Cirrus: a Serverless Framework for End-to-end ML Workflows

Joao Carreira
University of California, Berkeley
joao@berkeley.edu

Pedro Fonseca
Purdue University
pfonseca@purdue.edu

Alexey Tumanov
Georgia Institute of Technology
atumanov@gatech.edu

Andrew Zhang
University of California, Berkeley
andrewmzhang@berkeley.edu

Randy Katz
University of California, Berkeley
randykatz@berkeley.edu

ABSTRACT

Machine learning (ML) workflows are extremely complex. The typical workflow consists of distinct stages of user interaction, such as preprocessing, training, and tuning, that are repeatedly executed by users but have heterogeneous computational requirements. This complexity makes it challenging for ML users to correctly provision and manage resources and, in practice, constitutes a significant burden that frequently causes over-provisioning and impairs user productivity. Serverless computing is a compelling model to address the resource management problem, in general, but there are numerous challenges to adopt it for existing ML frameworks due to significant restrictions on local resources.

This work proposes Cirrus—an ML framework that automates the end-to-end management of datacenter resources for ML workflows by efficiently taking advantage of serverless infrastructures. Cirrus combines the simplicity of the serverless interface and the scalability of the serverless infrastructure (AWS Lambdas and S3) to minimize user effort. We show a design specialized for both serverless computation and iterative ML training is *needed* for robust and efficient ML training on serverless infrastructure. Our evaluation shows that Cirrus outperforms frameworks specialized along a single dimension: Cirrus is 100x faster than a general purpose serverless system [36] and 3.75x faster than specialized ML frameworks for traditional infrastructures [49].

CCS CONCEPTS

• **Computer systems organization** → **Cloud computing**; • **Computing methodologies** → *Distributed programming languages.*

KEYWORDS

Serverless, Distributed Computing, Machine Learning

ACM Reference Format:
Joao Carreira, Pedro Fonseca, Alexey Tumanov, Andrew Zhang, and Randy Katz. 2019. Cirrus: a Serverless Framework for End-to-end ML Workflows. In *SoCC '19: ACM Symposium of Cloud Computing conference, Nov 20–23, 2019, Santa Cruz, CA.* ACM, New York, NY, USA, 12 pages.

SoCC '19, November 20-23, Santa Cruz, CA

ACM ISBN 978-1-4503-9999-9/18/06…$15.00
DOI: 10.1145/3357223.3362711

1 INTRODUCTION

The widespread adoption of ML techniques in a wide-range of domains, such as image recognition, text, and speech processing, has made machine learning one of the leading revenue-generating datacenter workloads. Unfortunately, due to the growing scale of these workloads and the increasing complexity of ML workflows, developers are often left to manually configure numerous *system-level* parameters (e.g., number of workers/parameter servers, memory footprint, amount of compute, physical topology), in addition to the ML-specific parameters (learning rate, algorithms, neural network structure).

Importantly, modern ML workflows are iterative and increasingly comprised of multiple heterogeneous stages, such as (a) preprocessing, (b) training, and (c) hyperparameter searching. As a result, due to the iterative nature and diversity of stages, the end-to-end ML workflow is highly complex for users and demanding in terms of resource provisioning and management, detracting users from focusing on ML specific tasks—the domain of their expertise.

The complexity of ML workflows leads to two problems. First, when operating with coarse-grained VM-based clusters the provisioning complexity often leads to overprovisioning. Aggregate CPU utilization levels as low as 20% are not uncommon [27, 45]. Second, the management complexity is increasingly an obstacle for ML users because it hinders the interactive and iterative use-cases, degrading user productivity and model effectiveness.

This work proposes Cirrus, a distributed ML training framework that addresses these challenges by leveraging serverless computing. Serverless computing relies on the cloud infrastructure, not the users, to automatically address the challenges of resource provisioning and management. This approach relies on a more restricted unit of computation, the stateless *lambda function*, which is submitted by developers and scheduled to execute by the cloud infrastructure. Thus, obviating the need for users to manually configure, deploy, and manage long-term compute units (e.g., VMs). The advantages of the serverless paradigm have promoted its fast adoption by datacenters and cloud providers [2, 5, 6, 9, 10, 15] and open source platforms [8, 16, 18, 32].

However, the benefits of serverless computing for ML hinge on the ability to run ML algorithms *efficiently*. The main challenge in leveraging serverless computing is the significantly small local resource constraints (memory, cpu, storage, network) associated with lambda functions, which is fundamental to serverless computation because the fine-granularity of computation units enables scalability and flexibility. In contrast, existing ML systems commonly assume abundant resources, such as memory. For instance, Spark [51] and Bosen [49, 50] generally load all training data into memory.

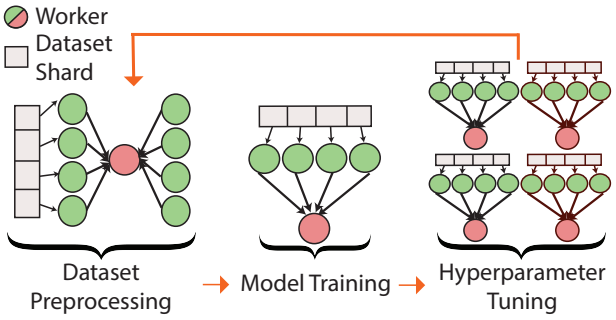


Figure 1: Typical end-to-end machine learning workflow. (1) dataset preprocessing typically involves an expensive map/reduce operation on data. It is common to take multiple passes over data, e.g., when normalization is required. (2) model training (parameter server). Workers consume data shards, compute gradients, and synchronize with a parameter server. (3) hyperparameter optimization to tune model and training parameters involves running multiple training instances, each configured with a different set of tuning parameters.

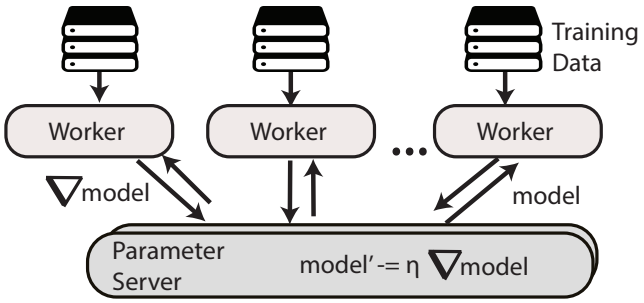


Figure 2: Distributed stochastic gradient descent training with parameter server. The parameter server iteratively computes a new model based on the model gradients it receives from workers. Workers then compute new model gradients from a subset of training data (minibatch) and the model distributed by the parameter server. This iterative process continues until the model converges.

Similarly, some frameworks require data to be sharded or replicated across all workers, implicitly assuming resource longevity for the duration of long-running compute.

Frameworks specifically designed to deal with the resource limitations of serverless infrastructures have been proposed. However, we find that they face fundamental challenges when used for ML training tasks out of the box; in addition to having no support for ML workflows. As an example, PyWren [36] uses remote storage for intermediate computation results, adding significant overheads to fine-grain iterative compute tasks which are typical of ML workloads. Importantly, the reliance on external storage by such frameworks is fundamental to their design, enabling them to scale to large data-intensive jobs (e.g., map-reduce computations). However, we observe that ML workflow computations are heterogeneous and involve frequent fine-grained communication between computational nodes which requires a novel design to ensure efficiency.

Importantly, Cirrus is designed to efficiently support the entire ML workflow. In particular, Cirrus supports fine-grain, data-intensive serverless ML training and hyperparameter optimization efficiently. Based on the parameter server model (see Figure 2), Cirrus provides an easy-to-use interface to perform scalable ML training leveraging the high scalability of serverless computation environments and cloud storage. Cirrus unifies the benefits of specialized serverless frameworks with the benefits of specialized ML training frameworks and provides an easy-to-use interface (§4) that enables typical ML training workflows and supervised learning algorithms (e.g., Logistic Regression, Collaborative Filtering) for end-to-end ML workflows on serverless infrastructure.

