SC20: International Conference for High Performance Computing, Networking, Storage and Analysis | 978-1-7281-9998-6/20/$31.00 ©2020 IEEE | DOI: 10.1109/SC41405.2020.00073

BATCH: Machine Learning Inference Serving on Serverless Platforms with Adaptive Batching

Ahsan Ali[§]
University of Nevada, Reno
Reno, NV
aali@nevada.unr.edu

Riccardo Pinciroli[§]
William and Mary
Williamsburg, VA
rpinciroli@wm.edu

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

Evgenia Smirni
William and Mary
Williamsburg, VA
esmirni@cs.wm.edu

***Abstract*—Serverless computing is a new pay-per-use cloud service paradigm that automates resource scaling for stateless functions and can potentially facilitate bursty machine learning serving. Batching is critical for latency performance and cost-effectiveness of machine learning inference, but unfortunately it is not supported by existing serverless platforms due to their stateless design. Our experiments show that without batching, machine learning serving cannot reap the benefits of serverless computing. In this paper, we present BATCH, a framework for supporting efficient machine learning serving on serverless platforms. BATCH uses an optimizer to provide inference tail latency guarantees and cost optimization and to enable adaptive batching support. We prototype BATCH atop of AWS Lambda and popular machine learning inference systems. The evaluation verifies the accuracy of the analytic optimizer and demonstrates performance and cost advantages over the state-of-the-art method MArk and the state-of-the-practice tool SageMaker.**

***Index Terms*—Machine-learning-as-a-service (MLaaS), Inference, Serving, Batching, Cloud, Serverless, Service Level Objective (SLO), Cost-effective, Optimization, Modeling, Prediction**

I. Introduction

Serverless (also referred to as Function-as-a-Service (FaaS) or cloud function services) is an emerging cloud paradigm provided by almost all public cloud service providers, including Amazon Lambda [1], IBM Cloud Function [2], Microsoft Azure Functions [3], and Google Cloud Functions [4]. Serverless offers a true pay-per-use cost model and hides instance management tasks (e.g., deployment, scaling, monitoring) from users. Users only need to provide the function and its trigger event (e.g., HTTP requests, database uploads), as well as a single control system parameter *memory size* that determines the processing power, allocated memory, and networking performance of the serverless instance. Intelligent transportation systems [5], IoT frameworks [6–8], subscription services [9], video/image processing [10], and machine learning tools [11–13] are already being deployed on serverless.

Machine Learning (ML) Serving. ML applications have typically three phases: model design, model training, and model inference (or model serving).[1]. In the model design phase, the model architecture is designed manually or through automated methods [14, 15]. Then, the crafted model with initial parameters (weights) is trained iteratively until convergence. A trained model is published in the cloud to provide inference services, such as classification and prediction. Among the three phases, model serving has a great potential to benefit from serverless because the arrival intensity of classification/prediction requests is dynamic and their serving has strict latency requirements.

Serverless vs. IaaS for ML Serving. Serverless computing simplifies the deployment process of ML Serving as application developers only need to provide the source-code of their functions without worrying about virtual machine (VM) resource management, such as (auto)scaling and load balancing. Despite the great capabilities of serverless, existing works show that in public clouds, the serverless paradigm for ML serving is more expensive compared to IaaS [16, 17]. Recent works have shown that it is possible to use serverless to improve the high cost for ML serving [16, 18] but ignore one important feature of serving workloads in practice: burstiness [19,20]. Bursty workloads are characterized by time periods of low arrival intensities that are interleaved with periods of high arrival intensities. Such behavior makes the VM-based approach very expensive: over-provisioning is necessary to accommodate sudden workload surges (otherwise, the overhead of launching new VMs, which can be a few minutes long, may significantly affect user-perceived latencies). The scaling speed of serverless can solve the sudden workload surge problem. In addition, during low arrival intensity periods, serverless contributes to significant cost savings, thanks to its pay-per-use cost model.

Serverless for ML Serving: Opportunities and Challenges. Another important factor that heavily impacts both cost and performance of ML serving inference is *batching* [16,21]. With batching, several inference requests are bundled together and served concurrently by the ML application. As requests arrive, they are served in a non-work-conserving way, i.e., they wait in the queue till enough requests form a batch. In contrast to batch workloads in other domains that are typically treated as background tasks [22–25], batching for ML inference serving is done online, as a foreground process. Batching dramatically improves inference throughput as the input data dimension determines the parallelization optimization opportunities when executing each operator according to the computational graph [26,27]. Provided that the monetary cost of serverless on public cloud is based on invocations, a few large batched requests are cheaper than many small individual requests. Therefore, judicious parameterization of batching can potentially improve performance while reducing cost.

[§]Both authors contributed equally to this research.
[1]We use the terms model serving and model inference interchangeably

SC20, November 9-19, 2020, Is Everywhere We Are
978-1-7281-9998-6/20/$31.00 ©2020 IEEE

Authorized licensed use limited to: TIANJIN UNIVERSITY. Downloaded on November 11,2022 at 12:32:55 UTC from IEEE Xplore. Restrictions apply.

Due to the stateless property of serverless design, i.e., no data (state) is kept between two invocations, batching is not supported as a native feature within the serverless paradigm. An additional challenge is that batching introduces two configuration parameters: *batch size* (i.e., the maximum number of requests to form a batch) and *timeout* (i.e., the maximum time it allows to wait for requests to form the batch). These parameters need to be adjusted on the fly according to the arrival intensity of inference requests to meet inference latency SLOs. The performance and cost effectiveness of batching depends strongly on the judicious choice of the above parameters. No single parameter choice can work optimally for all workload conditions.

Our Contribution. Here, we address the above challenges by developing an autonomous framework named BATCH [28]. The first key component of BATCH is a dispatching buffer that enables batching support for ML serving on serverless clouds. BATCH supports automated adaptive batching and its associated parameter tuning to provide tail latency performance guarantees and cost optimization while maintaining low system overhead. We develop a lightweight profiler that feeds a performance optimizer. The performance optimizer is driven by a new analytical methodology considers the salient characteristics of bursty workloads, serverless computing, and ML serving to estimate inference latencies and monetary cost.

We prototype BATCH atop of AWS Lambda and evaluate its effectiveness using different ML serving frameworks (TensorFlow Serving and MXNet Model Server) and image classification applications (MoBiNet, Inception-v4, ResNet-18, ResNet-50, and ResNet-v2) driven both by synthetic and real workloads [29,30]. Results show that the estimation accuracy of the analytical model is consistently high (average error is lower than 9%) independent of the considered system configurations (i.e., framework, application, memory size, batch size, and timeout). Controlling the configuration parameters on the fly, BATCH outperforms SageMaker [31] (the state-of-the-practice), MArk [16] (the state-of-the-art), and vanilla Lambda from both cost and performance viewpoints. More importantly, BATCH supports a full serverless paradigm for online ML serving, enabling fully automated and efficient SLO-aware ML serving deployment on public clouds.

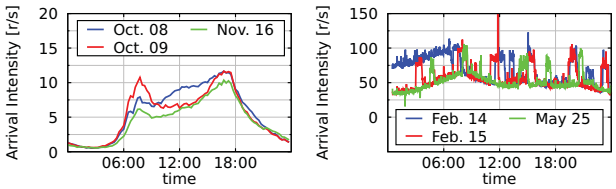
II. Motivation and Challenges

We discuss the potential advantages of using serverless for ML serving and summarize its open challenges.

A. ML Serving Workload is Bursty

Workload burstiness is omnipresent in non-laboratory settings [32–35]. Past work has showed that if burstiness is not incorporated into performance models, the quality of their prediction plummets [36, 37].

Fig. 1(a) plots the arrival intensity of cars passing through the New York State (NYS) Thruway during three business days in the fourth quarter of 2018 [29]. This trace represents a typical workload of an image recognition application that detects the plate of vehicles that pass under a checkpoint



(a) NYS Thruway (b) Twitter

Fig. 1: Real-world traces from [29] and [30].

[18]. The three days have distinct arrival intensities that are non-trivial to accurately predict. Distinct traffic peaks are also observed in a Twitter trace [30] that represents a typical arrival of tweets processed for sentiment analysis that has been used for ML serving elsewhere [16]. Fig. 1(b) shows the arrival intensity of three distinct days in the first semester of 2017.

Observation #1. ML inference application often have bursty arrivals. Burstiness must be incorporated into the design of any framework for performance optimization.

B. Why Serverless for ML Serving

There are two types of approaches to cope with sudden workload surges: resource over-provisioning and autoscaling. For bursty workloads, resource over-provisioning is not cost-effective as the difference between high and low intensities can be dramatic and lots of paid computing resource is left idle for extensive periods.

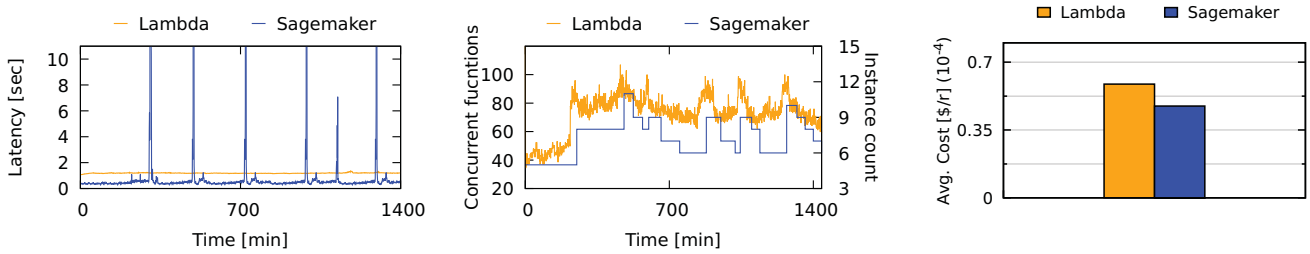
Autoscaling is the current industry solution: Amazon's SageMaker [31] facilitates ML tasks and supports AWS autoscaling [38]. With SageMaker, users can define scaling metrics such as when and how much to scale.

We examine Sagemaker's scaling in Fig. 2. Here, arrivals are driven by the Twitter trace illustrated in Fig. 1(b) (May 25, 2017) and the inference application is Inception-v4. The figure shows that SageMaker's scaling is very slow, as it requires several minutes before responding to an arrival intensity surge, which is not acceptable as requests' performance during this period suffers, see the latency results of Fig. 2(a). SageMaker being SLO-oblivious cannot ensure quality of service.

Serverless is potentially a good solution for solving the resource allocation problem, thanks to its automated and swift resource scaling capabilities. Take AWS's serverless platform Lambda (also referred in this paper as Vanilla Lambda) for example: Fig. 2(b) demonstrates the number of instances launched overtime, we can see that the number of instances closely follows the bursty arrival intensity shown in Fig. 1(b). SageMaker stays almost flat due to its long reaction window.

Another advantage of serverless compared to IaaS (e.g., SageMaker) is its true pay-per-use cost model as Lambda charges based on the number of invocations, which can potentially reduce cost, especially during time periods with few arrival. Here, we use the term *cost* to refer to the monetary cost per request, i.e., total cost over the number of executed requests. The total cost is calculated as in [39]

Authorized licensed use limited to: TIANJIN UNIVERSITY. Downloaded on November 11,2022 at 12:32:55 UTC from IEEE Xplore. Restrictions apply.



