MArk: Exploiting Cloud Services for Cost-Effective, SLO-Aware Machine Learning Inference Serving

Chengliang Zhang   Minchen Yu   Wei Wang
HKUST
{czhangbn, myuaj, weiwa}@cse.ust.hk

Feng Yan
University of Nevada, Reno
fyan@unr.edu

Abstract

The advances of Machine Learning (ML) have sparked a growing demand of ML-as-a-Service: developers train ML models and publish them in the cloud as online services to provide low-latency inference at scale. The key challenge of ML model serving is to meet the response-time Service-Level Objectives (SLOs) of inference workloads while minimizing the serving cost. In this paper, we tackle the dual challenge of SLO compliance and cost effectiveness with MArk (Model Ark), a general-purpose inference serving system built in Amazon Web Services (AWS). MArk employs three design choices tailor-made for 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 for inference serving, 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 the occasional load spikes that are hard to predict. We evaluated the performance of MArk using several state-of-the-art ML models trained in popular frameworks including 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.

1   Introduction

Driven by the sustained advances of Machine Learning (ML), the past few years have seen a surging demand of ML-as-a-Service (MLaaS). A typical workflow of MLaaS covers the two phases of ML in the cloud: *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 [43, 56, 75]. In the inference phase, the trained models are published as *online services* in data center or cloud and can be queried by end users with new input. The service makes prediction decisions (inference) for a given input using the trained model [30] (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* on 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) [41], such as at least 98% 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 [11], Google [37], and Microsoft [52] offer a rich selection of service provisioning options, ranging from VMs and containers to the emerging serverless functions. For each provisioning option, there is a large configuration space (e.g., CPU, memory, and hardware accelerators) coupled with diverse pricing models offering tradeoffs between service guarantees and cost savings (e.g., on-demand and spot instances [17]). A key challenge of provisioning model serving in the cloud is: how should 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 guideline given by the cloud providers, nor has it been studied in the prior work [10, 25, 42, 45, 58, 59, 63, 70] which mainly targets at general workload. To bridge this gap, we perform extensive measurement studies of inference serving in AWS [11] and Google Cloud [37] by means of VMs (IaaS), containers (CaaS), and serverless functions (FaaS). We briefly summarize three key findings as follows.

First, our measurements suggest that among the three options, IaaS offers the best performance-cost ratio for inference serving, but it incurs long instance provisioning latency and is hence unable to quickly adapt to the changing workload. CaaS suffers from a similar problem as IaaS (though less

severe) with worse performance-cost ratio. Compared to IaaS and CaaS, FaaS scales much faster but is the most expensive.

Second, inference serving can gain significant benefits from *batching* when performed using costly hardware accelerators (e.g., GPU and TPU). Nevertheless, 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. Also, many ML models, especially deep learning, have *deterministic inference time* [41, 74]—they take fixed-size input vectors and have input-independent control flows. This also brings an opportunity for better resource planing and latency control.

Motivated by these observations, in this paper, we propose MArk (Model Ark), a low-latency, cost-effective inference serving system in the public cloud. MArk takes use of the unique characteristics of ML model serving while also addressing the distinctive challenges posed by it. 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 employing FaaS to quickly fill the service gap when the system is undergoing horizontal/vertical scaling. MArk uses *predictive scaling* to hide 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 bring down the cost, MArk also supports the use of the 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 [11] with pluggable backend model servers supporting a range of ML frameworks such as Tensorflow Serving [55], MXNet Model Server [24], and customized Keras [29] server with Theano [26] backend. We have evaluated MArk on AWS using several state-of-the-art ML models for image recognition, language modeling, and machine translation: Inception-V3 [67], NASNet [76], LSTM-ptb [51], and OpenNMT [47]. The results show that MArk yields up to 7.8× cost reduction while achieving comparable or even better latency compared to the state-of-the-practice solution SageMaker [13], and also comply 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, 24, 30, 55, 72]. These systems place the trained models in *containers* and handle model inference requests through REST APIs. For example, systems like Clipper [30], Rafiki [72], and MXNet Model Server [24] host each model in a separate Docker [4] container to ensure process isolation; TensorFlow Serving [55] 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 [30]. The recently proposed *white box* model serving [49] 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 [41] 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, where the cloud provisioning options (e.g., IaaS, CaaS, FaaS) and their pricing models are not relevant. Amazon's SageMaker [13] 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 [11].

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 [59] 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 [12, 13] and Kubernetes in Google Cloud [38, 39]. These systems perform scaling following some customized rules such as "adding 2 instances if CPU

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 [33]	Feedback control	Private cluster	✓	✓	×	×
Ali-Eldin et al. [9]	Predictive	IaaS	×	×	×	×
Barrett et al. [25]	Predictive	IaaS	×	×	×	×
Urgaonkar et al. [70]	Predictive	IaaS	✓	×	×	×
Han et al. [42]	Predictive	IaaS	✓	×	×	×
Qu et al. [58]	Feedback control	IaaS	×	✓	✓	×
SpotCheck [63]	–	IaaS	×	✓	✓	×
He et al. [45]	–	IaaS	×	✓	✓	×
Swayam [41]	Predictive	Private cluster	✓	×	–	×
SageMaker [13]	Feedback control	IaaS	×	×	×	✓
MArk	Predictive	IaaS and FaaS	✓	✓	✓	✓

utilization reaches 70%," or tracking a target such as "maintaining 100 queries per minute per instance" [15].

