Optimizing Inference Serving on Serverless Platforms

Ahsan Ali
aali@nevada.unr.edu
University of Nevada, Reno
Reno, NV, USA

Riccardo Pinciroli
riccardo.pinciroli@gssi.it
Gran Sasso Science Institute
L'Aquila, Italy

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

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

ABSTRACT

Serverless computing is gaining popularity for machine learning (ML) serving workload due to its autonomous resource scaling, easy to use and pay-per-use cost model. Existing serverless platforms work well for image-based ML inference, where requests are homogeneous in service demands. That said, recent advances in natural language processing could not fully benefit from existing serverless platforms as their requests are intrinsically heterogeneous.

Batching requests for processing can significantly increase ML serving efficiency while reducing monetary cost, thanks to the pay-per-use pricing model adopted by serverless platforms. Yet, batching heterogeneous ML requests leads to additional computation overhead as small requests need to be "padded" to the same size as large requests within the same batch. Reaching effective batching decisions (i.e., which requests should be batched together and why) is non-trivial: the padding overhead coupled with the serverless auto-scaling forms a complex optimization problem.

To address this, we develop Multi-Buffer Serving (MBS), a framework that optimizes the batching of heterogeneous ML inference serving requests to minimize their monetary cost while meeting their service level objectives (SLOs). The core of MBS is a performance and cost estimator driven by analytical models supercharged by a Bayesian optimizer. MBS is prototyped and evaluated on AWS using bursty workloads. Experimental results show that MBS preserves SLOs while outperforming the state-of-the-art by up to 8 × in terms of cost savings while minimizing the padding overhead by up to 37 × with 3 × less number of serverless function invocations.

PVLDB Reference Format:
Ahsan Ali, Riccardo Pinciroli, Feng Yan, and Evgenia Smirni. Optimizing Inference Serving on Serverless Platforms. PVLDB, 15(10): 2071 - 2084, 2022.
doi:10.14778/3547305.3547313

PVLDB Artifact Availability:
The source code, data, and/or other artifacts have been made available at https://github.com/Iam-ahsan/MBS.

Proceedings of the VLDB Endowment, Vol. 15, No. 10 ISSN 2150-8097.
doi:10.14778/3547305.3547313

1 INTRODUCTION

Serverless computing is widely adopted for data analytics [6, 33] and machine learning (ML) [3–5, 15, 35, 43, 44, 77] tasks including training [16, 28, 37, 77] and inference serving [3, 81], thanks to the attractive pay-per-use pricing model and autonomic resource provisioning. Recent work shows that the serverless paradigm can serve bursty ML inference workloads in a cost effective manner while meeting service level objectives (SLOs) for image recognition tasks with deterministic request sizes and service demands [3]. In recent years, ML is being widely adopted in natural language processing (NLP), including language translation, speech recognition, text to speech, and speech to text conversion. Such applications have request sizes equal to a word, sentence, paragraph, or even an entire document, resulting in heterogeneous service demands.

Cloud computing is the most common choice for serving ML workloads thanks to the on-demand resource availability and flexible pricing (cost) model [62]. During the inference serving stage of the ML development life cycle, trained models are deployed in a cloud environment for classification or prediction. ML inference serving workloads often demonstrate sudden variations in the arrival intensities (known as bursty arrivals [17, 18]) which requires dynamic resource provisioning to meet SLOs. Cloud service providers such as Amazon AWS [63], Microsoft Azure [50], and Google cloud [66] offer Virtual Machines (VMs), container as a service (CaaS), and function as a service (FaaS, also known as serverless) for ML inference workloads. When the above options are used for serving ML inference workloads, drawbacks exist: barebone VMs do not provide dynamic load balancing, CaaS requires a manual selection of appropriate VMs from hundreds of different options, and serverless can result in high monetary cost [81].

Serverless is gaining popularity for serving ML inference workloads due to its automated resource management (e.g., scaling, monitoring) and simple application deployment logic [8]. Users only need to provide a trigger event (e.g., HTTP requests, database uploads), the execution function, and the memory capacity of the serverless instance. The computing capacity and networking performance is automatically scaled using the memory capacity configuration. Serverless supports fast auto-scaling of the number of instances, which helps fulfill the fast changing resource demands when serving bursty inference workloads. During time periods of intense arrivals, instances are transparently created to accommodate workload demands. During low arrival intensity periods

instances are scaled down and the serverless paradigm provides significant cost savings thanks to its pay-per-use cost model. Still, one of the main reservations for adopting serverless for ML inference serving is its higher price, despite the pay-per-use model.

To address the above challenges, recent works [3, 38, 46] propose serving ML inference workloads using *dynamic batching*, which processes requests in bulk to increase system throughput via parallelization. Dynamic batching adjusts batching parameters (batch size, batch timeout) and memory allocation according to changes in workload intensity. The main benefits of dynamic batching for serving ML inference workloads in a serverless setting come from (i) the reduction of the number of function invocations and (ii) the reduction of the service time of requests due to execution parallelism. Dynamic batching results in significantly cheaper costs as the pricing model of serverless is based on the number of function invocations and the length of each function call [61].

While dynamic batching approaches are effective for serving homogeneous ML inference workloads, they fall short when serving heterogenous workloads (i.e., requests with different serving demands) such as in NLP. Our experiments illustrate that for heterogeneous workloads, dynamic batching results in soaring monetary costs and increased request latencies. To solve this problem, we identify three key challenges.

(1) Processing a batch requires all requests within a batch to have the same size. Thus creating a batch with requests in different sizes requires *padding*, typically achieved by adding zeros to smaller requests. This results in additional computations that can lead to longer inference latency, resource waste, and increased monetary cost. Therefore, it is critical for batching decisions to be aware of padding overheads. One solution to control the padding overhead is by separating requests *by size*, i.e., by aiming to batch together requests of *similar* size.
(2) Conventional load balancing methods are designed for systems that are fundamentally different to the serverless paradigm. Typical load balancing methods [83] do not consider the performance implications of padding. More importantly, serverless platforms target a "queueless" design by automatically launching new instances when more requests are present.
(3) The optimization space is prohibitively large due to the many control factors and the complex relationships among them. For the homogeneous workload case, only the batching parameters (batch size, batch timeout) and memory size need to be optimized. In the heterogeneous workload case, we also need to optimize how requests are to be "grouped" together (i.e., the number of buffers where requests of "similar" size accumulate for batching, this number may change according to workload conditions) as well as how to route batched requests to serverless instances.

To address the above challenges, we develop Multi-Buffer Serving[1] (MBS), a framework for serving heterogeneous ML inference workloads with SLO guarantees on serverless platforms. The core of MBS is an analytical model for batching that is aware of the effect of padding overhead on request latency and serverless cost. To address the large optimization space challenge, particularly due to the number of buffers and routing decisions, we supercharge the analytical model with a Bayesian optimizer. To demonstrate the effectiveness of MBS, we prototype it atop AWS Lambda and use DeepSpeech, a popular NLP application, for its evaluation. We validate the adaptability of MBS under varying workload intensities and different request size distributions. MBS achieves up to 8 × cost savings with SLO guarantees compared to the state-of-the-art methods by minimizing the padding overhead by up to 37 × with 3 × less number of function invocations.

[1] https://github.com/Iam-ahsan/MBS

2 MOTIVATION AND CHALLENGES

In this section, we discuss challenges that surface when processing bursty and heterogeneous ML workloads using the serverless paradigm. Our discussion centers on the performance (i.e., request latency and monetary cost) of the state-of-the-practice and the state-of-the-art techniques and presents opportunities for improvements. We motivate the need for a new framework that can minimize the cost of serving modern ML workloads.

2.1 Bursty ML Workloads

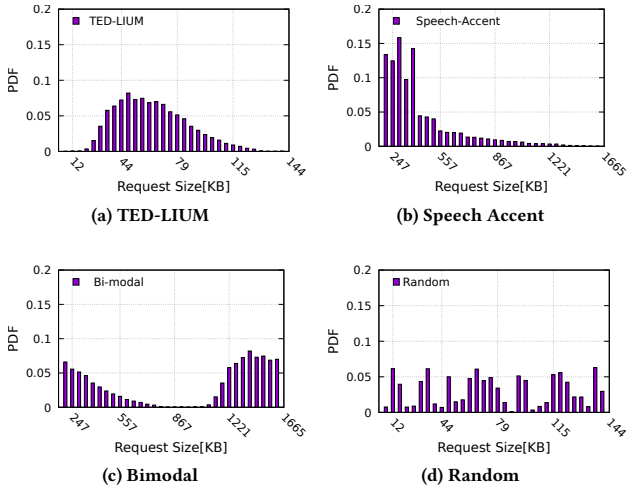
Burstiness is an inherent characteristic of real-world workloads [79], thus performance models need to account for it for accuracy [48, 49]. Major cloud providers (e.g., Amazon AWS [63], Microsoft Azure [50], and Google Cloud [66]) offer Machine-Learning-as-a-Service (MLaaS) to support the scalable execution of ML inference requests that arrive in bursts. Amazon SageMaker [60], an example of such services, allows users to choose when and how much their applications should scale to accommodate all incoming requests. Users are responsible for (i) choosing the computational resources (among hundreds of potential configurations with varying computational capabilities and costs) that best adapt to the workload characteristics, and (ii) selecting auto-scaling parameters that allow the system to scale in or out properly. This makes it extremely challenging for users without extensive expertise to optimize the required parameters for efficient auto-scaling. Moreover, the slow scaling speed of VM-based MLaaS solutions makes their adoption impractical for serving bursty ML workloads within a given SLO [3].

Different frameworks are proposed [3, 32, 81] to overcome the limitations of state-of-the-practice solutions. State-of-the-art approaches leverage serverless computing (i.e., a pay-per-use cloud paradigm that intrinsically supports auto-scaling) to serve bursty ML workloads. While these frameworks work efficiently with requests whose service demand is homogeneous (e.g., models that require fixed-length feature vectors or fixed-length sequences), they perform poorly with heterogeneous workloads (see Section 4) since they are not aware of the service demand difference among requests and the batching overhead caused by the request size difference.

2.2 Heterogeneous ML Inference Requests

Modern real-world applications are characterized by heterogeneous workloads [7, 82], i.e., requests with different service demands. In this paper, we consider ML inference requests with different service demands due to their size, i.e., large inference requests take longer to be processed by the ML model. Inference requests with different size are observed in many ML workloads, e.g., TED-LIUM dataset [69] and Speech Accent [75], two NLP workloads, The former consists of audio files totaling 112 hours of Ted talks from

different speakers, the latter is built from 177 individuals pronouncing words and sentences with different accents. The probability density function (PDF) of their request size is shown in Figures 1(a) and 1(b), respectively, where the x-axis represents the request size in KB and the y-axis shows the probability of observing such a request size in the considered dataset. Graph Neural Network (GNN) applications such as graph classification [42], node classification [22, 39], or link prediction [71] also exhibit similar characteristics with varying number of nodes and edges in each graph. The variability of the request size and, as a consequence, the request service demand pose new challenges during the serving process, especially when requests are executed in batches as done by state-of-the-art approaches [3, 24].



(a) TED-LIUM (b) Speech Accent

(c) Bimodal (d) Random

Figure 1: Request size probability density function.

