QoS-Aware and Cost-Efficient Dynamic Resource Allocation for Serverless ML Workflows

Hao Wu, Junxiao Deng, Hao Fan, Shadi Ibrahim, Song Wu, Hai Jin

QoS-Aware and Cost-Efficient Dynamic Resource Allocation for Serverless ML Workflows

Hao Wu*, Junxiao Deng*, Hao Fan*, Shadi Ibrahim†, Song Wu*, Hai Jin*
*National Engineering Research Center for Big Data Technology and System
Services Computing Technology and System Lab, Cluster and Grid Computing Lab
School of Computer Science and Technology, Huazhong University of Science and Technology, China
†Inria, Univ. Rennes, CNRS, IRISA, France
Email: {wuhao5, dengjunxiao, haofan, wusong, hjin}@hust.edu.cn, shadi.ibrahim@inria.fr

***Abstract*—*Machine Learning* (ML) workflows are increasingly deployed on serverless computing platforms to benefit from their elasticity and fine-grain pricing. Proper resource allocation is crucial to achieve fast and cost-efficient execution of serverless ML workflows (specially for hyperparameter tuning and model training). Unfortunately, existing resource allocation methods are static, treat functions equally, and rely on offline prediction, which limit their efficiency. In this paper, we introduce CE-scaling – a Cost-Efficient autoscaling framework for serverless ML workflows. During the hyperparameter tuning, CE-scaling partitions resources across stages according to their exact usage to minimize resource waste. Moreover, it incorporates an online prediction method to dynamically adjust resources during model training. We implement and evaluate CE-scaling on AWS Lambda using various ML models. Evaluation results show that compared to state-of-the-art static resource allocation methods, CE-scaling can reduce the job completion time and the monetary cost by up to 63% and 41% for hyperparameter tuning, respectively; and by up to 58% and 38% for model training.**

***Index Terms*—serverless computing, distributed machine learning, resource provisioning**

I. Introduction

The fine-grain pricing and easy management of serverless computing (or Function-as-a-Service) promote it as a major platform for building next-generation web services [1]. Serverless computing is especially attractive for distributed *Machine Learning* (ML) due to its high scalability [2]. According to *Gartner*, 70% of *Artificial Intelligence* (AI) applications will be built on containers and serverless computing by 2023 [3].

ML workflows comprise multiple phases, among which hyperparameter tuning and model training are the two most common yet critical ones. When deployed in serverless platforms, the ML workflows are decoupled into a group of functions that are performed on different datasets in parallel [4]–[6], i.e., each function trains the model on its own data partition, while periodically synchronizing the model with other functions.

Proper resource allocation is crucial to satisfy *Quality of Service* (QoS) objective (i.e., *Job Completion Time* – JCT) and to meet budget constraints of serverless ML workflows. Existing resource allocation methods distribute resources evenly across functions; or treat ML workflows like traditional data analytic applications (e.g., MapReduce and linear algebra) that implement static resource allocation methods [2], [7], [8]. Specifically, they employ offline prediction to select resource allocations for each epoch in ML workflows before they actually start [5], [9]. Unfortunately, these methods are inefficient and may fail in practice due to the following reasons:

– *Assume that functions are homogeneous and distribute resources evenly across them in hyperparameter tuning.* In order to find optimal hyperparameter configuration (e.g., learning rate and momentum), users implement hyperparameter tuning [10], i.e., train the model with different hyperparameter configurations in functions (trials) and stop functions with low accuracy prior to their completion. However, existing methods treat functions equally and therefore distribute resources evenly across them. This may lead to a dramatic waste of resources on functions that are terminated early.

– *Rely on offline prediction in model training.* Unlike traditional analytic applications which exhibit predictable performance, the training algorithm is stochastic, such as *Stochastic Gradient Descent* (SGD) [11]. The number of epochs required to converge to the target accuracy for model training is uncertain. As a result, relying on offline prediction of JCT and cost for training jobs may lead to inefficient resource allocation (i.e., inaccurate number and memory size of functions).

– *Overlook various external storage services.* External storage services (e.g., S3 [12], DynamoDB [13], and etc.) vary in terms of prices and performance, and can significantly impact the communication overhead of parameter synchronization. Thus, attaching a proper external storage service is essential to achieve good trade-off between JCT and cost of serverless ML workflows. However, existing methods consider only one type of external storage [4], [9], [14].

A large body of work has been dedicated to providing efficient resource allocation for ML workflows in clouds [10], [15]–[17] and serverless computing [4], [5], [9], [14]. However, they usually target coarse-grain resources allocation (i.e., virtual machines and containers) [10], [16], [17]. In addition, they overlook the interplay between the number of functions, memory sizes, and external storage services [4], [5], [9]; and implement offline predication and static allocations to avoid high scheduling overhead [14]. Consequently, they still result in resource waste and inefficient use of resources, and cannot provide dynamic allocation at runtime. Given that providing high resource efficiency is the main motivation

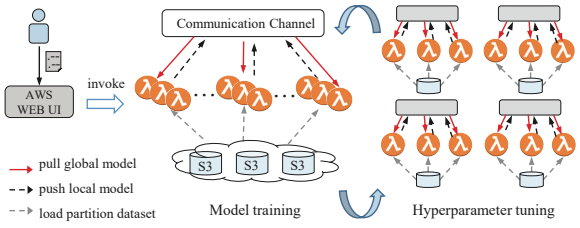


Fig. 1: A typical serverless ML workflow in AWS Lambda

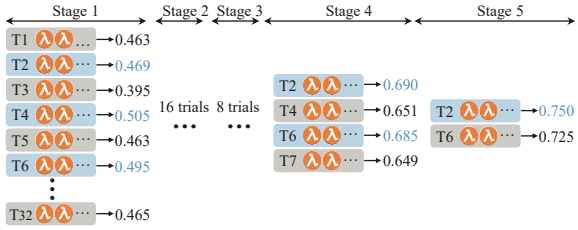


Fig. 2: An example of an early-stopping hyperparameter tuning procedure produced by SHA (with a reduction factor of 2). Each rectangle represents a trial, and each trial represents a training job which includes multiple functions.

behind introducing serverless computing, *the main technical problem addressed by this work is how to provide a fine-grain efficient resource partitioning and a dynamic and multi-dimensional resource allocation for serverless ML workflows*.

Contribution. We present CE-scaling – a Cost-Efficient autoscaling framework for serverless ML workflows. To address the above problems, CE-scaling builds JCT and cost models that characterize the interplay among multi-dimensional factors (i.e., number of functions, memory sizes, and external storage services). CE-scaling implements resource partitioning that optimally distributes resources across stages in hyperparameter tuning, thereby reducing the amount of resources "wasted" by terminated trials. Specifically, given a QoS or a budget constraint, the optimization of resource partitioning is NP-hard. CE-scaling performs an iterative greedy algorithm to solve the problem. Moreover, CE-scaling incorporates online prediction to dynamically adapt resources for model training at runtime. Finally, to quickly search the space of resource allocations and reduce the time overhead, CE-scaling uses Pareto boundary to prune out bad allocations.

We implement CE-scaling on Amazon Lambda and evaluate it using various ML models. Compared to existing static resource allocation methods [4], [9], [14]; CE-scaling improves the performance and reduces cost by up to 63% and 58% for hyperparameter tuning respectively; and by up to 41% and 38% for model training.

In summary, we make the following contributions:

- We analyze the limitations of existing resource allocation methods, and motivate the need for a dynamic resource allocation along with an appropriate selection of external storage service for serverless ML workflows.
- We propose a novel scheduling framework for efficient resource allocation in hyperparameter tuning and model training. Moreover, we optimize our framework to run with low overhead.
- We implement CE-scaling on Amazon Lambda and evaluate its effectiveness in reducing JCT and the cost.

