SandPiper: A Cost-Efficient Adaptive Framework for Online Recommender Systems

Prashanth Thinakaran*, Kanak Mahadik†, Jashwant Gunasekaran†, Mahmut Taylan Kandemir*, Chita R. Das*
*The Pennsylvania State University †Adobe Research

2022 IEEE International Conference on Big Data (Big Data) | 978-1-6654-8045-1/22/$31.00 ©2022 IEEE | DOI: 10.1109/BigData55660.2022.10020465

***Abstract*—Online recommender systems have proven to have ubiquitous applications in various domains. To provide accurate recommendations in real time it is imperative to constantly train and deploy models with the latest data samples. This retraining involves adjusting the model weights by incorporating newly-arrived streaming data into the model to bridge the accuracy gap. To provision resources for the retraining, typically the compute is hosted on VMs, however, due to the dynamic nature of the data arrival patterns, stateless functions would be an ideal alternative over VMs, as they can instantaneously scale on demand. However, it is non-trivial to statically configure the stateless functions because the model retraining exhibits varying resource needs during different phases of retraining. Therefore, it is crucial to dynamically configure the functions to meet the resource requirements, while bridging the accuracy gap. In this paper, we propose *Sandpiper*, an adaptive framework that leverages stateless functions to deliver accurate predictions at low cost for online recommender systems. The three main ideas in *Sandpiper* are (i) we design a data-drift monitor that automatically triggers model retraining at required time intervals to bridge the accuracy gap due to incoming data drifts; (ii) we develop an online configuration model that selects the appropriate function configurations while maintaining the model serving accuracy within the latency and cost budget; and (iii) we propose a dynamic synchronization policy for stateless functions to speed up the distributed model retraining leading to cloud cost minimization. A prototype implementation on AWS shows that *Sandpiper* maintains the average accuracy above 90%, while 3.8× less expensive than the traditional VM-based schemes.**

***Index Terms*—Serverless, Systems-for-ML, Continual Learning**

I. Introduction

Online machine learning (ML) has transformed the way recommendations are delivered especially in the areas such as cybersecurity, e-commerce, e-health, media analytics, etc,. The majority of these ML algorithms are deployed in a dynamic setting, where the streaming data trend continuously changes over time [1], and as a result, these models need to be continuously updated with the new data. The impact of serving recommendations using an inaccurate model can lead to severe performance and cost implications [2]. Continuous updates through online learning provide these models the power to adapt to the changes in user preferences or data patterns, also referred to as *data drifts*. These drifts require *model retraining* to incorporate the fresh data samples. Therefore, online learning systems incorporate the training data samples at regular intervals, where the frequency of retraining is critical in serving high-quality recommendations. Typically the frequency of data drifts was in the order of days, with the recent advancements in data integration and the velocity of data in ML training. The increase in the new data generated in turn leads to frequent changes in user preferences and data trends, thus demanding much more frequent retraining of the models used in recommender systems, compared to the past.

Model serving of these recommender systems are usually hosted on a public cloud such as AWS. A key challenge lies in *resource procurement and provisioning adapting to the dynamic nature of model retraining* without affecting the model quality. Therefore, these retraining tasks have to meet tighter deadlines to bridge the accuracy gap. Often, to guarantee the service level objectives (SLO) of these tasks to meet the retraining deadline, compute resources are *over-provisioned*, especially in the case of Infrastructure-as-a-Service (IaaS or VMs) [3], leading to substantial wastage in resources and massive provisioning costs. This motivates the primary proposition of our work: *In contrast to VMs, can the serverless functions help alleviate the over-provisioning costs?*

The resource requirements for incremental model training with data drifts are significantly lower than that of offline training [4]. This inherent nature of resource requirements when combined with the relatively smaller datasets only requires light-weight model updates, which in turn make these online retraining tasks a naturally convincing fit to the serverless function-based computing model. Depending on the data arrival trends and drifts the resource requirements for these model updates are unpredictable and too slow to react to auto-scaling of VMs leading to variable training latencies. While prior works [3], [5] propose schemes to determine the frequency of training, they are limited to using VMs, which are slow in reacting to the actual resource requirement along with huge retraining costs.

Although stateless functions have been exploited for deploying traditional ML training [6], there are several unexplored challenges in the context of instant retraining for recommender systems. First, along with designing each training epoch as a stateless workflow, it is necessary to examine the accuracy trade-offs, ML-retraining update latency, and cost-effectiveness for varying data arrival trends. The recommender accuracy depends on the time to trigger retraining, whereas the retraining latency and cost depend on the number of cloud functions invocations. Second, different online algorithms have distinct scaling characteristics. For example, applications that execute topic modeling [7] and matrix factorization [8] have different scaling behaviors based on the number of topics

modeled and incoming users, respectively. T
challenges collectively instigates the centi
work: *how to solve the complex optimi
designing low-cost stateless training, with
low latency for time-varying resource dema
we design *Sandpiper* which leverages serv
develop a cost-efficient online continuous le

Sandpiper adopts a four-pronged appr
optimization problem. First, it uses a da
automatically trigger model retraining in
recommender accuracy. Second, it integrat
formance model in its core design, whic
timizing three objective metrics – *model
model accuracy*, and *retraining cost*. Thes
turn, depend on five cloud task configuration parameters (mini-batch size, number of epochs, function memory, arrival rate, and synchronization ratio) for each retraining epoch. We judiciously consider these configuration parameters in a cohesive fashion towards designing and implementing *Sandpiper*. Third, it models it as an online optimization problem, which leverages Gaussian Process (GP-OPT) approach to effectively predict the task completion times. Finally, to reduce the synchronization overheads caused by straggler workers, *Sandpiper* employs a novel selective synchronization policy, under which the system will continue to launch worker functions for subsequent epochs with partial updates, leading to faster convergence. Our extensive evaluations on AWS Lambda platform reveal that *Sandpiper* performs model updates 3.8× quicker and 1.7× cheaper than IaaS-based recommender systems.