Batching is a technique used for improving the throughput and resource utilization of ML models during training and inference phases [24]. It consists of grouping multiple requests to form a batch and processing the batched requests in parallel. Batching is easier to do during model training as it is an offline process, where all the samples (and their attributes) to form the batch are known and batches can be formed before starting the training phase. When batching is used to process ML inference workloads on serverless platforms [3], three parameters (i.e., batch size, timeout, and memory size) need to be tuned to optimize the inference phase. Specifically, the batch size is the maximum number of requests grouped in a single batch, timeout is the maximum time that can be waited to create a batch, and memory size is the amount of memory allocated to a serverless function. If the workload is homogeneous (i.e., as in [3]), inference requests are batched together according to their arrival. Since only requests of the same size can be batched together [52], workloads that are heterogeneous pose further challenges.

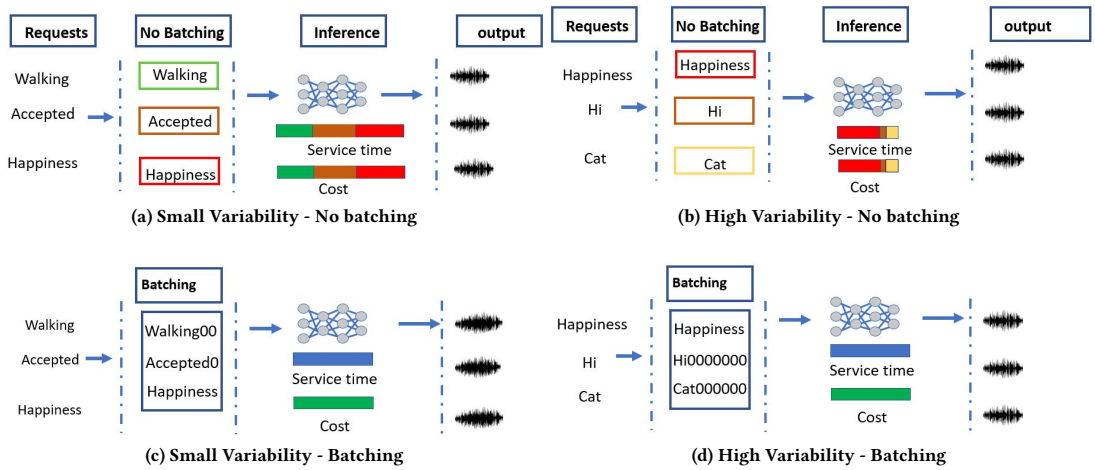
2.2.1 Padding Overhead. Padding is the process of adding dummy information to requests such that all requests of the batch obtain the dimension of the largest request [57]. When ML models process batches with padded requests, padding results in overhead that can affect the latency and monetary cost of inference serving. This is depicted in Figure 2 where (no) batching solutions are visualized for workloads with low and high request size variability. If request size variability is low, processing all requests separately, Figure 2(a), generates inefficiencies that lead to long latency and high monetary cost. Instead, when requests are batched and processed together, Figure 2(c), the padding overhead is negligible while performance and monetary cost improvements are well visible. If request size variability is high, it might be more beneficial to process each request individually, Figure 2(b), since the effect of padding overhead on the request latency is no longer negligible. Improvements observed by batching requests together are not worth the longer latency due to padding overhead, Figure 2(d). Batching heterogeneous ML requests is a non-trivial task, an approach that wisely chooses which requests should be batched (based on their size) is crucial to get the benefits of batching without observing the performance deterioration due to padding overhead.

Figure 3 shows the impact of batching requests for a GNN in a community detection application [22] using the Cora dataset [76]. Figure 3(a) shows the service demand for processing a GNN with fixed edges to 1844 and number of nodes set to 474 and 1096. In the Mixed case, graphs with 474 and 1096 nodes are processed concurrently. When requests are served individually (i.e., the batch size is 1), the service demand depends on the number of nodes in the GNN. Otherwise, if the batch size is larger than 1, batching requests improves performance. In the Mixed case for both experiments (that results in heterogeneous requests), we note that all requests are padded to the largest case and service demands are as long as those observed for GNNs with the largest nodes or edges, see Figure 3(a) and Figure 3(b), respectively.

2.2.2 Buffers and Efficient Request Routing. Due to the stateless nature of serverless functions, batching requests for parallel processing is challenging. Following [3], we deploy queues (i.e., buffers) on a front-end server to hold requests that are batched together before forwarding the batch to one of the available serverless functions. BATCH [3] uses only a single buffer to collect requests and one serverless function to process batches. When serving heterogeneous workloads, a single buffer solution can incur high latency and cost penalties and cause SLO violations as the time and monetary cost required to process dummy information (i.e., padding overhead) can be quite high. On the other hand, deploying a dedicated buffer and serverless function for each request size reduces batching opportunities and diminishes the benefits of batching (e.g., high throughput and low latency [24]). To strike the balance between batching opportunities and padding overhead, it is important to (i) estimate the optimal number of buffers and serverless functions required for processing the observed workload and (ii) route similar requests to the same buffer.

2.3 Optimization Space

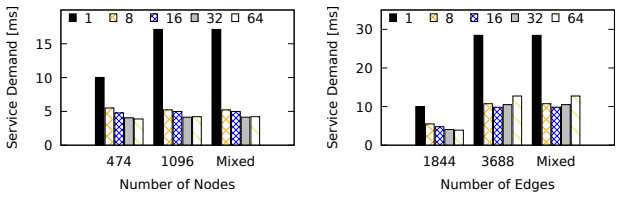
Many parameters need to be tuned to optimize the processing of bursty and heterogeneous ML workloads on serverless computing. Specifically, one should consider: 1) the number of serverless functions (F) and the routing strategy (R) that minimize the padding overhead and maximize batching opportunities; 2) the amount of memory (M) to allocate to instantiated functions such that the SLO is not violated; 3) the batching configuration, i.e., batch size



(a) Small Variability - No batching (b) High Variability - No batching

(c) Small Variability - Batching (d) High Variability - Batching

Figure 2: Serving heterogeneous requests with and without batching.



(a) GNN: Fixed number of edges (b) GNN: Fixed number of nodes

Figure 3: Impact of varying the number of nodes or edges when requests are processed in batches of 1, 8, 16, 32, or 64.

(B) and timeout (T), which allows reducing the number of function invocations while meeting given performance requirements. Profiling all possible configurations (i.e., $R \times F \times B \times T \times M$) to solve an optimization problem characterized by these dimensions is unfeasible. If every system configuration takes t time units on average to achieve statistical stability, the profiling cost would be $R \times F \times B \times T \times M \times t$ time units. Analytical approaches significantly reduce the time needed to compute the performance and monetary cost of each system configuration. However, the time required to optimize a system may still be too long due to the size of all considered dimensions. Specifically, F, B, and T may grow to infinity since cloud providers do not limit these parameters; R depends on how many functions are available and how much heterogeneous is the considered workload; M is finite (e.g., AWS Lambda allows choosing among 80 different memory capacities [61]), but more memory configurations might be available in the future for serverless computing. This poses new challenges since analytical solutions cannot be used to evaluate all possible system configurations (i.e., exhaustive search). We propose to supercharge analytical solutions with Bayesian optimization, an approach that limits the search space by early identifying promising system configurations.

3 MBS DESIGN AND IMPLEMENTATION

To address the aforementioned challenges we develop MBS. In the following, we provide an overview of the proposed framework, introduce the optimization problem that drives MBS, describe Bayesian optimization, and discuss the analytical model that determines the optimal system configuration.

3.1 Overview of MBS

An overview of MBS is shown in Figure 4. The main components of MBS are *Profiler*, *Optimizer*, *Router*, and *Buffers*. Dashed lines show the request workflow and solid ones the control flow.

ML inference requests submitted by users to the system ⓿ are analyzed by the *Profiler* to extract their main attributes (i.e., arrival intensity and request size distribution). Requests and their extracted attributes are forwarded ❶ to the *Router* and the *Optimizer*, respectively. Simultaneously, the user-defined SLO and the batch service time (previously profiled, see Section 3.4.1) are communicated to the *Optimizer*. The *Optimizer* solves the optimization problem of Eq. (1) using Bayesian optimization (see Section 3.3) to reduce the number of system configurations that are evaluated using the analytical model (see Section 3.4). The *Optimizer* provides *the optimal system configuration* that allows to minimize the cost of serving ML requests. This configuration is used ❷ for instantiating the optimal number of buffers and serverless functions, for setting the maximum batch size and timeout of all *Buffers*, for allocating the desired memory to each serverless function, and for guiding the *Router* to forward requests to one of the available *Buffers* based on the request size. After receiving the setting configuration, ❸ the *Router* directs user requests to *Buffers* by routing similar requests to the same buffer to minimize their padding overhead while aiming for an equal load distribution across buffers. Once at a buffer, requests are stored until the buffer reaches the maximum batch size or until the time since the first request is stored exceeds the timeout. As soon as one of the above conditions holds, the requests stored in the buffer form a batch ❹ that is sent to the serverless function associated with the buffer (i.e., a buffer sends all its batches to the function deployed with enough memory to serve the requests

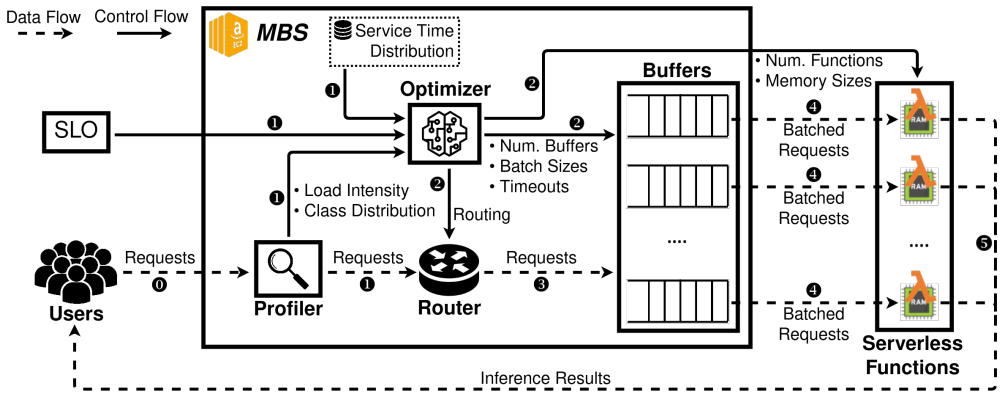


Figure 4: Design of MBS. Dashed and solid lines represent the data and control flows, respectively.

stored in that buffer). After the requests in the batch are processed by the serverless function, inference results are sent ❺ to users.

