AsyFunc: A High-Performance and Resource-Efficient Serverless Inference System via Asymmetric Functions

Anonymous Author(s)
Submission Id: #181

ABSTRACT

Recent advances in deep learning (DL) have spawned various intelligent cloud services with well-trained DL models. Nevertheless, it is nontrivial to maintain the desired end-to-end latency under bursty workloads, raising critical challenges on high-performance while resource-efficient inference services. To handle burstiness, some inference services have migrated to the serverless paradigm for its rapid elasticity. However, they neglect the impact of the time-consuming and resource-hungry model-loading process when scaling out function instances, leading to considerable resource inefficiency for maintaining high performance under burstiness.

To address the issue, we open up the black box of DL models and find an interesting phenomenon that the sensitivity of each layer to the computing resources is mostly anti-correlated with its memory resource usage. Motivated by this, we propose *asymmetric functions*, where the original Body Function still loads a complete model to meet stable demands, while the proposed lightweight Shadow Function only loads a portion of resource-sensitive layers to deal with sudden demands effortlessly. By parallelizing computations on resource-sensitive layers, the surging demand can be well satisfied, though the rest layers are performed serially in Body Functions only. We implement asymmetric functions on top of Kubernetes and build a high-performance and resource-efficient inference serving system named AsyFunc with a new auto-scaling and scheduling engine. Experimental results driven by production traces indicate that compared to the state-of-the-art, AsyFunc improves resource efficiency by 23.4% while providing consistent performance guarantees under burstiness.

KEYWORDS

serverless computing, real-time inference serving, request burstiness, resource efficiency

ACM Reference Format:
Anonymous Author(s). 2023. AsyFunc: A High-Performance and Resource-Efficient Serverless Inference System via Asymmetric Functions. In *Proceedings of 14th ACM Symposium on Cloud Computing (SoCC '23).* ACM, New York, NY, USA, 14 pages. https://doi.org/10.1145/nnnnnnn.nnnnnnn

1 INTRODUCTION

Deep Learning (DL) has enabled a variety of intelligent applications in recent years, from virtual assistants to smart traffic management. According to data statistics from China's largest local life service platform, millions of queries are processed every minute using DL models [62]. On Facebook alone, more than 200 trillion inference queries are processed daily [34]. Typically, these models go through two phases for each application scenario: offline *training* to achieve the desired accuracy by iteratively tuning the model parameters, and online *inference* to perform user-facing tasks in real-time. In contrast to training, model inference imposes strict requirements on real-time performance, especially the end-to-end latency specified by service level objectives (SLOs) [10, 64]. For example, 98% of user requests should complete within 200 ms, and SLO violations result in a poor user experience and potentially lower business revenue [64]. A recent report shows that a 100 ms increase in latency can lead to a 1% decrease in revenue [23].

However, the ubiquitous *bursts* of user requests make it difficult to maintain desired SLOs [2, 41] as more resources are suddenly requested but are not available at the moment (e.g., occupied by another service or idle but not initialized). A common practice to deal with bursts is *over-provisioning*, i.e., preparing sufficient resources in advance, which would result in a considerable waste of resources during valley periods [61]. The problem posed by bursts is exacerbated in an inference platform because DL models are generally resource-hungry. For example, the GPT-3 model [7] consumes 325 GB of memory for storing its parameters, as well as necessary computing resources to run the inference, meaning that prohibitive amounts of resources need to be allocated in advance. This overhead increases dramatically as more DL models are served simultaneously [27] and as the models grow larger over time, especially with the recent emergence of large language models such as ChatGPT [9]. Therefore, a question emerges about *providing high-performance yet resource-efficient inference services despite bursts.*

There have already been some attempts at this question. Amazon SageMaker [4] is a well-known inference platform that uses virtual machines (VMs) to execute inferences. Despite low operational costs, the bulky VMs make scaling too slow to meet real-time requirements during sudden spikes in requests. More recently, *serverless computing* offers opportunities for dealing with bursts [59]. For example, MArk [64] combines sparse VMs and elastic serverless functions[1] for model serving under bursts, demonstrating the potential benefits of serverless. Industry products such as Amazon Alexa [3] and Netflix content delivery [45], have also gradually deployed their services on serverless platforms, which can respond to fluctuating workload levels in a timely and cost-effective manner due to the rapid elasticity and fine-grained billing that serverless offers [2, 27, 62].

Despite the prominent advantage of serverless in handling bursts, we note that the unavoidable model-loading process when creating new function instances significantly limits the benefits of auto-scaling for the following two reasons. (1) *High model-loading latency invalidates the reactive on-demand scaling policy.* According to our measurements, the model-loading latency can be 10 to 100 times higher than the inference latency, making real-time services impossible during bursts. (2) *High resource requirements prevent the proactive prediction-based scaling policy.* Since a DL model can consume hundreds to thousands of megabytes of memory, prewarming

[1]The serverless function contains a piece of user code and can be instantiated as an individual execution unit.

a sufficient number of function instances in advance can result in significant resource consumption. Given the unpredictability of future requests, pre-warmed instances typically far surpass the actual demands [46]. These issues ultimately preclude high performance while resource-efficient inference services under bursts.

By addressing the scaling problem as a consequence of the time-consuming and resource-hungry model-loading process, we aim to arm serverless functions with resource-efficient scaling capabilities while maintaining consistent performance. Rather than viewing the entire DL model as a complete black box, as has been the case in previous works [2, 27, 62], we identify unique opportunities that arise from the heterogeneous behavior of the internal layers. In particular, we find that the sensitivity of each layer to computing resources is almost negatively correlated with its parameter size, as shown later in Figure 2. This implies that loading a small number of resource-sensitive layers can achieve comparable inference latency as loading a complete model, but significantly reduces the overhead of loading the model when provisioning a new instance. For example, for one of the latest object detection models YOLOv8x [44], if the top 10% most resource-sensitive layers get loaded, the model-loading time and memory consumption can be reduced from 100 ms and 131 MB to 11 ms and 6 MB, respectively, while the inference latency can remain almost unchanged by allocating slightly more CPU cores temporarily.

Driven by the observation on the completeness of DL models, we propose a fine-grained layer-level scaling policy in combination with the existing coarse-grained model-level scaling policy. The former scales out functions with a portion of resource-sensitive layers which we call *Shadow Function*, and the latter scales out functions still with a complete model which we call *Body Function*. Body Function basically loads a complete model to maintain consistent performance under stable demands, and adjusts periodically to long-term fluctuations in workload levels (e.g., 30 s). In comparison, Shadow Function loads only a portion of resource-sensitive layers so that it can respond quickly to sudden demands. When a burst arises, it will be too slow to provision an additional Body Function, but the Shadow Function can be provisioned in a timely manner. By pairing an existing Body Function with a new "asymmetric" Shadow Function, they can perform inference executions on the resource-sensitive layers together, while only the Body Function is still responsible for the rest layers, achieving high resource efficiency to maintain consistent performance at bursts.

To fulfill Asymmetric Functions, we develop a serverless-oriented inference serving system called AsyFunc. Specifically, we make the following four contributions:

- We investigate the scaling problem of current serverless inference platforms caused by the time-consuming and resource-hungry model-loading process, and propose the key concept of *asymmetric functions* with different levels of model completeness (i.e., Body Function vs. Shadow Function) to solve this problem.
- We develop a heuristic algorithm for the model-level scaling (MLS) that adapts periodically and a priority-based heuristic algorithm for the layer-level scaling (LLS) that adapts on demand. Both of them aim to maximize the resource efficiency without hurting the performance. To make full use of the asymmetric functions, we devise an adaptive scheduling scheme to dispatch requests in real time.
- To enable fine-grained scaling at the layer level, we implement a high-performance and resource-efficient inference serving system AsyFunc on top of Kubernetes. For efficient coordination between Body and Shadow Functions during collaborative inference executions, we establish an efficient communication and synchronization mechanism that imposes negligible overhead on inference performance and resource consumption.
- We conduct extensive experiments to evaluate the performance of AsyFunc. Based on real-world traces, the evaluation results demonstrate the effectiveness of AsyFunc, which improves resource efficiency by up to 23.4% over the state-of-the-art and keeps the SLO violation rate at a low level despite the bursts.

The rest of the paper is organized as follows. In Section 2, we present the background of serverless inference and analyze the opportunities and challenges of layer-level scaling, which motivates the design of AsyFunc in Section 3. To achieve high-performance and resource-efficient inference serving, we present well-designed model-level and layer-level scaling policies in Section 4. For real-time inference serving, we develop an adaptive scheduling scheme and an efficient coordination mechanism in Section 5. We implement AsyFunc on top of Kubernetes in Section 6 and perform extensive experiments to evaluate AsyFunc in Section 7. Finally, we discuss related work in Section 8 and present concluding remarks in Section 9.

2 BACKGROUND AND MOTIVATION

In this section, we first analyze the scaling issue of existing serverless inference platforms. Then, we illustrate the layer heterogeneity of DL models. Finally, we discuss the opportunities and challenges of layer-level scaling.

2.1 Scaling Issue of Serverless Inference

DL models consist of different types of neural network layers with different parameters (a.k.a. weights) [42], such as convolutional layers, fully connected layers, and pooling layers. The layers and their connections form a computationally directed acyclic graph (DAG), and inference executions are performed on these layers sequentially along the DAG. The DL models have recently achieved impressive performance in many areas, from image classification [21, 52] to natural language processing [14]. Major cloud providers, such as Amazon and Alibaba, have deployed DL models to provide inference services as a key part of their applications [43, 57]. However, online inference services are usually both latency-critical and resource-hungry, which leads to an inevitable trade-off between performance and resource efficiency, especially given the significant burstiness observed in the inference requests from users [6, 17, 29].

Serverless computing is considered to be a promising choice for handling bursty workloads due to its rapid elasticity and fine-grained billing [30, 36]. Therefore, model inference based on serverless platforms (i.e., serverless inference) has received widespread attention [2, 27, 62, 64]. In serverless inference, each DL model is deployed separately in a function instance (e.g., Docker [35],

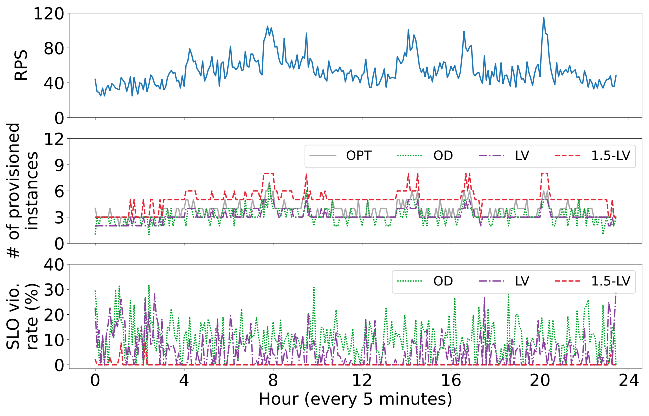


Figure 1: RPS over time, and the corresponding number of provisioned instances and SLO violation rate under different scaling policies. Existing policies involve inevitable trade-offs between SLO satisfaction and resource efficiency.