Cirrus builds on three key design properties. First, Cirrus provides an ultra-lightweight (~80MB vs 800MB for PyWren's runtime) worker runtime that adapts to the full range of lambda granularity, providing mechanisms for ML developers to find the configuration that best matches their time or cost budget. Second, Cirrus saves on the cost of provisioning large amounts of memory or storage—a typical requirement for ML training frameworks. This is achieved through a combination of (a) streaming training minibatches from remote storage and (b) redesigning the distributed training algorithms to work robustly in the serverless environment. Third, Cirrus adopts stateless worker architecture, which allows the system to efficiently handle frequent worker departure and arrival as expected behavior rather than an exception. Cirrus provides the best of both serverless-specialized and ML-specialized frameworks through the combined benefit of different contributions, e.g., a data prefetching iterator (10x speedup). This yields a 3.75x improvement on time-to-accuracy compared to the best-performing configuration ML specialized frameworks [19, 49] (§6.2) and 100x compared to the best-performing configuration of PyWren (§6.5).

2 DEMOCRATIZING MACHINE LEARNING

2.1 End-to-end ML Workflow Challenges

Machine learning researchers and developers execute a number of different tasks during the process of training models. For instance, a common workflow consists of dataset preprocessing, followed by model training and finally by hyperparameter tuning (Figure 1). In the dataset preprocessing phase, developers apply transformations (e.g., feature normalization or hashing) to datasets to improve the performance of learning algorithms. Subsequently, in the model training phase, developers coarsely fit a model to the dataset, with the goal of finding a model that performs reasonably well and converges to an acceptable accuracy level. Finally, in the hyperparameter tuning phase, the model is trained multiple times with varying ML-parameters to find the parameters that yield best results.

ML training tasks have been traditionally deployed using systems designed for clusters of virtual execution environments (VMs) [19, 20, 26, 49, 51]. However, such designs create two important challenges for users: (a) they can lead to over-provisioning (b) they require explicit resource management by users.

Over-provisioning. The heterogeneity of the different tasks in an ML workflow leads to a significant resource imbalance during the execution of a training workflow. For instance, the coarse-granularity and rigidity of VM-based clusters as well as the design of the ML frameworks specialized for these environments causes developers

User responsibility	Description
Sharding data	Distribute datasets across VMs
Configuring storage systems	Setup a storage system (e.g., NFS)
Configuring OS/drivers	Choosing OS and drivers
Deploying frameworks	Install ML training frameworks
Monitoring	Monitor VMs for errors
Choosing VM configuration	Choosing VM types
Setup network connections	Make VMs inter-connect
Upgrading systems	Keep libraries up-to-date
Scaling up and down	Adapt to workload changes

Table 1: Typical responsibilities ML users face when using a cluster of VMs.

to frequently *over-provision* resources for peak consumption, which leads to significant waste of datacenter resources [27, 45]. The over-provisioning problem is exacerbated by the fact that, in practice, developers repeatedly go back and forth between different stages of the workflow to experiment with different ML parameters.

Explicit resource management. The established approach of *exposing low-level VM resources*, such as storage and CPUs, puts a significant burden on ML developers who are faced with the challenge of provisioning, configuring, and managing these resources for each of their ML workloads. Thus, systems that leverage VMs for machine learning workloads generally require users to repeatedly perform a series of onerous tasks we summarize in Table 1. In practice, over-provisioning and explicit resource management burden are tightly coupled—ML users often resort to over-provisioning due to the difficulty and human cost of accurately managing resource allocation for the different stages of their training workflow.

2.2 Serverless Computing

Serverless computing is a promising approach to address these resource-provisioning challenges [31, 35]. It simultaneously simplifies deployment with its intuitive interface and provides mechanisms to avoid over-provisioning, with its fine-grain serverless functions that can run with as few as 128MB of memory (spatial granularity) and time out in a few minutes (temporal granularity). This ensures natural elasticity and agility of deployment. However, serverless design principles are at odds with a number of design principles of existing ML frameworks today. This presents a set of challenges in adopting serverless infrastructures for ML training workflows. This section discusses the major limitations of existing serverless environments and the impact they have for machine learning systems.

Small local memory and storage. Lambda functions, by design, have very limited memory and local storage. For instance, AWS lambdas can only access at most 3GB of local RAM and 512MB of local disk. It is common to operate with lambdas provisioned for as little as 128MB of RAM. This precludes the strategy often used by many machine learning systems of replicating or sharding the training data across many workers or of loading all training data into memory. These resource limitations prevent the use of any computation frameworks that are not designed with these resource constraints in mind. For instance, we have not been able to run Tensorflow [19] or Spark [51] on AWS lambdas or on VMs with such resource-constrained configurations.

Low bandwidth and lack of P2P communication. Lambda functions have limited available bandwidth when compared with a regular VM. We find that the largest AWS Lambda can only sustain 60MB/s of bandwidth, which is drastically lower than 1GB/s of bandwidth available even in medium-sized VMs. Further restrictions are imposed on the communication topology. Serverless compute units such as AWS Lambdas do not allow peer-to-peer communication. Thus, common communication strategies used for datacenter ML, such as tree-structured or ring-structured AllReduce communication [43], become impossible to implement efficiently in such environments.

Short-lived and unpredictable launch times. Lambda functions are short-lived and their launch times are highly variable. For instance, AWS lambdas can take up to several minutes to start after being launched. This means that during training, lambdas start at unpredictable times and can finish in the middle of training. This requires ML runtimes for lambdas to tolerate the frequent departure and arrival of workers. Furthermore, it makes runtimes such as MPI (used, for instance, by Horovod [47] and Multiverso [7]) a bad fit for this type of architecture.

Lack of fast shared storage. Because lambda functions cannot connect between themselves, shared storage needs to be used. Because ML algorithms have stringent performance requirements, this shared storage needs to be low-latency, high-throughput, and optimized for the type of communications in ML workloads. However, as of today there is no fast serverless storage for the cloud that provides all these properties.

3 CIRRUS DESIGN

CIRRUS is an end-to-end framework specialized for ML training in serverless cloud infrastructures (e.g., Amazon AWS Lambdas). It provides high-level primitives to support a range of tasks in ML workflows: dataset preprocessing, training, and hyperparameter optimization. This section describes its design and architecture.

3.1 Design Principles

Adaptive, fine-grained resource allocation. To avoid resource waste that arises from over-provisioning, CIRRUS should flexibly adapt the amount of resources reserved for each workflow phase with fine-granularity.

Stateless server-side backend. To ensure robust and efficient management of serverless compute resources, CIRRUS, by design, operates a stateless, server-side backend (Figure 3). The information about currently deployed functions and the mapping between ML workflow tasks and compute units is managed by the client-side backend. Thus, even when all cloud-side resources become unavailable, the ML training workflow does not fail and may resume its operation when resources become available again.

End-to-end serverless API.. Model training is not the only important task an ML researcher has to perform. Dataset preprocessing, feature engineering, and parameter tuning are other examples of

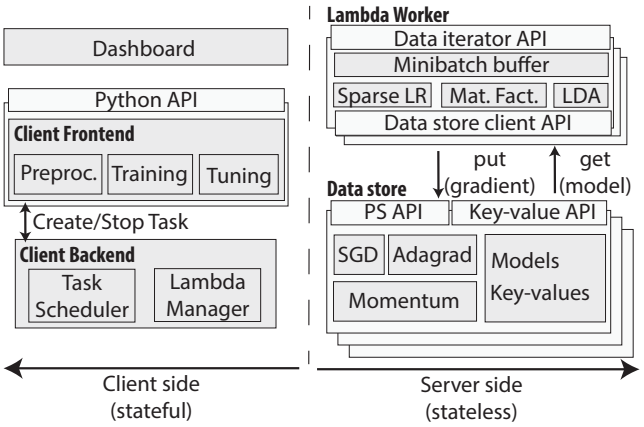


Figure 3: Cirrus system architecture. The system consists of the (stateful) client-side (left) and the (stateless) server-side (right). The client-side contains a user-facing frontend API and supports preprocessing, training, and tuning. The client-side backend manages cloud functions and the allocation of tasks to functions. The server-side consists of the Lambda Worker and the high-performance Data Store components. The lambda worker exports the data iterator API to the client backend and contains efficient implementation for a number of iterative training algorithms. The data store is used for storing gradients, models, and intermediate preprocessing results.

tasks equally important for yielding good models. Cirrus should provide a complete API that allows developers to run these tasks at scale with minimal efforts.

High scalability. ML tasks are highly compute intensive, and thus can take a long time to complete without efficient paralellization. Hence, Cirrus should be able to run thousands of concurrent workers and hundreds of concurrent experiments.

