Practical and Scalable Serverless Computing

by

Joao Carlos Menezes Carreira

A dissertation submitted in partial satisfaction of the

requirements for the degree of

Doctor of Philosophy

in

Computer Science

in the

Graduate Division

of the

University of California, Berkeley

Fall 2020

Practical and Scalable Serverless Computing

Abstract

Practical and Scalable Serverless Computing

by

Joao Carlos Menezes Carreira

Doctor of Philosophy in Computer Science

University of California, Berkeley

Professor Randy Katz, Co-chair

Professor Pedro Fonseca, Co-chair

Serverless computing is a new paradigm for developing cloud applications popularized by Amazon's AWS Lambda service. In serverless computing, applications are decomposed into fine-grained serverless functions developed in high-level languages such as Python and Javascript. Developers can then submit these functions to serverless providers which then deploy and execute them. Unlike VM-based platforms, in serverless computing applications run inside containerized execution environments called *lambdas* or *lambda functions*, a term popularized by AWS Lambda. Due to the lightweight and fine-grained nature of lambdas, serverless computing can provide higher elasticity and higher resource utilization. Furthermore, in serverless computing infrastructure and operational aspects of running applications in the cloud are delegated to the cloud provider, and this relieves developers from many complex and onerous tasks. This paradigm shift towards serverless computing is poised to radically change the way developers build cloud applications.

We identify two main challenges with leveraging serverless computing for highly-distributed applications, such as Big Data Analytics and Machine Learning. The first challenge has to do with automatic management of resources through higher-level abstractions for serverless applications. The second has to do with the performance and scalability of distributed network communication on serverless platforms. In this thesis we present two systems that tackle both of these challenges. We solve the first one with a system, Cirrus, for automatic serverless ML end-to-end workflows. We solve the second one with Zip, a system that provides high-performance and scalable distributed primitives for inter-lambda serverless communication.

In this thesis we show that it is possible to provide simple APIs to developers with significantly better performance than today's approaches. For instance, Cirrus provides 2 orders of magnitude more updates per second in model training than when using PyWren, a

MapReduce serverless framework, because it provides a high-level API backed by a highly optimized backend for ML tasks. Similarly, Zip provides 1.3-12x speedup for different communication patterns compared to the next best alternative, using a memory-backed store for inter-lambda communication.

To my parents and brother.

Contents

List of Figures

List of Tables

Acknowledgments

I am deeply indebted to my advisors, Randy Katz and Pedro Fonseca, for their guidance during my PhD. Randy taught me how to methodically approach systems research through careful systems design and evaluation methodologies. Pedro's constant drive to continuously seek simple answers to complex questions taught me how to approach research problems. To them I owe a great deal.

None of this work is possible without the help of Alexey Tumanov. Alexey was one of my first mentors in the RISELab. We first started doing research in the area of disaggregated resources and eventually pivoted towards the topic of this thesis, serverless computing. Alexey's contagious energy and drive taught me one of the most important lessons in every researcher's career: to persevere after every rejection, never losing sight of the end goal.

One of the most rewarding aspects of my PhD was the opportunity to mentor brilliant undergraduate and master's students at UC Berkeley, Andrew Zhang, Andy Wang, Jeff Yu, Neel Somani, Nikhil Athreya, Ryan Yang, Shea Conlon, Tyler Davis. It was an immense satisfaction to work with them.

I am also indebted to Rodrigo Rodrigues for giving me the opportunity to do research in the Max Plank Institute for Software Systems in Saarbrücken, Germany. It was there that I took the very first steps in systems research. This experience has impacted my professional trajectory more than anything else.

I would like to acknowledge the Portuguese funding institution FCT (Fundação para a Ciência e a Tecnologia) for supporting my research.

Lastly, I dedicate this dissertation to my parents and brother for their endless support and encouragement.

Chapter 1

Introduction

1.1 Roadmap

In this chapter we introduce the main problem we solve in this dissertation: how to enable simple and high-performance serverless computing for distributed applications. In Section 1.2 we provide a brief introduction to cloud computing and identify two problems with the cloud model: complexity of resource management and the poor fit for highly interactive workloads. In Section 1.3, we briefly introduce serverless computing and why it is a promising model to address these problems. In Section 1.4 we identify challenges related to developing distributed applications for serverless computing. In Section 1.5 we present the thesis of this dissertation. In Section 1.6 we present the roadmap for the rest of this document.

1.2 The Cloud Revolution

The advent of Cloud computing in the late 2000s arose from the need to reduce the complexity of managing physical hardware resources by developers and internet companies. Early cloud providers, such as Amazon AWS [8], for the first time started providing hardware resources as a utility to developers through an Infrastructure as a Service model based on virtualized environments (Virtual Machines). The cloud revolution took several years of maturation but eventually cloud computing became a great success story. The Cloud brought about a number of radical changes to developers, such as an on-demand resource reservation model, greater cost savings through economies of scale and simpler infrastructure operations [44].

While the cloud revolution has produced great strides in software development at large, infrastructure operations remains a significant challenge for the development of applications. For instance, to deploy an application in the cloud, developers still have to reserve individual VMs, configure them with the necessary software dependencies, deploy their code and data, and finally continuously monitor the VMs. These steps are highly onerous and time-consuming to developers and impact their productivity.

Furthermore, while cloud platforms can scale to tens of thousands of VMs, they are not a good fit for highly elastic workloads. For instance, for interactive use cases such as data queries on a database generated by a business analyst, tasks need to be executed in a short amount of time to provide good response times. This requires that the underlying platform provides fast allocation and invocation of these tasks. When using VM-based platforms, developers have to provision a number of idle servers that are in standby, ready to serve any new requests. In such situations developers have to trade higher cost, due to underutilized resources, for better response times.

1.3 The Rise of Serverless Computing

Serverless computing, an emerging paradigm for the Cloud, is a step towards reducing the complexity of managing cloud infrastructure. The serverless computing model achieves this by (a) leveraging lightweight and short-lived strongly isolated environments called *lambda functions* (or just *lambdas*) and (b) delegating infrastructure operations to the cloud provider. In serverless computing, developers design their applications as individual functions in high-level languages such as Python or JavaScript. Developers then submit these functions to the cloud to be executed. The cloud provider then schedules and executes these functions on one of its VMs. Alternatively, developers can hook these functions to specific events (e.g., a file gets uploaded) that triggers an execution of the function. Serverless computing is gaining rapid adoption, and is available through many commercial [17, 48, 20, 6, 30, 95, 101, 92] and open-source platforms employed in private clouds [10, 54, 79, 68].

The delegation of resource management to the cloud provider and the use of lightweight runtimes are critical aspects of serverless computing. First, in serverless computing, the provider is responsible for all the steps necessary to run the developer's code from the moment it is submitted to the cloud provider. This obviates the need for developers to spend their time in onerous tasks related to resource management. For instance, serverless computing platforms allow on-demand fine-grained elasticity to serve requests/events without the intervention of the developer. Furthermore, because resources are only reserved when needed, developers don't need to pay for resources when they are not being used. Secondly, serverless lambda use less resources than their VM counterparts and are executed inside lightweight isolated environments. This means that serverless platforms are more elastic than typical VM platforms. For instance, in today's serverless platforms, such as AWS Lambda [17] we can launch thousands of lambdas in just a few seconds. This elasticity is very valuable for highly interactive workloads that can require large number of resources, such as data exploration workloads.

Despite the significant benefits of serverless computing, we argue that it has not yet fulfilled its promise as a simple and scalable computing platform for distributed workloads. First, current serverless computing platforms lack simple abstractions for automatic resource management of workloads such as ML workflows. Second, modern serverless computing platforms lack primitives for high-performance distributed communication between lambdas.

In the next section we explore these challenges in more detail.

1.4 The Missing Pieces in Serverless

1.4.1 Abstractions for State/Compute Orchestration

Serverless computing takes a step towards the disaggregation of storage and compute resources. In terms of local storage, serverless functions have a limited amount of memory (few GBs). Hence, developers resort to external storage system as a way to expand the available storage for their working data. While the use of external storage is often a good strategy for storing intermediate data, developers should make use of external storage judiciously due to the extra read/write overheads when compared to local memory. For instance, in our own system Cirrus, we mitigate these overheads by performing the computation of gradients during ML training in parallel with the reading of data from remote storage. Similarly. in terms of compute resources, each lambda can only use up to 1 CPU. This means that developers cannot leverage common strategies and algorithms that take advantage of data sharing across multiple cores. Developers are, again, forced to architect their applications to meet this constraint.

The process of designing an application for serverless computing can be daunting due to the complexities that arise from the fine-grained nature of lambdas. Poor system designs for such applications can lead to unnecessary overheads in reading/writing data from external storage, overheads in sending message across lambdas or poor scalability. Serverless developers desperately need good abstractions that can abstract many of the intricacies of managing the state and compute aspects of particular workloads.

1.4.2 Serverless distributed communication

The scalable and elastic nature of serverless computing make it a promising model for distributed computing. However, many distributed workloads require communication and synchronization across workers to execute tasks. For instance, SQL databases and map reduce frameworks make use of distributed shuffles for `GROUP BY` SQL operations [11]. Similarly, many ML training frameworks rely on distributed implementations of stochastic gradient descent (SGD) [22] to train models. SGD is a widely used iterative ML training algorithm that finds good model parameters by iteratively updating an initial model in small increments – *gradients* – towards a local or global minima. The distributed implementation of SGD iteratively performs distributed broadcasts, in which the most up-to-date model is sent to all workers, and distributed reduces, in which the systems sum all the gradients computed in different machines, to train a model. In both workloads, distributed communication is often times the most expensive part of the whole workload.

Adapting such frameworks to run on serverless computing is challenging. First, serverless computing platforms do not allow direct network communication between two lambda

Application Class	Serverless Network Challenges
Real-time video compression	Poor support for fine-grained communication
MapReduce	Data shuffles scale poorly
Linear algebra	Hard to implement efficient broadcast
ML pipelines	Hard to implement efficient broadcast and aggregation
Databases	Lack of support for inbound connectivity

Table 1.1: Major network challenges in serverless computing, identified by Jonas et al [63], for 5 large application classes.

functions. The first generation of serverless platforms were designed for an overly restricted class of applications in which functions perform a specific computation on a given input in isolation. This limitation means that developers have to use external storage systems to pass messages between lambdas. This results in scalability, performance and cost inefficiencies. Second, because serverless platforms abstract infrastructure details from developers, serverless applications cannot take advantage of the topology of the deployment to optimize communications. For instance, two serverless functions always incur the overheads of communicating through an external storage system even when running on the same machine. This limitation prevents the use of optimizations such as local reduces, in which all the workers processes in each machine compute the sum of their gradients before sending the result to another machine through the network.

This problem is a major blocker for the adoption of serverless for wide classes of applications. For instance, the Berkeley View on Serverless Computing [63] analyzes a set of applications (real-time video compression, MapReduce, linear algebra, ML pipelines, and databases), each corresponding to a large class of applications, and lays out the most pressing challenges of porting such applications to serverless (see Table 1.1). For all these applications, distributed communication is listed as one of the main challenging aspects of developing them in serverless.

1.5 This Dissertation

In this dissertation we demonstrate that it is possible to achieve the simplicity of the serverless computing model with significantly better performance. To achieve the simplicity-performance holy grail we present innovations in two areas: (1) workload-specific abstractions for serverless, such as serverless machine learning workflows, and (2) high-level high-performance distributed communication APIs.