3.2 Problem Formulation: Multi-Buffer System

Here, we extend the optimization problem for homogeneous workloads that is first presented in [3] to support heterogeneous workloads *and* multi-buffers. MBS is driven by Equation 1, which minimizes the monetary cost of serving ML requests with serverless functions while meeting additional constraints. The cost depends on the maximum batch size and timeout of each buffer (i.e., $\vec{B}$ and $\vec{T}$, respectively), as well as on the memory allocated to each serverless function (i.e., $\vec{M}$). The length of the three vectors is equal to the number of instantiated buffers and serverless functions, i.e., $\|\vec{B}\| = \|\vec{T}\| = \|\vec{M}\| = K$.

$$\begin{aligned} \text{minimize} \quad & \text{Cost}_{request}(\vec{B}, \vec{T}, \vec{M}) \\ \text{subject to} \quad & F_R^{-1}(i/100) \leq \text{SLO} \quad \text{(a)} \\ & K \geq 1 \quad \text{(b)} \\ & B_k \geq 1 \quad \text{(c)} \\ & T_k > 0 \text{ msec} \quad \text{(d)} \\ & 128\text{MB} \leq M_k \leq 10240 \text{ MB}, \quad \text{(e)} \end{aligned} \tag{1}$$

The additional constraints are: (a) i% of requests are served within a user-defined SLO, (b) at least one buffer and one serverless function are instantiated, (c) the maximum batch size, (d) buffer timeout, and (e) the memory allocated to each serverless function (between 128 MB and 10 GB, i.e., values accepted by AWS Lambda [61]). Eq. (1) defines an infinite search space, thus it is not practical to solve this problem via exhaustive search. Instead, we propose to use an analytical model that relates the system configurations (i.e., number of buffers and serverless functions K, the maximum batch size B_k, buffer timeout T_k, and memory allocated to each function M_k) to the latency distribution and the monetary cost of requests. To further power the analytical model, we use Bayesian optimization by evaluating only a small share of all points [73] to solve Eq. (1).

3.3 Bayesian Optimization

MBS adopts Bayesian optimization [73] to contain the time required to solve that optimization problem in Eq. (1) and power the analytical model, see Section 3.4 that is the core of MBS. Differently from exhaustive search, Bayesian optimization finds the optimal point (i.e., $\vec{x}_{opt}$) which minimizes the function, $f(\vec{x})$, by analyzing only a finite number of system configurations (i.e., $\vec{x} \in X$), that is:

$$\vec{x}_{opt} = \arg\min_{\vec{x} \in X} f(\vec{x}). \tag{2}$$

In this paper, $\vec{x}$ is a vector containing all parameters of a configuration (i.e., $\vec{B}$, $\vec{T}$, and $\vec{M}$), X is the set of all possible configurations, and the function f is the optimization function in Eq. (1), i.e., $\text{Cost}_{request}$. Bayesian optimization leverages an acquisition function, which depends on the knowledge obtained from the prior belief, to choose the new point $\vec{x}$ (i.e., number of buffers, maximum batch size, timeout, and allocated memory) to be considered next [14]. Specifically, the MBS implementation of Bayesian optimization assumes a *Gaussian process* prior and adopts the *Upper Confidence Bound* (UCB) acquisition function[2]. A Gaussian process [59] is a stochastic process for which any combination of random variables (i.e., $\mathbf{x} = \{\vec{x}_i : \vec{x}_i \in X\}$) follows a Gaussian distribution defined by a mean function, $\mu(\mathbf{x})$, and a variance function, $\sigma^2(\mathbf{x})$. UCB [74] is an acquisition function that minimizes the regret, i.e., the difference between the return obtained with the optimal policy and the one achieved with the policy suggested by the acquisition function. For this purpose, UCB selects the next point to evaluate by moving from *exploration* (i.e., take the action with more uncertainty) when the knowledge is small, to *exploitation* (i.e., take the action that looks the best) when the knowledge increases, as [11]:

$$\vec{x}_{next} = \mu(\vec{x}|\mathbf{x}) - \kappa \cdot \sigma(\vec{x}|\mathbf{x}), \tag{3}$$

where κ can be tuned to prefer exploitation to exploration. We refer the interested reader to [12, 29, 73] for further details.

3.4 Analytical Model

We describe the analytical model used by MBS to compute the request latency distribution, $F_R(t) = P(R \leq t)$, and the monetary cost of serving ML inference requests using a serverless platform, $\text{Cost}_{request}$. The model leverages Markovian Arrival Processes (MAPs) that capture the performance implications of workload burstiness to accurately predict the relationship between the request latency distribution and the system configuration, i.e., the number of buffers and serverless functions (K), buffer parameters

[2]This acquisition function is called *Lower Confidence Bound* in the context of minimization as in our case, see Eq. (1). However UCB is a standard term in the literature.

(maximum batch size, B_k, and timeout, T_k), and the memory allocated to each serverless function (M_k). In the following, we provide definitions to support the analytical model described in Section 3.4.2 that solves the optimization problem in Eq. (1).

3.4.1 Definitions. This section summarizes how MBS use MAPs to model bursty workloads, the profiling of batch service times, and the batch size. Typically, analytical models require as inputs stochastic models that capture the arrival and service processes. MAPs are widely used in the literature to capture burstiness in arrivals [10, 18, 20, 53, 68], this is the approach that we follow here. To model the service process, we extend the methodology presented in [3] to address the challenges of heterogeneous requests. In the following we give an overview of MAPs and the methodology used in [3] to model homogeneous requests.

Arrival Process. MBS uses *KPC-Toolbox* [19] to fit the inter-arrival time of each buffer with a MAP, a non-renewal stochastic process that models general distribution. A MAP(m) is defined by two $m \times m$ matrices, **D0** and **D1**: the former (a matrix with negative diagonal and non-negative off-diagonal elements) represents *hidden* events, i.e., events that are not related to an arrival; the latter (a non-negative matrix) represents *observable* events that correspond to an arrival. The infinitesimal generator matrix **Q** of a MAP(m) is defined by matrices **D0** and **D1** as:

$$\mathbf{Q} = \begin{bmatrix} \mathbf{D0} & \mathbf{D1} & \mathbf{0} & \mathbf{0} & \dots \\ \mathbf{0} & \mathbf{D0} & \mathbf{D1} & \mathbf{0} & \ddots \\ \mathbf{0} & \mathbf{0} & \mathbf{D0} & \mathbf{D1} & \ddots \\ \vdots & \ddots & \ddots & \ddots & \ddots \end{bmatrix}. \tag{4}$$

Here, the arrival process of each buffer is modeled by a MAP(2) (i.e., a two-state map) defined by matrices:

$$\mathbf{D0} = \begin{bmatrix} -(\lambda_1 + \lambda_{12} + \omega_1) & \omega_1 \\ \omega_2 & -(\lambda_2 + \lambda_{21} + \omega_2) \end{bmatrix} \quad \text{and} \quad \mathbf{D1} = \begin{bmatrix} \lambda_1 & \lambda_{12} \\ \lambda_{21} & \lambda_2 \end{bmatrix}, \tag{5}$$

where ω_1 and ω_2 are rates of exponential distributions at which the process changes its phase, λ_1 and λ_2 are the arrival rates observed during each phase, and λ_{12} and λ_{21} are rates at which requests arrive while the phase changes simultaneously. The state space of a buffer k whose arrival process is defined by the MAP(2) in Eq. (5) is shown in Figure 5. Each state (i, j) represents the buffer when i requests are collected and the arrival process is in phase $\phi = j$, with $j = \{1, 2\}$ since the arrival process is defined by a MAP(2). The number of requests concurrently stored in a buffer k is larger than 0 (i.e., the buffer state is monitored after the first request of a batch arrives and the timeout, T_k, starts) and smaller than B_k (i.e., the maximum batch size of the buffer).

Batch Service Time. MBS needs the service time distribution of batches (besides the user-defined SLO, the load intensity, and the request size distribution) to find the system configuration that solves Eq. (1). Such information is retrieved empirically through profiling. Exhaustive profiling is not appropriate due to the enormous search space (defined by the maximum batch size, memory, and request size) that must be considered when profiling the service time distribution of batches. Indeed, the maximum batch size does not have an upper-bound, the request size depends on the analyzed scenario, and AWS allows allocating from 128 MB to 10 GB (in 1 MB increments) to a serverless function. MBS adopts a *lightweight profiling approach* to profile the batch service time of only a few configurations and estimates missing points through regression. Since the batch service time of machine learning inference is typically deterministic [80] for fixed memory, maximum batch size, and request size, the lightweight profiling strategy allows obtaining accurate results in a short time.

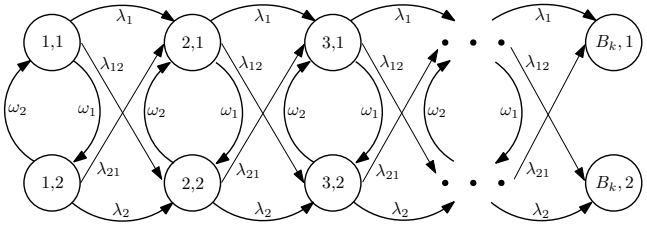


Figure 5: State space of a buffer k [3] in MBS, whose arrival process is defined by the MAP(2) in Eq. (5).

Batch Size Distribution. MBS must compute the request latency distribution to guarantee that the user-defined SLO is met. Since the service time of a batch (and its requests) depends on the *batch size*, MBS derives the batch size distribution (i.e., the number of requests collected by a buffer k within its timeout T_k) for all buffers in the system. Since no more than B_k requests can be collected in a buffer k (i.e., B_k is the maximum batch size of a buffer k), the arrival process of the k-th buffer is defined by the $2B_k \times 2B_k$ matrix:

$$\hat{\mathbf{Q}} = \begin{bmatrix} \mathbf{D0} & \mathbf{D1} & & \\ & \ddots & \ddots & \\ & & \mathbf{D0} & \mathbf{D1} \\ & & & \mathbf{0} \end{bmatrix}. \tag{6}$$

The probability that there are $0 < n \leq B_k$ requests in the buffer k at time T_k and its arrival process is in phase $\phi = \{1, 2\}$ is:

$$\vec{\pi}(T_k) = \vec{\pi}(0) e^{\hat{\mathbf{Q}} T_k}, \tag{7}$$

where $\vec{\pi}(T_k) = \{\pi_{n,\phi}(T_k)\}$ is the state space vector at time T_k, $\vec{\pi}(0)$ is the initial state probability vector, and $e^{\hat{\mathbf{Q}} T_k}$ is the matrix exponential:

$$e^{\hat{\mathbf{Q}} T_k} = \sum_{i=0}^{\infty} \hat{Q}^i \cdot \frac{T_k^i}{i!}. \tag{8}$$

Aggregating $\vec{\pi}(T_k)$ over all phases ϕ, i.e., $\sum_{\phi=1}^{2} \pi_{n,\phi}(T_k)$, we derive the probability that n requests are in buffer k at time T_k.

The initial state probability vector, $\vec{\pi}(0)$, defines the probability that the arrival process of buffer k is in phase ϕ when the first request of a batch is stored. Although there are $2B_k$ elements in $\vec{\pi}(0)$, only $\pi_{1,1}(0)$ and $\pi_{1,2}(0)$ may be non-zero and $\pi_{1,1}(0) + \pi_{1,2}(0) = 1$, i.e., when the buffer timeout starts at time $t = 0$, only one request is in the buffer and the arrival process is either in phase $\phi = 1$ or in phase $\phi = 2$. $\vec{\pi}(0)$ is computed by deriving the average arrival rate of each phase of the arrival process, $\vec{\lambda} = (\lambda_1, \lambda_2)$, as:

$$\vec{\lambda} = \left(\frac{\omega_2}{\omega_1 + \omega_2}, \frac{\omega_1}{\omega_1 + \omega_2} \right) \times \mathbf{D1}, \tag{9}$$

where $\frac{\omega_\phi}{\omega_1+\omega_2}$, for $\phi = \{1, 2\}$, is the probability to be in phase 2 or 1, respectively. The average rate of phase ϕ is divided by the expected batch size as:

$$\alpha_\phi = \frac{\lambda_\phi}{\min\left(B_k, \lambda_\phi \cdot T_k + 1\right)}, \tag{10}$$

and the probability that the arrival process is in phase $\phi = \{1, 2\}$ when the first request of a batch arrives to the buffer is:

$$\pi_{1,\phi}(0) = \frac{\alpha_\phi}{\alpha_1 + \alpha_2}. \tag{11}$$

3.4.2 Modeling Heterogeneous Requests. This section extends the analytical model in [3] to tackle new challenges that are faced while serving ML applications and their heterogeneous requests (i.e., we assume C types of requests). Specifically, we describe in detail how MBS determines the number of buffers (and serverless functions, K) that will serve inference requests, the routing strategy used to distribute incoming requests to available buffers, and models for request latency and monetary cost used to solve Eq. (1).

