Performance Characterization of Multi-Container Deployment Schemes for Online Learning Inference

Peini Liu
Barcelona Supercomputing Center
Universitat Politècnica de Catalunya
Barcelona, Spain
peini.liu@bsc.es

Jordi Guitart
Barcelona Supercomputing Center
Universitat Politècnica de Catalunya
Barcelona, Spain
jordi.guitart@bsc.es

Amir Taherkordi
University of Oslo, Norway
NTNU, Trondheim, Norway
amirhost@ifi.uio.no

***Abstract*—Online machine learning (ML) inference services provide users with an interactive way to request for predictions in real-time. To meet the notable computational requirements of such services, they are increasingly being deployed in the Cloud. In this context, the efficient provisioning and optimization of ML inference services in the Cloud is critical to achieve the required performance and meet the dynamic queries by end-users. Existing provisioning solutions focus on framework parameter tuning and infrastructure resources scaling, without considering deployments based on containerization technologies. The latter promises reproducibility and portability features for ML inferences services. There is limited knowledge about the impact of distinct deployment schemes at the container-level on the performance of online ML inference services, particularly on how to exploit multi-container deployments and its relation with processor and memory affinity. In light of this, in this paper we investigate experimentally the containerization of ML inference services and analyze the performance of multi-container deployments that partition the threads belonging to an online learning application into multiple containers in each node. This paper shares the findings and lessons learned from conducting realistic client patterns on an image classification model across numerous deployment configurations, especially including the impact of container granularity and its potential to exploit processor and memory affinity. Our results indicate that fine-grained multi-container deployments and affinity are useful for improving performance (both throughput and latency). In particular, our experiments on single-node and four-node clusters show up to 69% and 87% performance improvement compared to the single-container deployment, respectively.**

***Index Terms*—Machine Learning Inference, Kubernetes, Deployment Schemes, Multi-container, Affinity.**

I. Introduction

Machine Learning (ML) is increasingly becoming popular in various data analysis tasks such as image classification, machine translation, recommendation systems, and speech recognition [1] [2] [3] [4]. *ML inference* is an important phase that uses trained ML models to make predictions from new data. From a runtime perspective, ML inference can be conducted either as a batch process, where predictions can be generated asynchronously from a batch of samples with no specific time limit to receive the results, or more interactively, through an online ML inference service, which receives dynamic queries from end-users and serves the predictions in real-time (subject to a latency bound) [5] [6] [7] [8].

To meet the notable computational requirements of ML inference services, especially in the prediction step of the pipeline, those services are increasingly being deployed in the Cloud, which provides access to countless computational resources and allows to automatically scale the services by elastically deploying more or fewer instances to meet the changing demand. In this context, the objective of online ML inference service provisioning in the Cloud must be to find suitable deployment schemes such that inference services use the hardware efficiently and achieve the required performance (e.g., throughput) to meet the dynamic queries by end-users.

To address this challenge, existing work considers online ML inference services provision and optimizations at different layers. In the application layer, different serving frameworks used by online ML inference services support configuration settings [9] [10]. Experienced Data Engineers could tune the best parameter settings of these serving runtimes to improve the service performance [11]. In the infrastructure layer, the backends of a ML inference service can be horizontally- or vertically-scaled to use more resources [12]. These autoscaling frameworks [7] [8] [13] provide efficient ways for ML inference services to use resources while meeting Service Level Agreements (SLAs).

On top of that, current Cloud deployments are tightly coupled with containerization technology, which makes services easily reproducible and portable by encapsulating the code and dependencies. Furthermore, it isolates services so that they can be scaled or updated individually and failures do not affect the entire workload. Online ML inference services also aim to benefit from these features, to enable a seamless transition from training environments or to retrain (and redeploy) new models with the incoming new data, while meeting the performance requirements for the predictions. A typical application, in this context, is monitoring the performance of networks through analyzing the network traffic streams, which call for real-time and online learning data analytics and predictions [14].

However, there is limited knowledge about the impact of containerization on the performance of online ML inference services. In addition, there are no well-defined guidelines on proper deployment schemes in terms of exploiting the potential of containerization and constraining containers easily to a single NUMA (Non-Uniform Memory Access) domain

© 2023 IEEE. Personal use of this material is permitted. Permission from IEEE must be obtained for all other uses, in any current or future media, including reprinting/republishing this material for advertising or promotional purposes,creating new collective works, for resale or redistribution to servers or lists, or reuse of any copyrighted component of this work in other works. DOI 10.1109/CLOUD60044.2023.00046

or pinning them to specific processors. In particular, *multi-container deployments* which partition the processes that belong to each application into multiple containers in each node are worth considering. Those deployments have been demonstrated to improve the performance of some multi-process HPC throughput workloads, which consist of the execution of loosely-coupled CPU-intensive processes [15] [16]. These characteristics resemble ML inference services, as numerous serving frameworks can exploit request-level parallelism to execute independent computationally-intensive prediction queries performed by various end-users through parallel threads.

In this paper, we investigate suitable deployment schemes for allocating online ML inference services in the Cloud, focusing on container-level considerations (i.e., fine-grained multi-container deployments and CPU/memory affinity settings). Our contributions are as follows:

- We define multiple deployment schemes for online ML inference services that feature different degrees of container granularity and we set the corresponding distribution of working threads and resources to each container to serve the model.
- We enable the definition of the CPU/memory affinity for each container belonging to an online ML inference service, as part of the former deployment schemes.
- We establish an evaluation system on a Kubernetes cluster and evaluate our multi-container deployments using typical ML inference benchmarks (i.e., MLPerf) with different realistic client patterns.
- We present a systematic performance comparison, focusing on container-level considerations, to guide the Data Engineers on how to deploy their ML workloads to optimize the performance.

The paper is organized as follows: Section II discusses the related work. Section III describes the architecture of the evaluated system and shows the detailed container granularity and affinity setting schemes. The results of enabling multi-container and affinity deployments are reported in Section IV. The conclusions and future work are described in Section V.

II. Related Work

A. Enabling Online ML Inference Services

ML inference systems are complex due to the hardware and software diversity in the ML world [17]. From the hardware side, ML inference systems can utilize different hardware processors and accelerators such as CPUs, GPUs, or FPGAs. From the software side, the operating system used on each machine may vary. In addition, ML models are trained through diverse ML frameworks and libraries, thus, the runtime frameworks supporting the ML inference services are also different. Early works on ML inference systems such as Clipper [18] and Rafiki [19] deployed models in containers using custom runtime and implemented an abstract layer between clients and models to achieve model selection and request batching. Currently, there are several open-sourced runtime for online ML inference services in production and most of them support containerization, such as Tensorflow serving [10], TorchServe [9], Kserve [20], or Seldon [21]. These runtime may vary but they contain similar functionalities (e.g., model version management and model warmup) and configuration settings (e.g., parallel threading model, batching, and caching) [10].

While deploying the ML inference services in the Cloud, experienced Data Engineers could tune the parameter settings of the model serving runtime to improve the performance. For instance, [11] studied how to auto-tune the threading model for Tensorflow serving and Intel Math Kernel Library (MKL) [22] CPU backends.

The objective of our paper *is not* to test the diversity of ML model inference systems described above or tune runtime parameters to improve the performance. In our experiments, we aim to analyze the performance of multi-container deployment schemes for deploying online ML inference services on a Kubernetes cluster with multi-core machines, and we take Tensorflow serving as a representative runtime to serve ML models. Nevertheless, understanding the runtime and some of the server settings (i.e., threading model) is still relevant and complimentary to this work.