To solve the complexity of developing distributed workloads in serverless, we present Cirrus, a high-level API and a system for end-to-end ML workflows. The Cirrus API is optimized for serverless: it provides a system for intermediate data storage, an ultra-lightweight runtime and ML-specific optimizations to mitigate the overheads of distribution in serverless. Even though Cirrus is specialized to ML workflows, our approach can be applied to other workloads.

To solve the problem of performance and scalability overheads in serverless for distributed applications, we present Zip, an API and implementation that provides efficient distributed communication between lambdas for 3 widely used distributed communication patterns: reduce, broadcast and shuffle. Zip's design is an extension to the design of modern serverless platforms, such as Apache Openwhisk [10].

1.6 Dissertation Roadmap

The rest of the dissertation is organized as follows. Chapter 2 describes the background of serverless computing and previous work in the areas of distributed serverless computing, network communication and serverless storage. In this chapter we identify the progress made in these areas of serverless computing, but also what aspects of serverless could be improved.

Chapter 3 describes a solution for automating ML workflows on top of serverless platforms. In this chapter we propose Cirrus, a system that provides a simple ML interface for the different stages of the ML workflows and leverage the properties of serverless to help data scientists and ML researchers.

Chapter 4 describes a solution to the problem of distributed communication in serverless computing. Here we propose Zip, a system that provides a distributed communication API for functions.

Chapter 5 discusses current open challenges of serverless computing and presents some directions for how to address them. In particular, we discuss the overheads that arise from the lack of co-design between the serverless platforms and the runtimes on which the lambdas are executed, and the opportunities for designing interactive computing environments backed by serverless platforms.

Finally, in Chapter 6 we summarize this dissertation and conclude.

Chapter 2

Background and Previous work

2.1 Roadmap

In Section 2.2 we describe the history of the cloud and its different mutations over time. We also provide definitions for the different types of cloud offerings, namely for the IaaS (Infrastructure as a Service), PaaS (Platform as a Service), and SaaS (Software as a Service) models. This section is critical to understand the reasons why cloud computing was such a successful model, and its implications for the software development process for developers.

In Section 2.3 we start by analyzing the serverless computing model in abstract and then dive into how it has been initially instantiated by large cloud providers such as Amazon, Google and Microsoft. In this section we discuss the limitations of these current commercial offerings, and how they impact the design, performance and usability of serverless computing from a developers perspective. This section provides background information to understand how applications can be designed to run on serverless computing, and how developers go about making those design decisions.

In Section 2.4 we discuss the main limitations of the serverless computing model and of today's serverless platforms. These limitations create significant challenges for the development of applications in serverless computing, in particular of distributed applications.

In Section 2.5 we discuss previous work on distributed computing systems for serverless. Here we are particularly interested in workloads that are likely to benefit from the scalability, elasticity and fine-grained resources of serverless, namely data analytics and distributed ML training. For data analytics, serverless computing allows frameworks to elastically adapt their resources to particular stages of the computation. For distributed ML, serverless enables interactive exploration of models and scalability for training.

In Section 2.6 we look into current approaches for moving and storing data in serverless platforms. Due to the stateless nature of serverless lambdas, developers have to rely on external storage systems for the inputs, outputs and intermediate data of their computations. Similarly, developers rely on such systems for communicating data between lambdas. In this section we discuss these different alternatives and their pros and cons.

Last, in Section 2.7 we discuss technical improvements to existing serverless platforms in different aspects such as invocation latency and memory overheads. Related work in this area exposes the gaps between the goals of the serverless computing model and the reality of today's serverless platforms. For instance, the fine-grained nature of lambdas can be a useful property towards providing better resource utilization in the cloud. However, memory overheads that arise due to the strong isolation between lambdas within a machine can negate those benefits.

2.2 Cloud Computing

2.2.1 From IaaS to SaaS

Before the boom of cloud computing (before the late 2000s), internet companies were responsible for procuring, installing and configuring machines, networks, power supplies and all the other necessary equipment necessary to run their online services. These tasks required careful planning months in advance to predict how many resources would be necessary to provide the service to a predicted number of users and traffic. Purchasing and installing hardware also entailed waiting months for the equipment to be received, and weeks of installation, configuration and testing. Only once the service was up and running these companies were able to test whether their product was having the user adoption they had initially predicted. If user adoption was lower than initially expected, online companies would have wasted precious money and time acquiring unnecessary hardware. On the other hand, if user adoption was higher than expected, online companies had to wait a few more months until they had the hardware capacity necessary to adjust to the unexpected demand. This meant that delivery of new software features and services by software developers had to be done in tight synchronization with hardware planning and management teams.

Cloud computing arose from the need to accelerate software delivery by decoupling the software development cycle from the hardware procurement and management cycle. If hardware was a utility resource, developers could tap into it only when needed. Developers would not have to purchase, install and configure racks of compute and storage resources in advance for an online service that could either be a commercial failure or a success.

The first generation of cloud computing offerings from the major cloud providers, Google [45], Amazon [7], and Microsoft [74], were based on the Infrastructure-a-a-Service (IaaS) model. In IaaS, cloud providers host and manage the hardware resources and expose them to developers in virtualized environments called Virtual Machines (VMs). These VMs carve out resources (e.g., CPUs, memories, disks) from underlying physical machines in which they run and expose them to developers, giving them the illusion of ownership of a single independent physical machine.

At the same time that the IaaS model gained popularity, another time of cloud offering called Platform-as-a-Service (PaaS) competed as an alternative approach for developers to move their applications into the cloud. Examples of PaaS products are Google App

Engine [46], Heroku [55], Parse [80], and Firebase [50]. In PaaS, cloud providers provide platforms, i.e., a set of pre-configured environments, libraries, runtimes, services and tools. Developers then develop their applications in these platforms. In this model, unlike in IaaS, developers don't have control over the underlying infrastructure. PaaS can provide lower barriers to entry for developers in the cloud, at the cost of greater lock-in to a particular platform or interface. Over the years, since the inception of the cloud, IaaS has gained greater adoption compared to PaaS. It is possible that developers found the infrastructure-level offerings of cloud providers IaaS a more familiar environment to port their applications to. Furthermore, that approach presented lower risks in the scenario the cloud would not end up gaining enough adoption and success.

Lastly, Software as a Service (SaaS) provides a different offering model for the cloud. In SaaS, users access finished applications through the internet. Unlike PaaS, in SaaS these products provide a set number of features. Examples of SaaS products are Google Workspace [51] and Dropbox [38].

2.3 Serverless Computing

2.3.1 Why Serverless

Despite the wide benefits of cloud computing for developers, the complexity and time-consuming nature of cloud infrastructure management tasks are still a reality in the practice of software development today. For instance, consider the use case proposed by AWS upon the launch of their serverless offering, AWS Lambda [17]. In this use case, a developer wants to develop a simple service that transforms (e.g., compress the image and generate a thumbnail) every image uploaded to a web server. For such a use case, the effort to setup the necessary infrastructure, and guarantee the service availability and scalability, would far outweigh the effort to develop the image transformation functionality. Serverless computing arose from the need to provide developers with an easier way to deploy this type of applications with simple and well-defined inputs and outputs.

2.3.2 Serverless = FaaS + BaaS

The first serverless computing commercial service was AWS Lambda, launched in 2015. In AWS Lambda, users can register pieces of code written in high-level languages and configure them to be executed with certain triggers (e.g., the upload of an image) or to be executed at a simple invocation command by the developer. When such triggers or invocations occur, the cloud provider instantiates a a lambda, in which the code is executed. In this model, lambda functions are only billed for the time they run. FaaS (Functions as a Service) provides a significantly more friendly environment to develop simple applications because the cloud provider automatically handles all aspects of running the lambdas, such as autoscaling, scheduling, fault tolerance, dependencies updates, and security.

	Azure Functions	AWS Lambda	Google Functions	IBM Functions	Cloudflare Workers
Max. Memory	14GB	10GB	2GB	2GB	128MB
Max. Runtime	Unlimited	15 min.	9 min.	10 min	50ms
Max. Number of Concurrent Lambdas	100	Hundreds of Thousands	3000	1000	Unlimited
GPU Support	No	No	No	No	No
Direct λ-to-λ Communication Support	No	No	No	No	No
Max. Function Size	100MB	Terabytes	Hundreds of MBs	48MB	1MB
Languages Natively Supported	C#, JavaScript, F#, Java, PowerShell, Python, TypeScript	Java, Go, PowerShell, Node.js, C#, Python, Ruby	Node.js, Python, Go, Java, .NET	Node.js, Python, PHP	JavaScript, WebAssembly, Python

Table 2.1: Comparison between serverless computing (FaaS) offerings.

FaaS is not the only piece in serverless computing. Cloud providers also offer a set of specialized services in a wide array of applications, such as storage (e.g., AWS S3 [18], AWS Elasticache [13], DynamoDB [14], Cloud Firestore [28], Cloud Pub/Sub [29]), machine learning (e.g., AWS SageMaker [19]), data analytics (e.g., AWS EMR [15], Google BigQuery [47], and Google Cloud Dataflow [27]). These services are categorized as BaaS (Backend as a Service). Even though many of these services were already available when FaaS was just becoming available to the public, they can be considered specialized serverless services because they also abstract developers from the underlying hardware resources they run on. Serverless computing is the composition of the general FaaS platform and the support services of BaaS. For instance, lambda functions often use BaaS storage services such as AWS S3 to store their input and output data. For this reason, serverless computing is considered to be the combination of FaaS and Baas.

2.4 Limitations of Serverless Platforms

In this section we describe the most significant limitations of current serverless platforms: (a) explicit resource management of storage, (b) poor communication scalability, (c) no indirect lambda-to-lambda communication allowed, (d) ad-hoc communication APIs, (e) lack of predictable performance, and (f) lack of hardware accelerators.

Explicit resource management Some external storage systems used for communication in serverless platforms, such as Redis [87], require explicit provisioning of VMs, configuration, deployment, and monitoring. Crucially, the type and quantity of resources provisioned for communication are critical to achieve good performance [84]. However, excessive allocation of resources to these storage systems unnecessarily increases the cost of deployment.

Alternatively, AWS S3 and Pocket can adapt their resources at run-time to meet the needs of the application.

Poor communication scalability Approaches that rely on storage systems, such as AWS S3, Redis, and Pocket, restrict the communication scalability to the scalability of the external storage system. For instance, PyWren has been found to not scale for some data analytics workloads [84] due to throughput throttling in AWS S3 (see Figure 4.1b). Similarly, systems such as Redis or Pocket need careful provisioning to achieve good performance and cost. Accurate provisioning of resources for such systems is hard to accomplish for many workloads. This leads to developers either overprovisioning external storage services, which leads to unnecessary cost, or undeprovisioning which hinders performance.

Indirect lambda-to-lambda communication Today's serverless platforms do not allow direct, lambda-to-lambda connections, thus any message between any two lambdas has to pass through an external storage system before reaching its destination. This means that messages have to traverse 2x network hops which incurs a higher end-to-end latency (see Figure 4.1a).