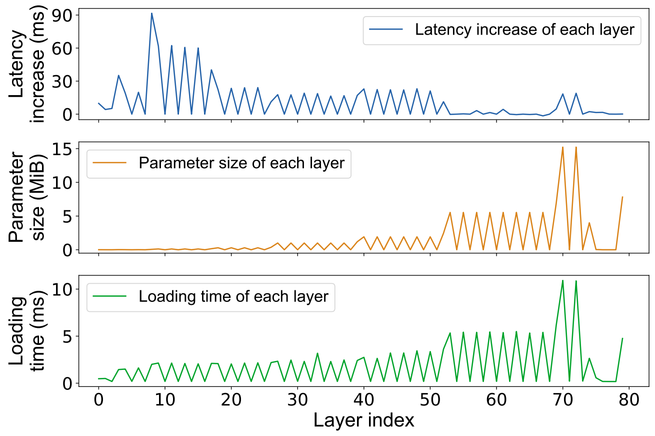


Figure 2: The latency increase of each layer when the batch size increases, and the parameter size and loading time of each layer in the EfficientNet-b5 model. Surprisingly enough, the latency increase is almost negatively correlated with the parameter size and the layer's loading time.

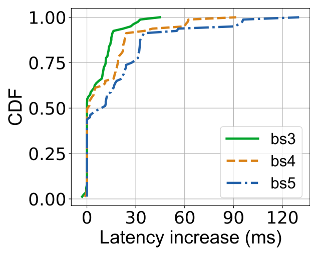


Figure 3: The CDF of latency increase of each layer when the batch size increases from 2 to 3, 4, and 5.

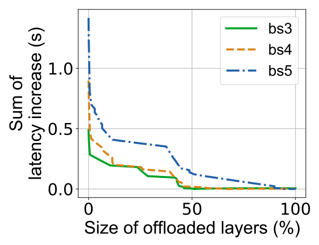


Figure 4: The increase in inference latency when loading different percentages of layers into the Shadow Function.

Firecracker [1]) that scales out/in as workload levels grow/drop. It is worth noting that before providing inference services, the function instance needs to be created first and then complete the model-loading process, i.e., loading the model file into the memory and initializing the model, which is very time-consuming. To maintain low inference latency in case of bursts, one common remedy is prewarming the instances and pre-loading the model beforehand [12, 16]. However, prewarming a sufficient number of instances for each DL model can cause a huge waste of resources, especially when there are many different models and under significant workload fluctuation. A recent study [37] proposes a remedy that saves resources by only prewarming the instances without instantiating any model and thus can be shared by all models. In this way, the corresponding model will be loaded into the instance on demand. However, we notice that the time for loading the model is still much longer than that for performing an inference execution (e.g., the gap is 15× for the InceptionV3 model), which disables persistent real-time inference serving.

We attribute the above scaling problem to the time-consuming and resource-hungry behavior of the DL *model-loading process*. We conduct an experiment to verify this problem with a real-world trace from Twitter [54]. We select three classical scaling policies, namely on-demand scaling (OD), prediction-based scaling with the last value (LV) [8], and prediction-based scaling by multiplying the last value by a factor k (k-LV, k is 1.5 here), and also present the optimal scaling policy (OPT) that assumes zero model-loading latency. The requests per second (RPS) during a day, the number of provisioned instances[2] and the SLO violation rate are plotted in Figure 1, where we do not present the SLO violation rate of OPT since it is always zero.

Two conclusions can be summarized from this experiment: (1) The long model-loading process is likely to cause SLO violations if we scale out instances conservatively (the average SLO violation rates of the OD and LV are 10.52% and 5.46%, respectively); (2) The high demand for computing and storing will lead to a waste of resources if we scale out instances aggressively, where many instances are provisioned but actually unused (the average number of created instances of the k-LV is 5, about 1.3× of the actual demand as indicated by the OPT). We notice that **the contradiction between SLO satisfaction and resource efficiency arises from the existing coarse-grained model-level scaling** that always loads the whole DL model into the new instances. Thus, we wonder whether we can reduce unnecessary resources without incurring perceptible SLO violations by opening the black box of DL models and studying its internal layers.

[2]We refer to function instances that pre-load DL models as provisioned instances.

2.2 Heterogeneous Behavior of Layers

To improve the scaling efficiency, we identify the opportunity provided by the internal layers. It is worth noting that *batching* is a useful approach to increase the processing rate by grouping a number of user requests together, and the batch size denotes this number of requests. A larger batch size can increase the processing rate but at the expense of latency performance. We measure

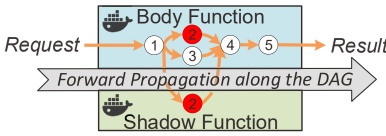


Figure 5: Illustration of collaboration between Body Function and Shadow Function in the layer-level scaling. The red circles denote the resource-sensitive layer and computations on this layer are partially offloaded to the Shadow Function.

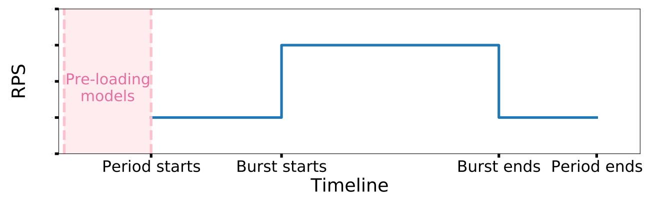
the inference latency increase, the parameter size, and the loading time of each layer inside the EfficientNet-b5 model [53] when the batch size increases from 2 to 4, as shown in Figure 2. As we can see, the sensitivity of each layer to the computing resources varies. Specifically, **the inference latency increase is almost negatively correlated with the layer size**, while **the model-loading latency is generally proportional to the layer size**[3]. That is to say, both the memory consumption and model-loading latency can be reduced significantly by loading a small number of resource-sensitive layers into the instances.

We further plot the cumulative distribution function (CDF) of the latency increase of each layer when the batch size increases from 2 to 3, 4, and 5, in Figure 3. It is obvious that the latency increase of most layers is only marginal, while only a few layers show a high latency increase. For example, when the batch size doubles to four, although the highest latency increase is 92 ms, the latency of about 90% layers increases no more than 24 ms, and the latency of about half of the layers remains almost unchanged (Note that the inference latency of the whole model is increased from 673 ms to 1571 ms). The above observation motivates the design of *fine-grained layer-level scaling* for handling bursts, where only some of the resource-sensitive layers will be loaded rather than a complete model.

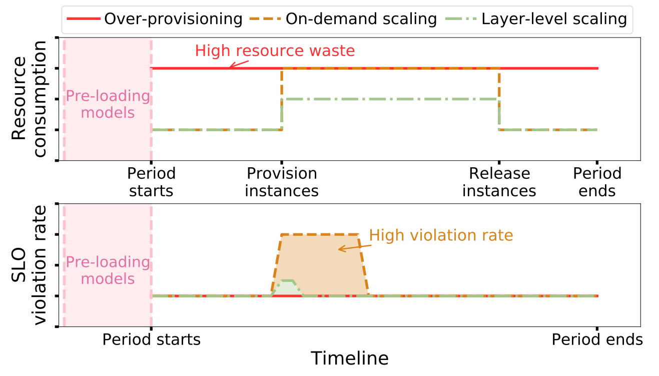
2.3 Opportunities of the Layer-level Scaling

Opportunities to handle bursts. In existing inference serving systems, to avoid perceptible latency increase because of a sudden spike of user requests, the heavy *model-level scaling* policy re-directs the additional requests to new instances that load a complete model, and these instances process their respective requests separately. Based on the observation explained in Section 2.2, we dig out the potential benefits of the *layer-level scaling* policy here. Specifically, when bursts arise, the layer-level scaling allows prewarmed function instances to load only a small number of resource-sensitive layers in a timely manner which we call *Shadow Function*. During the inference process, the original function instances loaded with the complete model which we call *Body Function*, can offload the computations from additional requests on these resource-sensitive layers to the Shadow Function, as illustrated in Figure 5. The two instances perform computations on these common layers collaboratively and in parallel, and Body Function is also responsible for computations on the remaining layers. This helps handle sudden spikes in demand without heavy resource waste or performance degradation.

[3]We provide a brief explanation and more proof from other DL models of this anti-correlation phenomenon in Appendix.



(a) The workload level over an adaptation period.



(b) The resource consumption and SLO violation rate over an adaptation period.

Figure 6: Comparison among three scaling policies when dealing with bursts.

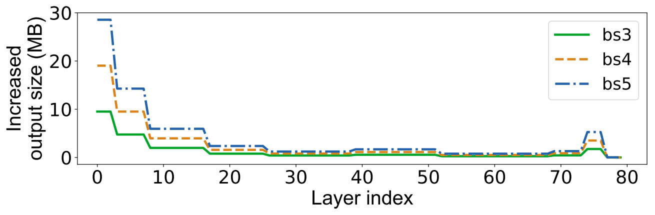


Figure 7: The increased output size of each layer in the EfficientNet-b5 model as the batch size increases from 2 to 3, 4, and 5.

Evidence supporting the benefits. Figure 4 shows the increase in the inference latency when offloading computations on different percentages of layers, where the layers are sorted in descending order by the latency increase as the batch size increases unless otherwise noted. Note that 100% of the offloaded layers indicate that all the computations from additional requests are offloaded to a new instance, which is equivalent to the model-level scaling policy. It is worth noting that the latency increase drops significantly although computations on only a small portion of layers get offloaded. In particular, the latency increase is almost zero when calculations on merely half of the layers get offloaded.