II. Background and Motivation

We start with providing a brief overview of online continual learning, followed by a detailed analysis of its performance to motivate the need for *Sandpiper*.

A. Continuous Leaning-Based Frameworks

Online ML models take input from a series of data and serve recommendations on them. The accuracy of the model serving the requests depends on how quickly the model learns and adapts to the dynamic data arrival patterns $S_i = \{s_1, s_2, ..s_i\}$ in real-time. Therefore, the model is continuously retrained by adjusting the weights with freshly arrived data samples, with the primary objective of minimizing the loss function, where the loss function is defined as the error in prediction. The loss function ϕ_{t+1} at time $(t+1)$ for a series of X data samples arrived at time t is given by 1:

$$\phi_{t+1} \Leftarrow f(\phi_t, s_t) \Leftarrow \underbrace{\frac{1}{X}\sum_{n=1}^{X} f(\phi_t, s_t)}_{minimize}. \quad (1)$$

Continuous learning and model serving system enables retraining by incorporating fresh updates to the online model. As seen in Figure 1, the data from various client-ends ❶ arrive at an API gateway, which serves as a front end to the continuous learning ML models. The API gateway ❷ performs auxiliary functions such as data schema and format conversion

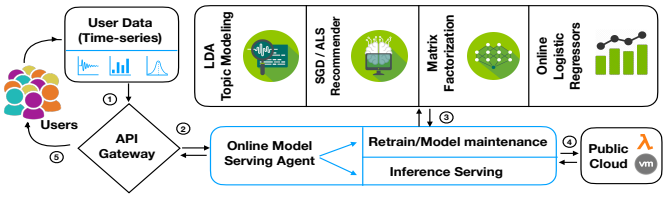


Fig. 1: Continuous learning system deployed across various public cloud offerings such as VMs and Serverless.

based on the training requirements. Subsequently, the data updates are stored in a data-store, while the model serving agent triggers ❸ the appropriate ML retraining service. This retraining is performed on a public cloud setting ❹ where the continuously-updated model can be simultaneously used to serve inference requests from different users. The inference requests from the trained model are served back to the user ❺.

The continuous model training could be performed either on IaaS-based stateful platforms or stateless compute offerings such as AWS Lambda [9]. IaaS provides the infrastructure in the form of VMs or containers by multiplexing them over physical machines. In contrast, the serverless architecture provides the abstraction of stateless functions packaged in their own runtime and programming language environment. However, stateless functions offer a unique proposition of being event-driven and pay-as-you-go model, which is directly relevant for online retraining since the model retrains on a discreet set of events such as data drifts. There have been a handful of studies in the past evaluating the cost and resource efficiency of leveraging serverless offerings [10] when compared to traditional IaaS offerings. The fundamental difference in resource usage efficiency and cost is due to the fact that an external auto-scaling policy provisions for resource capacity in the case of IaaS, whereas serverless functions are event-driven.

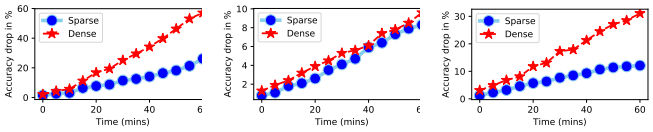
B. Characterizing Online Learning Models

We studied four popular online machine learning models used in three different applications (Table I), to understand the resource consumption footprints in continuous model training for cost-efficient deployment on public clouds.

Model	Dataset	Framework	Application
LDA [11]	Meme [12]	Mallet [13]	Topic Modeling
SGD (Fact)	Movielens [14]	sklearn [15]	Movie Recommendation
Mat (ALS)	Movielens [14]	sklearn [15]	Movie Recommendation
Logistic Reg	Boston [16]	sklearn [15]	Housing Value Prediction

TABLE I: Candidate ML models used for continuous training.

Towards this, we start with a base model trained by batching the first five minutes of data updates and gradually retraining on further data updates based on a Poisson distribution, which is representative of production web server data arrivals. We profile four online ML models listed in Table I for different runtime utilization metrics such as cost, CPU, memory, disk, and network consumption. Serverless functions are configured and billed based on their compute and memory requirements. Latent Dirichlet Allocation (LDA) performs online topic modeling on web page contents crawled over the internet using the memetracker dataset [12]. We used the Mallet tool to persist intermediate models and perform constant model retraining to



(a) Mat Factorization **(b)** Log Regression **(c)** Latent Dirichlet

Fig. 2: Accuracy loss due to drifts in case of Online ML models not being continuously updated/retrained for period of 1hr.

maintain high accuracy. On the other hand, we analyzed matrix factorization based on Alternating Least Squares (ALS) and Stochastic Gradient Descent (SGD), which serve recommendations of movies to the users based on the reviews from the Movielens dataset [14].

Figure 2 plots the accuracy loss due to model decay in case of dense and sparse data arrivals for a period of 1 hour. The data arrivals are statically generated using Poisson distribution, where sparse is data arrival of less than 150 data samples per second while the dense arrivals range from 150 to 400 data samples per second. The accuracy loss is measured by the gap in recommendation quality from a stale model compared to a constantly retrained model in which the data arrivals are incorporated through constant retraining. Different machine learning models decay over time at different rates due to the gradient staleness caused by fresh updates. Models like MF and LDA are very sensitive to these updates as shown in Figure 2c – in the case of LDA, it can get up to 60% drop in accuracy due to the lack of continuous retraining. On the other hand, logistic regression suffers less than 10% accuracy drop due to this stateless nature as the retraining happens on discreet data samples. Therefore, the accuracy of a recommender system for some models is heavily reliant on monitoring these drifts and incorporating the fresh data samples in real-time.