3.2 Cirrus Building Blocks

Cirrus makes use of three system building blocks to achieve the aforementioned principles. First, Cirrus provides a Python frontend for ML developers. This frontend has two functions: a) provide a rich API for all stages of ML training, and b) execute and manage computations at scale in serverless infrastructure. Second, to overcome the lack of offerings for low-latency serverless storage, Cirrus provides a low-latency, distributed data store for all intermediate data shared by the workers. Third, Cirrus provides a worker runtime that runs on serverless lambdas. This runtime provides efficient interfaces to access training datasets in S3 and intermediate data in the distributed data store.

3.2.1 ***Python frontend.*** Cirrus provides an API for all stages of the ML workflow that is practical and easy to use by the broader ML community for three reasons. First, the API is totally contained within a Python package. Because many existing frameworks are developed in Python or have Python interfaces (e.g., Tensorflow, scikit-learn), developers can transition easily. Second, the Cirrus API provides a high-level interface that abstracts the underlying system-level resources. For instance, developers can run experiments with thousands of concurrent workers without having to provision any of those resources. Last, the Cirrus Python package provides a user interface through which developers can visualize the progress of their work.

The Cirrus Python API is divided in three submodules. Each submodule packages all the functions and classes related to each one of the stages of the workflow.

Preprocessing. The preprocessing submodule allows users to preprocess training datasets stored in S3. This submodule allows different types of dataset transformations: min-max scaling, standardization, and feature hashing.

Training. Cirrus's training submodule supports ML models that can be trained with stochastic gradient descent. Currently Cirrus supports Sparse Logistic Regression, Latent Dirichlet Allocation, Softmax and Collaborative Filtering.

Hyperparameter optimization. The hyperparameter optimization submodule allows users to run a grid search over a given set of parameters. Cirrus allows users to vary both ML training parameters (e.g., learning rate, regularization rate, minibatch size) as well as system parameters (e.g., lambda size, # concurrent workers, filtering of gradients). Cirrus can parallelize this task.

3.2.2 ***Client-side backend.*** The Python frontend provides an interface to Cirrus's client backend. This backend sits behind the frontend and does a number of tasks: parse training data and load it to S3, launch the Cirrus workers on lambdas, manage the distributed data store, keep track of the progress of computations and return results to the Python frontend once computations complete.

There is a module in the backend for every stage of the workflow (preprocessing, training, and hyperparameter optimization). These modules have logic specific to each stage of the workflow and know which tasks to launch. They also delegate to the low-level scheduler the responsibility to launch, kill and regenerate tasks. The low-level scheduler keeps track of the state of all the tasks.

3.2.3 ***Worker runtime.*** Cirrus provides a runtime that encapsulates all the functions that are shared between the different computations the system supports. This simplifies the development of new algorithms. The system runtime meets two goals: 1) lightweight, to run within memory-constrained lambdas, and 2) high-performance, to mitigate communication and computation overheads exacerbated by serverless infrastructures.

The worker runtime provides two interfaces. First, it provides a smart iterator for training datasets stored in S3. This iterator prefetches and buffers minibatches in the lambda's local memory in parallel with the worker's computations to mitigate the high-latency (>10ms) of accessing S3. Second, it provides an API for the distributed data store. This API implements: data compression, sparse transfers of data, asynchronous communication and sharding across multiple nodes.

3.2.4 ***Distributed data store.*** Cirrus's data store serves the purpose of storing intermediate data to be shared by all workers. Because inter-communication between lambdas is not allowed in existing offerings, lambdas require a shared storage. A storage for serverless lambdas needs to meet three goals. First, it needs to be

API	Description
int send_gradient_X(ModelGradient* g)	Sends model gradient
SparseModel get_sparse_model_X(const std::vector<int>& indices)	Get subset of model
Model get_full_model_X()	Get all model weights
set_value(string key, char* data, int size)	Set intermediate state
std::string get_value(string key)	Get intermediate state

Table 2: Cirrus's data store provides a parameter-server style interface optimized for communication of models and gradients. Cirrus's interfaces to send gradients and get model weights are specialized to each model to achieve the highest performance. The data store also provides a general key-value interface for other intermediate state.

AWS Lambda Challenges	Cirrus System Design
Limited lifetime (e.g., 15 min)	Stateless workers coordinate through data store
Memory-constrained (e.g., 128MB)	Runtime prefetches minibatches from remote store
High-variance start time	Runtime tolerates late workers
No P2P connections	Stateful frontend coordinates workers through data store
Lack of low-latency serverless storage with rich API for ML	Data store with parameter-server and key-value API

Table 3: Technical challenges of using lambda functions in Amazon AWS and Cirrus's design choices that address them

low-latency (we achieve as low as 300μs) to be able to accommodate latency-sensitive workloads such as those used for ML training (e.g., iterative SGD). Second, it needs to *scale to hundreds of workers* to take advantage of the almost linear scalability of serverless infrastructures. Third, it needs to have a *rich interface* (Table 2) to support different ML use cases. For instance, it's important that the data store supports multiget (§6.5), general key/value put/get operations, and a parameter-server interface.

To achieve low-latency, we deploy our data store in cloud VMs. It achieves latencies as low as 300μs versus $\approx$ 10ms for AWS S3. This latency is critical to maximize system updates/sec for model updates during training.We use sparse representations for gradients and models, to achieve up to 100x compression ratio for data exchange with the store, and batch requests.

To achieve high scalability Cirrus includes the following mechanisms: (1) sharded store, (2) highly multithreaded, (3) data compression, (4) gradient filters, and (5) asynchronous communication.

3.3 End-to-End ML Workflow Stages

This section describes in detail the computations Cirrus performs. We structure this according to the different stages of the workflow.

3.3.1 ***Data Loading and Preprocessing.*** Cirrus assumes training data is stored in a global store such as S3. For that reason, the very first step when using Cirrus is to upload the dataset to the cloud. The user passes the path of the dataset to the system which then takes care of parsing and uploading it. In this process, Cirrus transforms the dataset from its original format (e.g., csv) into a binary format. This compression eliminates the need for deserialization during the training and hyperparameter tuning phases which helps reduce the compute load in the lambda workers. Second, Cirrus generates similarly-sized partitions of the dataset and uploads them to an S3 bucket.

Cirrus can also apply transformations to improve the performance of models. For instance, for the asynchronous SGD optimization methods Cirrus implements, training is typically more effective after features in the dataset have been normalized. Because normalization is a recurrent data transformation for the training models Cirrus provides, the system allows users to do different types of per-column normalization such as min-max scaling.

For these transformations, Cirrus launches a large map-reduce job – one worker per input partition. In the map phase, each worker computes statistics for its partition (e.g., mean and standard deviation). In the reduce phase, these local statistics are aggregated to compute global statistics. In the final map-phase, the workers transform each partition sample given the final per-column statistics. For large datasets, the map and reduce phase aggregates per-column statistics across a large number of workers and columns. This generates a large number of new writes and reads per second, beyond the transactions throughput supported by S3. For this reason, we use Cirrus's low-latency distributed data store to store the intermediate results of the maps and reduces.

3.3.2 ***Model training.*** For model training Cirrus uses a distributed SGD algorithm. During training workers run on lambda functions and are responsible for iteratively computing gradient steps. Every gradient computation requires two inputs: a minibatch and the most up-to-date model. The minibatches are fetched from S3 through the Cirrus's runtime iterators. Because the iterator buffers minibatches within the worker's memory, the latency to retrieve a minibatch is very low. The most up-to-date model is retrieved synchronously from the data store using the data store API (*get_sparse_model_X*).

For every iteration each worker computes a new gradient. This gradient is then sent asynchronously to the data store (*send_gradient_X*) to update the model.

