Enabling Cost-Effective, SLO-Aware Machine Learning Inference Serving on Public Cloud

Chengliang Zhang, *Student Member, IEEE*, Minchen Yu, *Student Member, IEEE*, Wei Wang, *Member, IEEE*, and Feng Yan, *Member, IEEE*

Abstract—The remarkable advances of Machine Learning (ML) have spurred an increasing demand for ML- as-a-Service on public cloud: developers train and publish ML models as online services to provide low-latency inference for dynamic queries. The primary challenge of ML model serving is to meet the response-time Service-Level Objectives (SLOs) of inference workloads while minimizing serving cost. In this article, we proposes MArk (Model Ark), a general-purpose inference serving system, to tackle the dual challenge of SLO compliance and cost effectiveness. MArk employs three design choices tailored to inference workload. First, MArk dynamically batches requests and opportunistically serves them using expensive hardware accelerators (e.g., GPU) for improved performance-cost ratio. Second, instead of relying on feedback control scaling or over-provisioning to serve dynamic workload, which can be too slow or too expensive, MArk employs predictive autoscaling to hide the provisioning latency at low cost. Third, given the stateless nature of inference serving, MArk exploits the flexible, yet costly serverless instances to cover occasional load spikes that are hard to predict. We evaluated the performance of MArk using several state-of-the-art ML models trained in TensorFlow, MXNet, and Keras. Compared with the premier industrial ML serving platform SageMaker, MArk reduces the serving cost up to $7.8\times$ while achieving even better latency performance.

Index Terms—Machine-learning-as-a-service, inference serving, SLO awareness, cost minimization, cloud computing

1 Introduction

Machine Learning (ML) technologies have advanced by leaps and bounds in the past few years, leading to a burgeoning demand of Machine-Learning-as-a-Service (MLaaS) systems on the public cloud. A typical workflow of MLaaS system covers the two phases of ML, *training* and *inference*. In the training phase, developers build ML models from the training dataset using an array of ML frameworks. Efficient training in cloud environments has been well explored in the recent work [53], [67], [89]. In the inference phase, the trained models are published as *online cloud services* and can be queried by users with new input. The service makes prediction decisions (inference) for a given input using the trained model [39] (e.g., recognizing human faces in a given photo) and returns the inference results to the querier.

Unlike training which runs offline and may take hours to days to complete, inference must be performed in *real-time* over dynamic queries with stringent latency requirements (e.g., tens to hundreds of milliseconds per query). These requirements are often specified as the *response-time Service-Level Objectives* (SLOs) [51], e.g., at least 98 percent of inference queries must be served in 200 ms. Failing to comply with the SLOs results in compromised quality of service or even financial loss, e.g., end users will not be charged for queries not responded in time. Therefore, an ML model serving system should strive to meet the target SLOs while minimizing the cost of provisioning the serving instances in the cloud.

However, achieving these two objectives can be challenging. Cloud providers like Amazon [16], Google [47], and Microsoft [63] offer a wide variety of service provisioning options, ranging from VMs and containers to the emerging serverless functions. There is a wide configuration space for each provisioning option (e.g., CPU, memory, and hardware accelerators) coupled with diverse pricing models offering flexible tradeoffs between service guarantees and cost savings (e.g., on-demand and spot instances [22]). A key challenge of provisioning model serving in the cloud is: how does a serving system choose from a bewildering array of cloud services to provide low-latency, cost-effective inference at scale?

Unfortunately, there is no general guidance provided by the cloud providers, nor has it been studied in previous research [15], [34], [52], [55], [69], [70], [74], [82] which mainly targets general applications. To bridge this gap, we perform comprehensive measurement studies of inference serving in AWS [16] and Google Cloud [47] using VMs (IaaS), containers (CaaS), and serverless functions (FaaS). We briefly summarize our three key findings as follows.

First, among the three alternatives, Infrastructure-as-a-Service (IaaS) provides the best performance-cost ratio for inference serving, but it requires long instance provisioning latency and is unable to adapt to the changing workload timely. Container-as-a-Service (CaaS) suffers from a similar, yet less severe, problem with even worse performance-cost ratio. Compared to IaaS and CaaS, Function-as-a-Service (FaaS) scales much faster but is the most costly.

- *Chengliang Zhang, Minchen Yu, and Wei Wang are with the Department of Computer Science and Engineering, Hong Kong University of Science and Technology, Clear Water Bay, Kowloon, Hong Kong. E-mail: {czhangbn, myuaj, weiwa}@ust.hk.*
- *Feng Yan is with the Department of Computer Science and Engineering, University of Nevada, Reno, NV 89557 USA. E-mail: fyan@unr.edu.*

Manuscript received 9 April 2020; revised 28 June 2020; accepted 30 June 2020. Date of publication 2 July 2020; date of current version 6 September 2022.
(Corresponding author: Wei Wang.)
Recommended for acceptance by C. Wu.
Digital Object Identifier no. 10.1109/TCC.2020.3006751

Second, inference serving benefits from *batching* greatly when performed using costly hardware accelerators (e.g., GPUs). Nonetheless, the benefits are not always guaranteed but critically depend on the batch size control knobs and their interactions with query arrivals: when there is not enough load, serving inference queries using GPUs is not economically justified. Therefore, a serving system should judiciously determine when to *scale up* from CPU to GPU instances and how to perform batching over GPUs.

Third, ML inference usually performs *stateless* computations. This opens up an opportunity of using serverless functions as a *handover service* when the system is provisioning new instances for scaling up/out. In addition, many ML models, especially deep learning, have *deterministic inference time* [51], [86]—they take fixed-size input vectors and have input-independent control flows. This also brings an opportunity for better resource planing and latency control.

Based on these observations, in this paper, we propose Model Ark (MArk), a low-latency, cost-effective inference serving system on the public cloud. MArk takes advantage of the unique characteristics of ML model serving while addressing the distinctive challenges it raises. In particular, MArk allows developers to specify the target SLOs through common APIs. To attain high performance-cost ratio, it uses IaaS as the primary means of provisioning while utilizing FaaS to quickly fill the service gap when the system is undergoing horizontal/vertical scaling. MArk uses *predictive scaling* to mask the instance provisioning latency in IaaS. Unpredicted load spikes are covered by serverless functions to reduce over-provisioning. Based on the predicted workload, MArk opportunistically uses costly GPU instances to serve *batched queries* for improved performance-cost ratio. To further cut costs, MArk also supports the use of heavily discounted, yet *interruptible instances* (e.g., spot instances) with an interruption-tolerant mechanism that uses transient servers to handle instance interruptions at low cost.

We have prototyped MArk as a general-purpose serving platform in AWS [16] with pluggable backend model servers supporting a range of ML frameworks such as Tensorflow Serving [66], MXNet Model Server [32], and customized Keras [38] server with Theano [35] backend. We have evaluated MArk with several state-of-the-art ML models for image recognition, language modeling, and machine translation: Inception-V3 [79], NASNet [90], LSTM-ptb [62], and OpenNMT [58]. The results show that MArk yields up to $7.8\times$ cost reduction with comparable or even shorter latency than the state-of-the-practice solution SageMaker [18], while complying with the predefined SLO requirements. MArk is open-sourced for public access.[1]

2 Background and Related Work

In this section, we survey related work on model serving systems and autoscaling techniques. We also provide background information on cloud services and their pricing models.

2.1 Machine Learning Model Serving

A wide array of ML inference serving systems have been proposed to facilitate model deployment [7], [8], [32], [39], [66], [84]. These systems place the trained models in *containers* and handle model inference requests through REST APIs. For example, systems like Clipper [39], Rafiki [84], and MXNet Model Server [32] host each model in a separate Docker [4] container to ensure process isolation; TensorFlow Serving [66] deploy models as *servables*, which are executed as black box containers and can also be used for version management. In order to provide low-latency inference, these systems employ a number of model-agnostic optimizations such as batching, buffering, and caching [39], while relying on conventional container orchestration for scaling. The recently proposed *white box* model serving [60] enables model-specific optimizations with fine-grained resource sharing and parameter re-use.

However, existing inference serving systems mainly focus on streamlining model deployment in server machines, without addressing the scalability and cost minimization issues for model serving on the public cloud. Microsoft's Swayam [51] is among a few inference serving systems that focus on infrastructure scalability and resource efficiency. Yet Swayam is a proprietary system for model deployment in Microsoft's private MLaaS clusters. Nexus [75] is a GPU cluster engine that optimizes DNN inference throughput on a private cluster through techniques including dependency-aware scheduling, model fragmentation, and batching. The objective of Nexus is to increase the utilization of a pre-allocated GPU cluster dedicated for inference serving while we aspire to reduce the provisioning cost in public cloud. Amazon's SageMaker [18] offers scalable model serving over EC2 [1] instances. However, it only supports IaaS provisioning and requires manual specification of the provisioning instances. SageMaker is also *agnostic* to the response-time SLOs and serves inference queries in a best-effort manner. In contrast, MArk meets SLOs at low cost by choosing from a complex selection of provisioning services in AWS [16].

2.2 Autoscaling Dynamic Workload in Cloud

There is a large body of work on autoscaling dynamic workload for general web services hosted in the cloud. We refer to [70] for an extensive survey of this topic and compare some related work with MArk in Table 1. In general, there are two scaling approaches used to serve dynamic workload.

Feedback Control Scaling. This approach monitors hosted applications and *reactively* adjusts resource provisioning based on the monitored metrics (e.g., utilization, throughput, and latency). Feedback control scaling is adopted in many industrial serving platforms to autoscale dynamic workload, e.g., SageMaker in AWS [17], [18] and Kubernetes in Google Cloud [48], [49]. These systems perform scaling following some customized rules such as "adding two instances if CPU utilization reaches 70 percent," or tracking a target such as "maintaining 100 queries per minute per instance" [20].

Feedback control scaling makes no prediction and is easy to implement. However, owing to its reactive nature, it incurs long instance provisioning delay when used to serve the changing workload [70]. Over-provisioning is therefore needed in case of load spikes. For example, SageMaker recommends to start with 100 percent over- provisioning and adjust thereafter [21]. As ML model serving is often compute-intensive and requires costly CPU/GPU instances, solely relying on over-provisioning is economically not viable.

[1] https://github.com/marcoszh/MArk-Project

TABLE 1
A Comparison of MArk and Existing Work on Autoscaling Dynamic Workload in the Cloud

Autoscaler	Scaling approach	Means of Provisioning	SLO-aware	Heterogeneous instances	Interruptible instances	Hardware accelerators
MBRP [42]	Feedback control	Private cluster	✓	✓	×	×
Ali-Eldin *et al.* [14]	Predictive	IaaS	×	×	×	×
Barrett *et al.* [34]	Predictive	IaaS	×	×	×	×
Urgaonkar *et al.* [82]	Predictive	IaaS	✓	×	×	×
Han *et al.* [52]	Predictive	IaaS	✓	×	×	×
Qu *et al.* [69]	Feedback control	IaaS	×	✓	✓	×
SpotCheck [74]	–	IaaS	×	✓	✓	×
He *et al.* [55]	–	IaaS	×	✓	✓	×
Swayam [51]	Predictive	Private cluster	✓	×	–	×
SageMaker [18]	Feedback control	IaaS	×	×	×	✓
MArk	Predictive	IaaS and FaaS	✓	✓	✓	✓

Predictive Scaling. This approach makes predictions about the future workload, based on which it *proactively* autoscales the serving instances to reduce over- provisioning. Predictive scaling has been widely employed to serve general workload (e.g., web services and VM demands) using a number of time-series-based prediction algorithms, such as linear regression [36], autoregressive models [43], [72], and neural networks [26], [64], [68], [77]. Predictive scaling is often complemented with feedback control scaling, where the two approaches operate at different time scales [52], [82]. For example, predictive scaling can be used for resource planning in hours to days, with reactive provisioning operating in minutes to respond to flash crowds or unexpected deviations from long-term behaviors [82].