II. Background and Motivation

A. Serverless ML Workflows

To reduce the complexity of deploying ML workflows and ease the resource management in the cloud [5], implementing ML workflows on serverless becomes increasingly compelling. Fig. 1 shows the implementation of a ML workflow [14] in a serverless platform (i.e., AWS Lambda [18]). In this work, we only focus on the two iterative phases (i.e., model training and hyperparameter tuning). Data preprocessing and data uploading from long-term storage are other research topics for serverless ML workflows. The two topics have been addressed in previous efforts [4], [7] and are complementary to our work.

Model training. Training is an iterative process over multiple epochs. In each epoch, according to the configuration file uploaded by the user, multiple functions are created to act as workers. The training datasets are initially stored in external storage (e.g., Amazon S3). These functions compute gradients based on input data, and then they update and synchronize the global model via external storage. The training is stopped when the loss reaches an objective value or when a given number of epochs have been processed [5].

Hyperparameter tuning. The quality of a ML model depends on the choice of its hyperparameters [15]. Accordingly, users conduct hyperparameter tuning, i.e., partially train the model with different hyperparameter configurations, and select the one with the highest accuracy. Similar to many ML systems [10], [19], [20], we adopt *Successive Halving* (SHA) technique to accelerate hyperparameter tuning. The training job associated with one hyperparameter configuration is referred to as a trial. As shown in Fig. 2, the hyperparameter tuning is executed in sequential stages, with multiple concurrent trials executing several epochs in each stage, followed by synchronous evaluation and termination of the bottom-performing trials. The number of stages, trials, and epochs are predefined by users. Note that other methods for hyperparameter tuning (e.g., BOHB [20]) share the same idea of repeatedly terminating poorly performing trials until the one with the highest accuracy is found. Thus, our work can be applied to them.

B. Cost of Serverless ML Workflows.

In general, the cost of serverless ML workflows includes two parts [21]: (a) *Cost of functions*. Users are charged based on the execution time and resource usage of all functions. (b) *Cost of external storage*. Because functions are stateless, they leverage external storage [22] to synchronize parameters.

TABLE I: Comparison of different external storage services

	Elastic scaling	Latency	Pricing pattern	Cost
S3	Auto	High	Data request	$
DynamoDB	Auto	Medium	Data request	$$
Elasticache	Manual	Low	Execution time	$$$
VM-PS	Manual	Low	Execution time	$$$

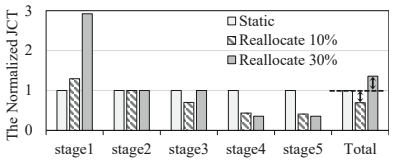
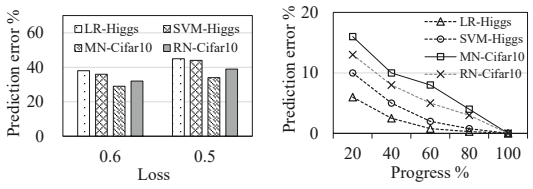


Fig. 3: Comparison of JCT of each stage between static resource allocation and reallocating different proportions of resources from stage 1. There are 5 stages, with 32 trials in the first stage and a reduction factor of 2.



(a) *Offline prediction* (b) *Online prediction*

Fig. 4: Comparison of online and offline prediction errors. The prediction error here is the difference between the estimated total number of epochs required to achieve the target accuracy and the actual number of epochs needed.

External storage services are classified into two categories: stateless storage services (e.g., AWS S3 [12], ElastiCache [23], and DynamoDB [13]) and user-defined storage (e.g., parameter server built with virtual machines (VM-PS) [14]). As shown in Table I, external storage services have different latency, pricing pattern, and cost. The JCT and cost of serverless ML workflows vary according to the allocated resources (i.e., the number of functions and the resource provisioned for functions) and the type of attached external storage that facilitates the communication among functions.

C. Limitations of Existing Resource Allocation Methods

Proper resource allocation not only accelerates the execution of ML workflows, but also saves money for users [5]. In this part, we study prior resource allocation methods for serverless ML workflows and discuss their limitations and deficiencies due to employing static allocation, relying on offline prediction, and overlooking the impact of storage services.

1) ***Resource partitioning in hyperparameter tuning.*** Current static methods treat trials across stages equally, therefore, resources are evenly partitioned across trials of different stages. Usually, the number of trials is reduced by a factor of 2 across stages [10] by terminating trials with low accuracy. Thus more resources are allocated to early stages according to the number of trials, but without taking into consideration that half of them will be terminated prior to their completion. This may result in resource waste and also impact the performance of later stages. Fig. 3 shows an example of hyperparameter tuning with 5 stages. For each trial, the resource allocations of all stages under the static method are the same, which means that the cost of each stage is proportional to the number of executed trials. The first three stages account for 90% of the overall cost, while the last stage accounts for only 3%.

As shown in Fig. 3, by reallocating resources from trials in the early stages (10% of the resources in the first stage) to trials in the later stages, JCT is reduced by 39%. The reason is that there are fewer trials in the later stages, and the resources of each trial in later stages are increased by nearly 2× thus can run faster. We can also observe that when reallocating resources from the early stages aggressively (30% of the resources from the first stage), JCT is increased by 36% compared to the static allocation method. This is due to the sharp drop in the performance of the first stage caused by resource competition. It is worth noting that increasing the allocated resources to later stages is not always beneficial as it may increase the communication overhead.

Finding 1: *The majority of trials in early stages are terminated earlier. In addition, the resource requirements of trials change with stages. Hence, reallocating resources from early stages – considering terminated trials – to later ones can improve the performance and cost-efficiency of hyperparameter tuning.*

2) ***Stochasticity in model training.*** Predicting the JCT and cost of a training job is challenging because of the stochasticity of training algorithm (e.g., SGD). Traditional resource allocation strategies which rely on offline prediction may not be efficient due to the inaccurate prediction. LambdaML [14] proposes a sampling-based method [11] to estimate the number of required epochs by pre-training the model on a small set of training data. As shown in Fig. 4(a), this sampling-based method has a high average prediction error of up to 40%. Siren [9] proposes to use reinforcement learning to improve the prediction. However, training this black-box model is time-consuming. Moreover, when training job changes, the prediction model needs to be retrained.

On the other hand, online prediction, i.e., fitting the convergence curve by collecting real-time training data, has been applied by many works [16], [17] to monitor the convergence speed of ML workflows. Based on the fitted model and the target accuracy, we can easily calculate the number of remaining epochs to get the target accuracy at runtime. Fig. 4(b) shows that the error of online prediction decreases gradually as more state data is collected during the training process, and the average error of the online prediction is about 5%. Hence, online prediction can help to better estimate and allocate resources of model training.

Finding 2: *Existing static resource allocation methods rely on inaccurate offline predictions, which limits their efficiencies. Online predictions are more accurate and therefore it is worth investigating how to employ them for efficient resource allocation in model training.*

3) ***Heterogeneity of external storage services.*** The JCT of serverless ML workflows is highly affected by the resource allocation and also by the performance of external storage, as all functions need to perform parameter synchronization at the end of each epoch. As shown in Fig. 5, parameter synchronization is the aggregation of the gradient data from each function. Stateless external storage, such as S3, aggregates data with the help of functions, while VM-based external storage can directly aggregate data locally. VM-based synchronization is more efficient but more expensive than S3.

When the resource allocation of a training job changes, the synchronization pattern and requirements change. Thus, it is important to re-evaluate external storage services when chang-

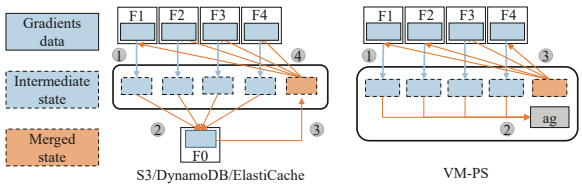


Fig. 5: Synchronization patterns of different external storage services

ing the resource allocation and attach the most appropriate one accordingly. Unfortunately, previous works overlooked the heterogeneity of external storage services. For example, Siren [9] and Cirrus [4] use S3 and VM-PS, respectively, by default. Table II presents the JCT and cost when training Logistic Regression and MobileNet under Cirrus with different resource allocations and external storage services (Table I). We have the following observations. First, under the same number of functions, the JCT and cost of the training models vary according to the used external storage service. Second, using the faster but expensive external storage services (i.e., Elasticache, VM-PS) does not necessarily result in best JCT nor lowest cost. When the number of functions is small, DynamoDB is faster and cheaper (when model size is less than 400KB) because the communication requirement is low. But when the number of functions increases, communication requirement increases. VM-based and Elasticache are more efficient, because VM-PS can significantly reduce the communication overhead and Elasticache can efficiently handle concurrent requests.