Request Size Distribution. MBS derives the request size distribution by observing the system workload. The request size distribution is critical since it is used to derive the optimal number of buffers and their arrival process.

Number of Buffers. Determining the number of buffers to store incoming requests and create batches allows optimizing the monetary cost and serving latency. MBS can serve each request individually to reduce the latency or store all requests together (i.e., in a single buffer, $K = 1$) to facilitate the creation of large batches before the timeout expires. Since one of the parameters determining the monetary cost of a serverless function is the number of invocations [62], serving more requests in a single batch decreases the monetary cost. This comes at the expense of a longer latency since the batch service time increases with its size (i.e., the number of requests included in the batch). On the contrary, smaller batches decrease latency but increase monetary cost. The optimal number of buffers (as well as other system parameters, i.e., maximum batch size, timeout, and allocated memory) is provided by Bayesian optimization (see Section 3.3).

Routing Strategy. After solving the optimization problem in Eq. 1, the *Optimizer* communicates the request routing strategy to the *Router*. Routing depends on the number of instantiated buffers and on the request sizes (types) observed by the *Profiler*. The routing strategy *i)* equally splits incoming requests among all available buffers and *ii)* routes requests of similar size to the same buffer. As an example, assume that the *Profiler* classifies incoming requests into three types (e.g., 70% small, 5% medium, and 25% large) and that the *Optimizer* instantiates two buffers only. The *Router* redirects half of the load to the first buffer and the other half to the second one, while taking care of routing similar requests to the same buffer, i.e., the first buffer serves only small requests and the second one serves all other requests.

Request Service Time Distribution. When the number of request classes, C, is larger than the number of buffers, K, a batch might be made of different request classes since requests with different characteristics (i.e., size) may be routed to the same buffer. MBS computes the probability that a batch created by buffer k is made of specific request sizes using a Monte Carlo approach [47], which generates a large number of requests in a short time. Specifically, MBS derives the probability, $p_n(\gamma)$, that a batch of size $n \leq B_k$ has a composition $\gamma \in \Gamma(C_k, n)$, where C_k is a set containing all request classes routed to a buffer k by the *Router*. $\Gamma(C_k, n)$ is another set which contains the $\|C_k\|^n$ permutations (i.e., compositions) of a batch of size n, where $\|C_k\|$ is the cardinality of C_k. The probability, $\bar{\rho}_{ck}$, that a request of class $c \in C_k$ is included in a batch of size n is:

$$\bar{\rho}_{cn} = \frac{\rho_{cn}}{\sum_n \rho_{cn}}. \tag{12}$$

Specifically,

$$\rho_{cn} = \sum_{\gamma \in \Gamma(C_k, n)} count(c, \gamma) \cdot p_n(\gamma), \tag{13}$$

and $count(c, \gamma)$ is the number of requests of class c in a batch with composition γ.

Due to the request padding mechanism described in Section 1, the service time of a batch depends on its size (n) and its largest request (c_{max}). All requests batched with c_{max} spend the same amount of time of c_{max} to be processed by the serverless function. Thus, MBS computes the service time of a batch (and all its requests) using the batch service time distribution obtained through lightweight profiling, the batch size distribution as in Eq. (7), and the probability of having a request c_{max} in a batch of size n, Eq. (12).

Latency and Cost CDF. After deriving the service time distribution of all request classes, S_c, MBS computes the latency distribution of each class, i.e., $F_{R_c}(t) = P(R_c \leq t)$, by adding the estimated waiting time W_c to the request service time. The waiting time is computed as:

$$W_c = \begin{cases} 0 & if \quad B_k = 1 \\ \frac{T_k}{n} & if \quad n < B_k \\ \min\left(T_k, \frac{n-1}{\hat{\lambda}}\right) & if \quad n = B_k \end{cases}, \tag{14}$$

where n is the size of the batch and $\hat{\lambda}$ is the average arrival rate that, for a MAP(2), is derived through Eq. (9) as:

$$\hat{\lambda} = \vec{\lambda} \times \begin{pmatrix} 1 \\ 1 \end{pmatrix} = \left(\frac{\omega_2}{\omega_1 + \omega_2}, \frac{\omega_1}{\omega_1 + \omega_2}\right) \times \mathbf{D1} \times \begin{pmatrix} 1 \\ 1 \end{pmatrix}. \tag{15}$$

The service time of a batch with n requests whose the largest one has size is c_{max}, i.e., $S(n, c_{max})$, is used to compute the monetary cost for serving such a batch and its n requests. Serving a batch created with requests stored in a buffer k costs [62]:

$$\text{Cost}_{batch} = S(n, c_{max}) \cdot M_k \cdot K_1 + K_2, \tag{16}$$

where M_k is the memory allocated to the serverless function, K_1 (i.e., $1.66667 \cdot 10^{-5}$ \$/GB-s) is the memory cost, and K_2 (i.e., $2 \cdot 10^{-7}$ \$) is the invocation cost. The cost of each request in a batch of n requests is derived from Eq. (16) as: $\text{Cost}_{request} = \text{Cost}_{batch}/n$.

4 EVALUATION

We analyze MBS using speech-to-text applications. After evaluating the accuracy of the model, we study the performance of MBS under various workload intensities and request size distributions.

4.1 Experimental Setup

MBS prototype. We prototype MBS atop a single AWS EC2 instance (i.e., t2.xlarge) due to its small computational requirements. MBS is placed between the request source and the serverless infrastructure. In this paper, the serverless platform of choice is AWS Lambda [61]. However, MBS can operate and interact with any serverless environment (e.g., Google Cloud [65], Azure [67]) as it does not use any cloud provider specific feature. The main differences among these platforms are the underlying hardware capabilities and cost, which are the input of MBS.

Baseline Policies. We compare MBS to four different static scheduling policies and BATCH, the state-of-the-art approach [3]. The three static policies use a fixed number of buffers (i.e., 8, 16, and 20) to process heterogeneous ML inference requests. BATCH can serve bursty inference workloads using the FaaS paradigm.

ML Applications and Request Size Distribution. The performance of MBS and baseline policies is evaluated with four heterogeneous workloads (i.e., TED-LIUM dataset [69], Speech Accent [75], Bi-modal, and Random) whose PDFs are shown in Figure 1. All bins of those PDFs have the same width, i.e., 4 KB. TED-LIUM corpus (TL) [69] is a real-world NLP workload that consists of audio files totaling 112 hours of read speech from different speakers. The size of its files varies from 12 KB to 144 KB, with mean request size of 68.73 KB, and standard deviation of 20.19 KB. Another real-world workload is obtained from the Speech Accent (SA) dataset [75] that consists of words and sentences pronounced by 177 individuals with different accents. Its request size distribution is skewed toward small requests (i.e., mean request size is 475.26 KB and standard deviation is 252 KB). We generate also two synthetic workloads: Bi-modal (B) and Random (R). The former shows two peaks (i.e., modes) in the PDF, one for small and the other for large requests. The latter has different (random) probabilities for each request size.

Bursty Workloads. We evaluate the performance of MBS using two arrival traces: a real-world one from Twitter [64], see Figure 7(a), and a synthetic one, see Figure 9(a). The load intensity observed in the Twitter trace over a period of 240 minutes varies between 1900 and 2500 req/min. The synthetic trace is used to evaluate MBS in the presence of high and sudden request arrival patterns as observed in Microsoft production traces [31], where the arrival intensity varies by 50 ×. We generate the synthetic trace by scaling the arrival intensity of the Twitter trace 4 × and 10 × to generate alternating periods of sudden low/high arrival intensity.

4.2 Model Validation

We investigate the accuracy of the analytical model presented in Section 3.4 by predicting the latency of ML inference requests executed on AWS Lambda [61]. We compute the error as:

$$Error = \frac{|Latency_{Model} - Latency_{AWS}|}{Latency_{AWS}} \cdot 100, \qquad (17)$$

where $Latency_{Model}$ is the request latency predicted by the analytical model and $Latency_{AWS}$ is the latency observed to process the request on AWS Lambda. Figure 6 shows the error (y-axis) made by the proposed model when the arrival intensity of requests is driven by the real-world trace (i.e., Twitter). Independently of the number of buffers used by MBS to store incoming requests (see the x-axis) and the adopted request size distribution, the analytical model shows a small prediction error, i.e., the maximum error observed is always less than or equal to 10%.

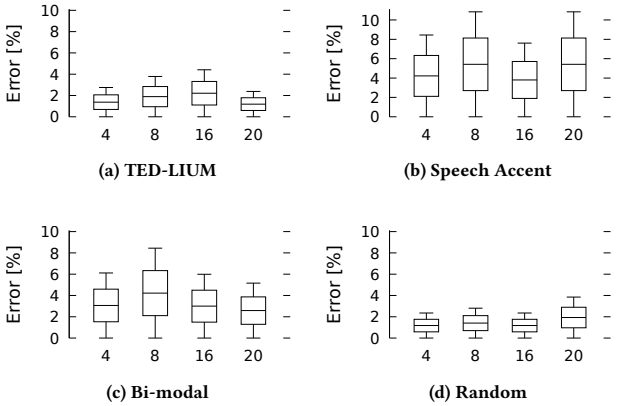


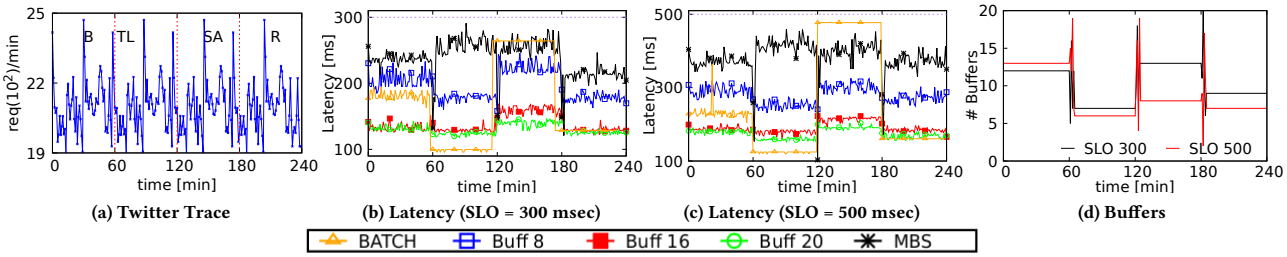
Figure 6: Model error distribution.

4.3 A Real-World Workload: Twitter Traces

In this section, we evaluate the effect of heterogeneous workloads with arrivals driven by the Twitter trace, see Figure 7(a). To highlight the benefit of using MBS to serve ML inference requests in a serverless environment, we compare the performance and cost of the proposed framework with the baseline approaches.