However, due to the mismatch of target workload, existing predictive autoscalers do not work well for ML model serving. As summarized in Table 1, they only consider provisioning over homogeneous instances on IaaS cloud [14], [34], [52], [82]. They also do not support hardware accelerators (e.g., GPUs) and cheaper, yet interruptible instances (e.g., spot servers), hence missing opportunities of cutting provisioning costs. In addition, many predictive autoscalers are *unaware* of the response-time SLOs and only provide best-effort services [14], [34]. As a result, such approach is seldomly adopted in real world deployment.

2.3 Cloud Provisioning Services

Compared with private clusters, model serving on public clouds is far more complex. Leading cloud platforms such as AWS [16], Google Cloud [47], and Microsoft Azure [63] offer a variety of provisioning services that can be used for model serving. We briefly survey these services, with a main focus on AWS.

Infrastructure-as-a-Service (IaaS). With IaaS, cloud customers run virtual instances (VMs) of various configurations in terms of CPUs, memory, storage, network, and accelerators (e.g., GPU, TPU, and FPGA). Customers can then configure and deploy ML model serving softwares [32], [39], [80] on running instances to serve model inference requests.

IaaS cloud provides flexible pricing options to allow customers to choose between service guarantees and cost savings. Taking Amazon EC2 [1] as an example, customers can run instances *on-demand* and pay for compute capacity by per hour or per second depending on the instance types. Alternatively, customers can run *spot instances* at steep discounts of the on-demand price, under the condition that a running spot instance can be *interrupted indefinitely* [22]. EC2 also allows customers to *reserve* an instance in a long term by making an upfront payment [29]. During the reservation period, the instance usage is subject to a heavy discount compared to the on-demand price. All three IaaS pricing options are also available in Google Cloud [47].

Container-as-a-Service (CaaS). With CaaS (e.g., Amazon ECS [2] and Google Kubernetes Engine [6]), customers encapsulate services and implementations in containers (Docker images [4]), and run containers with specified resource configurations. Compared with IaaS, CaaS simplifies software configurations and deployment without the complexity of maintaining the server infrastructure. In Amazon ECS, users pay for the container capacity by per second, where the pricing is based on requested vCPU cores and memory.

Function-as-a-Service (FaaS). With FaaS, customers run applications as *serverless functions* (e.g., AWS Lambda [3] and Google Cloud Functions [5]) and let the cloud platform to handle resource provisioning and management. In Lambda, customers can only specify the memory allocation for a function instance, and pay for the total number of requests and the compute time [3]. FaaS is particularly suitable for *stateless computations* and has become popular in serving ML models [81].

Given a complex selection of provisioning options in the public cloud, which one should be used for ML model serving? We answer this question in the next section.

3 Characterizing Model Serving on the Cloud

In this section, we characterize ML serving performance with IaaS, CaaS, and FaaS as well as their configuration space. Our characterizations are mainly based on AWS [16] (Sections 3.1-3.4), a leading cloud platform offering the most diversified service options. We validate the major results in Google Cloud [47] where possible (Section 3.5).

3.1 What Service to Use: IaaS, CaaS, or FaaS?

We choose three representative ML models, Inception-v3 [79], Inception-ResNet [78], and OpenNMT-ende [58], for common prediction tasks such as image classification and machine translation, and evaluate their peak inference performance with TensorFlow Serving [66]. Table 2 summarizes the cost and average latency of serving 1 million requests using AWS EC2 (IaaS), ECS (CaaS), and Lambda (FaaS), respectively.[2]

[2] Costs of instances are all based on AWS `us-east-1` region.

TABLE 2
Cost ($) and Average Latency ($t$) of Serving 1 Million Requests of Three ML Models in AWS

ML Model	EC2		ECS		Lambda	
	$	t (ms)	$	t (ms)	$	t (ms)
Inception-v3	5.0	210	9.17	217	19.0	380
Inception-ResNet	9.3	398	16.4	411	39.3	785
OpenNMT-ende	51.5	2180	96.3	2280	155	3100

We choose `c5.large` EC2 instance (2 vCPUs and 4GB memory) as it is the most cost-effective. Each ECS container is allocated the same vCPUs and memory as `c5.large`; each Lambda instance has 3GB memory to achieve comparable latency with `c5.large`.

TABLE 3
The Average Latency (t) and Cost ($) of Serving 1 Million Model Inferences With Bursted `t2` Instances

AWS `t2` Instance Size		micro	small	medium	large
Inception-v3	t (ms)	268.6	268.3	140.37	142.5
	$	0.87	1.71	1.81	3.75
Inception-ResNet	t (ms)	603.0	593.2	311.8	309.8
	$	1.94	3.79	4.01	7.96
OpenNMT-ende	t (s)	4.30	4.19	2.20	2.14
	$	13.85	24.83	28.36	56.71

IaaS versus CaaS. In EC2 [1], customers can choose among predefined instance types with fixed vCPUs and memory allocation. In Table 2, we choose the compute-optimized instance `c5.large` as the reference, as it is proven to be the most cost-effective choice in Section 3.3. AWS's container service ECS [2], on the other hand, lets users choose the number of vCPUs they want. We allocate each container with 2 vCPUs to match the capacity of `c5.large`, and with the minimum memory allowed. Compared with `c5.large`, the ECS container has similar serving latency but is more expensive.

FaaS. As for the serverless computing service Lambda [3], the pricing is on a per-request basis, and the cost per request depends on the resource allocation and runtime of the request. Customers specify memory allocation in Lambda, and CPU resource is allocated proportionally to memory [19]. For a fair comparison, we evaluate the Lambda cost of serving the same amount of requests that `c5.large` can serve in an hour, with the maximum memory allocated for best performance. The cost is significantly higher, and the latency is longer, too.

Scalability. EC2 has long provisioning overhead (e.g., several minutes), because more time is needed to load and set up large ML model serving in addition to standard overhead, as Microsoft suggests with their production traces [51]. The overhead makes it challenging to accommodate demand surge without high margin of over-provisioning. The high launching overhead also penalizes frequent provisioning and deprovisioning, since customers are billed during the instance launching period as well. Similar to EC2, ECS also needs dozens of seconds of provisioning overhead. Lambda, on the contrary, is able to spawn thousands of new ML inference instances in less than a few seconds, and once an instance is ready, it can continuously serve requests without incurring additional overhead [59]. The cold start overhead of Lambda can be amortized by warming up function instances [59]. Compared with EC2 and Lambda, ECS has no obvious advantage.

Summary. A natural question is that can we exploit the cost-effectiveness of IaaS service while also taking advantage of the high scalability of FaaS? Conventional cloud provisioning schemes have to over-provision because of the weak scalability of IaaS or CaaS. Now that ML serving is eligible for the highly scalable FaaS, we can reduce over-provisioning by combining IaaS and FaaS. The former is used as the primary serving option, with the latter providing transient service while new IaaS instances are launching. Moreover, FaaS can potentially handle the short lasting demand surges (short spikes), so that the overhead of frequent provisioning and deprovisioning can be eliminated. Although FaaS is costly, we believe the cost reduction from less over-provisioning can justify its high price tag.

With IaaS as the primary serving option, we shall determine how to choose from a bewildering array of instance families and sizes, which we discuss next.

3.2 IaaS: Can We Use Burstable Instances?

IaaS providers typically categorize instances into various families. Within a family, instances share the similar physical hardware but may have different sizes in terms of vCPUs, memory, and network bandwidth. For CPU instances, EC2 offers four main instance families: the general-purpose `m-family`, the compute-optimized `c-family`, the burstable `t-family`, and the memory-optimized `r-family`.

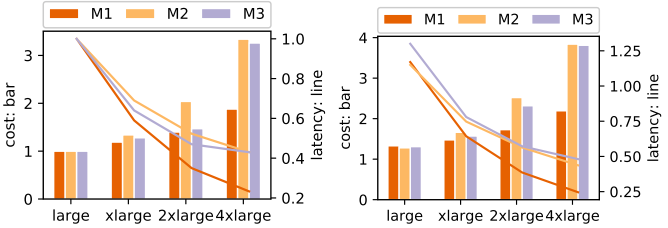
Among all instance types, burstable instances (`t-family`) have the lowest hourly rate, but they are aggressively multiplexed on overbooked servers [83], [85]. Burstable instances provide a baseline level (10 percent in AWS) of CPU performance with the ability to burst when required by the workload, yet with limited timespan according to a throttle policy (a new `t2` instance can sustain 100 percent utilization for 30 minutes) [30], [31].

We profiled `t2` instances' performance for ML serving, and the results are summarized in Table 3. We see that the latency drops proportionally with more vCPUs but adding more memory does not benefit the inference performance (e.g., upgrading from micro to small or from medium to large). Although it seems that `t2` instances are of low cost with viable latency for ML serving, these results are obtained in the bursted mode and do not sustain for a long time. Such drawback suggests that burstable instances are not for compute-intensive services [61].

Summary. While burstable instances are plausible for transient ML serving usage, they should not be used as the main long-running resources.

3.3 IaaS: Big Instances or Small Instances?

We further investigate CPU instance families compute-optimized `c-family` and general-purpose `m-family`, where we focus on the latest generation `c5` and `m5`. We exclude memory-optimized instances (`r-family`) from consideration, as our measurements on `t2` instances indicate that 4 GB of memory already does not bound the inference performance. In EC2, the configurations (vCPUs and memory)



(a) c5 instances (b) m5 instances

Fig. 1. The latency (lines) and cost (bars) of serving 1 million model inference requests with c5 and m5 instances. M1, M2, and M3 respectively denote Inception-v3, Inception-ResNet, and OpenNMT-ende. The values are normalized by that of c5.large (182.5ms with $4.3 for M1; 389ms with $9.4 for M2; 2.18s with $51.5 for M3).

and prices of m5 and c5 instances are proportional to their sizes. So it is important to see how scaling up to larger instances would affect the ML serving performance.

Figs. 1a and 1b depict the measured latency (lines) and cost (bars) of serving 1 million inference requests of three ML models using c5 and m5 instances of different sizes. In general, c5 instances are cheaper and result in lower latency than m5 instances because of more advanced CPU models, even though the latter have larger memory. Our results also suggest that, for CPU instances of the same family, smaller instances are more cost-effective, as the serving throughput grows *sub-linearly* with the instance size. At the same time, by scaling from a smaller instance to a bigger one, the latency drops sub-linearly as well.