Why do drifts occur? Online models need to be constantly updated by incorporating fresh data points. The frequency at which the model needs retraining is determined by drop in recommender accuracy due to model decay. The two reasons why a model can decay are *data drift* and *concept drift*. In the case of data drifts, the pattern of data rapidly changes with time potentially introducing previously unseen trends, thus resulting in loss of accuracy and demanding to retrain. Note that there is no impact on previously labeled data with changes to the weights. On the other hand, with concept drift, the existing model's interpretation of the data itself changes with time, which can be addressed by either changing the model architecture or using multiple models like model ensembling [17]. This is orthogonal to retraining the same model over time. Out of these two, data drifts are more predominant in production systems and in this work, we focus on adaptive exclusively retraining for data drifts.

Further, for retraining, it is important to bridge the data drifts in real-time as it plays a crucial factor in determining the recommender accuracy. In some cases like matrix factorization using SGD, these drifts cannot be bridged using serial retraining, thereby requiring a parameter-server style distributed retraining. The runtime system should be able to learn the actual resource requirements for model retraining to pick a configuration that can satisfy both the retraining latency and the deployment cost while maintaining the overall accuracy.

III. Prelude to Sandpiper

A. Comparing Cost of Retraining

We analyze the cost of retraining using data updates modeled after prominent traces such as Wikipedia [18], Twitter [19], Berkeley [20], and WITS [21]. This aids in better understanding of the model decay rate with respect to the real-world data update frequency.

1) Trace Analysis. We consider a two hour portion of data updates of these traces for our studies. The breakdown of DUPS is given in Table II. As shown in Figure 3, the WITS and Twitter traces (Figure 3c and 3d) have large variations in peaks compared to the Wiki and Berkeley traces (Figure 3a and 3b). The data updates arrive every second over the entire trace duration. These traces exhibit highly dynamic loads resembling the data arrival scenario in the case of production recommender systems. Some portions of the Twitter trace have recurring patterns (e.g., minute of the hour), whereas the WITS trace contains unpredictable load spikes (e.g., flash-crowd scenario). To demonstrate the cost difference between the stateful and stateless offerings, we run retraining loops for the learning models listed in Table I. In the case of VMs, we use c5.large on-demand instance and for Lambdas we configure the function memory size to be 512Mb in serverless-static. However, in the case of serverless-dynamic, we vary the function memory based on the size of the data update. Based on the arrivals rates from the datacenter traces shown in Figure 3, we launch retraining tasks to avoid model quality deterioration. Also, for VMs, we conservatively ignore the provisioning latencies.

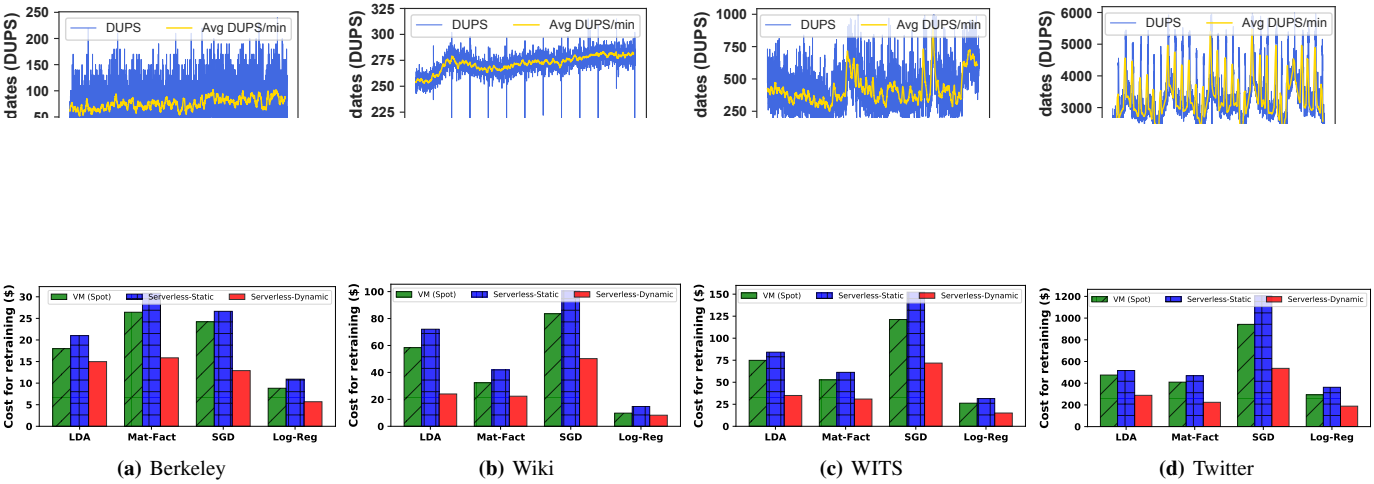
Trace	Min	Max	Avg
WITS	90	2335	423.45
Berkeley	10	240	74.95
Twitter	1214	6978	3346.76
Wiki	218	331	279.68

TABLE II: Breakdown of data updates per second (DUPS).