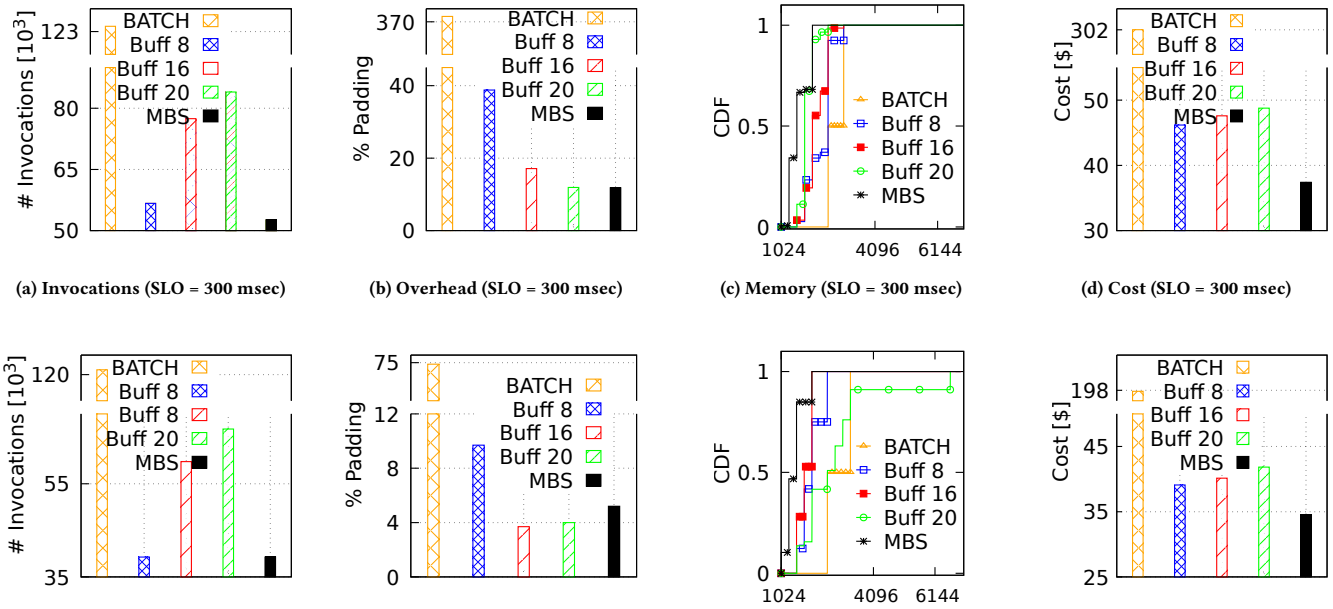
4.3.1 Request Latency. The 95th percentile latency of ML inference requests obtained using MBS and other baselines is shown in Figures 7(b) and 7(c) when the SLO (i.e., horizontal purple line) is set to 300 and 500 msec, respectively. To evaluate how well MBS adapts to variations in the request size distribution, we change the size of processed requests every 60 minutes, i.e., Bi-modal up to 60 minutes, TED-LIUM from 60 to 120 minutes, Speech Accent between 120 and 180 minutes, and Random until the end of the experiment, see the x-axis of Figure 7(a) where vertical red lines show the time at which the request size distribution changes. All considered approaches always meet the SLO independently of its value and the request size distribution as depicted in Figures 7(b) and 7(c). MBS stays closer to the SLO than other approaches, i.e., it uses the available resources in a more judicious manner. The only exception is observed with the Speech Accent request size distribution (i.e., from 120 to 180 minutes) for both SLO values, when BATCH performs as well as or better than MBS.

4.3.2 Number of Buffers. The number of buffers instantiated over time by MBS to optimize the cost of serving ML inference requests with serverless computing is shown in Figure 7(d) for both SLO values. Based on the observed request size distribution and load intensity, MBS changes the number of buffers used for collecting inference requests to minimize the monetary cost of the system while meeting the given SLO. With stringent SLOs, MBS instantiates a large number of buffers to generate small batches of requests that are processed in a short time. Loose SLOs allow MBS to prioritize the monetary cost over the performance as a few buffers are used to create large (and slow) batches.



(a) Twitter Trace (b) Latency (SLO = 300 msec) (c) Latency (SLO = 500 msec) (d) Buffers

Figure 7: Twitter trace (a), the request latency provided by MBS and baseline frameworks when *SLO=300* msec (b) and *SLO=500* msec (c), and the number of buffers used by MBS for both SLOs (d).



(a) Invocations (SLO = 300 msec) (b) Overhead (SLO = 300 msec) (c) Memory (SLO = 300 msec) (d) Cost (SLO = 300 msec)

(e) Invocations (SLO = 500 msec) (f) Overhead (SLO = 500 msec) (g) Memory (SLO = 500 msec) (h) Cost (SLO = 500 msec)

Figure 8: Performance of MBS and baseline policies to serve ML inference requests whose arrival process is defined by the Twitter trace and the SLO is set to 300 and 500 msec.

4.3.3 Number of Invocations. By adjusting the number of buffers, serverless functions, and timeout to the workload changes, MBS optimizes the number of invocations to serverless functions, i.e., one of the parameters which contribute to the monetary cost of serverless computing, see Eq. (16). Figures 8(a) and 8(e) depict the number of functions called by all approaches when the SLO is set to 300 and 500 msec, respectively. These results show that MBS reduces the number of invocations of serverless functions when compared to other approaches. Since MBS instantiates more buffers when the SLO is strict, the number of function calls is larger when SLO = 300 msec.

4.3.4 Padding Overhead. The padding overhead of MBS and baselines is shown as a percentage of additional processed information in Figures 8(b) and 8(f) for *SLO* = 300 msec and *SLO* = 500 msec, respectively. BATCH is the framework which shows the largest overhead since it is not aware of heterogeneity, requests are all collected in a single buffer and batched together independently of their size. Other policies that use a fixed number of buffers have less overhead than BATCH. Using multiple buffers allows routing incoming requests to different serverless functions and batches are created using only similar requests that are collected in the same buffer. This means that when the number of buffers increases the observed overhead is small since grouping together identical requests allows creating batches without padding requests. MBS shows the smallest padding overhead when *SLO* = 300 msec, overhead is slightly higher than with 16 and 20 buffers if the SLO is set to 500 msec. This is due to MBS preferring a smaller number of invocations, see Figure 8(e), to a smaller padding overhead in order to optimize the monetary cost.

4.3.5 Memory Allocation. A serverless function with high memory allocation can process a request in a short time but the amount of

allocated memory is proportional to the monetary cost of serverless functions, see Eq. (16). Hence, we compare the distribution of memory allocated to serverless functions by MBS and baselines when the SLO is set to 300 and 500 msec, see Figures 8(c) and 8(g), respectively. The two figures show the allocated memory (in MB) on the x-axis and its distribution (i.e., CDF) on the y-axis. MBS always allocates less memory than all other approaches and achieves to minimize the monetary cost while meeting the given SLO.

4.3.6 Monetary Cost. The monetary cost of serving ML inference requests with serverless computing by using MBS and other approaches is shown in Figures 8(d) and 8(h) for $SLO = 300$ msec and $SLO = 500$ msec, respectively. For both SLO values, MBS shows a 25% improvement on monetary cost comparing to others. Compared to BATCH, MBS enables $8\times$ and $7\times$ monetary cost saving for $SLO = 300$ msec and $SLO = 500$ msec, respectively. MBS decreases the monetary cost of the considered system by increasing batching opportunities, optimizing the padding overhead, and efficiently using available resources (i.e., memory).

4.4 Large and Sudden Load Variations

We evaluate the efficiency of MBS using synthetic workloads with load intensity surges larger than those observed in the Twitter trace. Figure 9(a) depicts the workload used for experiments shown in this section over 360 minutes. The load intensity changes every hour; during time frames 0–60 and 240–300 it is the same that is observed in the Twitter trace. Time frames 120–180 and 300–360 show a $4\times$ larger intensity than previous than the Twitter one. From 60 to 120 and from 180 to 240, the load intensity is $10\times$ larger. Two request size distributions are considered with this workload, i.e., TED-LIUM and Bi-modal.

4.4.1 Request Latency. Figures 9(b) and 9(c) show the 95th percentile latency of requests processed with MBS and other scheduling techniques when the SLO is set to 300 and 500 msec, respectively. MBS keeps request latency consistently close to (and shorter than) the SLO, independently of the load intensity and the request size distribution. Overall, MBS outperforms all from latency perspective. Other techniques cope only with SLO and load variations hence, although they do not violate the latency constraints and in some cases they get closer to the SLO than MBS, the latency of requests served with these strategies is generally far from the given constraint, i.e., available resources are wasted while serving ML requests.

4.4.2 Number of Buffers. MBS quickly adapts the number of deployed buffers to system conditions even with high and sudden load variations. This is shown in Figure 9(d), where the number of buffers used to collect ML requests for both SLO values is depicted as a function of time. Similar to the Twitter trace, MBS instantiates more buffers with a tighter SLO between 0 and 120 minutes.

4.4.3 Number of Invocations. As depicted in Figures 10(a) and 10(e) for $SLO = 300$ and $SLO = 500$ msec, respectively, MBS minimizes the number of function calls even when high and sudden surges are observed in the workload. For $SLO = 300$ msec, MBS performs as well as one of the static approaches (i.e., 8 buffers). Otherwise, MBS is largely better than competitors.

4.4.4 Padding Overhead. The padding overhead of MBS and other approaches is depicted in Figures 10(b) and 10(f) for $SLO = 300$ and $SLO = 500$ msec, respectively. When the SLO is set to 300 msec, MBS is the approach that allows minimizing the request padding. For $SLO = 500$ msec, MBS performs worse than the static approach using 20 buffers. However, the padding overhead alone is not a sign of bad performance. As observed also for the Twitter trace, MBS prefers a high padding overhead to significantly reduce the number of invocations, see Figure 10(e).

4.4.5 Memory Allocation. Figures 10(c) and 10(g) show the distribution of memory allocated by MBS and other static approaches to serve ML requests with $SLO = 300$ and $SLO = 500$ msec, respectively. Also with large and sudden variations in the workload, MBS outperforms other techniques.

4.4.6 Monetary Cost. Figures 10(d) and 10(h) show the monetary cost of processing ML inference requests with serverless computing and different serving frameworks when the SLO is set to 300 and 500 msec, respectively. Compared to static approaches, MBS provides the smallest monetary cost even when the workload is subject to sudden and high intensity variations.

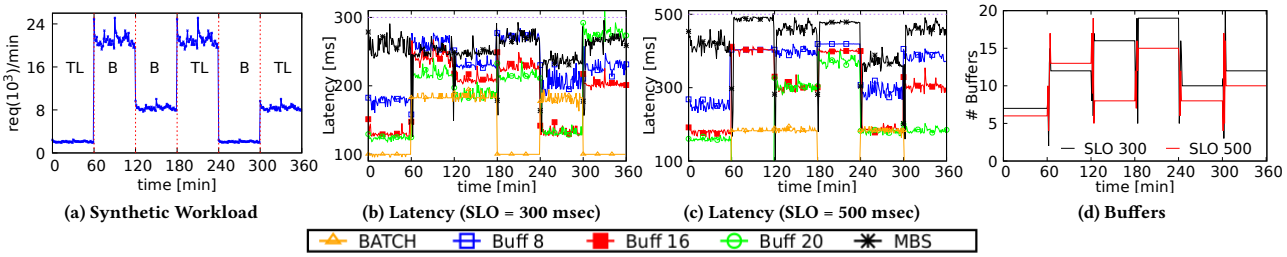
4.5 Monetary Cost and SLO

The monetary cost of ML models is affected by the user-defined SLO since looser performance requirements allow for greater savings. In Figure 11, we investigate the effect of SLO on the monetary cost of serving heterogeneous ML requests with BATCH and MBS. For this purpose, we use the Twitter trace to drive the workload intensity and we assume that the request size distribution is defined by the Bi-modal function.