Preliminary experiments. We perform an experiment to compare the performance of the *layer-level scaling* policy with the model-level one (including *over-provisioning* and *on-demand scaling*) as workload level grows unexpectedly, as shown in Figure 6. The *over-provisioning* policy provisions enough instances that pre-load models in case of sudden request surges, while the *on-demand scaling* policy only provisions some instances for stable workloads (e.g., the most common workload level) and provisions additional instances when bursts happen. By contrast, although the *layer-level*

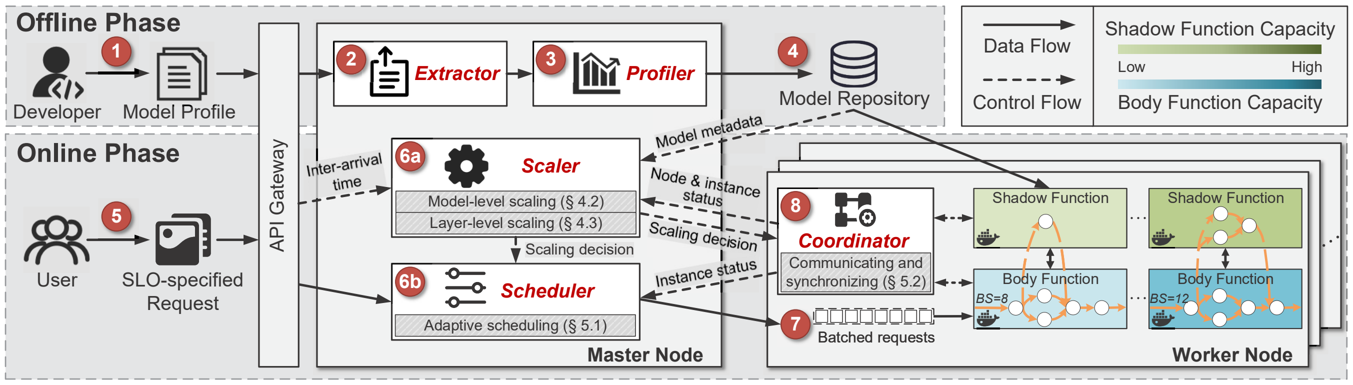


Figure 8: System overview of AsyFunc.

scaling policy also provisions instances only for stable workloads, it scales rather fast when bursts come by loading only a small number of resource-sensitive layers. According to our measurements, the model-loading time can be reduced by one to two orders of magnitude compared to that of loading a complete model. As shown in Figure 6, the *layer-level scaling* policy can save resource usage and waste significantly while keeping a low SLO violation rate, revealing the great potential of the layer-level scaling policy.

Challenges to address. Despite proving to have great potential for serverless inference, the layer-level scaling policy faces critical challenges to facilitating efficient coordination between Body Function and Shadow Function. Firstly, the offloaded layers of the DL model and the amount of offloaded requests need to be tuned dynamically in accordance with request fluctuations, which affects the inference latency and overall resource efficiency. These two parameters must be determined at runtime based on the current instances' status and actual workload levels. Moreover, offloading partial layers' computations to another instance causes the inference process to span both instances, so the output of some layers needs to be transferred between instances. Figure 7 shows the size increase of the output data of each layer when the batch size increases from 2 to 3, 4, and 5. It can be seen that the layers are heterogeneous in terms of the output data size. Thus, when deciding on offloaded layers during the layer-level scaling, we need to consider the output data size as well to reduce the data transfer overhead. Finally, since current serverless platforms are a poor fit for supporting fine-grained coordination between instances, a new communication and synchronization mechanism needs to be devised as a module with minimum overhead.

3 SYSTEM DESIGN

In this section, we present the design of AsyFunc, a serverless inference system that supports both coarse-grained model-level scaling and fine-grained layer-level scaling.

3.1 Design Philosophy

To take advantage of the fine-grained layer-level scaling discussed in Section 2.3, we develop a high-performance and resource-efficient serverless inference system with a new auto-scaling and scheduling engine. The main idea behind the engine lies in two aspects: (1) leveraging the *Body Function* that loads a complete model to handle stable inference workloads, and (2) leveraging the *Shadow Function* that loads a selective percentage of resource-sensitive layers to handle spiky inference workloads effortlessly. When bursts arise, the resource-sensitive parts of the DL model can be executed in a parallel manner using the Shadow Function, while other non-resource-sensitive parts are executed serially in the Body Function only. In this way, AsyFunc aims to achieve a good balance between SLO satisfaction and resource efficiency despite the ubiquitous bursts of inference requests.

3.2 System Overview

Based on the above design philosophy, we establish AsyFunc. Figure 8 shows the system overview of AsyFunc with the *Extractor*, *Profiler*, *Scaler*, *Scheduler*, and *Coordinator* as its core components.

In the offline phase, ❶ after the well-trained DL model is submitted to the platform, ❷ the *Extractor* automatically extracts its layer information (including the layer type and parameters) and structure information (i.e., the connections between adjacent layers). Then, ❸ the *Profiler* analyzes the resource sensitivity of each layer as well as other necessary metadata information. ❹ Finally, the layer, structure, and metadata information are stored in the model repository.

In the online phase, ❺ after receiving an SLO-specified request from users through the API Gateway, 6a the *Scaler* on the master node collects the inter-arrival time information that will be used to calculate historical RPS for generating the scaling decision periodically. The scaling decision includes scaling out/in new Body Function instances periodically (i.e., the model-level scaling), and adjusting the maximum supported batch size of existing Body Function instances at bursts by scaling up/down Shadow Function instances (i.e., the layer-level scaling). Meanwhile, 6b the *Scheduler* on the master node is responsible for dispatching the incoming request to an appropriate Body Function instance on worker nodes. ❼ Requests are cached in the waiting queue until they reach the maximum batch size or timeout. Then, all requests in the queue are batched together and sent to an instance on worker nodes. ❽ The Body Function instance can offload partial layers' computations to its paired Shadow Function instance at bursts, and the two instances are coordinated by the *Coordinator* on the worker node. In the following, we introduce each module in detail.

3.3 Extractor & Profiler

As illustrated in Section 2.1, a DL model is composed of various types of layers that exhibit different behaviors. AsyFunc needs to extract layer information from model profiles submitted by developers and analyze the resource sensitivity of every layer. The *Extractor* and *Profiler* modules are responsible for these tasks.

The *Extractor* first parses the model profile and generates a profile for each layer as an individual file which contains the layer name, type, and parameters. In this way, layers can be selectively loaded as a sub-model into a Shadow Function instance at runtime, and the instance can perform inference computations only on these layers. Then, the *Extractor* reads the DAG structure information, i.e., connections between layers. With this connection information, the output data from the previous layer can be correctly passed to the next layer during an inference execution.

As we find that the resource sensitivity is directly related to the number of multiply–accumulate operations (MACs) of the layer, the *Profiler* first estimates its sensitivity based on the MACs, groups adjacent layers with similar sensitivity into a *layer block*, and filters out layer blocks with low sensitivity or very large memory consumption. Instead of each individual layer, the layer block as a whole will be selectively loaded into the Shadow Function to avoid frequent data transfer during an inference execution. For example, as shown in Figure 2, the layers numbered 8 to 15 will be merged into one block and the layer numbered 3 will be treated as another separate block. Next, the *Profiler* obtains the metadata information of each layer block by performing inference executions on these blocks separately, including the inference latency (l), parameter size (p), and output data size (d) under different numbers of CPU cores (c) and batch sizes (b), as a 5-tuple $< c, b, l, p, d >$.

3.4 Scaler

The *Scaler* is responsible for scaling out/in Body Functions periodically and scaling up/down Shadow Functions at bursts.

On the one hand, the *Scaler* estimates the average workload level during the next period based on historical inter-arrival time information. When it is expected to rise or decline, the *Scaler* will make scaling decisions through model-level scaling. Based on the model metadata and real-time node & instance status (including the number of free CPU cores on each node and the number of allocated CPU cores to each instance), the *Scaler* decides the configuration of the Body Function (i.e., # of CPU cores) and which worker node to accommodate it. On the other hand, when bursts arise unexpectedly within each period, the *Scaler* will make scaling decisions through layer-level scaling. The purpose of the layer-level scaling policy is to increase the maximum supported batch size of existing Body Function instances temporarily, by scaling up well-sized Shadow Function instances for a collaborative inference execution. Based on the model metadata and the reserved resources on each worker node, the *Scaler* decides which worker node to provision the Shadow Function and which layer blocks to load. The scaling details will be discussed in Section 4.

Note that the container pool is distributed among worker nodes. In other words, each worker node reserves one empty container that can load any layer blocks from any models on demand. In this way, the Shadow Function can be provisioned quickly by loading only the resource-sensitive layers without waiting for creating a container. Those reserved resources can also make room for other Body Functions when the remaining resources on that worker node are insufficient at high workload levels. In Section **??**, we will evaluate the impacts of resource reservation on performance and resource efficiency.

3.5 Scheduler & Coordinator

Both the *Scheduler* and *Coordinator* help realize real-time inference serving. First, the *Scheduler* is responsible for scheduling incoming requests to the best candidate instance, either the Body Function instance only or the Body and Shadow Function instance pair. Specifically, the *Scheduler* collects the instance status on all worker nodes, including # of allocated CPU cores, the maximum supported batch size, and the instance state (i.e., idle or busy). As requests continue arriving in the waiting queue, the *Scheduler* forwards them as a batch to the best idle Body Function instance. In short, we regard the instance that achieves the maximum throughput (i.e., the actual batch size divided by the inference latency) per CPU core as the best one. The scheduling details will be discussed in Section 5.1.

Second, the *Coordinator* is responsible for coordinating each instance pair during a collaborative inference execution in three ways: creating and destroying instances, facilitating efficient data transfer, and controlling correct synchronization. Note that the Body Function and its Shadow Function partner would be created on the same worker node as much as possible to avoid time-consuming cross-server communication. The coordination details will be discussed in Section 5.2.

4 FINE-GRAINED SCALING MECHANISM

In this section, we describe in depth the design of model-level scaling and layer-level scaling in the *Scaler*.

4.1 Scaling Principle

First, we summarize the following two scaling principles:

(1) The model-level scaling policy for Body Functions aims at satisfying stable workloads which are represented by the *expected average RPS*.
(2) The layer-level scaling policy for Shadow Functions aims at satisfying spiking workloads which are represented by the *unexpected instantaneous RPS*.