2) Cost of IaaS* vs *Serverless. We plot the cost for retraining in Figure 4 for the different traces. Serverless functions are billed based on memory-requirement of the function over time. In general, across all the four models, naively using the serverless functions with fixed memory (3GB) is always more expensive than the VMs of same size. For example, in the case of Figure 4c, to run SGD on AWS Lambda function is 18% expensive than running on VM. This is due to the fact that the cost per GB-s is high in the case of Lambda when it is statically provisioned when compared to VMs. However, if these Lambda functions are right-sized in terms of memory, they turn out to be 10-40% cheaper compared to VMs. This trend is common across all trace-based model retraining. Therefore, *it is important to right-size the Lambda functions for cost efficiency.*

B. On-Demand Distributed ML-Retraining

Besides cost, the online models pose a strict deadline (QoS) requirement for their retraining jobs. For instance, a retraining



(a) Berkeley **(b)** Wiki **(c)** WITS **(d)** Twitter

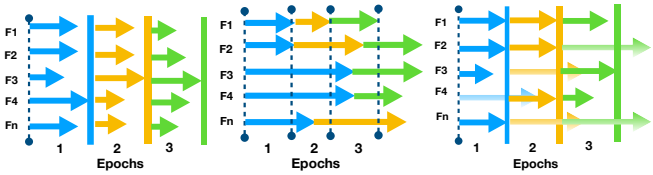
Fig. 4: Cost to retrain online-ML models across different cloud offerings based on data updates from traces.

worker function set should lead to model convergence within tens of seconds. This is due to the fact that the delays in retraining would serve inaccurate recommendations. Hence, depending on the volume of data arrivals, these machine learning models need to be trained in a distributed fashion. Further, in the case of distributed learners the model the straggler workers can result in increased retraining latency due to the synchronization delays cascading as SLO violations. Therefore, it is important to address this issues through an appropriate *synchronization policy* across distributed learners.

Typically, depending on the size of the data sample s_t, the framework decides to launch retraining tasks by breaking s_t in to X minibatches and repeat the training for multiple epochs till the loss function is minimized. In the case of distributed retraining, each worker function works on its mini-batch of data and performs updates on the model parameters for that portion of its mini-batch. The next round of worker function set is launched only after all the worker functions of the previous epoch finish their model parameter sync.

Limitations of Fully Synchronous Models: Due to the unique demands of our problem space, we need to explore every factor that impacts the latency of model retraining. The time per epoch grows with the number of learners due to straggling learners that slow down randomly. To understand this, we take an example in the case of distributed stochastic gradient descent (DSGD), where the synchronization between epochs can be done in multiple ways, as seen in Figure 5. Parameter Server, as shown in Figure 5a, waits for all learners to push gradients before it updates the model parameters and launches the workers for the subsequent epoch leading to long tail latency due to straggling workers. On the other hand, an asynchronous policy could be adopted as shown in recent works [22], trading off stale parameters for faster training time as it leads to convergence faster, as can be observed in Figure 5b. However, aggressively launching workers with stale weights can impact the recommender accuracy.

Advantages of Selective Synchronous: Recent studies [23] have found that a balance can be struck between the synchronous and asynchronous policies by selecting the minimum



(a) Synchronous **(b)** Asynchronous **(c)** Select Synchronous

Fig. 5: Distributed SGD synchronization policies (lighter shade denotes that the updates from that worker function are ignored while launching the subsequent training epoch).

number of workers to finish before launching the subsequent epoch as seen from Figure 5c. The client waits for the first κ worker functions to finish out of X before launching the subsequent epoch say t. However, the remaining $1-\kappa$ straggler workers' update would be taken into account in $t+1$ epoch. As given in Equation 2, the model $\phi_{t,j}$ learns and adapts to the dynamic data arrival patterns $S_{t,j}$ in real time where s_t samples arrived at time t. The workers pushed till worker j and $\gamma(t,j)$ gives the worker index in a particular epoch when the t-th learner last pushed to PS, such that t is less than j. We refer to this scheme as selective synchronous.

$$\phi_{t+1} \Leftarrow \phi_t - \frac{1}{\kappa}\sum_{n=1}^{\kappa} f(\phi_{\gamma(t,j)}, s_{(t,j)}). \tag{2}$$

IV. Sandpiper Design and Integration

We design a novel runtime system, called *Sandpiper*, that incorporates selective synchronization technique combined with serverless functions, discussed in the above section. *Sandpiper* (i) takes in data updates, (ii) detects accuracy losses due to drifts, (iii) selects the cloud function configuration dynamically based on the loss in recommender quality (accuracy), and (iv) bridges them through retraining in real-time within a given cost-budget. The key components of *Sandpiper* design are explained in detail in the following subsections.

A. Architectural Overview

Sandpiper enables online ML models to remain up to date with minimal overheads. As shown in Figure 6 ❶, the initial model metrics are aggregated from the cloud for every

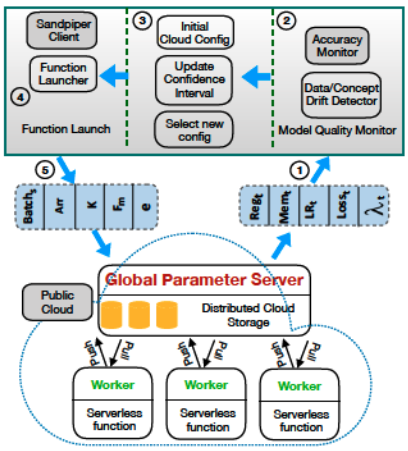


Fig. 6: Overview of *Sandpiper* design.

heartbeat interval to determine if there is a drop in accuracy due to data drift. *Sandpiper* uses an adaptive windowing technique (ADWIN) to detect the drifts to trigger retraining. This can be either run on a local client or on a VM. Once a drift that is more than the preset threshold has been detected, *Sandpiper* launches its performance model which has GP-OPT at its core to identify the optimum cloud configuration vector ❷. Subsequently, the retraining cloud tasks are launched based on the selected cloud configuration ❺ through function launcher ❹ via public cloud APIs.