B. Deployment Schemes for Online ML Inference Services

Online communities have shared some lessons regarding how different settings can impact the performance of online ML inference services deployments. Park and Paul tested a Tensorflow Serving deployment of an image classification model across numerous deployment configurations, such as different infrastructures, trade-offs between more or fewer servers (but by using different sizes of the virtual machines), number of threads for deployments, and dynamic batching considerations [23]. Morgan et al. studied the batch size and core count scaling for the BERT-like model, as well as manually tuned multi-stream and affinities [24] [25]. These works consider different deployment and scaling options, but they do not directly assess multi-container deployments. Moreover, their evaluation does not consider realistic client scenarios.

General approaches for infrastructure-layer autoscaling of online services on the Cloud have been also proposed [12] [26]. Moreover, some works have focused specifically on the deployment and scaling of online ML inference services. MArk (Model Ark) [7], a low-latency, cost-effective inference serving system on the Cloud, used predictive scaling to mask the instance provisioning latency. PRETZEL [8] opened a black box of a model-serving application and enabled model-specific optimization with resource sharing. Nexus [27] performed detailed scheduling of GPUs for DNNs. Its design enabled several optimizations in batching and allowed more efficient resource allocation. Swayam [13] derived a global state estimate from the local state and employed a globally consistent protocol to proactively scale-out service instances for SLA compliance, and passively scale-in unused backends for resource efficiency. However, all these works mainly focus on infrastructure resource scaling to satisfy the SLAs and save costs, not considering container count scaling in a host.

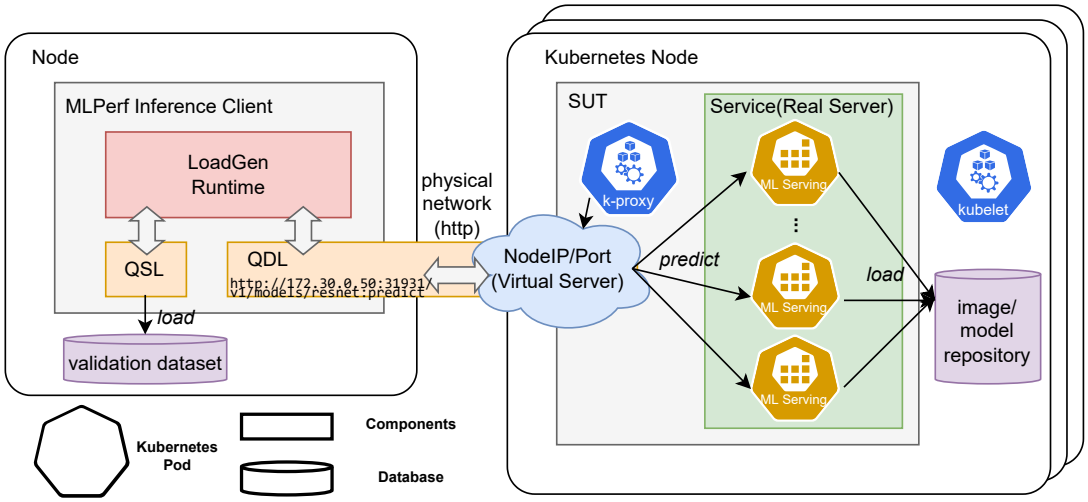

Fig. 1. Evaluation system architecture of multi-container deployment schemes for ML model inference.

The containerized deployment and affinity schemes studied in this paper provide an additional dimension of deployment configurations at the container level. Therefore, these schemes could be seen as complementary to infrastructure-layer scaling approaches and could be used together to optimize the performance of online ML inference services.

C. Multi-container Deployment Schemes

Multi-container deployments have been studied by some authors, although they have not considered online ML inference services. In particular, Medel et al. [28] conducted a performance analysis over Kubernetes considering the deployment and initialization overhead as well as understanding the performance of different pod settings. Moreover, they provided a rule to decide the number of containers per pod by considering the characteristics of the application. Liu and Guitart [15] [16] demonstrated through standalone executions that some types of containerized HPC applications achieve better performance when exploiting multi-container deployments. Their multi-container deployments partition the processes that belong to each application into multiple containers in each node, and constrain each of those containers to a single NUMA (Non-Uniform Memory Access) domain or pin them to specific processors. Moreover, their latest work [29] has enabled fine-grained scheduling in a real Cloud orchestrator (Kubernetes), showing these multi-container deployment schemes are useful when deploying containerized HPC workloads in real multi-programmed and multi-tenant Cloud environments.

Those multi-container deployments have been demonstrated to improve the performance of HPC workloads comprising loosely-coupled CPU-intensive processes, which resemble the characteristics of ML inference services. This served as inspiration to explore as well these schemes for online ML inference services.

The multi-container deployment paradigm can appropriately fulfill the need for distributed learning of relevant online ML services, for instance, monitoring the performance of networks through analyzing the network traffic streams (e.g., discovering hidden traffic stream patterns, and fault and security management). The high-speed data streaming nature of networks requires real-time (or near real-time) and online data analytics and prediction [14].

III. Evaluation Methodology

Our multi-container deployment schemes for containerized ML inference services are evaluated on a Kubernetes platform. This section describes the architecture of the evaluated system and the container granularity and affinity settings.

A. Evaluation System

The complete picture of the architecture of this system is depicted in Fig. 1.

MLPerf Inference Client (LoadGen): MLPerf Inference is a benchmark suite for measuring how fast systems can run models in a variety of deployment scenarios [17]. LoadGen is the MLPerf client, which generates traffic for scenarios as formulated by a diverse set of experts, and efficiently and fairly measures the performance of ML inference systems. LoadGen is not dataset or model aware, thereby we had to implement custom versions of the Query Sample Library (QSL) and the Query Dispatch Library (QDL) tailored to the datasets/models used in the paper. QSL is responsible for loading the data and includes untimed preprocessing. QDL is used to dispatch queries to the System Under Test (SUT) over a physical network, receive the responses, and pass them back to LoadGen.

System Under Test (SUT): The System Under Test refers to the ML inference system which provides an online ML inference service through several real server backends receiving queries from the client. In our experiments, SUT is established on a multi-core Kubernetes cluster, and models are served by Tensorflow Serving instances running inside multiple containers, each one wrapped as a Kubernetes Pod. Kubelet and Kube-

proxy components from Kubernetes generate Pods on each node and distribute the queries among those Pods, respectively.

We consider various deployment options for the SUT depending mainly on two factors. First, the **container granularity** of the online ML inference service. In this paper, we assess the impact of deploying an online ML model inference service with different numbers of containers per host, that is, different multi-container deployment scenarios. Second, the **resource affinity** of the containers running the online ML inference service. In this paper, we assess the impact of different CPU/memory affinity settings for each container.

B. Granularity Settings

Granularity settings define how we partition the online ML inference service into multiple containers (i.e., increasing the number of containers but decreasing the threads and resources on each container). A given SUT can have a single or multiple servers (each deployed within a container and with its own inference model), but the number of working threads and resources for the SUT are kept constant. We assume each SUT requires a number of CPU cores S_{cpu} and some amount of memory S_{mem} in GB. Tensorflow Serving running within the SUT contains multiple working threads inside the server, namely `tensorflow_inter_op_parallelism`, `tensorflow_intra_op_parallelism`, and `rest_api_num_threads`. For each SUT, we define these numbers of threads as N_{inter}, N_{intra}, and N_{rest}.