Results show that MBS always outperforms BATCH independently of the considered SLO. Specifically, MBS saves $4\times$ more than BATCH to serve workloads with tight SLOs, i.e., less than 300 msec. MBS is aware of the request size distribution and batches similar requests together to minimize the monetary cost while meeting the performance constraint. Instead, BATCH cannot distinguish among different requests and, with tight SLOs, it serves all requests individually to meet the given SLO at the expense of a higher monetary cost. When the SLO is relaxed (i.e., larger than 300 msec), BATCH starts batching requests and processing them in parallel. Although this is an inefficient batching since BATCH is unaware of request size distribution and padding overhead, it allows a significant monetary cost reduction. However, MBS is still 2 to 3 times cheaper than BATCH since it batches requests in such a way that padding overhead (and the associated cost) is minimized.

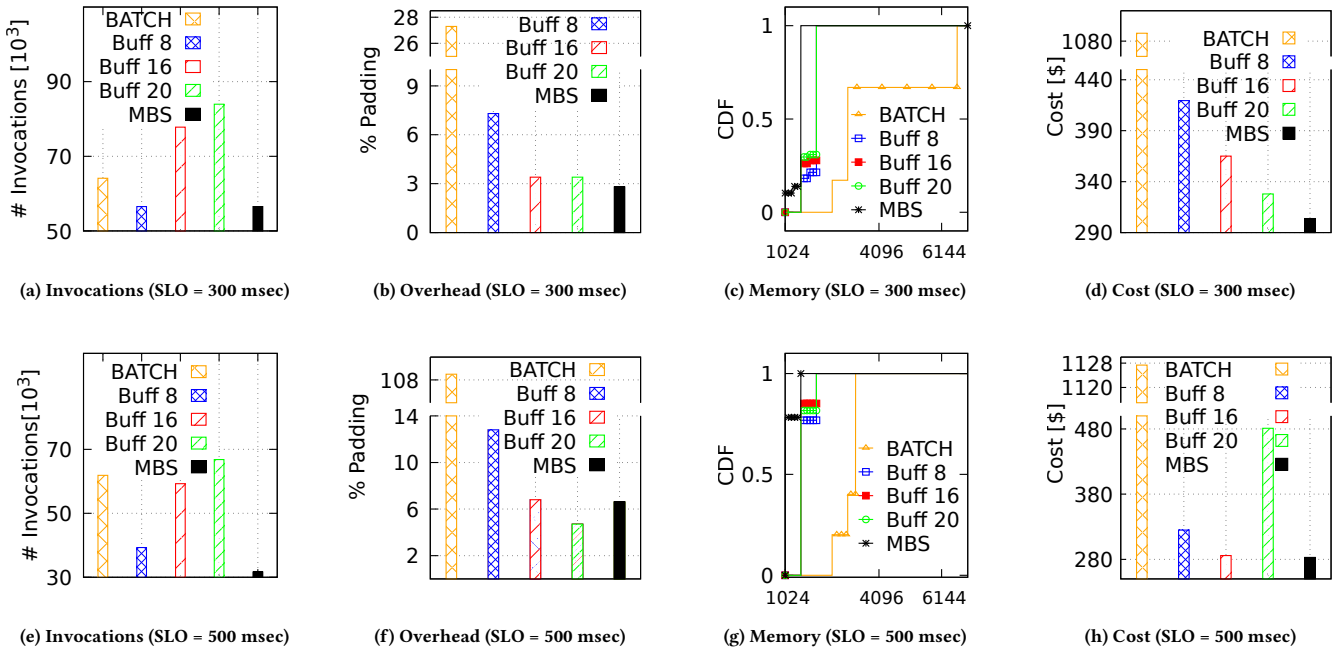
4.6 MBS vs. Exhaustive Search

We evaluate how effectively MBS identifies the system configuration that minimizes the monetary cost and meets the given SLO. For this purpose, we compare the performance of MBS with the one obtained using an Exhaustive Search approach (i.e., the best system configuration is selected among all possible ones). For this purpose, we assume the Exhaustive Search approach has full knowledge of workload intensity and request size distribution. This way, the Exhaustive Search approach always returns the system configuration that minimizes the monetary cost of the system and meets the SLO.



(a) Synthetic Workload (b) Latency (SLO = 300 msec) (c) Latency (SLO = 500 msec) (d) Buffers

Figure 9: Synthetic trace (a), the request latency provided by MBS and baseline frameworks when *SLO=300* msec (b) and *SLO=500* msec (c), and the number of buffers used by MBS for both SLOs (d).



(a) Invocations (SLO = 300 msec) (b) Overhead (SLO = 300 msec) (c) Memory (SLO = 300 msec) (d) Cost (SLO = 300 msec)

(e) Invocations (SLO = 500 msec) (f) Overhead (SLO = 500 msec) (g) Memory (SLO = 500 msec) (h) Cost (SLO = 500 msec)

Figure 10: Performance of MBS and baseline policies to serve ML inference requests whose arrival process is subject to large and sudden variations. The SLO is set to 300 and 500 msec.

Figure 12 depicts the request latency distribution and the average monetary cost of MBS and the Exhaustive Search approach when they are used to control the workload in Figure 9(a) with different SLOs. MBS generally serves ML inference requests with the same (or very similar) latency of the Exhaustive Search approach, see Figures 12(a) and 12(b) for $SLO = 300$ msec and $SLO = 500$ msec, respectively. In some cases (less than 5%), MBS serves requests faster than Exhaustive Search due to MBS having no prior knowledge of the workload intensity and the request size distribution. The monetary cost of MBS and Exhaustive Search is comparable (i.e., the difference is less than 2% for both SLOs), see Figure 12(c).

Overall, MBS selects (close to) optimal system configurations and provides results (i.e., request latency and monetary cost) that are similar to the one of the Exhaustive Search approach. Thanks to Bayesian Optimization, MBS detects the best system configuration in a hundredth of the time taken by the Exhaustive Search approach.

5 RELATED WORK

Despite some limitations of the FaaS paradigm [2, 9, 40] (e.g., the slow exchange of data between functions which leverage remote storage [37, 45]), serverless computing has been recently adopted for running applications such as live streaming [58], video processing [6], data processing [51, 55, 78], and IoT services [23]. Among others, also machine learning applications have been deployed and evaluated on serverless platforms [35] and recent work [41] aims to facilitate the deployment of ML models on FaaS for users with different expertise (e.g., statisticians and data scientists).

Training ML models (no inference) on serverless computing platforms presents many challenges and opportunities [28]. Carreira et al. [15] analyze the feasibility of ML model training on serverless platforms and propose a general framework architecture to tackle the most immediate challenges. They also develop CIRRUS [16], a

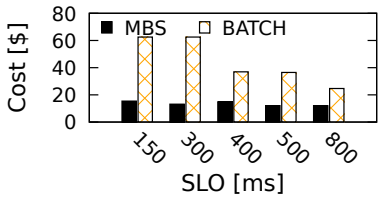
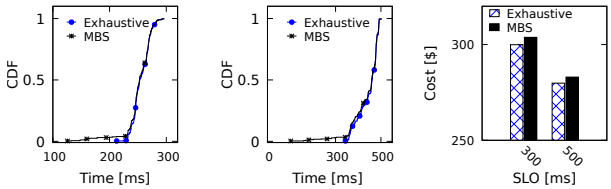


Figure 11: Monetary cost against different SLO values. The arrival process is driven by the Twitter trace and the request size distribution follows the Bi-modal function.



(a) Latency (SLO = 300 msec) **(b) Latency (SLO = 500 msec)** **(c) Average Cost**

Figure 12: Request latency CDF of the Exhaustive Search approach and MBS, when *SLO=300* msec (a) and *SLO=500* msec (b). Average monetary cost of the two strategies (c).

framework that efficiently supports ML training and hyperparameter optimization. Wang et al. [77] propose SIREN, a framework that reduces ML model training up to 44% when compared to traditional benchmarks executed on AWS EC2. Jiang et al. [37] proposes LambdaML, a framework of ML training on serverless computing along with an analytical model to evaluate the performance trade-off of FaaS- and IaaS-based ML training. Differently from these approaches, MBS aims to optimize the processing of heterogeneous ML inference workloads on serverless platforms.

Elordi et al. [26] use MLPerf to benchmark deep neural network inference on AWS Lambda. BARISTA [13] enables horizontal and vertical scaling of serverless resources when they are employed to process ML inference requests with a bursty arrival process. The performance of BARISTA is not analyzed on public serverless systems. Zhang et. al [81] develop MArk, a framework that decreases the cost of ML model inference by flanking IaaS instances with serverless computing. MArk uses serverless resources to comply with SLOs when workload variations are detected. However, it does not promptly react to sudden workload changes that result in longer latency tails. Ali et al. [3] propose BATCH to process bursty ML inference requests. Their framework leverage serverless computing and dynamic batching, but does not support multiclass workloads. Jarachanthan et al. [36] implement AMPS-Inf, a framework that enables model partitioning to increase the cost efficiency of model inference in serverless computing. However, AMPS-Inf does not provide any guarantee on tail latency and is not evaluated against bursty workloads. Gao et al. [30] propose a white box approach called cellular batching for serving heterogeneous requests. To minimize the padding overhead cellular batching makes the batching decisions at the granularity of an RNN cell. Compared to MBS, this approach does not take into consideration the dynamic variations in the workload arrival intensity and request size distribution and is SLO oblivious. In addition, cellular batching requires modifications within existing ML frameworks. Similarly, other approaches have been proposed in [27, 34, 72] to improve the energy efficiency or system utilization. However, these approaches either require modification of the ML serving framework or do not take into consideration the variation in workload intensity.

Different applications batch and process requests together to improve the system performance, e.g., OLTP systems [25] and in-memory machine learning [70]. Stout [46] uses dynamic batching to improve the throughput of cloud storage applications, but it only works with performance average values (no distribution) and does not allow users to define SLOs. Crankshaw et al. [24] adopt dynamic batching for improving the performance of ML inference requests and implement it in Clipper, a prediction service system that reduces ML inference latency. The exhaustive profiling strategy used by Clipper to find optimal parameters (i.e., the batch size) makes the framework unsuitable for serverless platforms whose workload experiences sudden variations. Moreover, Clipper does not allow controlling the memory size of serverless functions, and its reactive nature makes it unable to meet user-defined SLOs. Dynamic batching is also implemented in GrandSLAm [38], a framework to process microservice requests. This tool copes only with Poisson distributed inter-arrival times and does not serve bursty workloads.

