU-based heterogeneous clusters. In Proceedings of the 21st International Symposium on High-Performance Parallel and Distributed Computing, Delft, The Netherlands, 18–22 June 2012; pp. 109–120.
47. Kato, S.; Lakshmanan, K.; Rajkumar, R.; Ishikawa, Y. {TimeGraph}:{GPU} Scheduling for {Real-Time}{Multi-Tasking} Environments. In Proceedings of the 2011 USENIX Annual Technical Conference (USENIX ATC 11), Portland, OR, USA, 15–17 June 2011.
48. Chen, Q.; Yang, H.; Guo, M.; Kannan, R.S.; Mars, J.; Tang, L. 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, Xi'an, China, 8–12 April 2017; pp. 17–32.
49. Jog, A.; Bolotin, E.; Guz, Z.; Parker, M.; Keckler, S.W.; Kandemir, M.T.; Das, C.R. Application-aware memory system for fair and efficient execution of concurrent gpgpu applications. In Proceedings of the Workshop on General Purpose Processing Using GPUs, Salt Lake City, UT, USA, 1 March 2014; pp. 1–8.

50. Xu, F.; Xu, J.; Chen, J.; Chen, L.; Shang, R.; Zhou, Z.; Liu, F. iGniter: Interference-Aware GPU Resource Provisioning for Predictable DNN Inference in the Cloud. *arXiv* **2022**, arXiv:2211.01713.
51. Albahar, H.; Dongare, S.; Du, Y.; Zhao, N.; Paul, A.K.; Butt, A.R. SCHEDTUNE: A Heterogeneity-Aware GPU Scheduler for Deep Learning. In Proceedings of the 2022 22nd IEEE International Symposium on Cluster, Cloud and Internet Computing (CCGrid), Taormina, Italy, 16–19 May 2022; pp. 695–705.
52. Mars, J.; Tang, L. Whare-map: Heterogeneity in "homogeneous" warehouse-scale computers. In Proceedings of the 40th Annual International Symposium on Computer Architecture, Tel-Aviv, Israel, 23–27 June 2013; pp. 619–630.
53. Wang, H.; Yang, Y.; Huang, P.; Zhang, Y.; Zhou, K.; Tao, M.; Cheng, B. S-CDA: A smart cloud disk allocation approach in cloud block storage system. In Proceedings of the 2020 57th ACM/IEEE Design Automation Conference (DAC), Virtual Event, 20–24 July 2020; pp. 1–6.
54. Istio. Available online: https://github.com/istio/istio (accessed on 19 September 2022).
55. DCGM-Exporter. NVIDIA GPU Metrics Exporter for Prometheus Leveraging DCGM. Available online: https://github.com/NVIDIA/dcgm-exporter (accessed on 13 October 2022).
56. Prometheus. Available online: https://github.com/prometheus/prometheus (accessed on 19 October 2022).
57. Twitter Streaming Traces. Available online: https://archive.org/details/archiveteam-twitter-stream-2021-03 (accessed on 22 October 2022).
58. Gupta, U.; Hsia, S.; Saraph, V.; Wang, X.; Reagen, B.; Wei, G.Y.; Lee, H.H.S.; Brooks, D.; Wu, C.J. Deeprecsys: A system for optimizing end-to-end at-scale neural recommendation inference. In Proceedings of the 2020 ACM/IEEE 47th Annual International Symposium on Computer Architecture (ISCA), Virtual Event, 30 May–3 June 2020; pp. 982–995.
59. Reddi, V.J.; Cheng, C.; Kanter, D.; Mattson, P.; Schmuelling, G.; Wu, C.J.; Anderson, B.; Breughe, M.; Charlebois, M.; Chou, W.; et al. Mlperf inference benchmark. In Proceedings of the 2020 ACM/IEEE 47th Annual International Symposium on Computer Architecture (ISCA), Virtual Event, 30 May–3 June 2020; pp. 446–459.
60. AWS EC2 Pricing. Available online: https://aws.amazon.com/ec2/pricing/on-demand/ (accessed on 28 October 2022).