Before making the model-level scaling decision periodically, the *Scaler* needs to evaluate whether the existing instances can meet the resource demand under the expected average RPS. Specifically, since it is Body Function instances that directly serve the user requests, the *Scaler* calculates the maximum supported RPS by all the Body Function instances paired with Shadow Function instances or not. Suppose there are n Body Function instances distributed in different worker nodes. For a Body Function instance i, we use b^i, t_q^i, and t_l^i to represent the batch size, queuing time, and the inference latency of the instance, respectively. As t_q^i approaches zero, the RPS that existing instances can handle (denoted as R_{max}) reaches the maximum. Thus, R_{max} can be calculated by:

$$R_{max} = \sum_{i=1}^{n} R_{max}^i = \sum_{i=1}^{n} \max\{\frac{b^i}{t_l^i} \mid t_l^i \leq t_{SLO}\}, \quad (1)$$

where t_{SLO} is the latency SLO. With Equation (1), the *Scaler* makes the following decisions:

(1) $R > \alpha R_{max}$. It means that existing instances cannot satisfy the predicted average RPS. The *Scaler* will scale out new Body Function instances by using the model-level scaling policy. We denote the residual RPS as R_k which is equal to $R - \alpha R_{max}$.
(2) $\beta R_{max} \le R \le \alpha R_{max}$. It indicates that existing instances can serve requests stably. The *Scaler* should take no action to avoid system instability caused by frequent scaling and node status switching.
(3) $R < \beta R_{max}$. It means that existing instances are beyond the demand under the predicted average RPS. The *Scaler* will release instances to reduce resource waste.

Once the *Scaler* detects that the instantaneous RPS exceeds the maximum supported RPS, i.e., $R_{burst} > \gamma R_{max}$, it will provision Shadow Functions through the layer-level scaling policy. Since only the most resource-sensitive but memory-efficient layers will be loaded, the online model-loading process makes no perceptible impact on the latency performance. When the instantaneous RPS drops below the maximum supported RPS, the Shadow Functions will be released.

4.2 Model-Level Scaling

Based on Equation (1), the *Scaler* scales out new Body Function instances to satisfy the residual RPS or scales in to save resources.

However, it is nontrivial to decide on an appropriate configuration (i.e., # of CPU cores) as the RPS fluctuates. As the number of allocated CPU cores grows, the instance could achieve a lower inference latency or process a larger batch at a time with similar latency. In terms of resource efficiency, allocating more CPU cores would reduce the processing efficiency represented by the maximum throughput per core but increase the memory efficiency as fewer instances will be created. Thus, to select the best configuration, the model-level scaling policy is formulated as follows:

$$\min \sum_{i=1}^{N_1} \left(c^i + \rho m^i\right) \tag{2}$$

$$\text{s.t. } R_k \le \alpha \sum_{i=1}^{N_1} \max\{\frac{b^i}{t_l^i} \mid t_l^i \le t_{SLO}\}, \tag{3}$$

where c^i and m^i represent # of CPU cores and memory consumption of the instance $i \in 1 \ldots N_1$ to be created, and ρ is a normalizing factor related to the DL model, calculated by dividing the MACs of the whole model by its parameter size. Considering that the Body Function instance loads a complete model in which m^i is a fixed value, the objective function (2) can be converted into min. $(\sum_i^{N_1} c^i + N_1 \cdot m)$. To solve this problem, we develop a heuristic algorithm for model-level scaling (MLS) which decides the number of created instances N_1 and their configuration c^i.

As shown in Algorithm 1, when deciding to *scale out* (Line 2), for each core number and batch size (Lines 3-4), MLS first estimates the average longest service time that is equal to the average longest queuing time plus inference latency (Line 5). If it is within the latency SLO (Line 6), MLS calculates the **resource efficiency** that is defined as the maximum throughput per unit of computing resources plus that per unit of memory resources (Line 7). Note that the maximum number of CPU cores and the maximum batch size are restricted to C_{max} and B_{max}, respectively, because it would be ineffective to further reduce latency by allocating more cores. Finally, MLS chooses the configuration that maximizes the resource efficiency (Lines 8-10) to create Body Function instances until $R \le \alpha R_{max}$ (Lines 16-17). On the contrary, when deciding to *scale in* (Line 19), MLS will calculate the maximum resource efficiency achieved by each instance in X and sort them in ascending order (Line 20). Then, MLS will destroy them one by one until $R \ge \beta R_{max}$.

Algorithm 1 Heuristic Algorithm for Model-Level Scaling

```
1:  $X$: the set of existing Body Function instances;
    $x_i$: the i-th instance in $X$;
    $t_l^{c,b}$: the estimated inference latency when the number of CPU
    cores and batch size is $c$ and $b$, respectively;
    $t_s^{c,b}$: the estimated longest service time when the number of
    CPU cores and batch size is $c$ and $b$, respectively;
    $c_{best}$: the selected configuration of # of CPU cores of the newly
    created Body Function;
    $\eta_{max}$: the maximum achievable resource efficiency, initialized
    as zero;
2:  if $R > \alpha R_{max}$ then
3:      for $c = 1, 2, \cdots, C_{max}$ do
4:          for $b = B_{max}, B_{max} - 1, \cdots, 1$ do
5:              $t_s^{c,b} = \frac{b-1}{R} + t_l^{c,b}$
6:              if $t_s^{c,b} \le t_{SLO}$ then
7:                  $\eta = \frac{b}{c \cdot t_l^{c,b}} + \frac{b}{\rho m \cdot t_l^{c,b}}$;
8:                  if $\eta > \eta_{max}$ then
9:                      $\eta_{max} = \eta$;
10:                     $c_{best} = c$;
11:                     break;
12:                 end if
13:             end if
14:         end for
15:     end for
16:     while $R > \alpha R_{max}$ do
17:         Create a Body Function with $c_{best}$ of cores on the worker
            node with the most available cores.
18:     end while
19: else if $R < \beta R_{max}$ then
20:     Sort $X$ by $\eta_{max}$ in ascending order;
21:     for $i = 1, 2, \cdots, |X|$ do
22:         if $R < \beta R_{max}$ then
23:             Destroy $x_i$;
24:         else
25:             break;
26:         end if
27:     end for
28: end if
```

Algorithm 2 Heuristic Algorithm for Layer-Level Scaling

```
1:  $X$: the set of unpaired Body Function instances sorted by # of CPU cores in descending order;
    $x_i$: the i-th instance in $X$;
    $c^i$: # of CPU cores of the i-th instance in $X$;
    $c_a^i$: the number of available CPU cores on the worker node where $x_i$ resides;
    $b_B$, $b_S$: the batch size supported by the Body and Shadow Functions, respectively, on the parallel part;
    $b$: the batch size supported by the Body Function on the non-parallel part, where $b = b_B + b_S$;
    $L$: the set of candidate layer blocks sorted in descending order by the ratio of its MACs and its parameter size;
    $\Phi$: the set of selected layer blocks, where $\Phi$ is initialized as $\varnothing$ for each Shadow Function;
    $m_\Phi$: the memory consumption of the selected blocks $\Phi$;
    $\Phi_{best}$: the best set of layer blocks for the Shadow Function;
2:  for $i = 1, 2, \cdots, |X|$ do
3:    if $R_{burst} \le \gamma R_{max}$ and $c_a^i > 0$ then
4:      break;
5:    end if
6:    $c = \min\{c_a^i, C_{max}\}$;
7:    for $\iota$ in $L$ do
8:      $\phi \leftarrow \phi \cup \iota$;
9:      Update $m_\Phi$;
10:     for $b = B_{max}, B_{max} - 1, \cdots, 1$ do
11:       Select $b_B$ to min $|t_{pB}^{c^i, b_B} - t_{pS}^{c, b-b_B}|$;
12:       $t_l^{c,b} = \max\{t_{np}^{c^i,b} + t_{pB}^{c^i,b_B}, t_{np}^{c^i,b} + t_{pS}^{c,b_S}\}$;
13:       $t_s^{c,b} = t_{load}(\phi) + t_l^{c,b}$;
14:       if $t_s^{c,b} \le t_{SLO}$ then
15:         $\eta = \frac{b_S}{c \cdot t_l^b} + \frac{b_S}{\rho m \cdot t_l^b}$;
16:         if $\eta > \eta_{max}$ then
17:           $\eta_{max} = \eta$;
18:           $\phi_{best} = \phi$;
19:           break;
20:         end if
21:       end if
22:     end for
23:   end for
24:   Provision a Shadow Function instance that loads the layer blocks of $\phi_{best}$ on the worker node.
25: end for
```

4.3 Layer-Level Scaling

As shown in Section 2, although the average RPS often varies slowly, the instantaneous RPS fluctuates severely and unexpectedly. When bursts arise, the Shadow Function helps improve R_{max} by increasing the maximum supported batch size of existing Body Function instances. In the following, we propose a layer-level scaling policy that scales up Shadow Function instances in a timely manner to satisfy sudden demands with minimum resource consumption.

Each Body Function instance can be paired with one Shadow Function instance. Suppose N_2 represents the number of unpaired Body Function instances on all worker nodes, so there are at most N_2 Shadow Function instances to be scaled up, where the *Scaler* needs to determine their configurations, including # of CPU cores and the set of loaded layer blocks. The layer-level scaling policy is formulated as follows:

$$\min \sum_{j=1}^{N_2} \left(c^j + \rho m^j\right) \tag{4}$$

$$\text{s.t. } R_{burst} \le \gamma \sum_{j=1}^{N_2} \max\{\frac{b^j}{t_l^j} \mid t_l^j \le t_{SLO}\}, \tag{5}$$

$$t_l^j = \max\{t_{np}^j + t_{pB}^j, t_{np}^j + t_{pS}^j\}, \tag{6}$$

where c^j and m^j represent # of CPU cores and memory consumption of the instance $j \in 1 \ldots N_2$, and t_{np}^j, t_{pB}^j, and t_{pS}^j refer to the inference latency on the non-parallel part of the Body Function, the inference latency on the parallel part of the Body Function, and the inference latency on the parallel part of the Shadow Function plus the data transmission time. Taking the case shown in Figure 5 as an example, t_{np}^j refers to the inference latency of layers numbered 1, 4, and 5, t_{pB}^j is the inference latency of layers numbered 2 and 3 in the Body Function instance, and t_{pS}^j is the inference latency of the layer numbered 2 in the Shadow Function instance as well as the data transmission time. The overall inference latency is the highest one of $(t_{np}^j + t_{pB}^j)$ and $(t_{np}^j + t_{pS}^j)$.