3.3.3 ***Hyperparameter optimization.*** Hyperparameter optimization is a search for model parameters that yield the best accuracy. A typical practice is to perform a grid search over the multi-dimensional parameter space. The search may be brute-force or adaptive. It is common to let the grid search run to completion in its entirety and post-process the results to find the best configuration. This is a costly source of resource waste. Cirrus obviates this over-provisioning *over time* by providing a hyperparameter search dashboard. Cirrus hyperparameter dashboard provides a unified interface for monitoring a model's loss convergence over time. It lets the user select individual loss curves and terminate the corresponding training experiment. Note that this scopes the termination to the appropriate set of serverless functions, and provides immediate cost savings. Thus, Cirrus offers (a) the API and execution backend for launching a hyperparameter search, (b) the dashboard for monitoring model accuracy convergence, (c) the ability to terminate individual tuning experiments and save on over-provisioning costs.

```python
import cirrus
import numpy as np

local_path = "local_criteo"
s3_input = "criteo_dataset"
s3_output = "criteo_norm"

cirrus.load_libsvm(local_path, s3_input)

cirrus.normalize(s3_input, s3_output,
                 MIN_MAX_SCALING)
```

(a) Pre-process

```python
params = {
 'n_workers': 5,
 'n_ps': 1,
 'worker_size': 1024,
 'dataset': s3_output,
 'epsilon': 0.0001,
 'timeout': 20 * 60,
 'model_size': 2**19,
}

lr_task =  cirrus.LogisticRegression(params)
result = lr_task.run()
```

(b) Train

```python
# learning rates
lrates = np.arange(0.1, 10, 0.1)
minibatch_size = [100, 1000]

gs = cirrus.GridSearch(
 task=cirrus.LRegression,
 param_base=params,
 hyper_vars=["learning_rate", "minibach_size"],
 hyper_params=[lrates, minibatch_size])

results = gs.run()
```

(c) Tune

Figure 4: Cirrus API example. Cirrus supports different phases of ML development workflow: (a) preprocessing, (b) training, and (c) hyperparameter tuning.

3.4 Summary

Serverless compute properties, such as spatiotemporal fine-granularity of compute, make it a compelling candidate for transparent management of cloud resources for scalable, iterative ML training workflows. The benefits of those properties are eclipsed by the challenges they create (Table 3) for existing ML training frameworks that assume (a) abundant compute and memory resources per worker and (b) fault-tolerance as an exception, not a rule. Cirrus, by design, addresses these challenges by (a) embracing fault-tolerance as a rule with its stateless server-side backend, (b) tracking the scalability afforded by cloud-provided serverless functions with a low overhead, high-performance worker runtime and the data store. Cirrus obviates the need to over-provision by leveraging fine-grain serverless compute as well as an interactive dashboard to track and manage costs at a higher, application level for the hyperparameter optimization stage. To the best of our knowledge, Cirrus is the first framework that is simultaneously specialized for ML training and serverless execution environments, morphing the benefits of both.

4 SYSTEM USAGE MODEL

Cirrus provides a lightweight Python API for ML users. Its API lets users perform a wide-range of ML tasks, such as: (1) dataset loading, with support for commonly used data formats, (2) dataset preprocessing, (3) model training, and (4) hyperparameter tuning at scale, from within a single, integrated framework.

We designed Cirrus's API with four goals in mind. First, the API should be simple and easy to use. Our interface should abstract users away from the underlying hardware. Second, the API should cover computations from the beginning to the end of a workflow. Third, the API should facilitate experimentation with different model and optimization parameters because ML users generally spend a significant amount of their time and effort on model and parameter exploration. Fourth, the API should be general, to enable extensibility to other use cases, such as ML pipelines.

We demonstrate the capabilities of the Cirrus API with a toy example – the example in Figure 4 consists of developing an efficient model for the prediction of the probability of a user clicking an ad for a dataset of display ads. This example is based on the Criteo Kaggle competition [4].

The first step in the workflow with Cirrus is to load the dataset and upload it to S3. For instance, the user can call the *load_libsvm* method to load a dataset stored in the LIBSVM [25] format. Behind the scenes Cirrus parses the data, automatically creates partitions for it and then uploads it to S3. The front-end partitions datasets in blocks of roughly 10MB. We chose this size because data partitions in Cirrus are the granularity of data workers transfer from S3. We have found this size allows lambda workers to achieve good network transfer bandwidth. In addition, this keeps the size of each worker's minibatch cache small.

Once the data is loaded into S3 it can be immediately preprocessed. Cirrus provides a submodule with different preprocessing functions. For instance, the user can normalize the dataset by calling the *cirrus.normalize* function with the path of the dataset in S3. Once the data is loaded, the user can train models and see how they perform (with a real-time user interface running on a Jupyter notebook) and subsequently tune the model through hyperparameter search.

5 CIRRUS IMPLEMENTATION

The Cirrus implementation is composed of four components: (1) python frontend, (2) client backend, (3) distributed data store, and (4) worker runtime. The frontend and client backend were implemented in Python for ease of use and to enable the integration of Cirrus with existing machine learning processes. The distributed data store and workers runtime were implemented in C++ for efficiency. Table 4 lists the different components implemented as well as their size and implementation language. The worker runtime code includes the iterators interface and the data store client implementation. The worker's runtime and the datastore communicate through TCP connections. We implemented a library of shared components, which includes linear algebra libraries, general utilities, and ML algorithms that are shared by all system components. We have released publicly the implementation with an Apache 2 open source licence [1].

Python frontend. The frontend is a thin Python API that, by default, abstracts all the details from developers but also provides the ability to override internal configuration parameters (e.g., optimization algorithm) through parameters to the API. This flexibility is important because machine learning requires a high degree of experimentation. The frontend also provides a user interface running on Plotly [34] for users to monitor the progress of the workloads and start/stop tasks.

[1] https://github.com/ucbrise/cirrus

Component	Lang.	LOC
Data store	C++	1070
Client backend	Python	977
Python frontend and shared components	Python/C++	7017
Worker runtime	C++	1065

Table 4: Cirrus components.

Client backend. The client backend abstracts the management of lambdas from the frontend algorithms. Internally, the client backend keeps a list of the lambdas currently active and keeps a list of connections to the AWS Lambda API (each one used to launch a lambda). Lambdas that are launched during training are relaunched automatically when their lifetime terminates (every 15 minutes). Launching hundreds of lambdas quickly from a single server can be challenging due to the specifics of the lambda API. To address this, the backend keeps a pool of threads that can be used for responding to requests for new lambda tasks.

Distributed data store. Cirrus's distributed data store provides an interface that supports all the use cases for storing intermediate data in the ML workflow. This interface supports a key-value store interface (*set*/*get*) and a parameter-server interface (*send gradient* / *get model*).

A key goal of our data store is to provide very fast access to shared intermediate data by Cirrus's workers. We implemented several optimizations to achieve this performance goal. First, to update models with high throughput we developed a multithreaded server that distributes work across many cores. We found that utilizing multiple cores allows the datastore to serve 30% more updates per second for a Sparse Logistic Regression workload. However, eventually the server becomes bottlenecked by the network and adding more cores does not improve performance. Second, to reduce pressure on the network links of the store we implement data compression for the gradients and models transferred to/from the store. Our experiments show this optimization reduces the amount of data transferred by 2x. Last, our data store further optimizes communication by sending and receiving sparse gradient and model data structures. This reduces the amount of data to transfer by up to 100x. The modular design of the data store allows users to change, or even add, new ML optimization algorithms (e.g., Adam) easily.

Worker runtime. Cirrus's runtime (Figure 5) provides a) general abstractions for ML computations and b) data primitives to access training data, parameter models and intermediate results. These can be used to add new ML models to Cirrus. To ease the development of new algorithms, the runtime provides a set of linear algebra routines. Initial versions of Cirrus used external linear algebra libraries such as Eigen [30] for gradient computations. To reduce the amount of time spent serializing and deserializing data to be processed by Eigen, we ended up developing our own routines. For data access, the runtime provides a minibatch-based iterator backed by a local memory ring-buffer that allows workers to access training minibatches with low latency. In addition, it provides an efficient API to communicate with the distributed data store.

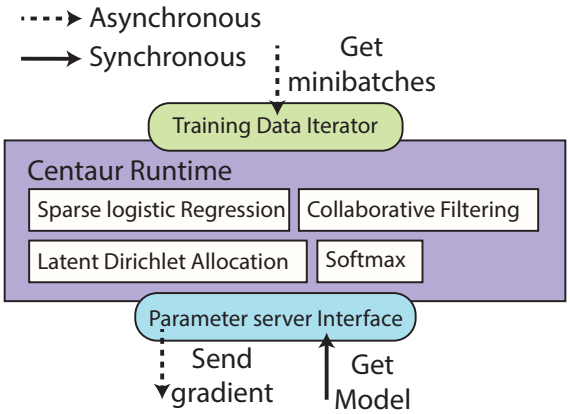


Figure 5: Cirrus worker runtime. Minibatches are asynchronously prefetched and cached locally within each lambda's memory (depending on the size of the lambda used). Similarly, gradients are sent to the parameter server asynchronously. The model is retrieved from the parameter server synchronously every iteration.

6 EVALUATION

This section compares Cirrus with tools specialized for ML training under traditional execution environments (§6.2 and §6.3) and with PyWren, a framework for general serverless infrastructure computation (§6.5). We complement our evaluation with a discussion on Cirrus's scalability (§6.4), an ablation study (§6.5), and a microbenchmark (§6.6).