Feedback control scaling makes no prediction about the future and is easy to implement. However, owing to its reactive nature, it incurs long instance provisioning delay when used to serve changing workload [59]. Over-provisioning is therefore needed in case of load spikes. For example, SageMaker recommends to start with 100% over-provisioning and adjust thereafter [16]. As ML model serving is often compute-intensive and requires costly CPU/GPU instances, solely relying on over-provisioning is economically not viable.

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 [27], autoregressive models [34, 61], and neural networks [19, 53, 57, 65]. Predictive scaling is often complemented with feedback control scaling, where the two approaches operate at different time scales [42, 70]. For example, predictive scaling can be used for resource planning at the time scale of hours or days, while reactive provisioning operates in minutes to respond to flash crowds or unexpected deviations from long-term behaviors [70].

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 in IaaS [9, 25, 42, 70]. 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 cost. In addition, many predictive autoscalers are unaware of the response-time SLOs and only provide best-effort services [9, 25].

2.3 Cloud Provisioning Services

Compared with private clusters, model serving in public clouds is more complex. Leading cloud platforms such as AWS [11], Google Cloud [37], and Microsoft Azure [52] offer a variety of provisioning services that can be used for model serving. We briefly review 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 vCPUs, memory, storage, network, and accelerators (e.g., GPU, TPU, and FPGA). Customers can then configure and deploy ML model serving softwares [24, 30, 68] 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 compared to the on-demand price, under the condition that a running spot instance can be *interrupted indefinitely* [17]. EC2 also allows customers to *reserve* an instance in a long term by making an upfront payment [21]. 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 [37].

Container-as-a-Service (CaaS). With CaaS, customers encapsulate services and implementations in containers (e.g., Docker images [4]), and run containers with specified resource configurations in the cloud, e.g., Amazon ECS [2] and Google Kubernetes Engine [6]. 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* in the cloud without provisioning or managing servers, e.g., AWS Lambda [3] and Google Cloud Functions [5]. In Lambda, customers can only specify the memory allocation for an instance, and pay for the total number of requests and the duration of compute time [3]. FaaS is particularly suitable for *stateless computations* and

Table 2: Cost ($) and average latency ($t$) of serving 1 million requests of three ML models in AWS. 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 vC-PUs and memory as `c5.large`; each Lambda instance has 3GB memory to achieve comparable latency with `c5.large`.

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

has recently been used to provision ML model serving [69].

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 in 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 [11] (§3.1-3.4), a leading cloud platform offering the most diversified service options. We validate the major results in Google Cloud [37] where possible (§3.5).

3.1 What service to use: IaaS, CaaS, or FaaS?

We choose three representative ML models, Inception-v3 [67], Inception-ResNet [66], and OpenNMT-ende [47], for common prediction tasks such as image classification and machine translation, and evaluate their peak inference performance with TensorFlow Serving [55]. Table 2 summarizes the cost and average latency of serving 1 million requests using AWS EC2 (IaaS), ECS (CaaS), and Lambda (FaaS), respectively.[2]

IaaS vs. CaaS. In EC2 [1], customers can choose among predefined instance types with fixed vCPU and memory allocation. In Table 2, we choose the cheapest compute-optimized instance `c5.large` as the reference, since it is proven to be the most cost-effective one in §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 per-request based, 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 [14]. For a fair comparison, we compare the Lambda cost of serving the same amount of requests `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 additional time is needed to load and set up large ML model serving atop standard overhead, as Microsoft suggests with their production traces [41]. 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 [48]. The cold start overhead of Lambda can be amortized by warming up [48]. Compared with EC2 and Lambda, ECS shows 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. IaaS is used as the primary serving option, while FaaS can provide 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 price.

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

3.2 IaaS: Can we use burstable instances?

IaaS providers typically categorize instances into families. Within a family, instances share the similar physical hardware but may have various 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 [71, 73]. Burstable instances provide a baseline level (10% 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% utilization for 30 minutes) [22, 23].

We profiled `t2` instances' performance for ML serving and show the results in Table 3. We see that the latency drops linearly with the CPU allocation but adding more memory does

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

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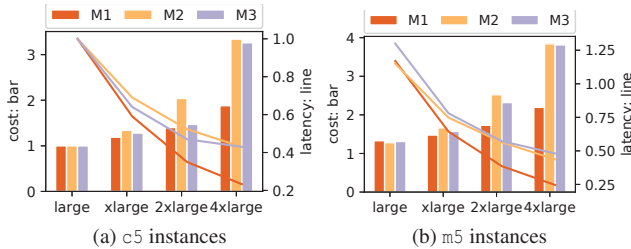
Figure 1: The latency (lines) and cost (bars) of serving 1 million model inference requests with `c5` and `m5` instances. M1, M2, and M3 respectively denote Inception-v3, Inception-ResNet, and OpenNMT-ende. The values are normalized by that of `c5.large` (182.5ms with $4.3 for M1; 389ms with $9.4 for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

not benefit inference performance. 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 a long time. This fatal disadvantage means that burstable instances are not for compute-intensive services [50].