Ad-hoc communication APIs Today there is no serverless framework/library developers can leverage to perform distributed communication. Abstractions such as network sockets and MPI [41]are widely used for network communication in the cloud but are a poor fit for serverless platforms. For instance, network sockets are a low-level abstraction. This leaves developers with the responsibility of developing non-trivial distributed communication algorithms, and handling connections between ephemeral workers. MPI is a widely used framework that provides higher-level communication abstractions such as distributed reduces and broadcasts. However, MPI does not tolerate changes in the set of workers communicating during runtime. This is an important limitation in the context of serverless computing where lambdas can start and terminate at unpredictable times.

Lack of predictable performance Today's serverless platforms provide little or no performance guarantees for lambda invocations. Performance unpredictability on serverless platforms arises from two aspects: (a) lambda invocation latency, and (b) uncertainty in the underlying resources allocated to each lambda. Regarding invocation latency, we found that for some serverless platforms, such as AWS Lambda, lambdas can take anywhere from less than a second up to more than 1 minute to be launched. Results from ExCamera [42] indicate that this problem arises primarily when the cloud provider cannot reuse active lambdas to serve a request (cold starts). The second factor has to do with the uncertainty regarding which particular hardware resources are allocated to each lambda. Today's cloud providers do not guarantee any specific resources to users, which providers them with greater flexibility to utilize spare resources in the datacenter.

Lack of hardware accelerators Serverless platforms today do not allow developers to use hardware accelerators, such as GPUs, for their computations. This prevents serverless from being used for applications such as ML training and ML serving of deep learning networks.

2.5 Serverless Distributed Computing

2.5.1 Data Analytics: Map-Reduce and DAG computing

The first generation of tools and frameworks for distributed computation on serverless platforms focused on providing push-button solutions for embarrassingly parallel computations in the cloud [64, 31, 40, 75]. More concretely, these frameworks provided a simple Map Reduce interface for serverless platforms. For instance, PyWren, the first Map Reduce framework for serverless, provides a *map* and a *reduce* APIs. The *map* allows developers to run a Python function on a particular input, and the *reduce* allows the computation of a reduce function on the results of previous map invocations (see Figure 2.1). Other frameworks with similar designs, such as Corral and FaastJS, have also been proposed.

```
def map_square_function(value):
    return value * value

def reduce_sum_function(squares):
    return sum(squares)

exec = pywren.default_executor()
futures = exec.map(map_square_function, [x for x in _ range(1, 101)])

result = exec.reduce(reduce_sum_function, futures).result()
```

Figure 2.1: PyWren example that computes the sum of squares of numbers from 1 to 100. The example showcases PyWren's *map* and *reduce* APIs for distributed computations.

When using a map-reduce framework such as PyWren, the developer writes an application that calls the *map* and *reduce* PyWren interface functions to achieve the desired computation. Upon calling PyWren's *map* function, PyWren launches the required number of lambdas in AWS Lambda to run the map function. For instance, in Figure 2.1, PyWren runs *map_square_function* 100 times in separate lambdas to process the 100 integers from the input. Each one of these map invocations generates a result that is represented by the *future* object in the application's code. This *future* can then be passed to a the *reduce* PyWren function to apply a function to the output of all the lambdas, in the example a sum of the map outputs. PyWren greatly simplifies the process of developing and deploying applications on serverless. As the example illustrates, PyWren is responsible for launching the lambdas, passing the input data to the lambdas, getting the result of each lambda and running the reduce computation locally.

Other systems such as Lambada [75] extend the map-reduce interface of PyWren and provide a richer DAG-based computational model. For instance, Lambada provides primitives for explicitly loading data from AWS S3 and allows more types of computations, such as filters (see Example 2.2). In contrast with the PyWren model, in Lambada developers compose different computations to build a pipeline. This allows the framework to understand the end-to-end execution of the application, and thus allows it to platform-specific optimizations. For instance, DAG-based systems such as Lambada can reorder and merge operations to reduce the number of lambda invocations and amount of data transfers between the lambda and external storage.

```
# count number of rows with 2nd column > 1
data = lambda.from_parquet("s3://path_to_data")
             .filter(lambda x: x[0] > 1) # filter rows
             .map(lambda x : 1) # filter 2nd column
             .reduce(lambda x, y: x+y)
```

Figure 2.2: Lambada DAG-based example. Lambada enables serverless-specific optimizations through composition of tasks.

Alternatively, systems such as Lambdata [97] provide APIs for developers to make their data access patterns explicit to the serverless platforms. This allows the platform to cache data within each lambda, co-locate lambdas that access the same data, and enable pipelining between stages of a computation. The authors find that this approach can result in an average speedup of 1.5x.

2.5.2 Machine Learning

Machine learning is another large class of workloads that can potentially benefit from serverless computing. Machine learning workloads are very diverse and can be categorized in many ways, from model training and inference, to data analysis and model tuning. In this subsection we focus on model training and model serving since these two categories encompass many of the most relevant ML workloads today. Next we explain each one of these types of ML workloads in more detail, how they can benefit from serverless computing, and discuss the challenges in doing so.

2.5.2.1 ML Training

ML training is the process of training a model to perform a specific task, for example training a model to check whether there is a STOP sign in a picture of a road. ML training approaches can be very varied, but some of the most widely used models, such as deep learning models, are trained in highly parallel and distributed environments with frameworks such as Tensorflow [1], PyTorch [81] and Caffe [62]. The process of training ML models is data and compute-intensive. In terms of data, models often are better trained with larger datasets and they often require high throughput reads of input datasets. This means that ML training setups need to provision large amounts of storage, close to where the data is needed, to provide both the storage and throughput capacity required for these workloads. In terms of compute, ML training often require complex computations. For instance, deep learning networks can contain many large layers with millions, or even billions, of parameters. Training for such models often requires specialized accelerators, such as GPUs. Another common complementary strategy to improve the end-to-end times of ML training tasks is to leverage the resources of multiple machines.

ML training has not yet received significant attention in the context of serverless computing because today's serverless platforms are not able to meet the data and compute demands of many such workloads. In terms of data, serverless lambdas are limited in storage and memory and thus cannot store large datasets. An alternative would be to store these datasets in external storage systems, and read them into the lambda only when needed. However, we find that in AWS Lambda lambdas can only receive data at most at a rate of 80MB/s, which is not enough for the needs of these workloads. In terms of compute, today's serverless platforms do not provide support for hardware accelerators such as GPUs, and these are critical for many modern ML training workloads.

This does not mean that serverless computing is an inadequate computational model for ML training. However, for specific ML training workloads, such as distributed deep learning training, current serverless platforms do not yet provide the necessary performance to compete satisfactorily with traditional VM-based approaches.

2.5.2.2 ML serving

Another important category of ML workloads is ML serving (or ML inference). ML serving is the process of running the model to make a prediction. For example, asking the model to tell us whether there is a STOP sign on a given picture. Unlike ML training, ML serving does not require large amounts of data and compute. In ML serving, the only elements needed to make a prediction are the model and the input data (e.g., a picture). However, in ML serving, the latency with which we can make a prediction is critical because this prediction is often performed in the context of a real-time decision. For instance, whether to stop the car in front of a STOP sign, or whether to show a specific user a certain online ad. Clipper [32] defines 100ms as the maximum threshold for serving ML predictions in modern online services.

Serverless computing is a promising model for ML serving because it allows the instantiation of computational units as needed to serve individual requests. The serverless computing model already provides many of the components of VM-based serving systems, such as request routing, scheduling and auto-scaling.

However, previous work [61, 5] has found that today's serverless platforms are not well suited for serving ML predictions. The main challenges these works find have to do with invocation overheads and overheads in setting up the prediction environment in the lambda. First, invocation overheads are highly unpredictable and can often take several seconds or even minutes [42]. Second, for every invocation the serving application needs to download and setup the model locally. For instance, reading a medium-sized 100MB deep learning model from S3 to a lambda takes more than 1 second. Other steps such as deserialization make these overheads even higher.

2.6 Serverless Communication and Storage

Another aspect of serverless computing that has received attention is distributed communication for data analytics on serverless platforms. Data analytics workloads often times require distributed shuffles to aggregate data. For instance, when calculating the average per-day sales records of a set of stores, Map Reduce systems such as Apache Spark require that all data of a particular store is sent to the same server in order to compute the store's average. Such operation rely on distributed shuffles for data aggregation.

Distributed shuffling in today's serverless computing platforms is a challenging operation because serverless computing platforms do not allow functions to establish network connections between each other. Furthermore, even if such network communications were allowed, it would be challenging to perform data transfers between ephemeral workers that can die at any point in time.

First-generation serverless frameworks such as PyWren [64], FaastJS [40] and numpywren [93] relied on cloud storage systems such as AWS S3 for inter-lambda communication. PyWren [64] showed that such storage systems are a good fit as secondary storage for intermediate data due to their high throughput scalability and thousands of reads/writes per second. For instance, PyWren showed that AWS S3 can scale linearly up to 2.5K workers and achieve 80GB/s of throughput. However, such systems can have high latency (10s of milliseconds) and can throttle traffic for a high number of storage transactions per second.

In the context of data shuffles for data analytics, [84] concluded that throttling of requests can severely limit the scalability of serverless workloads that require communication. To address this problem, the authors propose a shuffling primitive for serverless computing that leverages a mix of disk-based (*slow*) and memory-based (*fast*) storage for inter-function communication. This way, applications can use *slow* storage for storing bulk data (e.g., input/output data) and use *fast* storage for performance-sensitive communication between lambdas. The authors show that this combination can lead to 59% lower resource usage and 4-500x performance improvements over a solution that only leverages *slow* storage.

However, this approach has several drawbacks. First, it requires developers to provision and manage an external storage system for the sole purpose of communication. This negates the serverless principle of delegating the resource management responsibilities to the cloud provider. Furthermore, it requires developers to accurately predict the amount of resources required for the storage system. If the developer overestimates, it ends up paying more than necessary and if it underestimates it leads to poor end-to-end performance of the application.

To address this problem, Pocket [83] proposes a new abstraction for serverless storage that does not require explicit resource management and can scale to the needs of the application. To this end Pocket provides a key-value store architecture backed by different types of storage resources (DRAM, Flash, disk). To transparently scale to the needs of the application, the authors propose a right-sizing policy. This policy takes into account user-provided performance hints (e.g., latency, aggregate throughput, capacity) and the utilization of the resources to scale-up or scale-down the resources as needed.

Pocket is a good approach to solve the trade-off between performance and need for explicit resource management that serverless developers have to make. However, Pocket is an inadequate solution for workloads that require complex distributed communication patterns or for applications with communication patterns that are hard to predict.

2.7 Improving Serverless Platforms

2.7.1 Invocation Latency

Most lambda functions are short-lived. For instance, Shahrad et al [92] find that on average in Azure Functions, 50% of the lambdas last at most 1 second of execution and a staggering 90% last at most 10 seconds. For such short-lived function invocations, the overheads of allocating resources, scheduling, and starting the runtime can easily surpass the useful time of the lambda's computation. Fast invocation latencies also open serverless computing to a larger number of important latency-sensitive workloads, such as ML serving. To address this problem, systems such as SOCK [78], Particle [98] and SEUSS [23] have proposed different approaches. For instance, SOCK [78] identifies different sources of scalability and performance bottlenecks, such as network and mount namespaces, cgroups primitives, and downloading and installing lambda dependencies. To address these scalability issues, SOCK leverages knowledge about the performance and scalability of Linux operations to optimize the performance of the serverless platforms. Concretely, SOCK proposes (a) using bind mounts to stitch together a root directory, (b) avoiding the use of network namespaces, (c) reusing cgroups. The authors find that these strategies reduce the overall platform overheads 2.8-5.3x when compared to AWS Lambda and Apache OpenWhisk [10]. This seems to suggest that serverless platform are not yet fully optimized for the vast majority of short-lived invocations.

