Serverless Data Science - Are We There Yet? A Case Study of Model Serving

Yuncheng Wu
National University of Singapore
Singapore
wuyc@comp.nus.edu.sg

Tien Tuan Anh Dinh
Singapore University of Technology and Design
Singapore
dinhtta@sutd.edu.sg

Guoyu Hu
National University of Singapore
Singapore
guoyu.hu@u.nus.edu

Meihui Zhang
Beijing Institute of Technology
P. R. China
meihui_zhang@bit.edu.cn

Yeow Meng Chee
National University of Singapore
Singapore
ymchee@nus.edu.sg

Beng Chin Ooi
National University of Singapore
Singapore
ooibc@comp.nus.edu.sg

ABSTRACT

Machine learning (ML) is an important part of modern data science applications. Data scientists today have to manage the end-to-end ML life cycle that includes both model training and model serving, the latter of which is essential, as it makes their works available to end-users. Systems of model serving require high performance, low cost, and ease of management. Cloud providers are already offering model serving choices, including managed services and self-rented servers. Recently, serverless computing, whose advantages include high elasticity and a fine-grained cost model, brings another option for model serving.

Our goal in this paper is to examine the viability of serverless as a mainstream model serving platform. To this end, we first conduct a comprehensive evaluation of the performance and cost of serverless against other model serving systems on Amazon Web Service and Google Cloud Platform. We find that serverless outperforms many cloud-based alternatives. Further, there are settings under which it even achieves better performance than GPU-based systems. Next, we present the design space of serverless model serving, which comprises multiple dimensions, including cloud platforms, serving runtimes, and other function-specific parameters. For each dimension, we analyze the impact of different choices and provide suggestions for data scientists to better utilize serverless model serving. Finally, we discuss challenges and opportunities in building a more practical serverless model serving system.

CCS CONCEPTS

• **Computing methodologies** → **Distributed computing methodologies**; *Machine learning*; • **Information systems** → Information systems applications.

KEYWORDS

data science; serverless computing; model serving; design space

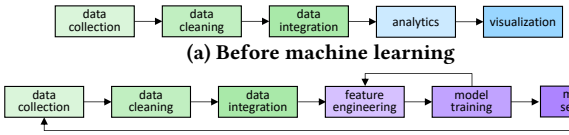
SIGMOD '22, June 12–17, 2022, Philadelphia, PA, USA

ACM ISBN 978-1-4503-9249-5/22/06.
https://doi.org/10.1145/3514221.3517905

ACM Reference Format:
Yuncheng Wu, Tien Tuan Anh Dinh, Guoyu Hu, Meihui Zhang, Yeow Meng Chee, and Beng Chin Ooi. 2022. Serverless Data Science - Are We There Yet? A Case Study of Model Serving. In *Proceedings of the 2022 International Conference on Management of Data (SIGMOD '22), June 12–17, 2022, Philadelphia, PA, USA.* ACM, New York, NY, USA, 10 pages. https://doi.org/10.1145/3514221.3517905

1 INTRODUCTION

Machine learning (ML) has transformed data science [25, 28, 51]. Figure 1 illustrates the changes. Until recently, a typical data science pipeline consists of data collection, cleaning and integration, data analytics, and visualization. Now, the pipeline includes the full ML life cycle: feature engineering, model training, and model serving, turning data scientists into end-to-end data engineers [35]. Consider a nutrition analysis application, FoodLG[1], as an example. A data scientist first collects a set of labeled food images from nutritionists, transforms the data, and builds a deep learning model [41, 54] that classifies food images and analyzes the nutrition according to a food knowledge base (e.g., CalorieKing [8]). After that, the data scientist deploys the model and makes it available to users' mobile apps. Then a user sends food images as requests to the deployed model and receives the predicted food nutrition (e.g., calorie, protein). The application can record the user's daily intake, analyze her eating habit, and provide suggestions for healthy dietary intake. New data is collected and fed back to the pipeline to improve model accuracy. In real-world applications, model serving plays a crucial role in bringing the works of data scientists to the end-users.



(a) Before machine learning

(b) With machine learning

Figure 1: A typical data science pipeline.

Model serving systems are interactive, that is, they handle inference requests from users in near real-time. As a result, their design goals differ from those of model training systems that focus on maximizing throughput. The first goal is *high performance*, which means that the system can process requests fast, even under bursty

[1] http://www.foodlg.com/

workloads. The second goal is *low cost*, which means that the system is cost-effective when handling a large number of requests. The third goal is *ease of management*, which means that the system allows data scientists to quickly deploy ML models without worrying about low-level details such as resource management. Existing cloud providers are offering several model serving choices, including managed ML services and self-rented servers [7, 17, 38]. These options, however, do not meet all three goals.

Serverless computing [26, 31], an emerging cloud computing paradigm, brings another option for model serving, and it has the potentials to achieve the three goals above. Figure 2 shows how serverless can support model serving for the aforementioned application. The data scientist first uploads a trained model to the cloud storage, then writes and deploys a function for executing model inference. The cloud provider creates an invokable endpoint (e.g., a URL) for the function, to which users can send their requests and receive inference results. The serverless platform takes care of provisioning computation instances to execute the function, and of scaling the number of instances according to the request workload. The data scientist is charged by the actual amount of consumed resources, instead of the time of reserved resources.

Serverless can achieve the first goal of model serving, i.e., high performance, because it adaptively scales to multiple instances very fast so that it can handle the requests timely. It can meet the second goal, i.e., low cost, because billing is only based on the actual resource consumption. This means that the scientist does not have to provision for peak load, which can be expensive, especially when using GPUs. The last goal, ease of management, is also met since provisioning and scaling of resources are handled automatically by the cloud platform. Despite these potentials, there are challenges in applying serverless to model serving. In particular, serverless has several limitations, including small memory size, limited running time, and lack of persistent states [26, 30, 32, 33, 40, 46, 49].

In this paper, we ask the following question: *can serverless computing be a mainstream model serving platform for data science applications?* To answer this, we conduct an extensive comparison between serverless and other cloud-based model serving alternatives with respect to performance and cost. We consider eight systems spanning two major cloud providers: Amazon Web Service [3] (AWS) and Google Cloud Platform [19] (GCP). In particular, we evaluate Lambda, Cloud Functions, SageMaker, AI Platform, and self-rented CPU and GPU systems from AWS and GCP. We use three deep learning models for the evaluation: MobileNet and VGG, two image classification models, and ALBERT, a natural language processing model. We evaluate these models on the eight systems under three different workloads. We compare the systems in three metrics: response latency, request success ratio, and cost.

Our results contain two surprising findings. First, while earlier works claimed that serverless is not suitable for model serving [26], we find that, in most cases, serverless outperforms managed ML services such as SageMaker and AI Platform in both cost and performance. Meanwhile, it has better performance than self-rented CPU systems but with a higher cost; nevertheless, serverless can be much faster if the cost is relatively comparable on AWS. Second, while other works suggested that serverless should be used as a complementary platform to a GPU-based system for handling excessive load [57], we show that there are settings under which serverless

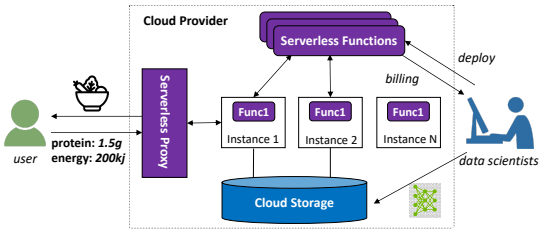


Figure 2: Serverless model serving example.

could be a better choice than self-rented GPU systems. In particular, on AWS, serving MobileNet with workload-200 (see Figure 4, which consists of 86000 requests within 15 minutes) with serverless results in an average latency of 0.097*s* and a cost of $0.186. In contrast, doing the same using a GPU server results in 7.52*s* and $0.187, respectively. With comparable cost, serverless achieves 77.5× latency improvement. Importantly, we find that serverless is less sensitive to changes in workloads or models, which means it can provide consistent performance even under bursty workloads or given large models. Our evaluation results provide an important insight, that is, serverless is a viable and promising option for model serving.