6.1 Methodology

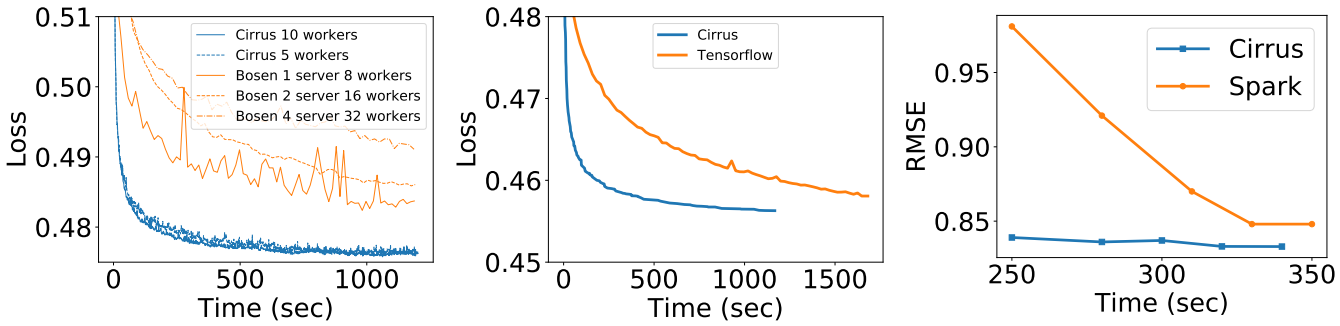
For our evaluation, we ran serverless systems, Cirrus and PyWren, on AWS Lambda [2]. In all experiments with serverless systems the training dataset was stored on AWS S3. Unless otherwise noted, we used the largest-sized lambdas (3GB of memory) for better performance. In one of the experiments (§6.5) we used a *cache.r4.16xlarge* Redis AWS instance (32 cores, 203GB of RAM, 10 Gbps NIC) to store intermediate results used by PyWren. To deploy Cirrus's distributed datastore, unless otherwise noted, we used a single *m5.large* instance (2 CPUs, 8GB of RAM, 10Gbps NIC). The datastore and Cirrus's workers were all deployed on the same AWS region (us-west-2).

To run Apache Spark we deployed three *m5.xlarge* (4 cores, 16.0GB of RAM, and 10 Gbps NIC) VMs from AWS. To run Bosen we used a varying number of *m5.2xlarge* Amazon AWS instances. For both systems we split the datasets evenly across the VMs before the start of the experiments.

For the Sparse Logistic Regression and Collaborative Filtering problems we used Cirrus's asynchronous SGD [44] implementation. For these experiments we configured all the systems to use a minibatch size of 20 samples.

6.2 Sparse Logistic Regression

We compared Cirrus's Sparse Logistic Regression implementation against two frameworks specialized for VM-based ML training: TensorFlow [19], and Bosen [49].



(a) Bosen **(b) Tensorflow** **(c) Spark**

Figure 6: (a) Loss over time comparison between Bosen and Cirrus with different setups. The best loss=0.485 achieved by Bosen is reached by Cirrus at least 5x faster (200sec vs. 1000sec). Cirrus can converge within the lifetime of one or two lambdas (300-600sec) faster and with lower loss than state-of-the-art ML training frameworks. (b) Convergence vs Time curve for Tensorflow Criteo_tft benchmark [17] and Cirrus. Tensorflow was executed on a 32-core node (performed better than on 1 Titan V GPU) and Cirrus ran in 10 lambdas. We implemented the same dataset preprocessing in Cirrus. (c) Curve showing the RMSE over time for Spark (ALS) and Cirrus when running the Netflix dataset until convergence. Spark spends the first 4 minutes processing data and terminates after converging (RMSE=0.85) in 5 iterations of ALS. Cirrus converges more quickly to a lower RMSE (0.833).

TensorFlow is a general-purpose dataflow engine used for ML computations. Bosen is a distributed and multi-threaded parameter server, developed at CMU and commercialized by Petuum [50], that is optimized for large-scale distributed clusters and machine learning algorithms with stale updates.

Logistic regression is the problem of computing the probability of any given sample belonging to two classes of interest. In particular, for our evaluation we compute the probability that a website ad is clicked, and evaluate the learning convergence as a function of time. We use the Criteo display ads dataset [3]. This dataset contains 45M samples and has 11GB of size in total. Each sample contains 13 numerical and 26 categorical features. Before training we normalized the dataset and we hashed the categorical features to a sparse vector of size 2^{20}. This hashing results in a highly-sparse dataset – all the systems we run in this experiment have support for sparse data. Each training sample has a 0/1 label indicating whether an ad was clicked or not.

To evaluate Bosen we use 1, 2 and 4 *m5.2xlarge* Amazon AWS instances (each with 8 CPUs and 32GB of RAM). We configure it to use all the 8 available cores on each instance. For each experiment with Bosen, we partitioned the dataset across all machines. To evaluate Cirrus we used Amazon AWS lambdas for workers, m5.large instances (2 CPUs, 8GB of RAM, 10Gbps networks) for the parameter server, and AWS S3 storage for training data and periodic model backups. We report the best result obtained from trying a range of learning rates for both systems. For Bosen, we only vary learning rate and number of workers. All the other configuration parameters were left with default values.

Figure 6a shows the logistic test loss achieved over time with varying numbers of servers (for Bosen) and AWS lambdas (for Cirrus). The loss was obtained by evaluating the trained model on a holdout set containing 50K samples. We find that Cirrus is able to converge significantly faster than Bosen. For instance, Cirrus with 10 lambdas (size 2048MB) reaches a loss of 0.49 and 0.48 after 12 and 46 seconds, respectively. On the other hand, Bosen with 2 servers (16 threads) reaches this loss only after 600 seconds and a loss of 0.48 after 4600 seconds. Through profiling, we found that Bosen's performance suffers from contention to a local cache shared by all workers that aggregates gradients before sending them to the parameter server; this design leads to slower convergence.

In Figure 6b, we compare Cirrus with TensorFlow using the same dataset and the same pre-processing steps. Similarly, Cirrus reaches the best loss TensorFlow achieves by t = 1500s 3.75x faster (by t = 400 seconds).

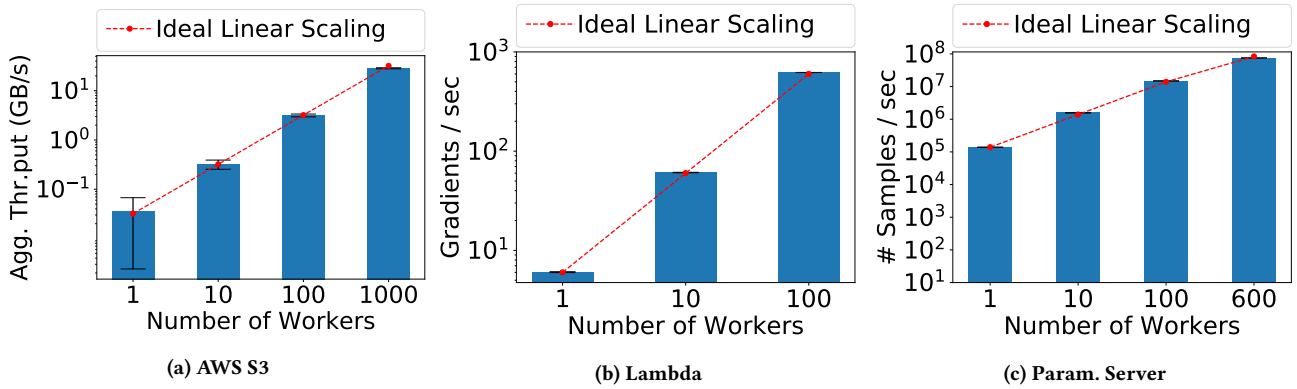
6.3 Collaborative Filtering

We also evaluate a second model supported by Cirrus: collaborative filtering. Collaborative filtering is often used to make recommendations to a user, based on her and other users' preferences.

We evaluate Cirrus on the ability to predict the ratings users give to other movies they have not seen. We use the Netflix dataset [13] for this experiment.

To solve this problem, Cirrus implements a collaborative filtering SGD learning algorithm that builds a matrix U of size $n_{users} \times K$ and a matrix M of size $n_{movies} \times K$. We chose K to be 10. Our metric of success for this experiments is the rate of convergence (Figure 6c). This dataset contains 400K users and 17K movies and a total of 100 million user-movie ratings.