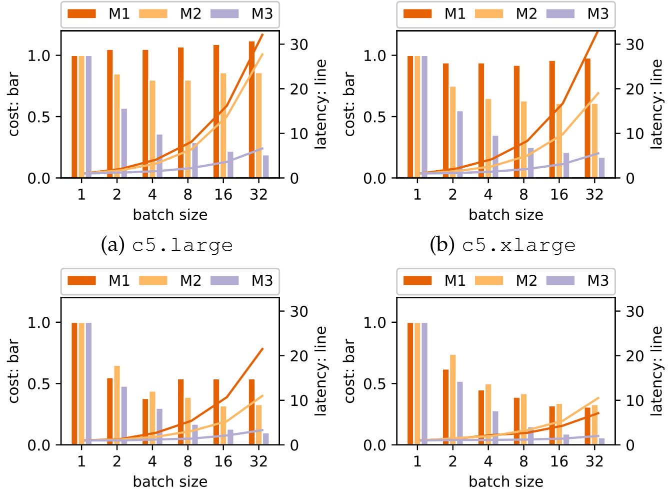
Summary. To sum up, smaller instances with advanced CPU models (c5.large in AWS) are favored as they achieve higher performance-cost ratio. Moreover, owing to the finer provisioning granularity, using smaller instances to serve dynamic workload improves the resource utilization. Note that the cost analysis presented here is based in on-demand market. Once we switch to the spot market, the cost-effectiveness is variable w.r.t. the change of spot price.

3.4 IaaS: How Does GPU Compare With CPU?

Many high-end IaaS instances are equipped with hardware accelerators, such as GPU and TPU (exclusive in Google Cloud), that can be used to speed up ML training and inference. The questions are: how would those hardware accelerators improve the latency of ML serving, and if such performance benefit can justify their high cost? For now, we focus on GPUs, which are the most accessible and popular general-purpose ML accelerators. We will extend our study to TPUs in Google Cloud in Section 3.5.

A GPU instance is more expensive than a CPU instance, but it can achieve up to 40× speedup due to its massive parallel nature according to NVIDIA [65]. In order to unleash the full power of its computing capability, it is essential to *batch* multiple inference requests and serve them in one go [80]. Batching benefits the performance in two ways. First, it amortizes the overhead of operations such as RPC calls and inter-device memory copy. Second, it can take advantage of batch operation optimization from both software and hardware [39], [73].

To disclose the intriguing performance difference between CPU instances and GPU instances as well as batching, we compare the inference performance of three ML models on c5 CPU instances and GPU instances p2.xlarge. We choose p2.xlarge as it is the smallest GPU instance in AWS (the next size available is p2.8xlarge which has 8 GPUs and is much more expensive). Fig. 2 shows the cost and latency of serving 1 million inference requests with various batch sizes (# of requests served in one batch) on c5 and p2.xlarge instances. For smaller CPU instances such as c5.large and c5.xlarge, the serving cost (bars) and latency improvement (lines) over batching is marginal (latency growing proportionally as the batch size), whereas bigger CPU instance (c5.4xlarge) displays certain improvement when batch size increases within a small range. GPU instances, on the other hand, benefit significantly from batching: the larger the batch, the lower the cost per request. This phenomenon suggests that batching can significantly improve the cost-effectiveness of larger CPU instances and GPU instances.



(a) c5.large (b) c5.xlarge (c) c5.4xlarge (d) p2.xlarge

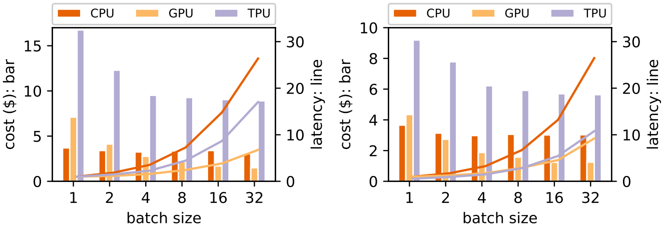
Fig. 2. The cost and batch latency of 1 million model inference with batching of various sizes. M1, M2, and M3 represents inception-v3, inception-resnet, and OpenNMT-ende. The cost and batch latency are normalized by the values when batch size is set to 1.

Serving multiple models on the same GPU is proposed by Nexus [75] to increase utilization in a pre-allocated GPU cluster. However, sharing GPUs incurs non-negligible context-switching overhead [57]. Since we focus on public cloud where users can choose from a rich selection of instance types to ensure a high instance utilization, the context-switching overhead of collocating models may not be justified as not much spare resources can be utilized on the rented instances.

Summary. With an appropriate batch size, GPU instances can achieve lower per-request cost and shorter inference latency than CPU instances. However, batch size cannot be increased arbitrarily as it leads to longer queuing latency and batch inference latency [39]. We will further discuss the batching configuration in Section 4 and formulate the problem in a latency-aware context.

3.5 Characterization in Google Cloud

So far, all our profiling experiments are based on AWS. To validate whether our main observations also apply to ML serving in the other cloud platforms, we extend our characterization to Google Cloud [47]. Google Cloud offers similar service and pricing options as AWS. In addition, it provides Tensor Processing Unit (TPU), the state-of-the-art ASIC dedicated to high-efficiency ML training.



(a) Inception-v3

(b) ResNet50

Fig. 3. The cost and batch latency of serving 1 million inference requests with various batch sizes. The batch latencies are normalized by the latency with no batching.

IaaS Remains the Best Option in Google Cloud. We first compare the cost and latency performance of ML serving using Google's IaaS, CaaS, and FaaS with the same workloads as in Section 3.1. All the experiments were run in `us-central1` region. Among the three provisioning options, IaaS remains the best with the lowest cost and shortest latency. For instance, the average latency and total cost of serving 1 million Inception-v3 requests on an customized IaaS instance with 1 vCPU and 2 GB memory are 317 ms and $3.70, respectively. In comparison, it takes 319 ms and $4.17 using the cheapest CaaS instance `n1-standard-1` (1 vCPU and 3.75 GB memory), and 527 ms and $17.4 using Google Cloud Functions (FaaS) with 2 GB memory.

Small Instances Offering Higher Performance-Cost Ratio. We then compare the cost and latency performance of CPU instances of various sizes within the same family. We made the similar observations as in AWS (Section 3.3): smaller instances offer higher performance-cost ratio than the bigger ones, though the latter leads to shorter latency. In particular, when serving 1 million Inception-v3 requests with `n1-standard-1`, `n1-standard-2`, and `n1-standard-4`, the cost (average latency) ends up with $4.16 (319 ms), $7.82 (296 ms), and $11.98 (227 ms), respectively.

3.6 How About ML ASICs?

Application-specific integrated circuits (ASICs) are deemed to be the step forward in delivering efficient ML. Premier cloud providers including Google and Amazon have all shown interests in developing ASICs for ML. Advanced ASIC products like TPUs from Google, NPUs from Cambricon [13], DLUs from Fujitsu, NNPs from Intel [25] are all well received in market. Among these efforts, Google's Tensor Processing Unit (TPU) is the front runner. It is the only one of its kind that is generally available on public cloud. The TPU, now in its 3rd iteration, is an ASIC built for training and inference of ML models. TPU benefits from thread parallelism like GPUs do, yet it removes any general-purpose additions in the architecture. TPUs are solely created and optimized from the ground up for ML, and are specialized for high-speed, low-precision floating-point operations. Compared with GPUs, TPUs are more power-efficient while achieving substantially better performance.

In order to understand how TPUs can be adopted in ML serving, we compare the cost and latency performance of using CPU, GPU, and TPU instances for ML serving with various batch sizes. We choose two popular image classification models, Inception-v3 and ResNet50 [54]. The results are shown in Fig. 3, where we use a customized CPU instance with 1 vCPU and 2 GB memory (`CPU`), the same instance with a K80 GPU attached to it (`GPU`), and a Cloud TPU-v2 instance (`TPU`). We observe the similar trend of cost and latency w.r.t. batch size for CPU and GPU instances as in AWS (Section 3.4). As for TPU, we find that its high price tag does not justify the performance benefit. In fact, TPU is a massively parallel accelerator optimized for training throughput rather than inference latency. Note that in Fig. 3, the batch size for TPU is calculated per core. As TPUv2 has 8 cores, the device batch size is actually 8 times the value. The design of TPU calls for large batch sizes to fully exploit its computing capacity [50]. However, the stringent latency requirement of real-time inference cannot wait for large batches to accumulate, leading to extremely low hardware utilization. In summary, TPUs are designed with ML training or large-batch offline inference in mind, thus not suitable for real-time ML serving under our setting.

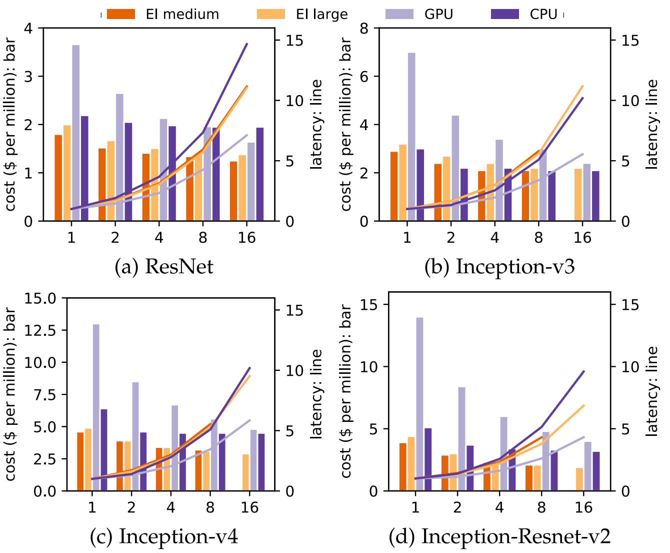
3.7 How About Dedicated Inference Accelerators?

Compared with training, model inference only performs forward propagation and has much smaller memory footprints. Exploiting these properties, cloud providers offer various specialized pieces of hardware optimized for ML inference. AWS recently offers Elastic Inference (EI) [9] and Inferentia [10]; Google offers Edge TPU [11]. Unlike TPUs that are with massive parallelism and optimized for throughput, these products have moderate computing power, and are designed specifically to facilitate low-latency inference tasks. Note that Edge TPUs are designed to be deployed physically on the edge as opposed to be accessible on cloud, so it is out of the scope of this paper. Besides that, to deploy trained models on Edge TPU or Inferentia, additional compilation processes including quantization are also needed, which require dedicated software SDK and may result in a model accuracy drop. Compared with Edge TPU and Inferentia, AWS EI, released in Spring 2019, is particularly attractive as it requires no additional engineering to the trained models. We hence explore the adoption of AWS EI to exploit the newly available inference accelerators.

AWS EI is essentially a service that offers large GPUs in small units. EI allows users to attach a GPU-powered inference accelerator to an EC2 or SageMaker instance. EI accelerators come with three sizes each having different capped FLOPS performance. Compared with renting standalone GPU instances, EI enables users to utilize GPU power for inference in a smaller granularity, meeting the lower computation demand of inference. The utilization of EI requires customized versions of ML frameworks, and some modifications of codes. EI accelerators are connected to the host machines via network, which has higher overhead than direct connections such as PCIe.

We conduct EI evaluations on MXNet with the configurations following the official guide [24]. We use the official MXNet inference benchmark script [12]. Fig. 4 illustrates the results. For EI, we use `c5.xlarge` as the host instance, and test EI accelerators `eia1.medium` and `eia1.large` respectively. For baselines, we use GPU instance `p2.xlarge` and CPU instance `c5.xlarge`.

