HarmonyBatch: Batching multi-SLO DNN Inference with Heterogeneous Serverless Functions

Jiabin Chen†, Fei Xu†*, Yikun Gu†, Li Chen‡, Fangming Liu§, Zhi Zhou¶
†Shanghai Key Laboratory of Multidimensional Information Processing, East China Normal University.
‡University of Louisiana at Lafayette. §Peng Cheng Laboratory. ¶Sun Yat-sen University.
Email: †*fxu@cs.ecnu.edu.cn*, †*li.chen@louisiana.edu*, §*fangminghk@gmail.com*, ¶*zhouzhi9@mail.sysu.edu.cn*

***Abstract*—Deep Neural Network (DNN) inference on serverless functions is gaining prominence due to its potential for substantial budget savings. Existing works on serverless DNN inference solely optimize batching requests from one application with a *single* Service Level Objective (SLO) on CPU functions. However, production serverless DNN inference traces indicate that the request arrival rate of applications is *surprisingly low*, which inevitably causes a long batching time and SLO violations. Hence, there is an urgent need for batching *multiple* DNN inference requests with diverse SLOs (*i.e., multi-SLO* DNN inference) in serverless platforms. Moreover, the potential performance and cost benefits of deploying *heterogeneous* (*i.e.,* CPU and GPU) functions for DNN inference have received scant attention.**

In this paper, we present *HarmonyBatch*, a cost-efficient resource provisioning framework designed to achieve predictable performance for multi-SLO DNN inference with heterogeneous serverless functions. Specifically, we construct an analytical performance and cost model of DNN inference on both CPU and GPU functions, by explicitly considering the GPU time-slicing scheduling mechanism and request arrival rate distribution. Based on such a model, we devise a two-stage merging strategy in *HarmonyBatch* to judiciously batch the multi-SLO DNN inference requests into application *groups*. It aims to minimize the budget of function provisioning for each application group while guaranteeing diverse performance SLOs of inference applications. We have implemented a prototype of *HarmonyBatch* on Alibaba Cloud Function Compute. Extensive prototype experiments with representative DNN inference workloads demonstrate that *HarmonyBatch* can provide predictable performance to serverless DNN inference workloads while reducing the monetary cost by up to 82.9% compared to the state-of-the-art methods.

***Index Terms*—serverless computing, resource provisioning, DNN inference, SLO guarantee**

I. Introduction

The rapid evolution of artificial intelligence across diverse fields has elevated Deep Neural Network (DNN) inference to a critical role in cloud-based workloads. Fueled by the bursty nature of DNN inference requests, the fast elasticity of serverless computing makes it compelling for hosting such inference workloads, without heavy server maintenance [1]. As DNN models grow in complexity, especially the emergence of large language models (LLMs) [2], the computational and memory resource demands of serverless inference services increase sharply [3]. To meet the stringent Service Level Objectives (SLOs) of DNN inference workloads, Alibaba has recently introduced GPU serverless functions [4]. Such a development in GPU functions not only provides opportunities for cost reduction [5] but also brings new challenges in deploying DNN inference on heterogeneous serverless functions [6].

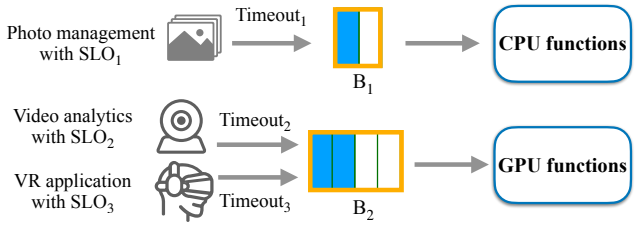


Fig. 1: A scenario of batching multiple DNN inference requests with diverse SLOs (*i.e.,* multi-SLO DNN inference) on heterogeneous serverless functions. $\texttt{Timeout}_i$ denotes the *batching timeout* for an application i. B_1 and B_2 denote the batch sizes of DNN inference executed on CPU and GPU functions, respectively.

Upon deploying a DNN model in public clouds, the mainstream serverless DNN inference service (*e.g.,* Amazon SageMaker Serverless Inference[1]) cannot provide SLO guarantees for users. Users only rely on their own experience to provision CPU/GPU function resources and configure the batch size for their inference workloads. While several recent studies guarantee SLOs for DNN inference by either minimizing the performance interference [7] or optimizing function resource provisioning plan [8], they mainly focus on the DNN inference scenario with a *single* SLO, which is impractical for the real-world situation of applications with low request arrival rates (typically less than one request per second), as evidenced in Sec. II-A. Such low request arrival rates inevitably cause a long batching time, which cannot meet the SLO requirements (typically in milliseconds) of DNN inference workloads with large batch sizes.

Fortunately, inference applications executed in public clouds can often be *grouped* and each group shares the same DNN model. Such application groups bring users an opportunity to aggregate multiple DNN inference requests with diverse SLOs into large batches. As an example, Hugging Face Inference Endpoints[2] provide the serverless inference API to user ap-

*Corresponding author: Fei Xu (fxu@cs.ecnu.edu.cn). This work was supported in part by the NSFC under Grant 62372184, the Science and Technology Commission of Shanghai Municipality under Grant 22DZ2229004, the NSF under Grants OIA-2019511 and OIA-2327452, the Louisiana Board of Regents under Contract LEQSF(2019-22)-RD-A-21, the National Key Research & Development (R&D) Plan under Grant 2022YFB4501703, and the Major Key Project of PCL (PCL2022A05).

[1]https://aws.amazon.com/sagemaker
[2]https://huggingface.co/inference-endpoints/serverless

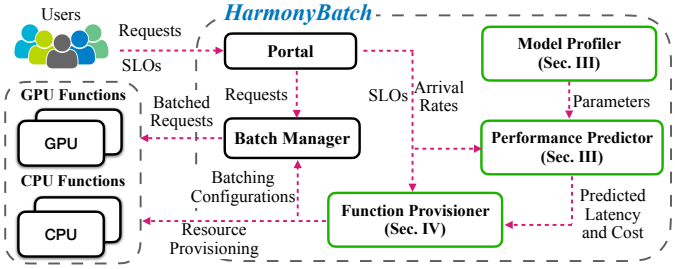


Fig. 2: Overview of *HarmonyBatch*.

plications with diverse SLO requirements. In such a scenario, we can batch the inference requests from video analytics with $\texttt{SLO}_2$ and VR smart assistants [9] with $\texttt{SLO}_3$ on GPU functions and thus obtain a lower monetary cost, as illustrated in Fig. 1. The rationale is that executing a large inference batch on GPU functions can achieve up to 37.0% of cost saving as compared to a small batch on CPU functions, as evidenced by Sec. II-B.

However, existing works (*e.g.,* BATCH [8], INFless [10]) have solely focused on batching requests from one application with a single SLO requirement and optimizing CPU function resources [11]. They cannot be readily applied to achieving multi-SLO serverless DNN inference, due to the two key challenges summarized below:

• ***Complex* batching for multi-SLO DNN inference.** State-of-the-art inference batching techniques *individually* choose a batch size and a batching timeout value for each application with a single SLO [8], [12]. However, it is complex to batch requests from a set of applications in the multi-SLO DNN inference scenario. This is because it requires categorizing the applications into *groups* cost-efficiently and determining the batching configuration (including the batch size for each group and a batching timeout for each application) without SLO violations. Moreover, it is difficult to estimate the monetary cost of DNN inference in such a multi-SLO scenario.

• ***Heterogeneous* function resource provisioning.** Significant efforts (*e.g.,* AWS Lambda Power Tuning [13], COSE [14]) have been devoted to provisioning homogeneous CPU functions for guaranteeing performance SLOs of applications, which overlook the potential performance and cost benefits of deploying *heterogeneous* functions. As the performance SLOs and request arrival rates of an inference application vary, the optimal provisioning plan can be *shifted* between CPU and GPU functions, leading to up to 83.0% of budget saving as elaborated in Sec. II-B. Moreover, the heterogeneous function provisioning can be even harder in the multi-SLO DNN inference scenario, which requires the co-optimization of the function resource allocation and application grouping as well as batching configuration.

To address these challenges above, we introduce *HarmonyBatch* as illustrated in Fig. 2, a cost-efficient function resource provisioning framework for achieving predictable DNN inference in serverless platforms. To the best of our knowledge, *HarmonyBatch* is the first work to provision heterogeneous (CPU and GPU) functions for judiciously batching multi-SLO DNN inference in public clouds. The main contributions of our paper are as follows:

▷ By obtaining model parameters through a lightweight workload profiling in the *model profiler* (Sec. III-A), we develop an analytical *performance predictor* (Sec. III) for DNN inference workloads on both CPU and GPU serverless functions. We explicitly consider the GPU time-slicing schedule mechanism and request arrival rate distribution.