(I) Multi-container deployments: Each SUT runs on a set of containers $CTN = \{ctn_i | i = 1, ..., N_{ctn}\}$ which use resources from a set of hosts $HOST = \{host_h | h = 1, ..., N_h\}$. Each container i has resources requirements $R^i_{\frac{S_{cpu}}{N_{ctn}}, \frac{S_{mem}}{N_{ctn}}}$ and a threading model $T^i_{\frac{N_{inter}}{N_{ctn}}, \frac{N_{intra}}{N_{ctn}}, \frac{N_{rest}}{N_{ctn}}}$, so that the total number of working threads and resources for the SUT are kept constant. Therefore, a multi-container deployment can be expressed as a set of containers each containing a subset of the threads and requiring a share of the resources.

$$SUT_{N_{cph}} = \bigcup_{i=1}^{N_{ctn}} ctn_i \rightarrow \begin{cases} R^i_{\frac{S_{cpu}}{N_{ctn}}, \frac{S_{mem}}{N_{ctn}}} \\ T^i_{\frac{N_{inter}}{N_{ctn}}, \frac{N_{intra}}{N_{ctn}}, \frac{N_{rest}}{N_{ctn}}} \end{cases} \tag{1}$$

where N_{cph} refers to the number of containers per host and is calculated as N_{ctn}/N_h.

(II) Baseline: This is the default strategy to deploy a SUT running Tensorflow Serving on Kubernetes. The baseline is deployed as a single-container-per-host deployment, thus, it has $N_{cph} = \frac{N_{ctn}}{N_h} = 1$. The resources requirements for each container are calculated in the same way as with multi-container deployments. However, the threading model of Tensorflow Serving is decided by default: the threading pool size will be set to the number of visible cores within each server.

$$SUT_{baseline} = \bigcup_{i=1}^{N_{ctn}} ctn_i \rightarrow \begin{cases} R^i_{\frac{S_{cpu}}{N_{ctn}}, \frac{S_{mem}}{N_{ctn}}} \\ T^i_{default} \end{cases} \quad s.t.\ N_{cph} = 1 \tag{2}$$

C. Affinity Settings

Affinity settings define the exact resources from the hardware perspective that the containers of the online ML inference service will use. The affinity settings for our multi-container deployment scenarios are called *ANY* and *CPUMEM*. We assume a number of hosts N_h, and a number of containers N_{ctn}. The number of containers per host (i.e., N_{cph}) is calculated as $N_{cph} = \frac{N_{ctn}}{N_h}$. The hardware platform provides a number of CPU cores and MEM nodes from one or several sockets $S = \{socket_s | s = 0, ..., N_{socket} - 1\}$, where each socket has P cores. Hence, for a set of containers $CTN = \{ctn_i | i = 1, ..., N_{ctn}\}$ which run on a set of hosts $HOST = \{host_h | h = 1, ..., N_h\}$, each affinity setting defines a mapping $Map_{h,i} \rightarrow CPU_{h,s,[x,y]} + MEM_{h,s}$ where h, s, and $[x, y] = \{n \in \mathbb{Z} | x \leq n \leq y\}$ denote the assigned host, socket, and set of cores, respectively. In particular, affinity settings *ANY* and *CPUMEM* are defined as follows:

(I) *ANY*: Containers do not have any processor or memory affinity and all of them could access all the resources provided to this service. The actual distribution of the resources is decided by the operating system. Thus, the mapping of ANY scenarios could be expressed as:

$$Map_{h,i} \rightarrow \begin{cases} \bigcup_{s=0}^{N_{socket}-1} CPU_{h,s,[s\times P, s\times P + \frac{N_{cpu}\times N_{cph}}{N_{socket}} - 1]} \\ \bigcup_{s=0}^{N_{socket}-1} MEM_{h,s} \end{cases} \tag{3}$$

(II) *CPUMEM*: We define a specific processor and memory affinity for each container to a set of cores belonging to a single socket and to the corresponding local memory node. The mapping of *CPUMEM* scenarios could be calculated as follows, provided that the number of cores requested by each container is lower than the cores each socket provides.

$$Map_{h,i} \rightarrow \begin{cases} CPU_{h,s_i,[x_i,y_i]} \\ MEM_{h,s_i} \end{cases} \tag{4}$$

$$s_i = \lceil \frac{i}{N_{cps}} \rceil - 1 \tag{5}$$

$$x_i = s_i \times P + N_{cpu} \times ((i-1) - s_i \times N_{cps}) \tag{6}$$

$$y_i = s_i \times P + N_{cpu} \times (i - s_i \times N_{cps}) - 1 \tag{7}$$

where N_{cps} refers to the number of containers per socket and is calculated as N_{cph}/N_{socket}.

IV. Evaluation

In this section, we present an empirical performance evaluation of multi-container deployments of ML inference services on Kubernetes clusters. In this evaluation, we consider several schemes where we increase the number of containers serving the model but decrease the number of parallel working threads of the model per container. In addition, we consider different affinity settings and several real-world client scenarios.

A. Experimental Setup

Hardware: Our experiments are executed on a five-node K8s cluster. Each host consists of 2 x Intel 2697v4 CPUs (18 cores each, hyperthreading disabled, CPU frequency scaling governor is set to max performance (i.e., scaling_governor=performance)), 256 GB RAM, 60 TB GPFS file system, and 1-Gigabit Ethernet network.

Software: For all the hosts, we use CentOS release 7.7.1908 with host kernel 3.10.0-1062.el7.x86_64. The Kubernetes platform uses Kubernetes v1.19.16 (with Docker 19.03.11, Etcd 3.4.9, Flannel 0.15.0, CNI 0.8.6, and CoreDNS 1.7.0). We use Tensorflow Serving v2.8.2 as the backend server and MLPerf Inference Client v0.7 (LoadGen) to emulate the clients.

Kubernetes Cluster Settings: Our Kubernetes cluster comprises five nodes. For each node, we reserve 4 cores for system and Kubernetes components, thus, 32 cores (16 from each socket) can be used for the allocation of ML inference services.

By default, K8s Kubelet sets the CPU manager policy as 'none', so all the containers can use all the allocatable CPU resources within the resident node. For those experiments that require enabling CPU/memory affinity for containers, we configure Kubelet using `--cpu-manager-policy=static`, which starts the containers on dedicated CPUs, and `--topology-manager-policy=best-effort`, which stores the preferred NUMA affinity for the containers attempting to align resources optimally on NUMA nodes if possible.

Kube-proxy is set to the IPVS (IP Virtual Server) mode, which can direct requests for TCP- and UDP-based services to the real servers and make real services appear as virtual services on a single IP address. The IPVS load balancing algorithm is kept as the default round-robin algorithm.

Tensorflow Serving Granularity Settings: Table I shows the different container granularity scenarios considered to deploy the online ML inference service, and the corresponding resources and thread pool size settings of each container.

$SUT_{baseline}$ is the baseline scenario which represents the basic deployment scheme of a Tensorflow Serving service. It normally contains one container per host and the container uses all the resources of the host. Each container also chooses its own thread settings, by default Tensorflow Serving will set the number of inter, intra, and rest threads as the number of visible cores within the container. In our case, even though the container can only use 32 CPUs (maximum available CPUs within one host), the threads will be set to 36 because the container can see all the cores in the host.

$SUT_{N_{cph}}$ refers to the various multi-container deployments of the Tensorflow serving service. For different granularity scenarios, we select a different number of containers to deploy the ML inference service, while partitioning the number of working threads and resources for each container. Thus, the total number of resources and threads for the inference service are kept constant in all the scenarios.

Affinity Settings: We consider two affinity settings: *ANY* and *CPUMEM*. The former means that all the containers can run on any CPUs and any memory node within hosts. The latter means that the containers will use dedicated CPUs and be bound to a specific memory node. These affinity settings are configured by an agent running on each node. For *ANY*, the agent will change all the containers' CPUSETs to a range of CPUs within a host, being the number of CPUs in the range equal to the number of requested CPUs per host. For *CPUMEM*, the CPU Manager and the Topology Manager in the Kubelet are set with the appropriate modes as discussed before. In addition, the agent will check the range of CPUs allocated to each container and set the corresponding memory node for this container after its deployment.