Fig. 4 shows that without batching, EI can achieve comparable inference throughput of a standalone GPU instance,



(a) ResNet

(b) Inception-v3

(c) Inception-v4

(d) Inception-Resnet-v2

Fig. 4. The inference cost and latency of four models with EI, regular GPU and CPU instances w.r.t. batch sizes. The batch latencies are normalized by the latency with no batching. Some EI medium results are left empty because the evaluation tasks encounter out-of-memory errors.

with even lower cost compared with CPU-only instance. However, EI's performance does not benefit as much from batching, and its performance is limited by its small memory allocation. In fact, with an appropriate batch size, standalone GPU can significantly outperform EI instances with similar costs.[3] It is worth noticing that in our evaluations, EI instances incur much longer launching overhead than regular EC2 instances: it takes more than 20 seconds for an EI instance to be ready, while a standalone GPU instance only requires 7.4 seconds. A deep-dive inspection shows that the high launching overhead of EI instances is caused by transferring voluminous ML software and models over the network with limited bandwidth.

Summary. In our evaluations, Elastic Inference shows balanced cost and performance ratio compared with regular EC2 instances. However, with appropriate batching, standalone GPU instances can outperform EI. Furthermore, that EI instances cannot be obtained in spot market renders it less competitive in price than the regular EC2 instances.

3.8 Characterization Summary

We summarize our key findings as follows: (1) IaaS achieves the best cost and latency performance for ML model serving, and combining it with FaaS can potentially reduce over-provisioning while remaining scalable to spiky workloads. (2) Burstable instances are viable to cover transient ML serving demand. (3) In on-demand CPU market, smaller instances have higher performance-cost ratio than the bigger ones, even though the latter provides shorter latency. (4) Only with appropriate batching can the use of GPU instances be justifiable to achieve lower cost and shorter latency than CPU instances.

4 MArk

In this section, we present MArk (Model Ark), a scalable system that provides cost-effective, SLO-aware ML inference serving in AWS. While MArk is built in AWS, nothing prevents our design from being extended to the other cloud platforms with similar service offerings, such as Google Cloud and Microsoft Azure.

4.1 Overview

Following our observations in Section 3, MArk uses EC2 as the primary means of provisioning ML serving. It also uses Lambda to quickly cover the service gap when there is a need to scale out/up. Fig. 5 illustrates the overall architecture of MArk. In particular, requests from clients are deposited to a request queue, and are grouped into batches by the *Batch Manager* (details in Section 4.3). MArk periodically measures the workload metrics, such as the request arrival rate, and sends them to a *Proactive Controller* which makes predictions and plans instances in advance to reduce over-provisioning (Section 2.2). The controller then sends the launching and destroying requests to EC2 instances, on which custom service backends such as Tensorflow Serving [66] are hosted. The controller also monitors the health status of all running instances. With predictive scaling, further actions are needed to handle prediction errors and unexpected load spikes. On each running EC2 instance, there is a *Bouncer* which monitors serving metrics and performs request admission control. Whenever there is an incoming request, Bouncer checks whether its own host instance can finish the inference within the specified time $RT_{\max}$. If not, the Bouncer rejects the request and reroutes it to be handled by Lambda instances immediately. In addition, MArk employs an *SLO Monitor* that keeps track of and maintains the SLO compliance with the method described in Section 4.4.

SLO Requirements. Following Swayam [51], we set two SLO requirements for MArk. (1) *Response Time Threshold*: a request is deemed fulfilled only if its response time is below $RT_{\max}$. (2) *Service Level*: the service is considered satisfactory only if at least $SL_{\min}$ percent of the requests are fulfilled.

4.2 Workload Prediction

MArk employs predictive scaling to reduce over-provisioning. To expose the long-term cost trade-off between different instances and resource provisioning, we need to estimate the maximum request rate in the near future, which requires multi-step workload prediction. Existing works employ many well-established resource estimation methods, such as linear regression [36], autoregressive models [43], [72], and neural networks [26], [64], [68], [77]. As the accuracy of prediction depends on the underlying workload, there is no universal method that works perfectly in all cases. Therefore, MArk exposes an API through which users can implement their own workload prediction methods that best fit their applications. The challenge is how to gracefully handle unavoidable prediction errors and unexpected load surges.

In this paper, we adopt a vanilla version of long short-term memory (LSTM) network [46] as an example for multi-step workload prediction as it is reported to have a generally good prediction performance [76]. It is worth noting that in general, it is hard to find a universally optimal

[3] All the said cost comparisons are calculated in on-demand market. In fact, EC2 GPU instances can enjoy generous discounts in spot market, while EI only supports on-demand market at the moment.

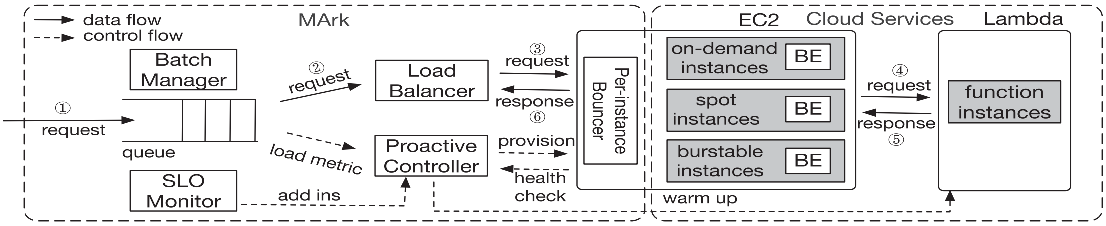


Fig. 5. An overview of the MArk model serving system.

prediction method for all diverse workloads. Users can replace the LSTM method with other prediction methods via MArk's APIs. In this paper we do not focus on finding the optimal prediction algorithm, but rather demonstrating the performance and cost benefits when the prediction is accurate, as well as the effectiveness of using FaaS as a quick handover approach when the prediction errs. In our implementation, the prediction unit (time interval) is P_u, and the prediction window is P_w, meaning that MArk updates the predicted load for the next $P_w P_u$ interval every P_u time units. During each unit, MArk keeps sampling the arrival rate in consecutive short sample windows of P_s. It keeps track of the maximum arrival rate of the unit, and gets the maximum arrival rate array for the next P_w units. In our evaluations, $[P_u, P_w, P_s]$ is set to $[1\text{min}, 60, 5\text{sec}]$ with the following justifications. We set the prediction unit to 1 minute as EC2 charges at least 1 minute for new instances. We set the prediction window to 60 steps because 1 hour of future trend is good enough to expose the long term trade-offs. The sample size is set to 5 seconds, since the arrival rate can be treated as stable in short time slots [86].

4.3 Instance Provisioning and Batching

We formulate the instance provisioning problem and show that it is intractable even in a simplified form. We hence turn to an effective online heuristic algorithm as the solution.

Formulation. Given the diverse cloud service options, MArk essentially orchestrates a heterogeneous cluster with fast changing demand. We formulate how to optimally choose the right instance types and their numbers to serve dynamic demands. Following the classic web service arrival analysis, we assume Poisson arrivals for the prediction requests and formulate a queueing model for the model servers. At the time of writing, AWS employs a new pricing scheme for spot instances, where the prices no longer fluctuate constantly but stay relatively stable most of the time [22]. It is hence safe to presume that the spot price will not change during our narrow prediction window. Our goal is to serve all the requests with minimum cost possible while meeting the SLO requirements.

We start by introducing the notations used in the problem formulation. Let I be the set of instance types available for model serving. Let P_i and O_i respectively denote the instance price per unit time and the launching overhead (i.e., the incurred cost during the instance launching period, which spans from the instance launching time to its readiness) of instance type $i \in I$. Let λ_t be the predicted arrival rate at time step $t \in T$. Let c_i be the service rate capacity of instance type i. We further denote $n_{i,t}$ as the number of type-i instances running at time t and $\lambda_{i,t}$ the arrival rate of the request load that is served by instances of type i. We assume *deterministic* inference time [86] and model the running servers of instance type i as an M/D/c_i queue [86], where c_i measures the inter-request parallelism. We formulate the following optimization problem that minimizes the instance provisioning cost while meeting the SLO requirements of the inference workload

$$\begin{aligned}
\text{minimize} \quad & \sum_{t\in T}\sum_{i\in I}\left[n_{i,t}P_i + O_i\max\left(n_{i,t}-n_{i,t-1},0\right)\right] \\
\text{subject to} \quad & \sum_{i\in I}\lambda_{i,t} \ge \lambda_t, \quad \forall t \in T; \\
& W^{M/D/c_i\left(\frac{\lambda_{i,t}}{n_{i,t}}\right)} \le \ell, \quad \forall i \in I, t \in T.
\end{aligned}$$

We explain our formulation in more detail. The optimization objective (i.e., provisioning cost) consists of two parts: the overall instance running cost plus the overhead of launching new instances at each time step. There are multiple constraints that must be satisfied. The first is the capacity constraint, meaning that the accumulated capacity of all running instances must be able to accommodate all requests in the predictable future. The second is the SLO constraint, where $W^{M/D/c_i}$ is the average latency of instance type i under load $\lambda_{i,t}$, and ℓ is the target average latency specified in SLO. The solution is to determine how many instances of type i should be running at each time t, i.e., finding the decision variable $n_{i,t}$.

However, we note that such a complex optimization problem has no closed-form solution even without considering request batching and instance pricing [86]. Given the intractability of this problem, we turn to a heuristic solution: instead of jointly considering batching and instance provisioning, we solve the two problems separately using heuristic algorithms.

Batching. Inspired by the adaptive batching in [39], we introduce two hyperparameters to control the batching behavior of an instance type: W_{batch} which is the maximum waiting time window for request batching, and N_{batch} which is the maximum batch size. The Batch Manager fetches requests from the queue, and submits the batched requests if either of the two limits is reached (Fig. 5). We tune the two hyperparameters to meet the following two requirements: (1) No SLO requirements can be violated, meaning that the waiting time window and the processing time of the batch together should be capped by response time threshold $RT_{\max}$; (2) the throughput with batching enabled must be greater than that of no batching. That is, the waiting time window and the batch processing time together should be less than the time needed to process all those requests sequentially without batching.

In practice, hyperparameter tuning requires light profiling for the target instance. We first profile the optimal processing rate of the target instance without batching, which we denote by μ^*_{nb}. We then gradually increase the batch size from 1 until one of the following constraints is violated

$$W_{\mathrm{batch}} + T_b \leq RT_{\max}, \quad W_{\mathrm{batch}} + T_b \leq \frac{b}{\mu^*_{nb}},$$

where b is the batch size, and T_b is the time needed to process a batch.

Now that we have the optimal batch size $N_{batch} \leftarrow b$ and the maximum processing rate μ^* under this configuration, together with their maximum waiting time window W_{batch}, we can simply treat the target instance as a black box with processing rate μ^*.

Instance Provisioning. We now solve the instance provisioning problem using an online heuristic algorithm that considers both long-term cost-effectiveness and the launching overhead, while at the same time attaining high utilization of running instances.

We first introduce the notations. Suppose that there are n types of instances that can be used for serving. At a given time t_0, let $R = \{r_1, r_2, \ldots, r_n\}$ be the set of running instances and $F = (F_1, \ldots, F_m)$ the predicted maximum request arrival rate for the next m steps, where F_t is the predicted maximum rate in step t. For each instance type i, let C_i be the instance capacity, measured by the maximum throughput of a given model (requests per hour). Let P_i be its unit price and O_i its launching overhead. Finally, let I be the set of available instance types. Given R, F, I and the target SLO, our problem is to determine what instances to launch and which instances to destroy at t_0, so as to minimize the cost while meeting the target SLO.

The challenge of finding the optimal solution in the long run is how to deal with the running instances at t_0. They may not be the most cost-effective in the next m steps, but keeping using them avoids additional launching overhead. We propose a greedy solution in Algorithm 1. Our intuition is to greedily find the most cost-effective instance from time period t_0 to t_m considering both the pay-as-you-go fee and the launching overhead. The running instances at t_0 can be treated as special ones with zero launching overhead.