Next, we describe the design space of serverless model serving, which comprises multiple dimensions, including serverless platforms, serving runtimes, and other function-specific parameters. We further investigate the impact of each dimension on serverless' performance and cost, and present suggestions for data scientists to better utilize serverless model serving. In particular, we observe that there is a gap in the cold start time between AWS and GCP serverless functions. For instance, using the same serving environment, namely TensorFlow1.15 [22], AWS takes about 9.49*s* on average to start an instance while GCP takes about 14.19*s* when serving ALBERT with workload-120 (see Figure 4). Another interesting finding is that a lightweight serving runtime could reduce both latency and cost in most cases. For instance, if serving with OnnxRuntime1.4 [39] instead of TensorFlow1.15, we can obtain up to 3.61× latency speedup and 4.55× cost reduction.

In summary, we make the following contributions in this paper.

- We conduct an extensive cost and performance comparison of serverless against other cloud-based systems for model serving. Our analysis covers eight different systems on two major public cloud providers, with three deep learning models and three different workloads.
- We present interesting findings indicating that serverless is a viable and promising option for model serving. We show that it outperforms managed ML services and CPU-based systems in most cases, and there are settings under which it can even achieve better performance than GPU-based systems.
- We further explore the impact of its design space on serverless model serving and present recommendations for data scientists on how to better use serverless for model serving. In addition, we discuss research challenges and opportunities in building a more practical serverless model serving system.

2 BACKGROUND AND RELATED WORK

2.1 Machine Learning Model Serving

Machine learning has shown great success in data science applications [25, 28, 51]. How to efficiently deploy the trained ML models to

serve end-users with low latency becomes increasingly important. Recently, there are a number of model serving systems [12, 23, 53] that focus on improving cost-effectiveness or model accuracy while meeting service level objective (SLO) on latency, by automatically selecting different configurations. However, these systems are mainly based on server machines and do not work well for ML model serving due to the mismatch of target workloads [57].

2.2 Serverless Computing

Serverless computing [13, 16, 31, 50] is a recent rise of cloud computing execution paradigm such that the cloud provider runs the server and dynamically manages the allocation of resources. It can simplify the deployment process of function code to the production stage. Meanwhile, it can automatically increase and release resources to adapt to the number of user invocations, making it elastic to handle various workloads. Pricing is based on the actual amount of resources consumed by the deployed function. Due to its high elasticity and fine-grained cost model, serverless computing has been adopted in many data science applications, such as database analytics [30, 40, 43, 44, 47, 55], and model training [9, 14, 29, 52].

2.3 Serverless Model Serving

In particular, serverless computing can be seamlessly utilized for model serving due to its stateless computations [24]. As mentioned in Section 1, data scientists (aka. function developers) can deploy a pre-trained model on the cloud via a function. For a deployed function, once received an *event* from the serverless proxy, the cloud provider will create a new instance or forward *event* to a warmed instance for execution. If an instance is newly created, it imports the serving dependencies, downloads the model (which was uploaded by the data scientists) from cloud storage, and loads the model into the serving runtime. Otherwise, the loaded model already exists. As a result, the instance parses the input sample from *event*, executes the inference, and returns the prediction.

The most related work to ours is MArK [57], which mainly evaluates the cost of several model serving systems on the cloud and proposes a serving system that combines self-rented servers and serverless to reduce the cost. BATCH [2] also considers serverless model serving, which designs a mechanism that batches requests and dynamically chooses the resource configuration of invoked serverless function, to satisfy latency constraints and minimize cost. However, neither of them has a comprehensive investigation of the performance of existing serverless systems for model serving.

Several other works [11, 36, 56] explore the characteristics of serverless platforms based on various applications (e.g., image resizing). In essence, these applications follow the same stateless execution paradigm as model serving, which reads the data, performs pre-defined calculations, and returns the results. However, they do not cover two specific characteristics of serverless model serving: it requires a much longer provisioning time to prepare for the serving environment [23], and model serving workloads are often unpredictable and highly bursty [57].

2.4 Model Serving Systems on the Cloud

Serverless model serving systems. We consider two serverless platforms: *AWS Lambda* [6] and *Google Cloud Functions* (CF) [18].

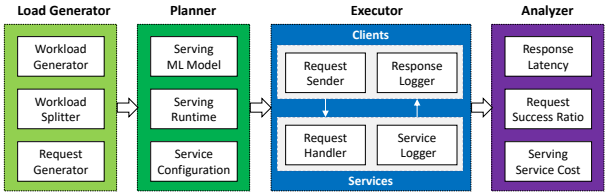


Figure 3: Evaluation framework.

On Lambda, model owners first package the serving environment (e.g., TensorFlow1.15) into a zip file or a container image and upload it to the cloud storage. Then, they create a function by specifying the configuration (e.g., memory size) for each instance and deploy the function based on the uploaded serving environment. We adopt the container image method in our experiments, where the image size is limited to 10GB. On CF, we can only specify the required environment in the *Requirement.txt*, and the platform will build the package automatically. Model owners are charged by the amount of consumed resources, which mainly depends on the selected memory size and the number of instances created at runtime.

Managed machine learning (ManagedML) services. Most cloud providers offer fully managed services for scientists to build, train, and deploy ML models easily. These ManagedML services are naturally applicable for model serving, with the ability to autoscale when the workload is too heavy to be processed by existing instances. In the experiments, we evaluate two ManagedML services: *AWS SageMaker* [7] and *Google AI Platform* [17]. On both services, model owners can create an endpoint by uploading a pre-trained model to the cloud storage and specifying a serving runtime to execute that model. The cost is computed based on the total execution time of active instances.

Self-rented servers for model serving. With self-rented servers, model owners can run virtual machines (VMs) with various configurations (e.g., vCPUs, memory, network, and storage) and deploy a model serving service by themselves on the rented instances. Cloud providers also provide flexible pricing options to the customers. In this paper, we deploy model serving services on both CPU servers and GPU servers on *AWS EC2* [4] and *Google Compute Engine* [20].

Unless stated otherwise, we use AWS-Serverless, GCP-Serverless, AWS-ManagedML, and GCP-ManagedML to denote Lambda, Cloud Functions, SageMaker, and AI Platform, for better presentation.

3 EVALUATION METHODOLOGY

We design a benchmarking framework for understanding and comparing the performance of different model serving systems. It consists of four main components as shown in Figure 3, namely a load generator, a planner, an executor, and an analyzer. The framework can be deployed over multiple nodes. It comes with default workloads and configuration, and it can be easily extended to support new models and new platforms.

Load generator. At first, we generate workloads and requests for the clients. We note that there are no publicly available workloads for model serving [57]. Therefore, we use the Markov-Modulated Poisson Process (MMPP) model [15, 45], also adopted in [2, 57], to generate synthetic workloads with different arrival rates and numbers of requests. The generated workloads are highly unpredictable as the occurrence and duration of demand surges are random.

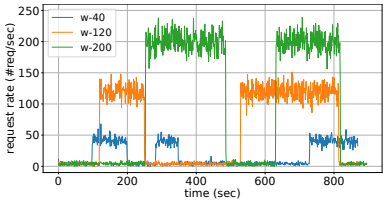


Figure 4: Generated MMPP workloads.

We implement a workload splitter to evenly divide the workloads such that we can employ multiple clients to send requests, and the aggregated request rate matches the original workloads. Besides, we implement a request generator that creates a pool of requests, from which a client randomly selects one request to send, ensuring that model serving systems do not cache the prediction results.

In this paper, we generate three workloads to simulate low, medium, and high request rates, as shown in Figure 4. The numbers 40, 120, and 200 in the workloads represent the higher arrival rate of the two Poisson processes we used in MMPP. Specifically, we refer to the synthetic workloads in [2, 57] to select the higher arrival rates. Meanwhile, the 40, 120, and 200 workloads consist of 15000, 51600, and 86000 requests, respectively (duration is about 15 minutes). Besides, the workloads are split for 8 clients, and the request pool size is set to 200.