As an NP-hard problem, we convert it into a heuristic algorithm for realizing the layer-level scaling (LLS). First, LLS gives priority to Body Functions with higher configurations to be paired with Shadow Functions. Secondly, layer blocks with higher values of MACs divided by its parameter size will be selected first. Thirdly, the number of CPU cores allocated to Shadow Functions is fixed at C_{max}; however, if the available CPU cores are insufficient on the node where the Body Function resides, all the remaining cores will be allocated. As shown in Algorithm 2, for each Body Function instance (Line 2), LLS first judges whether all instances can satisfy the sudden demand and whether there are available cores (Line 3). If so, LLS determines the number of cores for the Shadow Function, adds one layer block each time, and updates the memory usage (Lines 6-9). Next, for each batch size, LLS determines b_B and b_S to guarantee the two parallel parts almost finish at the same time (Lines 10-11). Then, LLS estimates the inference latency based on Equation (6) and the service time that is equal to the model-loading time and inference latency (Lines 12-13). If the latency SLO can be satisfied (Line 14), LLS chooses the configuration that maximizes the resource efficiency (Lines 15-18) to provision a Shadow Function instance (Line 24). For the next Body Function, LLS repeats the above steps until $R \le \gamma R_{max}$. Similar to MLS, as the burst passes off, LLS scales down by destroying Shadow Function one by one whose η is the smallest.

5 REAL-TIME INFERENCE SERVING

In this section, we present the scheduling algorithm to dispatch incoming requests to the best instance and the coordination mechanism to manage Body and Shadow Function instances, both of which ensure real-time inference serving.

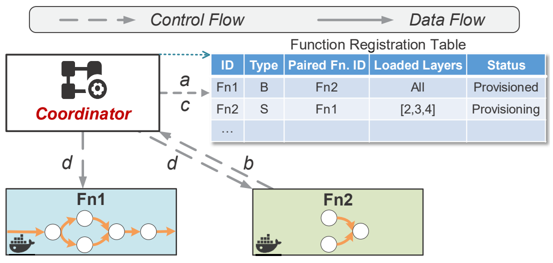
Algorithm 3 Adaptive Scheduling Algorithm

```
1:  $Q$: the waiting queue;
    $t_w$: the waiting time of the first request in $Q$;
    $X$: the set of all Body Function instances sorted by the maximum
    supported batch size in ascending order;
    $x_i$: the i-th instance in $X$;
    $t_l^i$: the estimated inference latency if choosing $x_i$;
    $t_s^i$: the estimated longest service time of all requests if choosing
    $x_i$;
2:  Flag = FALSE;
3:  while a request arrives in $Q$ do
4:    for $i = 0, 1, \ldots, |X|$ do
5:      if $x_i$ is idle then
6:        $t_s^i = t_w + t_l^i$;
7:        Record $t_s^i$ in $T$;
8:        if $t_s^i \leq t_{SLO} < t_s^i \cdot \frac{1+|Q|}{|Q|}$ then
9:          Schedule $Q$ to $x_i$ and clear $Q$;
10:         Flag = TRUE;
11:         break;
12:       end if
13:     end if
14:   end for
15:   if not Flag then
16:     if $t_s^i > t_{SLO}, \forall t_s^i \in T$, then
17:       Choose an idle instance $x_i$ with the smallest $i$;
18:       Schedule a portion of the latest requests in $Q$ to $x_i$ and
          clear $Q$;
19:     else
20:       Wait for the next request;
21:     end if
22:   end if
23: end while
```

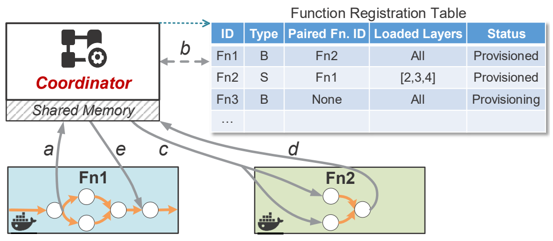
5.1 Adaptive Scheduling

As requests arrive in the waiting queue, the *Scheduler* needs to decide on both *the appropriate time* to dispatch all these requests as a batch and *the best instance* to serve them. The former impacts both the waiting time and inference time as the number of requests in the waiting queue increases, while the latter only impacts the inference time. To maximize the **processing efficiency** (defined as the actual throughput per core) while keeping a low SLO violation rate, the *Scheduler* makes scheduling decisions based on the following observation: Smaller instances generate higher processing efficiency, while larger instances are more robust to workload fluctuations as they can process a larger batch size a time.

Hence, we develop an adaptive scheduling algorithm that prioritizes smaller instances and leaves behind larger ones for dealing with bursts. As shown in Algorithm 3, as a new request arrives (Line 3), the *Scheduler* first calculates the estimated longest service time if choosing the idle instance x_i (Lines 4-6), and records it in a list (Line 7). If that instance satisfies the SLO but is expected to cause SLO violation if waiting for one more request, the *Scheduler* will dispatch all the requests in Q at once and change the flag (Lines 8-10). If the requests are not successfully scheduled in the end (Line 15), there are two cases: (1) None of the idle instances can satisfy the


Control Flow
Data Flow
Function Registration Table
ID | Type | Paired Fn. ID | Loaded Layers | Status
Fn1 | B | Fn2 | All | Provisioned
Fn2 | S | Fn1 | [2,3,4] | Provisioning
...
Coordinator
a
c
d
d
b
Fn1
Fn2

(a) The registration phase.


Function Registration Table
ID | Type | Paired Fn. ID | Loaded Layers | Status
Fn1 | B | Fn2 | All | Provisioned
Fn2 | S | Fn1 | [2,3,4] | Provisioned
Fn3 | B | None | All | Provisioning
...
Coordinator
Shared Memory
b
a
e
c
d
Fn1
Fn2

(b) The inference phase.

Figure 9: Demonstration of the coordination mechanism.

SLO. Hence, to ensure the SLO of as many as requests, the *Scheduler* will selectively dispatch a portion of the latest requests in the queue to an idle instance with the highest configuration (i.e., the smallest i value) and drop other earlier requests [29] (Lines 16-18); (2) All the idle instances are expected not to cause an SLO violation if waiting for one more request. Therefore, the *Scheduler* will wait for the next request (Line 20).

5.2 Coordination Mechanism

As introduced in Section 3.5, the *Coordinator* is distributed on each worker node and manages the life cycle of all functions. Specifically, AsyFunc maintains a Function Registration Table (FRT) that records the metadata information of each function, including the function ID, function type, ID of its paired function, indexes of loaded DL model layers, and function status. Upon receiving a scaling decision from the *Scaler*, the *Coordinator* will create or destroy corresponding functions and update the node and instance status as well as the FRT. At runtime, AsyFunc uses shared memory for fast data transmission between functions during a collaborative inference execution.

As plotted in Figure 9a, when receiving a layer-level scaling decision, ⓐ the *Coordinator* will parse the received message, insert a record into the FRT, and provision a Shadow Function. ⓑ When the provisioning process completes, ⓒ the *Coordinator* updates the function status and ⓓ sends a registration success message to the corresponding paired functions so that they start to process inference requests collaboratively. As plotted in Figure 9b, when the Body Function receives an invocation and then finishes the computations on the non-parallel part, ⓐ it sends an offloading message to the *Coordinator* with the intermediate data to be processed in parallel. Meanwhile, the *Coordinator* ⓑ queries the registration table, obtains the ID of the paired Shadow Function, and ⓒ then forwards the intermediate data to the Shadow Function. Once the processing is complete, ⓓ the Shadow Function sends a completion message to the *Coordinator* with the result data. ❸ After obtaining the result data from both the *Coordinator* and local processing, the Body Function merges them and continues to perform further computations.

6 IMPLEMENTATION

We implement a real system prototype for AsyFunc with about 3k lines of code in Python and C++. The real system is implemented on top of an existing production-ready container orchestration system, Kubernetes [24]. The main extension lies in arming existing serverless inference platforms with both coarse-grained model-level scaling and fine-grained layer-level scaling capability. Specifically, we implement the *API Gateway*, *Scaler*, and *Scheduler* based on the Python client library for Kubernetes [25], with over 1k lines of Python code. These modules implement the following functions including collecting and preprocessing user requests, and making scaling and scheduling decisions. As the *Coordinator* needs to use some Linux libraries (e.g., the `mmap` library to realize efficient communication between two function instances), we implement the *Coordinator* with nearly 500 lines of both Python and C++ code. We detail the implementation of the *Coordinator* as follows.

The *Coordinator* provides caching and communication capabilities to implement the coordination logic presented in Section 5.2. The cache is reflected by a directory shared by all instances on the same node, and communication is performed by reading and writing files in this directory. To minimize the communication latency, the cache is implemented using the memory filesystem [51] and mapped to the memory area of each instance using the Linux kernel's `mmap` system call [39]. At runtime, the *Coordinator* monitors file system events in the cache using the inotify [33] API in the Linux kernel. The file system events include creating or destroying instances and transferring data between instances. Specifically, (1) When a new instance is provisioned, it writes its metadata as a new file in the cache. Then, once the *Coordinator* identifies this event, it reads the metadata of the instance from the file, registers the instance in its registration table with the metadata, and finally deletes the file created by the instance; (2) When a Body Function instance wants to call a Shadow Function instance for offloading computations, it writes the offloaded data as a new file in the cache, and the file is differentiated by the instance's ID. Then, the *Coordinator* uses the ID to find the corresponding ID of the paired Shadow Function in the registration table and instructs the Shadow Function to read the data. After reading the data, the Shadow Function is responsible for deleting the file. Finally, the Shadow Function returns the inference result to the Body Function in the same way.

Note that each instance starts a TCP service to receive data notifications from the *Coordinator*. In this way, only the file's name is transmitted over TCP, while the content of the file (i.e., the intermediate data during an inference execution) is transferred directly using the `mmap` technology. According to our experiments, it takes only about 0.5 ms to transfer 1 MB of the data.

In addition, we implement the *Extractor* and *Profiler* to obtain model metadata from our benchmark models (as shown later in Table 2) offline. Specifically, we developed a toolkit with 600 lines of Python code for extracting and profiling the layers of the model, which is embedded as a Python package in our project. The toolkit is easy to use and requires little change to the code when profiling a new model. In particular, developers can simply add two lines of code to automatically analyze the model.

7 EVALUATION

In this section, we investigate the performance of AsyFunc with extensive experiments using real-world traces. We first introduce the experimental setup and then show the experimental results compared to the state-of-the-art.

7.1 Experimental Setup

Environment configuration. We deploy AsyFunc in a local private cluster to provide inference services, and the specification of each node can be found in Table 1. To accelerate the experiments, we switch our system to the emulation mode as previous work does [29], where the inference latency under different configurations and the data transferring delay are collected in advance. We have verified the accuracy of the emulator in our local private cluster.