In our algorithm, assuming most instances can get ready in τ time units after launching, we use the predicted load at $t_0 + \tau$ as the provisioning target, as it is safe to make instance provisioning decisions τ time units in advance. The values of τ can be easily adjusted based on the actual scenario. In our setup, τ is set to 5 minutes, and the scheduling time unit is set to 1 minute. In this case, the scheduling decisions are made every minute, targeting the load in 5 minutes. The launching requests should be sent right away once the $instance\_plan$ is ready; the destroying requests, on the other hand, should be sent after a predefined cool-down period to ensure better service quality [70].

It is worth mentioning that Algorithm 1 trivially meets the SLO requirement by ensuring that the latency performance of each selected instance comply to the target SLO individually.

4.4 SLO Tracking

The heuristic in Algorithm 1 plans instance capacity based on predictions. Yet not all demand surges are predictable, and such surges would result in SLO violations if solely relying on proactive provisioning [70]. To further improve the SLO compliance, MArk actively monitors request latency, and *reactively scales* the cluster as soon as SLO violations are detected. MArk constantly checks if the last M requests satisfy the SLO requirements. If not, L instances of type T will be launched (`c5.large` by default). All those parameters can be tuned for specific models and SLO requirements.

Algorithm 1. Greedy Algorithm

```
procedure Schedule $F, R, I$, SLO
  $S \leftarrow S \cup R$ ▷ Running instances are treated as special ones
  with zero launch overhead
  forall instance $i$ in $S$ do
    if instance $i$ cannot meet SLO requirement then
      $S = S \setminus \{i\}$                    ▷ Remove $i$ from $S$
  if $S = \emptyset$ then
    Report error       ▷ No candidate instance can meet SLO
  $instance\_plan \leftarrow \emptyset$        ▷ initialize provisioning plan
  Fill($F, S, instance\_plan$)
  Launch instances in $instance\_plan$ but not in $R$
  Destroy instances in $R$ but not in $instance\_plan$
procedure Fill $F, S, instance\_plan$
  $C^{\mathrm{sum}} \leftarrow$ total capacity of all instance $i$ in $instance\_plan$
  for $t = 1$ to $m$ do
    $\Lambda_t = F_t - C^{\mathrm{sum}}$ ▷ Unfulfilled requests predicted at step $t$
  if $\Lambda_\tau \leq 0$ then        ▷ Planned capacity is enough at step $\tau$
    return
  Find the largest $e$ such that there are unfulfilled requests
  from steps $\tau$ to $e$, i.e., $\Lambda_t \leq 0$ for all $\tau \leq t \leq e$
  $min\_cost \leftarrow \infty$ ▷ Greedily search the instance with the low-
  est per-request cost to cover unfilled requests from $\tau$ to $e$
  forall instance type $i \in S$ do
    $cost \leftarrow (O_i + (e - \tau)P_i)/N$, where $N$ is the number of
    unfulfilled requests that will be served by an instance $i$
    in $[\tau, e]$
    if $cost < min\_cost$ then
      $min\_cost \leftarrow cost$
      $j \leftarrow i$
  $instance\_plan \leftarrow instance\_plan \cup \{j\}$
  Fill($F, S, instance\_plan$)
```

4.5 Spot Instance and Lambda Cold Start

*Use of Spot Instances.*Note that Algorithm 1 does not differentiate between on-demand and spot instances which, if utilized, could further bring down the serving cost due to its heavy price discount. However, the adoption of spot instances poses the challenge of instance interruptions. Although the interruption of a spot instance will be notified 2 minutes in advance, it may not be long enough for a substitute spot instance to get ready. The question is how can we handle the outstanding requests in the presence of instance interruptions? AWS Lambda seems to be a viable choice, but it would result in increased latency and cost.

Our answer to this challenge is the burstable instances. As shown in Section 3.2, burstable instances are cheap instances which can sustain full utilization for about 30 minutes. The low cost and high peak performance make them a perfect fit for transient backups in case of short-term interruptions. Moreover, burstable instances can be resumed from stopped state in less than 2 minutes thanks to their small sizes. Therefore, when we use spot instances with MArk, we reserve a few stopped burstable instances as cold standbys. Once MArk receives interruption notices, it resumes the corresponding amount of burstable instances to handle the transient requests until the regular spot instances capacity is back to normal, after which those burstable instances are stopped.

Lambda Cold Start. Another potential challenge posed to MArk is the *cold start* issues of Lambda instances [83]. That is, every time a new Lambda instance is launched, it needs to load the ML model, framework library and codes in memory. These operations significantly increases the inference delay. Nevertheless, cold starts only occur when the request rate exceeds the concurrency, measured by the number of currently available lambda instances [41], [85]. Existing benchmarking experiments show that a Lambda instance is recycled after it stays inactive for 45 to 60 minutes [40]. To understand the potential impact that cold starts make to MArk, we evaluate the cold start rate with our workloads described in Section 5. We confirm that in realistic settings, cloud providers' keep-warm strategy can keep cold starts within a tolerable threshold. In our measurements, with more than 3 million requests, the cold start rate never exceeds 0.23 percent. We therefore conclude that the latency impact of Lambda cold starts is limited. The cost impact is also negligible. Our profiling with relatively large computer vision models shows that with $1 we can spin up 7K inception-v3 Lambda instances, which are capable of serving more than 20K requests per second. We therefore do not consider the cost impact of Lambda cold starts in Algorithm 1.

Despite the limited impacts of Lambda cold start in ML serving, our implementation employs strategical concurrency warm-up to further amortize its impact. When a potential Lambda request surge is expected, such as spot interruptions and unexpected workload surges, MArk sends concurrent pings to Lambda to warm up more instances as described in [41], such process can be easily adopted with opensource projects [44]. Furthermore, cloud platforms are actively working on resolving the cold start issues. AWS now offers Provisioned Concurrency for Lambda [28], developers can directly provision the number of function instances in advance. Azure also introduced a similar feature in its newly introduced premium Function subscription [33].

5 Evaluation

We have prototyped the proposed MArk system and conducted extensive experimental evaluations on AWS to validate its effectiveness and robustness. We first compare the performance of MArk using on-demand instances and spot instances respectively with the premier industrial ML platform SageMaker against production traces from Twitter. To ensure MArk's performance does not mainly rely on prediction accuracy, we then examine whether MArk is able to maintain its advantage under unpredictable, highly bursty workload. After that, we run a few microbenchmarks to demonstrate the robustness of MArk in terms of handling spot interruptions, and the ability to handle unexpected demand surges.

TABLE 4
ML Models and Frameworks Used in Evaluation

Model	Type	Framework	Size
Inception-v3	Image Classification	Tensorflow Serving	45 MB
NASNet	Image Classification	Keras	343 MB
LSTM-ptb	Language Modeling	MXNet Model Server	16 MB
OpenNMT-ende	Machine Translation	Tensorflow Serving	330 MB

5.1 Evaluation Setup

MArk. We have prototyped MArk on top of Amazon EC2 and Lambda services in two versions, *MArk-ondemand* which only uses on-demand instances, and *MArk-spot* which uses spot instances with interruption-tolerant mechanism, i.e., using burstable servers for smooth transition during unexpected instance interruption (Section 4.5).

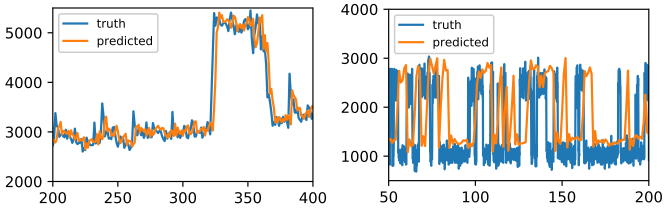
Testbed. We use AWS as the testbed for conducting extensive experiments. The types of instance used in our evaluation include all the `c5` and `m5` instances as examples of CPU instances and `p2.xlarge` instances as an example of GPU accelerators. In our experiments, we used up to 42 `c5` instances, 10 `m5` instances, and 12 `p2.xlarge` instances.

ML Models. We use four popular ML models that are of various sizes and cover diverse domains deployed in three popular ML serving software frameworks to evaluate MArk's performance, which are summarized in Table 4. To configure the batching of the ML models on EC2 instance, we performed lightweight profiling following the instructions detailed in Section 4.3. The optimal batching hyperparameters W_{batch} and N_{batch} for `p2.xlarge` instance found by our tuning algorithm outlined in Section 4.3 are 200 ms and 8 for Inception-v3, 750 ms and 16 for NASNet, 490 ms and 16 for OpenNMT-ende. For LSTM-ptb, we only performed experiments on CPU as MXNet Model Server does not support batching at the time of writing. For OpenNMT-ende on CPU instance, the optimal batching hyperparameter N_{batch} is found to be 2, and W_{batch} is set accordingly. For the other models on CPU instance, we do not use batching as it does not bring benefits (see Fig. 2).

SLO. Recall that the SLO requirement is specified as at least $SL_{\min}$ percent of requests must be served in $RT_{\max}$ time (Section 4.1). We set $SL_{\min}$ to 98 percent for all models, and set $RT_{\max}$ as 600, 1000, 100, and 1400 ms for Inception-v3, NASNet, LSTM-ptb, and OpenNMT-ende respectively.

Workload. In our evaluation, we drive the arrival process of ML workloads in two different ways. First, as there is no publicly available traces for ML serving, we synthesize ML requests based on the tweets traces from Twitter [27]. We believe that the Twitter traces serve as a good benchmark, as it represents a popular web service with highly dynamic load. The trace exhibits typically characteristics of ML inference workloads, containing recurring patterns (e.g., hour of the day, day of the week) as well as unpredictable load spikes (e.g., breaking news). In particular, the peak request rate in the traces is 4 times higher than the valley, a result of transient demand surges commonly found in industrial-scale web applications. Fig. 6a (a) illustrates a snapshot of the trace.

Second, to further evaluate the performance sensitivity of MArk w.r.t the workload, we synthesize random and bursty



(a) Twitter (b) MMPP

Fig. 6. Snapshots of the arrival process using Twitter and MMPP with the prediction results of LSTM-based algorithm.

ML request load using *Markov-Modulated Poisson process* (MMPP) [37], [45], [71]. The load generated by MMPP are highly unpredictable, as the occurrence and duration of demand surges are completely random, as shown in Fig. 6b.

In summary, we use the Twitter traces to evaluate how well MArk performs against synthesized real workload that can be largely predicted. Using MMPP-generated workload, we stress test MArk's performance in the presence of frequent, unpredictable load spikes.

Baseline. As discussed in Section 2.1, existing ML inference systems mainly focus on enabling inference or scheduling within pre-allocated private clusters instead of utilizing provisioned cloud resources as MArk does. Consequently, we use the state-of-the-practice SageMaker [18] as the baseline for evaluation. SageMaker is AWS's leading ML training and hosting system. SageMaker hosting employs AWS's new target tracking autoscaling policy [21], [23]. Given the dynamics in request arrival rate (i.e., the arrival rate can increase more than double in just a few minutes), to ensure service quality, we follow the AWS guidelines [21] and set the over-provisioning factor to 2 for SageMaker. We will show in Fig. 8 that even so the over-provisioning is still incapable of handling the volatile workload of the Twitter traces.