Similarly, Particle [98] finds that configuring the virtual network of lambda containers during the invocation of a lambda can take up to 84% of the total end-to-end time of an invocation. Particle proposes a more efficient way of setting up the virtual network that improves application runtime by 2.4-3x.

In SEUSS [23], the authors make the observation that many function lambdas execute the exact same code before running a lambda function. For instance, all functions have to construct the environment, initialize the runtime, import the code, generate the bytecode and import run arguments. To avoid this source of repeated overhead, the authors propose a system that takes snapshot of a function after each one of these steps. Afterwards, when a new lambda is executed, it can be spawned from one of the previously constructed snapshots to avoid these overheads. The authors show that this approach successfully reduces cold starts by 10x. FAASM [94] also leverages snapshots for faster invocation of lambdas.

2.7.2 Memory overheads

Another source of overheads in serverless platforms has to do with the strong isolation between lambdas. In a regular VM, processes share libraries, runtimes and other application state (e.g., input data). In contrast, in serverless platforms each lambda instantiates its own independent environment which prevents sharing.

For instance, Photons [39] finds that for a serverless image classification task, only between 6-29% of the memory cannot be shared between lambdas. To address this problem, Photons [39] pushes data isolation up the stack, from the container level to the language runtime level. This way, Photons uses the same runtime for different invocations with the same function. This allows explicit sharing of data that should be shared, while keeping the state private to each lambda isolated. The authors find that this approach reduces the function memory consumption by 25-98%.

Alternatively, FAASM [94] allows functions running in the same address space to share data through shared memory regions. This abstraction can be very useful for workloads in which lambdas do computations on the same large data. For instance, in distributed SGD training workers use the same large training dataset to train a model. For such workloads, the authors find that FAASM can reduce the memory overheads by 10x.

This is particularly relevant to understand the design decisions in the proposals we make in Chapters 3 and 4.

2.8 Summary

We started this chapter by explaining the evolution of the cloud and the different types of cloud offerings available today (from IaaS to SaaS). Then we transitioned to the new serverless computing paradigm that is aimed at simplifying resource management tasks that developers are responsible for in the cloud. We discussed the different limitations of today's serverless platforms, and workloads of interest for this dissertation: data analytics and machine learning. Within these workloads, we discussed existing approaches for computing them in serverless and in which ways those approaches don't fully meet the needs of developers. Additionally, we discussed the communication and storage aspects of serverless computing and we surveyed related work on reducing lambda invocation latency and lambda memory overheads.

In the next chapter we transition into concrete proposals and systems for addressing some of the issues with today's serverless computing. Concretely, we propose a system that provides support for serverless end-to-end ML workflows and does not require explicit management of resources.

Chapter 3

Automatic Management of ML Workflows

3.1 Roadmap

The first missing piece of serverless computing is a set of workload-specific APIs that can abstract the complexities of serverless for highly distributed workloads. In this chapter we present the design, implementation and evaluation of a system, Cirrus, that addresses this problem. Here we focus on a large class of applications, machine learning (ML) workflows, but the principles and approach are applicable to other classes as well.

In Section 3.2 we introduce the problem of complexity of interactive ML workflows that arises from resource over-provisioning and explicit resource management. We also explain how serverless computing is a promising approach to address this complexity. At the end, we provide a brief overview of how the limitations of serverless today's serverless platforms make the execution of ML workflows particularly challenging, and motivate how the Cirrus system is designed to be able to overcome these limitations.

In Section 3.3 we dig deeper into the inherent challenges of executing ML workflows: resource over-provisioning and explicit resource management. We also quickly summarize the serverless computing limitations that impact ML workflows, such as unpredictable launch times and lack of fast shared storage.

In Section 3.4 we propose the Cirrus design for serverless end-to-end ML workflows. We discuss the two major building blocks of the Cirrus design, the stateful client side and the stateless server side, and the respective sub-components (frontend, backend, dashboard, and worker runtime). We also explain the Cirrus data store and how it helps with communication and storage of intermediate data.

In Section 3.5 we illustrate the easiness of use of the Cirrus framework by showing an example of a ML workflow executed with Cirrus. Additionally, we showcase the Cirrus dashboard and how it can be used by the Cirrus users to track the progress of the Cirrus tasks.

In Section 3.6 we provide the details of the Cirrus implementation across the different components. Here we emphasize the technical aspects related with the communication between the lambda workers and the data store. These aspects are critical to achieve a high number of iterations per second during the training phases.

In Section 3.7 we demonstrate the application of Cirrus for distributed training of ML models on serverless platforms. We evaluate Cirrus training performance in terms of training loss when compared against other ML systems, such as Apache Spark, Tensorflow and Bosen. We also evaluate our system in terms of its scalability during training and in terms of cost per update.

In Section 3.9 we summarize our results and lessons and conclude.

3.2 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, and neural network structure).

Importantly, modern ML workflows are iterative and increasingly comprised of multiple heterogeneous stages, such as (a) pre-processing, (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 [88, 35]. 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.

We designed and developed 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 the restricted unit of serverless computation, *lambda function*, which is submitted by developers and scheduled for execution by the cloud infrastructure. This obviates 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 [17, 58, 48, 20, 6, 30] and open source platforms [10, 54, 16, 79].

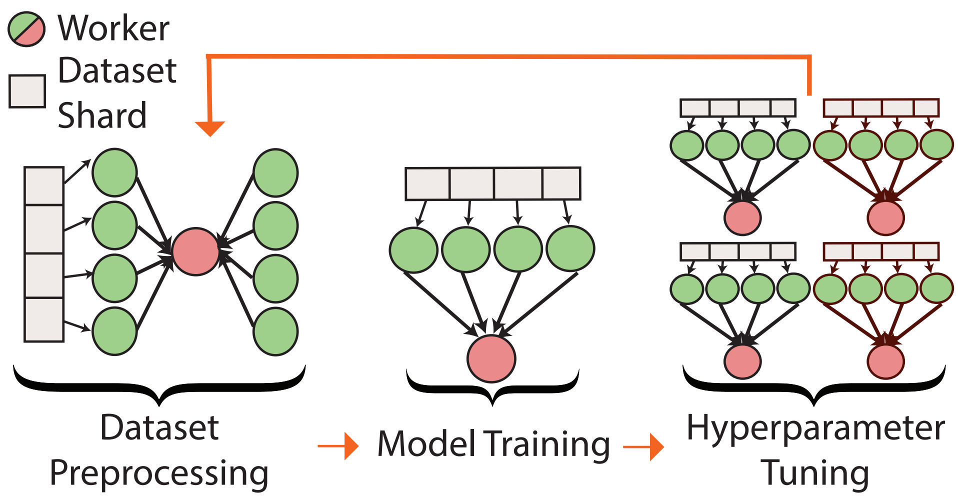


Figure 3.1: Typical end-to-end machine learning workflow. (1) dataset preprocessing typically involves an expensive map/reduce operation on data. It is common to take multiple passes over data, e.g., when normalization is required. (2) model training (parameter server). Workers consume data shards, compute 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.

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, and 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 [104] and Bosen [102, 103] generally load all training data into memory. 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 [64] 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 partic-

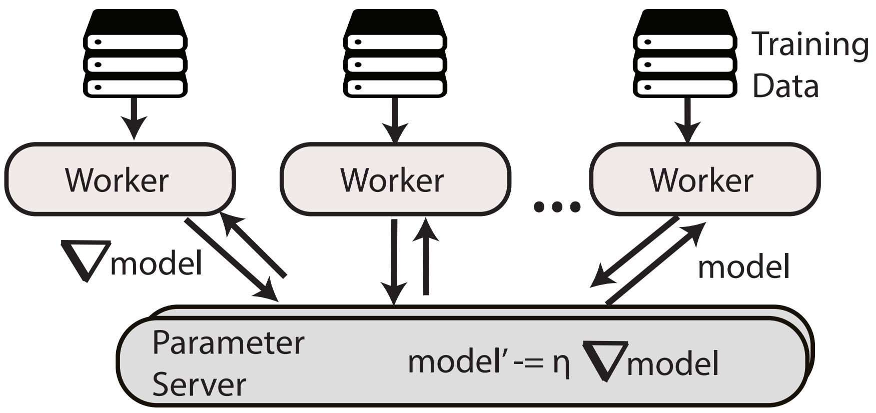


Figure 3.2: Distributed stochastic gradient descent training with parameter server. The parameter server iteratively computes a new model based on the model gradients it receives from workers. Workers then compute new model gradients from a subset of training data (minibatch) and the model distributed by the parameter server. This iterative process continues until the model converges.

ular, Cirrus supports fine-grain, data-intensive serverless ML training and hyperparameter optimization efficiently. Based on the parameter server model (see Figure 3.2), Cirrus provides a simple 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 a simple interface (3.5) 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 ($\sim$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 [102, 1] (3.7.2) and 100x compared to the best-performing configuration of PyWren (3.7.5).

3.3 Democratizing Machine Learning

3.3.1 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 (3.1). 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) [104, 1, 26, 2, 102]. 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 to frequently *over-provision* resources for peak consumption, which leads to significant waste of datacenter resources [88, 35]. 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 3.1. 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.

3.3.2 Serverless Computing

Serverless computing is a promising approach to address these resource-provisioning challenges [63, 53]. 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

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

(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 [1] or Spark [104] on AWS lambdas or 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 [82], 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 [91] and Multiverso [76]) 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.

3.4 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.

3.4.1 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 (3.3). 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 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.