All main deep learning frameworks (e.g., TensorFlow [1], MXNet [21], PyTorch [54]) support padding of requests with different size that are batched together [30]. Pinheiro et al. [56] propose a dynamic padding strategy that adapts to the usage of smart home networks. When low traffic is detected, padding is increased to maximize the request privacy. Padding is decreased when high traffic is observed to reduce the overhead. MBS implements a padding strategy that accounts for the largest request in each batch and increases the size of smaller requests accordingly. This reduces the padding overhead since smaller requests are padded based on the largest request in the batch, not the largest one in the whole system.

6 CONCLUDING REMARKS

We introduce MBS, a framework that leverages analytical models and Bayesian optimization to unburden the thorny tasks of manual tuning serverless functions to process heterogeneous ML inference workloads. MBS observes the system workload to detect the optimal batching and system configurations that minimize the monetary cost while preserving SLO. The performance of MBS is evaluated against state-of-the-art approaches using real (bursty) traces. Results show that MBS predicts the request latency distribution with high accuracy (maximum error smaller than 10%) regardless of load intensity and request heterogeneity (i.e., request size distribution). Compared to existing approaches, MBS preserves SLO while reducing the monetary cost by up to 8 × in public serverless platforms.

ACKNOWLEDGMENTS

This work is supported by the following grants: National Science Foundation CAREER-2048044, IIS-1838024 (using resources provided by Amazon Web Services as part of the NSF BIGDATA program), IIS-1838022, CCF-1717532, CNS-1950485, and MIUR PRIN project SEDUCE 2017TWRCNB. We thank the anonymous reviewers for their insightful comments that improved the paper.

REFERENCES

[1] Martín Abadi, Paul Barham, Jianmin Chen, Zhifeng Chen, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Geoffrey Irving, Michael Isard, Manjunath Kudlur, Josh Levenberg, Rajat Monga, Sherry Moore, Derek Gordon Murray, Benoit Steiner, Paul A. Tucker, Vijay Vasudevan, Pete Warden, Martin Wicke, Yuan Yu, and Xiaoqiang Zheng. 2016. Tensorflow: A system for large-scale machine learning. In *Proceedings of the Symposium on Operating Systems Design and Implementation (OSDI)*. USENIX, 265–283.

[2] Adil Akhter, Marios Fragkoulis, and Asterios Katsifodimos. 2019. Stateful functions as a service in action. *Proceedings of the VLDB Endowment* 12, 12 (2019), 1890–1893.

[3] Ahsan Ali, Riccardo Pinciroli, Feng Yan, and Evgenia Smirni. 2020. Batch: machine learning inference serving on serverless platforms with adaptive batching. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis (SC)*. IEEE, 1–15.

[4] Ahsan Ali, Hemant Sharma, Rajkumar Kettimuthu, Peter Kenesei, Dennis Trujillo, Antonino Miceli, Ian Foster, Ryan Coffee, Jana Thayer, and Zhengchun Liu. 2022. fairDMS: Rapid Model Training by Data and Model Reuse. *arXiv preprint arXiv:2204.09805* (2022).

[5] Ahsan Ali, Syed Zawad, Paarijaat Aditya, Istemi Ekin Akkus, Ruichuan Chen, and Feng Yan. 2022. SMLT: A Serverless Framework for Scalable and Adaptive Machine Learning Design and Training. *arXiv preprint arXiv:2205.01853* (2022).

[6] Lixiang Ao, Liz Izhikevich, Geoffrey M. Voelker, and George Porter. 2018. Sprocket: A serverless video processing framework. In *Proceedings of the Symposium on Cloud Computing (SoCC)*. ACM, 263–274.

[7] Tayebeh Bahreini, Hossein Badri, and Daniel Grosu. 2021. Mechanisms for resource allocation and pricing in mobile edge computing systems. *IEEE Transactions on Parallel and Distributed Systems* 33, 3 (2021), 667–682.

[8] Ioana Baldini, Paul C. Castro, Kerry Shih-Ping Chang, Perry Cheng, Stephen Fink, Vatche Ishakian, Nick Mitchell, Vinod Muthusamy, Rodric Rabbah, Aleksander Slominski, and Philippe Suter. 2017. Serverless computing: Current trends and open problems. In *Research Advances in Cloud Computing*. Springer, 1–20.

[9] Daniel Barcelona-Pons, Marc Sánchez-Artigas, Gerard París, Pierre Sutra, and Pedro García-López. 2019. On the faas track: Building stateful distributed applications with serverless architectures. In *Proceedings of the International Middleware Conference (Middleware)*. ACM, 41–54.

[10] Falko Bause, Peter Buchholz, and Jan Kriege. 2009. A comparison of Markovian arrival and ARMA/ARTA processes for the modeling of correlated input processes. In *Proceedings of the Winter Simulation Conference (WSC)*. IEEE, 634–645.

[11] Syrine Belakaria, Aryan Deshwal, Nitthilan Kannappan Jayakodi, and Janardhan Rao Doppa. 2020. Uncertainty-aware search framework for multi-objective Bayesian optimization. In *Proceedings of the AAAI Conference on Artificial Intelligence (AAAI)*, Vol. 34. AAAI Press, 10044–10052.

[12] Julian Berk, Sunil Gupta, Santu Rana, and Svetha Venkatesh. 2020. Randomised Gaussian Process Upper Confidence Bound for Bayesian Optimisation. In *Proceedings of the International Joint Conference on Artificial Intelligence (IJCAI)*. IJCAI, 2284–2290.

[13] Anirban Bhattacharjee, Ajay Dev Chhokra, Zhuangwei Kang, Hongyang Sun, Aniruddha Gokhale, and Gabor Karsai. 2019. BARISTA: Efficient and Scalable Serverless Serving System for Deep Learning Prediction Services. In *Proceedings of the International Conference on Cloud Engineering (IC2E)*. IEEE, 23–33.

[14] Eric Brochu, Vlad M. Cora, and Nando De Freitas. 2010. A tutorial on Bayesian optimization of expensive cost functions, with application to active user modeling and hierarchical reinforcement learning. *arXiv preprint arXiv:1012.2599* (2010).

[15] Joao Carreira, Pedro Fonseca, Alexey Tumanov, Andrew Zhang, and Randy Katz. 2018. A Case for Serverless Machine Learning. In *Workshop on Systems for ML and Open Source Software at NeurIPS*.

[16] Joao Carreira, Pedro Fonseca, Alexey Tumanov, Andrew Zhang, and Randy Katz. 2019. Cirrus: a Serverless Framework for End-to-end ML Workflows. In *Proceedings of the Symposium on Cloud Computing (SoCC)*. ACM, 13–24.

[17] Giuliano Casale, Ningfang Mi, Ludmila Cherkasova, and Evgenia Smirni. 2008. How to parameterize models with bursty workloads. *SIGMETRICS Performance Evaluation Review* 36, 2 (2008), 38–44.

[18] Giuliano Casale, Ningfang Mi, and Evgenia Smirni. 2010. Model-Driven System Capacity Planning under Workload Burstiness. *IEEE Transactions on Computers* 59, 1 (2010), 66–80.

[19] Giuliano Casale, Eddy Z. Zhang, and Evgenia Smirni. 2010. KPC-Toolbox: Best recipes for automatic trace fitting using Markovian Arrival Processes. *Performance Evaluation* 67, 9 (2010), 873–896.

[20] Giuliano Casale, Eddy Z. Zhang, and Evgenia Smirni. 2010. Trace data characterization and fitting for Markov modeling. *Performance Evaluation* 67, 2 (2010), 61–79.

[21] Tianqi Chen, Mu Li, Yutian Li, Min Lin, Naiyan Wang, Minjie Wang, Tianjun Xiao, Bing Xu, Chiyuan Zhang, and Zheng Zhang. 2015. Mxnet: A flexible and efficient machine learning library for heterogeneous distributed systems. *arXiv preprint arXiv:1512.01274* (2015).

[22] Zhengdao Chen, Xiang Li, and Joan Bruna. 2017. Supervised community detection with line graph neural networks. *arXiv preprint arXiv:1705.08415* (2017).

[23] Bin Cheng, Jonathan Fuerst, Gurkan Solmaz, and Takuya Sanada. 2019. Fog Function: Serverless Fog Computing for Data Intensive IoT Services. In *Proceedings of the International Conference on Services Computing (SCC)*. IEEE, 28–35.

[24] 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 *Proceedings of the Symposium on Networked Systems Design and Implementation (NSDI)*. USENIX, 613–627.

[25] Bailu Ding, Lucja Kot, and Johannes Gehrke. 2018. Improving optimistic concurrency control through transaction batching and operation reordering. *Proceedings of the VLDB Endowment* 12, 2 (2018), 169–182.

[26] Unai Elordi, Luis Unzueta, Jon Goenetxea, Sergio Sanchez-Carballido, Ignacio Arganda-Carreras, and Oihana Otaegui. 2020. Benchmarking deep neural network inference performance on serverless environments with MLPerf. *IEEE Software* 38, 1 (2020), 81–87.

[27] Jiarui Fang, Yang Yu, Chengduo Zhao, and Jie Zhou. 2021. TurboTransformers: an efficient GPU serving system for transformer models. In *Proceedings of the 26th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*. 389–402.

[28] Lang Feng, Prabhakar Kudva, Dilma Da Silva, and Jiang Hu. 2018. Exploring serverless computing for neural network training. In *Proceedings of the International Conference on Cloud Computing (CLOUD)*. IEEE, 334–341.

[29] Peter I. Frazier. 2018. Bayesian optimization. In *Recent Advances in Optimization and Modeling of Contemporary Problems*. INFORMS, 255–278.

[30] Pin Gao, Lingfan Yu, Yongwei Wu, and Jinyang Li. 2018. Low latency RNN inference with cellular batching. In *Proceedings of the EuroSys Conference (EuroSys)*. ACM, 31:1–31:15.

[31] 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 *Proceedings of the International Middleware Conference (Middleware)*. ACM, 109–120.

[32] Jashwant Raj Gunasekaran, Prashanth Thinakaran, Mahmut Taylan Kandemir, Bhuvan Urgaonkar, George Kesidis, and Chita Das. 2019. Spock: Exploiting serverless functions for slo and cost aware resource procurement in public cloud. In *Proceedings of the International Conference on Cloud Computing (CLOUD)*. IEEE, 199–208.

[33] Luis Felipe Herrera-Quintero, Julian Camilo Vega-Alfonso, Klaus Bodo Albert Banse, and Eduardo Carrillo Zambrano. 2018. Smart ITS sensor for the transportation planning based on IoT approaches using serverless and microservices architecture. *IEEE Intelligent Transportation Systems Magazine* 10, 2 (2018), 17–27.

[34] Connor Holmes, Daniel Mawhirter, Yuxiong He, Feng Yan, and Bo Wu. 2019. Grnn: Low-latency and scalable rnn inference on gpus. In *Proceedings of the Fourteenth EuroSys Conference 2019*. 1–16.

[35] Vatche Ishakian, Vinod Muthusamy, and Aleksander Slominski. 2018. Serving deep learning models in a serverless platform. In *Proceedings of the International Conference on Cloud Engineering (IC2E)*. IEEE, 257–262.

[36] Jananie Jarachanthan, Li Chen, Fei Xu, and Bo Li. 2021. AMPS-Inf: Automatic Model Partitioning for Serverless Inference with Cost Efficiency. In *Proceedings of the International Conference on Parallel Processing (ICPP)*. ACM, 1–12.

[37] 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 *Proceedings of the International Conference on Management of Data (SIGMOD)*. ACM, 857–871.