In *Sandpiper*, instead of having a dedicated stateful parameter server, we leverage the global block store such as AWS S3 to share common paths to push and pull the updated gradients. The model related metrics are pulled by the client from S3 before launching the workers for the subsequent retraining epoch. Depending on the size and arrival rate of the data updates s_t, *Sandpiper* launches single-threaded or multi-threaded retraining through the function launcher. For the distributed function launch, *Sandpiper* uses the optimization function between subsequent epochs to pick the right cloud configuration in order to minimize the retraining time and training loss.

Algorithm 1 Optimum configuration selection.

Require:
Price(x): cost per GB-s
Initial Config Vector:$cv_t < K, Batch_s, F_m, LR_t, lambda >$
Tmax : maximum run time of epoch
Search Vector: $< cv_1{}', cv_2{}', .., cv_n{}' ¿$
Ensure:
Optimal Function Config Vector: cv_x
1: Initial configuration cv_t to run the first epoch
2: Observe the metrics L,A,Reg_t
3: **repeat** for epoch t = 2, 3, $\cdots$
4: Select the config vector cv_t from Search Vector based on observed metrics;
5: Use the function configuration cv_t to launch workers;
6: **until** Reach the stop condition ($Loss < 0.01$);

B. Configuration Selection

The goal of the *Sandpiper* is to pick an optimal function configuration that satisfies the retraining latency (L) for a given budget. The notations used in our formulation are given in Table III. Initially, the model begins with a configuration vector, as shown in Algorithm 1. The worker functions are launched with the initial cloud configuration and the observed latency is fed into the performance model. This is used to estimate the confidence interval of the latency for candidate configuration vectors. The worker functions can be launched using different configuration types by varying the parameters shown in Table III. For instance, a worker function set is configured with batch size, selective cut off ratio, and learning rate for that epoch. Besides configuring the worker function set, the users have certain flexibility to configure resource types such as the VM size (IaaS) or size of memory in the case of serverless functions (Lambda).

In order to limit the search space, an online real-time configuration selection tool is needed. We use an adaptable ML-based model to configure resources for the chosen application execution mode. Using an online learning algorithm, we can directly capture the model uncertainty due to the vast search space. We leverage Gaussian-Process [24] based optimization function (GP-OPT) as it provides the feasible set of configurations as a distribution, rather than just one value as the prediction. Further, the GP-OPT policy quickly minimizes *regrets* (Regret in this context is defined as the loss in reward (cost) for not selecting the expected configuration), compared to other prediction models [25]. An abstract configuration model for minimizing the total cost (T_cost) is expressed as:

$$T\_cost = L(cv_t) \cdot U(cv_t),\ such\ that\ L(cv_t) \leq L_{Max}, \quad (3)$$

where, at a given time-step t, (i) cv_t is the cloud configuration vector, (ii) $L(cv_t)$ is the observed latency for the chosen configuration, (iii) $U(cv_t)$ is the cost of the chosen configuration per unit time as specified by the provider, and (iv) L_{Max} is the target latency for the application. cv_t captures all resource specific metrics such as the VM-instance type, memory capacity, #compute cores, cost-per-hour, etc. (obtained from sub-thrust 3.1.2). For serverless functions, cv_t will also include function execution time and memory allocated.

The GP-OPT will choose a configuration in the Search Vector that yields the minimum total cost by minimizing the total run time of the worker function set. Since the latency to retrain for an epoch is not known in advance, we need to have multiple test runs before we can identify the candidates for the search vector. Therefore, we run experiments with multiple configuration vector combinations that could yield better retraining times. However, it is not possible to exhaust all the combinations of configuration types since there can be multiple variations of the parameters in the search vector. To speed up the search for the candidate configuration, we need to design a *pruning scheme* that can find out the optimal worker set configuration within a reasonable number of runs.

1) Pruning using Gaussian Process To narrow down the candidate configuration list, we use Gaussian Process to select the candidate configuration vector. After every epoch, the candidate configuration that yields the maximum reward probabilities is further exploited to optimize for the OPT function. Before every epoch, we select a worker set configuration cv_t and the functions are launched, while the latency to retrain

Notation	Parameter	Range
L	Latency to retrain	1-100sec
A	Accuracy of the Online Model	90-100%
c	Cost for the retraining (Budget)	30-120$
K	Selective Sync Cutoff	1-K
$Batch_s$	Mini batch size	1-Max_learners
F_m	Function memory size	1Mb-3Gb [26]
Arr	data arrival rate per sec	10-6978 (Table II)
e	Total number of epochs	1-N (loss¡0.001)
Mem_t	Memory utilization at epoch t	
LR_t	Learning rate at epoch t	
$Loss_t$	Training loss at epoch t	
λ_t	Number of Functions at epoch t	
Reg_t	Regret at epoch t	
cv'	Optimal Configuration Vector	
cv*	Candidate Configuration Vector	

TABLE III: Notations & Parameters for cloud configuration.

parameter *(L)* is observed for the candidate configuration. This parameter is perturbed with a random noise that follows a Gaussian distribution. Based on the current observation and understanding, we select the next round of function configuration cv_{t+1}. This is repeated until the end of the algorithm, which is either the maximum tolerable latency L_{Max} or the target loss value of the worker functions reaching model convergence, e.g., *Loss* < 0.01.