Planner. Before serving clients' requests, we need to deploy the serving services on the cloud. Specifically, the deployment is based on three dimensions: model, runtime, and configuration, which exactly define the environment on the instances to serve the requests.

In this paper, we consider two serving runtimes: TensorFlow (TF) 1.15 and OnnxRuntime (ORT) 1.4. The first is used for a fair comparison between serverless and other serving systems, as it is one of the most popular deep learning frameworks and is well-supported by ManagedML service and self-rented servers on both clouds. In particular, GCP-ManagedML only supports TF for deep learning. The second runtime (with smaller runtime size and optimized executions) is used for exploring the performance improvement on serverless platforms, as will be discussed in Section 5.2.

Meanwhile, we use three deep learning models representing two popular data science applications: MobileNet [27] and VGG [48] for image classification, and ALBERT [34] for natural language processing. The model sizes are 16MB, 51.5MB, and 548MB, respectively. A noteworthy aspect is that there is a temporary directory storage limitation (i.e., 512MB) on AWS-Serverless [5], which means we cannot download the VGG model from cloud storage as the other two models. Therefore, we directly pack the VGG model into another directory inside the uploaded image for serving this model. By default, we use 2GB memory for both serverless platforms. We apply ml.m4.2xlarge (with 8vCPUs and 32GB memory) on AWS-ManagedML and n1-standard-8 (with 8vCPUs and 30GB memory) on GCP-ManagedML. Besides, we use similar configurations for self-rented CPU servers, while for GPU servers, we use g4dn.2xlarge and n1-standard-8 with 1 Tesla T4 on AWS and GCP, respectively.

Executor. After a serving service is deployed, the clients can send requests to the service according to the workloads. Specifically, each client randomly picks one request from the pool, sends it through designated APIs, and receives the response. We parse the response and record it into a log file, including response status and latency. For serverless serving and self-rented servers, we use

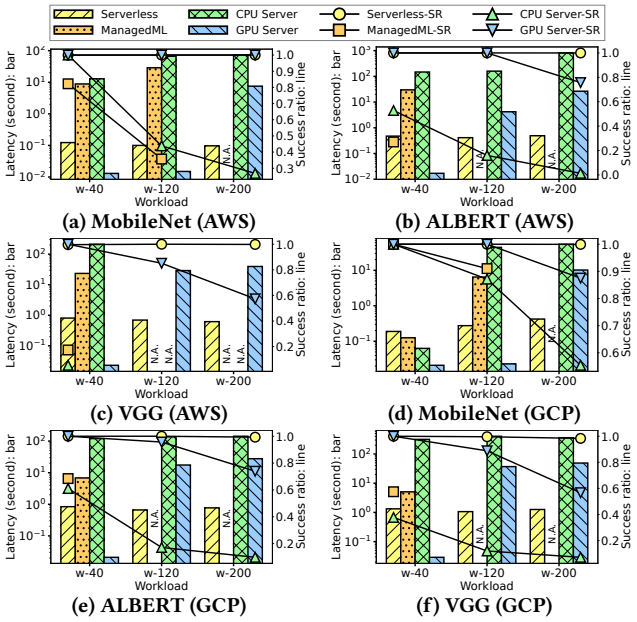


Figure 5: Model serving systems' performance comparison.

standard HTTP API to send the requests. While for ML services, requests need to be sent through the API provided by the cloud platform, e.g., Amazon Boto3 and Google API Client.

When receiving a client's request, the model serving systems allocate necessary resources to process the request, including preprocessing the request (e.g., resizing the image, downloading the required model), computing the model prediction, and retrieving the label, before responding to the client. The cloud systems also record the execution logs for further analysis.

Analyzer. After finishing the model serving for the workloads, we collect the results from both clients and cloud serving services, and analyze the systems using three main metrics.

- Response latency. We analyze the end-to-end response latency for each request on the clients, report the average latency of the successful requests, and measure the latency trends with respect to timestamps in the workloads.
- Request success ratio (SR). When the request rate exceeds the limit that a system can handle, some requests are dropped, or errors are returned. We analyze the ratio of successful requests over all the requests. The higher the ratio, the better.
- Cost. We analyze the charges for each experiment. For systems charged hourly (e.g., self-rented servers), we estimate the cost based on the actual execution time.

4 MODEL SERVING SYSTEMS COMPARISON

In this section, we compare the performance and cost of serverless model serving with alternative systems, including ManagedML, CPU server, and GPU server using serving runtime TensorFlow1.15. Figure 5 and Table 1 summarize the overall comparison.

4.1 Summary of Key Findings

We summarize three key findings before presenting the details. *First*, serverless has better performance and is more cost-effective than ManagedML in most cases. Specifically, for MobileNet with

Table 1: Costs for evaluated model serving systems[2]

Model Serving Systems		workload-40	workload-120	workload-200
AWS-Serverless	MobileNet	$0.050	$0.117	$0.186
	ALBERT	$0.223	$0.665	$1.326
	VGG	$0.492	$1.134	$1.993
AWS-ManagedML	MobileNet	$0.428	$0.610	-
	ALBERT	$0.445	-	-
	VGG	$0.436	-	-
AWS-CPU		$0.089	$0.089	$0.092
AWS-GPU		$0.181	$0.182	$0.187
GCP-Serverless	MobileNet	$0.065	$0.279	$0.537
	ALBERT	$0.299	$0.887	$1.511
	VGG	$0.507	$1.438	$2.467
GCP-ManagedML	MobileNet	$0.164	$0.313	-
	ALBERT	$0.468	-	-
	VGG	$0.872	-	-
GCP-CPU		$0.092	$0.092	$0.094
GCP-GPU		$0.176	$0.177	$0.182

workload-40, the average latency of AWS-ManagedML is 71.6× slower than that of AWS-Serverless, while its cost is 8.56× higher. *Second*, in general, serverless outperforms CPU servers but often incurs a higher cost. However, on AWS, serverless can be better in both performance and cost when serving a simple model under a low workload. For instance, for MobileNet with workload-40, AWS-Serverless is 104.5× faster than CPU server and also 1.78× lower in cost. *Third*, under a low workload, GPU servers are usually better than serverless. While under a high workload, serverless is more stable and could be a better choice than GPU servers. For instance, for MobileNet with workload-200, AWS-Serverless results in an average latency of 0.097*s* and a cost of $0.186, while those for a GPU server are 7.52*s* and $0.187, respectively. In other words, AWS-Serverless is 77.5× faster given comparable cost.

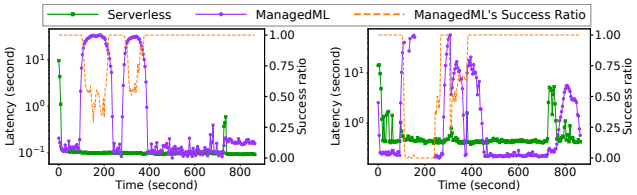
4.2 Serverless vs. ManagedML Service

We first compare serverless against managed ML services. For managed ML services, autoscaling is enabled, and the minimum number of running instances was set to 1.

AWS-Serverless vs. AWS-ManagedML Figure 5a-5c show the comparison between AWS-Serverless and AWS-ManagedML for the MobileNet, ALBERT, and VGG models, respectively. The average latency of AWS-Serverless is two orders of magnitude lower than that of AWS-ManagedML. Furthermore, there are almost no failed requests in AWS-Serverless, but there are many in AWS-ManagedML, especially when the request rate is high or the model is complex. For example, the success ratio for MobileNet drops from 82% (workload-40) to 36% (workload-120). For ALBERT and VGG, even with workload-40, the success ratios are only 27% and 17%, respectively, rendering the service unusable.