5.2 Macrobenchmarks

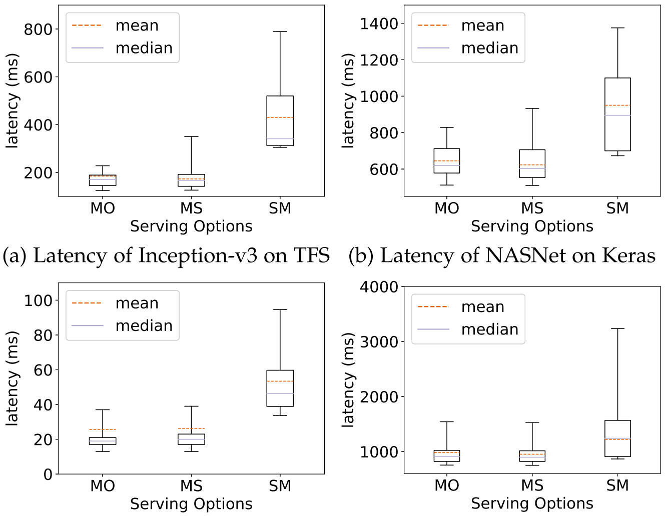
Workload Prediction. For Twitter traces, we use the data of the first 5 months to train the workload prediction model. For MMPP-generated arrival process, we use a period of 24-hour data for training. Fig. 6b demonstrates snapshots of the prediction results. We see that the prediction accuracy is in general good for the Twitter traces, yet unsatisfactory for the MMPP case. Since striving for the best workload prediction is NOT the focus of this paper, and we mainly use the LSTM-based algorithm as an example of the pluggable workload prediction component, we do not provide detailed evaluation of the prediction algorithm in the interest of space.

TABLE 5
Cost ($) Comparison of MArk-Ondemand (MO), MArk-Spot (MS), and SageMaker (SM) on 4 ML Models Using Twitter Workload

Setting	Inception-v3			NASNet		
	MO	MS	SM	MO	MS	SM
EC2	20.94	9.83	80.98	24.21	10.71	68.1
Lambda	1.34	3.2	NA	0.19	0.81	NA
Total	22.28	13.03	80.98	24.40	11.52	68.1
Setting	**LSTM-ptb**			**OpenNMT-ende**		
	MO	MS	SM	MO	MS	SM
EC2	6.17	2.24	14.9	27.54	10.79	87.1
Lambda	0	0.04	NA	0.12	0.33	NA
Total	6.17	2.28	14.9	27.66	11.12	87.1

Experimental Results Using Twitter Traces. We first compare MArk-ondemand, MArk-spot, and SageMaker on the ML models described in Section 5.1 by feeding the arrival rate extracted from Twitter traces. The experiments were performed on AWS spanning more than 8 hours each. We report two metrics: request latency in Fig. 7, and cost breakdown in Table 5. The request latency is measured as the time between request arriving at the serving system and getting response back, while the cost is the charge billed by AWS. The comparison results suggest that MArk can significantly reduce both the cost and latency compared with SageMaker. For cost reduction, compared with SageMaker, MArk-ondemand respectively achieves 3.63×, 2.79×, 2.41×, and 3.15× for the four ML models; MArk-spot achieves 6.21×, 5.91×, 6.64×, and 7.83×, respectively. For latency, MArk-ondemand achieves up to 57 percent reduction and MArk-spot achieves up to 60 percent reduction compared with SageMaker.

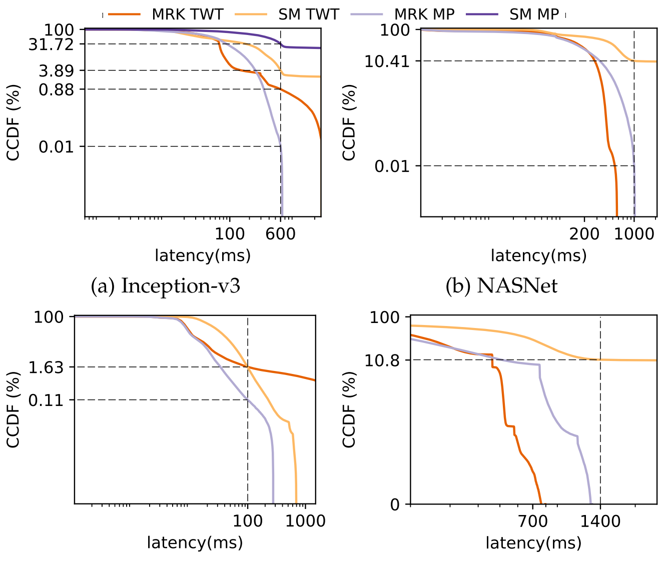
The latency advantage of MArk over SageMaker comes in three-fold. First, with appropriate batching configuration, GPU instances can reduce the overall latency by performing more efficient parallel computation. Second, the SLO-aware design of MArk helps reduce the queuing delay. In addition, the predictive scaling and SLO-awareness together form an efficient hybrid approach that enjoys the benefits in both proactive and reactive designs. It is worth pointing out the different performance behaviors between MArk-ondemand and MArk-spot. As shown in the latency box plots in Fig. 7, MArk-spot has longer latency tails, since more requests are handled by Lambda compared with MArk-ondemand, in case of interruptions. However, the average and median latencies of MArk-spot are usually the same or even better than MArk-ondemand. This is because in spot market, the performance-cost ratio is highly dynamic, which allows MArk-spot to opportunistically use large instances and GPU instances at cheaper price than on-demand, leading to better latency performance.



(a) Latency of Inception-v3 on TFS (b) Latency of NASNet on Keras

(c) Latency of LSTM-ptb on MMS (d) Latency of OpneNMT on TFS

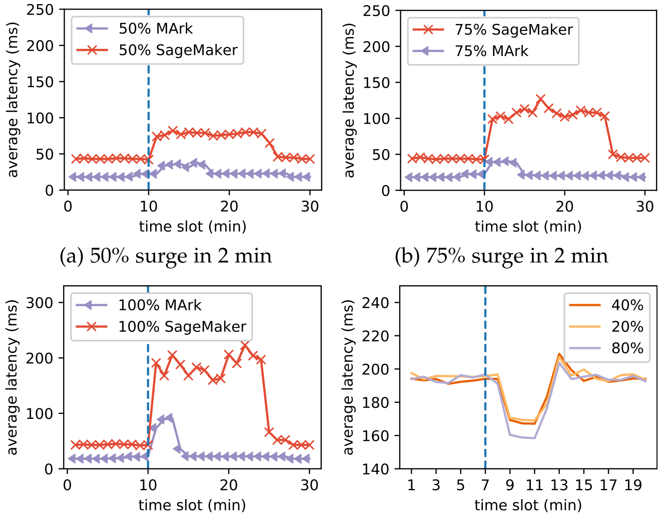
Fig. 7. Latency comparison of MArk-ondemand (MO), MArk-spot (MS), and SageMaker (SM) on 4 ML models using Twitter workload.



(a) Inception-v3 (b) NASNet

(c) LSTM-ptb (d) OpneNMT

Fig. 8. CCDF of latency comparison between MArk and SageMaker. RT_{max} is drawn as a black dashed vertical line (the black dashed horizontal line shows the corresponding CCDF value of RT_{max}). MRK and SM represents MArk and SageMaker, while TWT and MP represents Twitter and MMPP workload, respectively.



(a) 50% surge in 2 min (b) 75% surge in 2 min

(c) 100% surge in 2 min (d) spot instance interruption

Fig. 9. Microbenchmark results. (a), (b), (c): The latency change comparison during unexpected demand surge between MArk and SageMaker, where the surge starts at the 11th min shown by the dashed line. (d): The latency change when different percentages of spot instances are interrupted in MArk-spot, where the interruption notice is received at the 7th min.

We have also performed a case study of SLO compliance and report the *Complementary Cumulative Distribution Function* (CCDF) of request latency in Fig. 8. As expected, MArk managed to maintain its compliance with SLO requirements, thanks to the SLO-aware design. SageMaker, on the other hand, is SLO-oblivious, so the queuing delay adds up during high arrival periods, and the SLO is violated.

Experimental Results Using MMPP-Generated Load. Next we evaluate MArk using the more challenging, less predictable MMPP workload. We still use the same four ML models, and each experiment lasts about 4 hours on AWS. In the interest of space, we only demonstrate the SLO compliance results in Fig. 8. Fig. 8a shows that the SLO compliance of SageMaker is significantly degraded from Twitter case to MMPP case due to the much more dynamic and bursty behaviors in MMPP. However, MArk can still meet the SLO requirements even when the workload is highly dynamic and unpredictable, thanks to the SLO Monitor that can detect the failure of proactive prediction and timely add backup machines based on the feedback control algorithm. Note that we only evaluated SageMaker with MMPP-driven arrival process on Inception-v3 model as it is too expensive for us to run all of them. However, given the SLO-oblivious nature of SageMaker, we expect the behavior would be similar.

Sources of Improvements. The cost reduction of MArk comes from several aspects. First, predictive scaling together with Lambda services bring a more judicious over-provisioning design that can reduce the cost. For instance, in the LSTM-ptb model experiments, only CPU instances are used, thus the source of the 2× cost reduction in MArk-ondemand over SageMaker mainly comes from the reduced over-provisioning. Note that although Lambda service used by MArk is expensive in price, the cost of Lambda can be well justified by enabling more judicious over-provisioning. Second, exploiting batching (especially on GPU instances) further reduces the cost during high demand as the efficiency of computing is improved. For instance, the OpenNMT results demonstrate the highest cost reduction as they benefit the most from batching compared with CPU-only LSTM-ptb (see Fig. 2d). Third, employing interruptible instances further brings down the cost. MArk-spot reduces the cost by enjoying the spot market discounts compared with its on-demand counterpart. It is worth mentioning that a more accurate workload prediction may improve the cost reduction of MArk, but even in the worst case scenario where workloads are unpredictable (like MMPP), MArk can fall-back to an effective reactive scheduling thanks to the capability of using FaaS for prompt handover and the SLO monitor. Therefore, MArk provides both SLO guarantee and substantial cost savings regardless of the workloads and the prediction algorithm employed.

5.3 Microbenchmarks

In this section, we evaluate the robustness of MArk by taking a closer look at how MArk handles unexpected demand surges and spot interruptions.

Robustness Against Unexpected Surge. MArk harvests performance and cost benefits by using a judicious over-provisioning scheme. One important question is whether MArk can handle unexpected demand surges well in the presence of unforeseeable flash crowds or poor workload prediction accuracy. To answer this question, we increase the request rate for LSTM-ptb serving by 50, 75, and 100 percent in 2 minutes and compare the latency over time between MArk and SageMaker in Figs. 9a, 9b, and 9c.[4] Since the surge is unpredictable, both MArk and SageMaker handle it reactively. The results suggest that MArk acts faster and effectively than SageMaker during the unforeseeable surge, i.e., the increased latency period and amount are much smaller,

[4] Given that we only compare latency here, we show the results of MArk-spot as the latency results of MArk-ondemand can only be better.

thanks to the Lambda-based fallback mechanism, which can immediately take over and cap the latency to prevent queue building up like in SageMaker. In addition, MArk's SLO Monitor can detect the SLO violations and issue backup instance requests right away to adapt to the new arrival rate, while SageMaker is only able to react in the next scaling cycle.