TABLE I
SERVER SCENARIOS SETTINGS.

Scenarios ($SUT_{N_{cph}}$)	# of CTNs (N_{ctn})	Resources/CTN (R^i)	Threads/CTN (T^i)
$SUT_{baseline}$	$1 * N_h$	CPU=32cores MEM=128GiB	inter=36 intra=36 rest=36
SUT_1	$1 * N_h$	CPU=32cores MEM=128GiB	inter=32 intra=32 rest=64
SUT_2	$2 * N_h$	CPU=16cores MEM=64GiB	inter=16 intra=16 rest=32
SUT_4	$4 * N_h$	CPU=8cores MEM=32GiB	inter=8 intra=8 rest=16
SUT_8	$8 * N_h$	CPU=4cores MEM=16GiB	inter=4 intra=4 rest=8
SUT_{16}	$16 * N_h$	CPU=2cores MEM=8GiB	inter=2 intra=2 rest=4
SUT_{32}	$32 * N_h$	CPU=1core MEM=4GiB	inter=1 intra=1 rest=2

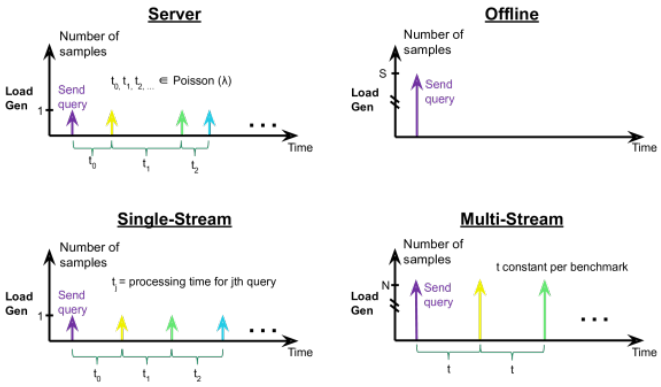

Fig. 2. The timing and number of queries from LoadGen [17]

MLPerf Inference Benchmark: As mentioned in Section III, our evaluation uses the MLPerf Inference Benchmark, which is a suite specifically designed to measure the performance of ML models during inference. It includes standard models, datasets, and evaluation metrics of different client scenarios, which enables fair and comparable measurements.

i) Model and Dataset: MLPerf provides computer vision applications with its associated reference model (i.e., a classifier network takes an image and selects the class that best

TABLE II
CLIENT SCENARIOS SETTINGS.

Scenarios	Query Generation	Metric	Sample/Query	Parameters
Single-Stream (SS)	Sequential	90th-percentile Latency	1	min_query_count=1664
Multi-Stream (MS)	Arrival Interval With Dropping	Number of Streams Subject to Latency Bound	N	min_query_count=2000 target_qps=32 max_async_queries=256 target_latency=8s
Server (S)	Poisson Distribution	Queries per Second Subject to Latency Bound	1	min_query_count=12800 target_qps=200 target_latency=20s
Offline (O)	Batch	Throughput	≥ 24576	min_query_count=32768 target_qps=200 max_batchsize=1,2,4,8

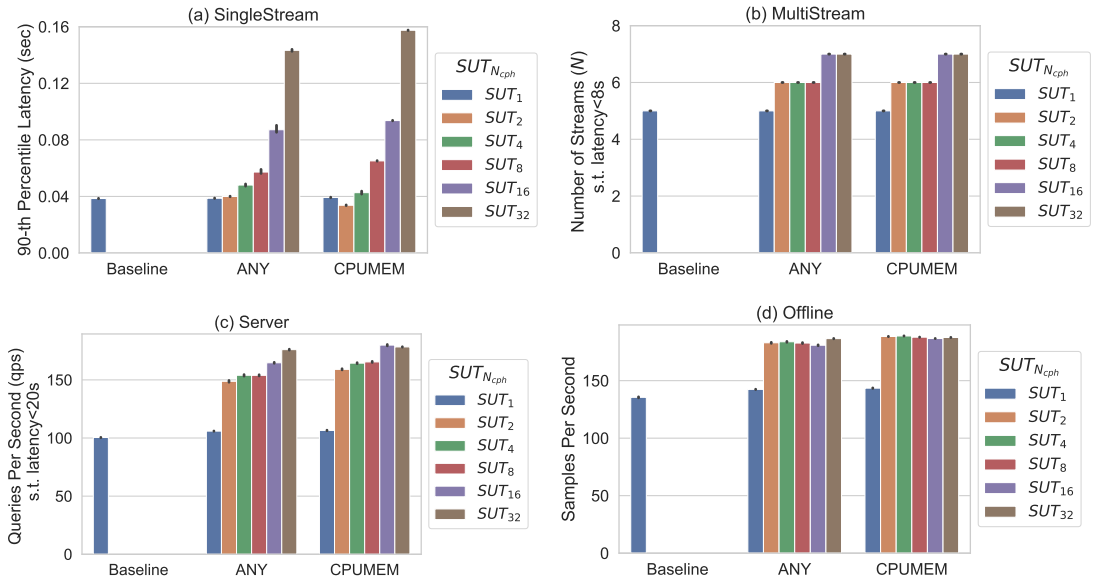

Fig. 3. Impact of container granularity and affinity in SUT performance on different client scenarios on a Kubernetes cluster.

describes it). In particular, for image classification, it provides a well-known vision model: the computationally-intensive Resnet50 [1], which accepts base64-encoded JPEG images as input and decodes them within the inference stage. We use the ImageNet 2012 dataset, crop the images to 224x224 in preprocessing, and send the strings of base64-encoded images through the physical network using REST APIs.

ii) Client Scenarios: MLPerf LoadGen provides four realistic end-user scenarios, namely Single-Stream (SS), Multi-Stream (MS), Server (S), and Offline (O), which represent many critical inference applications. Fig. 2 shows how LoadGen generates queries for each scenario and Table II summarizes their settings in our experiments, which we describe briefly below. Additional details can be found in [17].

- SS: The *Single-Stream* scenario represents the client sending inference-query streams one by one (i.e., the client waits for the completion of one query before issuing another) with a query sample size of 1. The objective is to assess the responsiveness of the SUT by means of the 90th percentile latency.
- MS: The *Multi-Stream* scenario represents the client sending inference-query streams with a fixed time interval. It assesses the maximum query sample size of each inference-query stream subject to a latency bound. No more than 1% of queries may exceed the latency bound.
- S: The *Server* scenario represents an application where the one sample-sized inference-query streams are arriving randomly at the SUT with a Poisson distribution. The SUT responds to each query within a benchmark-specific latency bound. No more than 1% of queries may exceed the latency bound. The performance metric is the Poisson parameter that indicates the queries-per-second (QPS) achievable while meeting the latency QoS requirement.
- O: The *Offline* scenario represents batch-processing applications where all the data are sent to the SUT as soon as possible and latency is unconstrained. The performance metric is the throughput measured in samples per second.

B. Experiment 1: Multi-container deployment and affinity evaluation on a single host

Fig. 3 shows the impact of container granularity and affinity in SUT performance on different client scenarios on a Ku-

bernetes cluster with a single node. The results are derived from 10 executions. Additionally, for some scenarios, we also analyze the inference time and the issue delay time for each individual sample.