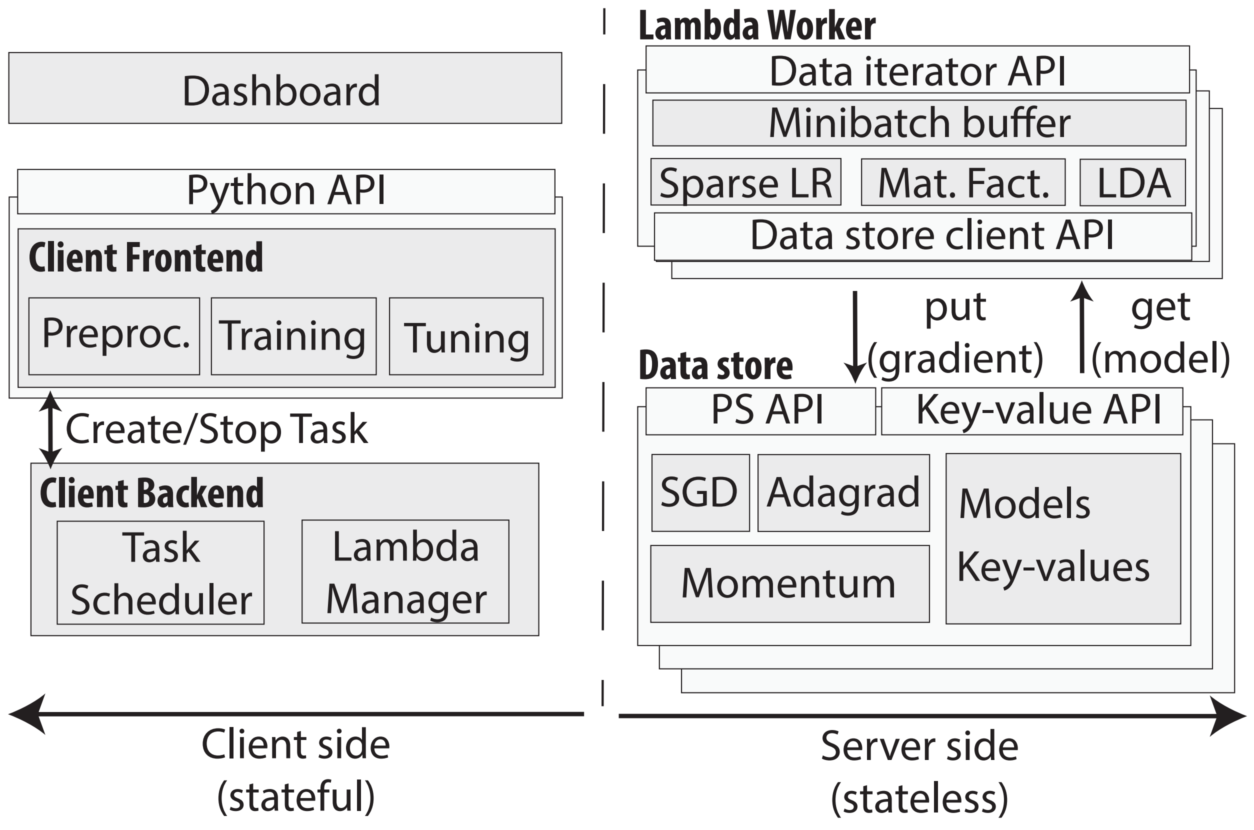


Figure 3.3: Cirrus system architecture. The system consists of the (stateful) client-side (left) and the (stateless) server-side (right). The client-side contains a user-facing frontend API and supports preprocessing, training, and tuning. The client-side backend manages cloud functions and the allocation 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.

3.4.2 Cirrus Building Blocks

Cirrus makes use of three system building blocks to achieve the aforementioned principles (see Figure 3.3). 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.

3.4.2.1 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 (SGD) [22]. 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.

3.4.2.2 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.

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 3.2: 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.

3.4.2.3 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.

3.4.2.4 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 *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 3.2) to support different ML use cases. For instance, it's important that the data store supports multiget (3.7.5), 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. Furthermore, Cirrus also supports computing multiple gradients every iteration locally on each lambda before communicating them with the Cirrus data store.

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.

3.4.3 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.

3.4.3.1 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.

3.4.3.2 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

AWS Lambda Challenges	**Zip 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 3.3: Technical challenges of using lambda functions in Amazon AWS and Cirrus's design choices that address them.

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.

3.4.3.3 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.

3.4.4 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 3.3) 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.

3.5 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.

To this end, 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.

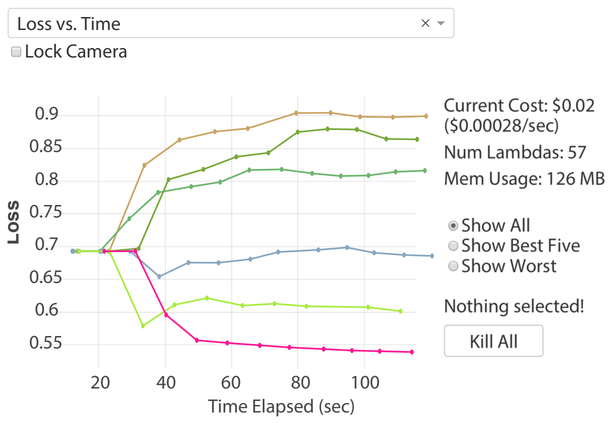
We demonstrate the capabilities of the Cirrus API with an example – the example in Figure 3.5 consists of developing an efficient model for the prediction of the probability of a user clicking an ad for a dataset of display ads. This example is based on the Criteo Kaggle competition [37].

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.

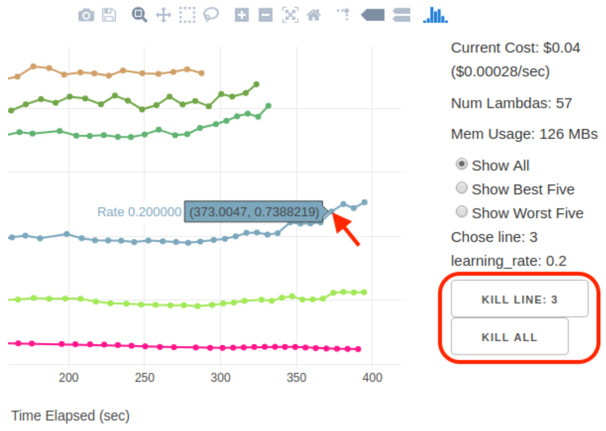
Once the data is loaded into S3 it can be immediately preprocessed. Cirrus provides a submodule with different preprocessing functions. For instance, the user can normalize the dataset by calling the *cirrus.normalize* function with the path of the dataset in S3. Once the data is loaded, the user can 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.

3.6 Implementation

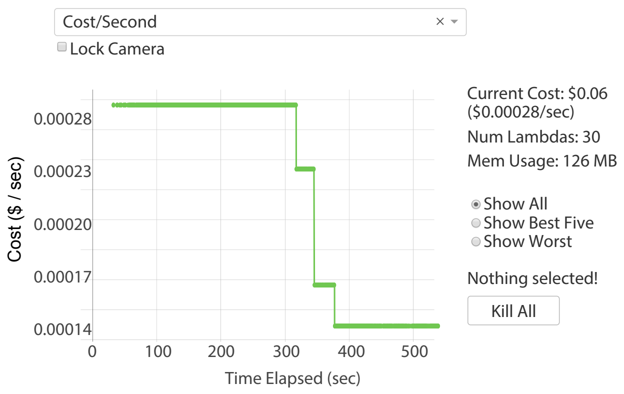
The Cirrus implementation is composed of four components: (1) python frontend, (2) client backend, (3) distributed data store, and (4) worker runtime. The frontend and client backend



(a) Convergence monitoring panel



(b) Experiment control panel



(c) Aggregate cost panel

Figure 3.4: Cirrus's hyperparameter search dashboard for visualizing and controlling tuning experiments. (a) plots loss over time for each hyperparameter search experiment in real time. (b) shows the experiment control panel with a kill widget for diverging experiments. (c) Cirrus's aggregate cost savings over time after terminating diverging experiments.

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

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

Table 3.4: Cirrus components.

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 3.4 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 [59] for users to monitor the progress of the workloads and start/stop tasks.

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

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

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 (3.6) 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 [52] 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.

3.7 Evaluation

This section compares Cirrus with tools specialized for ML training under traditional execution environments (3.7.2 and 3.7.3) and with PyWren, a framework for general serverless infrastructure computation (3.7.5). We complement our evaluation with a discussion on Cirrus's scalability (3.7.4), an ablation study (3.7.5), and a microbenchmark (3.7.6).

3.7.1 Methodology

For our evaluation, we ran serverless systems, Cirrus and PyWren, on AWS Lambda [17]. 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 (3.7.5) 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).

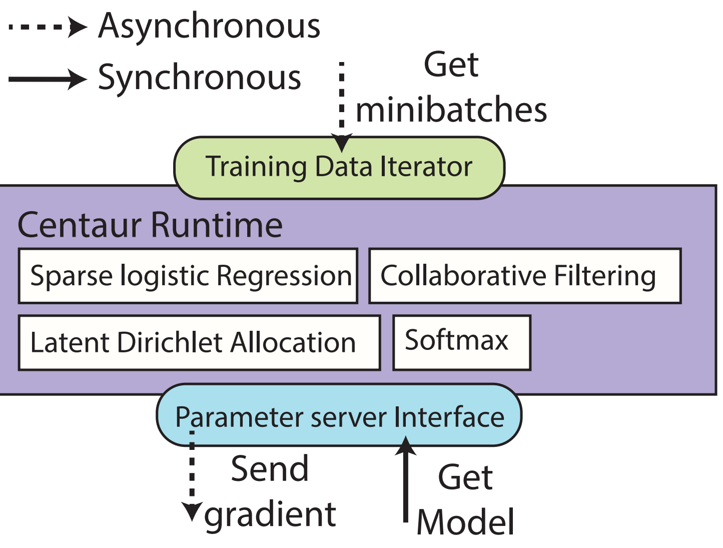


Figure 3.6: Cirrus worker runtime
. Minibatches are asynchronously prefetched and cached locally within each lambda's memory (depending on the size of the lambda used). Similarly, gradients are sent to the parameter server asynchronously. The model is retrieved from the parameter server synchronously every iteration.

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.

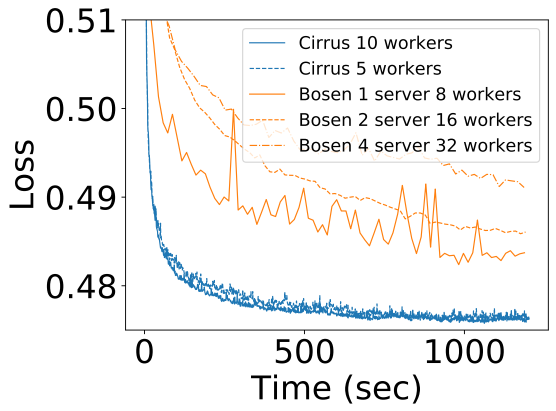
For the Sparse Logistic Regression and Collaborative Filtering problems we used Cirrus's asynchronous SGD [86] implementation. For these experiments we configured all the systems to use a minibatch size of 20 samples.

3.7.2 Sparse Logistic Regression

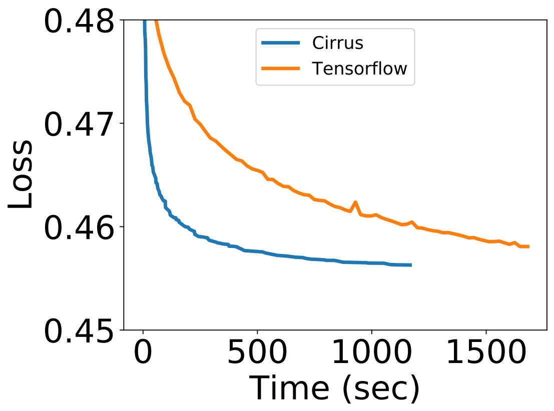
We compared Cirrus's Sparse Logistic Regression implementation against two frameworks specialized for VM-based ML training: TensorFlow [1], and Bosen [102].

TensorFlow is an open-source framework developed at Google and specialized for deep learning workloads. It is today the most widely used framework for deep learning. It provides a simple but general interface for building neural networks, and a highly-optimized backend that can train the networks in a parallel and distributed fashion. Bosen is a distributed and multi-threaded parameter server, developed at CMU and commercialized by Petuum [103], that is optimized for large-scale distributed clusters and machine learning algorithms with stale updates.