Finding 3: *External storage services can significantly impact the achieved performance and cost for a given resource allocation. Hence, we should co-jointly consider external storage and other resources when optimizing the resource allocation.*

III. CE-scaling Design

Given that existing resource allocation methods are inefficient when partitioning resources across stages in hyperparameter tuning and cannot accurately allocate resources in model training, and motivated by the aforementioned findings, we introduce CE-scaling. CE-scaling aims to reduce resource waste by providing an optimal partitioning of resources across stages. CE-scaling also provides an accurate estimation and dynamic allocation of resources during model training, thereby improving the performance and cost-efficiency of serverless

TABLE II: Comparison of existing external storage under Cirrus. We present the JCTs and costs, normalized to the JCT and cost of S3. A cost larger than 1 means that S3 is cheaper, whereas a JCT larger than 1 means that S3 is faster. N/A as the model size exceeds the limit of DynamoDB's object size.

Resource Allocation	Storage	Logistic Regression		MobileNet	
		JCT	Cost	JCT	Cost
10 functions/1769MB	**S3**	1	1	1	1
	DynamoDB	*0.83*	*0.95*	N/A	N/A
	Elasticache	0.97	1.73	0.97	0.82
	VM-PS	0.98	1.42	*0.89*	*0.77*
50 functions/1769MB	**S3**	1	1	1	1
	DynamoDB	0.94	0.97	N/A	N/A
	Elasticache	0.86	0.83	*0.85*	*0.74*
	VM-PS	*0.84*	*0.78*	0.90	0.78

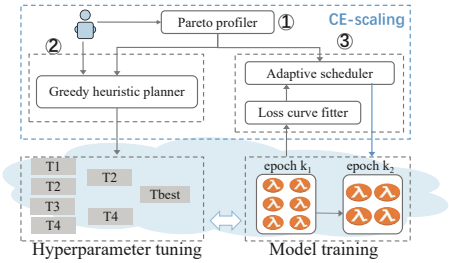


Fig. 6: CE-scaling system architecture

ML workflows. To attain these goals, CE-scaling follows three core design principles.

(1) *Partitioning resources across stages in hyperparameter should consider the exact usage of trials.* The majority of terminated trials happen in early stages. As these trials are terminated earlier, they require less resources compared to other trials across stages. Thus, focusing resources on later stages could be more cost-efficient. CE-scaling leverages this fact when partitioning resources across stages given QoS or budget constraints. In particular, CE-scaling formulates the optimization problem as a multiple-choices knapsack problem [24] – which is known to be NP-hard – and performs iterative greedy resource reallocation on the basis of static resource allocation (treating trials equally) to find an optimal resource partitioning.

(2) *Adaptive resource allocation in model training combined with online prediction.* CE-scaling relies on online prediction to provide an accurate estimation of resources in model training. However, the accuracy (average error) changes as the model training progresses. Accordingly, CE-scaling monitors the changes of the online prediction and adaptively adjusts the resources of model training at runtime. In addition, CE-scaling introduces a delayed function restart to reduce the overhead of resource adjustment.

(3) *Low overhead.*

There is a large search space of allocation plans, especially when considering co-jointly traditional resource allocation (i.e., number of functions and memory sizes) and external storage services. The overhead of estimation and resource scheduling can reach several minutes. This is unacceptable compared to second-level latency of function startup, especially for model training where resources are adjusted at runtime. CE-scaling focuses on a small subset of allocations (i.e., Pareto boundary of cost-JCT space) to reduce scheduling overhead.

A. System Architecture

As shown in Fig. 6, when users submit an ML model to the *Pareto profiler*, CE-scaling first builds analytical models for the cost and JCT of one epoch. Then, the *Pareto profiler* selects a subset of Pareto-optimal allocations. For hyperparameter tuning, *Greedy heuristic planner* outputs an optimal resource partitioning plan based on the budget or QoS constraints before hyperparameter tuning starts. For model training, *adaptive scheduler* adjusts resources at runtime based on the latest prediction result, and *loss curve fitter* tracks the status of model

TABLE III: Main notations

Notation	Definition
D	The size of dataset
M	The size of model
n	The number of provisioned functions
m	The size of function memory allocated
s	The type of external storage
θ	The resource allocation of one epoch
a	The resource allocation of stages in hyperparameter tuning
k, e	The number of training iterations, epochs
r_i	The number of epochs per stage
q_i	The number of trials per stage
$x_i(\theta)$	A binary variable which indicates whether stage i in hyperparameter tuning is configured to θ
$t'(\theta)$	The execution time for one epoch configured to θ
$c'(\theta)$	The cost for one epoch configured to θ
$\mathcal{P}$	The allocations on the Pareto boundary
b_s	Available network bandwidth of the external storage
ℓ_s	Latency of the external storage
p_f	The price of function
p_{ivk}	The price of function invocation
p_s	The price of external storage
b_c	The constraint of budget
τ	The constraint of QoS

training to fit the loss curve. Hereafter, we will describe in detail the main components of CE-scaling.

B. Pareto Boundary-based Profiling Estimation

In this section, we first build analytical models for the execution time and cost of one epoch in serverless ML workflows. These models characterize the interplay among multi-dimensional factors (i.e., number of functions, memory sizes, external storage services). Second, we select a small subset of allocations (Pareto boundary) from cost-JCT space. Our analytical models are applied for both model training and hyperparameter tuning. For clarity, important notations are listed in Table-III.

1) Execution time of one epoch. We denote the resource allocation of ith epoch as $\theta_i = (n_i, m_i, s_i)$. Let $\mathcal{M}$ denotes the set of memory configurations, $\mathcal{N}$ denotes the concurrency allowed, and $\mathcal{S}$ denotes the set of external storage services. Then, we can define the set of available allocations as:

$$\Theta = \{(n, m, s) \mid n \in \mathcal{N}, m \in \mathcal{M}, s \in \mathcal{S}\} \tag{1}$$

First, each function loads a dataset from external storage [4]. Then, each function calculates gradients based on the dataset. Finally, functions synchronize parameters through external storage. Communication of functions follows *Bulk Synchronous Parallel* (BSP) protocol, i.e., every function synchronizes parameters at each iteration, which has been widely used in production [5]. The number of iterations is $k = \frac{D}{n_i \cdot b_z}$, given a batch size b_z. We denote the execution time of epoch i as $t'(\theta_i)$, which consists of the time to load a dataset $t^l(\theta_i)$, the time of gradients calculation $t^g(\theta_i)$, and the time of parameter synchronization $t^p(\theta_i)$, we have:

$$\begin{aligned} t'(\theta_i) &= t^l(\theta_i) + k \cdot (t^g(\theta_i) + t^p(\theta_i)) \\ &= \frac{D}{n_i \cdot B_{S3}} + \frac{D}{n_i} \cdot k \cdot u(m_i) + k \cdot t^p(\theta_i) \end{aligned} \tag{2}$$

$$t^p(\theta_i) = \begin{cases} (3n_i - 2)(\frac{M}{b_s} + \ell_s), & \text{for stateless storage} \\ (2n_i - 2)(\frac{M}{b_s} + \ell_s), & \text{for VM-PS} \end{cases} \tag{3}$$

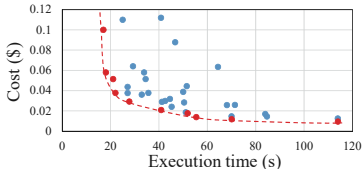


Fig. 7: A scatter plot of 50 allocations in the two-dimensional space of execution time and cost of one epoch, where the red dashed line is the Pareto boundary. These data are sampled from the ML model of logistic regression over higgs dataset.

where $u(m_i)$ is the processing time of 1MB data given m_i resource allocation, and the training dataset D is evenly distributed among functions. The communication time varies with the type of external storage. The difference in the constant, i.e., $(3n_i - 2)$ and $(2n_i - 2)$, is caused by the fact that stateless storage does not have a computation capacity (Fig. 5). So compared to VM-PS, stateless storage needs more time for data transfers as a function should re-pull the whole local model and then upload the aggregated model.