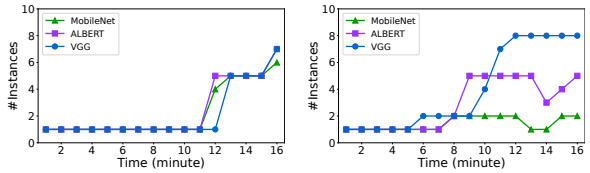
Figure 6a shows a detailed comparison for MobileNet under workload-40, where solid lines denote latency and dotted lines denote success ratio. At first, the average latency of AWS-Serverless (about 10 seconds) is higher than that of AWS-ManagedML. This is due to the cold-start time. However, as the system is warming up , AWS-Serverless performs better and is more stable than AWS-ManagedML. When the request rate becomes high (e.g., starting at around timestamp 100), AWS-ManagedML is unable to keep up, resulting in high latency and request failure. In fact, AWS-ManagedML's autoscaling takes several minutes to start new instances, leading to a large number of queued requests and thus delayed responses. Figure 7a shows the number of active instances

[2]The costs are the absolute values for the evaluated experiments.



(a) MobileNet with w-40 (AWS) (b) ALBERT with w-40 (GCP)

Figure 6: Serverless and ManagedML comparison.



(a) AWS ManagedML (b) GCP ManagedML

Figure 7: The number of instances on ManagedML services.

on AWS-ManagedML under workload-40. We can observe on the platform that it desires about 5 instances at timestamp 7 minutes. However, the instances are ready and in service at timestamp 11 minutes. While as will be shown in Figure 11a, AWS-Serverless can spawn new instances (e.g., tens or even a hundred of instances) quickly under the same workload for its high elasticity.

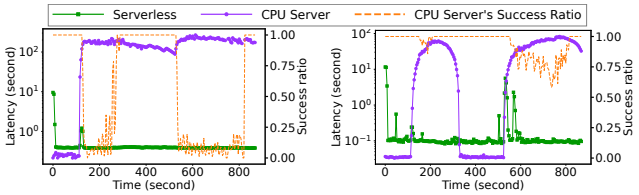
Regarding cost from Table 1, we can see that AWS-Serverless is more cost-efficient than AWS-ManagedML. The reason is that AWS-ManagedML needs several minutes to start new instances, and the evaluated workloads are relatively short. Most of the costs are spent on autoscaling instances rather than on doing the prediction.

GCP-Serverless vs. GCP-ManagedML. Figure 5d-5f show the results for Google systems under the same settings. For MobileNet under workload-40, there are no failed requests for both systems, but GCP-Serverless is slightly worse than GCP-ManagedML regarding latency, which is different from the comparison on AWS. The reason may be two-fold. One is that GCP-Serverless performs relatively worse than AWS-Serverless, which will be explored in detail in Section 5.1. The other is that GCP-ManagedML scales slightly better than AWS-ManagedML in the experiments. As shown in Figure 7b, GCP-ManagedML can scale to 2 instances at timestamp 6 minutes, which is earlier than AWS-ManagedML. Therefore, GCP-ManagedML can process the requests relatively faster, which results in better performance. However, when serving a larger model or with a higher workload, both latency and success ratio of GCP-ManagedML deteriorate significantly, while those of GCP-Serverless are stable and better. Figure 6b details the comparison for ALBERT with workload-40. Similarly, once the number of queued requests reaches a threshold, its performance degrades quickly.

For the cost, GCP-Serverless is slightly more efficient, but the advantage is not significant as the comparison on AWS. This is mainly because GCP-Serverless does not perform as well as AWS-Serverless. Nevertheless, if considering both performance and cost, GCP-Serverless is still a preferable choice over GCP-ManagedML.

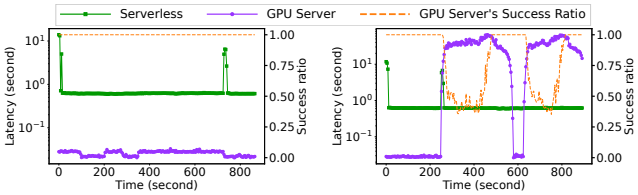
4.3 Serverless vs. CPU Server

AWS-Serverless vs. CPU server. From Figure 5a-5c, we observe that for all models, the average latency of AWS-Serverless is always smaller than CPU server, and the advantage is more pronounced



(a) ALBERT with w-120 (AWS) (b) MobileNet with w-120 (GCP)

Figure 8: Serverless and CPU server comparison.



(a) VGG with w-40 (AWS) (b) VGG with w-200 (AWS)

Figure 9: Serverless and GPU server comparison.

when the workload is higher or when the model is larger. For example, for the CPU server, the success ratios of MobileNet are 100%, 44%, and 27% with workloads 40, 120, and 200, respectively; meanwhile, under workload-40, the success ratios for MobileNet, ALBERT, and VGG are 100%, 53%, and 6%, respectively. This is because the CPU server is overloaded such that more requests are queued up under higher load or the execution time per request is longer, leading to increased latency and failure requests. Figure 8a details the performance of ALBERT with workload-120, from which we see the latency goes up sharply at the first request peak (timestamp 100) and stays at a high level. In contrast, AWS-Serverless' latency remains consistently low for its superior elasticity.

AWS EC2 also provides autoscaling for self-rented servers; thus, we create an autoscaling group for each serving model, such that new instances will be started and added in the group given a predefined template. Then, we create a load balancer for the group to receive clients' requests and forward the requests to existing instances. Nonetheless, similar to what we observed on AWS-ManagedML, it takes several minutes (i.e., 3 to 5 minutes) to start a new instance, making the service less reactive to bursty requests. Besides, there was no direct way to collect the cost of autoscaled instances on AWS EC2; thus, we do not report the results in this paper.

For the cost, AWS-Serverless is higher in most cases. However, we note that the average latency and success ratio of the CPU server are extremely low. A comparable situation is for MobileNet with workload-40, where AWS-Serverless is cheaper (i.e., $0.065) than CPU server (i.e., $0.089) while delivering better performance.

GCP-Serverless vs. CPU server. Figure 5d-5f illustrate a similar pattern to the comparison between GCP-Serverless and GCP-ManagedML. In particular, for MobileNet with workload-40, CPU server is slightly faster than GCP-Serverless. However, when the workload increases, the performance of CPU server degrades greatly, as shown in Figure 8b. At the two request peaks (timestamps 100 to 250 and 500 to 800), the average latency grows tens of seconds. For the cost, GCP-Serverless is more expensive. However, as mentioned above, CPU server cannot handle bursty requests effectively.

4.4 Serverless vs. GPU Server

AWS-Serverless vs. GPU server. Figure 5a-5c show that under workload-40, the GPU server's average latency is always lower than that of AWS-Serverless for the three models. Figure 9a gives a detailed comparison for the VGG model. That is because the GPU server can process each request quickly (e.g., about 0.02 seconds per request in our experiments). However, when given a higher workload, more requests are queued up, as shown in Figure 9b for VGG with workload-200. The results can be analyzed in three stages. First, at the beginning, the GPU server performs better than AWS-Serverless since the latter incurs cold-start overhead. Second, once AWS-Serverless instances are warmed up, AWS-Serverless outperforms the GPU server when the request rates are high. The rationale is that the request rate exceeds GPU server's capacity; thus, the request queue grows and leads to higher latency. Third, when the request rates are reduced, GPU server regains its advantage (e.g., around timestamp 600). However, in most periods, AWS-Serverless' latency is less than GPU server, resulting in lower average latency.

Regarding the cost, there are two main observations. First, for the MobileNet model, GPU server has better performance under workload-40 and workload-120, but its cost is also higher because the GPU server is under-utilized while still being charged. While for large models, e.g., ALBERT and VGG, GPU server is cheaper. The reason is that AWS-Serverless is charged by the number of requests and duration for processing each request. The execution time for these two models is relatively expensive, leading to a higher cost. Second, under a low workload, GPU server is better in both cost and performance in most cases, demonstrating its superior ability for model serving. However, under a high workload, GPU server has higher latency and request failures, and AWS-Serverless can be a better choice as its latency is less sensitive to the workloads and models, demonstrating its high elasticity. For example, for MobileNet with workload-200, AWS-Serverless is 77.5× faster than GPU Server given comparable cost. In addition, as mentioned in Section 4.3, it does not perform well to use autoscaling on CPU servers. More severely, we observe that when using autoscaling on GPU servers, it takes 8 to 10 minutes before a new instance can serve requests, rendering autoscaling useless in our workloads.