(a) Latency (b) Instance count (c) Cost

Fig. 2: Performance of inference using Inception-v4 deployed on SageMaker (with instance type *c5.4xlarge*) and Lambda. The number of instances used by Sagemaker varies between 5 to 11 based on the workload intensity.

without considering the free tier (i.e., free invocations and compute time available every month):

$$C_{Lambda} = (S \cdot M \cdot I) \cdot K_1 + I \cdot K_2, \quad (1)$$

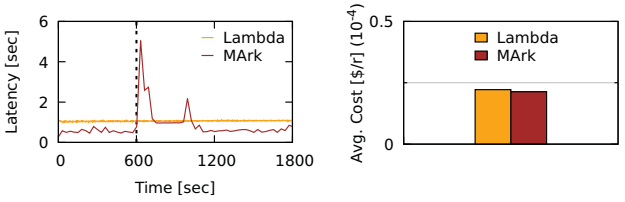
where S is the length of the function call (referred as *batch service time* here), M is the memory allocated for the function, I is the number of calls to the function that decreases when requests are batched together, K_1 (i.e., $1.66667 \cdot 10^{-5}$ \$/GB-s) is the cost of the memory, and K_2 (i.e., $2 \cdot 10^{-7}$ \$) is the cost of each call to the function. This is different from the AWS EC2 cost, i.e., $C_{EC2} = K \cdot H$, that only accounts for the cost per hour of an instance (K) and the number of power-on hours of the instance (H).

Observation #2. Compared to IaaS solutions (autoscaling), serverless computing is very agile, which is critical for performance during time periods of bursty workload conditions. In addition, the quick scale down and pay-per-use cost model contributes to the cost-effectiveness of serverless.

C. The State of the Art

Using serverless for ML serving has been explored in recent works [16–18]. The above works conclude that using serverless for ML serving is too expensive compared to IaaS-based solutions. Our experimental results are consistent with current literature findings, e.g., in Fig. 2(c), Lambda's cost is higher than SageMaker. To address the cost issue, MArk [16] has introduced a hybrid approach of using both AWS EC2 and serverless, where serverless is responsible for handling arrival bursts. MArk can dramatically improve the IaaS solution but is still insufficient in the serverless setting. First, it uses machine learning to predict the arrival intensity, this is too expensive for any practical use. Second, even though MArk is SLO-aware, it needs an observation window for the decision of whether to use serverless or not, therefore it reacts to arrival bursts after a lag, which impacts tail latency. Fig. 3(a) shows this outcome. With serverless, there are no such reaction delays as scaling is automatic.

Observation #3. While state-of-the-art systems employ a hybrid approach of using serverless and IaaS for ML serving, it suffers from slow reaction time and consequently long latency tails.



(a) Latency (b) Cost

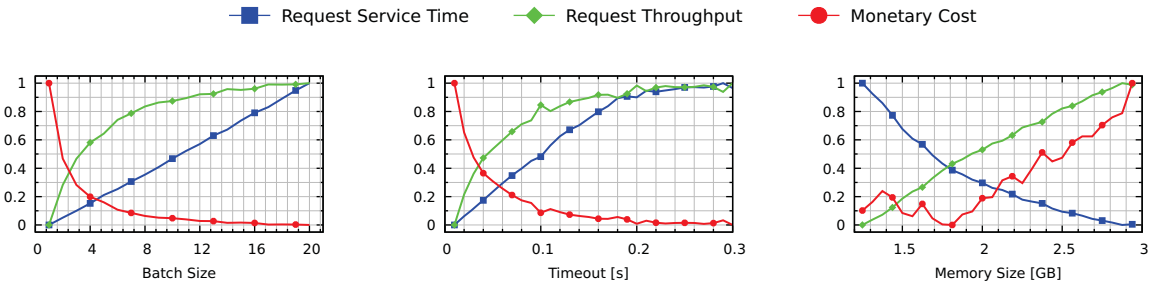
Fig. 3: 95th percentile latency and cost for Inception-v4 during unexpected workload surge with MArk and Lambda. The vertical dashed line marks the workload surge.

D. Serverless and Batching Configurations

One important factor that existing work ignores is batching. By batching several requests together, the input dimension of inference significantly increases, which provides great opportunities for parallelization. Batching results in more efficient inference since it enables a better exploitation of modern hardware [40]. In addition, as serverless charges users based on the number of invocations, a few large batched requests result in lower cost than many small individual ones.

Batching introduces two tuning parameters: batch size and timeout [13, 16]. These two parameters need to be adjusted based on the request arrival intensity. Changing these two parameters also affects the optimal memory size, the only parameter that controls the performance and cost of serverless in public cloud, since the memory required to serve a batch increases with the batch size. For example, the minimum memory required to serve a batch of size 1, when the inference application is ResNet-v2, is 1280 MB, while 1664 MB (i.e., 30% more) memory is required for processing a batch of size 20. To demonstrate the above, we do sensitivity analysis by adjusting the batching parameters and memory size and show results in Fig. 4. Fig. 4 depicts the normalized (min-max) average request service time (blue line), request throughput (green line), and monetary cost (red line). In Fig. 4(a), the batch size varies from 1 to 20, the memory size is set to 3008 MB, and the timeout is assumed to be long enough (i.e., 1 hour) to allow the system to form a batch with maximum size. Although cost reduces and throughput increases when

Authorized licensed use limited to: TIANJIN UNIVERSITY. Downloaded on November 11,2022 at 12:32:55 UTC from IEEE Xplore. Restrictions apply.



(a) Timeout = 1 hr, Memory = 3008 MB (b) Batch Size = 20, Memory = 3008 MB (c) Batch Size = 5, Timeout = 1 hr

Fig. 4: Effect of batch size, timeout, and memory size on average request service time, request throughput, and monetary cost of ResNet-v2 deployed on AWS Lambda. Service time, throughput, and cost are normalized (min-max).

batch size increases, request processing times suffer as earlier requests need to wait until the entire batch is formed. Similar considerations may be drawn when timeout varies, see Fig. 4(b). It is worth noting that for the considered configuration, the timeout effect on the system performance decreases when it is longer than 0.2 seconds. Long timeouts allow reaching the maximum batch size easier, especially if the batch size is small. Fig. 4(c) depicts the system performance as a function of the memory size, when the batch size is 5, and the timeout is long enough to guarantee that all requests are collected (e.g., 1 hour). Here, monetary cost is not monotonous since it depends on both memory size and the batch service time, see Eq. (1). In Fig. 4(c), both these parameters are varying at the same time given that the processing time decreases with more memory.

Observation #4. The effectiveness of batching strongly depends on its parameterization.

E. Challenges of ML Serving on Serverless

Despite the great potential of using serverless for ML serving, there are several challenges that need to be addressed to enable efficient ML serving on serverless:

No batching support: As shown in Section II-C, batching can drastically improve the performance [21] and monetary cost of ML serving. However, existing serverless platforms in public clouds do not support this important feature due to its stateless design, i.e., no data (state) can be stored between two invocations. To solve this challenge, we extend the current serverless design to allow for a dispatching buffer so that requests can form a batch before processing.

SLO-oblivious: ML serving usually has strict latency SLO requirements to provide good user experience. Past studies have shown that if latency increases by 100 ms, revenue drops by 1% [41]. Existing serverless platforms in public clouds are SLO-oblivious and do not support user specified latency requirements. We are motivated to develop a performance optimizer that can support strict SLO guarantees and concurrently optimize monetary cost.

Adaptive parameter tuning: To support user defined SLO while minimizing monetary cost, memory size (the single serverless parameter) and batching parameters need to be dynamically adjusted (optimized) according to the intensity of arrival requests. There is no existing method that can optimize the above parameters on-the-fly. We are motivated to develop an optimization methodology that can continuously tune these parameters to provide SLO support.

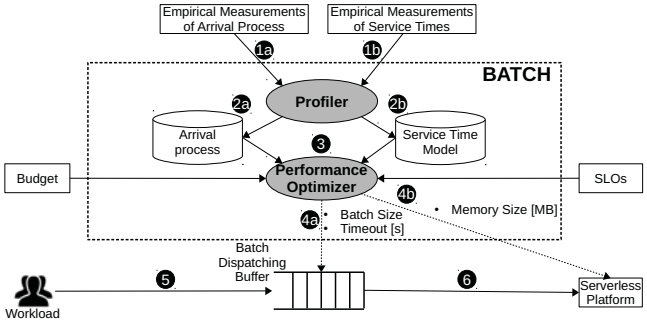


Fig. 5: Overview of BATCH.

Lightweight: Given that the cost model of serverless is pay-per-use, the aforementioned optimization methodology needs to be lightweight. Such requirement excludes learning or simulation based approaches. We are thus motivated to build the optimization methodology using analytical models.

III. BATCH Design

BATCH determines the best system configuration (i.e., memory size, maximum batch size, and timeout) to meet user-defined SLOs while reducing the cost of processing requests on a serverless platform. The workflow of BATCH is shown in Fig. 5. Initially, BATCH feeds the *Profiler* with empirical measurements of the workload arrival process 1a and service times 1b. The *Profiler* uses the KPC-toolbox [42] to fit the sequence of arrivals into a stochastic process 2a and uses simple regression to capture the relationship between system configuration and request service times 2b. Using as inputs the fitted arrival process, request service times for different system configurations, the monetary budget, and the user SLO, BATCH uses an analytic model implemented within the *Performance Optimizer* component to predict the distribution of latencies 3 and determines the optimal serverless configuration to reach specific performance/cost goals while

Authorized licensed use limited to: TIANJIN UNIVERSITY. Downloaded on November 11,2022 at 12:32:55 UTC from IEEE Xplore. Restrictions apply.

complying with a published SLO. The optimal batch size and timeout are communicated to the *Buffer* (4a), while the optimal memory size is allocated to the function deployed on the serverless platform (4b). The *Buffer* uses the parameters provided by the *Performance Optimizer* to group incoming requests into batches (5). Once the batch size is reached or the timeout expires, the accumulated requests are sent from the *Buffer* to the serverless platform for processing (6). Further details are provided for each component of BATCH below.

A. Profiler

To determine the workload arrival process, the *Profiler* observes the inter-arrival times of incoming requests. BATCH uses the KPC-Toolbox [42] to fit the collected arrival trace into a Markovian Arrival Process (MAP) [43], a class of processes that can capture burstiness. To profile the inference time on the serverless platform, the *Profiler* measures the time required to process batches of different sizes for certain amounts of allocated memory. BATCH derives the batch service time model using multivariable polynomial regression [44] and assuming that inference times are deterministic. Past work has shown that inference service times are deterministic [45], our experiments (see Section VI-B) confirm this.

B. Buffer

Since serverless platforms do not automatically allow processing multiple requests in a single batch, BATCH implements a *Buffer* to batch together requests for serving. The performance optimizer determines the optimal batch size based on the arrival process, service times, SLO, and budget. The performance optimizer also defines a timeout value that is used to avoid waiting too long for collecting enough requests. Therefore, a batch is sent to the serverless platform as soon as either the maximum number (batch size) of requests is collected or timeout expires.