Table 1: Experimental environment.

Item	Specification
CPU Device	Intel Xeon Platinum 8269CY
Number of Sockets	1
Core(s) per Socket	26
Thread(s) per Core	2
Base Frequency	2.50 GHz
Memory Capacity	96 GB
Operating System	Ubuntu 18.04
DL Framework	PyTorch 1.12

System parameters. The maximum allowed CPU cores C_{max} and the maximum supported batch size B_{max} in an instance are set to 16 and 8, respectively. Further, the scaling parameters α and β are set to 0.8 and 0.6, respectively, based on our preliminary experiments, and the adaptation period of the model-level scaling is set as 10 s. Notably, we also measure the data transmission latency between instances under different data sizes as a reference for the following evaluation.

User requests. We use real-world traces from Twitter [54] to generate user requests in a sequence. It represents a typical arrival of tweets processed for sentiment analysis that has been widely used for inference serving elsewhere [2, 64]. Figure 1 shows the arrival intensity of a typical day in the first semester of 2017 which exhibits obvious burst characteristics.

Inference workloads. We select five representative DL models as the inference workloads, namely InceptionV3, EfficientNet-b5, YOLOv8x, SSD300, ResNet50, and VGG16. The benchmark models are built on PyTorch which is a popular machine learning framework with the dynamic computational graph support [40], and the model details are shown in Table 2. All their inference latency is measured on the server node. Based on the measurements, we choose the inference latency of the Body Function, when # of allocated cores is 16 (i.e., C_{max}) and the batch size is 8 (i.e., B_{max}), as the latency SLO for the following testing.

Table 2: Benchmark DL models.

Model	Parameter Size	# of Layers
InceptionV3 [52]	92 MB	193
EfficientNet-b5 [53]	117 MB	539
YOLOv8x [44]	131 MB	305
SSD300 [31]	136 MB	65
ResNet50 [21]	171 MB	245
VGG16 [50]	548 MB	45

Performance metrics and baselines. We use DL models shown in Table 2 to evaluate the performance of AsyFunc by comparing the following metrics including resource efficiency and the SLO violation rate with a state-of-the-art baseline BATCH [2]. For fairness of comparison, we integrate BATCH's scaling policy into our AsyFunc system, which determines the instance configuration offline and scales at a coarse-grained model level online. In contrast, AsyFunc combines coarse-grained model-level scaling policy with a fine-grained layer-level scaling policy to adaptively scale out/in instances. We also equip BATCH with the adaptive scheduling ability, and denote this baseline as BATCH^{+}. To show the benefit of our layer-level scaling, we disable the adaptive scheduling ability and denote this as AsyFunc^{-}.

7.2 Overall Performance

Memory resource efficiency. We first calculate the memory resource efficiency by counting the memory usage of all the provisioned instances in the cluster in each adaptation period. Figure 10 shows the violin plot of the memory footprint of different DL models under AsyFunc, AsyFunc^{-}, BATCH, and BATCH^{+} (note that the memory values are logarithmically spaced). The results reveal that AsyFunc-based systems (i.e., AsyFunc and AsyFunc^{-}) are more memory resource efficient on all six ML models with a lower memory footprint than BATCH-based systems (i.e., BATCH and BATCH^{+}). This is because, under AsyFunc-based systems, there are fewer active Body Function instances in the cluster simultaneously, and fewer instance replicas result in less memory consumption. We summarize two factors that contribute to this superiority. Firstly, due to a long adaptation period, BATCH-based systems are difficult to capture short-term variations in RPS through its conservative instance configuration selection strategy. On the contrary, AsyFunc-based systems can select instance configurations based on RPS. Thus, when the RPS increases/decreases, they tend to start a small number of instances equipped with more/fewer CPU cores. Secondly, given the same historical RPS data, the scaling policy of BATCH-based systems tends to pre-warm more instances for future requests than AsyFunc-based systems. This will also increase the memory footprint. The plot of the memory usage over time in Figure 11 also supports the above explanation. In summary, AsyFunc can reduce the memory footprint by up to 23.4%.

Computing resource efficiency. Then, we calculate the computing resource efficiency denoted as the CPU utility in each adaptation period. The CPU utility is defined as $\frac{\sum inference\_time * cpu\_cores}{\sum total\_time * cpu\_cores}$. As plotted in Figure 12, the CPU utility of AsyFunc is up to 96%, 12% higher than BATCH. The main reason is that there is a short

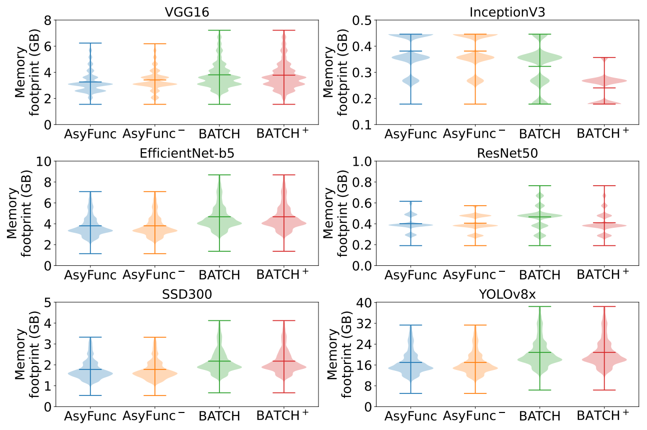


Figure 10: The violin plot of the memory usage.

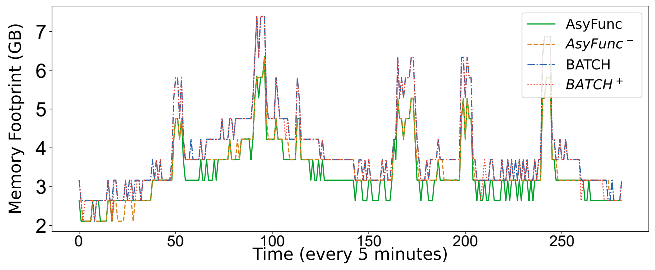


Figure 11: The pattern of the memory usage over time.

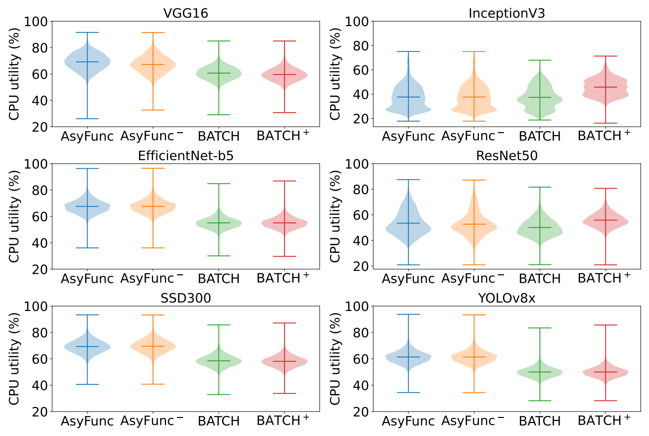


Figure 12: The violin plot of the CPU utility.

model-loading time of the Shadow Function that can be provisioned on demand and destroyed when the arrival intensity drops. It is worth mentioning that by comparing BATCH and BATCH^{+}, we find the adaptive scheduling can also improve the CPU utility to some extent. For example, for model InceptionV3, adaptive scheduling improves CPU utility by about 11%. The reason is that the adaptive scheduling can flexibly adjust the batch size of the instance during the inference serving to make the best use of CPU resources of instances as much as possible.

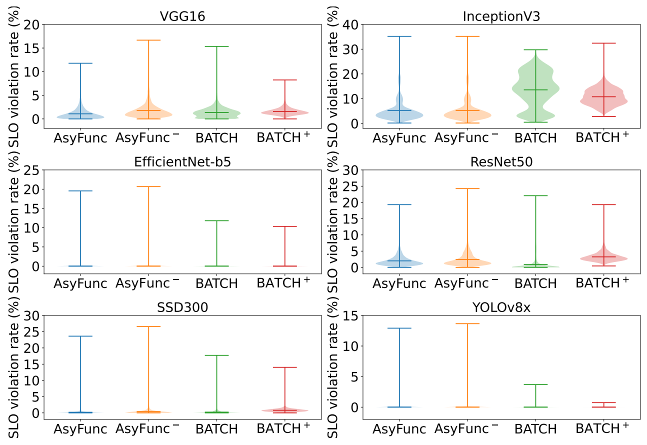


Figure 13: The violin plot of the SLO violation rate.

SLO violation rate. Finally, we calculate the SLO violation rate by counting the percentage of requests whose SLO constraint is not satisfied. Figure 13 shows the violin plot of the SLO violation rate of different DL models under four systems. As compared to BATCH, AsyFunc has no significant increase in the SLO violation rate and even performs better under some models, such as InceptionV3. This can be attributed to the lightweight layer-level scaling which can start a Shadow Function instance within a few milliseconds so that the Body Function instance can offload computations when there are insufficient resources. In contrast, without the layer-level scaling support, the SLO constraint rate gets higher under AsyFunc$^-$, especially for model VGG16, where the median SLO violation rate is 35% higher than that under AsyFunc. Besides, for some models, e.g., YOLOv8x, the adaptive scheduling scheme helps reduce the violation rate too.

The above experimental results verify the strengths of AsyFunc. In summary, AsyFunc can reduce the memory footprint by up to 23.4% while maintaining the SLO.

7.3 Extension of AsyFunc

Serving transformer-based models. According to our investigation, there may be no significant anti-correlation phenomenon in transformer-based models. Nevertheless, AsyFunc still outperforms the state-of-the-art thanks to our layer-level scaling support that achieve adaptive trade-off between resource consumption and performance guarantee by loading different percentage of layers. According to our experiments on the Vision Transformer (ViT) model, AsyFunc consumes about 8% fewer memory resources than BATCH with SLO guarantee.

Extension to GPU support. It is very common to use heterogeneous hardware, such as GPUs, for inference executions. As major serverless platforms do not support heterogeneous hardware at the moment, we do not include advanced heterogeneous hardware in our current version. Nervertheless, it complements AsyFunc's fine-grained scaling capabilities at the layer level. Here, we briefly illustrate how AsyFunc can be equipped with the GPU support. First, in the scaling algorithm, the GPU resources can be represented as the number of SM (vs. # of CPU cores) and/or the GPU memory usage (vs. the host memory) with virtualization technologies like virtual GPU and Multi-Instance GPU. Second, for the implementation, the CPU functions can be replaced by GPU-supported ones, such as *nvidia-docker*. There are indeed some technical issues to overcome, such as efficient coordination between CPU instances and GPU instances and the logic of data exchange between the Body Function and Shadow Function. It would be an interesting future work to extend our approach to these scenarios in practice.