GCP-Serverless vs. GPU server. The comparison is similar to that on AWS. We omit the description due to the space limitation.

5 DESIGN SPACE OF SERVERLESS SERVING

So far, we have shown that serverless is a viable and promising option for model serving.In this section, we further investigate the impact of serverless serving's design space on its performance, including serverless platforms, serving runtimes, and various function-specific parameters.

5.1 Serverless Platforms

We first compare the performance and cost of serverless platforms, as shown in Figure 5 and Table 1. We observe that AWS-Serverless is always better for the three metrics. For instance, given MobileNet under workload-200 (see Figure 5c and Figure 5f), the average latency, success ratio, and cost on AWS-Serverless are 0.097*s*, 100%, and $0.186, while those on GCP-Serverless are 0.422*s*, 99.94%, and $0.537, respectively. There are two main reasons.

First, one contributing factor is the instance's cold-start time. In particular, GCP-Serverless takes a longer time to execute the cold-start requests, for example, around 11.71*s* and 14.19*s* for MobileNet

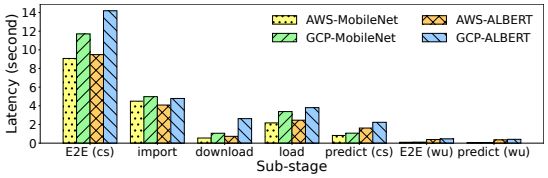
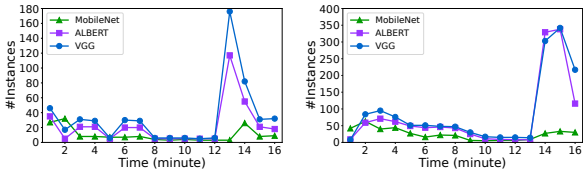


Figure 10: Breakdown comparison of serverless platforms, where "cs" and "wu" denote "cold-start" and "warm-up".



(a) AWS Serverless **(b) GCP Serverless**

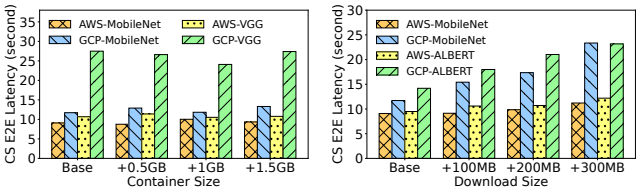
Figure 11: The number of instances on serverless platforms.

and ALBERT under workload-120, respectively. In contrast, AWS-Serverless takes only 9.08*s* and 9.49*s*, respectively. To attribute the difference, we break down the latency into cold-start and warm-up requests with sub-stages. For cold-start requests, we report: *end-to-end* (i.e., E2E) latency that between sending a request and receiving the result on each client; *import* time for importing the serving dependencies (e.g., TF1.15), *download* time for downloading the model (e.g., MobileNet) from cloud storage, *load* time for loading the model into the serving runtime, and *predict* time for calculating the model inference. For warm-up requests, we only report the E2E latency and predict time as the other sub-stages are not executed.

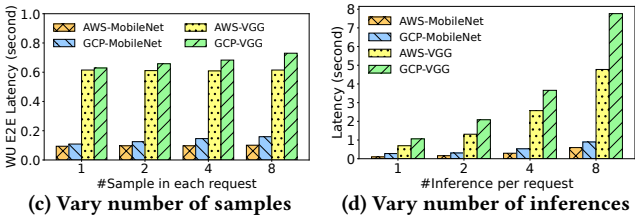
Figure 10 shows the breakdown comparison between the two serverless platforms given MobileNet and ALBERT with workload-120. We observe that the import sub-stage takes around 4-5 seconds and dominates the E2E cold-start latency for the two models on both platforms. Meanwhile, AWS-Serverless is faster than GCP-Serverless for all sub-stages (e.g., 1.89*s* and 1.34*s* faster for ALBERT *w.r.t.* the download time and load time, respectively), which demonstrates the AWS-Serverless' superior performance. Besides, the predict time of cold-start requests is much longer than that of warm-up requests on both platforms. This may be because the TensorFlow runtime has components that are lazily initialized, which can cause a higher latency for the first request sent to a loaded model [21].

Second, the long cold-start time also leads to over-provisioning problem [57], i.e., more instances are created than needed, as the service is still insufficient during the instance creation process, so that serverless platforms keep starting instances until the service is available. Figure 11 compares the number of active instances for the three models under workload-40. We observe that both platforms can scale very fast (i.e., up to hundreds of instances in one minute) to handle the bursty requests. However, GCP-Serverless always creates much more instances than AWS-Serverless. For example, for VGG on GCP-Serverless, we see around 100 instances created during the first request peak, while only 50 instances are needed during the second request peak (i.e., timestamp 5-6 minutes). In other words, many instances are over-provisioned. This problem is moderate on AWS-Serverless. Hence, the cost of AWS-Serverless is lower, and the advantage is more obvious when the model is more complex, or the workload is higher.

In-depth analysis. We further conduct a set of micro-benchmark experiments with workload-120 to study the behaviors of serverless



(a) Vary container size **(b) Vary downloaded size**



(c) Vary number of samples **(d) Vary number of inferences**

Figure 12: In-depth analysis with workload-120.

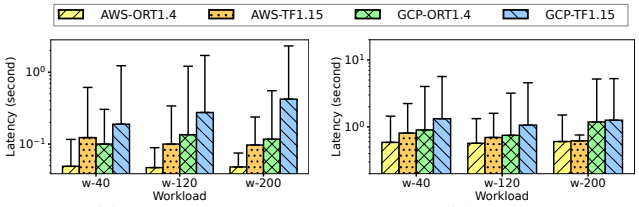
platforms. Specifically, we fix other sub-stages whenever possible and tune various inputs that may affect the performance.

First, Figure 12a shows the results for varying container sizes. We inject dummy files or dependencies with different sizes into the base images on both platforms. The base images are 1238MB on AWS-Serverless and 920MB on GCP-Serverless. We find that container size does not affect the E2E cold-start latency significantly. For instance, for MobileNet on GCP-Serverless, if we eliminate the impact of other sub-stages (i.e., import, download, load, and predict), the average remaining time of the base, base+0.5GB, base+1.0GB, and base+1.5GB containers are 1.213*s*, 1.303*s*, 1.267*s*, and 1.386*s*, respectively. This time includes the request and result transmission through the network and running the instance via the docker container. Nevertheless, the change is only 0.1*s*-0.2*s* on average. After careful analysis, we found that only around 1%-2% E2E cold-start latency is much larger than the others. For example, 9 out of 738 cold-start requests consume more than 20*s* for MobileNet with the base image. We speculate that the reason could be similar to that on OpenWhisk [42]. That is, the platform takes longer to run the first instance on a physical machine, i.e., it needs to pull the image from cloud storage. While for subsequent instances started on the same machine, the platform can run directly without pulling [56].

Second, we evaluate the impact of downloaded size by downloading extra dummy data besides the serving model, as shown in Figure 12b. The E2E cold-start latency goes up as the downloaded size increases on both platforms. However, AWS-Serverless is much faster than GCP-Serverless regarding the download time, and the gap is more significant given larger dummy data. For instance, AWS-Serverless takes an extra 2.39*s* to download 300MB of dummy data besides the ALBERT model, while GCP-Serverless takes 10.06*s*. This shows AWS-Serverless' higher downloading performance.

Third, Figure 12c illustrates the results for varying input sizes. Specifically, we pack more samples in each client request but only predict one sample inside the function (i.e., fix the prediction sub-stage). We can see that a larger input size can slightly increase the E2E warm-up latency as more data is transmitted through the network. However, its effect on the E2E warm-up latency is minor.