2) Monetary cost of one epoch. We denote the monetary cost of serverless ML workflows as $c'(\theta_i)$, which consists of two parts: cost of functions $c^f(\theta_i)$ and cost of external storage $c^s(\theta_i)$. The cost of a function is the sum of invocation cost and computation cost. Invocation cost is determined by the number of invoked functions, and the computation cost of a function is determined by the allocated memory and execution time. Then we have:

$$\begin{aligned} c'(\theta_i) &= k \cdot (c^f(\theta_i) + c^s(\theta_i)) \\ &= n_i \cdot p_{ivk} + k \cdot (n_i \cdot t'(\theta_i) \cdot p_f(m_i) + c^s(\theta_i)) \end{aligned} \tag{4}$$

$$c^s(\theta_i) = \begin{cases} k \cdot (10n_i + 2) \cdot p_s, & \text{charges for request} \\ (t'(\theta_i)/60 + 1) \cdot p_s, & \text{charges for runtime} \end{cases} \tag{5}$$

where the cost of external storage $c^s(\theta_i)$ varies with the type of external storage. There are two classes of external storage, characterized by the charging pattern: charging by requests (e.g., S3) and charging by runtime (e.g., VM-PS).

3) Pareto boundary of cost-JCT space. The entire search space of allocations is large. For example, we can allocate 1 to 10240 of MB memory to a function in AWS Lambda, and the concurrency of functions can reach 3000 [25]. For external storage, besides stateless storage services (e.g., S3, DynamoDB, and Elasticache), there are hundreds of types of EC2 that can be used to build VM-PS, varying in computing power, memory, and network bandwidth. The large search space makes it time-consuming to optimize resource allocation for hyperparameter tuning and model training.

Accordingly, we use Pareto boundary to prune out bad allocations so that we can focus on a small subset of allocations to avoid the overhead of searching the entire allocation space. Fig. 7 shows a scatter plot of 50 allocations in the two-dimensional space of cost and JCT. If θ_1 and θ_2 are two allocations such that $t'(\theta_1) < t'(\theta_2)$ and $c'(\theta_1) < c'(\theta_2)$, θ_2 is a bad allocation, and θ_1 is better than θ_2 in both execution time and monetary cost. The red line in Fig. 7 shows the Pareto boundary of such allocations for one epoch in the ML

workflows. We denote the Pareto subset of resource allocations as $\mathcal{P}$, which is applied in the allocation optimization for hyperparameter tuning and model training. *Note that pareto varies with different models and datasets, and can be quickly obtained – in few seconds – after users upload the model and the dataset.*

C. Smart Resource partitioning for Hyperparameter Tuning

We formulate the optimization problem of resource partitioning in hyperparameter tuning, and design a greedy heuristic planner to solve this problem. Given the number of stages d, we denote a resource partitioning plan as $a = (\theta_1, \theta_2, \cdots, \theta_d)$. Let $\mathcal{A}$ denotes the set of all possible resource partitioning plans. Furthermore, we use a binary variable $x_i(\theta)$ to indicate whether the ith stage is configured to θ:

$$x_i(\theta) = \begin{cases} 0, & i\text{th stage is not configured to } \theta \\ 1, & i\text{th stage is configured to } \theta \end{cases} \tag{6}$$

1) JCT minimization given a budget. Constrained by a particular budget b_c, we aim to minimize the JCT of hyperparameter tuning T^h. We formulate as follows:

$$\min T^h = \min_{a \in A} \sum\nolimits_{i=1}^{d} \sum_{\theta \in P} r_i \cdot t'(\theta) \cdot x_i(\theta) \tag{7}$$

$$\text{subject to:} \quad \sum\nolimits_{i=1}^{d} \sum_{\theta \in P} q_i \cdot r_i \cdot c'(\theta) \cdot x_i(\theta) \leq b_c \ , \tag{8}$$

$$T^h \leq \tau \tag{9}$$

where r_i represents the number of epochs in stage i, and q_i represents the number of trials in stage i. In this optimization problem, the objective is to minimize the sum of the execution times of each stage; and the cost is the sum of cost of all trials. Constraint (8) and (9) regulate that JCT and cost should satisfy the QoS constraint and the cost constraint.

This is a multiple-choices knapsack problem [24], a classic optimization problem which is known to be NP-hard. To solve this problem, we design a greedy heuristic planner through the insight of Section (II-C1), i.e., based on static allocation plan a^s, we reallocate some resources from the early stages to the trials in later stages. We warm-start the planner with the optimal static resource allocation. This plan is then improved in an iterative greedy fashion by (1) generating a set of new candidates from the current best partitioning plan, (2) predicting their JCT and cost separately, (3) selecting the best candidate, and (4) iterating until the best candidate partitioning plan no longer improves cost or exceeds time-constraints. We outline these steps in Algorithm 1 and elaborate them below.

Candidate generation. In each greedy step, the candidate partitioning plans $a_1 \cdots a_d$ are generated from the current best partitioning, a^*. Each candidate a_i is equivalent to a^* at every index, except at i, where the partitioning is to select a lower-cost partitioning or higher-cost partitioning.

Greedy selection. The planner selects the partitioning with the largest predicted JCT-marginal benefit:

Algorithm 1 Greedy heuristic resource partitioning planner

Input: budget-constraint b_c, warm-start static allocation plan a^s
Output: resource partitioning plan a^*

1: $a^e, a^l, a^* \leftarrow a^s$;
2: **while** $true$ **do**
3: $A_1 \leftarrow$generate_candidates_for_recycling (a^*);
4: $a^e \leftarrow$ select_best_candidate(A_1);
5: $a^l \leftarrow a^e$;
6: **while** $C^h(a^l) \leq C^h(a^s)$ **do**
7: $A_2 \leftarrow$generate_candidates_for_reallocating(a^l);
8: $a^l \leftarrow$ select_best_candidate(A_2);
9: **end while**
10: **if** JCT_reduction$(a^l, a^*) < \delta$ or violating t' **then**
11: break;
12: **end if**
13: $a^* \leftarrow a^l$;
14: **end while**
15: **while** $true$ **do**
16: $A_2 \leftarrow$generate_candidates_for_reallocating(a^*);
17: $a^l \leftarrow$ select_best_candidate$(A_2 - A_2')$;
18: **if** JCT_reduction$(a^l, a^*) < \delta$ or violating t **then**
19: break;
20: **end if**
21: **if** $C^h(a^l) \geq b_c$ **then**
22: insert a^l to A_2';
23: **end if**
24: $a^* \leftarrow a^l$;
25: **end while**
26: **return** a^*;

$$B^t(a_i) = \frac{T^h(a^*) - T^h(a_i)}{C^h(a_i) - C^h(a^*)} \tag{10}$$

where a_i is a candidate partitioning, T^h and C^h are JCT and cost of the hyperparameter tuning, respectively. We normalize cost saving by the corresponding increase in JCT to ensure a fair comparison between candidates. First, we reallocate resources from early stages (Lines 3-4) to later stages (Lines 7-8). Then, we repeat the process above until the reduction of JCT is negligible (Line 5). Last, we output the resource partitioning plan until the remaining budget is used up (Lines 15-25).