Summary. Burstable instances are plausible for transient ML serving usage, but not 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 4GB of memory already does not bound the inference performance. In EC2, the configurations (vCPUs and memory) 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 have lower latency than `m5` instances because of more advanced CPU models, even though the latter have higher memory than the former. 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 preferable 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? In this subsection, we focus on GPU instances, as GPU is the most accessible and popular general-purpose ML accelerator. We will extend our study to TPUs in Google Cloud in §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 [54]. 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 [68]. 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 [30, 62].

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 too 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), while 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.

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 in-

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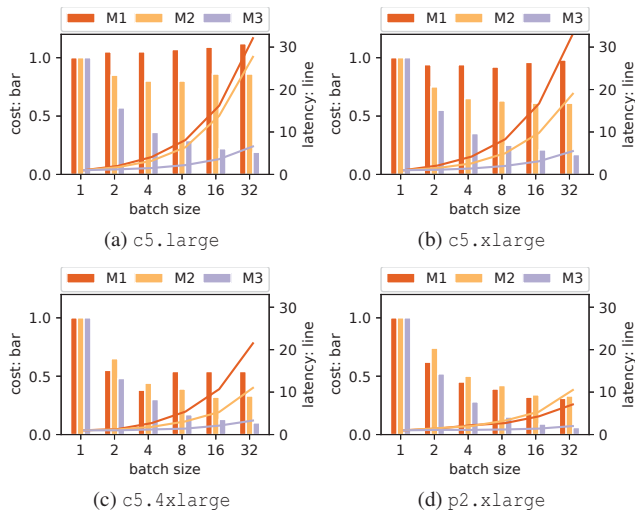
(c) c5.4xlarge   (d) p2.xlarge

Figure 2: The cost and batch latency of 1 million model inference with batching of various sizes. M1, M2, 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.

creased arbitrarily: increasing batch size leads to both longer queuing latency and batch inference latency [30]. We will further discuss the batching configuration in §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 [37] which offers similar service and pricing options as AWS, along with the Tensor Processing Unit (TPU), the state-of-the-art ML ASIC.

IaaS remains the best option. We first compare the cost and latency performance of ML serving using Google's IaaS, CaaS, and FaaS with the same workloads as in §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 2GB memory are 317ms and $3.70, respectively. In comparison, it takes 319ms and $4.17 using the cheapest CaaS instance `n1-standard-1` (1 vCPU and 3.75GB memory), and 527ms and $17.4 using Google Cloud Functions (FaaS) with 2GB memory.

Small instances win on 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 (§3.3): smaller instances offer higher performance-cost ratio than the bigger ones, though the latter leads to shorter latency. In particular, when serv-
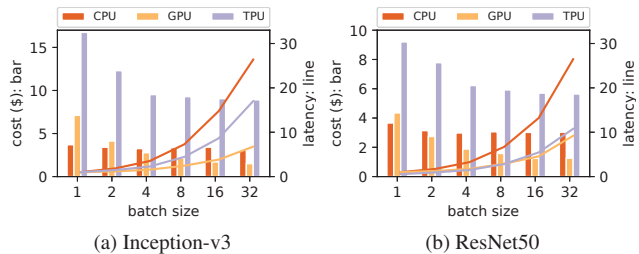

(a) Inception-v3   (b) ResNet50

Figure 3: The cost and batch latency of serving 1 million inference requests with various batch sizes. The batch latencies are normalized by the latency when there is no batching.

ing 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 (319ms), $7.82 (296ms), and $11.98 (227ms), respectively.

CPU, GPU, or TPU? Finally, we compare the cost and latency performance of using CPU, GPU, and TPU instances for ML serving with various batch sizes. We chose two popular image classification models, Inception-v3 and ResNet50 [44]. The results are shown in Fig. 3, where we used 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 (§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 [40]. 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 not suitable for real-time ML serving.

3.6 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 overprovisioning 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.
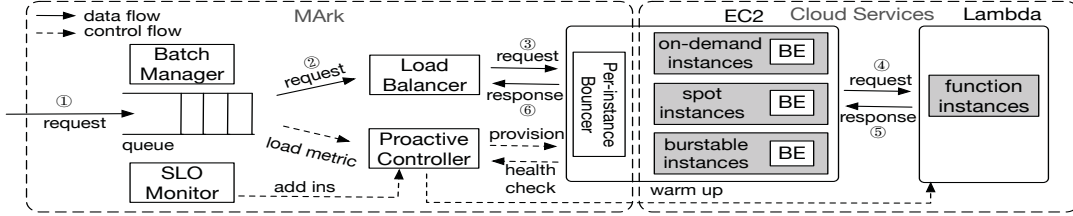
Figure 4: An overview of the MArk model serving system.

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.

4.1 Overview

Following our observations in §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. 4 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 §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 (§2.2). The controller then sends the launching and destroying requests to EC2 instances, on which custom service backends such as Tensorflow Serving [55] 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 surges. On each running EC2 instance, there is a *Bouncer* monitoring serving metrics and performing request admission control. If an incoming request cannot be served within a specified time RT_{max}, it will 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 §4.4.

SLO requirements. Following Swayam [41], 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 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 [27], autoregressive models [34, 61], and neural networks [19, 53, 57, 65]. As the accuracy of prediction depends on the underlying workload, there is no such a 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.