Fourth, we evaluate the impact of the predict sub-stage. We fix one sample in each request but execute the inference multiple times inside the function. Figure 12d reports the results. We observe that the overall latency grows significantly when the number of



(a) MobileNet (b) VGG

Figure 13: Runtime comparison: latency w.r.t. workloads, the error bar denotes the standard deviation of latency.

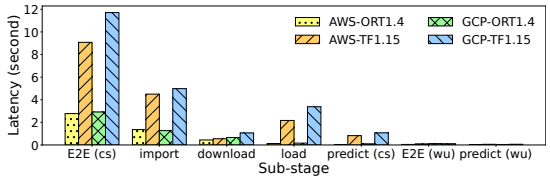


Figure 14: Breakdown comparison of different runtimes.

inferences increases. This is expected because the prediction sub-stage affects both cold-start and warm-up requests, and it dominates the overall latency when the predict time is large.

Takeaway: currently, AWS-Serverless' performance is better than that of GCP-Serverless. Data scientists could select AWS-Serverless as the primary platform for performant and cost-effective serving, especially with a complex model or under a high workload. Furthermore, among the factors that may affect the platforms' performance, the downloaded size and predict time are more significant.

5.2 Serving Runtime

Then, we evaluate the impact of serving runtime. Specifically, we compare OnnxRuntime1.4 (ORT1.4) with TensorFlow1.15 (TF1.15). Figure 13 and Table 2 report the average overall latency and cost for MobileNet and VGG, respectively. There are two main observations.

First, model serving with ORT1.4 is more efficient than with TF1.15 on both platforms. It has up to 2.51× lower in latency and 4.55× lower in cost for AWS-Serverless, and 3.61× and 1.97× for GCP-Serverless, respectively. The reason is two-fold. One is that the E2E cold-start latency is significantly reduced. Figure 14 details the breakdown comparison between the two runtimes for MobileNet with workload-120. Specifically, the E2E cold-start latency drops from 9.08s to 2.775s on AWS-Serverless, and from 11.71s to 2.917s on GCP-Serverless. Note that the container size for ORT1.4 is 391MB for MobileNet on AWS-Serverless, compared to 1238MB for TF1.15. Though the container size has little effect on the cold-start latency (as discussed in Section 5.1), we observe that using ORT1.4 greatly reduces the import time and load time. The other reason is that ORT1.4 is optimized for improved model inference, whereas TF1.15 is the standard runtime with a relatively longer execution time per request. For example, if analyzing the predict time of warm-up requests for MobileNet on GCP-Serverless, the average predict time is 0.061s for TF1.15, while that for ORT1.4 is 0.043s.

Second, the improvement for using ORT1.4 on MobileNet is more significant than on VGG. For instance, for MobileNet on GCP-Serverless, ORT1.4 is up to 3.61× faster than TF1.15 and 1.97× lower in cost, while for VGG, the improvements are 1.47× and 1.32×, respectively. The rationale is that the cold-start time dominates the average latency and cost for the MobileNet model as the actual execution time per request is very short. While for the VGG model, the actual execution time is longer, which has a relatively large impact on the performance and cost, leading to a moderate improvement.

Takeaway: data scientists should consider a smaller and faster serving runtime (e.g., ORT) on serverless platforms as the first choice as long as that runtime supports the desired model. This can greatly improve the performance and save cost.

Table 2: Costs for serverless serving with ORT1.4

Serving Services		workload-40	workload-120	worload-200
AWS-Serverless	MobileNet	$0.011	$0.037	$0.062
	VGG	$0.322	$0.931	$1.644
GCP-Serverless	MobileNet	$0.047	$0.160	$0.272
	VGG	$0.383	$1.108	$2.455

5.3 Memory Size

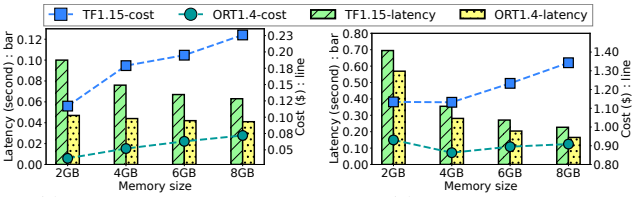
Next, we investigate the impact of several function-specific parameters on the performance of serverless model serving on AWS-Serverless. The first is memory size. Figure 15 shows the average latency and cost of MobileNet and VGG using TF1.15 and ORT1.4 with workload-120. For both models, the latency decreases as memory size increases because each request can be processed faster. Besides, the latency decrease of the VGG model is sharper than that of MobileNet as the memory size goes up. This is because the two runtimes already execute MobileNet very fast given 2GB, so that increasing memory cannot improve the performance as significantly as the VGG model. In addition, the predict time is less dominant in the end-to-end latency, e.g., for MobileNet served with ORT1.4, the average predict time is about 0.012s given 2GB, whereas the overall latency is about 0.047s, rendering its improvement (e.g., 0.009s given 4GB) insignificant to the overall performance.

Another interesting phenomenon is that a larger memory does not always increase the cost. For example, for the VGG model in Figure 15b, we observe that the 4GB memory can even slightly reduce the cost. There are two reasons. One is that a larger memory decreases the execution time per request due to the improved processing capacity (as discussed above). The other is that it also reduces the number of cold-started instances. For instance, by increasing 2GB to 4GB for VGG with ORT1.4, the number of cold-started instances is decreased from 408 to 37 in our experiment. However, when we keep increasing the memory size to 6GB or 8GB, the cost does not reduce further as the two improvements cannot compensate for the additional cost charged for larger memory.

Takeaway: if latency is the primary consideration, data scientists should choose a large memory. Besides, data scientists could run several warm-up requests beforehand and check the proportion of predict time to end-to-end latency. If the proportion is minor, they can use a small memory; otherwise, they could select a relatively large memory. A more sophisticated way is to use a memory tuning tool (e.g., [10]) to find an optimized size; nevertheless, it requires data scientists to have a better understanding of the workload.

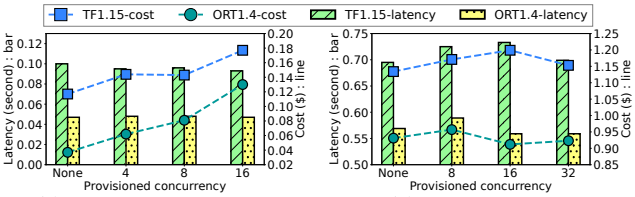
5.4 Provisioned Concurrency

We now study the impact of provisioned concurrency on AWS-Serverless, i.e., a number of instances are always keeping warm during the workload. Basically, this is a hybrid approach that combines serverless and server-based model serving. Figure 16 reports the performance and cost for MobileNet and VGG with workload-120. We observe that provisioned concurrency does not necessarily



(a) MobileNet with w-120 (b) VGG with w-120

Figure 15: Vary memory size on AWS-Serverless.



(a) MobileNet with w-120 (b) VGG with w-120

Figure 16: Vary provisioned concurrency on AWS-Serverless

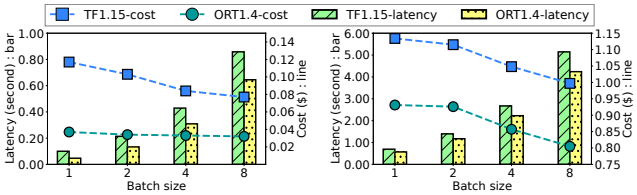
reduce average latency, and for the VGG model, the latency is even higher in some cases. This seems counter-intuitive as a portion of the requests can be forwarded to the provisioned instances and processed quickly, which should reduce the average latency. However, we found that the number of cold-started instances sometimes increases given provisioned concurrency. For example, for VGG serving with TF1.15, the number of cold-started instances are 614, 640, and 478, for provisioned concurrency 8, 16, and 32, respectively, compared to 409 for no provisioned concurrency. Since AWS-Serverless' scaling strategy is a black-box to users, we speculate that it adopts a more aggressive scaling policy when using provisioned concurrency, i.e., once it finds that the provisioned instances are insufficient, it scales more instances to handle the rest of requests than that without provisioned concurrency.