1) SingleStream: This scenario generates low load because the client sends queries one by one, thus, every time only one query is being processed at one of the containers of the SUT. From SUT_1 to SUT_{32}, that is, when deploying more containers per host (i.e., from 1 to 32), each one has lower allocated resources (i.e., from 32 CPUs/128 GiB to 1 CPU/4 GiB) and working threads (i.e., from 32 to 1). *SingleStream* does not fully show the benefits of using multiple containers to deploy the online ML inference service, because always only one backend is used at a time, that is, we can only exploit parallelism within a request, not among requests.

Fig. 3 *(a) SingleStream* shows the 90th percentile latency of different SUT deployments. For *ANY* scenario, running more containers per host increases the 90th latency (i.e., SUT_2–SUT_{32} increase by 4%–25%–46%–122%–271% with respect to SUT_1). This increment is caused by the lower amount of resources and threads for each container as we increase the number of containers. On the other side, SUT_2 with *CPUMEM* settings shows 13% 90th latency improvement regarding SUT_1, because running two containers, one in each socket, improves the cache usage and avoids remote memory accesses between two NUMA nodes. However, SUT_4–SUT_{32} still show 7%–67%–140%–303% degradation regarding SUT_1 because the better locality cannot compensate for the lower parallelism as we increase the number of containers (due to the reduction of resources and threads per container).

Similarly, when comparing *CPUMEM* and *ANY* settings, the former shows better performance in coarse-grained scenarios $SUT_2 - SUT_4$ because each container has enough resources and threads to exploit the parallelism of the NUMA node to which they are assigned while getting the corresponding locality benefits. However, *CPUMEM* shows worse performance in finer-grained scenarios SUT_8–SUT_{32} because each container has less resources and threads but, still, *CPUMEM* allocates them in dedicated CPUs from two NUMA nodes. Contrariwise, *ANY* settings allow the containers to be allocated in the entire range of available CPUs, and due to the lower amount of resources each container needs, the scheduler is able to consolidate all of them in a single NUMA node.

2) MultiStream, Server, and Offline: The impact of container granularity and affinity in SUT performance on *MultiStream* (MS), *Server* (S), and *Offline* (O) client scenarios is shown in Fig. 3 *(b) MultiStream*, which displays the maximum number of streams (subject to 99th latency $<$ 8s), Fig. 3 *(c) Server*, which displays the Queries per Second (qps) (subject to 99th latency $<$ 20s), and Fig. 3 *(d) Offline*, which displays the samples per second (the max_batch_size is set to 8 to optimize the performance in this scenario). For comparison purposes, we also display detailed qps of *MultiStream* scenario in Fig. 4. The three client scenarios show different patterns to send queries, but all of them generate a high load to the SUT, which consumes high computation resources, and allows to evaluate the impact of multi-container deployments.

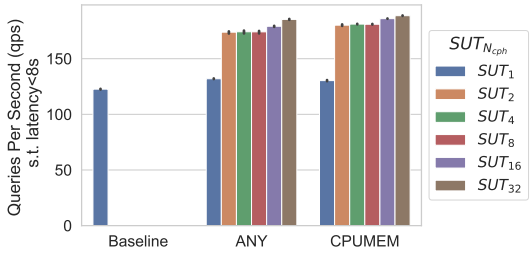

Fig. 4. Queries Per Second in *MultiStream* ANY and CPUMEM scenarios.

Baseline: SUT_1 *ANY* and *CPUMEM* have roughly the same performance improvement up to 8%, 6%, and 6% compared to the baseline in client scenarios MS (see Fig. 4), S (see Fig. 3(c)), and O (see Fig. 3(d)), respectively. This is because a single container in the baseline starts on all the CPUs within the host and Tensorflow Serving creates as many threads as visible CPUs (i.e., 36) within this container, whereas there are effectively only 32 CPUs available for ML inference in the host (we reserve 4 cores for Kubernetes and system). Thus, the baseline has more CPU migrations and context switches among more threads than SUT_1 with *ANY* or *CPUMEM*, which start a single container on 32 cores and threads.

Granularity: Regarding the container granularity, in MS (see Fig. 4), SUT_2–SUT_{32} have 32%–32%–32%–36%–40%, and 38%–39%–38%–42%–45% performance improvement regarding SUT_1 with *ANY* and *CPUMEM*, respectively; in S (see Fig. 3(c)), SUT_2–SUT_{32} have 40%–45%–45%–56%–66% and 49%–55%–55%–69%–67% performance improvement regarding SUT_1 with *ANY* and *CPUMEM*, respectively; in O (see Fig. 3(d)), SUT_2–SUT_{32} have 28%–29%–28%–27%–31%, 31%–32%–31%–30%–31% performance improvement regarding SUT_1 with *ANY* and *CPUMEM*, respectively. All the client scenarios show better performance with multi-container deployments. The difference is greater as we increase the number of containers for *MultiStream* and, especially, for *Server* scenarios. As shown in Fig. 5, which displays the mean latency in *Server* scenarios, multi-container deployments show up to 90% latency improvement with respect to SUT_1, and finer-grained containers (from SUT_2 to SUT_{32}) show increasingly better performance.

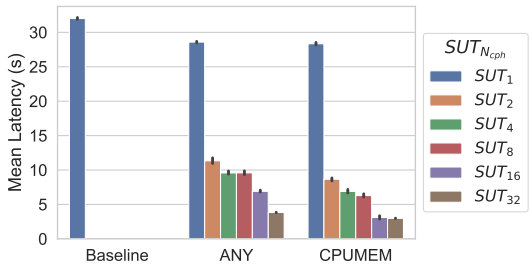

Fig. 5. Mean latency in *Server* ANY and CPUMEM scenarios.

In *Server* scenarios, the overall latency of a sample can

be broken down into the sample waiting time before the sample is processed (i.e., issue delay time) and the actual sample inference time. Fig. 6 and Fig. 7 show the inference time of individual samples in this scenario with *ANY* and *CPUMEM* settings, respectively, finer-grained multi-container deployments can use better the resources, and thus process the samples quicker, reducing their inference time (and consequently, their latency). Note how, in any case, the inference time is kept below the allowed latency bound (i.e., 20 s). In the same manner, finer-grained multi-container deployments also reduce the waiting time of the samples, as shown in Fig. 8 and Fig. 9. They display the issue delay time of individual samples with *ANY* and *CPUMEM*, respectively. In particular, the plots show that the saturation point, i.e., when the samples start to wait resulting in some delay time, appears later for finer-grained multi-container deployments.

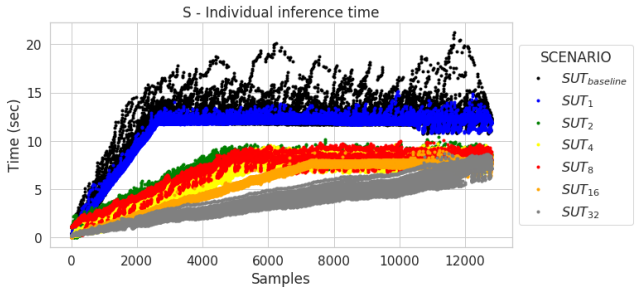

Fig. 6. Inference time of individual samples in *Server-ANY* scenario.

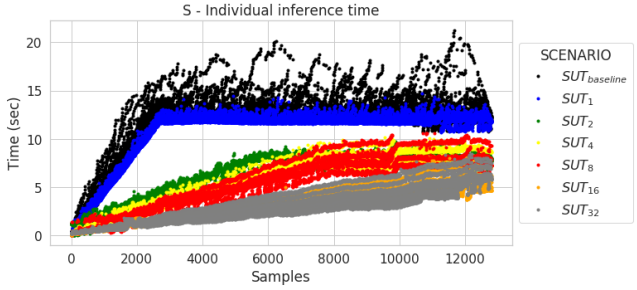

Fig. 7. Inference time of individual samples in *Server-CPUMEM* scenario.