C. Performance Optimizer

The *Performance Optimizer* is the core component of BATCH. It uses the arrival process and service time (both estimated by the profiler) to predict the time required to serve the incoming requests. Using the SLO and the available budget (both provided by the user), the performance optimizer determines which system configuration (i.e., memory size, batch size, and timeout) allows minimizing the cost (latency) while meeting SLOs on system performance (budget). To solve this optimization problem, BATCH uses an analytical approach that allows predicting the request latency distribution.

We opt for an analytical approach as opposed to simulation or regression for the following reasons. Analytical models are significantly *lightweight*, i.e., they do not require extensive repetitions to obtain results within certain confidence intervals as simulation does. Equivalently, they do not require extensive profiling experiments that regression traditionally requires.

IV. Problem Formulation and Solution

Since SLOs are typically defined as percentiles [46,47], the model must determine the request latency distribution while accounting for memory size, batch size, timeout, and arrival intensity. *Since the model is analytical, no training is required.*

The challenges for developing an analytical model here are three-fold: 1) the model needs to effectively capture burstiness in the arrival process for a traditional infinite server [48] that can model the serverless paradigm (i.e., there is no inherent waiting in a queue), 2) the model needs to effectively capture a *deterministic* service process which is challenging [45, 49], and 3) needs to predict performance in the form of latency percentiles [46, 47], this is very challenging since analytical models typically provide just averages [45, 50]. In the following, we give an overview of how we overcome the above challenges.

Problem Formulation:
BATCH optimizes system cost by solving the following:

$$\begin{aligned} &\text{minimize} && \mathit{Cost} \\ &\text{subject to} && P_i \leq \text{SLO}, \end{aligned} \tag{2}$$

where *Cost*, given by Eq. (1), is the price that the system charges to process the incoming requests and P_i is the ith-percentile latency that must be shorter than the user-defined SLO. BATCH can minimize the request latency by solving:

$$\begin{aligned} &\text{minimize} && P_i \\ &\text{subject to} && \text{Cost} \leq \text{Budget}, \end{aligned} \tag{3}$$

where *Budget* is the maximum price for serving a single request. These optimization problems allow minimizing either latency or cost (at the expense of the other measure, which must comply with the given target). The optimization in Eqs. (2) and (3) are solved via exhaustive search within a space that is quickly built by the analytical model (see Section V).

Analytical Model:
In order to determine the optimal system configuration, (i.e., the memory size, the maximum batch size B, and the timeout T) BATCH first evaluates the distribution of jobs in the *Buffer* by observing the arrival process. The probability that a batch of size k is processed by the serverless function is equal to the probability that $k \leq B$ requests are into the buffer by time T. In the following, we show how the prediction model operates with different arrival processes.

Poisson arrival process. We start with the simplest case where the arrival process is a Poisson distribution with rate λ and it is represented by the following $B \times B$ matrix:

$$\boldsymbol{Q} = \begin{bmatrix} -\lambda & \lambda & & \\ & \ddots & \ddots & \\ & & -\lambda & \lambda \\ & & & 0 \end{bmatrix}. \tag{4}$$

The state space of the buffer is represented by the continuous time Markov chain (CTMC) shown in Fig. 6. Since the timeout starts when the first request arrives at the buffer, each state i of the CTMC represents the buffer with $i+1$ requests. No more

Authorized licensed use limited to: TIANJIN UNIVERSITY. Downloaded on November 11,2022 at 12:32:55 UTC from IEEE Xplore. Restrictions apply.

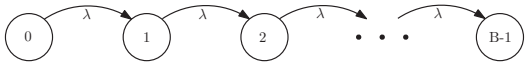


Fig. 6: Buffer state space for a Poisson arrival process.

than $B-1$ requests are collected into the buffer. To determine the probability that k requests besides the first one arrive into the buffer by the timeout T, $\pi_k(T)$, we solve the equation [51]:

$$\boldsymbol{\pi}(T) = \boldsymbol{\pi}(0) e^{\boldsymbol{Q}T} \tag{5}$$

where $\boldsymbol{\pi}(T) = (\pi_0(T), \pi_1(T), \ldots, \pi_k(T), \ldots, \pi_{B-1}(T))$ is the batch size distribution, $\boldsymbol{\pi}(0) = (1,0,0,\ldots,0)$ is the initial state probability vector, and $e^{\boldsymbol{Q}T}$ is the matrix exponential:

$$e^{\boldsymbol{Q}T} = \sum_{i=0}^{\infty} \boldsymbol{Q}^i \cdot \frac{T^i}{i!}. \tag{6}$$

Solving Eq. (5), we obtain:

$$\boldsymbol{\pi}(T) = \begin{cases} \frac{(\lambda T)^k}{k!} e^{-\lambda T} & 0 \le k < B-1 \\ 1 - \sum_{i=0}^{B-1} \pi_i(T) & k = B-1. \end{cases} \tag{7}$$

The probability that a request is processed in a batch of size $k+1$ at the end of the timeout, $\rho_{k+1}(T)$, is computed as:

$$\rho_k(T) = \frac{(k+1)\cdot\pi_k(T)}{\sum_{i=0}^{B-1}(i+1)\cdot\pi_i(T)}, \tag{8}$$

where $\pi_k(T)$ is weighted by the number of requests in the batch and normalized such that $0 \le \rho_k(T) \le 1$. When the batch service time is deterministic as it is common in serving [45], the CDF of the latency $F_R(t) = P(R \le t)$, can be computed as:

$$F_R(t) = \begin{cases} 0 & t < S_1 + T & ① \\ \frac{t-S_k}{T} \cdot \rho_k + \sum_{i=1}^{k-1} \rho_i & S_k < t < S_k + T, k < B & ② \\ \frac{t-S_B}{T} \cdot \rho_B + \sum_{i=1}^{B-1} \rho_i & S_B < t < S_B + \tau, k = B & ③ \\ \sum_{i=1}^{k} \rho_i & S_k + T \le t \le S_{k+1} & ④ \\ 1 & t > S_B + \tau & ⑤ \end{cases} \tag{9}$$

where S_k is the the service time of a batch of size k. Specifically for each of the above cases:

① If $B > 1$, the minimum request latency is $S_1 + T$ since a request served in a batch with size $k = 1$ must wait all the timeout before being processed with time S_1.

② If the batch size k is smaller than B, then the request latency t is between S_k and $S_k + T$. In this time interval, we assume that the timeout (to be added to the processing time S_k) is uniformly distributed (between 0 and T). If $T \gg S_{k+1} - S_k$, the above assumption (uniform distribution) may lead to model inaccuracies.

③ If $B-1$ requests are collected besides the first one, then the batch reaches the maximum allowed size and it is immediately sent to the serverless platform (i.e., without waiting for the timeout expiration). The time, $\tau < T$, to collect $B-1$ requests with a Poisson arrival process is:

$$\tau = (B-1)/\lambda. \tag{10}$$

④ If $T < S_{k+1} - S_k$, where $1 \le k < B$, the request latency cannot be in the interval $(S_k + T, S_{k+1})$. If a request is included in a batch of size k, it is processed in a time shorter than $S_k + T$ time units, while it takes at least S_{k+1} time units if it is included in a batch of size $k+1$. For this reason, the CDF in this time interval is flat.

⑤ Since the larger the batch the longer the service time, and the maximum batch size is B, all requests are served in a time shorter than $S_B + \tau$.

Eq. (9) accounts for the maximum batch size (B) and timeout (T) and is used by BATCH to model the request latency CDF. Recall that the memory affects the batch service time (S_k) as described in Section III-A.

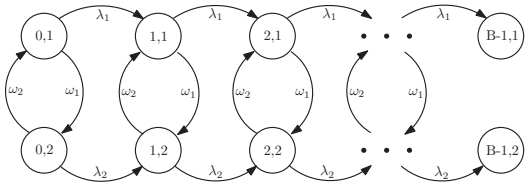


Fig. 7: Buffer state space for a MMPP(2) arrival process.

MMPP(2) arrival process. A Markov-modulated Poisson Process (MMPP) [52] may be used to capture burstiness. A MMPP generates correlated samples by alternating among m Poisson processes, each one with rate λ_i for $1 \le i \le m$ [53]. A MMPP(1) is a Poisson process with rate λ_1. MMPPs have two types of events: *observed* and *hidden* events [54]. The former determines the generation of a request with rate λ_i, the latter makes the MMPP change its phase with rate ω_i. A MMPP(2), is characterized by two phases (e.g., a quiet and a high-intensity), where it stays for exponentially distributed periods with mean rates ω_1 and ω_2, respectively [55]. A MMPP(2) is defined by the $2B \times 2B$ matrix:

$$\boldsymbol{Q} = \begin{bmatrix} \boldsymbol{D0} & \boldsymbol{D1} & & \\ & \ddots & \ddots & \\ & & \boldsymbol{D0} & \boldsymbol{D1} \\ & & & \boldsymbol{0} \end{bmatrix}, \tag{11}$$

where:

$$\begin{aligned} \boldsymbol{D0} &= \begin{bmatrix} -(\lambda_1+\omega_1) & \omega_1 \\ \omega_2 & -(\lambda_2+\omega_2) \end{bmatrix}, \\ \boldsymbol{D1} &= \begin{bmatrix} \lambda_1 & 0 \\ 0 & \lambda_2 \end{bmatrix}, \quad \text{and} \quad \boldsymbol{0} = \begin{bmatrix} 0 & 0 \\ 0 & 0 \end{bmatrix}, \end{aligned} \tag{12}$$

and its CTMC is shown in Fig. 7, where each state (i,j) describes the system when there are $i+1$ requests into the buffer and the arrival process is in phase j (since this is a MMPP(2), j values are 1 or 2 only). The request latency distribution is derived as for the Poisson arrival process with modifications. Differently from the Poisson case, the MMPP(2) process may be in any of its phases (i.e., 1 or 2) when the first request of each batch arrives to the buffer. Hence, the initial state, $\boldsymbol{\pi}(0)$, in Eq. (5) must be replaced in the case of a MMPP(2) process. To derive the probability that the arrival process is in phase $m = \{1,2\}$ when the first request of a batch arrives to the buffer

Authorized licensed use limited to: TIANJIN UNIVERSITY. Downloaded on November 11,2022 at 12:32:55 UTC from IEEE Xplore. Restrictions apply.