Takeaway: the current provisioned concurrency method may not be a good choice for serverless model serving. A related suggestion to cloud providers is to publicize their autoscaling policy so that users can better utilize it to improve performance and cost.

5.5 Batch Size

Finally, we investigate the impact of batch size on the performance and cost of AWS-Serverless. Given a batch size, each client sends an invocation to the serverless function only when the number of requests matches the batch size or reaches the end of the workload. Figure 17 presents the results for MobileNet and VGG with workload-120. There are two main observations. First, the average latency is approximately doubled when the batch size doubles. This is because most requests are delayed, and the execution time on serverless (i.e., response to the batched requests) grows, which increases the latency. Second, in most cases, batching could reduce the cost because: (1) the number of invocations is reduced; and (2) the request rate is reduced, resulting in a fewer number of cold-started instances. However, for MobileNet with ORT1.4, the cost reduction is insignificant since the model is simple and ORT1.4 is already able to handle requests effectively with very few instances.

Takeaway: if cost is the primary concern, data scientists could batch requests; otherwise, batching is not suggested as the response latency will increase significantly. A better way is to apply an adaptive batching strategy given the cost and latency constraints [2], but it requires a careful design to predict the request rate.



(a) MobileNet with w-120 (b) VGG with w-120

Figure 17: Vary batch size on AWS-Serverless.

6 CHALLENGES AND OPPORTUNITIES

Though existing serverless platforms have provided full-fledged functionalities that make serverless a viable model serving option for data science applications, there are still several research challenges and opportunities for serverless model serving.

The first challenge is how to address the over-provisioning problem [16, 50], which can greatly increase average latency and cost (see Section 5.1). This problem can be exacerbated when serving increasingly larger models or under higher workloads. Therefore, designing new scaling policies for serverless model serving is an important and promising research direction. Efficient policies may monitor the requests' execution time, predict the subsequent request rate, and allocate the needed instances precisely.

The second challenge is the protection of data security. With strict privacy regulations [1], data scientists now need to consider user's data security. A possible solution is based on trusted hardware [37, 58], to ensure secure computation. Unfortunately, most existing serverless platforms do not support this functionality. Besides, the data protection should be supported in a scalable manner to accommodate serverless model serving. Thus, more research work could be conducted to make serverless secure and efficient.

The third challenge is the complexity of the design space for serverless model serving. As investigated in Section 5, many factors affect both the performance and cost. The consequence is that the data scientists have more decisions to make, which affects their productivity. A potential direction is to build a navigation tool that automatically searches the design space for serverless deployment, and finds the best configuration under pre-defined constraints, such that data scientists can adopt serverless model serving effectively.

7 CONCLUSIONS

In this paper, we conduct a comprehensive comparison of serverless against other model serving systems from AWS and GCP. The evaluation results demonstrate that serverless is a viable and promising option for model serving. We further investigate the design space of serverless model serving regarding various choices and present recommendations for data scientists to better utilize serverless. Finally, we discuss challenges and opportunities in building a more practical serverless model serving system.

ACKNOWLEDGEMENTS

We thank Pan Zhou for his early contribution to this work. This research is supported by Singapore Ministry of Education Academic Research Fund Tier 3 under MOEs official grant number MOE2017-T3-1-007. Tien Tuan Anh Dinh's work is supported by Singapore University of Technology and Design's startup grant SRG-ISTD-2019-144. Meihui Zhang's work is supported by National Natural Science Foundation of China (62072033).

REFERENCES

[1] [n.d.]. Regulation (eu) 2016/679 of the european parliament and of the council of 27 april 2016 on the protection of natural persons with regard to the processing of personal data and on the free movement of such data, and repealing directive 95/46/ec (general data protection regulation). OJ, 2016-04-27. ([n. d.]).

[2] A. Ali, R. Pinciroli, F. Yan, and E. Smirni. 2020. BATCH: Machine Learning Inference Serving on Serverless Platforms with Adaptive Batching. In *SC*. 972–986.

[3] Amazon. 2021. Amazon Web Services. https://aws.amazon.com/.

[4] Amazon. 2021. AWS EC2. https://aws.amazon.com/ec2/. Accessed: 2021-06.

[5] Amazon. 2021. AWS Lambda Quotas. https://docs.aws.amazon.com/lambda/latest/dg/gettingstarted-limits.html. Accessed: 2021-10.

[6] Amazon. 2021. AWS Lambda. https://aws.amazon.com/lambda/. Accessed: 2021-10.

[7] Amazon. 2021. AWS SageMaker. https://aws.amazon.com/sagemaker/. Accessed: 2021-06.

[8] CalorieKing. 2021. CalorieKing. https://www.calorieking.com/us/en/foods/.

[9] Joao Carreira, Pedro Fonseca, Alexey Tumanov, Andrew Zhang, and Randy H. Katz. 2019. Cirrus: a Serverless Framework for End-to-end ML Workflows. In *SoCC*. 13–24.

[10] Alex Casalboni. 2021. AWS Lambda Power Tuning. https://github.com/alexcasalboni/aws-lambda-power-tuning.

[11] Marcin Copik, Grzegorz Kwasniewski, Maciej Besta, Michal Podstawski, and Torsten Hoefler. 2021. SeBS: A Serverless Benchmark Suite for Function-as-a-Service Computing. In *Middleware*.

[12] Daniel Crankshaw, Xin Wang, Guilio Zhou, Michael J. Franklin, Joseph E. Gonzalez, and Ion Stoica. 2017. Clipper: A Low-Latency Online Prediction Serving System. In *NSDI*. 613–627.

[13] Vojislav Dukic, Rodrigo Bruno, Ankit Singla, and Gustavo Alonso. 2020. Photons: lambdas on a diet. In *SoCC*. 45–59.

[14] Lang Feng, Prabhakar Kudva, Dilma Da Silva, and Jiang Hu. 2018. Exploring Serverless Computing for Neural Network Training. In *CLOUD*. 334–341.

[15] Wolfgang Fischer and Kathleen S. Meier-Hellstern. 1993. The Markov-Modulated Poisson Process (MMPP) Cookbook. *Perform. Evaluation* 18, 2 (1993), 149–171.

[16] Alexander Fuerst and Prateek Sharma. 2021. FaasCache: keeping serverless computing alive with greedy-dual caching. In *ASPLOS*. 386–400.

[17] Google. 2021. Google AI Platform. https://cloud.google.com/ai-platform. Accessed: 2021-06.

[18] Google. 2021. Google Cloud Functions. https://cloud.google.com/functions. Accessed: 2021-10.

[19] Google. 2021. Google Cloud. https://cloud.google.com/.

[20] Google. 2021. Google Compute Engine. https://cloud.google.com/compute. Accessed: 2021-06.

[21] Google. 2021. TensorFlow Saved Model. https://www.tensorflow.org/tfx/serving/saved_model_warmup.

[22] Google. 2021. TensorFlow. https://www.tensorflow.org/.

[23] Arpan Gujarati, Sameh Elnikety, Yuxiong He, Kathryn S. McKinley, and Björn B. Brandenburg. 2017. Swayam: Distributed Autoscaling to Meet SLAs of Machine Learning Inference Services with Resource Efficiency. In *Middleware*. 109–120.

[24] Arpan Gujarati, Reza Karimi, Safya Alzayat, Wei Hao, Antoine Kaufmann, Ymir Vigfusson, and Jonathan Mace. 2020. Serving DNNs like Clockwork: Performance Predictability from the Bottom Up. In *OSDI*. 443–462.

[25] Kim Hazelwood, Sarah Bird, David Brooks, Soumith Chintala, Utku Diril, Dmytro Dzhulgakov, Mohamed Fawzy, Bill Jia, Yangqing Jia, Aditya Kalro, James Law, Kevin Lee, Jason Lu, Pieter Noordhuis, Misha Smelyanskiy, Liang Xiong, and Xiaodong Wang. 2018. Applied Machine Learning at Facebook: A Datacenter Infrastructure Perspective. In *HPCA*.