▷ We design a cost-efficient inference resource provisioning strategy in the *function provisioner* (Sec. IV) for multi-SLO DNN inference. Our strategy uses a two-stage merging strategy to judiciously batch the multi-SLO DNN inference requests into application groups, with the aim of minimizing the budget of function provisioning for each application group while guaranteeing DNN inference performance SLOs.

▷ We implement a prototype of *HarmonyBatch* (https://github.com/icloud-ecnu/HarmonyBatch) with both CPU and GPU functions on Alibaba Cloud Function Compute [4]. Extensive prototype experiments with four representative DNN models (including LLMs) demonstrate that *HarmonyBatch* can achieve predictable performance for multi-SLO DNN inference, while saving the inference budget by up to 82.9%, compared to state-of-the-art methods (*e.g.,* BATCH [8], MBS [12]).

II. Background and Motivation

In this section, we first investigate the key factors that impact the performance and cost of serverless DNN inference. Then, we show that adequately batching requests onto *heterogeneous* functions can significantly save the user budget.

A. DNN Inference in Serverless Platforms

Deploying and batching DNN inference requests on serverless platforms can reduce the user budget [8]. However, the request arrival rate of applications is typically low (*i.e.,* less than 1 request per second), as analyzed by the average request rates of applications in the Microsoft Azure Functions trace [15] and Huawei Public Cloud trace [16]. Specifically, approximately 98.7% of applications in Microsoft Azure and 99.6% of applications in Huawei Cloud demonstrate a request rate of fewer than 1 request per second, as depicted in Fig. 3. As DNN inference workloads are typically latency-critical (in milliseconds), such low request arrival rates call for batching multi-SLO DNN inference requests on serverless platforms.

In general, there exist two types of serverless functions in mainstream public clouds. Specifically, CPU functions have finer resource allocation granularity (*i.e.,* 0.05 vCPU cores on Alibaba Cloud Function Compute [4]) and lower prices, while GPU functions are equipped with higher computing power (*i.e.,* 1 total GPU for a GPU function on Alibaba Cloud Function Compute [4]) and a larger amount of GPU memory. In such a case, it is *complex* for users to determine the appropriate CPU or GPU functions to be deployed together with the allocated function resources (*i.e.,* vCPU cores and GPU memory).

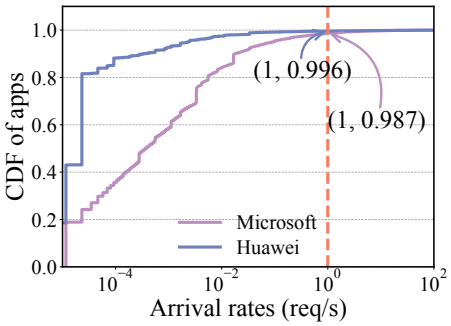


Fig. 3: CDF of request arrival rates per function in Microsoft Azure Functions trace [15] and Huawei Public Cloud trace [16].

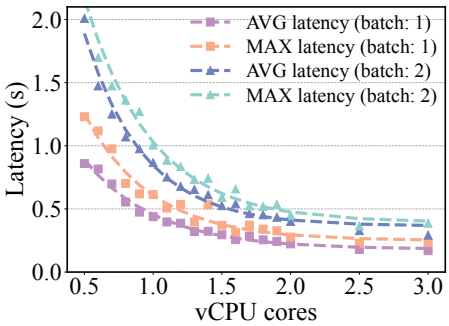


Fig. 4: Inference latency of VGG-19 executed on a CPU function by varying the allocated vCPU cores from 0.5 to 3.0.

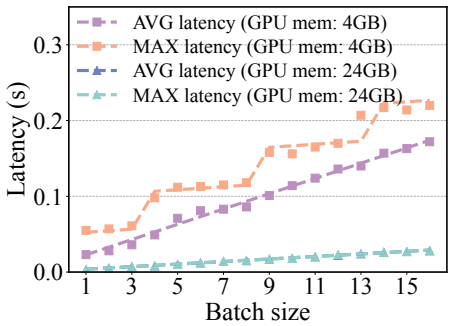


Fig. 5: Inference latency of VGG-19 executed on a GPU function by configuring the batch size from 1 to 16.

B. Characterizing Serverless Inference Performance and Cost

To explore the characteristics of serverless DNN inference performance (*i.e.,* latency) and monetary cost, we conduct motivation experiments of representative DNN models (*i.e.,* VGG-19 [17], BERT [18]) on CPU and GPU functions in Alibaba Cloud Function Compute [4]. We execute DNN inference workloads for 100 times with each function configuration in latency motivation experiments, while in monetary cost motivation experiments, we execute DNN inference for 10 minutes with each function configuration.

Latency. As shown in Fig. 4, the average and maximum inference latency of VGG-19 both show a *roughly exponential* decrease as more vCPU cores are allocated to CPU functions. This is because DNN inference workloads are commonly multi-core friendly, and the inference latency drops rapidly as we increase function resources at first. Our results indicate that provisioning more vCPU cores for CPU functions and increasing inference batch sizes can bring *marginal* performance benefits. As depicted in Fig. 5, the average and maximum inference latency on GPU functions *overlap* each other, both exhibiting a linear relationship with the batch size with the 24-GB configuration. Interestingly, when switching to the 4-GB configuration, the average inference latency is *roughly* linear to the batch size, while the maximum inference latency shows a *stepwise* increase with the batch size. We attribute such a result above to the time-slicing scheduling mechanism of GPU functions [19], which will be elaborated in Sec. III. In addition, there exists a noticeable difference between the average and maximum inference latency on both CPU and GPU functions, which indicates that DNN inference latency can be *unstable* due to the performance interference among functions [7], [20].

Cost. As illustrated in Fig. 6, the optimal function provisioning plans[3] under diverse SLOs ranging from stringent to relaxed turns out to GPU, CPU, and GPU functions. Under strict SLOs, CPU functions cannot meet the SLO requirements, while GPU functions with low batch sizes can meet such requirements. As SLOs become larger (more relaxed), CPU functions gain the advantage with their lower unit price at first, and then GPU functions become cost-efficient again, as longer batching time and requests with larger batch sizes are allowed. As shown in Fig. 7, the normalized cost decreases as the request arrival rate increases especially for GPU functions. This is because the batch size for inference gets large on GPU functions as the request arrival rate increases. The results indicate that batching requests from multiple applications for larger request arrival rates can bring significant cost benefits. In addition, the cost of DNN inference for CPU functions is insensitive to the changes in SLOs or request arrival rates.

[3]The optimal resource provisioning plan is obtained by iterating through all possible configurations via the grid search method.

C. An Illustrative Example

Batching multiple DNN inference requests from applications with diverse SLOs is challenging. In response, we design *HarmonyBatch* to identify cost-effective function resource provisioning for the predictable performance of DNN inference workloads in serverless platforms. As an example, we conduct motivation experiments using three applications (*i.e.,* App_1, App_2, and App_3) sharing the same model VGG-19 [17], but with different SLOs and arrival rates. Specifically, the performance SLOs of App_1, App_2, and App_3 are 0.5, 0.8, and 1 seconds, respectively. The request arrival rates of App_1, App_2, and App_3 are 5, 10, and 20 req/s, respectively.

As listed in Table I, *HarmonyBatch* can significantly reduce the function resource provisioning cost by up to 37.0% compared to the baselines. Specifically, the BATCH strategy [8] achieves the highest cost with several SLO violations. This is because it only batches requests from one application and simply assumes the inference latency following a deterministic distribution (*i.e.,* stable values). Though we modify the MBS^+ strategy to support heterogeneous function provisioning, it still brings a higher inference monetary cost compared to *HarmonyBatch*. This is because the MBS^+ strategy divides the requests from the three applications evenly, which aggregates App_1, App_2 and part of App_3 on a CPU function with the batch size set as 1. The remaining requests of App_3 are routed to a GPU function with a moderate batch size set as 11. In contrast, *HarmonyBatch* provisions three applications with heterogeneous functions, by adequately aggregating App_2 and App_3 as a large inference batch set as 13 on a GPU function.

Summary. *First*, judiciously batching *multi-SLO* inference requests can effectively save monetary cost while guaranteeing performance SLOs for DNN inference workloads. *Second*, provisioning *heterogeneous* functions for DNN inference ap-

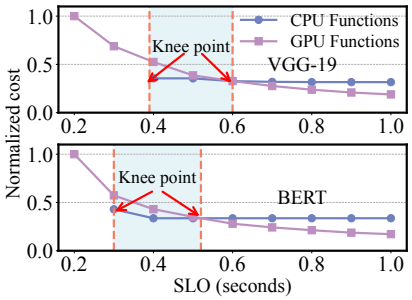


Fig. 6: Normalized cost of optimal function provisioning plan under different SLOs with the request arrival rate set as 20 req/s. The blue area denotes the optimal plan is CPU functions and two *knee* points exist.

