Sponge: Inference Serving with Dynamic SLOs Using In-Place Vertical Scaling

Kamran Razavi*
Technical University of Darmstadt

Saeid Ghafouri*
Queen Mary University of London

Max Mühlhäuser
Technical University of Darmstadt

Pooyan Jamshidi
University of South Carolina

Lin Wang
Paderborn University

ABSTRACT

Mobile and IoT applications increasingly adopt deep learning inference to provide intelligence. Inference requests are typically sent to a cloud infrastructure over a wireless network that is highly variable, leading to the challenge of dynamic Service Level Objectives (SLOs) at the request level.

This paper presents Sponge, a novel deep learning inference serving system that maximizes resource efficiency while guaranteeing dynamic SLOs. Sponge achieves its goal by applying in-place vertical scaling, dynamic batching, and request reordering. Specifically, we introduce an Integer Programming formulation to capture the resource allocation problem, providing a mathematical model of the relationship between latency, batch size, and resources. We demonstrate the potential of Sponge through a prototype implementation and preliminary experiments and discuss future works.

CCS CONCEPTS

• **Computer systems organization** → **Cloud computing**; **Reconfigurable computing**;

KEYWORDS

Inference Serving Systems, Vertical Scaling

1 INTRODUCTION

Within the domain of mobile and IoT applications, cloud-based Deep Learning (DL) inference plays an important role, with user satisfaction and resource efficiency serving as key performance indicators [1, 25, 26]. Since most DL-powered applications involve user interaction, they must comply with strict requirements on the inference latency, a.k.a. meeting the Service Level Objectives (SLOs) of the inference request. On the other hand, the resources needed to provision such a DL inference serving system should be minimized to reduce the cost [10, 17, 18, 23, 28, 31].

SLOs are comprehensively defined from end to end, with the variable network time required for transferring user requests and input data introducing dynamic time budgets for serving inference requests. Therefore, when setting expectations for mobile and IoT applications, it is important to define SLOs that cover both the network and computing aspects from start to finish. Ignoring the time it takes for information to travel through the network, inference serving systems may find themselves with not enough time to handle requests properly, resulting in SLO violation. Hence, resource allocation must consider a variety of time budgets of a single user using the same application. Managing this dynamism poses a critical challenge for inference serving systems, where the effective handling of diverse SLOs and the consideration of fluctuating network conditions are imperative to ensure the fulfillment of end-to-end SLOs.

Existing inference serving systems mostly consider only the inference part with static SLOs, i.e., all requests have the same SLOs when they reach computing units. Their horizontal scaling-based approach cannot incorporate diverse SLOs at the request level [10, 12, 18]. For example, FA2 [28] adjusts the number of minimum-resource instances to achieve the highest resource efficiency (throughput). Moreover, bringing new instances in horizontal scaling ties with the cold-start issue (a few seconds [15, 29]), which cannot cope with the dynamically changing network conditions. Jellyfish [27], on the other hand, aims to guarantee end-to-end SLOs while achieving high inference accuracy by using preloaded model-switching and trading accuracy for latency, which may not always be possible for all applications.

We propose a new system, Sponge, aiming to address this research gap. Our main insight is that the combination of in-place vertical scaling, dynamic batching, and request reordering is a powerful tool to combat request-level dynamism. In

*Equal Contribution

EuroMLSys '24, April 22, 2024, Athens, Greece

ACM ISBN 979-8-4007-0541-0/24/04…$15.00
https://doi.org/10.1145/3642970.3655833

particular, the new in-place vertical scaling feature of Kubernetes [3] allows developers to resize CPU/memory resources allocated to containers without restarting them, eliminating the cold-start issues of vertical scaling, while request reordering allows for requests with a lower remaining time budget to be processed earlier. At the same time, dynamic batching increases the system utilization to further reduce the needed computing resources. We formulate the problem and propose a method for inference serving with dynamic SLOs. Sponge relies on three adaptation strategies to capture per-request dynamic SLOs: ❶ in-place vertical scaling to change the computing resources of DL models in spot, ❷ request reordering to prioritize close-to-deadline requests, and ❸ dynamic batching to increase the utilization of the DL models. More specifically, Sponge achieves dynamic SLOs guarantee and high resource utilization by first providing a mathematical relation between vertical scaling with batching and processing latency of the DL model using historical data and then designing a request-based mathematical modeling of the entire framework to guarantee SLOs of all requests while minimizing the resources. Furthermore, we propose a simple algorithm for small cases to iterate over all possible configurations and find optimal resource and batch size allocations. The preliminary experimental results show a reduction in over 15× of the SLO violation compared to the existing approaches.