We have implemented a vanilla version of long short-term memory (LSTM) network [36] for multi-step workload prediction, as it is reported to give the state-of-the-art performance [64]. In our implementation, the prediction unit (time interval) is P_u, and the prediction window is P_w, meaning 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, we set the $[P_u, P_w, P_s]$ to $[1\text{min}, 60, 5\text{s}]$. Prediction unit is set to 1 minute, as EC2 charges at least 1 minute for new instances. Prediction window is set to 60 steps, since 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 [74]. MArk is designed to work for all ML serving workloads, so users can fine-tune this prediction algorithm or replace it with their own implementations for better prediction results.

4.3 Instance Provisioning and Batching

With workload prediction, we need to determine what and how many instances should be used to serve the requests. In general, this problem can be formulated as a compilation of queueing system [74], where instances of each type are modeled as an M/D/c queue with deterministic processing time and the predicted request arrival rate. However, as shown in [74], this problem has no closed-form solution even without considering request batching and instance pricing. Given this hardness result, we turn to a heuristic solution: instead of jointly considering batching and instance provisioning, we solve the two problems separately using heuristics.

Batching. Inspired by the adaptive batching in [30], 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. 4). We tune the two hyperparameters to meet the following two requirements: (1) No SLO requirements can be violated, meaning 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, denoted by μ_{nb}^*. We then gradually increase the batch size from 1 until at least one of the following constraints no longer holds, where b is the batch size, and T_b is the time needed to process a batch:

$$W_{\text{batch}} + T_b \leq RT_{\max},$$
$$W_{\text{batch}} + T_b \leq \frac{b}{\mu_{nb}^*}.$$

Now that we have the optimal batch size $N_{batch} \leftarrow b$ and the maximum processing rate μ^* under this configuration, together with their corresponding 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 launch overhead, while at the same time attaining high utilization of running instances.

We first introduce the notations. Suppose there are n types of instances that can be used for serving. At a given time t_0, let $R = \{r_1, r_2, \cdots, r_n\}$ be the set of running instances and $F = (F_1, \cdots, F_m)$ 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 launch overhead, i.e., cost due to the instance provisioning latency. 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, yet keeping using them avoids additional launch 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 launch overhead. The running instances at t_0 can be treated as special ones without launch overhead.

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
  for all 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($F,S,instance\_plan$)
  $C^{\text{sum}} \leftarrow$ total capacity of all instance i in $instance\_plan$
  for $t = 1$ to m **do**
    $\Lambda_t = F_t - C^{\text{sum}}$ ▷ Unfulfilled requests predicted at step t
  if $\Lambda_\tau \leq 0$ **then**   ▷ Planned capacity is enough at step τ
    return
  Find the largest e such that there are unfulfilled requests from steps τ 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 lowest per-request cost to cover unfilled requests from τ to e
  for all 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($F,S,instance\_plan$)

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, while destroying requests should be sent after a predefined cool-down period to ensure better service quality [59].

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 [59]. 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.

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 allows MArk to exploit the price discount of spot instances. 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, such a grace period 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? Lambda seems to be a choice, but it would take a toll on the latency and cost.

Our answer to this challenge is the burstable instance. As shown in §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 MArk faces is the *cold starts* in Lambda [71]. Every time a new Lambda instance is launched, it needs to load the ML model, framework library and code in memory, which results in a much longer inference delay. Nevertheless, cold starts only occur when the request rate exceeds the concurrency, measured by the number of currently available lambda instances [32,73]. Existing benchmarking shows that a Lambda instance is recycled after it stays inactive for 45 to 60 minutes [31]. Our evaluations further confirm that, with more than 3 million requests, the cold start rate never exceeds 0.23%. Therefore, the latency impact of cold starts is limited. The cost impact is also negligible. Our profiling shows that $1 can spin up 7K inception-v3 Lambda instances, which is capable of serving more than 20K requests per second. Algorithm 1 hence does not consider the cost impact of Lambda cold starts.

Despite the negligible impacts of Lambda cold start, 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 [32].

Table 4: ML models and frameworks used in evaluation.

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

5 Experimental 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.

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 (§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 §4.3. The optimal batching hyperparameters W_{batch} and N_{batch} for `p2.xlarge` instance found by our tuning algorithm outlined in §4.3 are 200ms and 8 for Inception-v3, 750ms and 16 for NASNet, 490ms 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
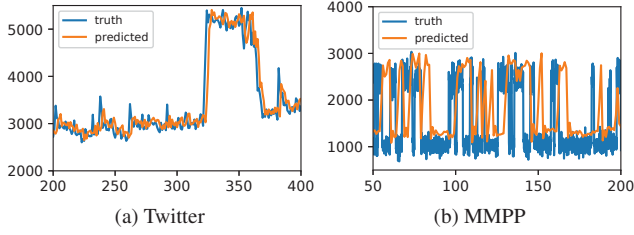
(a) Twitter   (b) MMPP

Figure 5: Snapshots of the arrival process using Twitter and MMPP with the prediction results of LSTM based algorithm.

least SL_{min} percent of requests must be served in RT_{max} time (§4.1). We set SL_{min} to 98% for all models, and set RT_{max} as 600ms, 1000ms, 100ms, and 1400ms for Inception-v3, NAS-Net, 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 [20]. 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. 5a(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 ML request load using *Markov-Modulated Poisson process* (MMPP) [28,35,60]. The load generated by MMPP are highly unpredictable, as the occurrence and duration of demand surges are completely random, as shown in Fig. 5b.