We need to maximize the reward function and minimize the optimization function (OPT). In other words, the *Sandpiper*'s model should maximize the job completion rate of the online retraining jobs while minimizing the OPT from predicting the optimum candidate configuration. The latency to run model retraining during an epoch is given by *L*; therefore, its job completion ratio is inverse of the time taken for the retraining *(1/L)*. We model the OPT in Equation 4 as follows:

$$OPT_X^s \Leftarrow \sum_{t=1}^{X} \frac{1}{L(cv_t') \cdot c(cv_t')} - \sum_{n=1}^{X} \frac{1}{L(cv*_t) \cdot c(cv*_t)}, \quad (4)$$

where X is the total number runs, cv' is the optimal configuration vector, and cv* is the selected function configuration for the test run. L and c are the latency to retrain and the cost associated with retraining, respectively.

To summarize, the core intention of GP-OPT is to minimize the optimization function while increasing the job completion ratio. This depends on the balance between the exploration and exploitation of different configuration vectors. A good candidate recommendation system should balance between the exploration and the exploitation. Therefore, the GP-OPT function, by combining the OPT function and the Gaussian process model, can strike the right balance between the candidate selection exploitation and exploration.

V. Evaluation Methodology

We have implemented *Sandpiper* on Python with AWS client APIs to launch retraining on AWS Lambda and EC2 VMs using MXnet Gluon API. Our prototype implementation consists of four parts: (1) the worker function and parameter server code encapsulating MXNet ML libraries; (2) the local client built with AWS SDK to invoke and manage both VMs and Lambda functions; (3) *Sandpiper* layer, which monitors the model accuracy and detects data drifts; and (4) The performance model to find out the optimal function launch configuration using GP-OPT.

Offline Profiling: We evaluated our backend ML models to characterize their resource consumption footprint in terms of compute, memory, I/O, and disk, as it is necessary to know the bounds on a single invocation of the serverless instance. Our model serving agent launches the serverless invocations adhering to these resource quotas. This is usually determined by two factors – first, the size of the batch/mini-batch data during model retraining, and second, the dominant resource consumption in terms of memory, I/O and compute for different models. Both the factors need to be considered for leveraging the benefits of serverless functions.

A. Stateless Model Training

We modified the training loops of ML models listed in Table I to make them stateless by persisting the intermediate model for inference in AWS S3, while the new data updates train on top of the intermediate models, instead of starting from scratch. Since our ML models are built using two different frameworks, namely, Sklearn and Mallet, we use two different mechanisms to enable model persistence. In the case of Sklearn, we use pickle library to serialize and deserialize the model weights for persistence, and we selectively fit partial functions to perform retraining on top of an existing model. Retraining in the case of Mallet is performed by extracting the compressed topic model through input state argument via the topic trainer. Further, this model is pulled from the data-store when required to perform retraining.

B. Evaluation Metrics and Policies

We deploy both single-threaded and parameter-server based training models to evaluate for key metrics such as performance in terms of retraining time, accuracy when compared to base-model, and cost of deploying the continuous retraining on public clouds using serverless functions. We measure the training time, and calculate the total cost when training completes (converges). All the experiments are conducted in (i) Amazon EC2, where we use c5.large on-demand instances (16 vCPUs, 32 GB memory and 5 Gbps link) at 39 cents/hr, and (ii) AWS Lambda. While calculating costs, we ignore the monthly AWS Lambda free quotas. As a head-to-head comparison, we enable auto-scaling in case of VMs to aggressively scaledown VMs based on the data arrivals to cut costs to be competitive with lambda functions.

We evaluate the *Sandpiper*'s GP-OPT based recommendation system and compare it with two other retraining policies, namely, *best-effort* and *cost-aware*. Best-effort is the most-accurate policy such that it triggers retraining for every time instant a fresh data arrives. Simultaneously worker functions are launched to initiate every retraining. However, it can lead to increased cost due to the number of worker-functions deployed. In contrast, the cost-aware scheme is modeled similar to a recent work [3], which launches the retraining updates by diligently batching the data. In order to do so, the cost-aware policy employs a threshold-based update policy.

VI. Results and Analysis

A. Retraining Speedup

We plot the average retraining time reported by the worker functions to incorporate the data updates generated by production traces. In Figure 7, we compare *Sandpiper* against two alternate retraining policies, namely, *cost-aware* and *best-effort*. The best-effort policy is targeted to incorporate data updates as soon as they arrive without batching or waiting. Therefore, the model is retrained continuously launching worker functions to incorporate the freshly arrived data. On the other hand, the cost-aware policy batches the incoming data samples by trading off accuracy by delaying the data updates. We batch the data updates for every 5-minute interval, and trigger retraining in the case of the cost-aware scheme.

In Figure 7a, we plot the retraining time for the Berkeley trace updates. It can be observed from these results that the best effort retraining policy performs retraining by up to 25-28% faster by incorporating data updates into the model. This is due to the fact that the best effort policy triggers retraining for incoming data sample without any data aggregation; as a result, these individual updates take lesser time when compared to *Sandpiper*. On the other hand, *Sandpiper* performs 61% better than the cost-aware policy. Since *Sandpiper* is able to detect the accuracy drifts in the data *dynamically*, it optimizes for accuracy and cost without exceeding the cutoff threshold for maximum latency. We pick this threshold depending on the arrival rate of the data updates. Typically, the cutoff threshold of maximum latency L_{Max} is twice the average latency of the best effort policy. Similarly, in the case of the wiki trace shown in Figure 7b and the Twitter trace shown in Figure 7d, *Sandpiper* launches retraining jobs that finish 35-40% faster when compared to the cost-aware policy.