Warm start. The planner must be warm started with an optimal static allocation. Because the search space is reduced to a single dimension, we can enumerate candidate static allocations from $\mathcal{P}$, predict their costs and JCTs, and return the cost-optimal static allocation.

Remark. Although this algorithm does not guarantee finding the optimal solution, it does guarantee that the found resource partitioning plan is no worse than the static allocation, because our solution is incrementally optimized based on the optimal static resource allocation. Furthermore, excessive reallocation of resources in early stages will lead to a sharp decline in the benefits due to resource competition, as mentioned in Section (II-C1). Therefore, the exploration of resource partitioning plan will end quickly. As we show in Section (IV-G), the overhead of resource partitioning planner is less than one minute.

Algorithm 2 Adaptive resource scheduler

Input: The budget b_c, target loss σ^*, current loss σ, current index of epoch e' , latest predicted result e, current allocation θ
Output: resource allocation for adjustment θ^*

1: $e^* \leftarrow 0, \theta^* \leftarrow \theta$;
2: **if** $e = 0$ **then**
3: $b \leftarrow b_c$;
4: $e^* \leftarrow$ predict_epoch_offline(σ^*);
5: $\theta^* \leftarrow$ select_best_allocation($b, \mathcal{P}, e^*$);
6: **return** θ^*;
7: **end if**
8: fit the loss curve with epoch_predict(σ);
9: $b \leftarrow b -$ cost_for_one_epoch (θ);
10: $e^* \leftarrow$ predict_epoch_online(σ^*);
11: **if** $(e^* - e)/e > \delta$ **then**
12: $b_1 \leftarrow b -$ cost_for_one_epoch (θ);
13: $\theta^* \leftarrow$ select_best_allocation($b_1, \mathcal{P}, e^* - e' - 1$);
14: **end if**
15: **return** θ^*;

2) Cost minimization given a QoS constraint. Given a constraint of QoS, we next consider the following cost minimization problem:

$$\min C^h = \min_{a \in A} \sum_{i=1}^{d} \sum_{\theta \in P} q_i \cdot r_i \cdot c'(\theta) \cdot x_i(\theta) \qquad (11)$$
$$\text{subject to:} \quad (9)(8)$$

where the total monetary cost is the sum of the costs of all trials. The QoS constraint and cost constraint are represented by constraints (8) and (9).

To find the resource partitioning plan with minimal cost, we implement the same method as JCT minimization. Specifically, we replace $C^h(a)$ (Lines 6 and 21) of Algorithm 1 with $T^h(a)$. Furthermore, we replace the JCT-marginal benefit with the cost-marginal benefit $B^c(a_i)$ defined as follow.

$$B^c(a_i) = \frac{C^h(a^*) - C^h(a_i)}{T^h(a_i) - T^h(a^*)} \qquad (12)$$

D. Adaptive Resource Allocation for Model Training

First, we model the cost and JCT of model training based on section (III-B). Second, we design an adaptive resource scheduler, which adjusts resources based on the online prediction and restarts new functions in parallel to hide the overhead of resource adjustment.

The JCT and Cost of model training. We build an analytical model for model training. The JCT of model training T^m is equal to the sum of the execution times of each epoch. The cost C^m is equal to the sum of the costs of each epoch. We formulate the optimization problem as follows:

$$\min T^m = \min_{\theta \in P} \; e \cdot t'(\theta) \qquad (13)$$
$$\text{subject to: } C^m \leq b_c \qquad (14)$$
$$\min C^m = \min_{\theta \in P} \; e \cdot c'(\theta) \qquad (15)$$
$$\text{subject to: } T^m \leq \tau \qquad (16)$$

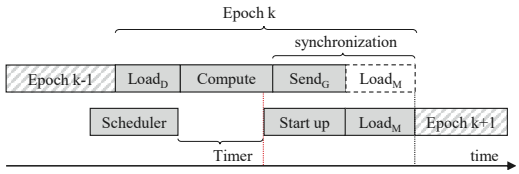


Fig. 8: An example of the optimal launch time for new functions. $Send_G$ is to upload gradient data, $Load_M$ is to pull the latest model data.

where e is the number of epochs required to achieve the target loss. As described in section (II-C2), predicting the number of required epochs with offline methods shows a high error. We will introduce how to allocate resources for model training based on an online prediction.

Adaptive resource adjustment. As described in Algorithm 2, first, we begin with the resource allocation based on offline prediction (Lines 3-6). Then, we collect training state data to continuously fit the loss curve, and monitor the number of epochs required to converge to the target loss σ^* (Lines 8-10). When the predicted epochs fluctuate beyond a threshold δ, resource adjustment is automatically triggered (Lines 11-13). Furthermore, for the problem of cost minimization and JCT minimization, we implement a greedy local search, i.e., selecting the local optimal allocation that fits the constraint of budget or QoS at each resource adjustment. Moreover, we only search resource allocations in Pareto subset $\mathcal{P}$ to avoid the overhead of searching the entire allocation space.

Delayed restart. Unlike the *greedy heuristic planner* of hyperparameter tuning, which plans resource allocation offline and pre-warms new functions before each stage starts, the resource scheduling for model training is performed at runtime. We need to hide the overhead of resource adjustment. Thus, we propose a method for fast resource adjustment, as shown in Fig. 8. If the scheduler decides to adjust resources at the end of epoch $k-1$, we start new functions during epoch k. Furthermore, we overlap the process of parameter synchronization, i.e., old functions are terminated after uploading the gradient data, and the new model parameters are pulled by the new functions directly. Based on Section III-B, we can easily calculate the optimal launch time for new function startup.

IV. Evaluation

We evaluate CE-scaling with a variety of ML models, and compare it to the latest allocation methods.

A. Experiment Setup

ML Models:– *Logistic Regression* (LR) is a linear model for classification. The number of the model parameters is equal to that of input features.
– *SVM* is a supervised learning model for classification analysis. The size of the model parameters is several KB.
– *MobileNet* (MN) is a popular lightweight NN model for image classification. The size of model parameters is 12MB.
– *ResNet50* (RN) is a neural network model for image classification. The size of model parameters is 89MB.
– *BERT-base* (Bert) is a transformer-based machine learning technique for *natural language processing* (NLP), developed by Google. The size of model parameters is 340MB.

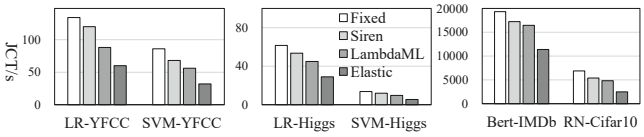


Fig. 9: Execution time of hyperparameter tuning given a budget. For hyperparameter tuning, JCT is the time from the start until the optimal trial is found.

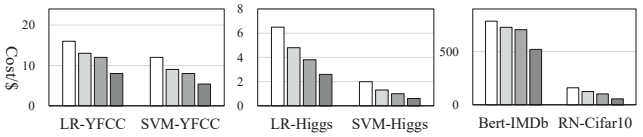


Fig. 10: Cost of hyperparameter tuning given a QoS constraint. Here we count the cost of all trials (training jobs with different hyperparameter configurations).

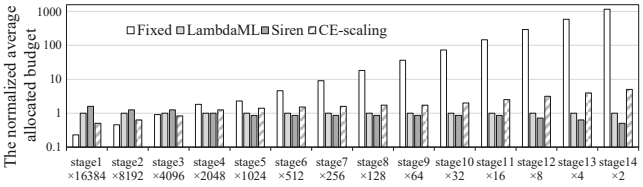


Fig. 11: The normalized average allocated resource (budget) for each trial in each stage for LR-Higgs. We mark the number of trials in each stage on the horizontal axis. The cost is normalized based on the static method (LambdaML) in each trial.