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. We use SageMaker [13] as the baseline for the evaluation. SageMaker is AWS's leading ML training and hosting system. SageMaker hosting employs AWS's new target tracking autoscaling policy [16,18]. 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 [16] and set the over-provisioning factor to 2 for SageMaker. We will show in Fig. 7 that even so the over-provisioning is still incapable of handling the volatile workload of the Twitter traces.
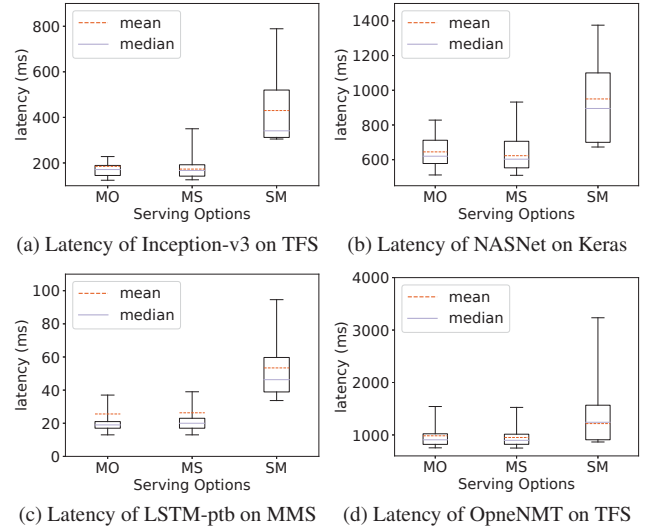

(a) Latency of Inception-v3 on TFS   (b) Latency of NASNet on Keras

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

Figure 6: Latency comparison of MArk-ondemand (MO), MArk-spot (MS), and SageMaker (SM) on 4 ML models using Twitter workload.

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. 5b demonstrates snapshots of the prediction results. We see that the prediction accuracy is in general good for the Twitter traces, yet very poor 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.

Experimental results using Twitter traces. We first compare MArk-ondemand, MArk-spot, and SageMaker on the ML models described in §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. 6, 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 $3.63\times$, $2.79\times$, $2.41\times$, and $3.15\times$ for the four ML models; MArk-spot achieves $6.21\times$, $5.91\times$, $6.64\times$, and $7.83\times$, respectively. For latency, MArk-ondemand achieves up to 57% reduction and MArk-spot achieves up to 60% reduction compared with SageMaker.

The latency advantage of MArk over SageMaker comes in three-fold. First, with appropriate batching configuration,

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

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. 6, 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.

MArk's cost reduction comes from the following aspects. First, predictive scaling together with Lambda services brings a more judicious over-provisioning design that can reduce the cost. The 2× cost reduction over SageMaker in MArk-ondemand using only CPU instances for LSTM-ptb is a good example. Second, GPU instances can further reduce the cost during high arrival rate as batching increases the efficiency of computing. The cost reduction is more significant for Open-NMT as it benefits the most from batching as shown in Fig. 2d. MArk-spot further brings down the cost by enjoying the spot market discounts. Note that although Lambda service used by MArk is expensive in price, but the cost of Lambda can be well justified by enabling more judicious over-provisioning.

We have also performed a case study of SLO compliance and report the *Complementary Cumulative Distribution Function* (CCDF) of request latency in Fig. 7. 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
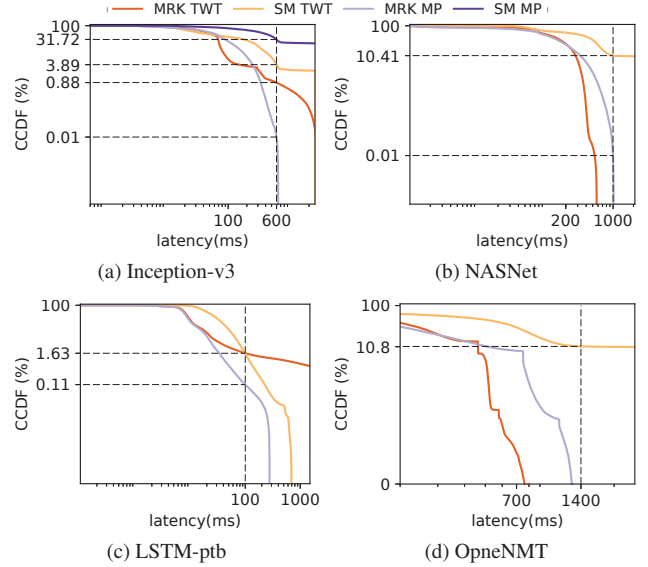

Figure 7: CCDF of latency comparison between MArk and SageMaker. RT_{max} is drawn as a black dashed vertical line (the black dashed horizontal line shows the corresponding CCDF value of RT_{max}). MRK and SM represents MArk and Sage-Maker, while TWT and MP represents Twitter and MMPP workload respectively.

the interest of space, we only demonstrate the SLO compliance results in Fig. 7. Fig. 7a 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.