We find that Cirrus converges faster and achieves a lower test loss than Spark (Figure 6c). Through profiling, we observe that Spark's ALS implementation suffers from expensive RDD overheads, as Spark loads the whole dataset to memory. This causes Spark to spend more than 94% of the time doing work not directly related to training the model. In contrast, Cirrus streams data from S3 continuously to the workers which allows them to start computing right away.



(a) AWS S3 (b) Lambda (c) Param. Server

Figure 7: Scalability of AWS storage (GB/s), AWS serverless compute (gradients/sec), and Cirrus data store (samples/sec). Each worker consumes 30MB/s of training data.

6.4 Scalability

Finally, scalability is an important property for ML workflow support. We show that the choice of serverless infrastructure rests on its impressive scalability (Figure 7) and that Cirrus scales linearly leveraging that advantage. We accomplish this level of scalability by designing the system to scale across 3 dimensions: storage of training data with S3, compute with lambdas, and shared memory with a distributed parameter server.

Scaling serverless compute for high-intensity ML workloads can be challenging as S3 quickly becomes the bottleneck at a high number of requests per second [36].

Storage scalability. Cirrus addresses this issue by splitting training datasets in S3 into medium-sized objects. We use 10MB objects because we find this size achieves good network utilization, while being small enough for even the smallest sized lambdas. By using large objects we reduce the number of requests per second. As a result, we are able to scale S3 throughput linearly to 1000 of Cirrus workers (Figure 7a), when each worker consumes 30MB/s of training data from S3.

While doing this experiment, we found that when launching a large number of lambdas (e.g., >3K) AWS Lambda often times takes tens of minutes until all the lambdas have started. This suggests the need for frameworks such as Cirrus that can handle the unpredictable arrival of workers.

Compute scalability. A second challenge is to be able to run a large number of workers that perform a compute-intensive operation such as the computation of a model gradient. We did an experiment to figure out how well the Cirrus workers can scale when the training dataset is backed by S3 (Figure 7b) – with no synchronization of models and parameters (we explore that case in the next experiment). Cirrus can achieve linear compute scalability by streaming input training data and computing gradients in parallel.

Parameter server scalability. At the parameter server level, the challenge arises from the limited network bandwidth of each VM as well as the compute required to update the model and serve requests from workers. Cirrus solves this problem with 1) model sharding, 2) sparse gradients/models, 3) data compression, and 4) asynchronous communication. Cirrus achieves linear scalability up to 600 workers (Figure 7c).

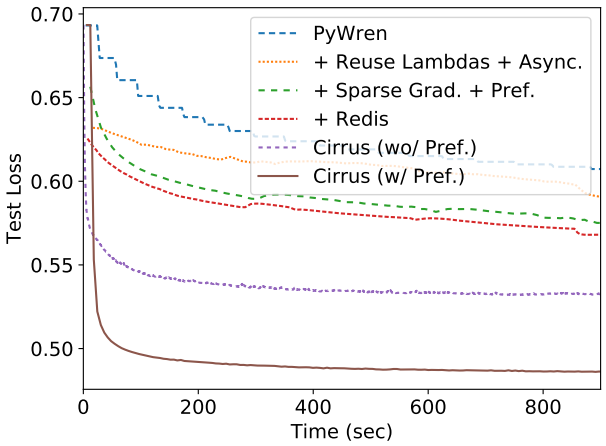
6.5 The Benefits of ML Specialization

To evaluate the advantages of a specialized system for ML, we compare Cirrus against PyWren [36]. PyWren is a map-reduce framework that runs on serverless lambdas. It provides *map* and *reduce* primitives that scale to thousands of workers. These PyWren primitives have been used to implement algorithms in fields such as large-scale sorting, data queries, and machine learning training. PyWren's runtime is optimized to run on AWS Lambdas, the same serverless platform we used for all Cirrus experiments.

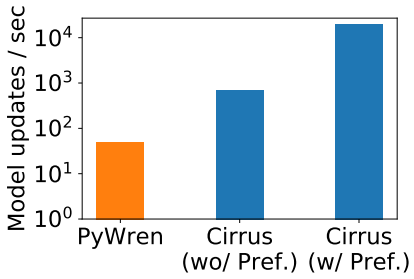
To perform this comparison we initially implemented a synchronous SGD training algorithm for Logistic Regression on PyWren. Our code uses PyWren to run a number of workers on lambdas (*map* tasks) and the gradients returned by these tasks are aggregated and then used to update the model (in the PyWren driver). The driver iteratively updates and communicates the latest model to workers through S3.

We take a step further and implement a set of optimizations to our PyWren baseline implementation. We compute the loss curve of the system after implementing each optimization (Figure 8a). The optimizations we implement are (cumulatively): (1) each lambda invocation executes multiple SGD iterations + asynchronous SGD, (2) minibatch prefetching and sparse gradients, (3) using a low-latency store (Redis) instead of S3. Additionally, we also evaluate the contribution of the Cirrus's data prefetching iterator to the performance of Cirrus.

Despite the optimizations that we implemented using Pywren, which improved its average time per model update by 700x (from 14 seconds to 0.02) it still achieves a significantly lower number of model updates per second (Figure 8b) and converges significantly slower (Figure 8a) than Cirrus. We attribute this performance gap to the careful design and high-performance implementation of Cirrus that specializes both for the serverless environments (e.g., data prefetching, lightweight runtime) and for the iterative ML training workloads with stringent performance requirements (e.g., sparse gradients, optimized data copying, multi-threaded data store).



(a) Convergence over time.



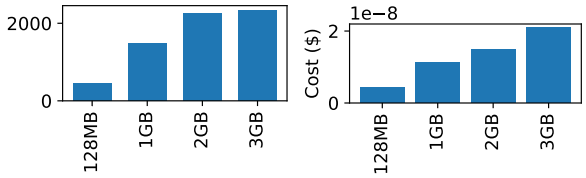
(b) Model updates per second.

Figure 8: PyWren and Cirrus's performance on a Sparse Logistic Regression workload when running on 10 lambdas. Cirrus achieves 2 orders of magnitude more model updates due to a combination of prefetching, reusing lambdas across model training iterations, and efficient model sharing through Cirrus's fast data store. In particular, training data prefetching masks the high access latency to S3 which results in an additional 10x more updates/second.

6.6 Microbenchmark

One of the system parameters Cirrus abstracts from users is the size of the lambda functions used for Cirrus's workers. Larger lambdas – measured in the size of available memory – result in more available CPU power and consequently in higher performance.

To understand how the performance of Cirrus varies with the size of the lambda functions, we performed the Sparse Logistic Regression workload (Section §6.2) with four lambda sizes (128MB, 1GB, 2GB and 3GB). The performance of each individual Cirrus's worker – measured in updates per second – with varying lambda sizes can be seen in Figure 9. Our results show that Cirrus's workers running on bigger lambdas can achieve a higher throughput. However, when we plot the cost per update with different lambdas sizes we see that small lambdas achieve the best cost per update. This explains why we get the best performance/cost configuration when Cirrus makes use of small lambdas.



(a) Updates/sec **(b) Cost/update**

Figure 9: Number of updates per second and cost per update of a single worker with different lambda sizes. We make an observation that, while cost grows linearly with lambda size, the performance gains are sub-linear. This key enabling insight helps Cirrus tap into significant performance per unit cost gains, leveraging its ability to operate with ultra-lightweight resource footprint.

7 RELATED WORK

Serverless computing. Previous works have proposed high-level frameworks for orchestrating large-scale distributed jobs running on serverless functions. For instance PyWren [36] is a map-reduce framework running on AWS Lambdas that uses S3 as an input/output data storage. ExCamera [29] is a library to leverage lambdas to compute intensive video-encoding tasks in a few minutes. gg [28] is a framework for serverless parallel threads that has been used for software compilation, unit tests, video encoding, and object recognition. These systems focus on supporting embarrassingly parallel jobs that don't require synchronization. Cirrus focuses instead on *ML workloads*, which require synchronization and have more stringent performance requirements. Other work [22, 42, 48] has focused on understanding and improving the performance of serverless architectures. Serverless systems, including our own, build on this work to increase efficiency.