Sponge currently does not consider pipelines of DL models. Complex applications such as intelligent virtual assistants consist of multiple DL models, coordinated with a Directed Acyclic Graph (DAG), collaboratively generating a meaningful output. Such applications require a more intricate solution due to data dependencies among DL models, resulting in a strong coupling of scaling decisions for different DL models. Furthermore, vertical scaling sustains workloads to some extent due to the DL model parallelization level and availability of computing resources in a sine node. Therefore, multiple instances of the same DL model (horizontal scaling) may need to reside in different computing nodes to support the incoming workload. We consider these directions as future works of Sponge.

This paper contributes by discussing the challenges of dynamic SLOs on DL inference serving systems. Then, we

- present the design of Sponge, a new DL inference serving system for dynamic SLOs based on the idea of in-place vertical scaling, request reordering, and dynamic batching.
- provide an Integer Programming formulation to encapsulate the problem of dynamic SLOs by introducing a mathematical modeling of the relation between latency, batch, and CPU in inference serving systems.

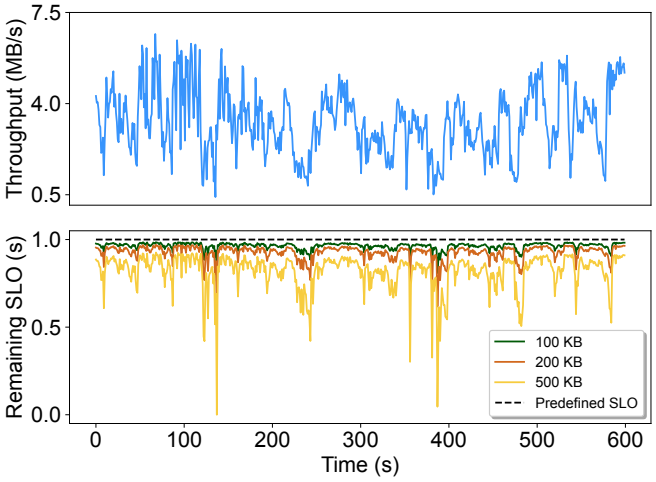

Figure 1: Bandwidth measurements in 4G networks provided by [34]. The bandwidth varies from 0.5MB/s to 7MB/s in a 10-minute range (top figure). The below figure demonstrates the remaining SLO for processing when the user sends a 100 KB, a 200 KB, or a 500 KB image over the same network's bandwidth.

- build a prototype system for Sponge [1] and evaluate it using 4G/LTE bandwidth logs datasets. Sponge reduces the SLO violation by over 15× compared to a horizontal state-of-the-art autoscaler.

2 MOTIVATION

In this section, we first discuss the challenges raised by variable networks and then identify the challenges in efficient in-place vertical scaling.

2.1 Dynamic SLO

Fluctuations in network bandwidths, e.g., caused by user mobility, are inevitable [8, 14], as illustrated in Figure 1 (top). This variability influences the transmission overhead associated with sending data across the network for remote processing, leading to a reduction in the time budget available for server-side deployed services, as depicted in Figure 1 (bottom). Consequently, service providers are compelled to account for network latency to ensure compliance with the end-to-end latency requirements specified in the SLO.

We use a simple human detection model trained on the ResNet architecture to motivate this work, where it detects a human in an image of 200 KB while the requests are being sent on a dynamically changing network (e.g., 4G) under a static workload of 100 requests per second with the SLO of 1000 ms (similar conditions to Figure 1). Table 1 shows the execution latency of the model with different allocated CPU

[1] https://github.com/saeid93/sponge

Table 1: Execution latency (P99) of a ResNet model (human detector) with different CPU cores using different batch sizes while guaranteeing SLO of 1000 ms under the workload of 100 RPS.

Cores	Batch	Latency (ms)	Throughput (RPS)	Total Cores
1	1	55	$18 \times 6 = 108$	$1 \times 6 = 6$
1	2	97	$20 \times 5 = 100$	$1 \times 5 = 5$
2	4	94	$40 \times 3 = 120$	$2 \times 3 = 6$
4	8	92	$80 \times 2 = 160$	$2 \times 4 = 8$
8	4	37	$108 \times 1 = 108$	$1 \times 8 = 8$
8	8	62	$128 \times 1 = 128$	$1 \times 8 = 8$

cores and batch sizes when considering the SLO. For calculating the required number of instances per instance type (different core numbers), we divide the incoming workload by the throughput of an instance. Following the approach in FA2 [28], where they use one-core instances, we need five instances to process a batch of 2 requests per 97 ms, which means that we can process a batch of 10 requests, or 20 requests per second (RPS), over a 1000 ms SLO. This approach works perfectly if the network is static. However, if the network latency takes up to half of the SLO, FA2 will drop all the requests, as there is no possible solution in their approach with one-core instances, even with the smallest batch size. Furthermore, even if the network latency takes just 40 ms, the system needs to bring up a new instance to avoid dropping requests or violating the SLO, meaning that the system will suffer from the cold start of a new instance until the system stabilizes again. Alternatively, meeting the SLO in the context of a dynamically changing network bandwidth could have been achieved through the dynamic modification of computing resources within the instance (in-place vertical scaling). In the same scenario, if we had up to 600 ms of network delay, we could still serve the requests without violating or dropping any request by changing the instance core from 1 core to 8 cores with a batch size of 4. InfAdapter [31] employs profiling data to determine CPU core allocation for DL models. For instance, under a workload of 100 RPS, the model's computing resources and batch size remain static. However, when faced with changes in the SLO, it switches to a different model variant with predefined CPU core allocation, encountering similar challenges as FA2 (cold start and static CPU core allocation).