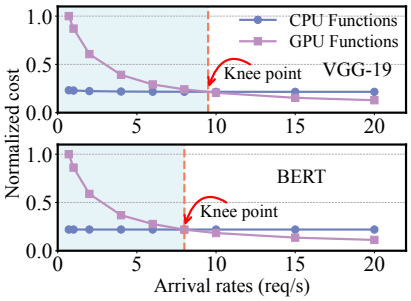


Fig. 7: Normalized cost of optimal function provisioning plan under different request arrival rates with the SLO set as 1 second. The blue area denotes the optimal plan is CPU functions and one *knee* point exists.

Strategies	Provisioning plans	Norm. cost
BATCH	$(1.55, 1, [0])_c, (1.5, 2, [0.48])_c, (1.5, 2, [0.48])_c$	1.00
MBS$^+$	$(1.6, 1, [0, 0, 0])_c$, $(2, 11, [0.65])_g$	0.88
HarmonyBatch	$(1.6, 1, [0])_c$, $(2, 13, [0.45, 0.65])_g$	**0.63**

TABLE I: Comparison of the normalized cost of VGG-19 with BATCH [8], MBS$^+$ [12], and *HarmonyBatch* strategies. The function provisioning plan is represented as a 3-tuple value: $(\texttt{vCPU cores}, \texttt{batch}, [\texttt{timeouts}])_c$ for CPU functions and $(\texttt{GPU memory}, \texttt{batch}, [\texttt{timeouts}])_g$ for GPU functions.

TABLE II: Key notations in our DNN inference latency and cost model in serverless platforms.

Notation	Definition
L^t_{avg}, L^t_{max}	Average, maximum inference latency of an inference workload on functions with a type $t = c, g$
L^g_0	Inference latency on a GPU function configured with the maximum GPU memory size M_{max}
$\alpha_b^{avg}, \beta_b^{avg}, \gamma_b^{avg}$	Model coefficients of inference average latency on CPU functions with the batch size set as b
$\alpha_b^{max}, \beta_b^{max}, \gamma_b^{max}$	Model coefficients of inference maximum latency on CPU functions with the batch size set as b
ξ_1, ξ_2	Model coefficients of inference latency on GPU functions
c, m	vCPU cores of provisioned CPU functions, GPU memory of provisioned GPU functions
$C^{\mathcal{X}}$	Average monetary cost of an inference request of the application group $\mathcal{X}$
K_1, K_2, K_3	Unit cost of a vCPU core, GPU memory, and a function invocation
$r^{\mathcal{X}}$, $b^{\mathcal{X}}$, $T^{\mathcal{X}}$	Total request arrival rate, batch size, equivalent batching timeout of the application group $\mathcal{X}$

plications can yield significant (up to 37.0%) cost benefits compared to homogeneous provisioning solutions. In particular, CPU functions are cost-effective for inference applications with *moderate* SLOs and *low* request arrival rates, while GPU functions are cost-effective for inference applications with *tight or loose* SLOs and *high* request arrival rates.

III. System Model

In this section, we model the inference latency with the CPU and GPU functions and leverage the request arrival rate to model the inference cost. The key notations in our serverless DNN inference model are summarized in Table II.

A. Modeling Latency of Serverless Inference

We focus on two key metrics including the average inference latency and maximum inference latency. The former is used to calculate the monetary cost of inference, while the latter is used to evaluate SLO violations.

CPU functions. As elaborated in Sec. II-B, the average inference latency on CPU functions L^c_{avg} decreases exponentially as more vCPU cores are provisioned. The average latency for batch size b on CPU functions can be given by

$$L^c_{avg} = \alpha_b^{avg} \cdot \exp\left(-\frac{c}{\beta_b^{avg}}\right) + \gamma_b^{avg}, \tag{1}$$

where $\alpha_b^{avg}, \beta_b^{avg}, \gamma_b^{avg}$ are the model coefficients with the batch size set as b, and the variable c denotes the allocated vCPU cores of functions. We apply a similar method to model the maximum inference latency on CPU functions L^c_{max} by leveraging the model coefficients $\alpha_b^{max}, \beta_b^{max}$ and γ_b^{max}.

GPU functions. When a GPU function is provisioned with the maximum GPU memory of M_{max} (*e.g.,* 24 GB for an NVIDIA A10 GPU), the GPU function exclusively occupies a whole GPU device, leading to a *stable* inference latency. As evidenced in Sec. II-B, the average and maximum latency overlap each other with the function GPU memory set as M_{max}. In addition, the inference latency on GPU functions with M_{max} is roughly linear to the batch size b. Accordingly, we formulate the inference latency L^g_0 on GPU functions with M_{max} as

$$L^g_0 = \xi_1 \cdot b + \xi_2, \tag{2}$$

where ξ_1 and ξ_2 are model coefficients for GPU functions.

We further model the average inference latency L^g_{avg} and maximum inference latency L^g_{max} on GPU functions. To facilitate serverless GPU functions, Alibaba Cloud Function Compute [4] deploys GPU temporal sharing mechanism (*i.e.,* cGPU [19]). Specifically, cGPU partitions the GPU's computing power into M_{max} units with each lasting for a duration of τ. It combines multiple time slices into a larger time slice $m \cdot \tau$ which is assigned to a GPU function with the GPU memory set as m. Though the latency of an inference request on GPU functions can be influenced by its arrival time, the average inference latency is still roughly linear to L^g_0 (in terms of the batch size), which is estimated as

$$L^g_{avg} = \frac{M_{max}}{m} \cdot L^g_0. \tag{3}$$

Furthermore, by assuming an inference request demands $2m \cdot \tau$ to complete the execution, the maximum and minimum inference latency can be obtained as $2M_{max} \cdot \tau$ and $(M_{max} + m) \cdot \tau$, respectively, as shown in Fig. 8. Accordingly, as for a request that demands L^g_0 to complete the execution and arrives at the

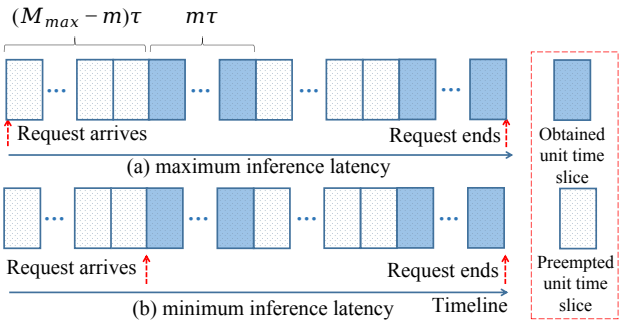


Fig. 8: Maximum and minimum inference latency scenarios caused by GPU time-slicing scheduling mechanism [19] on a GPU function with the GPU memory set as m. It hosts an inference request that demands $2m \cdot \tau$ time slices, where τ denotes a unit GPU time slice. (a) The request arrives at the beginning of a *preempted* time slice, resulting in the maximum inference latency $2M_{max} \cdot \tau$; and (b) the request arrives at the start of the *obtained* time slice, leading to the minimum inference latency $(M_{max} + m) \cdot \tau$.

start of the preempted time slice, it requires undergoing an additional number (*i.e.,* $\lceil \frac{L_0^g}{m \cdot \tau} \rceil$) of preempted time slices. As a result, we formulate the maximum inference latency on GPU functions as

$$L_{max}^g = \left\lceil \frac{L_0^g}{m \cdot \tau} \right\rceil \cdot (M_{max} - m) \cdot \tau + L_0^g. \tag{4}$$

Model coefficients acquisition. Based on our model above, we have six workload-specific coefficients (*i.e.,* $\alpha_b^{avg}, \beta_b^{avg}, \gamma_b^{avg}, \alpha_b^{max}, \beta_b^{max}, \gamma_b^{max}$) for CPU functions, two workload-specific coefficients (*i.e.,* ξ_1, ξ_2) and one hardware-specific parameter (*i.e.,* τ) for GPU functions. To determine the six coefficients for CPU functions, we execute DNN inference workloads 100 times on each configuration by varying function vCPU cores and batch sizes. As evidenced by Sec. II-B, CPU functions typically outperform GPU functions with small batch sizes. To obtain model coefficients, we only profile DNN inference workloads with small batch sizes (*i.e.,* ranging from 1 to 4), which significantly reduces the profiling overhead. To obtain the workload-specific coefficients for GPU functions, we configure the GPU function with the GPU memory set as M_{max} and execute DNN inference workloads only three times with two different batch sizes, as the inference latency on GPUs is stable. To identify τ, we get the L_{max}^g and L_0^g by running VGG-19 inference on a GPU function with a suitable GPU memory m and M_{max} for 100 times, respectively.