Serverless storage. Several proposals for serverless storage systems have emerged. For instance, Pocket [38, 39] is an elastic storage system for serverless workloads. Unlike Cirrus's data store, Pocket's API is not able to transfer sparse data structures (or multi-get), and does not support ML-specific logic on the server side. These properties are critical to provide high-performance storage for ML serverless workloads.

ML parameter servers. Past works have mostly focused on developing general-purpose large-scale parameter-server systems specialized for commodity cloud hardware. None of these existing systems is a good fit for serverless environments. For instance, Tensorflow's [19] runtime has high memory overhead and Bosen [49] loads all training data into memory. These design choices make these systems inefficient for running in lambdas, which only have available a few hundreds MBs of RAM. Other systems, such as Multiverso [7] and Vowpal Wabbit [20], leverage MPI as a runtime, making them a bad fit for an environment where tasks are ephemeral and need to be terminated and restarted frequently. Last, unlike Cirrus, systems such as [40, 41] shard the training data across all workers. Thus, each worker requires a large amount of local disk capacity or otherwise many server nodes need to be

allocated. Cirrus, on the other hand, has minimal local disk requirements because it continuously streams training data from remote storage.

Other ML Frameworks. General distributed computing frameworks such as Spark [51], and Hadoop [23] have also been used to implement large-scale distributed machine learning algorithms such as those used in our work. In contrast with these systems, Cirrus is optimized for both serverless and machine learning workloads. Cirrus achieves better performance by combining an ultra-lightweight runtime and a scalable distributed data store. Recent work on developing a prototype of Spark on AWS Lambda confirms that porting existing frameworks to lambdas requires significant architectural changes [14]. For instance, the current prototype of Spark on AWS Lambda does not support ML workloads and takes 2 minutes to start a Spark Executor inside the lambda.

Disaggregated architectures Recent work on disaggregated architectures has been proposed by both industry (e.g, HP [33], Intel [12], Huawei [11], and Facebook [1]) and academics (e.g., Firebox [24], Microsoft Research [46], VMWare [21] and others [37]). Disaggregated architectures are a promising path for accelerating large-scale serverless computations, such as ML workflows, through novel hardware/network platforms. For instance, Aguilera et al [21] propose a *refreshable vector* abstraction for keeping a stale data vector cached on each worker. Vectors on each worker get updated through sparse data communication. This abstraction can be used for caching and updating ML models, akin to what Cirrus's software data store interface provides. Similarly, a high-bandwidth high-radix network such as the one proposed by Firebox [24] can accelerate the communication between lambdas. Such architecture can be beneficial for large shuffles and reduces, commonly used during the initial preprocessing phase of the ML workflow. Last, Firebox's heterogeneous architecture enables hardware specialization for the different stages of ML workflows. Serverless systems such as Cirrus can build on top of such hardware architectures.

8 DISCUSSION AND CONCLUSION

This work proposes Cirrus—a distributed ML workflow framework for serverless infrastructure that aims to support and simplify the end-to-end ML user workflow by providing an easy button for ML workflow lifecycle. Serverless compute infrastructures provide desired spatiotemporal properties given the fine-granularity of resource allocation. The scalability of existing serverless infrastructures (e.g., AWS Lambdas) as well as the ease of resource management follows from this one fundamental property. However, it creates a challenge in adoption for ML training frameworks that assume abundant resources (coarse-grain in space) and longer-running compute instances (coarse-grain in time). As such, they either cannot run or suffer in performance when deployed on serverless infrastructures *as is*. Frameworks specialized for general serverless support fail to provide low-overhead, high-throughput runtimes required for iterative ML workflows. Cirrus morphs the benefits of both ML training and serverless frameworks to address key challenges in the adoption of serverless compute for machine learning workflows. Cirrus provides (a) transparent serverless resource management, (b) scalability, and (c) fault-tolerance by design. It obviates the need for expensive over-provisioning with fine-grain resource allocation afforded by serverless infrastructures.

Cirrus leverages a number of properties of serverless disaggregated infrastructure, particularly, the ease of use, low-latency lambda instantiation, and attractive performance per unit cost. Cirrus leverages a number of key observations we make about ML training workloads as well: training data consumption bandwidth is a good fit for streaming bandwidth provided by Amazon's S3, training data access patterns that make it possible to iterate and stream the remote dataset, and the ability to converge with asynchronous gradient updates. The latter makes it possible to deploy the inherently stateful ML training workload on a fleet of ephemeral serverless compute resources and robustly handle their churn. End-to-end ML workflow on serverless infrastructure needs a system that specializes in both. Cirrus outperforms a state-of-the-art ML training framework [49] in terms of time to best convergence as well as performance per unit cost, motivating the need for specialized ML training framework designed specifically to work on serverless infrastructure. Cirrus also outperforms a state-of-the-art general serverless framework [36] on ML training workloads, motivating the need for a specialized serverless framework designed specifically for iterative ML training workloads. Thus, we demonstrate both the need for and the feasibility of a serverless ML training framework that specializes in both, while dramatically simplifying the data scientists' and ML practitioners' model training workflow.

9 ACKNOWLEDGMENTS

We thank Jeff Yu, and Ryan Yang for implementing distributed LDA, and for numerous improvements in the frontend, respectively. We also thank our shepherd, Alvin Cheung, and our reviewers for their feedback. This work was supported by FCT (Portuguese Foundation for Science and Technology) under the PhD grant SFRH/BD/115046/2016.

REFERENCES

[1] 2013. Disaggregated Rack. http://www.opencompute.org/wp/wp-content/uploads/2013/01/OCP_Summit_IV_Disaggregation_Jason_Taylor.pdf. Open-Compute Summit.
[2] 2014. AWS Lambda. https://aws.amazon.com/lambda/.
[3] 2014. Criteo Dataset. http://labs.criteo.com/2014/02/kaggle-display-advertising-challenge-dataset/.
[4] 2014. Display Advertising Challenge. https://www.kaggle.com/c/criteo-display-ad-challenge.
[5] 2014. IBM Functions. https://console.bluemix.net/openwhisk/.
[6] 2016. Azure Functions. https://azure.microsoft.com/en-us/services/functions/.
[7] 2016. Multiverso. https://github.com/Microsoft/Multiverso.
[8] 2017. AWS Greengrass. https://aws.amazon.com/greengrass/.
[9] 2017. Cloudflare Workers. https://www.cloudflare.com/products/cloudflare-workers/.
[10] 2017. Google Cloud Functions. https://cloud.google.com/functions/.
[11] 2017. Huawei DC 3.0. www.huawei.com/ilink/en/download/HW_349607&usg=AFQjCNE0m-KD71dxJeRf1cJSkNaJbpNgnw&sig2=opyc-KxWX3Vb7Jj11dyaMA. [Online; accessed 20-Jan-2017].
[12] 2017. Intel Rack Scale Architecture. http://www.intel.com/content/www/us/en/architecture-and-technology/rack-scale-design-overview.html.
[13] 2017. Netflix Dataset. https://www.kaggle.com/netflix-inc/netflix-prize-data.
[14] 2017. Qubole Announces Apache Spark on AWS Lambda. https://www.qubole.com/blog/spark-on-aws-lambda/.
[15] 2018. Alibaba Functions. https://www.alibabacloud.com/products/function-compute.
[16] 2018. Apache OpenWhisk. https://openwhisk.apache.org/.
[17] 2018. Google cloudml-samples. https://github.com/GoogleCloudPlatform/cloudml-samples.
[18] 2018. OpenFaaS. https://www.openfaas.com/.

[19] Martín Abadi, Paul Barham, Jianmin Chen, Zhifeng Chen, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Geoffrey Irving, Michael Isard, et al. [n. d.]. TensorFlow: A System for Large-Scale Machine Learning.

[20] Alekh Agarwal, Olivier Chapelle, Miroslav Dudík, and John Langford. 2014. A reliable effective terascale linear learning system. *The Journal of Machine Learning Research* 15, 1 (2014), 1111–1133.

[21] Marcos K. Aguilera, Kimberly Keeton, Stanko Novakovic, and Sharad Singhal. 2019. Designing Far Memory Data Structures: Think Outside the Box. In *Proceedings of the Workshop on Hot Topics in Operating Systems (HotOS '19)*. ACM, New York, NY, USA, 120–126. https://doi.org/10.1145/3317550.3321433

[22] Istemi Ekin Akkus, Ruichuan Chen, Ivica Rimac, Manuel Stein, Klaus Satzke, Andre Beck, Paarijaat Aditya, and Volker Hilt. 2018. SAND: Towards High-Performance Serverless Computing. In *2018 USENIX Annual Technical Conference (USENIX ATC 18)*. USENIX Association, Boston, MA, 923–935. https://www.usenix.org/conference/atc18/presentation/akkus

[23] Apache. [n. d.]. Apache Hadoop. http://hadoop.apache.org.