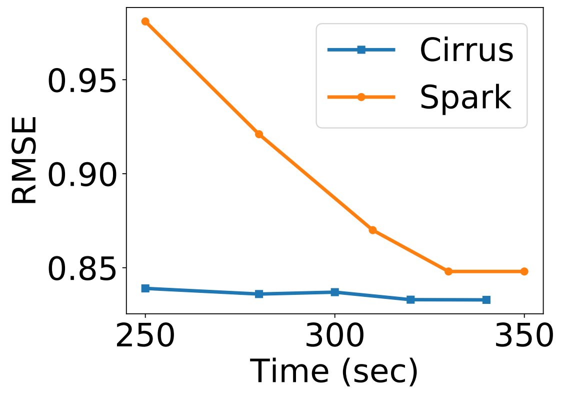
Logistic regression is the problem of computing the probability of any given sample be-



(a) Bosen



(b) Tensorflow



(c) Spark

Figure 3.7: Training performance comparison between Cirrus and Bosen, Tensorflow and Spark for different workloads. (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 [49] 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).

longing 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 [33]. 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 3.7a 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 3.7b, we compare Cirrus with TensorFlow using the same dataset and the same pre-processing steps. Similarly, Cirrus reaches the best loss TensorFlow achieves by $t = 1500$s 3.75x faster (by $t = 400$ seconds).

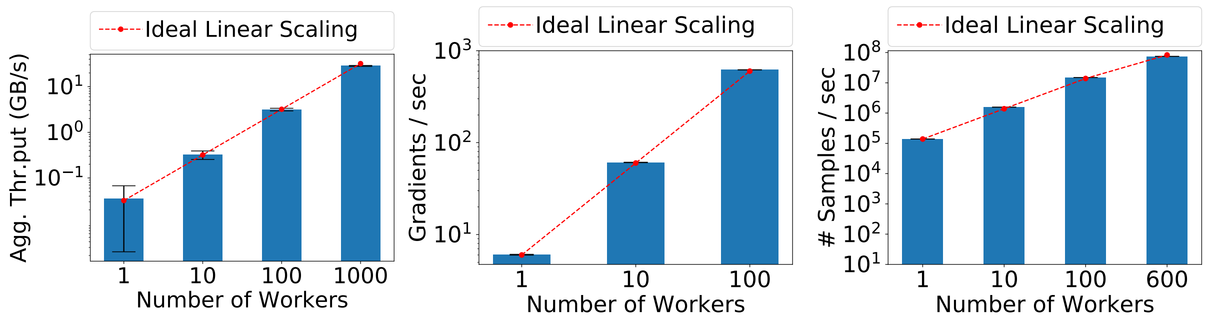
3.7.3 Collaborative Filtering

We also evaluate a second model supported by Cirrus: collaborative filtering (see Figure 3.7). Collaborative filtering is a technique 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 [77] 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 (3.7c). 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 (3.7c). Through profiling, we observe that Spark's ALS implementation suffers from expensive RDD



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

Figure 3.8: 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.

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.

3.7.4 Scalability

Finally, scalability is an important property for ML workflow support. We show that the choice of serverless infrastructure rests on its impressive scalability (3.8) 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 [64].

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 (3.8a), 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 (3.8b) – 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 (3.8c).

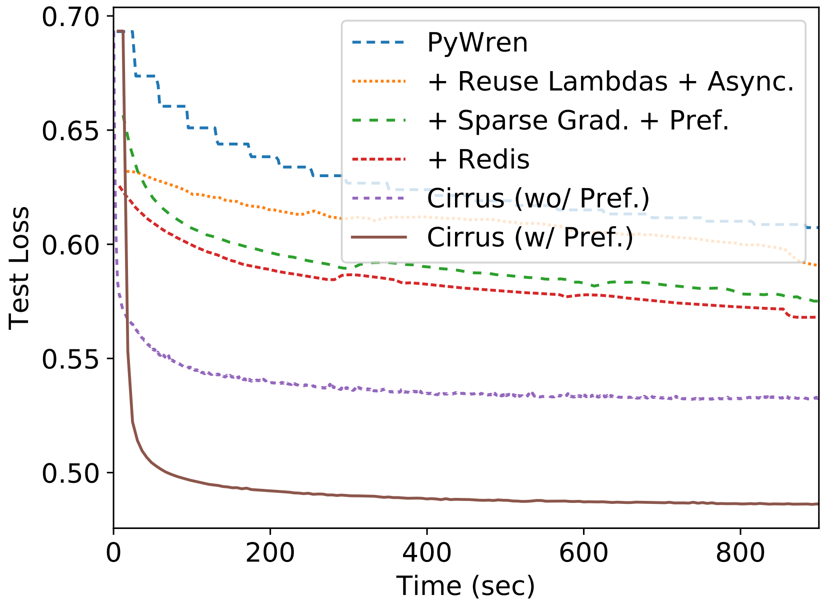
3.7.5 The Benefits of ML Specialization

To evaluate the advantages of a specialized system for ML, we compare Cirrus against PyWren [64]. 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.

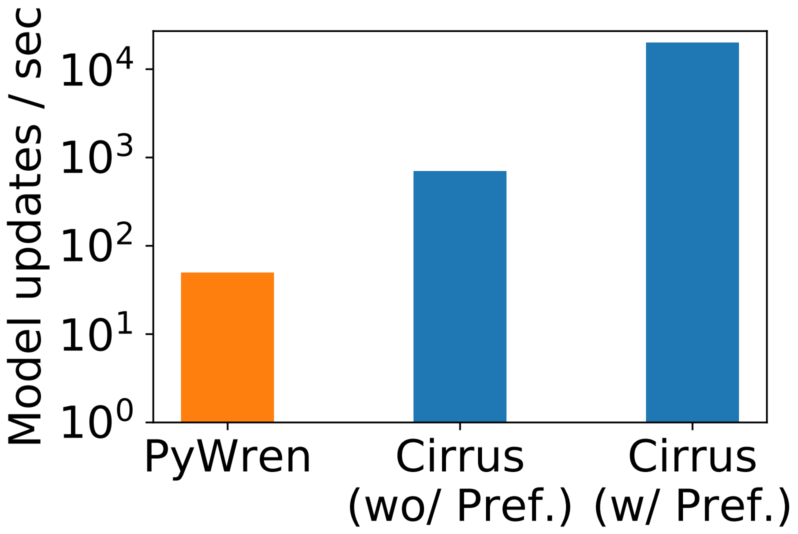
To perform this comparison we initially implemented a synchronous SGD training algorithm for Logistic Regression on PyWren. Our code uses PyWren to run a number of workers on lambdas (*map* tasks) and the gradients returned by these tasks are aggregated and then used to update the model (in the PyWren 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 (3.9a). 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 (3.9b) and converges significantly slower (3.9a) 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).

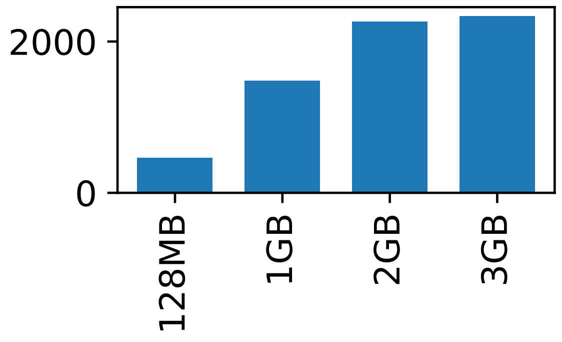


(a) Convergence over time.

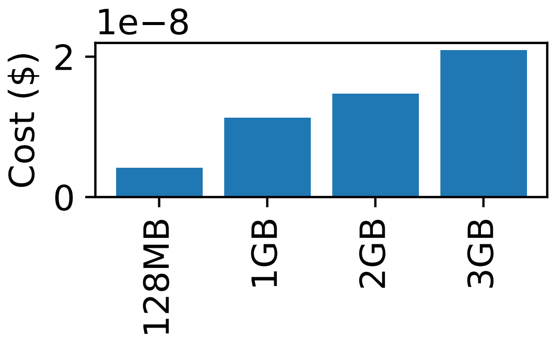


(b) Model updates per second.

Figure 3.9: PyWren and Zip's performance on a Sparse Logistic Regression workload when running on 10 lambdas. Zip 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 Zip'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.



(a) Updates/sec



(b) Cost/update

Figure 3.10: 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.

3.7.6 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 3.7.2) 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 3.10. 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.

3.8 Related Work

In this section we discuss and revisit related work that pertains to the goal of providing interactive ML workflows in serverless. Here we discuss alternative approaches for running distributed serverless computations in serverless, existing serverless storage systems used for λ-to-λ communication, ML training systems designed for VMs and the challenges of porting them to serverless, and how ideas from hardware disaggregated architectures can be used for this problem.

Serverless computing. Previous systems, such as PyWren, ExCamera and gg, allow running large-scale distributed jobs running on serverless functions. ExCamera [42] is a library to leverage lambdas to compute intensive video-encoding tasks in a few minutes. gg [43] 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 [78, 100, 4] 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 [67, 83] 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 multiget), 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 [1] runtime has high memory overhead and Bosen [102] 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 [76] and Vowpal Wabbit [2], 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 [72, 71] 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 [104], and Hadoop [9] 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 [85]. 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 [56], Intel [60], Huawei [57], and Facebook [36]) and academics (e.g., Firebox [12], Microsoft Research [89], VMWare [3] and others [66]). 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 [3] 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 [12] 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.

3.9 Summary

In this chapter we presented a system, Cirrus, that is aimed at addressing the complexities of serverless platforms for end-to-end ML workflows. Cirrus is 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.

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 [102] 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 [64] 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.

In the next chapter we focus our attention in the second missing piece of serverless: the lack of a high-performance and scalable abstraction for distributed communication.

Chapter 4

Scalable Serverless Communication

4.1 Roadmap

The second missing piece of serverless computing is a high-performance and scalable abstraction for distributed communication in serverless. To address this problem, in this Chapter we present Zip. Zip is a serverless framework that enables large-scale high-performance and scalable communication across lambdas. Zip achieves higher performance by extending the design of existing serverless platforms with high-level abstractions and a runtime for lambda communication. Furthermore, Zip implements a per-machine daemon that opportunistically optimizes performance by exploiting lambda locality.

In Section 4.2 we explain the challenges of performing distributed communication in serverless platforms and provide an overview of how Zip overcomes them.

In Section 4.3 we make the case for a new communication model for serverless applications that (a) allows direct network connections between lambdas, (b) provides a high-level API for distributed communication, and (c) optimizes communication within and across machines.

In Section 4.4 we present Zip's programming model and API for lambda communication based on Zip channels. We discuss how lambdas can join channels to perform specific distributed communication patterns and illustrate with an example.

In Section 4.5 we present the design and architecture of Zip. Here we show how Zip extends the design of existing serverless platforms, namely by adding a controller, and lambda library. We also discuss the per-machine daemon that optimizes specific types of communication within each machine.

In Section 4.6 we discuss the implementation details of the system. Here we emphasize the careful C++ backend to achieve high-throughput low-latency message passing. We also discuss optimizations aimed at avoiding unnecessary data copies and serialization of data for widely used Python data types such as *numpy arrays*.

In Section 4.7 we evaluate Zip. Here we demonstrate that it is possible to provide simple extensions to existing serverless platforms to provide significantly better performance for distributed communication. Here we show that, for a distributed sorting workload, Zip can provide a 1.4x speedup and 4.8x lower cost compared to the second fastest alternative. We also perform micro-benchmarks and show that Zip performs up to 12x faster than the second best alternative. Even if for some alternative approaches Zip occasionally performs similarly, unlike those approaches Zip does not require explicit resource management.