B. Modeling Monetary Cost of Multi-SLO DNN Inference

In a scenario with multiple applications (*i.e.,* a group $\mathcal{X}$), we assume that an inference application App$_i$ in the group $\mathcal{X}$ follows the Poisson distribution with the request arrival rate r_i. The application requests are first cached in a buffer with the capacity of $b^{\mathcal{X}}$ for batching. We set a batching timeout T_i for each application to avoid long request waiting in the buffer. Once any T_i expires, the cached requests (in a batch) are then sent to the functions for inference.

To execute the inference with the maximum batch size (*i.e.,* $b^{\mathcal{X}}$), the prerequisite $\lfloor r^{\mathcal{X}} \cdot T^{\mathcal{X}} \rfloor + 1 \geq b^{\mathcal{X}}$ needs to be held, where $\lfloor r^{\mathcal{X}} \cdot T^{\mathcal{X}} \rfloor + 1$ denotes the total number of requests received over a period $T^{\mathcal{X}}$, including the first request. In more detail, $r^{\mathcal{X}}$ is the total request arrival rate, and $T^{\mathcal{X}}$ is the *equivalent batching timeout* of group $\mathcal{X}$, which is considered as the *expectation value* of request waiting time in the buffer, as each application has its own batching timeout. To illustrate that, we start from two applications with their request arrival rates and batching timeouts (*i.e.,* App$_1$ with r_1, T_1 and App$_2$ with r_2, T_2). By assuming that T_1 is smaller than T_2, the equivalent batching timeout of $\mathcal{X}$ can be calculated by

$$T^{\mathcal{X}} = T_1 + \eta_2 \cdot \frac{1 - \exp\big(-r_1 \cdot (T_2 - T_1)\big)}{r_1}, \tag{5}$$

where $\eta_2 = \frac{r_2}{r_1 + r_2}$ denote the probability of the first request from App$_2$. The derivation can be found in Appendix A [21]. To obtain the equivalent batching timeout of a large group $\mathcal{X}$ with two more applications, we can *iteratively* apply Eq. (5) to a sequence of two applications in the group $\mathcal{X}$.

According to the pricing of function resources in Alibaba Cloud Function Compute [4], we further model the average monetary cost $C^{\mathcal{X}}$ of an inference request in terms of vCPU cores, GPU memory, and function invocations, which is given by

$$C^{\mathcal{X}} = \frac{1}{b^{\mathcal{X}}} \big[L_{avg}^t \cdot (c \cdot K_1 + m \cdot K_2) + K_3 \big], \tag{6}$$

where L_{avg}^t is the average inference latency with the batch size $b^{\mathcal{X}}$ and the function type t. K_1, K_2 is the unit cost of vCPU cores c and GPU memory m. K_3 is the constant cost of each function invocation. In particular, $m = 0$ represents a CPU function, and $c = 0$ represents a GPU function.

IV. Algorithm Design

In this section, we first formulate the optimization problem of function resource provisioning for multi-SLO DNN inference. We then design and implement *HarmonyBatch* to provide predictable performance for multi-SLO DNN inference with heterogeneous serverless functions.

A. Optimizing Serverless Inference Resource Provisioning

We assume a set of DNN inference applications $\mathcal{W} = \{w_1, w_2, ..., w_n\}$ sharing the same DNN model with the inference latency SLOs $\mathcal{S} = \{s^{w_1}, s^{w_2}, ..., s^{w_n}\}$ and request arrival rates $\mathcal{R} = \{r^{w_1}, r^{w_2}, ..., r^{w_n}\}$. We categorize the application set $\mathcal{W}$ into several *groups* $\mathcal{G} = \{\mathcal{X}_1, \mathcal{X}_2, ..., \mathcal{X}_m\}$. Each group $\mathcal{X} = \{w_j, w_{j+1}, ...\}$ is provisioned with an appropriate CPU function or GPU function, with the aim of meeting application SLO requirements while minimizing the average monetary cost of each inference request. Based on the DNN inference performance and cost models in Sec. III, we can formulate the optimization problem as

$$\min_{\mathcal{G}, \mathcal{F}, \mathcal{B}} \quad Cost = \sum_{\mathcal{X} \in \mathcal{G}} \eta^{\mathcal{X}} \cdot C^{\mathcal{X}} \tag{7}$$

$$\text{s.t.} \quad m^{\mathcal{X}} \geq M^{\mathcal{X}}, \ \forall \ \mathcal{X} \in \mathcal{G} \tag{8}$$

$$b^{\mathcal{X}} \leq \lfloor r^{\mathcal{X}} \cdot T^{\mathcal{X}} \rfloor + 1, \forall \ \mathcal{X} \in \mathcal{G} \tag{9}$$

$$t^w + L_{max}^t \leq s^w, \ \forall \ w \in \mathcal{X}, \ \mathcal{X} \in \mathcal{G} \tag{10}$$

where $\eta^{\mathcal{X}}$ is the ratio of the request arrival rate of a group $\mathcal{X}$ to the total request arrival rate. $C^{\mathcal{X}}$ is the average monetary cost of a group $\mathcal{X}$. Each group $\mathcal{X}$ is configured with a function of resource $f^{\mathcal{X}} \in \mathcal{F}$ (*i.e.,* a tuple of vCPU cores $c^{\mathcal{X}}$ and GPU memory $m^{\mathcal{X}}$, $f^{\mathcal{X}} = [c^{\mathcal{X}}, m^{\mathcal{X}}]$). Constraint (8) guarantees the GPU memory demands $M^{\mathcal{X}}$ of inference, which are proportional to the batch size. In addition, $b^{\mathcal{X}} \in \mathcal{B}$ denotes the batch size configured to group $\mathcal{X}$ and t^w is the timeout configured with the application w. Constraint (9) guarantees that DNN inference is executed with the batch size $b^{\mathcal{X}}$. Constraint (10) guarantees the latency SLO s^w for an application w, where the L^t_{max} is the maximum inference latency with the batch size $b^{\mathcal{X}}$. $T^{\mathcal{X}}$ is calculated by t^w in the group $\mathcal{X}$ by Eq. (5). To greedily enlarge the batching timeout $T^{\mathcal{X}}$, we set the t^w as the maximum value which meets the Constraint (10) (*i.e.,* $t^w = s^w - L^t_{max}$).

Problem analysis. Our group solution $\mathcal{G}$ has a large searching space of $B_{|\mathcal{W}|}$, which is the Bell number [22]. Given a group $\mathcal{G}$, the function resource $f \in \mathcal{F}$ can only take limited discrete values. Meanwhile, the batch size $b \in \mathcal{B}$ is constrained to integer values. Accordingly, the resource provisioning problem can be reduced to an integer programming problem. Obviously, the total average monetary cost and Constraint (10) is non-linear with the configuration parameters, and thus our optimization problem can further be reduced to a *non-linear integer programming* problem, which is an NP-hard problem [23]. We turn to designing a heuristic algorithm in Sec. IV-B to solve such an optimization problem.

B. Design of HarmonyBatch Strategy

HarmonyBatch divides the problem into two parts: *First* is to divide the applications into different groups, and *second* is to provision function resources for each application group.

Two-stage merging for application groups. We consider placing the adjacent applications sorted by their SLOs in *ascending order* into a group. According to Constraint (10), the batching timeout gap between two applications can be substantial if their SLO difference is large. If the two applications (with batching timeouts denoted as T_1 and T_2, where $T_1 < T_2$) are grouped together, the equivalent batching timeout $T^{\mathcal{X}}$ becomes much less than T_2 by analyzing Eq. (5). This causes a smaller *aggregated* batch size of DNN inference for all groups $\mathcal{X} \in \mathcal{G}$, thereby leading to a lower cost-efficient function resource provisioning, as evidenced by Sec. II-B.

Based on our analysis above, we design a two-stage group merging strategy in Alg. 1. Initially, each application is considered as a group, and the function provisioning plan and monetary cost for each group are calculated by our `funcProvision` strategy (lines 1-2). In the *first* stage, groups originally deployed on CPU functions are merged to be deployed on GPU functions as much as possible. To support the group merging for applications with adjacent SLOs, we sort the applications in $\mathcal{G}$ into a list $\mathbf{L}$ based on their SLOs, and only consecutive groups (or applications) in the list are merged (line 3). We leverage the *knee point* of the request arrival rate as illustrated in Fig. 7 to be the *threshold* r^* for group merging. If the total request arrival rate exceeds r^*, merging provides an opportunity to configure a more efficient GPU function (lines 4-13). In the *second* stage, groups deployed on GPU functions and adjacent groups based on SLOs are merged as much as possible to increase the request arrival rate of the merged groups. By iterating each group on GPU functions, *HarmonyBatch* examines whether they can reduce the monetary cost after group merging (lines 14-20). After that, *HarmonyBatch* outputs the application groups and their corresponding function provisioning plans.