Datasets: – *Higgs* is a dataset for binary classification, produced using Monte Carlo simulations. Higgs contains 11 million instances, and each instance has 28 features.
– *YFCC100* is a dataset from Yahoo [26] in which each data point represents one image with several label tags and a feature vector of 4096 dimensions.
– *Cifar10* is an image dataset that consists of 60000 32×32 images categorized in 10 classes.
– *IMDb* is a standard text classification dataset that consists of 25,000 sentences. The average length of sentences is 292.
Baselines: – *LambdaML* [14] is a state-of-the-art serverless ML system implemented on AWS lambda.
– *Cirrus* [4] is a serverless ML workflow framework that aims to support and simplify the end-to-end ML user workflow. Cirrus uses EC2 VM as an external storage.
– *Siren* [9] is an asynchronous distributed ML framework based on serverless computing. It determines the number and memory size of functions through deep reinforcement learning.

Implementation. CE-scaling is implemented on top of Lambda with almost 6000 LoC (mainly in python). It consists of three modulers: *Pareto profiler*, *Greedy heuristic planner*, and *online scheduler*. CE-scaling outputs a configuration file in JSON, which is then used by Lambda to invoke functions and select external storage services.

B. Performance of Hyperparameter Tuning

We compare CE-scaling against static and cluster-based methods for hyperparameter tuning. Static methods include LambdaML and Siren. They are implemented by removing

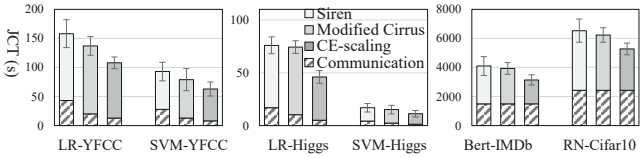


Fig. 12: Execution time of model training given a budget. For model training, JCT is the time from the start until the model converges to the target loss. The bottom of each bar – with pattern – indicates the overhead of communication. Note that the JCT includes the scheduling overhead.

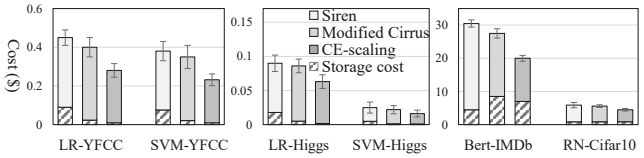


Fig. 13: Cost of model training given a QoS constraint. The bottom of each bar – with pattern – indicates the cost of storage.

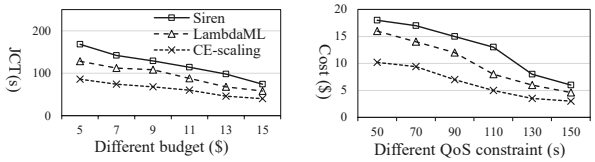
CE-scaling's *Greedy heuristic scheduler* (described in section III-C). The cluster-based method (Fixed) divides resources equally among stages and across trials in each stage. All experiments are performed on SHA-generated specifications [10]. We select the optimal hyperparameter configuration among 16384 trials with a reduction factor of 2. There are 14 stages in total, and 2 epochs are assigned in each stage.

As shown in Fig. 9 and 10, CE-scaling achieves lower JCT and cost compared to static methods. For a given budget, CE-sclaing reduces the JCT by up to 66%. For a given QoS constraint, CE-scaling achieves up to 42% cost reduction. The improvement in both JCT and cost is larger for large models (e.g., BERT-base and ResNet50). This is because large model jobs are more sensitive to resources, and unreasonable resource partitioning can seriously affect the JCT and cost. As expected, the fixed method has the worst JCT and cost, because it leads to serious resource competition in early stages, and the budget is wasted by the communication overhead in later stages. Moreover, LambdaML performs better than Siren. The reason is that Siren's reinforcement learning model tends to allocate more resources in the early stages, which leads to more resources wasted on trials that will be terminated early.

Fig. 11 shows the average allocated resources for each trial in each stage for LR-Higgs. Considering that early stages have more terminated trials, CE-scaling allocates less resources to early stages compared to static methods, and leaves more resources to later stages. Static methods treat each stage fairly and therefore give more resources to the trials in early stages, resulting in more than 80% of the resources consumed in the first two stages. Finally, for the *fixed* method, a large number of trials share less than 10% of the budget in early stages, resulting in serious resource competition.

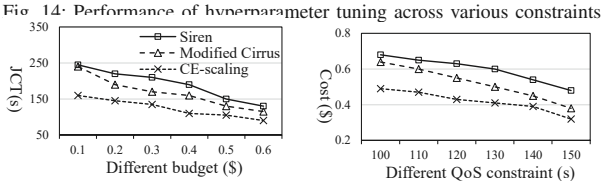
TABLE IV: Experimental configurations of different models

Model	Dataset	Batch size	Learning rate	Target loss
LR/SVM	Higgs	10k	0.01	0.66/0.48
LR/SVM	YFCC	800	0.01	50
MobileNet	Cifar10	128	0.01	0.2
ResNet50	Cifar10	32	0.01	0.4
BERT-base	IMDb	32	0.00005	0.6



(a) JCT across various budget (b) Cost across various QoS

Fig. 14: Performance of hyperparameter tuning across various constraints



(a) JCT across various budget (b) Cost across various QoS

Fig. 15: Performance of model training across various constraints

C. Performance of Model Training

We compare CE-scaling against three baselines for model training. Siren leverages reinforcement learning to provide resource allocation plan. Furthermore, we modify Cirrus to realize the same online prediction as CE-scaling. LambdaML is not included, because the offline prediction always results in violations in the constraints. We stop training when the training loss reaches an objective value. These experiment configurations are listed in Table IV. The experimental results are the average values of ten runs.

CE-scaling provides more appropriate and efficient resource allocation for model training jobs at runtime, thanks to online prediction. It can also select proper external storage services along with other resources. As shown in Fig. 12 and 13, CE-scaling achieves shorter JCT and lower cost compared to baselines. For a given budget, CE-scaling reduces JCT by up to 56%. For a given QoS constraint, CE-scaling achieves up to 35% cost reduction. We further break down the impact of external storage. The bottom of each bar – with pattern – indicates the communication overhead in Fig. 12 and represents the storage cost in Fig. 13. Siren can meet the constraints, but the external storage of Siren (S3) causes high synchronization overhead and increases the cost of running functions, especially for BERT-base and ResNet50. Moreover, Siren adjusts resources every epoch, which causes considerable overhead as we discuss in section IV-G. Modified Cirrus can improve synchronization time but at high monetary cost because of the unreasonable external storage and function restart overhead.

D. CE-scaling under Various Constraints

We evaluate the performance of CE-scaling with different QoS and cost constraints. We focus on the experiments of hyperparameter tuning and model training of LR-YFCC. As shown in Fig. 14 and 15, CE-scaling results in shorter JCT and lower cost under different budget and QoS constraints for both hyperparameter tuning and training. We observe that under tight QoS and cost constraints, the performance gap between CE-scaling and baseline methods is higher. When the constraints are relaxed, the performance advantage of CE-scaling becomes small, because baselines have sufficient budget or QoS.

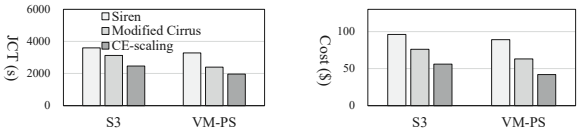


Fig. 16: Performance and cost of CE-scaling, Siren, and Cirrus under the same external storage (i.e., S3 and VM-PS) for hyperparameter tuning

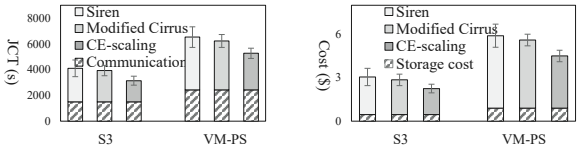


Fig. 17: Performance and cost of CE-scaling, Siren, and Cirrus under the same external storage (i.e., S3 and VM-PS) for model training. The bottom of each bar – with pattern – indicates the impact of storage.