In Section 4.8 we discuss how Zip compares to some of the related work. In particular, survey work that leverages external storage systems for communication, and previous attempts at enabling serverless communication.

In Section 4.9 we summarize the main takeaways of this chapter, including the performance and cost improvements provided by Zip.

4.2 Introduction

Serverless is a compelling model for highly distributed workloads because of its fine-grained elasticity. For instance, the problem of training and optimizing a machine learning model consists of a sequence of three sub-tasks: (1) data preprocessing and augmentation, (2) model training, and (3) model hyper-parameter tuning. On a traditional platform, developers provision their resources to handle the most compute-intensive task but typically leave resources underutilized during the other tasks.

Despite the wide-range of promising applications for serverless computing, today's serverless platforms remain a poor fit for many important distributed applications. One important problem is that serverless platforms lack support for *direct lambda-to-lambda communication*. As a result, developers rely on external *storage systems*, such as AWS S3 [18], Redis [87], and Pocket [83], that are inefficient for the purpose of passing messages between lambdas. This ad-hoc communication method presents a number of problems in serverless platforms. First, it is inefficient because data has to be transferred through two hops and it leads to duplication of data (e.g., lambda sending same data to group of lambdas). Second, it requires time-consuming deployment, configuration, and management of a separate system which negates the easiness of deployment promised by the serverless model. Last, it ties the scalability of the application to the scalability of the external storage system.

A major goal of the serverless model is to significantly simplify the development of cloud applications, however, the current model lacks a high-level communication model. This limitation forces developers to rely on their own ad-hoc communication mechanisms using external storage systems. In addition, existing communication abstractions, such as sockets and MPI [41], are challenging to use in the serverless environment due to the frequent arrival and departure of workers. This prevents the adoption by developers of traditional communication frameworks [41] in serverless environments. Furthermore, lambda isolation mechanisms combined with the lambda "schedule anywhere" approach of serverless model wastes communication optimization opportunities. Notoriously, lambda co-location and data sharing are some important performance optimizations that are excluded by existing serverless models.

This chapter proposes Zip, a serverless framework that extends the current serverless model with direct, scalable, and serverless inter-lambda communication. For developers, Zip provides a high-level communication API for communication between lambdas that allows *1:1*, *1:many*, and *many:1* group communication. Specifically, Zip provides a broadcast, reduce and shuffle functions, which are fundamental building blocks for many distributed workload, such as data analytics[104, 9, 84], distributed ML[1, 70], and others[34]. Zip provides a highly-optimized runtime that manages all aspects of communication, such as network connections, group membership, and failure handling.

Zip relies on a scalable distributed channel implementation to allow efficient and fault-tolerant direct lambda communication. The channels data path is implemented using a mesh of direct connections between lambdas in order to achieve high scalability that can support thousands of workers. In turn, a scalable centralized cluster controller allows workers to establish and maintain channels even in the presence of lambda arrival, departures, and failures. Last, Zip implements a per-machine daemon that exploit locality to de-duplicate messages between lambdas within the same VM, which significantly reduces communication cost for broadcast and reduce operations.

Zip extends the design of existing serverless platforms, such as Apache Openwhisk [10], to enable higher performance and scalable communication. We evaluated Zip using several macro and micro-benchmarks. Our analysis considered both the end-to-end performance and cost when running Zip in comparison with alternative communication approaches, in particular, AWS S3, Redis and Pocket. We found that Zip can provide up to 4.8x lower cost and 1.4x speedup for a distributed sorting workload, compared to the best memory-based store alternative.

This chapter makes the following contributions:

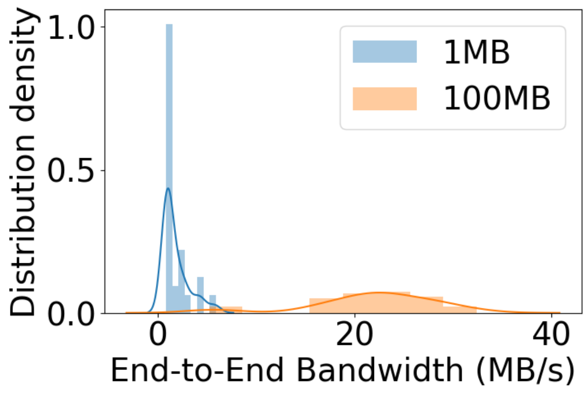
- A high-level serverless communication model that directly supports broadcast, reduce, and shuffle operations.
- The design and implementation of Zip, a serverless framework that lets developers build scalable and fully-serverless applications that communicate efficiently.
- An approach that exploits serverless communication locality and a per-machine communication daemon to increase communication scalability and performance.
- A performance and cost evaluation, for distributed serverless workloads, of Zip and existing approaches.

4.3 The need for a serverless communication abstraction

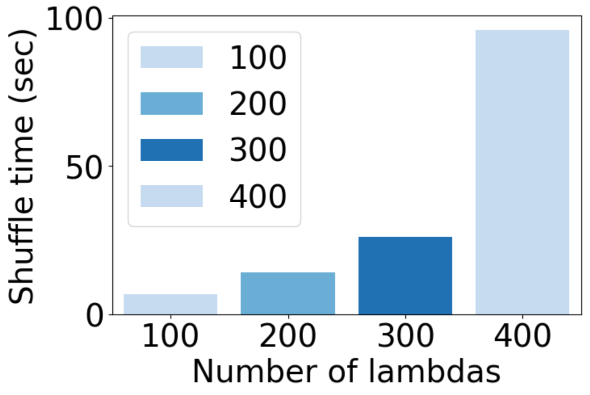
Recently there has been a proliferation of distributed frameworks for serverless computing [93, 64, 24, 40, 31, 43, 84, 75, 90]. These frameworks provide computational abstractions

System	Server-less	Comm. scalab.	λ-to-λ	High-level comm. API	Speed
AWS S3 [18]	Yes	Low	No	No	Slow
Redis [87]	No	Med.	No	No	Med.
Pocket [83]	Yes	Med.	No	No	Med.
Zip [this paper]	Yes	High	Yes	Yes	High

Table 4.1: Comparison between different serverless communication approaches.



(a) Distribution of end-to-end communication bandwidth between two lambdas through AWS S3 with 1MB and 100MB messages



(b) End-to-end time of shuffle between 100, 200, 300, and 400 lambdas using AWS S3

Figure 4.1: Communication performance between lambdas when using AWS S3 for point-to-point communication on shuffle workloads.

for distributed serverless applications in domains such as map-reduce, DAG-based computations, and big data analytics. However, whenever communication between lambdas is required, these frameworks resort to external storage systems such as AWS S3 [18], Redis [87] or Pocket [83].

Although existing frameworks can leverage large amounts of computational resources for a wide range of applications, a number of limitations arise from their reliance on external storage systems and APIs that were not designed for serverless workloads (see Table 4.1). Specifically, communication through such storage systems suffers from a combination of the following drawbacks: (1) requirement to explicitly manage and provision resources for the storage system, (2) (indirect) communication requires an extra hop (Figure 4.1a), (3) lack of scalability (Figure 4.1b), (4) and lack of high-level communication API (see Section 2.4).

Channel	Primitive	Description
Broadcast	`BroadcastChannel(name)`	Creates handle and joins channel.
	`send(data)`	Broadcasts data to all lambdas in the channel.
	`data = receive()`	Receives broadcast data.
Shuffle	`ShuffleChannel(name, role)`	Creates handle and joins channel. Developer needs to specify role (whether lambda just sends, just receives, or does both).
	`shuffle_send(data, hash_function)`	Send data to lambdas. `data` is a list of objects. Objects are distributed by lambdas according to the hash_function provided.
	`data = shuffle_receive()`	Receive data from shuffle senders.
	`data = shuffle_send_receive(data, hash_function)`	Send and receive data.
Reduce	`ReduceChannel(name, reduce_f=0, handler=0)`	Creates handle and joins channel. *reduce_f* function is used to reduce data. One lambda per channel can specify a handler to receive the result of the reduce.
	`reduce(data)`	Sends data to be reduced.

Table 4.2: Zip's API. Zip provides communication primitives for 3 types of communication patterns: broadcast, shuffle, and reduce. After a lambda joins a channel it can communicate with other lambdas in the channel.

4.3.1 How Zip Addresses Serverless Limitations

Zip addresses the communication limitations in serverless by leveraging a combination of techniques. First, Zip allows lambdas to establish direct network connections between each other. This obviates the need for a separate storage system and allows 1-hop communication between any two lambdas. Second, Zip provides a high-level API that provides primitives for widely used communication patterns and abstracts developers away from the complexities of doing distributed serverless communication, such as handling worker failures. Last, Zip improves end-to-end performance and scalability by optimizing communication between lambdas within each machine.

4.4 Zip's Programming Model

Zip provides a simple programming model for high-performance and scalable communication between lambdas. This section discusses the Zip channel abstraction and Zip's communication primitives and their semantics.

4.4.1 Zip Channels

Lambda execution time is generally limited in serverless computing (e.g., max. 15 minutes in AWS Lambda [17]). As a result, in contrast with serverful environments, the arrival and departure of lambdas is a common occurrence in serverless platforms. This makes distributed communication particularly challenging. For instance, the traditional MPI approach assumes that the number of workers does not change during execution [41]. Network sockets on the other hand provide high flexibility, but are low-level and require developers to address a wide-range of low-level details, which negate the simplicity goal of the serverless model.

To address the problem of worker churn, Zip relies on *Zip channels* (see Table 4.2). A Zip channel is a named group of lambdas that intend to communicate with each other. The membership set of a channel is dynamic and there are no restrictions on joining and leaving a channel, on the number of channels, or number of channel members. This flexibility allows developers to create channels for 1-to-many, many-to-many and many-to-1 communication. Once a lambda joins a channel, the lambda can participates in all the communications within the channel.

4.4.2 Communication primitives

Zip provides three major classes of inter-lambda communication primitives: broadcast, shuffle, and reduce (see Figure 4.2). These classes are included in Zip to simplify the development of a wide range of data-intensive distributed systems, such as MapReduce and ML workloads, that often rely on these patterns, and simultaneously exploit their semantic to allow system-level optimizations.

Broadcast A single lambda (broadcast sender) sends a copy of data to all the remaining lambdas (broadcast receivers) in the channel. For every broadcast communication there can only be one sender, but the sender can change between consecutive invocations of a broadcast channel.

Shuffle Zip shuffle channels let lambdas distribute data across all members according to a hash function provided by the developer.

Reduce The reduce operation performs a data reduce across all lambdas in the channel. The computation used to reduce data is specified by the developer. In reduce, one of the workers (reduce receiver) can register to receive the result of the reduce.