2.2 Autoscaling Challenges

Creating an effective in-place vertical scaling system for DL inference serving is a complex task. Precisely, we pinpoint the following challenges, which collectively differentiate the scaling problem in DL inference serving systems from those examined in other systems.

Dynamic SLO at the request level. In wireless networks conditions can change over time. This can be due to various factors, such as changes in network traffic, hardware performance, signal strength, and resource availability [22, 36]. These factors can cause variable delays in network transmission for inference requests, leading to requests with dynamic SLOs. Accommodating dynamic SLOs at the request level requires fine-grained control over resource allocation to ensure each request meets its SLO. This level of granularity is challenging to achieve with vertical scaling since changing the resources to guarantee one request SLO affects all the requests' processing latency in the system.

Batch size. DL inference serving systems commonly utilize request batching to enhance resource efficiency [9–11, 32]. More precisely, batching can increase throughput as more tasks or requests can be processed in a given amount of time. Furthermore, batching can help meet latency constraints with dynamic batching policies, where batch sizes are determined online, during runtime, depending on the latency constraints of each application [23, 28]. However, it is important to note that large and small batch sizes can have drawbacks if not properly managed. Large batch sizes can critically violate the latency of many requests within a batch, while small batch sizes could cause excessive queuing and may not exploit potential opportunities for increased throughput.

In the next section, we provide an in-place vertical-based autoscaler to capture the discussed challenges by first discussing how to reconcile vertical scaling and batch size in the context of inference serving systems, and second, providing a mathematical formulation to mimic the autoscaling problem with the consideration of dynamic SLOs.

3 SYSTEM DESIGN

This section provides our solution for inference serving systems with dynamic SLOs. Our goal is to use minimal resources to provision the DL model with in-place vertical scaling, request reordering, and dynamic batch sizing while guaranteeing all the requests' SLO.

3.1 Overview

Sponge consists of four components as is shown in Figure 2:
Monitoring. The monitoring component uses Prometheus [7] to observe the incoming workload to the system. It will monitor the workload destined for the model on a predefined time interval. Additionally, it receives the end-to-end request latency from the processing component to calculate the SLO violation rate and the accuracy of the performance model.
Queuing. The queuing component receives the request from the user, reorders the request based on the remaining SLO (Earliest Deadline First (EDF)), and creates a batch with the

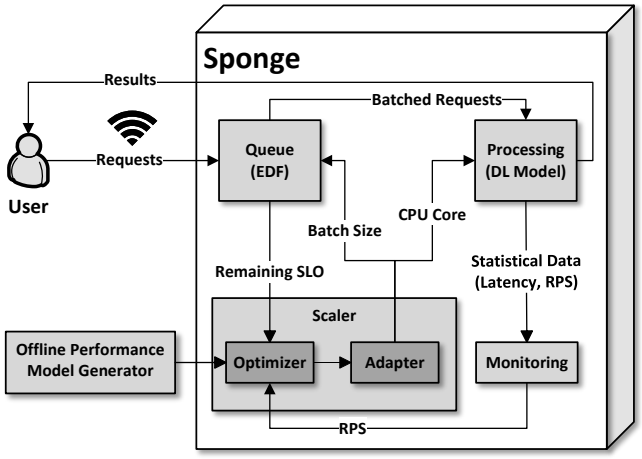

Figure 2: An overview of the Sponge architecture. The monitoring service collects metric data from the DL model. The queue prioritizes requests according to the EDF policy. The scaler is responsible for determining vertical scaling and batch size decisions for the DL model and adjusting the system accordingly.

given batch size from the solver. In addition, it sends the set of requests with their communication latency to the optimizer.

Processing. The processing component has the computing power to execute inferences. It receives batches from the queue, processes them, and sends them to the user. Furthermore, it sends the statistical data (queuing latency and processing latency) to the monitoring component.

Scaler. The scaler component first aims to find the vertical scaling CPU cores and batch size decisions to achieve the highest resource efficiency while respecting all the request SLOs in the system by using the workload (reported by the monitoring component) and the remaining SLOs of all the requests after being reordered by the queuing component in the optimizer. Next, its adapter part adjusts the system by sending a signal to the processing component with the new CPU core allocation and a signal to the queueing component with the new batch size configuration.