[24] Krste Asanović. 2014. FireBox: A Hardware Building Block for 2020 Warehouse-Scale Computers. FAST.

[25] Chih-Chung Chang and Chih-Jen Lin. 2011. LIBSVM: a library for support vector machines. *ACM transactions on intelligent systems and technology (TIST)* 2, 3 (2011), 27.

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

[27] Christina Delimitrou and Christos Kozyrakis. 2014. Quasar: Resource-efficient and QoS-aware Cluster Management. In *Proceedings of the 19th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS '14)*. ACM, New York, NY, USA, 127–144. https://doi.org/10.1145/2541940.2541941

[28] Sadjad Fouladi, Francisco Romero, Dan Iter, Qian Li, Shuvo Chatterjee, Christos Kozyrakis, Matei Zaharia, and Keith Winstein. 2019. From Laptop to Lambda: Outsourcing Everyday Jobs to Thousands of Transient Functional Containers. In *2019 {USENIX} Annual Technical Conference ({USENIX}{ATC} 19)*. 475–488.

[29] Sadjad Fouladi, Riad S. Wahby, Brennan Shacklett, Karthikeyan Vasuki Balasubramaniam, William Zeng, Rahul Bhalerao, Anirudh Sivaraman, George Porter, and Keith Winstein. 2017. Encoding, Fast and Slow: Low-Latency Video Processing Using Thousands of Tiny Threads. In *14th USENIX Symposium on Networked Systems Design and Implementation (NSDI 17)*. USENIX Association, Boston, MA, 363–376. https://www.usenix.org/conference/nsdi17/technical-sessions/presentation/fouladi

[30] Gaël Guennebaud, Benoît Jacob, et al. 2010. Eigen v3. http://eigen.tuxfamily.org.

[31] Joseph M Hellerstein, Jose Faleiro, Joseph E Gonzalez, Johann Schleier-Smith, Vikram Sreekanti, Alexey Tumanov, and Chenggang Wu. 2018. Serverless computing: One step forward, two steps back. *arXiv preprint arXiv:1812.03651* (2018).

[32] Scott Hendrickson, Stephen Sturdevant, Tyler Harter, Venkateshwaran Venkataramani, Andrea C Arpaci-Dusseau, and Remzi H Arpaci-Dusseau. [n. d.]. Serverless computation with openlambda. *Elastic* 60 ([n. d.]), 80.

[33] HP. 2017. HP The Machine. https://www.labs.hpe.com/the-machine. [Online; accessed 20-Jan-2017].

[34] Plotly Technologies Inc. 2015. Collaborative data science. https://plot.ly

[35] Eric Jonas, Johann Schleier-Smith, Vikram Sreekanti, Chia-Che Tsai, Anurag Khandelwal, Qifan Pu, Vaishaal Shankar, Joao Carreira, Karl Krauth, Neeraja Yadwadkar, et al. 2019. Cloud Programming Simplified: A Berkeley View on Serverless Computing. *arXiv preprint arXiv:1902.03383* (2019).

[36] Eric Jonas, Shivaram Venkataraman, Ion Stoica, and Benjamin Recht. 2017. Occupy the Cloud: Distributed Computing for the 99%. *CoRR* abs/1702.04024 (2017). http://arxiv.org/abs/1702.04024

[37] Ana Klimovic, Christos Kozyrakis, Eno Thereska, Binu John, and Sanjeev Kumar. 2016. Flash Storage Disaggregation. In *Proceedings of the Eleventh European Conference on Computer Systems (EuroSys '16)*. ACM, New York, NY, USA, Article 29, 15 pages. https://doi.org/10.1145/2901318.2901337

[38] Ana Klimovic, Yawen Wang, Christos Kozyrakis, Patrick Stuedi, Jonas Pfefferle, and Animesh Trivedi. 2018. Understanding Ephemeral Storage for Serverless Analytics. In *2018 USENIX Annual Technical Conference (USENIX ATC 18)*. USENIX Association, Boston, MA, 789–794. https://www.usenix.org/conference/atc18/presentation/klimovic-serverless

[39] Ana Klimovic, Yawen Wang, Patrick Stuedi, Animesh Trivedi, Jonas Pfefferle, and Christos Kozyrakis. 2018. Pocket: Elastic ephemeral storage for serverless analytics. In *13th {USENIX} Symposium on Operating Systems Design and Implementation ({OSDI} 18)*. 427–444.

[40] Mu Li, David G. Andersen, Jun Woo Park, Alexander J. Smola, Amr Ahmed, Vanja Josifovski, James Long, Eugene J. Shekita, and Bor-Yiing Su. 2014. Scaling Distributed Machine Learning with the Parameter Server. In *Proceedings of the 11th USENIX Conference on Operating Systems Design and Implementation (OSDI'14)*. USENIX Association, Berkeley, CA, USA, 583–598. http://dl.acm.org/citation.cfm?id=2685048.2685095

[41] Mu Li, David G Andersen, Alexander J Smola, and Kai Yu. 2014. Communication efficient distributed machine learning with the parameter server. In *Advances in Neural Information Processing Systems*. 19–27.

[42] Edward Oakes, Leon Yang, Dennis Zhou, Kevin Houck, Tyler Harter, Andrea Arpaci-Dusseau, and Remzi Arpaci-Dusseau. 2018. SOCK: Rapid Task Provisioning with Serverless-Optimized Containers. In *2018 USENIX Annual Technical Conference (USENIX ATC 18)*. USENIX Association, Boston, MA, 57–70. https://www.usenix.org/conference/atc18/presentation/oakes

[43] Pitch Patarasuk and Xin Yuan. 2009. Bandwidth optimal all-reduce algorithms for clusters of workstations. *J. Parallel and Distrib. Comput.* 69, 2 (2009), 117–124.

[44] Benjamin Recht, Christopher Re, Stephen Wright, and Feng Niu. 2011. Hogwild: A lock-free approach to parallelizing stochastic gradient descent. In *Advances in Neural Information Processing Systems*. 693–701.

[45] Charles Reiss, Alexey Tumanov, Gregory R Ganger, Randy H Katz, and Michael A Kozuch. 2012. Heterogeneity and dynamicity of clouds at scale: Google trace analysis. In *Proceedings of the Third ACM Symposium on Cloud Computing*. ACM, 7.

[46] Microsoft Research. 2017. Rack Scale Computing. https://www.microsoft.com/en-us/research/project/rack-scale-computing/.

[47] Alexander Sergeev and Mike Del Balso. 2018. Horovod: fast and easy distributed deep learning in TensorFlow. *arXiv preprint arXiv:1802.05799* (2018).

[48] Liang Wang, Mengyuan Li, Yinqian Zhang, Thomas Ristenpart, and Michael Swift. 2018. Peeking Behind the Curtains of Serverless Platforms. In *2018 USENIX Annual Technical Conference (USENIX ATC 18)*. USENIX Association, Boston, MA, 133–146. https://www.usenix.org/conference/atc18/presentation/wang-liang

[49] Jinliang Wei, Wei Dai, Aurick Qiao, Qirong Ho, Henggang Cui, Gregory R Ganger, Phillip B Gibbons, Garth A Gibson, and Eric P Xing. 2015. Managed communication and consistency for fast data-parallel iterative analytics. In *Proceedings of the Sixth ACM Symposium on Cloud Computing*. ACM, 381–394.

[50] Eric P Xing, Qirong Ho, Wei Dai, Jin Kyu Kim, Jinliang Wei, Seunghak Lee, Xun Zheng, Pengtao Xie, Abhimanu Kumar, and Yaoliang Yu. 2015. Petuum: A new platform for distributed machine learning on big data. *IEEE Transactions on Big Data* 1, 2 (2015), 49–67.

[51] Matei Zaharia, Mosharaf Chowdhury, Michael J. Franklin, Scott Shenker, and Ion Stoica. 2010. Spark: Cluster Computing with Working Sets. In *Proceedings of the 2Nd USENIX Conference on Hot Topics in Cloud Computing (HotCloud'10)*. USENIX Association, Berkeley, CA, USA, 10–10. http://dl.acm.org/citation.cfm?id=1863103.1863113