Algorithm 1: *HarmonyBatch*: Two-stage merging strategy for application groups.

Input : A set of applications $\mathcal{W}$ with their SLOs $\mathcal{S}$ and arrival rates $\mathcal{R}$.
Output: A set of group $\mathcal{G}$, sets of function provisioning plans $\mathcal{F}$ and batch size $\mathcal{B}$.

1 **Initialize:** $\mathcal{G} \leftarrow \{\{w_1\}, \{w_2\}, ..., \{w_n\}\}$;
2 Provision function resources for each $\mathcal{X} \in \mathcal{G}$, $C^{\mathcal{X}}, f^{\mathcal{X}}, b^{\mathcal{X}} \leftarrow$ `funcProvision`$(\mathcal{X}, s^{\mathcal{X}}, r^{\mathcal{X}})$;
3 Sort the applications in $\mathcal{G}$ with SLOs and initialize the group list $\mathbf{L} \leftarrow sortWithSLO(\mathcal{G})$;
`// Stage1: merging CPU functions`
4 Set index $i \leftarrow 0, j \leftarrow 0$; Set the request arrival rate $r \leftarrow 0$;
5 **while** $i < |\mathbf{L}|$ **do**
6 **if** $c^{\mathbf{L}[i]} > 0$ **then**
7 $r \leftarrow r + r^{\mathbf{L}[i]}$;
`//` r^* `is the arrival rate knee point`
8 **if** $r > r^*$ **then**
9 $\mathbf{L}, \_ \leftarrow$ `Merge`$(\mathbf{L}, j, i+1)$;
10 Set $i \leftarrow j$, $j \leftarrow j+1$ and $r \leftarrow 0$;
11 **else**
12 Set $j \leftarrow i+1$ and $r \leftarrow 0$;
13 Set $i \leftarrow i+1$;
`// Stage2: merging GPU functions`
14 Set index $i \leftarrow 0$;
15 **while** $i < |\mathbf{L}| - 1$ **do**
16 **if** $m^{\mathbf{L}[i]} > 0$ *or* $m^{\mathbf{L}[i+1]} > 0$ **then**
17 $\mathbf{L}, isMerged \leftarrow$ `Merge`$(\mathbf{L}, i, i+2)$;
18 **if** *isMerged* **then**
19 $i \leftarrow i-1$;
20 Set $i \leftarrow i+1$;
21 **return** $\mathcal{G}, \mathcal{F}, \mathcal{B}$;
22 **Function** `Merge`$(\mathbf{L}$, *low, high*$)$**:**
23 $\mathcal{X} \leftarrow \mathbf{L}[low] \cup \mathbf{L}[low+1] \cup ... \mathbf{L}[high-1]$;
24 Provision function resources for group $\mathcal{X}$, $C^{\mathcal{X}}, f^{\mathcal{X}}, b^{\mathcal{X}} \leftarrow$ `funcProvision`$(\mathcal{X}, s^{\mathcal{X}}, r^{\mathcal{X}})$;
25 **if** $C^{\mathcal{X}}$ *is lower than the cost before merging* **then**
26 Update $\mathcal{G}$, $\mathcal{F}$ and $\mathcal{B}$ with the function provisioning plan;
27 $\mathbf{L} \leftarrow \mathbf{L}[:low] + \mathcal{X} + \mathbf{L}[:high]$;
28 **return** $\mathbf{L}$, *True*;
29 **return** $\mathbf{L}$, *False*;

`funcProvision`**: Function provisioning for an application group.** We analyze the optimization of CPU/GPU function resource provisioning. The configuration space of

vCPU cores $c^{\mathcal{X}} \in [0.05, 16]$ for CPU functions with the step of 0.05 is larger than that of $b^{\mathcal{X}} \in [1, 4]$. The configuration space of GPU memory $m^{\mathcal{X}} \in [1, 24]$ for GPU functions with the step of 1 is smaller than that of $b^{\mathcal{X}} \in [1, 32]$. We find that the CPU function exhibits a smaller batch size configuration space (4 choices) but a larger resource configuration space (320 choices), whereas the GPU function shows the opposite characteristics. To speed up searching for the optimal solution, we derive two theorems by analyzing Eq. (6).

Theorem 1. *Given a batch size $b^{\mathcal{X}}$, the minimum cost of CPU function provisioning can be achieved with the allocated vCPU cores set as c^* (i.e., the relative minimum point or the boundary points).*

Proof. The proof can be found in Appendix B [21]. □

Theorem 2. *Given an amount of GPU memory $m^{\mathcal{X}}$, the minimum cost of GPU function provisioning can be achieved if the following condition holds.*

$$\lfloor r^{\mathcal{X}} \cdot T^{\mathcal{X}} \rfloor + 1 = b^{\mathcal{X}}. \tag{11}$$

Proof. The proof can be found in Appendix C [21]. □

Based on Theorem 1 and Theorem 2, we simply adopt the *binary search* method in *HarmonyBatch* to fast identify the cost-efficient function provisioning plan (*i.e.,* the vCPU cores for the CPU function and the batch size for the GPU function, respectively). Accordingly, *HarmonyBatch* can minimize the inference budget, while guaranteeing the latency SLOs for an application group $\mathcal{X}$.

Remark. The complexity of Alg. 1 is in the order of $\mathcal{O}(|\mathcal{W}| \cdot M_{max} \cdot \log_2 B_{max})$, where $|\mathcal{W}|$ is the number of the applications. M_{max} and B_{max} are the maximum GPU memory and the maximum batch size, respectively. This is attributed to the relatively small batch size considered by the CPU function, which makes the complexity of the function provision strategy mainly depend on the size of the search space of GPU function resource provisioning. The computation overhead of Alg. 1 is well contained, which will be evaluated in Sec. V-D.

C. Implementation of HarmonyBatch Prototype

The *HarmonyBatch* prototype is implemented on the Alibaba Compute Function platform [4] with over 1,400 lines of Python codes, which are publicly available on GitHub. We implement four representative DNN inference workloads based on PyTorch[4] v1.13.0 and ONNX Runtime[5] v1.16.1. We use the Alibaba FC-Open Python SDK [4] to update function resources. We deploy the *HarmonyBatch* on a dedicated cloud instance, which receives DNN inference requests from a set of user applications. We first set up a request queue for each application group. We then batch the inference requests in the queue and finally route them to the provisioned CPU/GPU functions. To handle the request arrival variations, *HarmonyBatch* can be *periodically* executed to provision functions for DNN inference workloads. *HarmonyBatch* mainly determines the batching and function resource configurations. The batching configurations are sent to the *batch manager* to control the request queue, while the function resource configurations are sent to the serverless platform to *vertically* scale up or scale down functions.

[4]https://pytorch.org
[5]https://onnxruntime.ai

TABLE III: DNN inference workloads deployed in our experiments.

Workloads	VideoMAE	VGG-19	BERT	GPT-2
Framework	PyTorch	PyTorch	Onnx Runtime	PyTorch
Domains	Video Analytics	Image Processing	NLP	NLP
Datasets	Kinetics-400	ImageNet	Wikipedia	ShareGPT

V. Performance Evaluation

In this section, we evaluate *HarmonyBatch* by conducting a set of prototype experiments with four representative DNN models (listed in Table III) deployed on Alibaba Cloud Function Compute [4]. We seek to answer the questions as follows.

- **Accurary:** Can our model in *HarmonyBatch* accurately predict the DNN inference latency with heterogeneous serverless functions? (Sec. V-B)
- **Effectiveness:** Can our function provisioning strategy in *HarmonyBatch* provide predictable performance for multi-SLO DNN inference while saving the monetary cost? (Sec. V-C)
- **Overhead:** How much runtime overhead does *HarmonyBatch* practically bring? (Sec. V-D)

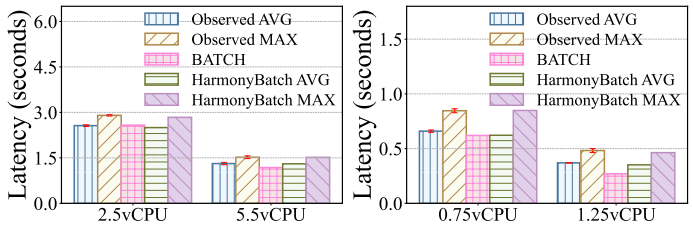
A. Experimental Setup

Configurations of DNN inference workloads. We select four representative DNN models as listed in Table III and replay the real-world trace from Azure Function [15] to evaluate the effectiveness of *HarmonyBatch*. The DNN models are selected from diverse fields, *i.e.,* VideoMAE [24] in video analytics, VGG-19 [17] in image processing, BERT [18] and GPT-2 [25] in NLP. We use several widely-used datasets including Kinetics-400[6], ImageNet[7], Wikipedia[8] and ShareGPT[9] for serving the four models above, respectively.