3.2 Performance Model

For effective decision-making within the solver, Sponge needs knowledge of the performance metrics, specifically the throughput $h(b,c)$ and latency $d(b,c)$, associated with the DL model. Previous research has indicated that the performance of DL inference tends to be highly predictable [11, 18, 23, 35]. We follow the same line and use profiling data and robust regressions [13] to build a model for any given DL model. GrandSLAm [11, 23] suggests a linear relationship between batch size and latency, that is, $l(b,c) = \alpha_1 \times b + \beta_1$, and FA2 [28] suggests a second-order quadratic polynomial for a

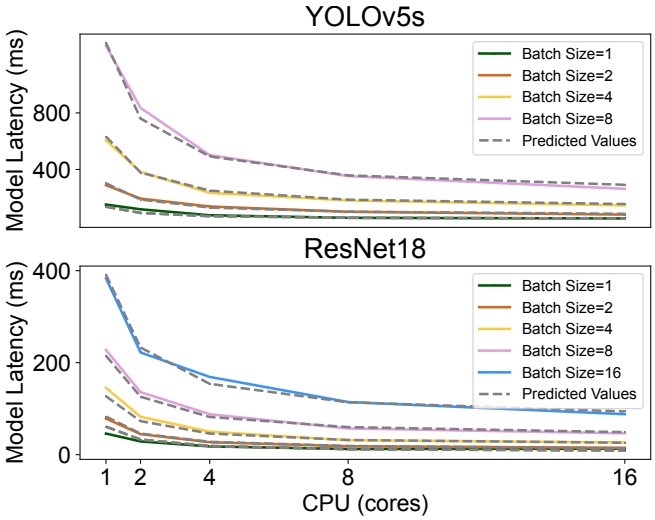

Figure 3: Latency vs. different CPU core allocations and batch sizes using real and predicted for the YOLOv5n and ResNet18 DL models.

lower total MSE. However, none of the above works consider changes in the computational resources (e.g., number of CPU cores) of the DL models. For simplicity, we use the linear relation in the current work.

To have a relation between latency and CPU, we use Amdahl's law [2] for latency prediction under a given batch size:

$$L(b,c) = \frac{\alpha_2}{c} + \beta_2 \tag{1}$$

Equation 1 states an inverse relation between the number of CPU cores and latency if the model can use additional CPU cores, which is the case in ML models.

On the other hand, the linear relation of batch size and latency suggests that α_1 and β_1 have inverse relations with CPU cores, e.g., $\alpha_1 = \gamma_1/c + \delta_1$ and $\beta_1 = \epsilon_1/c + \eta_1$ (otherwise, $l(b,c)$ would become linear in Equation 1). Therefore, to incorporate computational resources into batch/latency profiling, we combine the linear relation of batch/latency and the inverse relation of CPU/latency as follows.

$$\begin{aligned} l(b,c) =& (\frac{\gamma_1}{c} + \delta_1) \times b + \frac{\epsilon_1}{c} + \eta_1 \\ =& \frac{\gamma_1 \times b}{c} + \frac{\epsilon_1}{c} + \delta_1 \times b + \eta_1 \end{aligned} \tag{2}$$

Our preliminary evaluation with the data sets profiled from ResNet18 and YOLOv5n models used in Figure 3 confirms that the latency/CPU/batch model in Equation 2 provides a realistic estimation of latency with different CPU cores and batch sizes on different DL models. The throughput of a DL model is directly given as a function of batch size and CPU cores, e.g., $h(b,c) = b/l(b,c)$.

Table 2: Notations

Symbol	Description
R	Set of all requests
b	Model's batch size
c	Model's CPU allocation
cl_r	Communication latency associated with $r \in R$
cl_{max}	Highest cl_r in R
SLO	Pre-defined SLO for R
$l(b,c)$	Processing time of a model with allocation core c and batch size b
$q_r(b,c)$	Queuing time of $r \in R$ with allocation core c and batch size b
$h(b,c)$	Throughput of a model with allocation core c and batch size b
λ	Request arrival rate

3.3 Problem Formulation

The optimizer generates scaling decisions by solving an optimization problem. Now, we provide a formal formulation for the problem given that the end-to-end latency for a request is the aggregation of the communication latency (the time the request takes to be received by the system from the user device), the queuing (the time the request spends in the queue before being processed), and the processing latencies (inference latency) of the request.

Suppose that we are given a model and a set of requests R with a predefined SLO. Each request $r \in R$ has communication latency cl_r. The arrival rate of the application request is denoted by λ. Due to the instability of the network, as we have already discussed in Section 2, we apply the earliest-deadline-first (EDF) queue ($q(b,c)$), similar to GradnSLAm [23], since request reordering prioritizes the processing of requests with lower remaining SLOs due to their more stringent completion deadlines.

Let us denote the number of CPU cores allocated and the batch size of the model by c and b, respectively. In addition, we use $cl_{max} = max(cl_r, r \in R)$ to indicate the highest communication latency in the current requests.