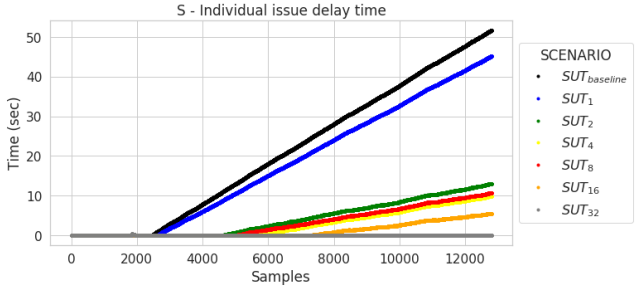

Fig. 8. Issue delay time of individual samples in *Server-ANY* scenario.

The better performance of the multi-container deployment schemes is a consequence of their ability to optimize the

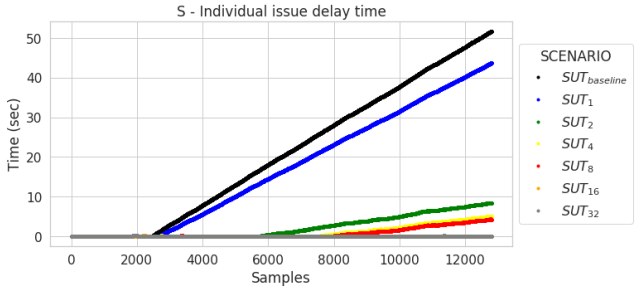

Fig. 9. Issue delay time of individual samples in *Server-CPUMEM* scenario.

scheduling of the serving threads onto the available resources (processors and memory nodes), mainly by favouring processor affinity, which reduces context switches and migrations and memory affinity exploits data locality, thus improving cache usage and reducing remote memory accesses in NUMA systems. With *CPUMEM* settings, the affinity for each container is enforced explicitly by the deployment scheme, which allocates dedicated CPUs to each of them. With finer-grain deployments, each container is allocated with fewer CPUs (and from a single NUMA node), thus, there are fewer chances for the serving threads to migrate. With *ANY* settings, the affinity for each container is not enforced explicitly, but indirectly encouraged through the scheduling of cgroups done by the Linux Completely Fair Scheduler (CFS). The processes within each container are grouped together in a cgroup, so they will be viewed by the scheduler as a single unit. CFS applies the principle of sharing the resources fairly among these cgroups at the same level of the hierarchy, which means it will first divide CPU time equally between all entities at the same level, and then proceed by doing the same in the next level [30]. In multi-container deployment scenarios, first, the CPUs are evenly distributed across cgroups. Then, the threads on each cgroup are scheduled on those CPUs. As a higher number of containers contain a lower number of threads, this scheduling within the group is simpler, allowing to exploit processor affinity better. Notably, in SUT_{32}, the sole thread in each container runs on a single CPU, akin to being pinned to it.

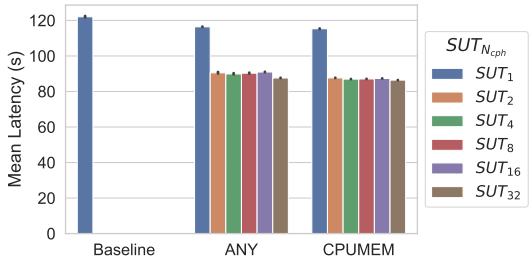

Fig. 10. Mean latency in *Offline* ANY and CPUMEM scenarios.

ANY/CPUMEM affinity: Regarding the affinity settings, SUT_1 behaves similarly with both *ANY* and *CPUMEM* settings because the single container deployed in both cases uses the same range of CPUs and memory from the two

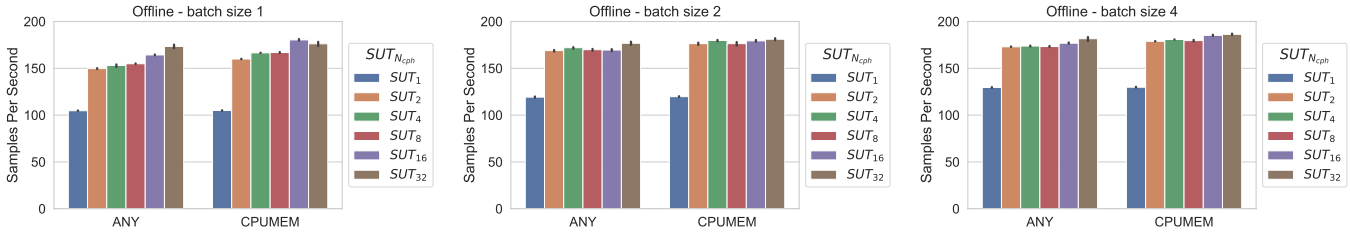

Fig. 11. Impact of container granularity and affinity in the *Offline* scenario with different client batch size.

sockets. SUT_2–SUT_{32} with *CPUMEM* settings show better performance than *ANY* up to 4%, 9%, and 3% in scenarios MS (see Fig 4), S (see Fig 3(c)), and O (see Fig 3(d)), respectively. This improvement is also observed when considering the mean latency. As shown in Fig. 5, SUT_2–SUT_{32} with *CPUMEM* settings in *Server* scenario have 23%–29%–35%–58%–23% mean latency improvements, respectively, with respect to *ANY*. Similarly, SUT_2–SUT_{32} with *CPUMEM* settings in *Offline* scenario also show 3%–3%–3%–4%–1% improvements on the mean latency regarding *ANY*, as shown in Fig. 10.

CPUMEM has better performance than *ANY* because it enforces CPU affinity, which restricts the number of assigned CPUs within each container. Hence, the threads running in finer-grained containers have fewer available CPUs where they could migrate. More importantly, memory affinity improves the cache utilization and prevents the remote memory accesses as well, thus reducing the memory latency. Note that the improvement of SUT_{32} with *CPUMEM* regarding SUT_{32} with *ANY* is less noticeable than in the rest of SUT_2–SUT_{16} scenarios because in SUT_{32} each container runs only on one CPU and containers are already well-distributed among cores, thus, the benefit of *CPUMEM* settings on SUT_{32} only comes from the memory access.

C. Experiment 2: Multi-container deployment evaluation with different client batch size on a single host

Batching calls to a remote service is a well-known technique to increase the performance. There are fixed processing costs for any interaction with a remote service, such as serialization, network transfer, and deserialization. Packaging many samples into a single batch minimizes the cost per sample.

Fig. 11 shows the impact of container granularity and affinity in the *Offline* scenario with various client batch sizes, namely 1, 2, and 4. By comparing this figure with Fig. 3(d), which set the batch size as 8, the overall performance is increased with larger batch sizes. For instance, the throughput of SUT_1 increases up to 26% from batch size 1 to 8.

Regarding the impact of container granularity, Fig. 11 shows that multi-container deployment schemes outperform the single container deployment for all the batch sizes. In particular, for batch size 1, SUT_2–SUT_{32} have 43%–47%–48%–57%–67% and 53%–59%–59%–72%–70% performance improvement regarding SUT_1 with *ANY* and *CPUMEM* settings; for batch size 2, SUT_2–SUT_{32} have 42%–45%–43%–43%–49% and 48%–51%–49%–50%–52% improvement regarding SUT_1 with *ANY* and *CPUMEM*; and for batch size 4, SUT_2–SUT_{32} have 33%–34%–33%–36%–41% and 38%–39%–39%–43%–44% improvement regarding SUT_1 with *ANY* and *CPUMEM*. Interestingly, smaller batch sizes can benefit more from multi-container deployments. As for affinity, *CPUMEM* also outperforms *ANY* for all the batch sizes, providing, again, a higher benefit for smaller ones. Notably, the throughput increases up to 10%–7%–6%–4% for batch sizes 1, 2, 4, and 8, respectively.