Configurations of serverless functions. We deploy our inference models in China Shanghai region of Alibaba Cloud Function Compute [4]. We adopt an `ecs.e-c1m1.large` ECS instance to deploy *HarmonyBatch*. During the period of our experiments (Nov. 2023), the unit price of vCPU cores is $K_1 = 1.3e^{-5}$\$/vCPU·s, and the unit price of GPU memory is $K_2 = 1.5e^{-5}$\$/GB·s, as well as the constant unit cost of a function invocation is $K_3 = 1.3e^{-7}$\$.

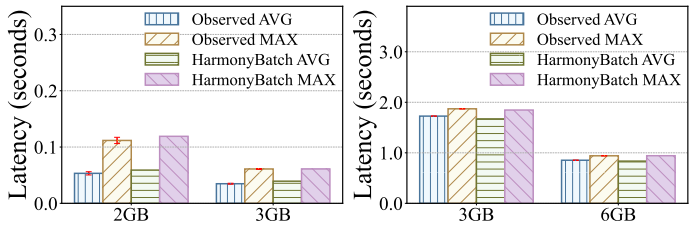
Baselines and metrics. We compare *HarmonyBatch* with two baselines: (1) BATCH [8]: It leverages multi-variable parametric regression to model the latency, and separately provisions CPU functions only for each application using an

[6]https://www.deepmind.com/open-source/kinetics
[7]https://image-net.org/challenges/LSVRC/2017
[8]https://en.wikipedia.org/wiki/English_Wikipedia
[9]https://sharegpt.com



(a) VideoMAE (b) VGG-19

Fig. 9: Comparison of the observed and predicted inference latency of VideoMAE and VGG-19 executed on CPU functions.



(a) BERT (b) GPT-2

Fig. 10: Comparison of the observed and predicted latency of BERT and GPT-2 executed on GPU functions. BATCH does support GPU functions.

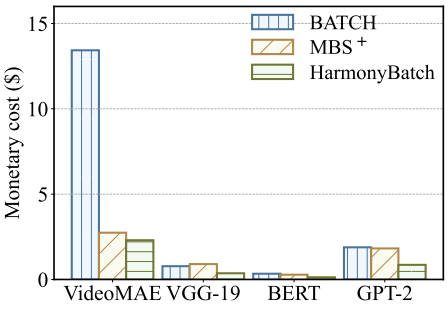


Fig. 11: Comparison of the monetary cost of various function resource provisioning strategies for representative DNN models.

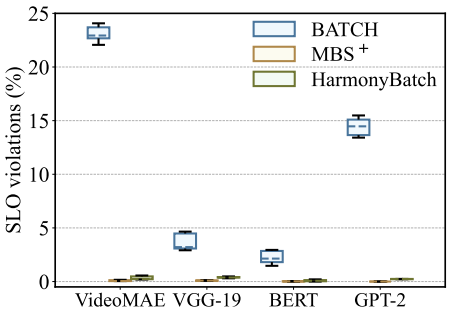


Fig. 12: Comparison of the SLO violations of various function resource provisioning strategies for representative DNN models.

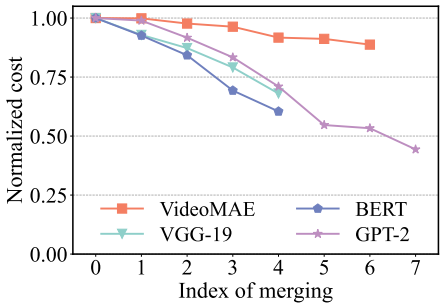


Fig. 13: Normalized monetary cost of four representative DNN models by increasing the merging index of application groups over time.

exhaustive search. (2) MBS$^+$: It is the extended MBS [12], which employs Bayesian-based provisioning to distribute requests into several groups evenly. We incorporate our performance model into MBS$^+$ to support heterogeneous function provisioning for application groups. We focus on three metrics: *SLO violations*, *monetary cost*, and *runtime overhead*.

B. Validating Inference Performance Model of HarmonyBatch

Can *HarmonyBatch* well predict the DNN inference latency? We first evaluate the latency of VideoMAE and VGG-19 on the CPU function by setting the batch size as 1. As depicted in Fig. 9, *HarmonyBatch* can well predict the average and maximum inference latency of VideoMAE and VGG-19 with the prediction error ranging from 0.2% to 6.1%. In contrast, BATCH [8] poorly predicts the DNN inference latency as it treats the model latency as a deterministic distribution with the prediction error ranging from 0.7% to 43.0%. We next evaluate the inference latency of BERT and GPT-2 on the GPU function by setting the batch size as 8. As depicted in Fig. 10, *HarmonyBatch* exhibits strong predictive capabilities, accurately predicting the average and maximum inference latency of BERT and GPT-2 with prediction errors ranging from 0.1% to 11.4%. In particular, the accuracy in predicting the maximum latency of BERT achieves a low prediction error of 0.1%, which can be attributed to the integration of GPU time-slicing scheduling mechanism into our inference performance model.

C. Effectiveness of HarmonyBatch Function Provisioning

Can *HarmonyBatch* guarantee the DNN inference performance while minimizing the monetary cost? To evaluate the efficacy of *HarmonyBatch*, we utilize 8 applications for each DNN model (32 applications in total). Specifically, we set the application SLOs between 0.2 and 1.0 seconds with an interval of 0.1 seconds for VGG-19 and BERT. Also, we set the application SLOs between 1.0 and 2.4 seconds with an interval of 0.2 seconds for VideoMAE and GPT-2.

As shown in Fig. 11, *HarmonyBatch* can save the cost by up to 82.9% compared with the two baselines. Specifically, BATCH can hardly reduce the cost with predictable performance, because it only batches requests for individual applications. Meanwhile, it overlooks the significant cost reduction of GPU functions, particularly for resource-intensive DNN models like VideoMAE. In contrast, MBS$^+$ and *HarmonyBatch* implement a heterogeneous serverless provisioning strategy, which leverages GPU functions to significantly enhance the advantages of batching, thereby gaining a substantial cost benefit. Furthermore, *HarmonyBatch* achieves cost reductions of 16.1%, 59.6%, 54.5% and 52.9% for the four workloads compared to MBS$^+$. This is attributed to the fact that MBS$^+$, with its evenly distributed requests, commonly aggregates inference requests with significant differences in SLOs into application groups. Moreover, it distributes the inference requests from the application with high request arrival rates to several functions, resulting in smaller batch sizes.

Regarding the SLO violations as illustrated in Fig. 12, BATCH can hardly provide predictable inference performance for DNN models, because it treats the inference request latency as a deterministic distribution, which leads to an excessively large SLO violations across large DNN models. In contrast, both *HarmonyBatch* and MBS$^+$ can guarantee inference performance for all DNN models. Furthermore, *HarmonyBatch* can obtain the function provisioning plans much faster than

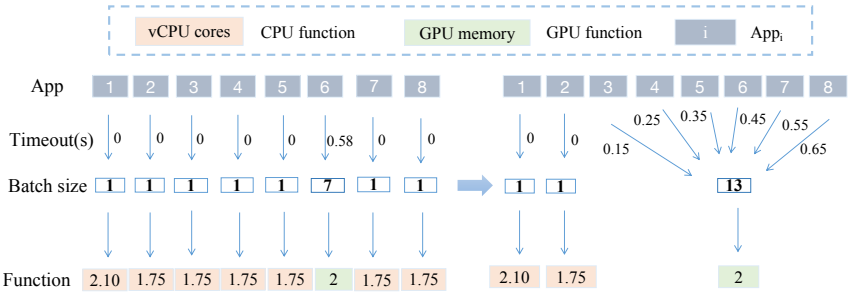


Fig. 14: Function provisioning plans of VGG-19 before and after the group merging.

Number of applications	BATCH	MBS$^+$	*HarmonyBatch*
1	34	176	2
6	222	3,497	23
12	393	8,156	38

TABLE IV: Computation time (in milliseconds) of different strategies.

MBS$^+$ as elaborated in Sec. V-D.

Can *HarmonyBatch* reduce the monetary cost as the merging of application groups proceeds? As depicted in Fig. 13, *HarmonyBatch* separately provisions function resources for each application at the initial time. After 4 to 7 merging operations, the four DNN inference workloads achieve notable cost reduction by $13.1\% - 62.4\%$. By taking VGG-19 as an example, it experiences a significant reduction in the monetary cost by up to 38.4%. The reason is that before the merging, the request rates of 7 applications were low, making them be deployed on CPU functions. However, after the merging, 6 applications are grouped and deployed on a GPU function with a large batch size, achieving a significant cost reduction while guaranteeing application SLOs.