E. The Effectiveness of CE-scaling Internals

1) ***CE-scaling and baselines under same external storage.*** We evaluate CE-scaling and baselines under the same external storage (i.e., S3 and VM-PS) with MobileNet on Cifar dataset. As shown in Fig. 16, for hyperparameter tuning, CE-scaling still outperforms Siren and Cirrus, in terms of JCT and cost, even when they all use high-performance external storage (i.e., VM-PS). The reason is that CE-scaling allocates the "exact" resources needed by functions in each stage. Fig. 17 shows the JCT and cost for model training. CE-scaling obtains the lowest JCT and cost under both storage services. This demonstrates the effectiveness of CE-scaling in adaptively adjusting the number of functions and memory sizes; and the efficiency of the delayed restart in sustaining low overhead.

2) ***CE-scaling under different external storage.*** Fig. 18 shows the performance and cost of CE-scaling when restricting CE-scaling to use only one type of external storage: DynamoDB, S3, Elasticache, or VM-PS. We train two different models: LR on Higgs dataset and MobileNet on Cifar10. We have the following observations. First, JCT and the cost vary across different external storage services. Second, as discussed in Section II, Elasticache and VM-PS do not always lead to best performance and lowest cost, and that the selection of best external storage service strongly depends on the ML models. DynamoDB achieves the best trade-off between performance and cost for LR, while Elasticache obtains the best performance with the lowest cost for MobileNet. Third, the performance of the external storage services may impact the computation time of the training. Fourth and importantly, achieving lower computation time for the training model by considering only the number functions and the size of memory may not result in best overall JCT or cost. Hence, this shows the importance of our work in jointly considering the allocation of function, memory, and external storage to navigate the cost-performance trade-off for different models.

F. Validation of the Analytical Models

To validate the correctness of our analytical models, we compare the estimated JCT and cost when using our models to the actual runtime and billing data observed by Amazon

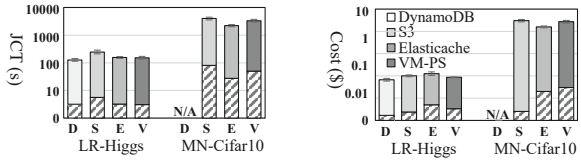


Fig. 18: CE-scaling under fixed external storage for model training. D, S, E, and V denote DynamoDB, S3, Elasticache, and VM-PS. N/A as the model size exceeds DynamoDB's object size limit. The bottom of each bar – with pattern – indicates the impact of storage.

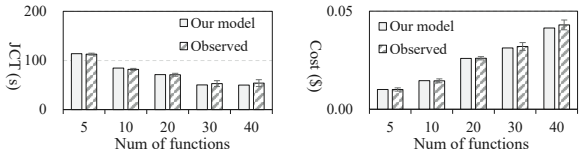


Fig. 19: JCT and cost of model training estimated by our model with different number of functions

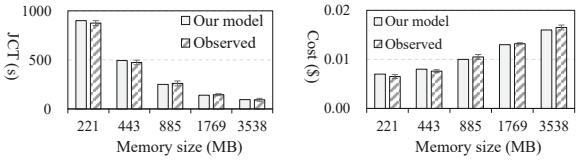


Fig. 20: JCT and cost of model training estimated by our model with different memory sizes

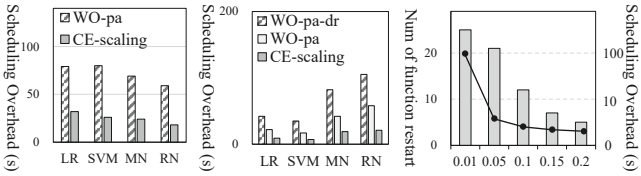


Fig. 21: Scheduling overhead

CloudWatch tool [27]. We train a LR model on Higgs dataset while varying the number of functions and memory sizes, and use S3 as the external storage. As shown in Fig. 19, when fixing memory size to 1769 MB, our model can accurately estimate training time with an error of 0.56-4.9 percent, and estimate training cost with an error of 0.2-3.72 percent. The largest error is observed when the number of functions is set to 40. This is caused by the high overhead of parameter synchronization due to network instability. When fixing the number of functions to 10, as shown in Fig. 20, our model can accurately estimate training time with an error of 2.1-4.3 percent, and estimate cost with an error of 1.5-7.6 percent. This also implies that our model is sensitive to memory size.

G. Scheduling Overhead

For CE-scaling, the scheduling overhead is significantly reduced to seconds by reducing search space and overlapping the process of restarting functions. The scheduling overhead of hyperparameter tuning comes from the allocation search in the planning of resource partitioning. As shown in Fig. 20(a), we compare original CE-scaling to CE-scaling without Pareto optimization (WO-pa). By reducing search space with Pareto boundary, scheduling overhead of hyperparameter tuning is reduced by 69% on average. For model training, as the scheduling overhead comes from resource adjustment and allocation search, we compare the original CE-scaling with CE-scaling without Pareto (WO-pa) and CE-scaling without Pareto and delayed restart (WO-pa-dr). As shown in Fig. 20(b), with the delayed restart optimization, the scheduling overhead is reduced by 55% compared to WO-pa-dr. With the pareto optimization, the scheduling overhead is reduced by 64% compared to WO-pa. The average scheduling overhead is only a few seconds per epoch, which is acceptable compared to the second-level cold start overhead of functions. *Note that all experimental results in our previous sections include the scheduling overhead.*

The impact of δ. δ guides how to adaptively adjust resource allocation based on the latest prediction results. Specifically, it helps to avoid excessive switch in resource allocations (i.e., the frequency of restarting functions) when the online prediction results change. Thus, we adjust resource allocation only if the difference between latest and previous prediction is larger than this threshold. We study its impact by varying δ from 0.01, 0.05, 0.1, 0.15, to 0.2. Note that a value higher than 0.2 will not respond to the latest prediction on time and lead to early termination. As shown in Fig. 21(c), a higher value leads to slow response to changes in predication results, thus the number of function restarts is small. On the other hand, a low value causes frequent restarting of functions, resulting in serious scheduling overhead. By default, δ is set to 0.1.

V. Related Works

Serverless ML framework. Previous works propose frameworks for orchestrating distributed ML workflows on serverless platforms. Pyren [2] performs ML training jobs as MapReduce tasks on serverless clouds. Developers have to refactor their existing models with PyWren's MapReduce-like APIs. Cirrus [4] presents a prototype to execute machine learning workflows with lambda functions and uses EC2 as intermediate data storage. LambdaML [14] comparatively analyzes the difference between the deployment of machine learning jobs on FaaS and IaaS. However, these systems do not consider the cost and efficiency of deployment.

Cost/performance optimization for serverless applications. There have been several studies devoted to resource provisioning for serverless applications. Astra [7] focuses on MapReduce workflows and estimates resource allocation by modeling the job completion time and monetary cost of the jobs. Siren [9] targets ML workflows and tries to optimize the deployment cost of serverless ML workflows using deep reinforcement learning. However, Siren focuses on asynchronous training, and does not consider the impact of external storage. λDNN [5] builds a lightweight analytical performance model to allocate resources for DNN training. However, users need to specify the number of epochs in advance for λDNN. Moreover, all above solutions do not consider hyperparameter tuning.

Cost/performance optimization for ML workflows on clouds. Recent efforts have studied how to efficiently allocate resources on cloud for ML workflows. SLAQ [17] is a cluster scheduler for ML training jobs that aims to maximizes overall

job quality. SLAQ maximizes system-wide quality improvement across multiple jobs. Optimus [16] utilizes online prediction to estimate the performance of training considering the number of allocated containers. They schedule tasks to workers to improve the performance and reduce the communication cost. In contrast, CE-scaling focuses on the trade-off between performance and cost when selecting the number of functions, their memory sizes, and storage services. In addition, CE-scaling realizes efficient resource allocation from a novel serverless computing perspective by targeting low scheduling and resource switching overheads. RubberBand [10] is the first framework for elastic execution of hyperparameter tuning in the cloud. RubberBand focuses on reducing the resource competition by allocating more resources (in terms of number of VMs) to the early stages as the degree of parallelism is higher in early stages. In contrast, the resource partitioning in CE-scaling goes one level deeper and allocates resources to stages according to their exact usage.