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% in 2 minutes and compare the latency over time between MArk

(a) 50% surge in 2 min

(b) 75% surge in 2 min

(c) 100% surge in 2 min

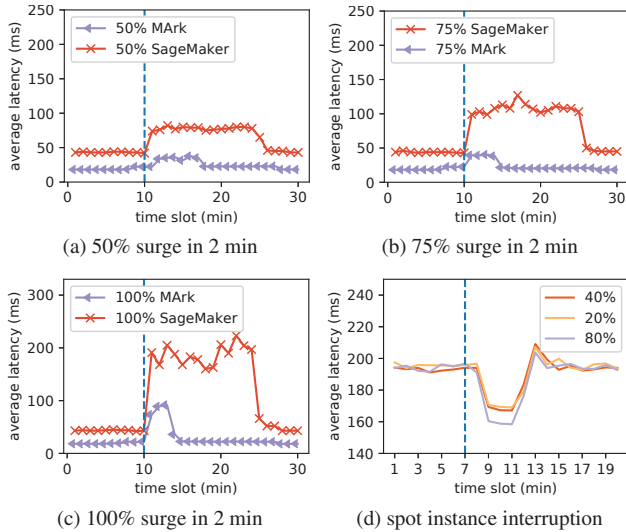(d) spot instance interruption

Figure 8: 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.

and SageMaker in Figs. 8a, 8b, and 8c. [3] 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, 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% of the instances respectively. Fig. 8d 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 §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. The measurements and evaluations in this paper are mainly based on AWS. However, the main design of MArk can be generally extended to other major cloud platforms, as they offer both IaaS and FaaS services, as well as flexible pricing models. Nevertheless, 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 they will bring down the cost of serving stable inference demands in a long run, and will leave it as a future work.

Large models. Deep learning models are becoming increasingly large and may not fit into the memory of Lambda (or even IaaS) instances. A possible solution goes to distributed inference under the model parallel scheme, which is not supported in our current design. We will leave it as a future work.

Hardware accelerator. We used the most common ML accelerator GPU as an example of utilizing hardware accelerators. The same batching formulation can be applied to other accelerators (e.g., FPGA) as they benefit from batching similarly.

MArk's architecture requires a master machine to make provisioning decisions. While such design has limitations on scalability and is vulnerable to the single point of failure, these problems can be easily addressed with mature industrial solutions such as Zookeeper [46].

7 Concluding Remark

In this paper, we 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 proposed MArk, a cost-effective and SLO-aware ML serving system. We prototyped MArk on AWS and showed that compared with the premier autoscaling ML platform SageMaker, MArk yields significant cost reduction (up to 7.8×) while complying with the SLO requirements with even better latency performance.

Acknowledgement

---

[3] 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.

References

[1] Amazon EC2. https://aws.amazon.com/ec2/, 2018.

[2] Amazon ECS. https://aws.amazon.com/ecs/, 2018.

[3] AWS Lambda. https://aws.amazon.com/lambda/, 2018.

[4] Docker. https://www.docker.com, 2018.

[5] Google Cloud Functions. https://cloud.google.com/functions/, 2018.

[6] Google Kubernetes Engine. https://cloud.google.com/kubernetes-engine/, 2018.

[7] PredictionIO. https://predictionio.apache.org, 2018.

[8] RedisML. https://github.com/RedisLabsModules/redis-ml, 2018.

[9] ALI-ELDIN, A., KIHL, M., TORDSSON, J., AND ELMROTH, E. Efficient provisioning of bursty scientific workloads on the cloud using adaptive elasticity control. In *Proceedings of the 3rd ACM Workshop on Scientific Cloud Computing* (2012).

[10] ALI-ELDIN, A., TORDSSON, J., AND ELMROTH, E. An adaptive hybrid elasticity controller for cloud infrastructures. In *IEEE Network Operations and Management Symposium* (2012).

[11] AMAZON. Amazon Web Services. https://aws.amazon.com/, 2018.

[12] AMAZON. AWS autoscaling. https://aws.amazon.com/autoscaling/, 2018.

[13] AMAZON. Build, train, and deploy machine learning models at scale. https://aws.amazon.com/sagemaker/, 2018.

[14] AMAZON. Configuring Lambda functions. https://docs.aws.amazon.com/lambda/latest/dg/resource-model.html, 2018.

[15] AMAZON. Dynamic scaling for Amazon EC2 auto scaling. https://amzn.to/2W2jvhc, 2018.

[16] AMAZON. Load testing for variant automatic scaling. https://docs.aws.amazon.com/sagemaker/latest/dg/endpoint-scaling-loadtest.html, 2018.

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

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

[19] ANIELLO, L., BONOMI, S., LOMBARDI, F., ZELLI, A., AND BALDONI, R. An architecture for automatic scaling of replicated services. In *Networked Systems*. Springer, 2014, pp. 122–137.

[20] ARCHIVETEAM. Twitter streaming traces, 2017.

[21] AWS. Amazon EC2 reserved instances. https://aws.amazon.com/ec2/pricing/reserved-instances/, 2018.

[22] AWS. Burstable performance instances. https://amzn.to/2APg4hG, 2018.

[23] AWS. Right sizing: Provisioning instances to match workloads. https://amzn.to/2VdIiK9, 2018.

[24] AWSLABS. MXNet model server. https://github.com/awslabs/mxnet-model-server, 2018.

[25] BARRETT, E., HOWLEY, E., AND DUGGAN, J. Applying reinforcement learning towards automating resource allocation and application scalability in the cloud. *Concurrency and Computation: Practice and Experience 25*, 12 (2013), 1656–1674.