Robustness Against Spot Interruption. MArk-spot utilizes spot instances to reduce the cost. However, the interruption of spot instance can cause performance degradation if not handled properly. We evaluate MArk-spot by zooming in the interruption handling periods under different interruption ratio of instances. We launched a 20-instance Inception-v3 cluster, and manually interrupted 20, 40, and 80 percent of the instances respectively. Fig. 9d illustrates the latency change during the interruption. The interruption happens at the 7th minute (vertical dashed line), and MArk resumes `t2` instances as transient resources upon receiving interruption notice. The proactive controller then adjusts the provisioning plan and requests new instances. At the 13th minute new spot instances are ready, and the latency goes back to normal. The average latency drops during transient period because burstable `t2` instances can have temporal boosted performance as discussed in Section 3.2. The short latency bump at the 13th minute is due to the switching overhead (i.e., warm up of new instances).

To sum up, the results above confirm that MArk can handle unexpected surge and spot interruption robustly.

6 Discussion

Cloud Platform. Our measurements and evaluations in this paper are mainly based on AWS. However, the main design of MArk can be generally extended to the other major cloud platforms, as they offer similar IaaS and FaaS services, as well as flexible pricing models. Having said that, some hyperparameters used in the algorithm are platform-dependent, and must be re-tuned. Also, we have not considered reserved instances, as they require a long-term usage commitment. We believe their usage will bring down the cost of serving stable inference demands in a long run. We will leave it as a future work.

VM Selection. MArk conducts profiling experiments to identify the most profitable VM instance to use. To speed up the VM selection process, intelligent methods like Bayesian optimization [87] and analytical modeling [53] can be employed.

Large Models. In AWS (and other cloud platforms), a Lambda instance can have no more than 3GB memory, which may not be sufficient to hold large deep learning models. A possible solution would be utilizing serverless workflow services like AWS Step Function. Another possible solution goes to distributed inference under the model parallel scheme. We will leave further explorations as a future work.

Hardware Accelerator. We have used the most common ML accelerator GPU as an example of utilizing hardware accelerators. We believe that the same batching formulation can be applied to other accelerators (e.g., FPGA) as they benefit from batching in a similar manner.

MArk's architecture requires a centralized master machine to make provisioning decisions. One natural concern is that such a centralized design might have poor scalability and be vulnerable to the single point of failure. Fortunately, as MArk's master node only performs lightweight computations, the potential scalability and reliability problems can be easily addressed with mature industrial solutions such as Zookeeper [56], or by deploying the master node on a dedicated cloud server instead of a VM.

7 Concluding Remark

In this paper, we have conducted a systematic study of serving ML models on cloud and concluded that combining FaaS and IaaS can achieve scalable ML serving with low over-provisioning cost. Driven by the unique characteristics of ML model serving, we have proposed MArk, a cost-effective and SLO-aware ML serving system. We have prototyped MArk on AWS and showed that compared with the premier autoscaling ML platform SageMaker, MArk yields significant cost reduction (up to $7.8\times$) while complying with the SLO requirements with even better latency performance.

Acknowledgments

This work was supported in part by RGC ECS Grant 26213818, US National Science Foundation Grant CCF-1756013, and IIS-1838024 (using resources provided by AWS as part of the NSF BIGDATA Program). The work of Chengliang Zhang and Minchen Yu was supported by the Hong Kong PhD Fellowship Scheme and the Huawei PhD Fellowship Scheme, respectively. Part of this article has appeared in [88]. This new version contains substantial revision with further improvements, discussions and evaluations.

References

[1] Amazon EC2, 2018. [Online]. Available: https://aws.amazon.com/ec2/
[2] Amazon ECS, 2018. [Online]. Available: https://aws.amazon.com/ecs/
[3] AWS Lambda, 2018. [Online]. Available: https://aws.amazon.com/lambda/
[4] Docker, 2018. [Online]. Available: https://www.docker.com
[5] Google Cloud Functions, 2018. [Online]. Available: https://cloud.google.com/functions/
[6] Google Kubernetes Engine, 2018. [Online]. Available: https://cloud.google.com/kubernetes-engine/
[7] PredictionIO, 2018. [Online]. Available: https://predictionio.apache.org
[8] RedisML, 2018. [Online]. Available: https://github.com/RedisLabsModules/redis-ml
[9] AWS Elastic Inference, 2019. [Online]. Available: https://aws.amazon.com/machine-learning/elastic-inference/
[10] AWS Inferentia, 2019. [Online]. Available: https://aws.amazon.com/machine-learning/inferentia/
[11] Edge TPU - Run Inference at Edge, 2019. [Online]. Available: https://cloud.google.com/edge-tpu/
[12] Mxnet inference benchmark, 2019. [Online]. Available: https://github.com/apache/incubator-mxnet/blob/master/example/image-classification/benchmark_score.py
[13] NPU from Cambricon, 2019. [Online]. Available: http://www.cambricon.com/index.php?m=content&c=index&a=lists&catid=15
[14] A. Ali-Eldin, M. Kihl, J. Tordsson, and E. Elmroth, "Efficient provisioning of bursty scientific workloads on the cloud using adaptive elasticity control," in *Proc. 3rd ACM Workshop Sci. Cloud Comput.*, 2012, pp. 31–40.
[15] A. Ali-Eldin, J. Tordsson, and E. Elmroth, "An adaptive hybrid elasticity controller for cloud infrastructures," in *Proc. IEEE Netw. Operations Manag. Symp.*, 2012, pp. 204–212.

[16] Amazon, "Amazon web services," 2018. [Online]. Available: https://aws.amazon.com/

[17] Amazon, "AWS autoscaling," 2018. [Online]. Available: https://aws.amazon.com/autoscaling/

[18] Amazon, "Build, train, and deploy machine learning models at scale," 2018. [Online]. Available: https://aws.amazon.com/sagemaker/

[19] Amazon, "Configuring Lambda functions," 2018. [Online]. Available: https://docs.aws.amazon.com/lambda/latest/dg/resource-model.html

[20] Amazon, "Dynamic scaling for Amazon EC2 auto scaling," 2018. [Online]. Available: https://amzn.to/2W2jvhc

[21] Amazon, "Load testing for variant automatic scaling," 2018. [Online]. Available: https://docs.aws.amazon.com/sagemaker/latest/dg/endpoint-scaling-loadtest.html

[22] Amazon, "New Amazon EC2 spot pricing model: Simplified purchasing without bidding and fewer interruptions," 2018. [Online]. Available: https://aws.amazon.com/blogs/compute/new-amazon-ec2-spot-pricing/

[23] Amazon, "Target tracking scaling policies for Amazon EC2 auto scaling," 2018. [Online]. Available: https://docs.aws.amazon.com/autoscaling/ec2/userguide/as-scaling-target-tracking.html

[24] Amazon, "Use elastic inference with MXNet," 2019. [Online]. Available: https://docs.aws.amazon.com/dlami/latest/devguide/tutorial-mxnet-elastic-inference.html

[25] AnandTech, "Intel shipping Nervana neural network processor," 2017. [Online]. Available: https://bit.ly/2YYOMjH

[26] L. Aniello, S. Bonomi, F. Lombardi, A. Zelli, and R. Baldoni, "An architecture for automatic scaling of replicated services," in *Proc. Int. Conf. Netw. Syst.*, 2014, pp. 122–137.

[27] ArchiveTeam. Twitter streaming traces, 2017. [Online]. Available: https://bit.ly/2VOpWSP

[28] AWS, "New for AWS lambda – predictable start-up times with provisioned concurrency," 2019. [Online]. Available: https://go.aws/3fPQnPY

[29] AWS, "Amazon EC2 reserved instances," 2018. [Online]. Available: https://aws.amazon.com/ec2/pricing/reserved-instances/

[30] AWS, "Burstable performance instances," 2018. [Online]. Available: https://amzn.to/2APg4hG

[31] AWS, "Right sizing: Provisioning instances to match workloads," 2018. [Online]. Available: https://amzn.to/2VdIiK9

[32] Awslabs, "MXNet model server," 2018. [Online]. Available: https://github.com/awslabs/mxnet-model-server

[33] Azure, "Announcing the azure functions premium plan for enterprise serverless workloads," 2019. [Online]. Available: https://bit.ly/2B1crIl

[34] E. Barrett, E. Howley, and J. Duggan, "Applying reinforcement learning towards automating resource allocation and application scalability in the cloud," *Concurrency Comput., Practice Experience*, vol. 25, no. 12, pp. 1656–1674, 2013.

[35] J. Bergstra *et al.*, "Theano: Deep learning on GPUs with Python," in *Proc. Neural Inf. Process. Syst. Big Learn. Workshop*, 2011, pp. 1–4.

[36] P. Bodík, R. Griffith, C. Sutton, A. Fox, M. I. Jordan, and D. A. Patterson, "Statistical machine learning makes automatic control practical for internet datacenters," in *Proc. Conf. Hot Topics Cloud Comput.*, 2009, Art. no. 12.

[37] G. Casale, E. Z. Zhang, and E. Smirni, "Trace data characterization and fitting for Markov modeling," *Perform. Eval.*, vol. 67, no. 2, pp 61–79, 2010.

[38] F. Chollet *et al.*, "Keras: Deep learning library for Theano and TensorFlow," 2015. [Online]. Available: https://keras.io

[39] D. Crankshaw, X. Wang, G. Zhou, M. J. Franklin, J. E. Gonzalez, and I. Stoica, "Clipper: A low-latency online prediction serving system," in *Proc. 14th USENIX Conf. Netw. Syst. Des. Implementation*, 2017, pp. 613–627.

[40] Y. Cui, "How long does AWS Lambda keep your idle functions around before a cold start?" 2018. [Online]. Available: https://bit.ly/2tb7bLJ

[41] Y. Cui, "I'm afraid you're thinking about AWS lambda cold starts all wrong," 2018. [Online]. Available: https://bit.ly/2Q1rrcr

[42] R. P. Doyle, J. S. Chase, O. M. Asad, W. Jin, and A. Vahdat, "Model-based resource provisioning in a web service utility," in *Proc. USENIX Symp. Internet Technol. Syst.*, 2003, pp. 5–5.

[43] W. Fang, Z. Lu, J. Wu, and Z. Cao, "RPPS: A novel resource prediction and provisioning scheme in cloud data center," in *Proc. IEEE Int. Conf. Serv. Comput.*, 2012, pp. 609–616.

[44] Fidel, "Serverless warmup plugin," [Online]. Available: https://github.com/FidelLimited/serverless-plugin-warmup/.

[45] W. Fischer and K. Meier-Hellstern, "The Markov-modulated Poisson process (MMPP) cookbook," *Perform. Eval.*, vol. 18, no. 2, pp. 149–171, 1993.

[46] F. A. Gers, J. Schmidhuber, and F. Cummins, "Learning to forget: Continual prediction with LSTM," in *Proc. 9th Int. Conf. Artif. Neural Netw.*, 1999, pp. 850–855.

[47] Google, "Google cloud," 2018. [Online]. Available: https://cloud.google.com/

[48] Google, "Google cloud autoscaling," 2018. [Online]. Available: https://cloud.google.com/compute/docs/autoscaler/

[49] Google, "Kubernetes horizontal scaling," 2018. [Online]. Available: https://kubernetes.io/docs/tasks/run-application/horizontal-pod-autoscale/

[50] Google, "Cloud TPU performance guide," 2019. [Online]. Available: https://cloud.google.com/tpu/docs/performance-guide

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

[52] R. Han, M. M. Ghanem, L. Guo, Y. Guo, and M. Osmond, "Enabling cost-aware and adaptive elasticity of multi-tier cloud applications," *Future Gener. Comput. Syst.*, vol. 32, pp. 82–98, 2014.