The monitoring system continuously reports to the adapter the average number of requests served by the model in a given period. To ensure the stability of the system, that is, no back pressure should form in the queue, and the throughput of the model should be no less than the expected request rate, that is, $h(b,c) \geq \lambda$. Such a constraint ensures that the model is sufficiently provisioned. As a result, the queuing of requests on the model will be under control.

The optimization problem is to decide c and b for the model such that under the workload λ, none of the request SLOs are violated. The goal is to minimize the amount of resources (CPU cores) used for the model. The problem can be formulated with the following integer program (IP):

$$\begin{aligned} \text{Minimize} \quad & c + \delta \times b \\ \text{subject to} \quad & l(b,c) + q_r(b,c) + \text{cl}_{max} \leq SLO, \quad \forall r \in R \\ & h(b,c) \geq \lambda \\ & b, c \in \mathbb{Z}^+ \end{aligned} \tag{3}$$

In the objective function, in addition to CPU cores, we incorporate an insignificant penalty term δ into the batch size to mitigate unnecessary latencies. The first constraint ensures that all requests for SLOs, including communication latency, will be satisfied. We use the smallest SLO in the current batch for all requests in the same batch because we do not intend to violate any remaining SLO requests. The second and third constraints are designed to maintain system stability, necessitating that the CPU cores and the batch size be constrained to positive integer values. The objective is to minimize the total amount of resources, that is, the total number of CPU cores given to the model. All the notation used is available in Table 2.

3.4 Solution

With IP and a single model, we use a brute force approach shown in Algorithm 1. We feed the requests with their remaining SLOs to a queue and then reorder them based on the EDF policy (lines 1–2). After finding the maximum communication latency in the set of requests (line 4), we then iterate over all possible batch sizes and CPU core allocations (lines 5–6). Furthermore, we check if the current configuration and all the requests in the subsequent batches will satisfy their remaining SLOs (lines 10–15). Note that there will be a waiting time for the subsequent batches equal to the processing latency of the previous batches, calculated in line 14. Finally, if there is no objection against the current batch size and CPU core allocation configurations (line 15), we send the found configuration to be enforced to the system. The algorithm generates the optimal CPU core allocation with the smaller batch size with the current allocation, since it iterates from 1 to the maximum CPU core and batch size allocations.

4 PRELIMINARY EVALUATION

Sponge is implemented in 6K lines of Python. For evaluation, we use a physical machine from Chameleon Cloud [24] equipped with Intel(R) Xeon(R) Gold 6240R (48 threads). To enable the in-place vertical scaling, we install the experimental branch of minikube [6] since the in-place vertical scaling feature is not yet in the official releases [3].

Baseline. We compare Sponge with a state-of-the-art horizontal autoscaler in inference serving systems, FA2, and static 8-core and 16-core instances. All approaches (including Sponge) use a YOLOv5s [33] with the performance modeling in Figure 3 to detect humans in images. We also set b_{max} and

Algorithm 1: Optimal CPU and batch size finder

input : SLO, Set of requests $r \in R$ with communication latency, Performance model
output : c, b

$q \leftarrow R$
Reorder q (EDF policy)
$n = len(R)$
Calculate cl_{max}
for c *in* $[1, c_{max}]$ **do**
  for b *in* $[1, b_{max}]$ **do**
    Calculate $l(b, c)$
    $better = True$
    $q_r = 0$
    for i *in* $[1, n, b]$ **do**
      if $l(b, c) + cl_{max} + q_r \geq SLO$ **then**
        $better = False$
        break
      $q_r = q_r + l(b, c)$
    if $better = True$ **then**
      return c, b

c_{max} to 16 for Sponge as there is no significant gain afterward. For the adaptation period, we set one second same as the network bandwidth interval in the dataset.

Workload generator. In order to assess Sponge in scenarios with dynamic network bandwidth, we design a workload generator that produces requests asynchronously at a fixed rate of 20 RPS with predefined SLOs similar to Figure 1. We use gRPC [16] to handle communication between all components of the system, including the workload generator.

Performance evaluation. Figure 4 demonstrates the overall performance of Sponge, FA2, and statically assigned CPU cores under a dynamic network bandwidth. Under a given workload and the remaining SLOs, FA2 violates a large number of requests' SLO (roughly 5% and over 50% violation in some severe cases (Time = 1 and 360 in the same Figure) when the bandwidth becomes limited since bringing new instances is tied with the cold startup issue, and FA2 needs roughly 10 seconds to find a new configuration, adjust itself, and stabilize the system. The statically assigned 8-core instance experiences SLO violations after a few seconds due to insufficient computational resources to handle the requests, necessitating a more powerful instance. Conversely, the 16-core instance shows almost no SLO violations, indicating potential over-provisioning of the DL model. Sponge solves the resource waste by dynamically changing the allocated CPU cores in response to the network bandwidth changes and reduces the amount of allocated CPU by over 20% while sacrificing less that 0.3% of SLO violations, compared to statically assigned 16-core instance.