VI. Conclusion

In this paper, we propose a dynamic resource allocation framework for serverless ML workflows, named CE-scaling, to reduce the cost for users and improve performance. First, we implement an exact resource partitioning across stages in hyperparameter tuning, and design a greedy heuristic algorithm to find the optimal partitioning plan. Second, we adaptively allocate resources in model training by utilizing online prediction. Third, in order to reduce the overhead of searching the whole allocation space, we use Pareto boundary to prune out bad allocations. We implement CE-scaling in AWS Lambda and compare it to the state-of-the-art methods. CE-scaling improves the performance and saves cost by up to 63% and 41% for hyperparameter tuning, respectively; and by up to 58% and 38% for model training.

Acknowledgment

This work was supported in part by the National Key Research and Development Program of China under grant 2022YFB4500704 and in part by the National Science Foundation of China under grants 62032008 and 62232011. Hao Fan is the corresponding author.

References

[1] E. Oakes, L. Yang, D. Zhou, and Houck, "SOCK: rapid task provisioning with serverless-optimized containers," in *Proceedings of the USENIX Annual Technical Conference*, 2018, pp. 57–70.

[2] E. Jonas, Q. Pu, S. Venkataraman, I. Stoica, and B. Recht, "Occupy the cloud: distributed computing for the 99%," in *Proceedings of the ACM Symposium on Cloud Computing*, 2017, pp. 445–451.

[3] Gartner, 2019. [Online]. Available: https://www.gartner.com/smarterwithgartner/gartner-predicts-the-future-of-ai-technologies

[4] J. Carreira, P. Fonseca, A. Tumanov, A. Zhang, and R. Katz, "Cirrus: a serverless framework for end-to-end ML workflows," in *Proceedings of the ACM Symposium on Cloud Computing*, 2019, pp. 13–24.

[5] F. Xu, Y. Qin, L. Chen, Z. Zhou, and F. Liu, "λDNN: achieving predictable distributed DNN training with serverless architectures," *IEEE Transactions on Computers*, vol. 71, no. 2, pp. 450–463, 2021.

[6] M. Abadi, P. Barham, J. Chen, Z. Chen, A. Davis, J. Dean, M. Devin, S. Ghemawat, G. Irving, and M. Isard, "Tensorflow: a system for large-scale machine learning," in *Proceedings of the USENIX Conference on Operating Systems Design and Implementation*, 2016, pp. 265–283.

[7] J. Jarachanthan, L. Chen, F. Xu, and B. Li, "Astra: autonomous serverless analytics with cost-efficiency and QoS-awareness," in *Proceedings of the IEEE International Parallel and Distributed Processing Symposium*, 2021, pp. 756–765.

[8] V. Shankar, K. Krauth, K. Vodrahalli, Q. Pu, B. Recht, I. Stoica, J. Ragan-Kelley, E. Jonas, and S. Venkataraman, "Serverless linear algebra," in *Proceedings of the ACM Symposium on Cloud Computing*, 2020, pp. 281–295.

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

[10] U. Misra, R. Liaw, L. Dunlap, R. Bhardwaj, K. Kandasamy, J. E. Gonzalez, I. Stoica, and A. Tumanov, "Rubberband: cloud-based hyperparameter tuning," in *Proceedings of the European Conference on Computer Systems*, 2021, pp. 327–342.

[11] Z. Kaoudi, J.-A. Quiané-Ruiz, S. Thirumuruganathan, S. Chawla, and D. Agrawal, "A cost-based optimizer for gradient descent optimization," in *Proceedings of the ACM International Conference on Management of Data*, 2017, pp. 977–992.

[12] AWS S3. [Online]. Available: https://aws.amazon.com/s3/

[13] DynamoDB. [Online]. Available: https://aws.amazon.com/dynamodb/

[14] J. Jiang, S. Gan, Y. Liu, F. Wang, G. Alonso, A. Klimovic, A. Singla, W. Wu, and C. Zhang, "Towards demystifying serverless machine learning training," in *Proceedings of the ACM International Conference on Management of Data*, 2021, pp. 857–871.

[15] R. Liaw, R. Bhardwaj, L. Dunlap, Y. Zou, J. E. Gonzalez, I. Stoica, and A. Tumanov, "Hypersched: dynamic resource reallocation for model development on a deadline," in *Proceedings of the ACM Symposium on Cloud Computing*, 2019, pp. 61–73.

[16] Y. Peng, Y. Bao, Y. Chen, C. Wu, and C. Guo, "Optimus: an efficient dynamic resource scheduler for deep learning clusters," in *Proceedings of the European Conference on Computer Systems*, 2018, pp. 1–14.

[17] H. Zhang, L. Stafman, A. Or, and M. J. Freedman, "Slaq: quality-driven scheduling for distributed machine learning," in *Proceedings of the ACM Symposium on Cloud Computing*, 2017, pp. 390–404.

[18] AWS lambda. [Online]. Available: https://aws.amazon.com/lambda/

[19] L. Li, K. Jamieson, A. Rostamizadeh, E. Gonina, J. Ben-Tzur, M. Hardt, B. Recht, and A. Talwalkar, "A system for massively parallel hyperparameter tuning," in *Proceedings of the Conference on Machine Learning and Systems*, 2020, pp. 230–246.

[20] S. Falkner, A. Klein, and F. Hutter, "Bohb: robust and efficient hyperparameter optimization at scale," in *Proceedings of the International Conference on Machine Learning*, 2018, pp. 1437–1446.

[21] AWS Lambda Pricing. [Online]. Available: https://aws.amazon.com/ec2/pricing/ondemand/

[22] A. Klimovic, Y. Wang, P. Stuedi, A. Trivedi, J. Pfefferle, and C. Kozyrakis, "Pocket: elastic ephemeral storage for serverless analytics." in *Proceedings of the USENIX Conference on Operating Systems Design and Implementation*, 2018, pp. 427–444.

[23] ElastiCache. [Online]. Available: https://aws.amazon.com/elasticache/

[24] Z. Wen, Y. Wang, and F. Liu, "Stepconf: Slo-aware dynamic resource configuration for serverless function workflows," in *Proceedings of the IEEE Conference on Computer Communications*, 2022, pp. 1868–1877.

[25] AWS Lambda Limitations. [Online]. Available: https://docs.aws.amazon.com/lambda/latest/dg/gettingstarted-limits.html

[26] YFCC. [Online]. Available: http://projects.dfki.uni-kl.de/yfcc/

[27] Amazon CloudWatch. [Online]. Available: https://docs.aws.amazon.com/AmazonCloudWatch/latest/monitoring/WhatIsCloudWatch.html

[28] openwhisk. [Online]. Available: https://console.bluemix.net/openwhisk/

[29] Azure Functions. [Online]. Available: https://azure.microsoft.com/en-us/services/functions/

[30] H. Zhang, Y. Tang, A. Khandelwal, J. Chen, and I. Stoica, "Caerus: NIMBLE task scheduling for serverless analytics," in *Proceedings of the USENIX Symposium on Networked Systems Design and Implementation*, 2021, pp. 265–283.

[31] M. Zinkevich, M. Weimer, L. Li, and A. Smola, "Parallelized stochastic gradient descent," in *Proceedings of the Annual Conference on Neural Information Processing Systems Systems*, 2010, pp. 23–34.

[32] Z. Zhang, J. Jiang, W. Wu, C. Zhang, L. Yu, and B. Cui, "Mllib*: fast training of glms using spark mllib," in *Proceedings of the IEEE International Conference on Data Engineering*, 2019, pp. 1778–1789.