[53] A. Harlap, A. Tumanov, A. Chung, G. R. Ganger, and P. B. Gibbons, "Proteus: Agile ML elasticity through tiered reliability in dynamic resource markets," in *Proc. 12th Eur. Conf. Comput. Syst.*, 2017, pp. 589–604.

[54] K. He, X. Zhang, S. Ren, and J. Sun, "Deep residual learning for image recognition," in *Proc. IEEE Conf. Comput. Vis. Pattern Recognit.*, 2016, pp. 770–778.

[55] X. He, P. Shenoy, R. Sitaraman, and D. Irwin, "Cutting the cost of hosting online services using cloud spot markets," in *Proc. 24th Int. Symp. High-Perform. Parallel Distrib. Comput.*, 2015, pp. 207–218.

[56] P. Hunt, M. Konar, F. P. Junqueira, and B. Reed, "ZooKeeper: Wait-free coordination for internet-scale systems," in *Proc. USENIX Conf. USENIX Annu. Tech. Conf.*, 2010, Art. no. 11.

[57] M. Jeon, S. Venkataraman, J. Qian, A. Phanishayee, W. Xiao, and F. Yang, "Multi-tenant GPU clusters for deep learning workloads: Analysis and implications," 2018. [Online]. Available: https://bit.ly/2CdUZAU

[58] G. Klein, Y. Kim, Y. Deng, J. Senellart, and A. M. Rush, "OpenNMT: Open-source toolkit for neural machine translation," 2017, arXiv: 1701.02810.

[59] H. Lee, K. Satyam, and G. Fox, "Evaluation of production serverless computing environments," in *Proc. IEEE 11th Int. Conf. Cloud Comput.*, 2018, pp. 442–450.

[60] Y. Lee, A. Scolari, B.-G. Chun, M. D. Santambrogio, M. Weimer, and M. Interlandi, "PRETZEL: Opening the black box of machine learning prediction serving systems," in *Proc. 13th USENIX Conf. Operating Syst. Des. Implementation*, 2018, pp. 611–626.

[61] P. Leitner and J. Scheuner, "Bursting with possibilities: An empirical study of credit-based bursting cloud instance types," in *Proc. IEEE/ACM 8th Int. Conf. Utility Cloud Comput.*, 2015, pp. 227–236.

[62] S. Merity, N. S. Keskar, and R. Socher, "Regularizing and optimizing LSTM language models," 2017, *arXiv: 1708.02182*.

[63] Microsoft, "Microsoft Azure cloud computing platform & services," 2018. [Online]. Available: https://azure.microsoft.com/en-us/

[64] A. Y. Nikravesh, S. A. Ajila, and C.-H. Lung, "Towards an autonomic auto-scaling prediction system for cloud resource provisioning," in *Proc. IEEE 10th Int. Symp. Softw. Eng. Adaptive Self-Manag. Syst.*, 2015, pp. 35–45.

[65] NVIDIA, "NVIDIA TensorRT," 2018. [Online]. Available: https://developer.nvidia.com/tensorrt

[66] C. Olston *et al.*, "TensorFlow-Serving: Flexible, high-performance ML serving," 2017, *arXiv: 1712.06139*.

[67] Y. Peng, Y. Bao, Y. Chen, C. Wu, and C. Guo, "Optimus: An efficient dynamic resource scheduler for deep learning clusters," in *Proc. 13th EuroSys Conf.*, 2018, Art. no. 3.

[68] R. Prodan and V. Nae, "Prediction-based real-time resource provisioning for massively multiplayer online games," *Future Gener. Comput. Syst.*, vol. 25, no. 7, pp. 785–793, 2009.

[69] C. Qu, R. N. Calheiros, and R. Buyya, "A reliable and cost-efficient auto-scaling system for web applications using heterogeneous spot instances," *J. Netw. Comput. Appl.*, vol. 65, pp. 167–180, 2016.

[70] C. Qu, R. N. Calheiros, and R. Buyya, "Auto-scaling web applications in clouds: A taxonomy and survey," *ACM Comput. Surv.*, vol. 51, no. 4, 2018, Art. no. 73.

[71] A. Rajabi and J. W. Wong, “MMPP characterization of web application traffic,” in *Proc. IEEE 20th Int. Symp. Model. Anal. Simul. Comput. Telecommun. Syst.*, 2012, pp. 107–114.

[72] N. Roy, A. Dubey, and A. Gokhale, “Efficient autoscaling in the cloud using predictive models for workload forecasting,” in *Proc. IEEE 4th Int. Conf. Cloud Comput.*, 2011, pp. 500–507.

[73] J. Sanders and E. Kandrot, *CUDA by Example: An Introduction to General-Purpose GPU Programming*. MA, USA: Addison-Wesley, 2010.

[74] P. Sharma, S. Lee, T. Guo, D. Irwin, and P. Shenoy, “SpotCheck: Designing a derivative IaaS cloud on the spot market,” in *Proc. 10th Eur. Conf. Comput. Syst.*, 2015, pp. 1–15.

[75] H. Shen *et al.*, “Nexus: A GPU cluster engine for accelerating DNN-based video analysis,” in *Proc. 27th ACM Symp. Operating Syst. Princ.*, 2019, pp. 322–337.

[76] X. Shi, Z. Chen, H. Wang, D.-Y. Yeung, W.-K. Wong, and W.-C. Woo, “Convolutional LSTM network: A machine learning approach for precipitation nowcasting,” in *Proc. 28th Int. Conf. Neural Inf. Process. Syst.*, 2015, pp. 802–810.

[77] B. Song, Y. Yu, Y. Zhou, Z. Wang, and S. Du, “Host load prediction with long short-term memory in cloud computing,” *J. Supercomput.*, vol. 74, pp. 6554–6568, 2018.

[78] C. Szegedy, S. Ioffe, V. Vanhoucke, and A. A. Alemi, “Inception-v4, inception-ResNet and the impact of residual connections on learning,” in *Proc. 31st AAAI Conf. Artif. Intell.*, 2017, Art. no. 12.

[79] C. Szegedy, V. Vanhoucke, S. Ioffe, J. Shlens, and Z. Wojna, “Rethinking the inception architecture for computer vision,” in *Proc. IEEE Conf. Comput. Vis. Pattern Recognit.*, 2016, pp. 2818–2826.

[80] TensorFlow, “TensorFlow Serving batching guide,” 2018. [Online]. Available: https://bit.ly/2VOpb9O

[81] Z. Tu, M. Li, and J. Lin, “Pay-per-request deployment of neural network models using serverless architectures,” in *Proc. Conf. North Amer. Chapter Assoc. Comput. Linguistics: Demonstrations*, 2018, pp. 6–10.

[82] B. Urgaonkar, P. Shenoy, A. Chandra, P. Goyal, and T. Wood, “Agile dynamic provisioning of multi-tier internet applications,” *ACM Trans. Auton. Adaptive Syst.*, vol. 3, no. 1, 2008, Art. no. 1.

[83] C. Wang, B. Urgaonkar, A. Gupta, G. Kesidis, and Q. Liang, “Exploiting spot and burstable instances for improving the cost-efficacy of in-memory caches on the public cloud,” in *Proc. 12th Eur. Conf. Comput. Syst.*, 2017, pp. 620–634.

[84] W. Wang *et al.*, “Rafiki: Machine learning as an analytics service system,” 2018, *arXiv: 1804.06087*.

[85] F. Yan, L. Ren, D. J. Dubois, G. Casale, J. Wen, and E. Smirni, “How to supercharge the Amazon T2: Observations and suggestions,” in *Proc. IEEE 10th Int. Conf. Cloud Comput.*, 2017, pp. 278–285.

[86] F. Yan, O. Ruwase, Y. He, and E. Smirni, “SERF: Efficient scheduling for fast deep neural network serving via judicious parallelism,” in *Proc. Int. Conf. High Perform. Comput. Netw. Storage Anal.*, 2016, pp. 300–311.

[87] J. Yi, C. Zhang, W. Wang, C. Li, and F. Yan, “Not all explorations are equal: Harnessing heterogeneous profiling cost for efficient MLaaS training,” in *Proc. 34th IEEE Int. Parallel Distrib. Process. Symp.*, 2020, pp. 1–10.

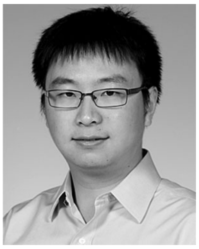
[88] C. Zhang, M. Yu, W. Wang, and F. Yan, “MArk: Exploiting cloud services for cost-effective, SLO-aware machine learning inference serving,” in *Proc. USENIX Conf. Usenix Annu. Tech. Conf.*, 2019, pp. 1049–1062.

[89] H. Zhang, L. Stafman, A. Or, and M. J. Freedman, “SLAQ: Quality-driven scheduling for distributed machine learning,” in *Proc. ACM Symp. Cloud Comput.*, 2017, pp. 390–404.

[90] B. Zoph, V. Vasudevan, J. Shlens, and Q. V. Le, “Learning transferable architectures for scalable image recognition,” 2017, *arXiv: 1707.07012*.

Chengliang Zhang (Student Member, IEEE) received the BEng degree from the School of Software, Harbin Institute of Technology, Harbin, China. He is currently working toward the PhD degree with the Department of Computer Science and Engineering, Hong Kong University of Science and Technology, Hong Kong, since 2016. His research interests cover cloud computing and distributed systems. He currently focuses on system design and performance optimization issues in distributed machine learning and data analytics systems.

Minchen Yu (Student Member, IEEE) received the BEng degree in software engineering from Nanjing University, Nanjing, China, in 2018. He is currently working toward the PhD degree with the Department of Computer Science and Engineering, Hong Kong University of Science and Technology, Hong Kong. His research interests include machine learning systems and serverless computing. He currently focuses on the application and optimization of serverless cloud.



Wei Wang (Member, IEEE) received the BEng (hons.) and MEng degrees from Shanghai Jiao Tong University, Shanghai, China, and the PhD degree from the University of Toronto, Toronto, Canada, in 2015, all with the Department of Electrical and Computer Engineering. He is an assistant professor with the Department of Computer Science and Engineering, Hong Kong University of Science and Technology (HKUST). He is also affiliated with HKUST Big Data Institute. His research interests cover the broad area of distributed systems, with special emphasis on big data and machine learning systems, cloud computing, and computer networks in general. He was a recipient of the prestigious Chinese Government Award for Outstanding PhD Students Abroad in 2015 and the Best Paper Runner-up Award at USENIX ICAC 2013. He was recognized as the distinguished TPC member of IEEE INFOCOM 2018–20.

Feng Yan (Member, IEEE) received the BS degree in computer science from Northeastern University, Boston, Massachusetts, and the MS and PhD degrees in computer science from the College of William and Mary, Williamsburg, Virginia, in 2011 and 2016, respectively. He is an assistant professor with the Department of Computer Science and Engineering, University of Nevada, Reno. He has a broad interest in big data and system areas. His current research focus includes machine learning, cloud/edge/fog computing, high performance computing, storage, and cross-disciplinary topics among them and others. He worked with Microsoft Research (2014–2015) and HP Labs (2013–2014). He was a recipient of the Best Paper Award at CLOUD 2019 and the Best Student Paper Award at IEEE CLOUD 2018.