[26] BERGSTRA, J., BASTIEN, F., BREULEUX, O., LAMBLIN, P., PASCANU, R., DELALLEAU, O., DESJARDINS, G., WARDE-FARLEY, D., GOODFELLOW, I., BERGERON, A., ET AL. Theano: Deep learning on GPUs with Python. In *NeuralPS, Big Learning Workshop* (2011).

[27] BODÍK, P., GRIFFITH, R., SUTTON, C., FOX, A., JORDAN, M. I., AND PATTERSON, D. A. Statistical machine learning makes automatic control practical for internet datacenters. In *USENIX HotCloud* (2009).

[28] CASALE, G., ZHANG, E. Z., AND SMIRNI, E. Trace data characterization and fitting for markov modeling. *Perform. Eval. 67*, 2 (2010), 61–79.

[29] CHOLLET, F., ET AL. Keras: Deep learning library for Theano and TensorFlow. https://keras.io, 2015.

[30] CRANKSHAW, D., WANG, X., ZHOU, G., FRANKLIN, M. J., GONZALEZ, J. E., AND STOICA, I. Clipper: A low-latency online prediction serving system. In *NSDI* (2017), pp. 613–627.

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

[32] CUI, Y. I'm afraid you're thinking about aws lambda cold starts all wrong. https://bit.ly/2Q1rrcr, 2018.

[33] DOYLE, R. P., CHASE, J. S., ASAD, O. M., JIN, W., AND VAHDAT, A. Model-based resource provisioning in a web service utility. In *USENIX Symposium on Internet Technologies and Systems* (2003), vol. 4, pp. 5–5.

[34] FANG, W., LU, Z., WU, J., AND CAO, Z. Rpps: a novel resource prediction and provisioning scheme in cloud data center. In *IEEE International Conference on Services Computing* (2012).

[35] FISCHER, W., AND MEIER-HELLSTERN, K. The Markov-modulated Poisson process (MMPP) cookbook. *Perform. Eval. 18*, 2 (1993), 149–171.

[36] GERS, F. A., SCHMIDHUBER, J., AND CUMMINS, F. Learning to forget: Continual prediction with LSTM. In *9th International Conference on Artificial Neural Networks* (1999).

[37] GOOGLE. Google cloud. https://cloud.google.com/, 2018.

[38] GOOGLE. Google cloud autoscaling. https://cloud.google.com/compute/docs/autoscaler/, 2018.

[39] GOOGLE. Kubernetes horizontal scaling. https://kubernetes.io/docs/tasks/run-application/horizontal-pod-autoscale/, 2018.

[40] GOOGLE. Cloud TPU performance guide. https://cloud.google.com/tpu/docs/performance-guide, 2019.

[41] GUJARATI, A., ELNIKETY, S., HE, Y., MCKINLEY, K. S., AND BRANDENBURG, B. B. Swayam: distributed autoscaling to meet slas of machine learning inference services with resource efficiency. In *Proceedings of ACM/IFIP/USENIX Middleware Conference* (2017), ACM, pp. 109–120.

[42] HAN, R., GHANEM, M. M., GUO, L., GUO, Y., AND OSMOND, M. Enabling cost-aware and adaptive elasticity of multi-tier cloud applications. *Future Generation Computer Systems 32* (2014), 82–98.

[43] HARLAP, A., TUMANOV, A., CHUNG, A., GANGER, G. R., AND GIBBONS, P. B. Proteus: Agile ML elasticity through tiered reliability in dynamic resource markets. In *Proceedings of ACM EuroSys* (2017).

[44] HE, K., ZHANG, X., REN, S., AND SUN, J. Deep residual learning for image recognition. In *Proceedings of IEEE CVPR* (2016).

[45] HE, X., SHENOY, P., SITARAMAN, R., AND IRWIN, D. Cutting the cost of hosting online services using cloud spot markets. In *Proceedings of the 24th International Symposium on High-Performance Parallel and Distributed Computing* (2015), ACM, pp. 207–218.

[46] HUNT, P., KONAR, M., JUNQUEIRA, F. P., AND REED, B. Zookeeper: Wait-free coordination for internet-scale systems. In *Proceedings of USENIX ATC* (2010).

[47] KLEIN, G., KIM, Y., DENG, Y., SENELLART, J., AND RUSH, A. M. Opennmt: Open-source toolkit for neural machine translation. *arXiv preprint arXiv:1701.02810* (2017).

[48] LEE, H., SATYAM, K., AND FOX, G. Evaluation of production serverless computing environments. In *Proceedings of IEEE CLOUD* (2018).

[49] LEE, Y., SCOLARI, A., CHUN, B.-G., SANTAMBROGIO, M. D., WEIMER, M., AND INTERLANDI, M. PRETZEL: Opening the black box of machine learning prediction serving systems. In *Proceedings of USENIX OSDI* (2018).

[50] LEITNER, P., AND SCHEUNER, J. Bursting with possibilities: An empirical study of credit-based bursting cloud instance types. In *Proceedings of IEEE/ACM Utility and Cloud Computing* (2015).

[51] MERITY, S., KESKAR, N. S., AND SOCHER, R. Regularizing and optimizing LSTM language models. *arXiv preprint arXiv:1708.02182* (2017).