We look into the adjustments on function provisioning plans made by *HarmonyBatch* for VGG-19 during the two-stage merging process. As shown in Fig. 14, *HarmonyBatch* provisions 7 CPU functions and 1 GPU function at first. After the merging process, *HarmonyBatch* decreases the application groups as 3 and optimizes the function provisioning to only 2 CPU functions and 1 GPU function. It directs 79.0% of requests to the GPU function for DNN inference. Meanwhile, the batch size of the GPU function is increased from 7 to 13 after the merging process. As a result, *HarmonyBatch* prioritizes assigning inference requests to GPU functions whenever feasible, with the aim of greedily creating large batches and thus significantly reducing the DNN inference budget.

D. Runtime Overhead of HarmonyBatch

We evaluate the runtime overhead of *HarmonyBatch* including the workload profiling time and the computation time of Alg. 1. Specifically, the profiling time for obtaining model coefficients of VideoMAE [24], VGG-19 [17], BERT [18], and GPT-2 [25] on CPU functions are 13, 5, 11, and 18 minutes, respectively. The profiling time for each model listed in Table III is less than 1 minute on GPU functions. In addition, the profiling time to obtain the minimum time slice τ is merely 0.1 minutes which requires profiling only once.

To evaluate the computation time of *HarmonyBatch*, we provision a VGG-19 model with different numbers of applications from 1 to 12. As listed in Table IV, the algorithm computation time of *HarmonyBatch* is roughly linear to the number of applications, which is negligible as compared to the other two strategies. This is because we adopt a two-stage merging grouping strategy, which significantly reduces the complexity of the application grouping and function provisioning algorithms to $\mathcal{O}(|\mathcal{W}|\cdot M_{max}\cdot\log_2 B_{max})$. In particular, we evaluate the computation overhead of `funcProvision` for an application group. Its computation time is still acceptable because we obtain function resource provision plan using the binary search method instead of an exhaustive search.

VI. Related Work

Optimizing DNN inference with serverless functions. To reduce the serverless inference budget, BATCH [8] introduces batching inference requests in serverless platforms. MBS [12] further optimizes the padding overhead by aggregating similar-size requests in a batch. To reduce the memory consumption, Tetris [26] combines batching and concurrent executions in a serverless inference system. Different from prior works above, *HarmonyBatch* aims to adequately configure the batch size for multi-SLO DNN inference workloads by considering their SLOs explicitly. INFless [10] develops a serverless inference system with CPU and GPU resources by unifying their computing power using the floating point operations per second (FLOPS) metric. In contrast, *HarmonyBatch* builds an analytical model for the inference performance and cost of public heterogeneous functions. Moreover, *HarmonyBatch* can benefit from several recent DNN inference optimizations such as scalability improvements (*e.g.,* MArk [27], AsyFunc [9]) and fine-grained selective batching techniques (*e.g.,* Orca [28]).

Resource provisioning of serverless functions. To optimize the function provisioning, AWS Lambda Power Tuning [13] adopts workload profiling with all possible memory configurations to identify the optimal memory allocation for functions, which brings heavy overhead. Sizeless [29] adopts a machine learning model to predict the inference execution time, which brings a non-negligible model training cost. To mitigate such a training cost, COSE [14] and MBS [12] employ Bayesian Optimization to identify the cost-effective resource provisioning plan, which highly depends on initial sampling and seeding. In contrast, *HarmonyBatch* identifies a cost-effective function provisioning plan using an inference performance model. ElasticFlow [30] proposes a greedy algorithm to allocate GPU resources on serverless platforms dynamically which only works for DNN training workloads.

Performance modeling of DNN inference. To model DNN inference performance, BARISTA [31] employs the maximum likelihood estimation approach to obtain the distribution of

inference latency. BATCH [8] leverages multi-variable parametric regression to model the inference latency as a deterministic distribution. Instead of identifying the latency distribution, *HarmonyBatch* develops an analytical model of the average and maximum inference latency, which significantly reduces workload profiling overhead. A recent work named iGniter [7] emphasizes performance interference for GPU-based inferences. INFless [10] adopts a lightweight Combined Operator Profiling method to predict inference latency with GPU spatial-sharing. In contrast, we focus on modeling inference latency on public heterogeneous functions by explicitly considering the GPU time-slicing scheduling mechanism.

VII. Conclusion and Future Work

This paper presents the design and implementation of *HarmonyBatch*, a cost-efficient resource provisioning framework that achieves predictable performance for multi-SLO DNN inference with heterogeneous serverless functions. *HarmonyBatch* consists of a lightweight performance and cost model of DNN inference on heterogeneous functions and a two-stage merging strategy, which judiciously batches the multi-SLO DNN inference requests into application groups and provisions each group with adequate CPU or GPU function resources. Prototype experiments on Alibaba Cloud Function Compute demonstrate that *HarmonyBatch* can deliver predictable DNN inference performance on serverless platforms while saving the monetary cost by up to 82.9% compared to the state-of-the-art methods, yet with acceptable runtime overhead.

We plan to extend *HarmonyBatch* in two directions: (1) supporting large model inference by leveraging model partitioning, and (2) supporting other public serverless platforms (*e.g.,* AWS Lambda) when GPU functions are enabled.

References

[1] J. Jarachanthan, L. Chen, F. Xu, and B. Li, "AMPS-Inf: Automatic Model Partitioning for Serverless Inference with Cost Efficiency," in *Proc. of ACM ICPP*, Aug. 2021, pp. 1–12.

[2] T. Brown, B. Mann, N. Ryder, M. Subbiah, J. D. Kaplan, P. Dhariwal, A. Neelakantan, P. Shyam, G. Sastry, A. Askell *et al.*, "Language Models Are Few-shot Learners," *Advances in Neural Information Processing Systems*, vol. 33, pp. 1877–1901, Dec. 2020.

[3] Y. Fu, L. Xue, Y. Huang, A.-O. Brabete, D. Ustiugov, Y. Patel, and L. Mai, "ServerlessLLM: Locality-Enhanced Serverless Inference for Large Language Models," *arXiv preprint arXiv:2401.14351*, 2024.

[4] Alibaba. (2023, Oct) Function Compute. [Online]. Available: https://www.alibabacloud.com/product/function-compute

[5] J. Gu, Y. Zhu, P. Wang, M. Chadha, and M. Gerndt, "FaST-GShare: Enabling Efficient Spatio-Temporal GPU Sharing in Serverless Computing for Deep Learning Inference," in *Proc. of ACM ICPP*, Aug. 2023, pp. 635–644.

[6] D. Du, Q. Liu, X. Jiang, Y. Xia, B. Zang, and H. Chen, "Serverless Computing on Heterogeneous Computers," in *Proc. of ACM ASPLOS*, 2022, pp. 797–813.

[7] F. Xu, J. Xu, J. Chen, L. Chen, R. Shang, Z. Zhou, and F. Liu, "iGniter: Interference-Aware GPU Resource Provisioning for Predictable DNN Inference in the Cloud," *IEEE Transactions on Parallel and Distributed Systems*, vol. 34, no. 3, pp. 812–827, 2023.

[8] A. Ali, R. Pinciroli, F. Yan, and E. Smirni, "BATCH: Machine Learning Inference Serving on Serverless Platforms with Adaptive Batching," in *Proc. of IEEE SC*, Nov. 2020, pp. 1–15.

[9] Q. Pei, Y. Yuan, H. Hu, Q. Chen, and F. Liu, "AsyFunc: A High-Performance and Resource-Efficient Serverless Inference System via Asymmetric Functions," in *Proc. of ACM SOCC*, Oct. 2023, pp. 324–340.

[10] Y. Yang, L. Zhao, Y. Li, H. Zhang, J. Li, M. Zhao, X. Chen, and K. Li, "INFless: A Native Serverless System for Low-Latency, High-Throughput Inference," in *Proc. of ACM ASPLOS*, Feb. 2022, pp. 768–781.

[11] S. Cai, Z. Zhou, K. Zhao, and X. Chen, "Cost-Efficient Serverless Inference Serving with Joint Batching and Multi-Processing," in *Proc. of ACM APSys*, Aug. 2023, pp. 43–49.

[12] A. Ali, R. Pinciroli, F. Yan, and E. Smirni, "Optimizing Inference Serving on Serverless Platforms," *Proceedings of the VLDB Endowment*, vol. 15, no. 10, pp. 2071–2084, 2022.

[13] A. Casalboni. (2023, Oct) AWS Lambda Power Tuning. [Online]. Available: https://github.com/alexcasalboni/aws-lambda-power-tuning

[14] N. Akhtar, A. Raza, V. Ishakian, and I. Matta, "COSE: Configuring Serverless Functions using Statistical Learning," in *Proc. of IEEE Infocom*, Jul. 2020, pp. 129–138.