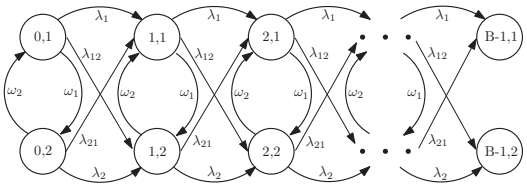


Fig. 8: Buffer state space of a MAP(2) arrival process.

we proceed as follows. First, we derive the average number of requests generated during each phase, $\text{ev}_m = \lambda_m/\omega_m$, as the product of the request arrival rate, λ_m, and the average duration of each phase, $1/\omega_m$. Then, we compute the average number of times that the buffer is in state $(0,m)$ when the first request of a batch arrives. Knowing that ev_m is the number of requests generated during phase m, we divide it by the expected batch size:

$$\alpha_m = \frac{\text{ev}_m}{\min\left(B, \lambda_m \cdot T + 1\right)}. \tag{13}$$

The initial state, $\boldsymbol{\pi}(0)$, to be used in Eq. (5) in the MMPP(2) case, is computed as:

$$\boldsymbol{\pi}(0) = \left(\frac{\alpha_1}{\alpha_1+\alpha_2}, \frac{\alpha_2}{\alpha_1+\alpha_2}\right). \tag{14}$$

With a MMPP(2) arrival process, also the required time, τ, to collect $B-1$ requests besides the first one (i.e., the time to reach the maximum batch size) is different from the Poisson case since it depends on the arrival process. Hence, Eq. (10) in case ③ of the proposed model, i.e., Eq. (9), needs to be replaced by:

$$\tau = (B-1) \cdot \frac{\omega_1+\omega_2}{\lambda_1 \cdot \omega_2 + \lambda_2 \cdot \omega_1}. \tag{15}$$

MAP(2) arrival process. MAPs are flexible non-renewal stochastic processes that can model general distributions and are commonly used to describe correlated and bursty events [50, 56–59]. MAPs are a generalization of MMPPs [52] that allows changing a phase also during observed events. A two-phase MAP is defined by the matrix in Eq. (11), where:

$$\begin{aligned} \boldsymbol{D0} &= \begin{bmatrix} -(\lambda_1+\lambda_{12}+\omega_1) & \omega_1 \\ \omega_2 & -(\lambda_2+\lambda_{21}+\omega_2) \end{bmatrix} \\ \text{and} \quad \boldsymbol{D1} &= \begin{bmatrix} \lambda_1 & \lambda_{12} \\ \lambda_{21} & \lambda_2 \end{bmatrix}, \end{aligned} \tag{16}$$

and the state space of the buffer is represented by the CTMC in Fig. 8. A MAP(2) is defined by four parameters (i.e., mean, square coefficient of variation, skewness, and lag-1 autocorrelation) and can easily fit bursty traces [60]. The request latency distribution with a MAP(2) arrival process is computed with the exact same steps as for the MMPP(2) case.

V. Prototype Implementation

We prototype BATCH atop AWS Lambda and discuss the implementation choices for its key components in this section.

Serving package development and deployment. We develop the serving package according to the guidelines provided by AWS [61]. The serving package is implemented using two widely used ML frameworks, Tensorflow [26] and MXNet Model Server [62]. A function is created using the *create_function* method from the boto3 library [63] for deploying the package. We minimize the invocation delay by 2× through persisting the model graphs in memory.

Serverless Inference. Once the function is created and the package is deployed, the framework is ready for serving. The serving function takes incoming requests (e.g., images) as inputs in the form of a list, each item on the list represents a request. As soon as the function is invoked, the list of requests are transformed into a batch by the serving function and the batched requests are processed through the ML model for inference. For each request, typically the top inferred value is returned from the model and sent back to the end users.

Profiler. The most time consuming task of the Profiler is measuring the service time of the ML application under different system configurations (i.e., memory size, batch size, and timeout). To reduce the profiling time, the Profiler measures the workload service time under only *a few* different system configurations and estimates service times of the remaining ones through regression. We prototype the Profiler atop of AWS CloudWatch [64]. Since this is an offline phase and the profiler requires little computational power, it can be collocated with the Buffer and Performance optimizer.

Buffer. The buffer module uses a proxy server to collect incoming requests to form a batch. The proxy server requires little computational power and can be deployed on a cheap burstable instance (e.g., *t2.nano* [65], less than 0.14 $/day) along with the Profiler. Alternatively, the buffer can be implemented using the streaming service offered by AWS (e.g., *Amazon Kinesis* [66]) but at the premium charged by AWS.

Performance optimizer. The performance optimizer implements the model described in Section IV to determine the best system configuration based on the actual workload intensity. The analytical model can be collocated with the Profiler and Buffer. It quickly builds the state space of all considered configuration options and selects the optimal system configuration in less than 10 seconds. Figure 9 shows the CPU and memory usage of an AWS *t2.nano* instance ($0.14/day) hosting BATCH components. The maximum memory utilization is less than 200 MB (over 512 MB available for a *t2.nano* instance). Since the optimizer runs every hour and takes less than 10 seconds to find an optimal solution, CPU utilization is negligible. The computation time of BATCH depends only on the number of explored configurations and is not affected by the intensity of the arrival process since the state space is finite.

VI. Results

Here, we evaluate the accuracy of the multivariable polynomial model for the estimation of the batch service time (see Section III-A). BATCH is validated experimentally on AWS Lambda and compared to other available strategies.

A. Experimental Setup

We evaluate BATCH with two **ML Serving Frameworks:**

Authorized licensed use limited to: TIANJIN UNIVERSITY. Downloaded on November 11,2022 at 12:32:55 UTC from IEEE Xplore. Restrictions apply.

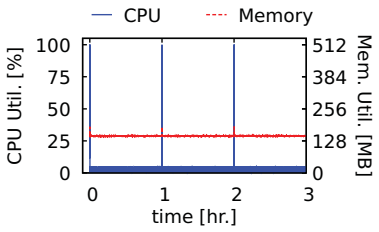
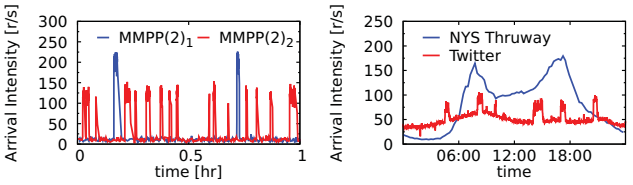


Fig. 9: CPU and memory usage over three hours of an AWS *t2.nano* instance hosting BATCH components. The daily cost of a *t2.nano* instance is less than $0.14.



(a) Arrival rate of MMPPs (b) Arrival rate of real traces

Fig. 10: Intensity of arrival processes used to evaluate BATCH.

- TensorFlow [26] is a widely adopted ML framework which supports a wide range of ML applications such as image classification, speech recognition, and more. We are unable to use the TensorFlow serving due to package size limitations (250 MB) imposed by AWS Lambda. We implement our own serving system based on TensorFlow.
- MXNet Model Server [62] is a popular ML framework that supports a wide range of ML applications. MXNet does not inherently support dynamic batch size for inference. We extend MXNet for dynamic batching.

ML Applications. We use popular computer vision ML models with serving requests extracted from the ImageNet-22K dataset [67] (image resolution: $224 \times 224 \times 3$):

- MoBiNet [68] is a Mobile Binary Network for lightweight image classification.
- ResNet-18 and ResNet-50 [69] are medium size models with residual functions (18 and 50 layers, repsectively).
- Inception-v4 [70] is a large deep convolutional neural network (48 layers) with tens of millions of parameters.
- ResNet-v2 [70] is a deep convolutional neural networks (162 layers) with hundreds of millions of parameters.

Workload. BATCH is evaluated with MMPP arrivals (i.e., two synthetic arrival processes) and real arrival traces from NYS Thruway [29] and Twitter [30]. Figs. 10(a) and 10(b) depict the arrival intensities of MMPPs and real traces, respectively. To highlight the capability of BATCH to handle heavy traffic, the arrival rate observed in [29] (i.e., between 0 and 12 reqs/sec) is increased by 15 times. This is motivated by the arrival intensity observed from Microsoft production traces [19], where the arrival rate to a single front-end server may vary from 0 to 50 reqs/sec (Figure 6 in [19]). Since the ML platform proposed in [19] is made of multiple front-end servers, it is reasonable to assume that the arrival intensity of real MLaaS clusters is similar to (or larger than) the one used in this paper. Twitter traces are not scaled and are used without any modification.

Static Choices. The performance of BATCH is compared with Vanilla Lambda, the state-of-the-practice tool SageMaker [31], and the state-of-the-art approach MArk [16]. Vanilla Lambda does not implement batching and only the memory size may be tuned. We consider two strategies to select the memory size: *1)* maximum value (i.e., 3 GB, if application requirements are not known) or *2)* cherry pick the value that allows minimizing the cost (this strategy requires to profile the application). The experiments on SageMaker are conducted on *c5.4xlarge* instances with auto-scaling enabled following the AWS guidelines [71]. To accommodate the expected traffic without resource over provisioning auto-scaling must be configured manually. We deploy MArk on CPU instances (*c5.4xlarge*) that, as suggested in [16], are more cost-effective than GPU instances.

B. Service Time Model: Is it Truly Deterministic?

We first confirm the deterministic service time assumption. Fig. 11 shows that the batch service time of image recognition applications can be approximated with a deterministic process since the empirical coefficient of variation (CV) is always smaller than 0.1 (the deterministic distribution has CV=0).

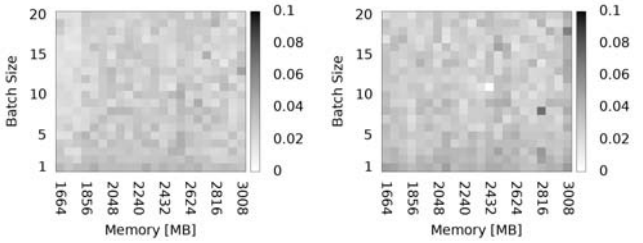
To determine the mean of the service time distribution with *as few experiments as possible* for the various memory/batch size configurations, BATCH uses a multivariable regression model that is trained with a few configurations (in our experiments, less than 3%). Results show that BATCH can reduce the profiling time by more than 97% compared to exhaustively profiling all system configurations. The accuracy of the regression model is validated against configurations that are not used for training. The mean absolute percentage error is always less than 2%.

C. Validation of the Analytical Model

In the heart of BATCH lies the analytical model. Because the target of BATCH is to predict SLOs, essentially *tail latencies*, it is important to evaluate its accuracy regarding *how well it can predict the probability distribution of latencies.*

Arrivals Driven by MMPP(2). We evaluate model accuracy and robustness on *AWS Lambda* using different applications (i.e., MoBiNet, ResNet-v2, and ResNet-18), workload arrival patterns, maximum batch sizes (i.e., 15 and 20), timeouts (i.e., 10, 100, and 1000 ms), and memory sizes (i.e., 1536 and 3008 MB). A large maximum batch size and different timeouts allow testing the model accuracy with different batch buffer sizes. The actual batch size is also controlled by the timeout value. The arrival process for the above experiments is driven by traces generated from the two MMPP(2) processes described in Section VI-A, results are shown in Fig. 12. Model predictions with different system/workload configurations are remarkably close to the AWS measurements with an error consistently less than 9%. The analytical model remains accurate regardless of the application, the arrival process, and

Authorized licensed use limited to: TIANJIN UNIVERSITY. Downloaded on November 11,2022 at 12:32:55 UTC from IEEE Xplore. Restrictions apply.



(a) Inception-v4 (b) ResNet-v2

Fig. 11: Coefficient of variation (CV) of service time for various memory/batch size configurations.

the memory size. In the interest of space, similar results for different system configurations are omitted.

Arrivals Driven by Real Traces. The accuracy of the prediction model is also evaluated using real traces from the NYS Thruway and Twitter. *Note that in this experiment we assume that we know the arrival process a priori.* Since the analytic model needs to admit an input for the arrival process in a MAP(2) form, we first fit the trace data using the KPC-Toolbox [42,72] that is publicly available. The MAP returned by the KPC-Toolbox is passed to the prediction model to forecast the request latency CDF. Fig. 13 depicts predicted and observed latency CDFs for NYS Thruway and Twitter traces. The effectiveness of the analytic model is again remarkably accurate: independently of the considered system configuration, the error is consistently smaller than 8%.