On the other hand, in the case of the WITS trace shown in Figure 7c, *Sandpiper* performs almost the same as the best effort and slightly better than the cost-aware. This is because, the WITS trace has a high peak-to-median ratio of DUPS compared to other traces, and as a result, none of the dynamic policies could optimize for latency without trading off accuracy. Therefore, *Sandpiper* performs similar to the best effort by launching more retraining functions to bridge the wide gap in accuracy.

B. Cost vs Accuracy Comparison

We plot the public cloud cost of running *Sandpiper* to retrain through DSGD for data updates in case of the WITS trace. The other traces also show a similar trend, which is in proportion to the peak-to-median ratio of data updates. We compare *Sandpiper* along with other continuous training policies such as best-effort, cost aware, VM (IaaS), and VM-Autoscaler. As shown in Figure 8, compared to the EC2-based VM cluster, *Sandpiper* maintains the model accuracy at 90% for 3.8× lesser cost (green region), on average. This cost saving is due to the nature of burstiness in the WITS trace, where the peak-to-median ratio is high requiring over-provisioning of VMs in case of peak data update arrivals to meet the retraining deadline. This provisioning leads to inherent resource wastage and under-utilization. Enabling aggressive auto-scaling for VM provisioning could reduce the cost by 43% as the VMs are scaled down aggressively avoiding resource wastage. However it is still 3.2× expensive when compared to GP-OPT policy of Sandpiper to maintain an average model accuracy of 90%. We do not use transient VM provisioning since the VM revocations would violate the deadline while loosing the model state as they could be in the order of tens of seconds.

Sandpiper's GP-OPT diligently adjusts its parameters such as batch size, learning rate, and number of worker functions launched for every retraining epoch based on the reward to minimize the optimization function. This dynamic worker function launches minimize the wasted computations, leading to model convergence in case of retraining at a cheaper cost.

intain the
, the best-
to higher
oning, as

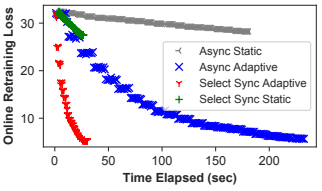


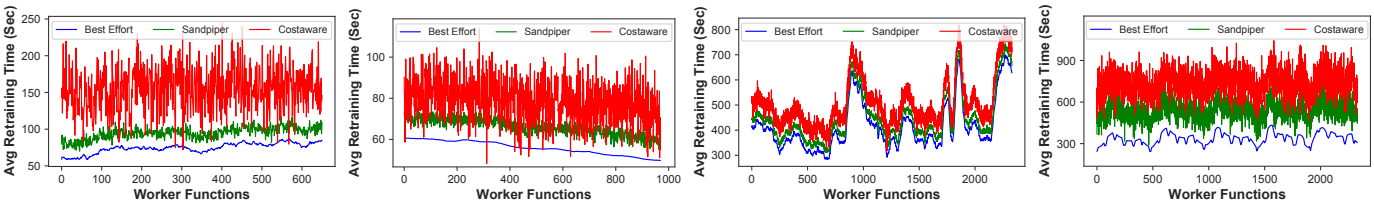
Fig. 9: Training loss of DSGD worker functions in case of different sync policies inside *Sandpiper*. For brevity, we only plot till one of the schemes converged. (worker function training loss $<$ 0.001 for 5 consecutive epochs).

We evaluate the effectiveness of the proposed GP-OPT-based function configuration selection along with the two different synchronization policies. From the previous results, we understand the effectiveness of the select-sync policy when compared to other fully-synchronous or asynchronous schemes. However, picking the right number of workers to synchronize for a given epoch is crucial for *both* accuracy and retraining latency. Therefore, we leverage the GP-OPT's dynamic policy (adaptive) to select the right ratio which minimizes the job completion time. We evaluate the effectiveness of this adaptive select-sync and compare with other static selection and asynchronous policies.

Towards this, we plot two metrics in Figure 9 – *retraining loss* and *time taken to converge*. The time elapsed is measured spanning from the beginning of training to model convergence. Convergence is reached when the retraining loss is below the target value for at least five consecutive epochs. We compare the select sync adaptive scheme against several asynchronous static and adaptive schemes. It can be observed from the results that *Sandpiper* significantly improves the retraining time by minimizing the retraining loss by up to 6× when compared to the nearest adaptive scheme which uses asynchronous scheme to synchronize the candidate workers.

VII. Related Work

The concept of continuous learning has been widely adopted from stream processing frameworks such as Spark streaming [27] and Flink. The fundamental factors that influence



(a) Berkeley **(b)** Wiki **(c)** WITS **(d)** Twitter

Fig. 7: Comparison of average retraining time of DSGD worker functions for data updates from different production traces (worker function's maximum runtime is capped at 900secs due to the maximum run time limits of AWS Lambda function).

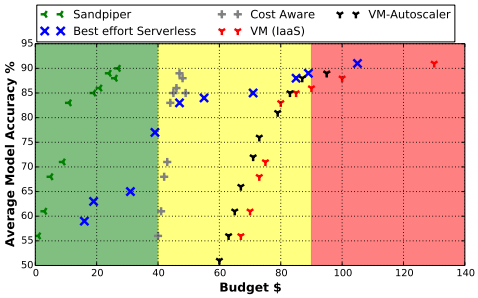


Fig. 8: Cost and accuracy tradeoff for the WITS trace.

stream processing are similar to our problem and include throughput, latency, and processing time of data. Similarly, there have been commercially available open source frameworks for ML model serving such as clipper [28]. In contrast, there have been few academic works such as Continuum [3] that study online recommender systems at depth, particularly on retraining efficiency. Such approaches are specifically catered towards single-threaded retraining at a static arrival setting. Continuum minimizes training costs by delaying model updates and sacrificing accuracy. It does not leverage automated drift detection to maintain model accuracy. In addition, it does not leverage the stateless cloud offerings and cannot scale efficiently in terms of cost. Furthermore, to our knowledge, none of the prior works considers the synchronization overheads in distributed training, to meet the strict deadlines for retraining.