[15] M. Shahrad, R. Fonseca, I. Goiri, G. Chaudhry, P. Batum, J. Cooke, E. Laureano, C. Tresness, M. Russinovich, and R. Bianchini, "Serverless in the Wild: Characterizing and Optimizing the Serverless Workload at a Large Cloud Provider," in *Proc. of USENIX ATC*, Jul. 2020, pp. 205–218.

[16] A. Joosen, A. Hassan, M. Asenov, R. Singh, L. Darlow, J. Wang, and A. Barker, "How Does It Function? Characterizing Long-term Trends in Production Serverless Workloads," in *Proc. of ACM SOCC*, Oct. 2023, pp. 443–458.

[17] K. Simonyan and A. Zisserman, "Very Deep Convolutional Networks for Large-Scale Image Recognition," *arXiv preprint arXiv:1409.1556*, 2014.

[18] J. Devlin, M.-W. Chang, K. Lee, and K. Toutanova, "BERT: Pre-training of Deep Bidirectional Transformers for Language Understanding," in *Proc. of NAACL-HLT*, Jun. 2019, pp. 4171–4186.

[19] Alibaba. (2023, Oct) Install and Use cGPU on a Docker Container. [Online]. Available: https://www.alibabacloud.com/help/en/egs/developer-reference/install-and-use-cgpu-on-a-docker-container

[20] S. Li, W. Wang, J. Yang, G. Chen, and D. Lu, "Golgi: Performance-Aware, Resource-Efficient Function Scheduling for Serverless Computing," in *Proc. of ACM SOCC*, Oct. 2023, pp. 32–47.

[21] J. Chen, F. Xu, Y. Gu, L. Chen, F. Liu, and Z. Zhou. (2024, May) HarmonyBatch: Batching multi-SLO DNN Inference with Heterogeneous Serverless Functions. [Online]. Available: https://github.com/icloud-ecnu/HarmonyBatch/blob/main/pdf/harmonybatch.pdf

[22] E. T. Bell, "The iterated exponential integers," *Annals of Mathematics*, pp. 539–557, Jul. 1938.

[23] S. P. Boyd and L. Vandenberghe, *Convex Optimization*. Cambridge university press, 2004.

[24] Z. Tong, Y. Song, J. Wang, and L. Wang, "Videomae: Masked Autoencoders Are Data-efficient Learners for Self-supervised Video Pre-training," *Advances in Neural Information Processing Systems*, vol. 35, pp. 10 078–10 093, Nov. 2022.

[25] A. Radford, J. Wu, R. Child, D. Luan, D. Amodei, I. Sutskever *et al.*, "Language Models are Unsupervised Multitask Learners," *OpenAI blog*, vol. 1, no. 8, pp. 1–24, 2019.

[26] J. Li, L. Zhao, Y. Yang, K. Zhan, and K. Li, "Tetris: Memory-efficient Serverless Inference through Tensor Sharing," in *Proc. of USENIX ATC*, Jul. 2022, pp. 473–488.

[27] C. Zhang, M. Yu, W. Wang, and F. Yan, "MArk: Exploiting Cloud Services for Cost-Effective, SLO-Aware Machine Learning Inference Serving," in *Proc. of USENIX ATC*, Jul. 2019, pp. 1049–1062.

[28] G.-I. Yu, J. S. Jeong, G.-W. Kim, S. Kim, and B.-G. Chun, "Orca: A Distributed Serving System for Transformer-Based Generative Models," in *Proc. of USENIX OSDI*, Jul. 2022, pp. 521–538.

[29] S. Eismann, L. Bui, J. Grohmann, C. Abad, N. Herbst, and S. Kounev, "Sizeless: Predicting the Optimal Size of Serverless Functions," in *Proc. of ACM/IFIP Middleware*, Dec. 2021, pp. 248–259.

[30] D. Gu, Y. Zhao, Y. Zhong, Y. Xiong, Z. Han, P. Cheng, F. Yang, G. Huang, X. Jin, and X. Liu, "ElasticFlow: An Elastic Serverless Training Platform for Distributed Deep Learning," in *Proc. of ACM ASPLOS*, Mar. 2023, pp. 266–280.

[31] 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 *Proc. of IEEE IC2E*, Jun. 2019, pp. 23–33.

APPENDIX

A. Derivation of Equivalent Batching Timeout

The probability that the first request arriving in the buffer belongs to App_i ($i = 1$ or 2) is $\eta_i = \frac{r_i}{r_1+r_2}$, where r_1 and r_2 represent the arrival rates for App_1 and App_2, respectively. If there are no subsequent requests after the first one, the request waiting time becomes T_i. However, if another request (belongs to App_j, $j = 1$ or 2) arrives in the buffer before T_i elapses, the request waiting time is updated to $\min(T_i, t + T_j)$, where t is the time at which the App_j request arrives.

More specifically, we assume that the buffer capacity limit is infinite and T_1 is shorter than T_2. We consider the following scenario: if the first request in the buffer is from App_2 and a request from App_1 arrives within a time frame of $T_2 - T_1$, the request waiting time is updated to $t+T_1$, where t represents the time at which the App_1 request arrives. However, the request waiting time remains to be T_2, if no request from App_1 arrives within $T_2 - T_1$.

Based on the above, App_1 follows a Poisson distribution with the request rate r_1. The equivalent batching timeout of the application group $\mathcal{X}$ with App_1 and App_2 can be calculated by

$$\begin{aligned} T^{\mathcal{X}} =& \eta_1 \cdot T_1 + \eta_2 \cdot \Big(\int_0^{T_2-T_1} f(t) \cdot (t + T_1) dt + \\ & T_2 \cdot (1 - \int_0^{T_2-T_1} f(t) dt) \Big) \\ =& T_1 + \eta_2 \cdot \frac{1 - \exp(-r_1 \cdot (T_2 - T_1))}{r_1}, \end{aligned} \tag{12}$$

where $f(t) = r_1 \cdot \exp(-r_1 \cdot t)$ represents the probability density function indicating the likelihood of a request from App_1 arriving at time t.

B. Proof of Theorem 1

Proof. By substituting Eq. (1) – (5) into Eq. (6), the average monetary cost of a group $\mathcal{X}$ configured with CPU functions can be reduced to

$$C^{\mathcal{X}} = \frac{1}{b^{\mathcal{X}}} \cdot [(\alpha_{b^{\mathcal{X}}}^{avg} \cdot \exp\left(-\frac{c}{\beta_{b^{\mathcal{X}}}^{avg}}\right) + \gamma_{b^{\mathcal{X}}}^{avg}) \cdot c \cdot K_1 + K_3]. \tag{13}$$

Based on the equation above, $C^{\mathcal{X}}$ has at most one relative minimum point, which can obtained by solving the equation of $(C^{\mathcal{X}})' = 0$ using the binary search method. The derivative of $C^{\mathcal{X}}$ with respect to c is given by

$$(C^{\mathcal{X}})' = \frac{K_1}{b^{\mathcal{X}}} \cdot [\alpha_{b^{\mathcal{X}}}^{avg} \cdot (1 - \frac{c}{\beta_{b^{\mathcal{X}}}^{avg}}) \cdot \exp\left(-\frac{c}{\beta_{b^{\mathcal{X}}}^{avg}}\right) + \gamma_{b^{\mathcal{X}}}^{avg}]. \tag{14}$$

Based on the above, the minimum value of $C^{\mathcal{X}}$ can only be obtained at the relative minimum point c_0 or boundary points c_{min} and c_{max}. Accordingly, the minimal monetary cost can be achieved with the allocated vCPU cores set as

$$c^* = \arg\min C^{\mathcal{X}}(c), \quad c \in \{c_0, c_{min}, c_{max}\}. \tag{15}$$

□

C. Proof of Theorem 2

Proof. By substituting Eq. (1) – (5) into Eq. (6), the average monetary cost of a group $\mathcal{X}$ configured with GPU functions can be obtained by

$$C^{\mathcal{X}} = M_{max} \cdot K_2 \cdot \xi_1 + \frac{M_{max} \cdot \xi_2 + K_3}{b^{\mathcal{X}}}. \tag{16}$$

According to the equation above, the cost $C^{\mathcal{X}}$ can be minimized, as $b^{\mathcal{X}}$ reaches its maximum value. We set t^w as the maximum value (*i.e.,* $s^w - L_{max}^t$) that satisfies the Constraint (10). As $b^{\mathcal{X}}$ increases, the corresponding value of t^w for applications in a group $\mathcal{X}$ decreases, leading to a reduced equivalent batching timeout $T^{\mathcal{X}}$, which is likely to violate Constraint (9). Accordingly, the maximum value of $b^{\mathcal{X}}$ is achieved when $\lfloor r^{\mathcal{X}} \cdot T^{\mathcal{X}} \rfloor + 1 = b^{\mathcal{X}}$, which can be determined efficiently through the binary search method. □