D. Experiment 3: Multi-container deployment and affinity evaluation on a four-node cluster

Experiments in the previous sections were run in a single node. Nevertheless, we anticipate that most of the performance insights obtained in those sections would still hold for multi-container deployment schemes in a larger cluster.

Fig. 12 (top) shows the impact of container granularity and affinity in the *Offline* scenario on a four-node cluster. SUT_2–SUT_{32} have 13%–35%–49%–55%–69% and 15%–24%–27%–27%–36% throughput and mean latency improvement, respectively, regarding SUT_1 with *ANY* settings. SUT_2–SUT_{32} have 87%–86%–84%–80%–78% and 32% (for all the SUT_i) throughput and mean latency improvement, respectively, regarding SUT_1 with *CPUMEM*. Latency improvements with *ANY* and *CPUMEM* are comparable, but throughput improvements are considerably higher with *CPUMEM* affinity, as it shows up to 68% improvement with respect to *ANY*. As anticipated, the performance observations and conclusions described in Experiment 1 also apply here.

Fig. 12 (bottom) shows the 99th and 99.9th tail latencies. For *ANY*, finer-grained containers have better tail latency. In particular, 99th latency of SUT_2–SUT_{32} improves 13%–28%–35%–37%–44% regarding SUT_1 and 99.9th latency of SUT_2–SUT_{32} improves 12%–26%–33%–36%–42% regarding SUT_1. Fig. 13 shows the individual inference time of each sample at *ANY*, where the last samples up to 3% show a tail latency increase in all the scenarios. This is because the tail is less CPU-intensive and all the containers are about to finish their tasks. When tasks in one container (i.e., one cgroup) become idle and are not using any CPU time, the leftover time is collected in a global pool of CPU cycles that can be used by other containers (i.e., other cgroups) from this pool. Finer-grained deployments show a better tail latency because they have more cgroups, thus, each container releases fewer CPU cycles when it finishes, causing fewer CPU migrations for the rest of running containers.

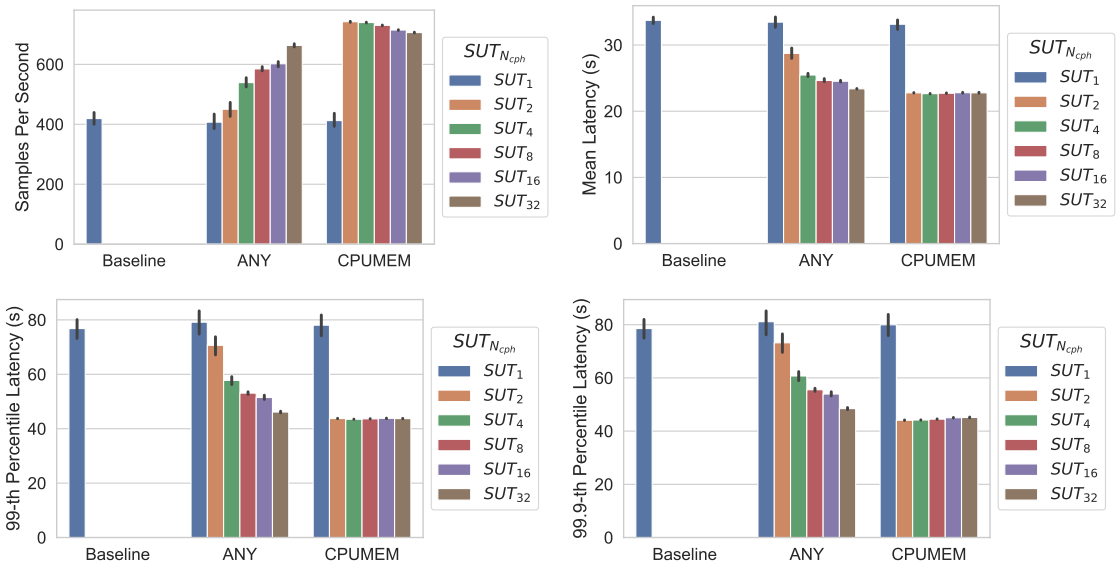

Fig. 12. Impact of container granularity and affinity in the *Offline* scenario at scale.

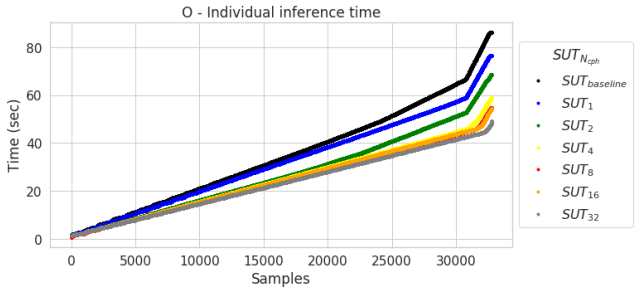

Fig. 13. Inference time of individual samples in *Offline-ANY* scenario.

For *CPUMEM*, multi-container deployments show up to 47% and 46% improvement on 99th and 99.9th latency, respectively, regarding SUT_1, but there is a minor difference among the various multi-container schemes on 99th latency (less than 1%) and 99.9th latency (2% difference). Multi-container deployments in *CPUMEM* show almost no overhead in the tail latency because each container has its own cgroup without the CPUs overlap.

V. Conclusion and Future Work

This paper presented multi-container deployment schemes for containerized online ML inference services on Kubernetes. We focused on the container layer to understand how the container granularity and its combination with CPU/memory affinity impact the performance of online ML inference services. We concluded that multi-container deployments show significant performance improvements up to 69% and 87% regarding the single-container deployment on single-node and four-node clusters, respectively. Finer-grained deployments show better performance because they favour process affinity in a similar way to when threads are pinned explicitly. Consequently, these deployments fit very well with explicit CPU/memory affinity settings for each container. As demonstrated in our experiments, those settings can sum up to 9% and 68% to the granularity gains on single-node and four-node clusters, respectively. The benefit of multi-container deployment schemes with affinity also shows up with different client batch sizes and in larger clusters.

All in all, we demonstrated that it is worth considering (and optimizing) the containerization dimension when provisioning ML inference services to benefit not only from its encapsulation, security, and fault isolation, but also gain performance. Moreover, the granularity/affinity settings at the container-level are complimentary to other optimizations such as batching and autoscaling and, therefore, can be combined to derive better deployment and scheduling policies for ML inference services. In addition, we anticipate that the performance insights about our proposed schemes are platform-independent, hence, other containerization/orchestration platforms can also take advantage of multi-container deployment schemes and CPU/memory affinity to improve the performance of ML inference services.

In the future, we plan to further expand our research by evaluating other diverse ML models and conducting experiments on GPUs. We will also consider the performance insights in this paper about the container-level settings (i.e., container granularity and affinity) to derive placement policies integrated within the Kubernetes scheduler/Kubelet agent for the efficient deployment of online multi-model ML inference services in a multi-programmed and multi-tenant Cloud environment.

Acknowledgment

We thank Lenovo for providing the infrastructure to run the experiments in this paper. This work was partially supported by Lenovo as part of Lenovo-BSC collaboration agreement, by the Spanish Government under contract PID2019-107255GB-C22, and by the Generalitat de Catalunya under contract 2021-SGR-00478 and under grant 2020 FI-B 00257.