8 RELATED WORK

In this section, we briefly review some related work in both academia and industry.

Conventional model serving. Many efforts have been devoted to designing efficient scheduling mechanisms for model serving to achieve various objectives, e.g., low end-to-end latency [11, 18, 29, 60], high throughput [15, 20, 29], high resource efficiency [32, 58], and good fairness [26]. However, they only focus on the application layer, without digging into the bottom inference platforms, leaving a large performance gap to be filled. Additionally, facing the widespread burstiness in the production environment, these prior arts cannot adapt to the fluctuating workload efficiently, leading to a significant trade-off between performance and resource efficiency. Although AlpaServe [29] employs statistical multiplexing with model parallelism to reduce serving latency for bursty workloads, the biggest difference is that it focuses on static provisioning through automatic parallelization and placement of models, ignoring the auto-scaling capability that serverless native supports.

To meet latency requirements of model serving, especially with the recent advent of large language models such as ChatGPT [9], there has been a great deal of work on inference optimization. These include techniques such as quantization [13], knowledge distillation [22], and model pruning [19], with some of these optimizations aimed at stemming the tide of growing model sizes. However, these offerings are complementary to AsyFunc in that we provide fine-grained layer-level scaling capability that enables fast response to significant and unpredictable fluctuations of workload levels.

Serverless inference. Driven by the development of serverless computing, many works [2, 6, 27, 62–64] have attempted to deploy efficient machine learning inference serving on serverless platforms to make full use of its rapid elasticity and fine-grained billing ability. Nevertheless, although these techniques consider an auto-scaling setting, most of them regard the DL model as a complete black box [2, 6, 62, 64], which leads to resource inefficiency at a coarse-grained model-level scaling when directly applied for serverless inference serving. Recent literature has further attempted to open the box to reduce resource footprint during scaling through tensor sharing [27], but they still lack generalizability in dealing with a large model family as they only focus on the rare layer sameness. By contrast, AsyFunc fully exploits the differences between the widespread model layers to be more general applicability. Gillis [63] is another work to open the box, which partitions large DL models so that they can fit small functions. By comparison, AsyFunc focuses on the scaling problem of serverless platforms and only scales out resource-sensitive layers to perform additional computations as bursts arise.

Serverless cold start. Serverless cold start optimization has been an active research topic in recent years [1, 5, 28, 38, 47–49, 55, 56]. These works typically adopt two different technical paths, including (1) avoiding cold starts based on predictive prewarming techniques [5, 47] and container keep-alive strategies [38, 48], and (2) reducing the latency of a single cold start based on snapshots [55, 56] and lightweight runtime techniques [1, 28, 49]. Taking industrial practices as an example, Azure deploys a practical hybrid histogram policy by characterizing the serverless workloads to dynamically decide the values of the prewarming and keep-alive window, which significantly reduces the number of cold starts while spending fewer resources [48]. However, these works focus on OS-level cold starts, while AsyFunc deals with application-level cold starts of inference services. They are complementary to each other to further improve the resource efficiency of real-time serverless inference serving by cloud providers.

9 CONCLUSION

In this paper, we propose a high-performance and resource-efficient serverless inference serving system called AsyFunc in the presence of bursty workloads. By analyzing the measurement results on the impact of DL models' completeness, we find that the model layer's sensitivity to computational resources is largely anti-correlated with its parameter size, and the latter further determines the memory resource usage and model-loading time. Driven by this, we propose a new concept of *asymmetric functions* where the original Body Function still loads a complete model to satisfy stable demands, while the proposed lightweight Shadow Function loads only a portion of resource-sensitive layers to handle surging demands effortlessly. On top of Kubernetes, AsyFunc is equipped with a new fine-grained auto-scaling and scheduling engine to achieve the above goals. The experimental results using real-world traces show that AsyFunc outperforms the state-of-the-art system by 23.4% in resource efficiency while meeting the latency SLO of inference services during bursts.

REFERENCES

[1] Alexandru Agache, Marc Brooker, Alexandra Iordache, Anthony Liguori, Rolf Neugebauer, Phil Piwonka, and Diana-Maria Popa. 2020. Firecracker: Lightweight virtualization for serverless applications. In *17th USENIX symposium on networked systems design and implementation (NSDI 20)*. 419–434.
[2] Ahsan Ali, Riccardo Pinciroli, Feng Yan, and Evgenia Smirni. 2020. BATCH: Machine learning inference serving on serverless platforms with adaptive batching. In *SC20: International Conference for High Performance Computing, Networking, Storage and Analysis*. IEEE, 1–15.
[3] AWS. [n. d.]. Alexa skills. https://docs.aws.amazon.com/wellarchitected/latest/serverless-applications-lens/alexa-skills.html[Online Accessed, 8-June-2023].
[4] AWS. [n. d.]. Amazon SageMaker. https://aws.amazon.com/sagemaker/[Online Accessed, 8-June-2023].
[5] Vivek M Bhasi, Jashwant Raj Gunasekaran, Prashanth Thinakaran, Cyan Subhra Mishra, Mahmut Taylan Kandemir, and Chita Das. 2021. Kraken: Adaptive container provisioning for deploying dynamic dags in serverless platforms. In *Proceedings of the ACM Symposium on Cloud Computing*. 153–167.
[6] Anirban Bhattacharjee, Ajay Dev Chhokra, Zhuangwei Kang, Hongyang Sun, Aniruddha Gokhale, and Gabor Karsai. 2019. BARISTA: Efficient and scalable serverless serving system for deep learning prediction services. In *2019 IEEE International Conference on Cloud Engineering (IC2E)*. IEEE, 23–33.
[7] Tom Brown, Benjamin Mann, Nick Ryder, Melanie Subbiah, Jared D Kaplan, Prafulla Dhariwal, Arvind Neelakantan, Pranav Shyam, Girish Sastry, Amanda Askell, Sandhini Agarwal, Ariel Herbert-Voss, Gretchen Krueger, Tom Henighan, Rewon Child, Aditya Ramesh, Daniel Ziegler, Jeffrey Wu, Clemens Winter, Chris Hesse, Mark Chen, Eric Sigler, Mateusz Litwin, Scott Gray, Benjamin Chess, Jack Clark, Christopher Berner, Sam McCandlish, Alec Radford, Ilya Sutskever, and Dario Amodei. 2020. Language models are few-shot learners. *Advances in neural information processing systems* 33 (2020), 1877–1901.
[8] Brad Calder, Glenn Reinman, and Dean M Tullsen. 1999. Selective value prediction. In *Proceedings of the 26th annual international symposium on computer architecture*. 64–74.
[9] ChatGPT. 2022. *Introducing ChatGPT*. Retrieved May 25, 2023 from https://openai.com/blog/chatgpt
[10] Daniel Crankshaw, Xin Wang, Guilio Zhou, Michael J Franklin, Joseph E Gonzalez, and Ion Stoica. 2017. Clipper: A Low-Latency Online Prediction Serving System. In *14th USENIX Symposium on Networked Systems Design and Implementation (NSDI 17)*. 613–627.
[11] Weihao Cui, Han Zhao, Quan Chen, Hao Wei, Zirui Li, Deze Zeng, Chao Li, and Minyi Guo. 2022. DVABatch: Diversity-aware Multi-Entry Multi-Exit Batching for Efficient Processing of DNN Services on GPUs. In *2022 USENIX Annual Technical Conference (USENIX ATC 22)*. 183–198.
[12] Nilanjan Daw, Umesh Bellur, and Purushottam Kulkarni. 2020. Xanadu: Mitigating cascading cold starts in serverless function chain deployments. In *Proceedings of the 21st International Middleware Conference*. 356–370.
[13] Tim Dettmers, Mike Lewis, Younes Belkada, and Luke Zettlemoyer. 2022. GPT3. int8 (): 8-bit matrix multiplication for transformers at scale. *Advances in Neural Information Processing Systems* 35 (2022), 30318–30332.
[14] Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. 2018. Bert: Pre-training of deep bidirectional transformers for language understanding. *arXiv preprint arXiv:1810.04805* (2018).
[15] Aditya Dhakal, Sameer G Kulkarni, and KK Ramakrishnan. 2020. GSLICE: Controlled Spatial Sharing of GPUs for a Scalable Inference Platform. In *Proceedings of the 11th ACM Symposium on Cloud Computing*. 492–506.
[16] Dong Du, Tianyi Yu, Yubin Xia, Binyu Zang, Guanglu Yan, Chenggang Qin, Qixuan Wu, and Haibo Chen. 2020. Catalyzer: Sub-millisecond startup for serverless computing with initialization-less booting. In *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*. 467–481.
[17] Arpan Gujarati, Sameh Elnikety, Yuxiong He, Kathryn S McKinley, and Björn B Brandenburg. 2017. Swayam: distributed autoscaling to meet SLAs of machine learning inference services with resource efficiency. In *Proceedings of the 18th ACM/IFIP/USENIX middleware conference*. 109–120.
[18] Arpan Gujarati, Reza Karimi, Safya Alzayat, Wei Hao, Antoine Kaufmann, Ymir Vigfusson, and Jonathan Mace. 2020. Serving DNNs like Clockwork: Performance Predictability from the Bottom Up. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*. 443–462.
[19] Song Han, Jeff Pool, John Tran, and William Dally. 2015. Learning both weights and connections for efficient neural network. *Advances in neural information processing systems* 28 (2015).
[20] Johann Hauswald, Yiping Kang, Michael A Laurenzano, Quan Chen, Cheng Li, Trevor Mudge, Ronald G Dreslinski, Jason Mars, and Lingjia Tang. 2015. DjiNN and Tonic: DNN as a Service and Its Implications for Future Warehouse Scale Computers. In *2015 ACM/IEEE 42nd Annual International Symposium on Computer Architecture (ISCA)*. IEEE, 27–40.
[21] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. 2016. Deep Residual Learning for Image Recognition. In *2016 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*. 770–778. https://doi.org/10.1109/CVPR.2016.90
[22] Geoffrey Hinton, Oriol Vinyals, and Jeff Dean. 2015. Distilling the knowledge in a neural network. *arXiv preprint arXiv:1503.02531* (2015).
[23] Oleksiy Kovyrin. [n. d.]. Make Data Useful by Greg Linden. https://www.scribd.com/doc/4970486/[Online Accessed, 8-June-2023].
[24] kubernetes. 2023. *Production-Grade Container Orchestration*. Retrieved June 8, 2023 from https://kubernetes.io/
[25] Kubernetes Python Client. 2023. *Official Python client library for kubernetes*. Retrieved June 8, 2023 from https://github.com/kubernetes-client/python
[26] Tan N Le, Xiao Sun, Mosharaf Chowdhury, and Zhenhua Liu. 2020. AlloX: compute allocation in hybrid clusters. In *Proceedings of the Fifteenth European Conference on Computer Systems*. 1–16.
[27] Jie Li, Laiping Zhao, Yanan Yang, Kunlin Zhan, and Keqiu Li. 2022. Tetris: Memory-efficient Serverless Inference through Tensor Sharing. In *2022 USENIX Annual Technical Conference (USENIX ATC 22)*.
[28] Zijun Li, Jiagan Cheng, Quan Chen, Eryu Guan, Zizheng Bian, Yi Tao, Bin Zha, Qiang Wang, Weidong Han, and Minyi Guo. 2022. RunD: A Lightweight Secure Container Runtime for High-density Deployment and High-concurrency Startup in Serverless Computing. In *2022 USENIX Annual Technical Conference (USENIX ATC 22)*. 53–68.
[29] Zhuohan Li, Lianmin Zheng, Yinmin Zhong, Vincent Liu, Ying Sheng, Xin Jin, Yanping Huang, Zhifeng Chen, Hao Zhang, Joseph E Gonzalez, and Ion Stoica. 2023. AlpaServe: Statistical Multiplexing with Model Parallelism for Deep Learning Serving. In *17th USENIX Symposium on Operating Systems Design and Implementation (OSDI 23)*. 663–679.
[30] Yanying Lin, Kejiang Ye, Yongkang Li, Peng Lin, Yingfei Tang, and Chengzhong Xu. 2021. BBServerless: A Bursty Traffic Benchmark for Serverless. In *International Conference on Cloud Computing*. Springer, 45–60.