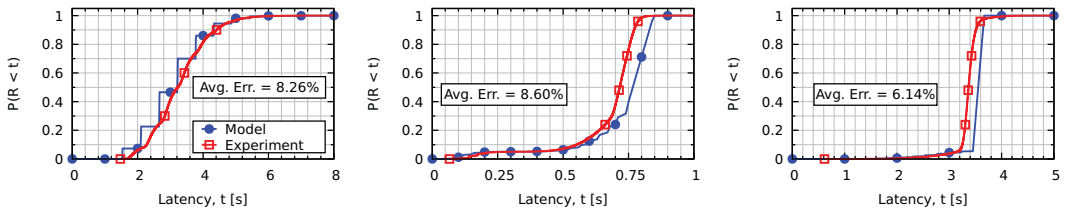
To the best of our knowledge this is the first analytical model that can capture accurately the shape of the latency distribution in the presence of bursty arrivals and deterministic service times.

D. Optimizing Cost or Latency

For these experiments we use the May 25, 2017 trace from Twitter [30] and the October 11, 2018 trace from NYS Thruway [29] and focus on the two optimization problems given in Eqs. (2) and (3). *Here, the arrival process is not known a priori.* To determine the arrival process to pass to BATCH for prediction, we monitor incoming requests with a 1-hour long (sliding) window and find the best fitting MAP using the KPC-toolbox [42]. When BATCH is deployed on a *t2.nano* instance, the minimum size of the sliding window is smaller than 10 seconds (i.e., the time to derive the optimal configuration, see Section V). Deploying BATCH on a more powerful instance decreases the computation time and the minimum size of the sliding window.

Cost Minimization and Latency SLOs: BATCH is tested with ResNet-v2 and ResNet-50 to minimize cost, while complying with the SLO defined on the 95th percentile latency, i.e., using Eq. (2). Results are shown in Fig. 14 for NYS Thruway (columns 1 and 2) and Twitter (columns 3 and 4). The first row of Fig. 14 shows the 95th percentile latency of different approaches normalized over the SLO (black line). BATCH is the only approach that always keeps the latency close to the SLO target without violating it, independently of the load and ML application. The second row depicts the complementary CDF (CCDF) of latencies (normalized over the SLO and note the log scale of the y-axis). Here, we opted for showing the CCDF to better illustrate the performance of tail latencies. The vertical solid line is the SLO and the horizontal dashed line is the 95th percentile. The third row shows the monetary cost of each approach. Values are normalized over the cost of Vanilla Lambda (with maximum memory). BATCH always minimizes cost. For ResNet-50 (columns 2 and 4), BATCH and Vanilla Lambda (cherry pick) achieve similar cost. This is due to a strict SLO that reduces the batching capability of BATCH since the request latency increases with the batch size. When BATCH can fully exploit batching (columns 1 and 3), it provides the smallest cost. SageMaker achieves the lowest 95th percentile latency and low cost when serving the Twitter load with ResNet-v2 (column 3). Differently from BATCH that automatically selects the optimal system configuration, SageMaker requires a long time to be manually configured. Note that we fine-tuned SageMaker here to optimize its performance.

Fig. 15 depicts the parameters selected by BATCH for the NYS Thruway trace with ResNet-v2 (column 1 in Fig. 14). It shows how BATCH dynamically changes the effective batch size, timeout, and memory to minimize the cost while complying with the given SLO. Cost minimization is enabled by the correct setting of the maximum batch size and timeout (the effective batch size varies from 6 to 13) and dynamic



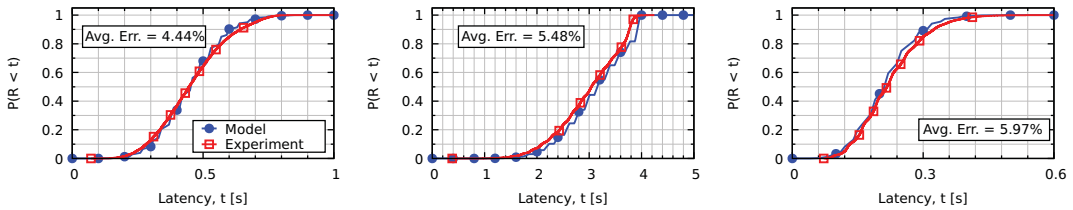
(a) ResNet-v2, MMPP(2)$_1$ Memory=3008 MB, Timeout=10ms Max Batch Size=20

(b) MoBiNet, MMPP(2)$_1$ Memory=3008 MB, Timeout=100ms Max Batch Size=20

(c) ResNet-18, MMPP(2)$_2$ Memory=1536 MB, Timeout=1000ms Max Batch Size=15

Fig. 12: Request latency distribution driven with arrivals generated by MMPP(2) processes.

Authorized licensed use limited to: TIANJIN UNIVERSITY. Downloaded on November 11,2022 at 12:32:55 UTC from IEEE Xplore. Restrictions apply.

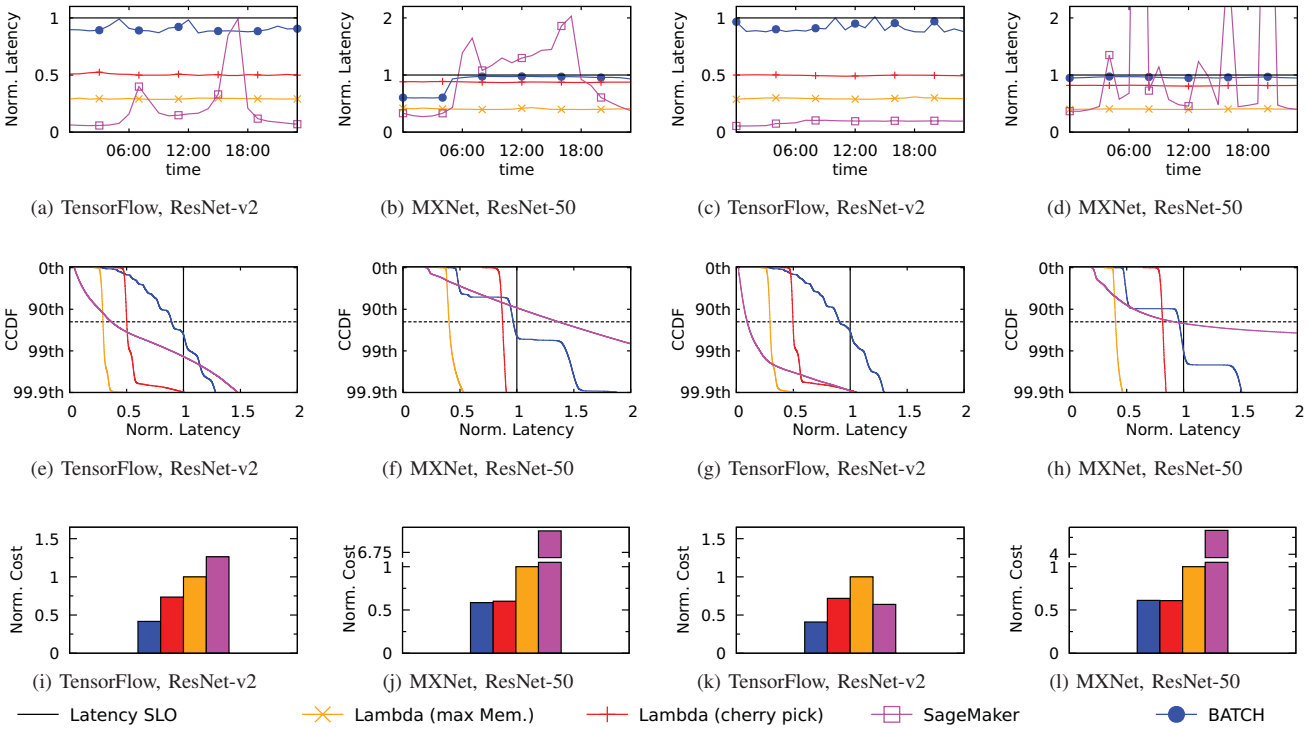


(a) MoBiNet, NYS Thruway, 6-7am
Memory=3008 MB, Timeout=100ms
Max Batch Size=20

(b) ResNet-18, NYS Thruway, 7-8am
Memory=2048 MB, Timeout=100ms
Max Batch Size=20

(c) MoBiNet, Twitter, 8-9pm
Memory=2496 MB, Timeout=50ms
Max Batch Size=20

Fig. 13: Request latency distribution with arrivals driven from real workload traces.



(a) TensorFlow, ResNet-v2
(b) MXNet, ResNet-50
(c) TensorFlow, ResNet-v2
(d) MXNet, ResNet-50
(e) TensorFlow, ResNet-v2
(f) MXNet, ResNet-50
(g) TensorFlow, ResNet-v2
(h) MXNet, ResNet-50
(i) TensorFlow, ResNet-v2
(j) MXNet, ResNet-50
(k) TensorFlow, ResNet-v2
(l) MXNet, ResNet-50

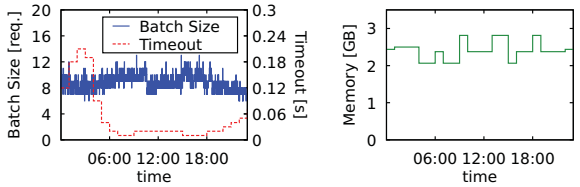
Fig. 14: Cost minimization for NYS Thruway (columns 1 and 2) and Twitter (columns 3 and 4) traces with latency SLO. The first row shows the 95th percentile latency over 24 hours, the second row shows the latency CCDF, and the third row shows the cost. Latency and cost are normalized (simple ratio) over the SLO and the cost of *Vanilla Lambda (max Mem.)*, respectively.

memory allocation.

We compare BATCH to MArk [16] using the MMPP$(2)_1$ arrival process. In this case, the SLO is defined on the 99th percentile of latency. Figs. 16(a) and 16(b) show the 99th percentile of latency and the latency CCDFs, respetively. All results are normalized over the SLO. Although the monetary cost of using either BATCH or MArk is similar (Fig. 16(c), costs are normalized over the cost of MArk), BATCH always meets the latency objective, even when sudden workload surges are observed, thanks to the autoscaling property of serverless. MArk routes requests to serverless only when it detects workload variations; if the surge detection takes too long, resource scaling is not optimal and latency repeatedly violates SLO due to the time overhead of slow detection.