REFERENCES

[1] K. He, X. Zhang, S. Ren, and J. Sun, "Deep residual learning for image recognition," 2015. [Online]. Available: https://arxiv.org/abs/1512.03385

[2] A. Krizhevsky, I. Sutskever, and G. E. Hinton, "Imagenet classification with deep convolutional neural networks," *Commun. ACM*, vol. 60, no. 6, pp. 84–90, may 2017.

[3] T. Luong, H. Pham, and C. D. Manning, "Effective approaches to attention-based neural machine translation," in *2015 Conference on Empirical Methods in Natural Language Processing*. Lisbon, Portugal: Association for Computational Linguistics, Sep. 2015, pp. 1412–1421.

[4] P. Covington, J. Adams, and E. Sargin, "Deep neural networks for youtube recommendations," in *10th ACM Conference on Recommender Systems*, ser. RecSys'16. ACM, 2016, pp. 191–198.

[5] P. Liu, G. Bravo-Rocca, J. Guitart, A. Dholakia, D. Ellison, and M. Hodak, "Scanflow: An end-to-end agent-based autonomic ML workflow manager for clusters," in *22nd International Middleware Conference: Demos and Posters*, ser. Middleware'21. ACM, 2021, pp. 1–2.

[6] ——, "Scanflow-k8s: Agent-based framework for autonomic management and supervision of ML workflows in kubernetes clusters," in *2022 22nd IEEE International Symposium on Cluster, Cloud and Internet Computing (CCGrid)*, 2022, pp. 376–385.

[7] C. Zhang, M. Yu, W. Wang, and F. Yan, "Enabling cost-effective, SLO-aware machine learning inference serving on public cloud," *IEEE Transactions on Cloud Computing*, vol. 10, no. 3, pp. 1765–1779, 2022.

[8] Y. Lee, A. Scolari, B.-G. Chun, M. D. Santambrogio, M. Weimer, and M. Interlandi, "PRETZEL: Opening the black box of machine learning prediction serving systems," in *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI'18)*. USENIX Association, oct 2018, pp. 611–626.

[9] PyTorch, "Torchserver," 2020. [Online]. Available: https://pytorch.org/serve/

[10] TensorFlow, "Tensorflow - serving models," 2021. [Online]. Available: https://www.tensorflow.org/tfx/guide/serving

[11] N. Hasabnis, "Auto-tuning tensorflow threading model for cpu backend," in *2018 IEEE/ACM Machine Learning in HPC Environments (MLHPC)*. IEEE Computer Society, nov 2018, pp. 14–25.

[12] T. Lorido-Botran, J. Miguel-Alonso, and J. A. Lozano, "A review of auto-scaling techniques for elastic applications in cloud environments," *Journal of Grid Computing*, vol. 12, no. 4, pp. 559–592, oct 2014.

[13] A. Gujarati, S. Elnikety, Y. He, K. S. McKinley, and B. B. Brandenburg, "Swayam: Distributed autoscaling to meet SLAs of machine learning inference services with resource efficiency," in *18th ACM/IFIP/USENIX Middleware Conference*, ser. Middleware'17. ACM, 2017, pp. 109–120.

[14] A. Shahraki, M. Abbasi, A. Taherkordi, and A. D. Jurcut, "A comparative study on online machine learning techniques for network traffic streams analysis," *Computer Networks*, vol. 207, p. 108836, 2022.

[15] P. Liu and J. Guitart, "Performance comparison of multi-container deployment schemes for HPC workloads: an empirical study," *Journal of Supercomputing*, vol. 77, pp. 6273–6312, 2020.

[16] ——, "Performance characterization of containerization for HPC workloads on InfiniBand clusters: an empirical study," *Cluster Computing*, vol. 25, pp. 847–868, 2022.

[17] V. J. Reddi, C. Cheng, D. Kanter, P. Mattson, G. Schmuelling, C.-J. Wu, B. Anderson, M. Breughe, M. Charlebois, W. Chou, R. Chukka, C. Coleman, S. Davis, P. Deng, G. Diamos, J. Duke, D. Fick, J. S. Gardner, I. Hubara, S. Idgunji, T. B. Jablin, J. Jiao, T. S. John, P. Kanwar, D. Lee, J. Liao, A. Lokhmotov, F. Massa, P. Meng, P. Micikevicius, C. Osborne, G. Pekhimenko, A. T. R. Rajan, D. Sequeira, A. Sirasao, F. Sun, H. Tang, M. Thomson, F. Wei, E. Wu, L. Xu, K. Yamada, B. Yu, G. Yuan, A. Zhong, P. Zhang, and Y. Zhou, "Mlperf inference benchmark," in *2020 ACM/IEEE 47th Annual International Symposium on Computer Architecture (ISCA)*, 2020, pp. 446–459.

[18] D. Crankshaw, X. Wang, G. Zhou, M. J. Franklin, J. E. Gonzalez, and I. Stoica, "Clipper: A Low-Latency online prediction serving system," in *14th USENIX Symposium on Networked Systems Design and Implementation (NSDI'17)*. USENIX Association, mar 2017, pp. 613–627.

[19] W. Wang, J. Gao, M. Zhang, S. Wang, G. Chen, T. K. Ng, B. C. Ooi, J. Shao, and M. Reyad, "Rafiki: Machine learning as an analytics service system," *Proc. VLDB Endow.*, vol. 12, no. 2, pp. 128–140, oct 2018.

[20] Kserve, "Highly scalable and standards based model inference platform on kubernetes for trusted AI," 2021. [Online]. Available: https://kserve.github.io/website/master/

[21] Seldon, "Machine learning deployment for enterprise," 2021. [Online]. Available: https://www.seldon.io/

[22] Intel, "Math kernel library," 2018. [Online]. Available: https://software.intel.com/en-us/mkl

[23] C. Park and S. Paul, "Load-testing TensorFlow Serving's REST Interface," July 2022. [Online]. Available: https://blog.tensorflow.org/2022/07/load-testing-TensorFlow-Servings-REST-interface.html

[24] Morgan Funtowicz, "Scaling up BERT-like model Inference on modern CPU - Part 1," April 2021. [Online]. Available: https://huggingface.co/blog/bert-cpu-scaling-part-1

[25] ——, "Scaling up BERT-like model Inference on modern CPU - Part 2," November 2021. [Online]. Available: https://huggingface.co/blog/bert-cpu-scaling-part-2

[26] S. Wang, Z. Ding, and C. Jiang, "Elastic scheduling for microservice applications in clouds," *IEEE Transactions on Parallel and Distributed Systems*, vol. 32, no. 01, pp. 98–115, jan 2021.

[27] H. Shen, L. Chen, Y. Jin, L. Zhao, B. Kong, M. Philipose, A. Krishnamurthy, and R. Sundaram, "Nexus: A GPU cluster engine for accelerating DNN-based video analysis," in *27th ACM Symposium on Operating Systems Principles*, ser. SOSP'19. ACM, 2019, pp. 322–337.

[28] V. Medel, R. Tolosana, J. A. Bañares, U. Arronategui, and O. F. Rana, "Characterising resource management performance in Kubernetes," *Computers & Electrical Engineering*, vol. 68, pp. 286–297, 2018.

[29] P. Liu and J. Guitart, "Fine-grained scheduling for containerized hpc workloads in kubernetes clusters," 2022. [Online]. Available: https://arxiv.org/abs/2211.11487

[30] Google, "Cgroups-cpus." [Online]. Available: https://kernel.googlesource.com/pub/scm/linux/kernel/git/glommer/memcg/+/cpu_stat/Documentation/cgroups/cpu.txt