[52] MICROSOFT. Microsoft Azure cloud computing platform & services. https://azure.microsoft.com/en-us/, 2018.

[53] NIKRAVESH, A. Y., AJILA, S. A., AND LUNG, C.-H. Towards an autonomic auto-scaling prediction system for cloud resource provisioning. In *Proceedings of IEEE International Symposium on Software Engineering for Adaptive and Self-Managing Systems* (2015).

[54] NVIDIA. NVIDIA TensorRT. https://developer.nvidia.com/tensorrt, 2018.

[55] OLSTON, C., FIEDEL, N., GOROVOY, K., HARMSEN, J., LAO, L., LI, F., RAJASHEKHAR, V., RAMESH, S., AND SOYKE, J. TensorFlow-Serving: Flexible, high-performance ML serving. *arXiv preprint arXiv:1712.06139* (2017).

[56] PENG, Y., BAO, Y., CHEN, Y., WU, C., AND GUO, C. Optimus: an efficient dynamic resource scheduler for deep learning clusters. In *Proceedings of ACM EuroSys* (2018).

[57] PRODAN, R., AND NAE, V. Prediction-based real-time resource provisioning for massively multiplayer online games. *Future Generation Computer Systems 25*, 7 (2009), 785–793.

[58] QU, C., CALHEIROS, R. N., AND BUYYA, R. A reliable and cost-efficient auto-scaling system for web applications using heterogeneous spot instances. *Journal of Network and Computer Applications 65* (2016), 167–180.

[59] QU, C., CALHEIROS, R. N., AND BUYYA, R. Auto-scaling web applications in clouds: A taxonomy and survey. *ACM Computing Surveys (CSUR) 51*, 4 (2018), 73.

[60] RAJABI, A., AND WONG, J. W. MMPP characterization of web application traffic. In *Proceedings of IEEE MASCOTS* (2012).

[61] ROY, N., DUBEY, A., AND GOKHALE, A. Efficient autoscaling in the cloud using predictive models for workload forecasting. In *Proceedings of IEEE CLOUD* (2011).

[62] SANDERS, J., AND KANDROT, E. *CUDA by example: an introduction to general-purpose GPU programming*. Addison-Wesley Professional, 2010.

[63] SHARMA, P., LEE, S., GUO, T., IRWIN, D., AND SHENOY, P. Spotcheck: Designing a derivative iaas cloud on the spot market. In *Proceedings of ACM EuroSys* (2015).

[64] SHI, X., CHEN, Z., WANG, H., YEUNG, D.-Y., WONG, W.-K., AND WOO, W.-C. Convolutional lstm network: A machine learning approach for precipitation nowcasting. In *Proc. NeuralPS* (2015).

[65] SONG, B., YU, Y., ZHOU, Y., WANG, Z., AND DU, S. Host load prediction with long short-term memory in cloud computing. *The Journal of Supercomputing* (2017), 1–15.

[66] SZEGEDY, C., IOFFE, S., VANHOUCKE, V., AND ALEMI, A. A. Inception-v4, inception-resnet and the impact of residual connections on learning. In *AAAI* (2017), vol. 4, p. 12.

[67] SZEGEDY, C., VANHOUCKE, V., IOFFE, S., SHLENS, J., AND WOJNA, Z. Rethinking the inception architecture for computer vision. In *Proceedings of the IEEE CVPR* (2016).

[68] TENSORFLOW. TensorFlow Serving batching guide. https://bit.ly/2VOpb9O, 2018.

[69] TU, Z., LI, M., AND LIN, J. Pay-per-request deployment of neural network models using serverless architectures. In *Proceedings of NAACL: Demonstrations* (2018).

[70] URGAONKAR, B., SHENOY, P., CHANDRA, A., GOYAL, P., AND WOOD, T. Agile dynamic provisioning of multi-tier internet applications. *ACM Transactions on Autonomous and Adaptive Systems (TAAS) 3*, 1 (2008), 1.

[71] WANG, C., URGAONKAR, B., GUPTA, A., KESIDIS, G., AND LIANG, Q. Exploiting spot and burstable instances for improving the cost-efficacy of in-memory caches on the public cloud. In *Proceedings of ACM EuroSys* (2017).

[72] WANG, W., WANG, S., GAO, J., ZHANG, M., CHEN, G., NG, T. K., AND OOI, B. C. Rafiki: Machine learning as an analytics service system. *arXiv preprint arXiv:1804.06087* (2018).

[73] YAN, F., REN, L., DUBOIS, D. J., CASALE, G., WEN, J., AND SMIRNI, E. How to supercharge the amazon t2: Observations and suggestions. In *Proceedings of IEEE CLOUD* (2017).

[74] YAN, F., RUWASE, O., HE, Y., AND SMIRNI, E. SERF: efficient scheduling for fast deep neural network serving via judicious parallelism. In *Proceedings of IEEE/ACM SC16* (2016).

[75] ZHANG, H., STAFMAN, L., OR, A., AND FREEDMAN, M. J. SLAQ: Quality-driven scheduling for distributed machine learning. In *Proceedings of ACM SoCC* (2017).

[76] ZOPH, B., VASUDEVAN, V., SHLENS, J., AND LE, Q. V. Learning transferable architectures for scalable image recognition. *arXiv preprint arXiv:1707.07012 2*, 6 (2017).