[38] Ram Srivatsa Kannan, Lavanya Subramanian, Ashwin Raju, Jeongseob Ahn, Jason Mars, and Lingjia Tang. 2019. GrandSLAm: Guaranteeing SLAs for Jobs in Microservices Execution Frameworks. In *Proceedings of the EuroSys Conference (EuroSys)*. ACM, 34:1–34:16.

[39] Thomas N Kipf and Max Welling. 2016. Semi-supervised classification with graph convolutional networks. *arXiv preprint arXiv:1609.02907* (2016).

[40] Ana Klimovic, Yawen Wang, Patrick Stuedi, Animesh Trivedi, Jonas Pfefferle, and Christos Kozyrakis. 2018. Pocket: Elastic Ephemeral Storage for Serverless Analytics. In *Proceedings of the Symposium on Operating Systems Design and Implementation (OSDI)*. USENIX, 427–444.

[41] Malte S. Kurz. 2021. Distributed Double Machine Learning with a Serverless Architecture. In *Proceedings of the International Conference on Performance Engineering (ICPE-C)*. ACM/SPEC, 27–33.

[42] Yujia Li, Oriol Vinyals, Chris Dyer, Razvan Pascanu, and Peter Battaglia. 2018. Learning deep generative models of graphs. *arXiv preprint arXiv:1803.03324* (2018).

[43] Zhengchun Liu, Ahsan Ali, Peter Kenesei, Antonino Miceli, Hemant Sharma, Nicholas Schwarz, Dennis Trujillo, Hyunseung Yoo, Ryan Coffee, Naoufal Layad, et al. 2021. Bridge data center AI systems with edge computing for actionable information retrieval. *arXiv preprint arXiv:2105.13967* (2021).

[44] Zhengchun Liu, Ahsan Ali, Peter Kenesei, Antonino Miceli, Hemant Sharma, Nicholas Schwarz, Dennis Trujillo, Hyunseung Yoo, Ryan Coffee, Naoufal Layad, et al. 2021. Bridging data center AI systems with edge computing for actionable information retrieval. In *2021 3rd Annual Workshop on Extreme-scale Experiment-in-the-Loop Computing (XLOOP)*. IEEE, 15–23.

[45] Ashraf Mahgoub, Karthick Shankar, Subrata Mitra, Ana Klimovic, Somali Chaterji, and Saurabh Bagchi. 2021. SONIC: Application-aware Data Passing for Chained Serverless Applications. In *Proceedings of the Annual Technical Conference (ATC)*. USENIX, 285–301.

[46] John C. McCullough, John Dunagan, Alec Wolman, and Alex C. Snoeren. 2010. Stout: An adaptive interface to scalable cloud storage. In *Proceedings of the Annual Technical Conference (ATC)*. USENIX, 47–60.

[47] Nicholas Metropolis and Stanislaw Ulam. 1949. The Monte Carlo Method. *Journal of the American Statistical Association* 44, 247 (1949), 335–341.

[48] Ningfang Mi, Giuliano Casale, Ludmila Cherkasova, and Evgenia Smirni. 2008. Burstiness in Multi-tier Applications: Symptoms, Causes, and New Models. In *Proceedings of the International Middleware Conference (Middleware)*. ACM, 265–286.

[49] Ningfang Mi, Qi Zhang, Alma Riska, Evgenia Smirni, and Erik Riedel. 2007. Performance impacts of autocorrelated flows in multi-tiered systems. *Performance Evaluation* 64, 9-12 (2007), 1082–1101.

[50] Microsoft. 2022. Azure. Create the games that you would play. https://azure.microsoft.com/en-us/. [Online; accessed 04-December-2019].

[51] Ingo Müller, Renato Marroquín, and Gustavo Alonso. 2020. Lambada: Interactive Data Analytics on Cold Data Using Serverless Cloud Infrastructure. In *Proceedings of the International Conference on Management of Data (SIGMOD)*. ACM, 115–130.

[52] Mahyar Najibi, Bharat Singh, and Larry Davis. 2019. AutoFocus: Efficient Multi-Scale Inference. In *Proceedings of the International Conference on Computer Vision (ICCV)*. IEEE, 9744–9754.

[53] Marcel F. Neuts. 1989. *Structured Stochastic Matrices of M/G/1 Type and Their Applications*. Vol. 5. Marcel Dekker New York.

[54] Adam Paszke, Sam Gross, Francisco Massa, Adam Lerer, James Bradbury, Gregory Chanan, Trevor Killeen, Zeming Lin, Natalia Gimelshein, Luca Antiga, Alban Desmaison, Andreas Köpf, Edward Z. Yang, Zachary DeVito, Martin Raison, Alykhan Tejani, Sasank Chilamkurthy, Benoit Steiner, Lu Fang, Junjie Bai, and Soumith Chintala. 2019. PyTorch: An Imperative Style, High-Performance Deep Learning Library. In *Proceedings of the Conference on Neural Information Processing Systems (NeurIPS)*. 8024–8035.

[55] Alfonso Pérez, Sebastián Risco, Diana María Naranjo, Miguel Caballer, and Germán Moltó. 2019. On-premises Serverless Computing for Event-Driven Data Processing Applications. In *Proceedings of the International Conference on Cloud Computing (CLOUD)*. IEEE, 414–421.

[56] Antônio J. Pinheiro, Paulo Freitas de Araujo-Filho, Jeandro de M. Bezerra, and Divanilson R. Campelo. 2021. Adaptive Packet Padding Approach for Smart Home Networks: A Trade-off between Privacy and Performance. *IEEE Internet of Things Journal* 8, 5 (2021), 3930–3938.

[57] Jonathan Ponader, Kyle Thomas, Sandip Kundu, and Yan Solihin. 2021. MILR: Mathematically induced layer recovery for plaintext space error correction of CNNs. In *Proceedings of the International Conference on Dependable Systems and Networks (DSN)*. IEEE, 75–87.

[58] Jeff Rajewski. 2018. System and method for live streaming content to subscription audiences using a serverless computing system. US Patent Application No. 15/369,473.

[59] Carl Edward Rasmussen and Christopher K. I. Williams. 2006. *Gaussian processes for machine learning*. MIT Press.

[60] Amazon Web Services. 2022. Amazon SageMaker. https://aws.amazon.com/sagemaker/. [Online; accessed 04-April-2022].

[61] Amazon Web Services. 2022. AWS Lambda. https://aws.amazon.com/lambda/. [Online; accessed 04-April-2022].

[62] Amazon Web Services. 2022. AWS Lambda Pricing. https://aws.amazon.com/lambda/pricing/. [Online; accessed 04-April-2022].

[63] Amazon Web Services. 2022. Start Building on AWS Today. https://aws.amazon.com. [Online; accessed 04-April-2022].

[64] Archive Team. 2017. The Twitter Stream Grab. https://archive.org/details/twitterstream?and[]=year%3A%222017%22. [Online; accessed 04-April-2022].

[65] Google Cloud Platform. 2022. Cloud Functions. https://cloud.google.com/functions/. [Online; accessed 04-April-2022].

[66] Google Cloud Platform. 2022. Dream, build, and transform with Google Cloud. https://cloud.google.com/ [Online; accessed 04-April-2022].

[67] Microsoft. 2022. Azure Functions. https://azure.microsoft.com/en-us/services/functions/. [Online; accessed 04-April-2022].

[68] Alma Riska and Evgenia Smirni. 2002. MAMSolver: A Matrix Analytic Methods Tool. In *Proceedings of the International Conference on Modelling Techniques and Tools for Computer Performance Evaluation (TOOLS)*. Springer, 205–211.

[69] Anthony Rousseau, Paul Deléglise, Yannick Esteve, et al. 2014. Enhancing the TED-LIUM corpus with selected data for language modeling and more TED talks.. In *LREC*. 3935–3939.

[70] Maximilian Schleich and Dan Olteanu. 2020. LMFAO: An Engine for Batches of Group-By Aggregates. *Proceedings of the VLDB Endowment* 13, 12 (2020), 2945–2948.

[71] Michael Schlichtkrull, Thomas N Kipf, Peter Bloem, Rianne van den Berg, Ivan Titov, and Max Welling. 2018. Modeling relational data with graph convolutional networks. In *European semantic web conference*. Springer, 593–607.

[72] Franyell Silfa, Jose Maria Arnau, and Antonio Gonzalez. 2020. E-BATCH: Energy-Efficient and High-Throughput RNN Batching. *arXiv preprint arXiv:2009.10656* (2020).

[73] Jasper Snoek, Hugo Larochelle, and Ryan P. Adams. 2012. Practical Bayesian Optimization of Machine Learning Algorithms. In *Proceedings of the Conference on Neural Information Processing Systems (NeurIPS)*. 2960–2968.

[74] Niranjan Srinivas, Andreas Krause, Sham M. Kakade, and Matthias W. Seeger. 2010. Gaussian Process Optimization in the Bandit Setting: No Regret and Experimental Design. In *Proceedings of the International Conference on Machine Learning (ICML)*. Omnipress, 1015–1022.

[75] Rachael Tatman. 2017. Speech Accent Archive. https://www.kaggle.com/rtatman/speech-accent-archive. [Online; accessed 04-April-2022].

[76] Petar Veličković, Guillem Cucurull, Arantxa Casanova, Adriana Romero, Pietro Lio, and Yoshua Bengio. 2017. Graph attention networks. *arXiv preprint arXiv:1710.10903* (2017).

[77] Hao Wang, Di Niu, and Baochun Li. 2019. Distributed Machine Learning with a Serverless Architecture. In *Proceedings of the Conference on Computer Communications (INFOCOM)*. IEEE, 1288–1296.

[78] Chenggang Wu, Vikram Sreekanti, and Joseph M. Hellerstein. 2020. Transactional Causal Consistency for Serverless Computing. In *Proceedings of the International Conference on Management of Data (SIGMOD)*. ACM, 83–97.

[79] Ji Xue, Robert Birke, Lydia Y. Chen, and Evgenia Smirni. 2016. Managing Data Center Tickets: Prediction and Active Sizing. In *Proceedings of the International Conference on Dependable Systems and Networks (DSN)*. IEEE, 335–346.

[80] Feng Yan, Olatunji Ruwase, Yuxiong He, and Evgenia Smirni. 2016. SERF: efficient scheduling for fast deep neural network serving via judicious parallelism. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis (SC)*. IEEE, 300–311.

[81] Chengliang Zhang, Minchen Yu, Wei Wang, and Feng Yan. 2019. MArk: Exploiting Cloud Services for Cost-Effective, SLO-Aware Machine Learning Inference Serving. In *Proceedings of the Annual Technical Conference (ATC)*. USENIX.

[82] Hong Zhang, Lan Zhang, Lan Xu, Xiaoyang Ma, Zhengtao Wu, Cong Tang, Wei Xu, and Yiguo Yang. 2020. A Request-level Guaranteed Delivery Advertising Planning: Forecasting and Allocation. In *Proceedings of the Conference on Knowledge Discovery and Data Mining (SIGKDD)*. ACM, 2980–2988.

[83] Qi Zhang, Alma Riska, Wei Sun, Evgenia Smirni, and Gianfranco Ciardo. 2005. Workload-Aware Load Balancing for Clustered Web Servers. *IEEE Transactions on Parallel and Distributed Systems* 16, 3 (2005), 219–233.