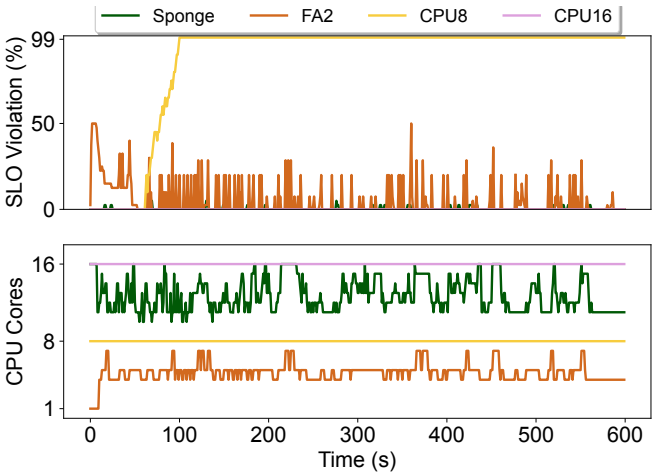

Figure 4: SLO violations and allocated CPU cores.

5 RELATED WORK

Inference serving with SLO guarantee. Multiple works have been proposed with SLO guarantees [12, 18, 31, 32]. Model switch [38] switches to a different model architecture in response to workload changes to ensure SLO. GrandSLAm [23] uses dynamic batching and request reordering to increase system throughput with the SLO guarantee. InFaaS [29] gets user preferences about accuracy, cost, or performance and provides a model variant to satisfy the requested SLO. Jellyfish [27] trades accuracy with latency by model switching and data adaptation to match the input of the model variant to guarantee latency SLO.

Autoscaling in inference serving. Autoscaling in inference serving has been extensively studied [10, 20, 30, 37]. Kubernetes VPA [5] and HPA [4] use threshold-based metrics such as CPU or memory usage to change computing resources or the number of instances of DL-based inference services. Clipper [11] provides an abstraction layer to simplify model deployment across frameworks and uses adaptive batching to increase system throughput. IPA [15] uses model switching and horizontal scaling to increase system accuracy while minimizing computing resources. Cocktail [19] uses a subset of model variants with a weighted scaling policy to ensure low cost, a predefined accuracy, and latency SLOs archived. FA2 [28] uses graph transformation and dynamic programming to design a new horizontal autoscaler to increase system utilization with SLO guarantees.

The mentioned approaches neither consider dynamic networks (wireless and 4G/5G) without changing the model variant that affects other metrics such as cost and accuracy nor use in-place vertical scaling, which Sponge has shown a necessity for state-of-the-art autoscalers to guarantee predefined latency SLO under a dynamic network bandwidth.

6 CONCLUSION & FUTURE WORK

In this work, we presented Sponge, the first inference serving system that uses in-place vertical scaling, request reordering, and dynamic batching with SLO guarantees. The preliminary evaluation shows that Sponge reduces the SLO violation to 0.3% while minimizing the CPU allocation in a dynamic network. We identify the following limitations of Sponge and consider them as future directions:

Model variant. There are variations of the same DL model with different configurations in terms of architecture that are capable of doing similar tasks with different objectives such as accuracy [27, 29, 38]. Incorporating model variants requires careful system design, since the three pillars of accuracy, latency, and CPU allocation (even without vertical scaling) have conflicting relations [31].

Pipeline. Many modern applications are composed of multiple DL models, such as Amazon Alexa, and are usually arranged as a DAG. Generalizing Sponge to support such applications requires a new algorithm design, since there is a data dependency [10, 15, 21, 28] between DL models and finding an optimal resource allocation for individual DL models requires consideration of all models in the system.

Multidimensional scaling. The resource requirements of a DL model can be influenced by the dynamic nature of workloads [17, 37], making them difficult to predict. Vertical scaling can support the incoming workload to a certain degree, meaning that horizontal scaling must be considered if the workload is too much for a single instance of a DL model. The joint optimization of horizontal scaling and vertical scaling mechanisms brings new challenges, such as changing an upstream DL model's processing latency rate (vertical scaling), which affects the input rates on downstream DL models and may require additional instances (horizontal scaling).

ACKNOWLEDGEMENTS

This work has been supported in part by NSF (Awards 2233873, 2007202, 2038080, and 2107463), Deutsche Forschungsgemeinschaft (DFG, German Research Foundation) – Project-ID 210487104 - SFB 1053, Roblox Corporation, and Chameleon Cloud.

REFERENCES

[1] Fawad Ahmad, Hang Qiu, Ray Eells, Fan Bai, and Ramesh Govindan. 2020. CarMap: Fast 3D Feature Map Updates for Automobiles. In *USENIX Symposium on Networked Systems Design and Implementation (NSDI)*. 1063–1081.