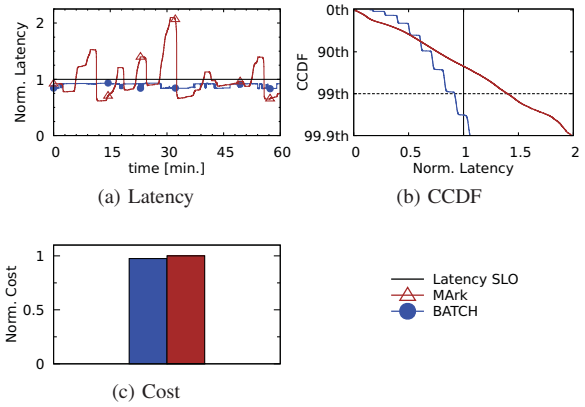
Latency on a Budget: BATCH can be used to control the application latency while complying with the available budget (cost), i.e., using Eq. (3). Results are shown in Fig. 17 for ResNet-v2 and Inception-v4, when the arrival process is driven by NYS Thruway traces (columns 1 and 2) or Twitter ones (columns 3 and 4). The first row depicts the 95th percentile latency of each approach normalized over the one of Vanilla Lambda (with maximum memory), while the second row shows the cost of each approach normalized over the budget constraint (horizontal line). BATCH always

Authorized licensed use limited to: TIANJIN UNIVERSITY. Downloaded on November 11,2022 at 12:32:55 UTC from IEEE Xplore. Restrictions apply.



(a) Batch size and Timeout (b) Memory

Fig. 15: Batch size, timeout, and memory used by BATCH for serving ResNet-v2 (NYS Thruway with SLO on latency).



(a) Latency (b) CCDF

(c) Cost

Fig. 16: Comparison between BATCH and MArk.

complies with the available budget. This comes with the trade-off of a generally longer request latency comparing to other configurations. SageMaker's latency may be much longer than the one of BATCH, see Fig. 17(b), due to the time required to turn on new EC2 instances to serve incoming requests. These results illustrate the ability of BATCH to keep latency low without budget overspending.

We emphasize that BATCH seamlessly adjusts the environment parameterization while effectively coping with workload burstiness, service characteristics, and optimization targets.

E. Discussion

GPU accelerators. Past work shows that inference latency on serverless platforms may be longer than expected when the ML model is deployed on IaaS [12, 16, 18].Differently from IaaS, serverless computing does not allow processing requests using GPUs. Since serverless computing is new [73] and since by 2021 its market size is estimated to grow by 310% comparing to 2016 [74], it is reasonable to assume that cloud providers will invest more resources to this technology. BATCH is easily adaptable because it only needs to re-train the regression model used by the profiler to estimate the service time of the application, the analytic model of Section IV can be adopted without any modification.

Cold start in serverless. Latency overhead when starting serverless functions is generally observed and it is called *cold start* [75, 76]. In AWS Lambda, cold start is observable especially when the time between two request arrivals (i.e., the function inactivity) is longer than 40 minutes [77]. Since the real traces considered here do not have such long inter-arrival times, cold start does not affect performance and is therefore not investigated. Solutions exist to reduce the overhead of cold start [78–80]. AWS has recently introduced *provisioned concurrency* [75] to keep functions ready to serve requests.

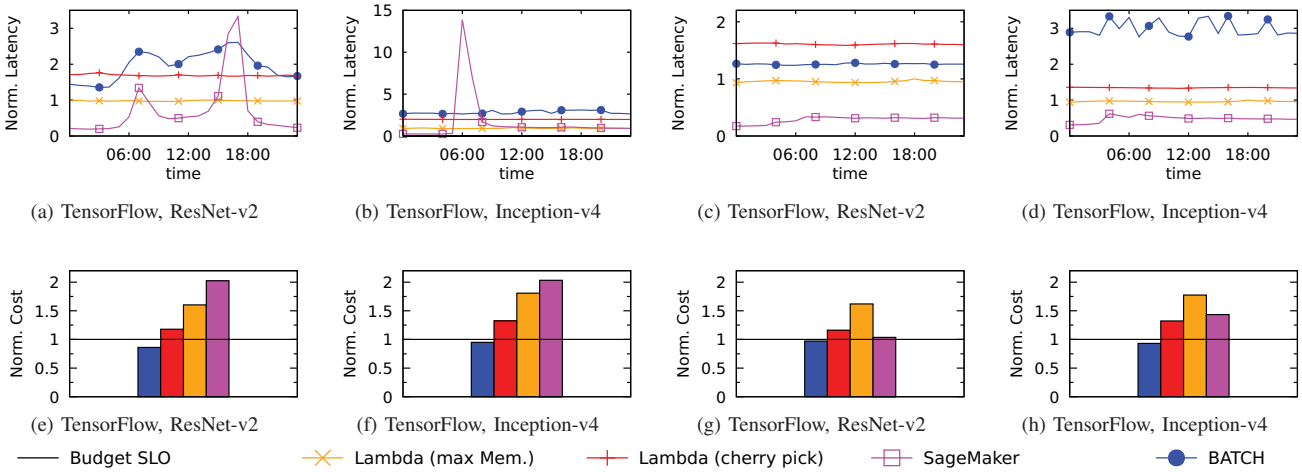
VII. Related Work

Ishakian et al. [12] investigate the suitability of serverless computing for deep learning models. Systems for ML training (but not for ML inference) on serverless platforms include [11, 13] Zhang et al. [16] propose MArk, a serving system for ML inference that combines IaaS and serverless to decrease cost while meeting SLOs. Since serverless computing is used only when workload variations are detected, unexpected workload surges result in longer latency tails. BARISTA [18] is a framework for horizontal and vertical scaling for ML services with bursty workloads but is not evaluated on a public serverless platform. Gujarati et al. [19] propose Swayam, an autoscaling framework deployed atop Microsoft Azure, that proactively allocates resources to increase resource efficiency of ML inference services, while complying with SLOs. Provisioning cost is not relevant for Swayam since it deploys models in private MLaaS clusters.

Dynamic batching for improving ML inference performance is used by Clipper [21]. Clipper is not suitable for the serverless paradigm because it uses an exhaustive profiling strategy (i.e., additive-increase-multiplicative-decrease scheme) to find a good batch size. While this may be suitable for an IaaS environment under a relatively stable arrival workload, it requires too much profiling for serverless if the workload is highly dynamic (the additional dimension of added memory size adds to the profiling complexity). Moreover, Clipper is SLO-oblivious due to its reactive design while BATCH acts proactively to offer SLO guarantees.

Dynamic batching is also adopted by Stout [81] and GrandSLAm [82]. The former implements dynamic batching for increasing the throughput of cloud storage application. Stout only considers average performance values (not percentiles) and the batch parameter cannot be tuned to comply with user-defined SLOs. GrandSLAm uses dynamic batching for processing microservice requests and increases the system throughput while complying with SLOs. Differently from BATCH, GrandSLAm accounts only for synthetic Poisson workloads, and not generic or bursty ones. The authors of [83] use a multi-formalism model to perform a capacity planning analysis of a system where dynamic batching is used for transmitting sampled data through a radio channel. In this case, no analytical solution is provided. AWS Batch [84] automatically provides the system with the optimal number of resources based on its workload, but it does not support serverless computing yet. BATCH is the first framework to implement ML batching on serverless platforms on the public cloud.

Authorized licensed use limited to: TIANJIN UNIVERSITY. Downloaded on November 11,2022 at 12:32:55 UTC from IEEE Xplore. Restrictions apply.



(a) TensorFlow, ResNet-v2 (b) TensorFlow, Inception-v4 (c) TensorFlow, ResNet-v2 (d) TensorFlow, Inception-v4

(e) TensorFlow, ResNet-v2 (f) TensorFlow, Inception-v4 (g) TensorFlow, ResNet-v2 (h) TensorFlow, Inception-v4

Fig. 17: Latency minimization for NYS Thruway (columns 1 and 2) and Twitter (columns 3 and 4) traces with budget SLO. The first row shows the 95th percentile latency over 24 hours and the second row shows the cost. Latency and cost are normalized (simple ratio) over the latency of *Vanilla Lambda (max Mem.)* and the SLO, respectively.

VIII. Concluding Remarks

We introduce BATCH, a novel framework for optimizing ML serving on serverless platforms. BATCH uses a lightweight profiling strategy and an analytical model to identify the best parameter configuration (i.e., memory size, batch size, and timeout) to improve the system performance while meeting user-defined SLOs. The efficiency of BATCH is evaluated using real traces and comparing its performance with other available strategies (e.g., AWS SageMaker). We show that BATCH decreases the cost of maintaining the system by 50% and can minimize the system performance while meeting the budget independently of the arrival intensity. Future working includes extending BATCH to support different service time distributions and adopting optimization algorithms that are faster than the exhaustive search used here to support co-optimization of latency and cost.

Acknowledgement

This work is supported in part by the following grants: National Science Foundation IIS-1838024 (using resources provided by Amazon Web Services as part of the NSF BIGDATA program), IIS-1838022, CCF-1756013, CCF-1717532, and CNS-1950485. We thank the anonymous reviewers for their insightful comments and suggestions that significantly improved the paper.

References

[1] "Aws lambda – serverless compute," https://aws.amazon.com/lambda/, [Online; accessed 04-December-2019].

[2] "Ibm cloud – cloud functions," https://www.ibm.com/cloud/functions/, [Online; accessed 04-December-2019].

[3] "Azure functionsserverless architecture – microsoft azure," https://azure.microsoft.com/en-us/services/functions/, [Online; accessed 04-December-2019].

[4] "Cloud functions – serverless environment to build and connect cloud services — google cloud platform," https://cloud.google.com/functions/, [Online; accessed 04-December-2019].

[5] L. F. Herrera-Quintero, J. C. Vega-Alfonso, K. B. A. Banse, and E. C. Zambrano, "Smart its sensor for the transportation planning based on iot approaches using serverless and microservices architecture," *IEEE Intelligent Transportation Systems Magazine*, vol. 10, no. 2, pp. 17–27, 2018.

[6] P. Persson and O. Angelsmark, "Kappa: serverless iot deployment," in *Proceedings of the 2nd International Workshop on Serverless Computing*. ACM, 2017, pp. 16–21.

[7] G. McGrath and P. R. Brenner, "Serverless computing: Design, implementation, and performance," in *2017 IEEE 37th International Conference on Distributed Computing Systems Workshops (ICDCSW)*. IEEE, 2017, pp. 405–410.

[8] B. Cheng, J. Fuerst, G. Solmaz, and T. Sanada, "Fog function: Serverless fog computing for data intensive iot services," in *2019 IEEE International Conference on Services Computing (SCC)*. IEEE, 2019, pp. 28–35.

[9] J. Rajewski, "System and method for live streaming content to subscription audiences using a serverless computing system," Jun. 7 2018, uS Patent App. 15/369,473.

[10] L. Ao, L. Izhikevich, G. M. Voelker, and G. Porter, "Sprocket: A serverless video processing framework," in *Proceedings of the ACM Symposium on Cloud Computing*. ACM, 2018, pp. 263–274.

[11] J. Carreira, P. Fonseca, A. Tumanov, A. Zhang, and R. Katz, "A case for serverless machine learning," in *Workshop on Systems for ML and Open Source Software at NeurIPS*, vol. 2018, 2018.

[12] V. Ishakian, V. Muthusamy, and A. Slominski, "Serving deep learning models in a serverless platform," in *2018 IEEE International Conference on Cloud Engineering (IC2E)*. IEEE, 2018, pp. 257–262.

[13] H. Wang, D. Niu, and B. Li, "Distributed machine learning with a serverless architecture," in *IEEE INFOCOM 2019-IEEE Conference on Computer Communications*. IEEE, 2019, pp. 1288–1296.

[14] T. Elsken, J. H. Metzen, and F. Hutter, "Neural architecture search: A survey." *Journal of Machine Learning Research*, vol. 20, no. 55, pp. 1–21, 2019.