VIII. Conclusion

In this paper, we investigate the resource provisioning challenges in terms of performance (latency), accuracy, and cost associated with retraining in public clouds. Towards this, we built *Sandpiper*, which is a run-time system based on the serverless framework that supports continuous learning for streaming applications. *Sandpiper* leverages an optimization function to achieve a balance among model retraining time, accuracy, and cost. It also monitors the accuracy drops due to data drifts and minimizes the retraining cost by finding an optimal job configuration in cloud. Our real-system evaluation of *Sandpiper* on AWS cloud with production traces shows that the GP-OPT policy used in *Sandpiper* can provide a near-optimal cloud configuration on AWS Lambda in fewer than 50 runs from thousands of candidate choices, which in turn translates to 3.8× savings in cost when compared to IaaS. Furthermore, *Sandpiper* performs 5.7× faster model updates when compared to accuracy-agnostic schemes.

References

[1] D. Sculley, G. Holt, D. Golovin, E. Davydov, T. Phillips, D. Ebner, V. Chaudhary, M. Young, J.-F. Crespo, and D. Dennison, "Hidden technical debt in machine learning systems," in *Neurips*, 2015.

[2] J. Gama, I. Žliobaitė, A. Bifet, M. Pechenizkiy, and A. Bouchachia, "A survey on concept drift adaptation," *CSUR*, 2014.

[3] H. Tian, M. Yu, and W. Wang, "Continuum: A platform for cost-aware, low-latency continual learning." in *SoCC*, 2018.

[4] D. Mudigere, Y. Hao *et al.*, "High-performance, distributed training of large-scale deep learning recommendation models," *CoRR*, 2021.

[5] M. Xie, K. Ren, Y. Lu, G. Yang, Q. Xu, B. Wu, J. Lin, H. Ao, W. Xu, and J. Shu, "Kraken: Memory-efficient continual learning for large-scale real-time recommendations," in *2020 SC*, 2020.

[6] J. Carreira, P. Fonseca, A. Tumanov, A. Zhang, and R. Katz, "Cirrus: A serverless framework for end-to-end ml workflows," ser. SoCC '19.

[7] Y. Liu, A. Niculescu-Mizil, and W. Gryc, "Topic-link lda: joint models of topic and author community," in *ICML*, 2009.

[8] Y. Koren, R. Bell, and C. Volinsky, "Matrix factorization techniques for recommender systems," *Computer*, vol. 42, no. 8, pp. 30–37, 2009.

[9] "AWS Lambda," https://aws.amazon.com/lambda/pricing/.

[10] J. Carreira, P. Fonseca, A. Tumanov, A. Zhang, and R. Katz, "A case for serverless machine learning," Learningsys 2018.

[11] M. Hoffman, F. R. Bach, and D. M. Blei, "Online learning for latent dirichlet allocation," in *Neurips*, J. D. Lafferty, C. K. I. Williams, J. Shawe-Taylor, R. S. Zemel, and A. Culotta, Eds., 2010.

[12] J. Leskovec, L. Backstrom, and J. Kleinberg, "Meme-tracking and the dynamics of the news cycle," in *SIGKDD*. ACM, 2009.

[13] A. K. McCallum, "Mallet: A machine learning for language toolkit," *http://mallet. cs. umass. edu*, 2002.

[14] F. M. Harper and J. A. Konstan, "The movielens datasets: History and context," *ACM TIIS), year=2016,.*

[15] F. Pedregosa, G. Varoquaux, A. Gramfort, V. Michel *et al.*, "Scikit-learn: Machine learning in python," *JMLR*, 2011.

[16] D. Harrison Jr and D. L. Rubinfeld, "Hedonic housing prices and the demand for clean air," *JEEM*, 1978.

[17] T. Beluch, A. Nürnberger, and J. M. Köhler, "The power of ensembles for active learning in image classification," CVPR'18.

[18] G. Urdaneta, G. Pierre, and M. Van Steen, "Wikipedia workload analysis for decentralized hosting," *Computer Networks*, 2009.

[19] "Twitter traces," https://archive.org/details/twitterstream, 2020.

[20] "UC Berkeley 18 days' worth of http traces," https://www.comp.nus.edu.sg/~cs5222/simulator/traces/berkeley/index.htm, accessed: 2020-05-03.

[21] "Waikato Internet Traffic Storage," https://wand.net.nz/wits/index.php.

[22] S. Gupta, W. Zhang, and F. Wang, "Model accuracy and runtime tradeoff in distributed deep learning: A systematic study," in *2016 ICDM*.

[23] G. Joshi, "Slow and stale gradients can win the race: Error-runtime trade-offs in distributed sgd," 2019.

[24] "Leveraging Gaussian Process for Machine Learning," http://www.gaussianprocess.org/gpml/chapters/RW.pdf.

[25] S. D. Whitehead, "A complexity analysis of cooperative mechanisms in reinforcement learning," ser. AAAI'91. AAAI Press, 1991, p. 607–613.

[26] "AWS Lambda resource limits and restrictions," https://docs.aws.amazon.com/lambda/latest/dg/limits.html, accessed: 2020-05-06.

[27] "Apache Spark Streaming," https://spark.apache.org/docs/latest/streaming-programming-guide.html#overview.

[28] "Clipper.AI ML serving system," http://clipper.ai.