[31] Wei Liu, Dragomir Anguelov, Dumitru Erhan, Christian Szegedy, Scott Reed, Cheng-Yang Fu, and Alexander C Berg. 2016. Ssd: Single shot multibox detector. In *European conference on computer vision*. Springer, 21–37.

[32] Zihan Liu, Jingwen Leng, Zhihui Zhang, Quan Chen, Chao Li, and Minyi Guo. 2022. VELTAIR: towards high-performance multi-tenant deep learning services via adaptive compilation and scheduling. In *Proceedings of the 27th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*. 388–401.

[33] Robert Love. 2005. Kernel korner: Intro to inotify. *Linux Journal* 2005, 139 (2005), 8.

[34] Peter Mattson, Vijay Janapa Reddi, Christine Cheng, Cody Coleman, Greg Diamos, David Kanter, Paulius Micikevicius, David Patterson, Guenther Schmuelling, Hanlin Tang, Gu-Yeon Wei, and Carole-Jean Wu. 2020. MLPerf: An Industry Standard Benchmark Suite for Machine Learning Performance. *IEEE Micro* 40, 2 (2020), 8–16.

[35] Dirk Merkel. 2014. Docker: Lightweight Linux Containers for Consistent Development and Deployment. *Linux J.* 2014, 239, Article 2 (mar 2014).

[36] Hai Duc Nguyen, Chaojie Zhang, Zhujun Xiao, and Andrew A Chien. 2019. Real-time serverless: Enabling application performance guarantees. In *Proceedings of the 5th International Workshop on Serverless Computing*. 1–6.

[37] Edward Oakes, Leon Yang, Dennis Zhou, Kevin Houck, Tyler Harter, Andrea Arpaci-Dusseau, and Remzi Arpaci-Dusseau. 2018. SOCK: Rapid task provisioning with Serverless-Optimized containers. In *2018 USENIX Annual Technical Conference (USENIX ATC 18)*. 57–70.

[38] Li Pan, Lin Wang, Shutong Chen, and Fangming Liu. 2022. Retention-aware container caching for serverless edge computing. *Proc. of IEEE INFOCOM, IEEE* (2022).

[39] Anastasios Papagiannis, Giorgos Xanthakis, Giorgos Saloustros, Manolis Marazakis, and Angelos Bilas. 2020. Optimizing Memory-mapped I/O for Fast Storage Devices. In *2020 USENIX Annual Technical Conference (USENIX ATC 20)*. 813–827.

[40] Adam Paszke, Sam Gross, Francisco Massa, Adam Lerer, James Bradbury, Gregory Chanan, Trevor Killeen, Zeming Lin, Natalia Gimelshein, Luca Antiga, Alban Desmaison, Andreas Kopf, Edward Yang, Zachary DeVito, Martin Raison, Alykhan Tejani, Sasank Chilamkurthy, Benoit Steiner, Lu Fang, Junjie Bai, and Soumith Chintala. 2019. PyTorch: An Imperative Style, High-Performance Deep Learning Library. In *Advances in Neural Information Processing Systems*, H. Wallach, H. Larochelle, A. Beygelzimer, F. d'Alché-Buc, E. Fox, and R. Garnett (Eds.), Vol. 32. Curran Associates, Inc. https://proceedings.neurips.cc/paper_files/paper/2019/file/bdbca288fee7f92f2bfa9f7012727740-Paper.pdf

[41] Qiangyu Pei, Shutong Chen, Qixia Zhang, Xinhui Zhu, Fangming Liu, Ziyang Jia, Yishuo Wang, and Yongjie Yuan. 2022. CoolEdge: hotspot-relievable warm water cooling for energy-efficient edge datacenters. In *Proceedings of the 27th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*. 814–829.

[42] Pytorch. [n. d.]. SAVING AND LOADING MODELS. https://pytorch.org/tutorials/beginner/saving_loading_models.html[Online Accessed, 8-June-2023].

[43] Aurick Qiao, Sang Keun Choe, Suhas Jayaram Subramanya, Willie Neiswanger, Qirong Ho, Hao Zhang, Gregory R Ganger, and Eric P Xing. 2021. Pollux: Co-adaptive cluster scheduling for goodput-optimized deep learning. In *15th USENIX Symposium on Operating Systems Design and Implementation (OSDI 21)*.

[44] Dillon Reis, Jordan Kupec, Jacqueline Hong, and Ahmad Daoudi. 2023. Real-Time Flying Object Detection with YOLOv8. *arXiv preprint arXiv:2305.09972* (2023).

[45] Mariliis Retter. [n. d.]. Serverless Case Study – Netflix. https://dashbird.io/blog/serverless-case-study-netflix/[Online Accessed, 8-June-2023].

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

[47] Rohan Basu Roy, Tirthak Patel, and Devesh Tiwari. 2022. IceBreaker: warming serverless functions better with heterogeneity. In *Proceedings of the 27th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*. 753–767.

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

[49] Simon Shillaker and Peter Pietzuch. 2020. Faasm: Lightweight isolation for efficient stateful serverless computing. In *2020 USENIX Annual Technical Conference (USENIX ATC 20)*. 419–433.

[50] Karen Simonyan and Andrew Zisserman. 2014. Very deep convolutional networks for large-scale image recognition. *arXiv preprint arXiv:1409.1556* (2014).

[51] Peter Snyder. 1990. tmpfs: A virtual memory file system. In *Proceedings of the autumn 1990 EUUG Conference*. 241–248.

[52] Christian Szegedy, Vincent Vanhoucke, Sergey Ioffe, Jon Shlens, and Zbigniew Wojna. 2016. Rethinking the Inception Architecture for Computer Vision. In *2016 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*. 2818–2826. https://doi.org/10.1109/CVPR.2016.308

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

[54] Twitter. [n. d.]. Twitter trace on May 25th. https://github.com/rickypinci/BATCH/tree/sc2020/traces[Online Accessed, 8-June-2023].

[55] Dmitrii Ustiugov, Plamen Petrov, Marios Kogias, Edouard Bugnion, and Boris Grot. 2021. Benchmarking, analysis, and optimization of serverless function snapshots. In *Proceedings of the 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*. 559–572.

[56] Kai-Ting Amy Wang, Rayson Ho, and Peng Wu. 2019. Replayable execution optimized for page sharing for a managed runtime environment. In *Proceedings of the Fourteenth EuroSys Conference 2019*. 1–16.

[57] Qizhen Weng, Wencong Xiao, Yinghao Yu, Wei Wang, Cheng Wang, Jian He, Yong Li, Liping Zhang, Wei Lin, and Yu Ding. 2022. MLaaS in the Wild: Workload Analysis and Scheduling in Large-Scale Heterogeneous GPU Clusters. In *19th USENIX Symposium on Networked Systems Design and Implementation (NSDI 22)*. 945–960.

[58] Jing Wu, Lin Wang, Qiangyu Pei, Xingqi Cui, Fangming Liu, and Tingting Yang. 2022. HiTDL: High-throughput deep learning inference at the hybrid mobile edge. *IEEE Transactions on Parallel and Distributed Systems* 33, 12 (2022), 4499–4514.

[59] Yuncheng Wu, Tien Tuan Anh Dinh, Guoyu Hu, Meihui Zhang, Yeow Meng Chee, and Beng Chin Ooi. 2021. Serverless Data Science-Are We There Yet? A Case Study of Model Serving. *arXiv e-prints* (2021), arXiv–2103.

[60] Yecheng Xiang and Hyoseung Kim. 2019. Pipelined data-parallel CPU/GPU scheduling for multi-DNN real-time inference. In *2019 IEEE Real-Time Systems Symposium (RTSS)*. IEEE, 392–405.

[61] Mengwei Xu, Zhe Fu, Xiao Ma, Li Zhang, Yanan Li, Feng Qian, Shangguang Wang, Ke Li, Jingyu Yang, and Xuanzhe Liu. 2021. From cloud to edge: a first look at public edge platforms. In *Proceedings of the 21st ACM Internet Measurement Conference*. 37–53.

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

[63] Minchen Yu, Zhifeng Jiang, Hok Chun Ng, Wei Wang, Ruichuan Chen, and Bo Li. 2021. Gillis: Serving large neural networks in serverless functions with automatic model partitioning. In *2021 IEEE 41st International Conference on Distributed Computing Systems (ICDCS)*. IEEE, 138–148.

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