[15] ——, "Neural architecture search," in *Automated Machine Learning*. Springer, 2019, pp. 63–77.

[16] C. Zhang, M. Yu, W. Wang, and F. Yan, "Mark: Exploiting cloud services for cost-effective, slo-aware machine learning inference serving," in *2019 USENIX Annual Technical Conference (USENIX ATC 19)*, 2019.

[17] J. M. Hellerstein, J. M. Faleiro, J. Gonzalez, J. Schleier-Smith, V. Sreekanti, A. Tumanov, and C. Wu, "Serverless computing: One step forward, two steps back," in *CIDR 2019, 9th Biennial Conference on Innovative Data Systems Research, Asilomar, CA, USA, January 13-16, 2019, Online Proceedings*. www.cidrdb.org, 2019. [Online]. Available: http://cidrdb.org/cidr2019/papers/p119-hellerstein-cidr19.pdf

Authorized licensed use limited to: TIANJIN UNIVERSITY. Downloaded on November 11,2022 at 12:32:55 UTC from IEEE Xplore. Restrictions apply.

[18] A. Bhattacharjee, A. D. Chhokra, Z. Kang, H. Sun, A. Gokhale, and G. Karsai, "Barista: Efficient and scalable serverless serving system for deep learning prediction services," in *2019 IEEE International Conference on Cloud Engineering (IC2E)*. IEEE, 2019, pp. 23–33.

[19] A. Gujarati, S. Elnikety, Y. He, K. S. McKinley, and B. B. Brandenburg, "Swayam: distributed autoscaling to meet slas of machine learning inference services with resource efficiency," in *Proceedings of the 18th ACM/IFIP/USENIX Middleware Conference*. ACM, 2017, pp. 109–120.

[20] X. Tang, P. Wang, Q. Liu, W. Wang, and J. Han, "Nanily: A qos-aware scheduling for dnn inference workload in clouds," in *2019 IEEE 21st International Conference on High Performance Computing and Communications; IEEE 17th International Conference on Smart City; IEEE 5th International Conference on Data Science and Systems (HPCC/SmartCity/DSS)*. IEEE, 2019, pp. 2395–2402.

[21] D. Crankshaw, X. Wang, G. Zhou, M. J. Franklin, J. E. Gonzalez, and I. Stoica, "Clipper: A low-latency online prediction serving system," in *14th USENIX Symposium on Networked Systems Design and Implementation (NSDI 17)*, 2017, pp. 613–627.

[22] F. Yan, S. Hughes, A. Riska, and E. Smirni, "Overcoming limitations of off-the-shelf priority schedulers in dynamic environments," in *2013 IEEE 21st International Symposium on Modelling, Analysis and Simulation of Computer and Telecommunication Systems, San Francisco, CA, USA, August 14-16, 2013*, 2013, pp. 505–514. [Online]. Available: https://doi.org/10.1109/MASCOTS.2013.72

[23] N. Mi, A. Riska, E. Smirni, and E. Riedel, "Enhancing data availability in disk drives through background activities," in *The 38th Annual IEEE/IFIP International Conference on Dependable Systems and Networks, DSN 2008, June 24-27, 2008, Anchorage, Alaska, USA, Proceedings*, 2008, pp. 492–501. [Online]. Available: https://doi.org/10.1109/DSN.2008.4630120

[24] F. Yan, S. Hughes, A. Riska, and E. Smirni, "Agile middleware for scheduling: meeting competing performance requirements of diverse tasks," in *ACM/SPEC International Conference on Performance Engineering, ICPE'14, Dublin, Ireland, March 22-26, 2014*, 2014, pp. 185–196. [Online]. Available: https://doi.org/10.1145/2568088.2568104

[25] N. Mi, A. Riska, Q. Zhang, E. Smirni, and E. Riedel, "Efficient management of idleness in storage systems," *ACM Trans. Storage*, vol. 5, no. 2, pp. 4:1–4:25, 2009. [Online]. Available: https://doi.org/10.1145/1534912.1534913

[26] M. Abadi, P. Barham, J. Chen, Z. Chen, A. Davis, J. Dean, M. Devin, S. Ghemawat, G. Irving, M. Isard *et al.*, "Tensorflow: A system for large-scale machine learning," in *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16)*, 2016, pp. 265–283.

[27] G. Neubig, Y. Goldberg, and C. Dyer, "On-the-fly operation batching in dynamic computation graphs," in *Advances in Neural Information Processing Systems*, 2017, pp. 3971–3981.

[28] A. Ali, R. Pinciroli, F. Yan, and E. Smirni, "Batch pre-release," Jun. 2020. [Online]. Available: https://doi.org/10.5281/zenodo.3872213

[29] "Nys thruway origin and destination points for all vehicles," https://data.ny.gov/Transportation/NYS-Thruway-Origin-and-Destination-Points-for-All-/em4e-ui5w, [Online; accessed 14-November-2019].

[30] ArchiveTeam, "Twitter streaming traces, 2017."

[31] "Amazon. build, train, and deploy machine learning models at scale." https://docs.aws.amazon.com/lambda/latest/dg/resource-model.html, [Online; accessed 03-December-2019].

[32] M. Arlitt and T. Jin, "A workload characterization study of the 1998 world cup web site," *IEEE network*, vol. 14, no. 3, pp. 30–37, 2000.

[33] A. Riska and E. Riedel, "Disk drive level workload characterization," in *Proceedings of the 2006 USENIX Annual Technical Conference, Boston, MA, USA, May 30 - June 3, 2006*, 2006, pp. 97–102. [Online]. Available: http://www.usenix.org/events/usenix06/tech/riska.html

[34] N. Mi, G. Casale, L. Cherkasova, and E. Smirni, "Injecting realistic burstiness to a traditional client-server benchmark," in *Proceedings of the 6th international conference on Autonomic computing*. ACM, 2009, pp. 149–158.

[35] J. Xue, R. Birke, L. Y. Chen, and E. Smirni, "Managing data center tickets: Prediction and active sizing," in *2016 46th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*. IEEE, 2016, pp. 335–346.

[36] N. Mi, Q. Zhang, A. Riska, E. Smirni, and E. Riedel, "Performance impacts of autocorrelated flows in multi-tiered systems," *Perform. Evaluation*, vol. 64, no. 9-12, pp. 1082–1101, 2007. [Online]. Available: https://doi.org/10.1016/j.peva.2007.06.016

[37] N. Mi, G. Casale, L. Cherkasova, and E. Smirni, "Burstiness in multi-tier applications: Symptoms, causes, and new models," in *Middleware 2008, ACM/IFIP/USENIX 9th International Middleware Conference, Leuven, Belgium, December 1-5, 2008, Proceedings*, ser. Lecture Notes in Computer Science, V. Issarny and R. E. Schantz, Eds., vol. 5346. Springer, 2008, pp. 265–286. [Online]. Available: https://doi.org/10.1007/978-3-540-89856-6_14

[38] "Aws autoscaling," https://aws.amazon.com/autoscaling, [Online; accessed 03-December-2019].

[39] "Aws lambda – pricing," https://aws.amazon.com/lambda/pricing/, [Online; accessed 30-October-2019].

[40] S. Xu, H. Zhang, G. Neubig, W. Dai, J. K. Kim, Z. Deng, Q. Ho, G. Yang, and E. P. Xing, "Cavs: An efficient runtime system for dynamic neural networks," in *2018 USENIX Annual Technical Conference (USENIX ATC 18)*, 2018, pp. 937–950.

[41] "Make data useful," http://www.scribd.com/doc/4970486/Make-Data-Useful-by-Greg-Linden-Amazoncom, [Online; accessed 15-January-2020].

[42] G. Casale, E. Z. Zhang, and E. Smirni, "Kpc-toolbox: Best recipes for automatic trace fitting using markovian arrival processes," *Performance Evaluation*, vol. 67, no. 9, pp. 873–896, 2010.

[43] M. F. Neuts, "A versatile markovian point process," *Journal of Applied Probability*, vol. 16, no. 4, pp. 764–779, 1979.

[44] P. Royston and W. Sauerbrei, *Multivariable model-building: a pragmatic approach to regression anaylsis based on fractional polynomials for modelling continuous variables*. John Wiley & Sons, 2008, vol. 777.

[45] F. Yan, O. Ruwase, Y. He, and E. Smirni, "Serf: efficient scheduling for fast deep neural network serving via judicious parallelism," in *SC'16: Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*. IEEE, 2016, pp. 300–311.

[46] T. Zhu, A. Tumanov, M. A. Kozuch, M. Harchol-Balter, and G. R. Ganger, "Prioritymeister: Tail latency qos for shared networked storage," in *Proceedings of the ACM Symposium on Cloud Computing*. ACM, 2014, pp. 1–14.

[47] N. Li, H. Jiang, D. Feng, and Z. Shi, "Pslo: enforcing the x th percentile latency and throughput slos for consolidated vm storage," in *Proceedings of the Eleventh European Conference on Computer Systems*. ACM, 2016, p. 28.

[48] L. Kleinrock, *Queueing systems. Volume I: theory*. wiley New York, 1975.

[49] F. Yan, Y. He, O. Ruwase, and E. Smirni, "Efficient deep neural network serving: Fast and furious," *IEEE Trans. Network and Service Management*, vol. 15, no. 1, pp. 112–126, 2018. [Online]. Available: https://doi.org/10.1109/TNSM.2018.2808352

[50] G. Casale, N. Mi, and E. Smirni, "Model-driven system capacity planning under workload burstiness," *IEEE Transactions on Computers*, vol. 59, no. 1, pp. 66–80, 2009.

[51] A. Reibman and K. Trivedi, "Numerical transient analysis of markov models," *Computers & Operations Research*, vol. 15, no. 1, pp. 19–36, 1988.

[52] W. Fischer and K. Meier-Hellstern, "The markov-modulated poisson process (mmpp) cookbook," *Performance evaluation*, vol. 18, no. 2, pp. 149–171, 1993.

[53] J. Bernardo, M. Bayarri, J. Berger, A. Dawid, D. Heckerman, A. Smith, and M. West, "The markov modulated poisson process and markov poisson cascade with applications to web traffic modeling," *Bayesian Statistics*, 2003.

[54] N. Bean, D. Green, and P. Taylor, "The output process of an mmpp/m/1 queue," *Journal of Applied Probability*, vol. 35, no. 4, pp. 998–1002, 1998.

[55] X. Lu, J. Yin, H. Chen, and X. Zhao, "An approach for bursty and self-similar workload generation," in *International Conference on Web Information Systems Engineering*. Springer, 2013, pp. 347–360.

[56] F. Bause, P. Buchholz, and J. Kriege, "A comparison of markovian arrival and arma/arta processes for the modeling of correlated input processes," in *Proceedings of the 2009 Winter Simulation Conference (WSC)*. IEEE, 2009, pp. 634–645.

[57] G. Casale, E. Z. Zhang, and E. Smirni, "Trace data characterization and fitting for markov modeling," *Perform. Evaluation*, vol. 67, no. 2, pp. 61–79, 2010. [Online]. Available: https://doi.org/10.1016/j.peva.2009.09.003

[58] M. F. Neuts, *Structured stochastic matrices of M/G/1 type and their applications*. Marcel Dekker New York, 1989, vol. 5.