[26] Joseph M. Hellerstein, Jose Faleiro, Joseph E. Gonzalez, Johann Schleier-Smith, Vikram Sreekanti, Alexey Tumanov, and Chenggang Wu. 2019. Serverless Computing: One Step Forward, Two Steps Back. In *CIDR*.

[27] Andrew G. Howard, Menglong Zhu, Bo Chen, Dmitry Kalenichenko, Weijun Wang, Tobias Weyand, Marco Andreetto, and Hartwig Adam. 2017. MobileNets: Efficient Convolutional Neural Networks for Mobile Vision Applications. *CoRR* abs/1704.04861 (2017).

[28] Junchen Jiang, Ganesh Ananthanarayanan, Peter Bodik, Siddhartha Sen, and Ion Stoica. 2018. Chameleon: scalable adaption of video analytics. In *SIGCOMM*.

[29] Jiawei Jiang, Shaoduo Gan, Yue Liu, Fanlin Wang, Gustavo Alonso, Ana Klimovic, Ankit Singla, Wentao Wu, and Ce Zhang. 2021. Towards Demystifying Serverless Machine Learning Training. In *SIGMOD*. 857–871.

[30] Eric Jonas, Qifan Pu, Shivaram Venkataraman, Ion Stoica, and Benjamin Recht. 2017. Occupy the cloud: Distributed computing for the 99%. In *SoCC*. 445–451.

[31] Eric Jonas, Johann Schleier-Smith, Vikram Sreekanti, Chia-Che Tsai, Anurag Khandelwal, Qifan Pu, Vaishaal Shankar, Joao Carreira, Karl Krauth, Neeraja Yadwadkar, Joseph E. Gonzalez, Raluca Ada Popa, Ion Stoica, and David A. Patterson. 2019. *Cloud Programming Simplified: A Berkeley View on Serverless Computing*. Technical Report. UC Berkeley.

[32] Ana Klimovic, Yawen Wang, Christos Kozyrakis, Patrick Stuedi, Jonas Pfefferle, and Animesh Trivedi. 2018. Understanding ephemeral storage for serverless analytics. In *Usenix ATC*. 789–794.

[33] Ana Klimovic, Yawen Wang, Patrick Stuedi, Animesh Trivedi, Jonas Pfefferle, and Christos Kozyrakis. 2018. Pocket: Elastic ephemeral storage for serverless analytics. In *OSDI*. 427–444.

[34] Zhenzhong Lan, Mingda Chen, Sebastian Goodman, Kevin Gimpel, Piyush Sharma, and Radu Soricut. 2020. ALBERT: A Lite BERT for Self-supervised Learning of Language Representations. In *ICLR*.

[35] Zhaojing Luo, Sai Ho Yeung, Meihui Zhang, Kaiping Zheng, Lei Zhu, Gang Chen, Feiyi Fan, Qian Lin, Kee Yuan Ngiam, and Beng Chin Ooi. 2021. MLCask: Efficient Management of Component Evolution in Collaborative Data Analytics Pipelines. In *ICDE*. 1655–1666.

[36] Pascal Maissen, Pascal Felber, Peter G. Kropf, and Valerio Schiavoni. 2020. FaaSdom: a benchmark suite for serverless computing. In *DEBS*. 73–84.

[37] Frank McKeen, Ilya Alexandrovich, Alex Berenzon, Carlos V. Rozas, Hisham Shafi, Vedvyas Shanbhogue, and Uday R. Savagaonkar. 2013. Innovative instructions and software model for isolated execution. In *HASP*. 10.

[38] Microsoft. 2021. Microsoft Azure Machine Learning. https://azure.microsoft.com/en-us/services/machine-learning.

[39] Microsoft. 2021. Onnx Runtime. https://github.com/microsoft/onnxruntime.

[40] Ingo Müller, Renato Marroquin, and Gustavo Alonso. 2020. Lambada: Interactive Data Analytics on Cold Data Using Serverless Cloud Infrastructure. In *SIGMOD*. 115–130.

[41] Beng Chin Ooi, Kian-Lee Tan, Sheng Wang, Wei Wang, Qingchao Cai, Gang Chen, Jinyang Gao, Zhaojing Luo, Anthony K. H. Tung, Yuan Wang, Zhongle Xie, Meihui Zhang, and Kaiping Zheng. 2015. SINGA: A Distributed Deep Learning Platform. In *ACM MM*. 685–688.

[42] OpenWhisk. 2021. Open Source Serverless Cloud Platform. https://openwhisk.apache.org/.

[43] Matthew Perron, Raul Castro Fernandez, David J. DeWitt, and Samuel Madden. 2020. Starling: A Scalable Query Engine on Cloud Functions. In *SIGMOD*. 131–141.

[44] Qifan Pu, Shivaram Venkataraman, and Ion Stoica. 2019. Shuffling, Fast and Slow: Scalable Analytics on Serverless Infrastructure. In *NSDI*.

[45] Ali Rajabi and Johnny W. Wong. 2012. MMPP Characterization of Web Application Traffic. In *MASCOTS*. 107–114.

[46] Hossein Shafiei and Ahmad Khonsari. 2019. Serverless Computing: Opportunities and Challenges. *CoRR* abs/1911.01296 (2019).

[47] Vaishaal Shankar, Karl Krauth, Qifan Pu, Eric Jonas, Shivaram Venkataraman, Ion Stoica, Benjamin Recht, and Jonathan Ragan-Kelley. 2018. numpywren: serverless linear algebra. *CoRR* abs/1810.09679 (2018).

[48] Karen Simonyan and Andrew Zisserman. 2015. Very Deep Convolutional Networks for Large-Scale Image Recognition. In *ICLR*.

[49] Vikram Sreekanti, Chenggang Wu, Xiayue Charles Lin, Johann Schleier-Smith, Joseph Gonzalez, Joseph M. Hellerstein, and Alexey Tumanov. 2020. Cloudburst: Stateful Functions-as-a-Service. *Proc. VLDB Endow.* 13, 11 (2020), 2438–2452.

[50] Dmitrii Ustiugov, Plamen Petrov, Marios Kogias, Edouard Bugnion, and Boris Grot. 2021. Benchmarking, analysis, and optimization of serverless function snapshots. In *ASPLOS*. 559–572.

[51] Leonid Velikovich, Ian Williams, Justin Scheiner, Petar Aleksic, Pedro Moreno, and Michael Riley. 2018. Semantic Lattice Processing in Contextual Automatic Speech Recognition for Google Assistant. In *Interspeech*.

[52] Hao Wang, Di Niu, and Baochun Li. 2019. Distributed Machine Learning with a Serverless Architecture. In *INFOCOM*. 1288–1296.

[53] Wei Wang, Jinyang Gao, Meihui Zhang, Sheng Wang, Gang Chen, Teck Khim Ng, Beng Chin Ooi, Jie Shao, and Moaz Reyad. 2018. Rafiki: Machine Learning as an Analytics Service System. *Proc. VLDB Endow.* 12, 2 (Oct. 2018), 128–140. https://doi.org/10.14778/3282495.3282499

[54] Wei Wang, Meihui Zhang, Gang Chen, H. V. Jagadish, Beng Chin Ooi, and Kian-Lee Tan. 2016. Database Meets Deep Learning: Challenges and Opportunities. *SIGMOD Rec.* 45, 2 (2016), 17–22.

[55] Michal Wawrzoniak, Ingo Müller, Gustavo Alonso, and Rodrigo Bruno. 2021. Boxer: Data Analytics on Network-enabled Serverless Platforms. In *CIDR*.

[56] Tianyi Yu, Qingyuan Liu, Dong Du, Yubin Xia, Binyu Zang, Ziqian Lu, Pingchao Yang, Chenggang Qin, and Haibo Chen. 2020. Characterizing serverless platforms with serverlessbench. In *SoCC*. 30–44.

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

[58] Wenting Zheng, Ankur Dave, Jethro G. Beekman, Raluca Ada Popa, Joseph E. Gonzalez, and Ion Stoica. 2017. Opaque: An Oblivious and Encrypted Distributed Analytics Platform. In *NSDI*. 283–298.