[2] Gene M Amdahl. 1967. Validity of the single processor approach to achieving large scale computing capabilities. In *Proceedings of the April 18-20, 1967, spring joint computer conference*. 483–485.

[3] The Kubernetes Authors. 2023. In-place Resource Resize for Kubernetes Pods. https://kubernetes.io/blog/2023/05/12/in-place-pod-resize-alpha/. (2023). Accessed on 30.01.2024.

[4] The Kubernetes Authors. 2024. Kubernetes Horizontal Pod Autoscaling. https://kubernetes.io/docs/tasks/run-application/horizontal-pod-autoscale/. (2024). Accessed on 30.01.2024.

[5] The Kubernetes Authors. 2024. Kubernetes Vertical Pod Autoscaling. https://cloud.google.com/kubernetes-engine/docs/concepts/verticalpodautoscaler/. (2024). Accessed on 30.01.2024.

[6] The Kubernetes Authors. 2024. Minikube. https://minikube.sigs.k8s.io/. (2024). Accessed on 30.01.2024.

[7] The Prometheus Authors. 2024. Prometheus monitoring and alerting toolkit. https://prometheus.io/. (2024). Accessed on 30.01.2024.

[8] Florian Brandherm, Julien Gedeon, Osama Abboud, and Max Mühlhäuser. 2022. BigMEC: Scalable Service Migration for Mobile Edge Computing. In *2022 IEEE/ACM 7th Symposium on Edge Computing (SEC)*. IEEE, 136–148.

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

[10] Daniel Crankshaw, Gur-Eyal Sela, Xiangxi Mo, Corey Zumar, Ion Stoica, Joseph Gonzalez, and Alexey Tumanov. 2020. InferLine: Latency-Aware Provisioning and Scaling for Prediction Serving Pipelines. In *ACM Symposium on Cloud Computing (SoCC)*. 477–491. https://doi.org/10.1145/3419111.3421285

[11] Daniel Crankshaw, Xin Wang, Guilio Zhou, Michael J Franklin, Joseph E Gonzalez, and Ion Stoica. 2017. Clipper: A {Low-Latency} Online Prediction Serving System. In *USENIX Symposium on Networked Systems Design and Implementation (NSDI)*. 613–627.

[12] Aditya Dhakal, Sameer G. Kulkarni, and K. K. Ramakrishnan. 2020. GSLICE: Controlled Spatial Sharing of GPUs for a Scalable Inference Platform. In *ACM Symposium on Cloud Computing (SoCC)*. 492–506. https://doi.org/10.1145/3419111.3421284

[13] Martin A. Fischler and Robert C. Bolles. 1981. Random Sample Consensus: A Paradigm for Model Fitting with Applications to Image Analysis and Automated Cartography. *Commun. ACM* 24, 6 (1981), 381–395.

[14] Saeid Ghafouri, Alireza Karami, Danial Bidekani Bakhtiarvan, Aliakbar Saleh Bigdeli, Sukhpal Singh Gill, and Joseph Doyle. 2022. MobileKube: Mobility-aware and energy-efficient service orchestration on kubernetes edge servers. In *2022 IEEE/ACM 15th International Conference on Utility and Cloud Computing (UCC)*. IEEE, 82–91.

[15] Saeid Ghafouri, Kamran Razavi, Mehran Salmani, Alireza Sanaee, Tania Lorido-Botran, Lin Wang, Joseph Doyle, and Pooyan Jamshidi. 2024. IPA: Inference Pipeline Adaptation to Achieve High Accuracy and Cost-Efficiency. (2024). arXiv:cs.DC/2308.12871

[16] grpc [n. d.]. gRPC. https://grpc.io. ([n. d.]). Accessed on 29.10.2021.

[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 *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 *USENIX Symposium on Operating Systems Design and Implementation (OSDI)*. 443–462.

[19] Jashwant Raj Gunasekaran, Cyan Subhra Mishra, Prashanth Thinakaran, Bikash Sharma, Mahmut Taylan Kandemir, and Chita R Das. 2022. Cocktail: A multidimensional optimization for model serving in cloud. In *USENIX NSDI*. 1041–1057.

[20] Yitao Hu, Rajrup Ghosh, and Ramesh Govindan. 2021. Scrooge: A cost-effective deep learning inference system. In *Proceedings of the ACM Symposium on Cloud Computing*. 624–638.

[21] Yitao Hu, Weiwu Pang, Xiaochen Liu, Rajrup Ghosh, Bongjun Ko, Wei-Han Lee, and Ramesh Govindan. 2021. Rim: Offloading Inference

to the Edge. In *Proceedings of the International Conference on Internet-of-Things Design and Implementation*. 80–92.

[22] Junxian Huang, Feng Qian, Alexandre Gerber, Z Morley Mao, Subhabrata Sen, and Oliver Spatscheck. 2012. A close examination of performance and power characteristics of 4G LTE networks. In *Proceedings of the 10th international conference on Mobile systems, applications, and services*. 225–238.