Authorized licensed use limited to: TIANJIN UNIVERSITY. Downloaded on November 11,2022 at 12:32:55 UTC from IEEE Xplore. Restrictions apply.

[59] A. Riska and E. Smirni, "Mamsolver: A matrix analytic methods tool," in *Computer Performance Evaluation, Modelling Techniques and Tools 12th International Conference, TOOLS 2002, London, UK, April 14-17, 2002, Proceedings*, ser. Lecture Notes in Computer Science, T. Field, P. G. Harrison, J. T. Bradley, and U. Harder, Eds., vol. 2324. Springer, 2002, pp. 205–211. [Online]. Available: https://doi.org/10.1007/3-540-46029-2_14

[60] G. Casale, N. Mi, L. Cherkasova, and E. Smirni, "How to parameterize models with bursty workloads," *ACM SIGMETRICS Performance Evaluation Review*, vol. 36, no. 2, pp. 38–44, 2008.

[61] "Lambda package documentation," https://aws.amazon.com/premiumsupport/knowledge-center/build-python-lambda-deployment-package/, [Online; accessed 19-June-2019].

[62] T. Chen, M. Li, Y. Li, M. Lin, N. Wang, M. Wang, T. Xiao, B. Xu, C. Zhang, and Z. Zhang, "Mxnet: A flexible and efficient machine learning library for heterogeneous distributed systems," *arXiv preprint arXiv:1512.01274*, 2015.

[63] "Boto 3 documentation," https://boto3.amazonaws.com/v1/documentation/api/latest/index.html, [Online; accessed 19-December-2019].

[64] "Aws cloudwatch," https://aws.amazon.com/cloudwatch/, [Online; accessed 04-April-2020].

[65] "Amazon ec2 t2 instances," https://aws.amazon.com/ec2/instance-types/t2/, [Online; accessed 7-January-2020].

[66] "Aws kinesis," https://aws.amazon.com/kinesis/?nc=sn&loc=0, [Online; accessed 04-April-2020].

[67] O. Russakovsky, J. Deng, H. Su, J. Krause, S. Satheesh, S. Ma, Z. Huang, A. Karpathy, A. Khosla, M. Bernstein *et al.*, "Imagenet large scale visual recognition challenge," *International journal of computer vision*, vol. 115, no. 3, pp. 211–252, 2015.

[68] H. Phan, D. Huynh, Y. He, M. Savvides, and Z. Shen, "Mobinet: A mobile binary network for image classification," *arXiv preprint arXiv:1907.12629*, 2019.

[69] K. He, X. Zhang, S. Ren, and J. Sun, "Deep residual learning for image recognition," in *Proceedings of the IEEE conference on computer vision and pattern recognition*, 2016, pp. 770–778.

[70] C. Szegedy, S. Ioffe, V. Vanhoucke, and A. A. Alemi, "Inception-v4, inception-resnet and the impact of residual connections on learning," in *Thirty-First AAAI Conference on Artificial Intelligence*, 2017.

[71] "Load testing for production variant automatic scaling," https://github.com/awsdocs/amazon-sagemaker-developer-guide/blob/master/doc_source/endpoint-scaling-loadtest.md, [Online; accessed 07-January-2020].

[72] G. Casale, E. Z. Zhang, and E. Smirni, "Kpc-toolbox: Simple yet effective trace fitting using markovian arrival processes," in *Fifth International Conference on the Quantitative Evaluaiton of Systems (QEST 2008), 14-17 September 2008, Saint-Malo, France*. IEEE Computer Society, 2008, pp. 83–92. [Online]. Available: https://doi.org/10.1109/QEST.2008.33

[73] P. Castro, V. Ishakian, V. Muthusamy, and A. Slominski, "The rise of serverless computing," *Commun. ACM*, vol. 62, no. 12, pp. 44–54, 2019. [Online]. Available: http://doi.acm.org/10.1145/3368454

[74] Businesswire, "$7.72 billion function-as-a-service market 2017 - global forecast to 2021: Increasing shift from devops to serverless computing to drive the overall function-as-a-service market - research and markets," https://www.businesswire.com/news/home/20170227006262/en/7.72-Billion-Funct\%20ion-as-a-Service-Market-2017---Global, [Online; accessed 16-December-2019].

[75] "New for aws lambda – predictable start-up times with provisioned concurrency," https://aws.amazon.com/blogs/compute/new-for-aws-lambda-predictable-start-up-times-with-provisioned-concurrency/, [Online; accessed 9-January-2020].

[76] L. Wang, M. Li, Y. Zhang, T. Ristenpart, and M. Swift, "Peeking behind the curtains of serverless platforms," in *2018 USENIX Annual Technical Conference (USENIX ATC 18)*, 2018, pp. 133–146.

[77] W. Lloyd, S. Ramesh, S. Chinthalapati, L. Ly, and S. Pallickara, "Serverless computing: An investigation of factors influencing microservice performance," in *2018 IEEE International Conference on Cloud Engineering (IC2E)*. IEEE, 2018, pp. 159–169.

[78] E. Oakes, L. Yang, D. Zhou, K. Houck, T. Harter, A. Arpaci-Dusseau, and R. Arpaci-Dusseau, "Sock: Rapid task provisioning with serverless-optimized containers," in *2018 USENIX Annual Technical Conference (USENIX ATC 18)*, 2018, pp. 57–70.

[79] A. Mohan, H. Sane, K. Doshi, S. Edupuganti, N. Nayak, and V. Sukhomlinov, "Agile cold starts for scalable serverless," in *11th USENIX Workshop on Hot Topics in Cloud Computing (HotCloud 19)*, 2019.

[80] K. Mahajan, S. Mahajan, V. Misra, and D. Rubenstein, "Exploiting content similarity to address cold start in container deployments," in *Proceedings of the 15th International Conference on emerging Networking EXperiments and Technologies*, 2019, pp. 37–39.

[81] J. C. McCullough, J. Dunagan, A. Wolman, and A. C. Snoeren, "Stout: An adaptive interface to scalable cloud storage," in *Proc. of the USENIX Annual Technical Conference–ATC*, 2010, pp. 47–60.

[82] R. S. Kannan, L. Subramanian, A. Raju, J. Ahn, J. Mars, and L. Tang, "Grandslam: Guaranteeing slas for jobs in microservices execution frameworks," in *Proceedings of the Fourteenth EuroSys Conference 2019*, 2019, pp. 1–16.

[83] R. Pinciroli, M. Gribaudo, M. Roveri, and G. Serazzi, "Capacity planning of fog computing infrastructures for smart monitoring," in *Workshop on New Frontiers in Quantitative Methods in Informatics*. Springer, 2017, pp. 72–81.

[84] "Aws batch," https://aws.amazon.com/batch/, [Online; accessed 13-January-2020].

Authorized licensed use limited to: TIANJIN UNIVERSITY. Downloaded on November 11,2022 at 12:32:55 UTC from IEEE Xplore. Restrictions apply.

Appendix: Artifact Description/Artifact Evaluation

SUMMARY OF THE EXPERIMENTS REPORTED

We prototype batch atop AWS lambda and conducted the experiments using two machine learning frameworks Tensorflow and MxNet. We use 4 popular computer vision applications (ResNet18, ResNet52, MoBiNet, Inception-V4, and ResNet-V2) to demonstrate the adaptability of BATCH. A theoretical MMPP arrival process, Traffic of the NYS Thruway traces and Twitter traces as our real-world traces are used as a workload. BATCH uses Boto 3 to enable communication with the serverless platform. Every time a new optimal memory size is identified by the performance optimizer, BATCH invokes a low-level API to update the function configuration (i.e., the memory allocated to the function). In order to predict the workload intensity, We used the KPC toolbox. All the performance logs i.e. billing time and latency are collected through AWS cloud watch with default settings. We deploy our model on lambda using the TensorFlow version 1.8.

ARTIFACT AVAILABILITY

Software Artifact Availability: Some author-created software artifacts are NOT maintained in a public repository or are NOT available under an OSI-approved license.

Hardware Artifact Availability: There are no author-created hardware artifacts.

Data Artifact Availability: There are no author-created data artifacts.

Proprietary Artifacts: None of the associated artifacts, author-created or otherwise, are proprietary.

Author-Created or Modified Artifacts:

```
Persistent ID: http://doi.org/10.5281/zenodo.3872213,
↪  https://github.com/rickypinci/BATCH.git
Artifact name: BATCH
```

BASELINE EXPERIMENTAL SETUP, AND MODIFICATIONS MADE FOR THE PAPER

Relevant hardware details: t2micro(Ec2 instance), AWS Lambda, AWS cloud watch

Operating systems and versions: Ubutnu

Applications and versions: ResNet18, ResNet52, MoBiNet, Inception-V4, and ResNet-V2

Libraries and versions: BOTO3, Tensorflow 1.8, MxNet model server

Key algorithms: Computer Vision

Input datasets and versions: ImageNet-22K dataset

URL to output from scripts that gathers execution environment information.

```
https://github.com/rickypinci/BATCH/blob/master/outp⌋
↪  ut.log
```

ARTIFACT EVALUATION

Verification and validation studies: We performed the verification and validation studies on AWS lambda using Tensorflow and MxNet model server. As theoretical arrival processes, MMPP arrivals, and as real arrival trace, we use Traffic of the NYS Thruway and Twitter traces for validating BATCH. Serving requests are extracted from the ImageNet-22K dataset. Popular computer vision ML models such as ResNet18, ResNet52, MoBiNet, Inception-V4, and ResNet-V2 are deployed on Lambda to demonstrate the adaptability of BATCH for different application sizes and complexities.

Accuracy and precision of timings: All the measured values are extracted from the AWS cloud watch and the precision of the timing values in up to milliseconds. We use the default cloudwatch configuration for the collection of logs. BATCH generates three different logs (Lambda logs, Batch logs, and Request logs) files after collecting the logs from cloudwatch. Lambda logs provide detailed information of excitation for each invocation i.e. initialization time, container id, execution time, memory utilization, memory allocated, and billing time. Batch logs provide the information related to each batch i.e. batch arrival time, batch departure time, and batch size. Finally, the request logs provide details of each request such as request arrival time, response time and departure time.

Used manufactured solutions or spectral properties: We used AWS lambda as our main serverless cloud service. Tensorflow 1.8 and MxNet model server is used as a machine learning API. various sizes of computer vision ML models as our serving application deployed on AWS Lambda.

Quantified the sensitivity of results to initial conditions and/or parameters of the computational environment: Overall the performance results are collected through AWS cloud watch and each experiment is conducted long enough to generate stable results(minimum duration 1 hour). During our experiments, we observed the behavior of AWS lambda to be stable for varying parameters such as memory size, batch size, and batch timeout.

Controls, statistics, or other steps taken to make the measurements and analyses robust to variability and unknowns in the system. All the results are collected through amazon's built-in logging mechanism for the cost (billing time) as well as performance. However, during the experiments, we do not take into consideration the network overhead. We take into consideration pure computation time and queuing time (time each request spent in the buffer).

Authorized licensed use limited to: TIANJIN UNIVERSITY. Downloaded on November 11,2022 at 12:32:55 UTC from IEEE Xplore. Restrictions apply.