[23] Ram Srivatsa Kannan, Lavanya Subramanian, Ashwin Raju, Jeongseob Ahn, Jason Mars, and Lingjia Tang. 2019. GrandSLAm: Guaranteeing SLAs for Jobs in Microservices Execution Frameworks. In *ACM European Conference on Computer Systems (EuroSys)*. 1–16. https://doi.org/10.1145/3302424.3303958

[24] Kate Keahey, Jason Anderson, Zhuo Zhen, Pierre Riteau, Paul Ruth, Dan Stanzione, Mert Cevik, Jacob Colleran, Haryadi S. Gunawi, Cody Hammock, Joe Mambretti, Alexander Barnes, François Halbach, Alex Rocha, and Joe Stubbs. 2020. Lessons learned from the Chameleon testbed. In *Proceedings of the 2020 USENIX Annual Technical Conference (USENIX ATC '20)*. USENIX Association.

[25] Veton Kepuska and Gamal Bohouta. 2018. Next-generation of virtual personal assistants (microsoft cortana, apple siri, amazon alexa and google home). In *2018 IEEE 8th annual computing and communication workshop and conference (CCWC)*. IEEE, 99–103.

[26] Luyang Liu, Hongyu Li, and Marco Gruteser. 2019. Edge assisted real-time object detection for mobile augmented reality. In *The 25th annual international conference on mobile computing and networking*. 1–16.

[27] Vinod Nigade, Pablo Bauszat, Henri Bal, and Lin Wang. 2022. Jellyfish: Timely Inference Serving for Dynamic Edge Networks. In *2022 IEEE Real-Time Systems Symposium (RTSS)*. 277–290. https://doi.org/10.1109/RTSS55097.2022.00032

[28] Kamran Razavi, Manisha Luthra, Boris Koldehofe, Max Mühlhäuser, and Lin Wang. 2022. FA2: Fast, accurate autoscaling for serving deep learning inference with SLA guarantees. In *2022 IEEE 28th Real-Time and Embedded Technology and Applications Symposium (RTAS)*. IEEE, 146–159.

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

[30] Francisco Romero, Mark Zhao, Neeraja J Yadwadkar, and Christos Kozyrakis. 2021. Llama: A Heterogeneous & Serverless Framework for Auto-Tuning Video Analytics Pipelines. In *ACM Symposium on Cloud Computing (SoCC)*. 1–17.

[31] Mehran Salmani, Saeid Ghafouri, Alireza Sanaee, Kamran Razavi, Max Mühlhäuser, Joseph Doyle, Pooyan Jamshidi, and Mohsen Sharifi. 2023. Reconciling High Accuracy, Cost-Efficiency, and Low Latency of Inference Serving Systems. In *Proceedings of the 3rd Workshop on Machine Learning and Systems*. 78–86.

[32] Haichen Shen, Lequn Chen, Yuchen Jin, Liangyu Zhao, Bingyu Kong, Matthai Philipose, Arvind Krishnamurthy, and Ravi Sundaram. 2019. Nexus: A GPU Cluster Engine for Accelerating DNN-Based Video Analysis. In *ACM Symposium on Operating Systems Principles (SOSP)*. 322–337. https://doi.org/10.1145/3341301.3359658

[33] ultralytics. 2024. YOLOv5. https://github.com/ultralytics/yolov5. (2024). Accessed on 30.01.2024.

[34] J. van der Hooft, S. Petrangeli, T. Wauters, R. Huysegems, P. R. Alface, T. Bostoen, and F. De Turck. 2016. HTTP/2-Based Adaptive Streaming of HEVC Video Over 4G/LTE Networks. *IEEE Communications Letters* 20, 11 (2016), 2177–2180.

[35] Chad Verbowski, Ed Thayer, Paolo Costa, Hugh Leather, and Björn Franke. 2018. Right-Sizing Server Capacity Headroom for Global Online Services. In *IEEE International Conference on Distributed Computing Systems (ICDCS)*. 645–659.

[36] Dongzhu Xu, Anfu Zhou, Xinyu Zhang, Guixian Wang, Xi Liu, Congkai An, Yiming Shi, Liang Liu, and Huadong Ma. 2020. Understanding operational 5G: A first measurement study on its coverage, performance and energy consumption. In *Proceedings of the Annual conference of the ACM Special Interest Group on Data Communication on the applications, technologies, architectures, and protocols for computer communication*. 479–494.

[37] 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.

[38] Jeff Zhang, Sameh Elnikety, Shuayb Zarar, Atul Gupta, and Siddharth Garg. 2020. Model-switching: Dealing with fluctuating workloads in machine-learning-as-a-service systems. In *12th {USENIX} Workshop on Hot Topics in Cloud Computing (HotCloud 20)*.