4.5 Zip design

Zip's architecture consists of three major components: the library, the controller, and the machine daemon (see Figure 4.3). The Zip's developer library exposes and implements the Zip API. The library is packaged with the application code, and runs with the application inside a container. These containers are deployed onto VMs. The Zip cluster-wide controller is responsible for managing the membership of lambdas in channels, managing the logical and physical connections between lambdas, and handling failure recovery. The controller is deployed as a single process in a separate machine and can serve many clients simultaneously. The Zip daemon is a per-machine process that runs on every machine in the cluster and is responsible for optimizing the communication between lambdas within and across machines.

```python
from zip import BroadcastChannel

bc = BroadcastChannel(ch_name)

def sender():
  bc.send(data=my_data)

def receiver():
  data = bc.receive()
```

(a) Broadcast

```python
from zip import ReduceChannel

def root(list_ints):
  def receive_result(result, num_lambdas):
    print("reduce result is: ", result)

  rc = ReduceChannel(ch_name, handler=receive_result)
  rc.reduce()

def worker(data):
  rc = ReduceChannel(ch_name, sum)
  rc.reduce(data)
```

(b) Reduce

```python
from zip import ShuffleChannel
def hash_ints(data, num_lambdas):
  return data % num_lambdas

def sender(data):
  sc = zip.ShuffleChannel(ch_name, role=SENDER)
  result = sc.shuffle_send(list_ints, hash_function=hash_ints)

def receiver(data):
  sc = zip.ShuffleChannel(ch_name, role=RECEIVER)
  result = sc.shuffle_receive()
```

(c) Shuffle

Figure 4.2: Code samples illustrating the broadcast, reduce, and shuffle Zip APIs. Lambdas join channels using the channel name. Zip automatically establishes connections between lambdas in the channel, optimizes communication within and across machines, and handles lambda arrival and departures.

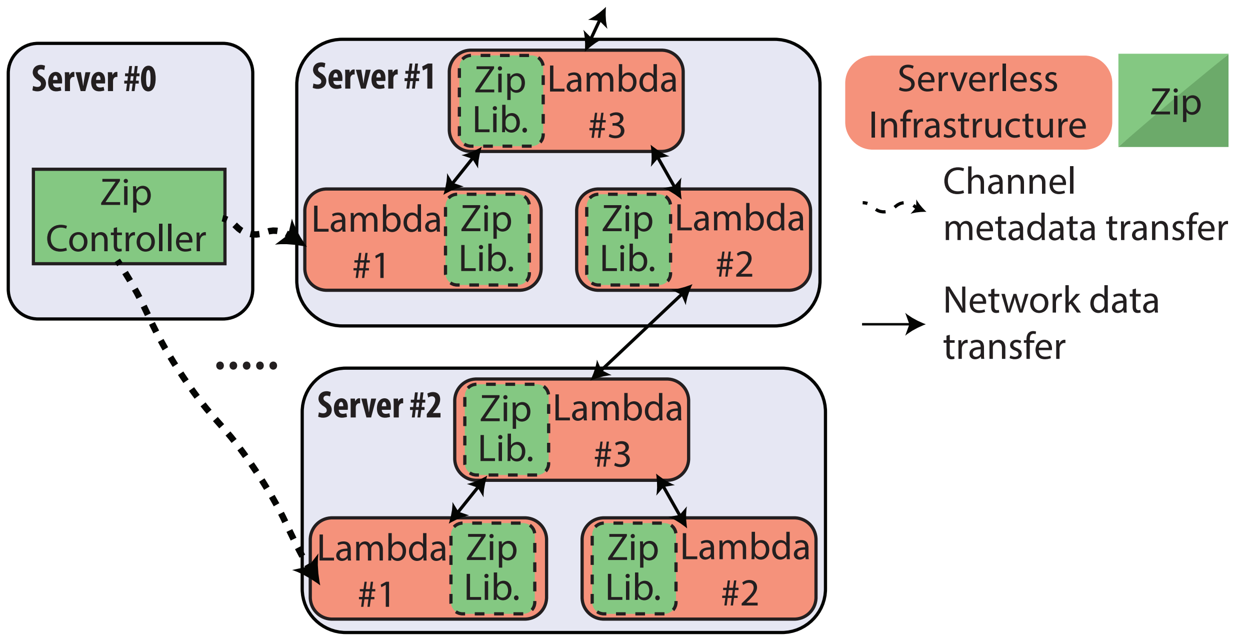


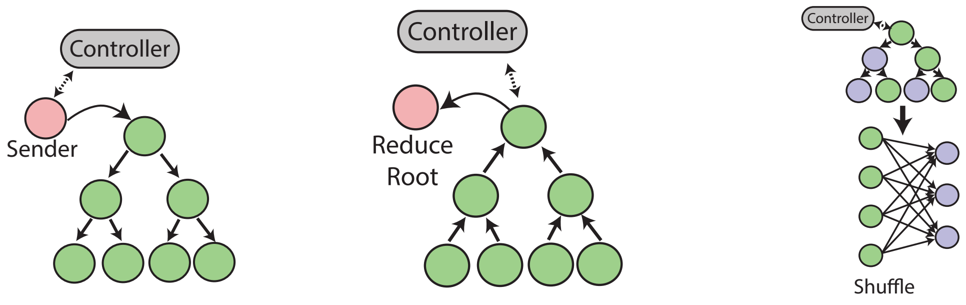
Figure 4.3: Zip architecture (without daemon). Lambdas run inside containers and connect to each other to form a tree-shaped connection mesh.

4.5.1 Zip Overview

Controller The Zip controller manages all the information about channels and lambdas, and manages the connections between lambdas. At the channel level, the controller manages which lambdas belong to which channel, and manages the connections between lambdas within a channel. To join a channel, lambdas send a request to the controller asking to join the channel. Then, the controller determines to which lambdas the new lambda should connect to. These connections between lambdas within a channel form a connection mesh that is then used for distributed communication. The controller then helps the lambda establish a connection with the channel's connection mesh. When there are unexpected lambda failures, or when a lambda graciously terminates, the controller updates the channel's connection mesh to maintain the correctness of communication within the channel.

Library The Zip library provides lambdas with an API for inter-lambda communication. This library is packaged into every lambda and can be imported to the application by developers. Under the hood, the Zip library communicates with the controller to join channels, to get information about a channel, and (c) to inform the controller of a lambda failure.

Daemon Zip uses a daemon that is responsible for leveraging more efficient ways for communication between lambdas within each machine (see Table 4.3). Specifically, the Zip



(a) Broadcast (b) Reduce (c) Shuffle

Figure 4.4: Broadcast, reduce and shuffle communication patterns in Zip. Zip organizes lambdas within a channel in a balanced tree of connected lambdas. Lambdas can participate or leave communications within a channel by contacting the controller to be attached/detached from the connections tree.

Task	Bandwidth (MB/s)
Ser. + deserializing numpy array	650
Network transfer (local)	1500
Network transfer (remote)	625
Shared mem. transfer	4300

Table 4.3: Network bandwidth observed with the different data transfers mechanisms used by Zip to communicate data between lambdas and with the Zip controller. We also show the cost of serializing deserializing a numpy array before a network transfer.

daemon provides two optimizations. For communication within a single server, it provides very fast message passing through shared memory. For communication across servers, it deduplicates repeated data that gets transferred through the network for the reduce and broadcast primitives. The Zip daemon only affects the performance of communication, not the correctness, and is an optional component of Zip.

4.5.2 Design principles

Sporadic lambda-controller communication Lambdas communicate with the controller in 2 cases: (a) when they join/leave channels, (b) to retrieve channel metadata required to execute a communication primitive. Case (a) is challenging if there are many lambdas in the channel, or many concurrent clients creating channels. Similarly, case (b) is challenging for workloads that execute a high number of collective communications per second (e.g., iterative distributed SGD training). Zip provides scalability for situation (a) by making the

Goals	**Zip's mechanisms**
Sporadic lambda-controller communication	Updates to channel membership require communication with small number of lambdas. Peers/channel metadata cached in each lambda.
High-throughput communication	Non-blocking communication between lambdas.
Fast inter-lambda communication	Co-located lambdas communicate through shared memory. Broadcast data is deduplicated within machine by the daemon.

Table 4.4: Zip is designed for scalable and high-performance communication.

event of a lambda joining a channel impact a small fraction of peer lambdas in the channel. This saves the controller from having to contact many lambdas every time a lambda joins or leaves a channel. Zip is design to scale for case (b) by designing communication primitives such that they rarely (or not at all) require to communicate with the controller.

High-throughput communication Zip maximizes work throughput by allowing lambdas to start communicating as soon as possible. To achieve this, Zip allows lambdas to receive data from other lambdas even before they have called into Zip to receive data. This way no lambda is blocking waiting for another lambda. Furthermore, Zip's design allows concurrent transfer of data from and to lambdas to maximize network utilization.

Fast inter-lambda communication Data communication between lambda lambdas should be at least as fast as communication between two lambdas in two distinct VMs. To achieve this goal, Zip is designed in the following way. First, with Zip communication between lambdas is done directly between VMs. Second, Zip leverages the information about the location of data and lambdas. Zip only sends data destined to two or more lambdas in the same VM once. As soon as the data arrives at the destination VM it is copied to the lambda. This can reduce substantially the amount of inter-VM communication.

4.5.3 Channels and primitives

4.5.3.1 Zip Channel

Zip groups lambdas that communicate with each other into channels. Lambdas within a channel establish connections with each other to form a tree-shaped mesh of connections (see Figure 4.4), which lets Zip maintain efficient and scalable distributed communications (see Table 4.5) while tolerating lambda churn.

Joining a channel Lambdas join a channel by getting a handle to it from the Zip API (see Figure 4.2). When lambdas get a handle to the channel the Zip backend contacts the controller to register the lambda in the channel. This way, the controller can keep track of all the channels created in the system and which lambdas participate in which channels. The controller then replies to the lambda with a list of lambdas to connect to, i.e., with its parent and children ports and IP addresses. Similarly, the controller also informs the parent

Operation	# Msg	# Controller Messages	Latency Complexity	Description
Join / Leave Channel	2	2	$O(1)$	1 msg. to controller, 1 msg. to parent
Broadcast	n	0	$O(\log_F(n))$	1 message to root, and then tree-broadcast
Reduce (Tree)	$n-1$	0	$O(\log_F(n))$	Tree-reduce
Shuffle	$n+SR$	0	$O(\log_F(n))+O(R)$	Broadcast disseminates shuffle information. Every worker sends a message to every worker

Table 4.5: Complexity of Zip's primitives for a channel with n lambdas, S shuffle senders, R shuffler receivers and F mesh tree fan-out. New workers join the tree-mesh as leaves. Broadcast requires asking controller for the address of the root of the tree-mesh.

of the new lambda of its new child.

Leaving a channel Lambdas can also leave channels when they no longer want to participate in the channel's communication. When a lambda leaves a channel it notifies the controller. In case of failure, if the controller detects that the lambda is not active (see Section 4.5.5), the controller proceeds in the same way.

4.5.3.2 Broadcast

A *broadcast* is used to send data from a single server to one or more receivers. A broadcast in Zip works in the following way. First, before participating in the broadcast, the lambda needs to get a handle to a *BroadcastChannel*. This handle can then be used to perform multiple executions of *broadcast* in that channel. When the handle to the channel is created, the lambda Zip's runtime automatically communicates with the controller to register the lambda in the channel. During this registration step, the controller informs the new lambda's runtime of its peers in the channel's tree-mesh. This way, the lambda can participate in the peer-to-peer communications in that channel. Second, once a lambda has generated a handle to the channel it can participate in that channel's broadcast as a receiver or as a sender (see Figure 4.4). In every broadcast round, there can only be one sender.

Sender lambdas can use the channel to broadcast data using the *send* method. The *send* method works as follows. First, the Zip runtime retrieves from the controller the address of the root of the channel mesh. To avoid incurring the cost of this communication for every broadcast, the runtime caches this address. Next, the sender sends the broadcast data to the channel's root. Then, the root lambda initiates the broadcast by transmitting the data from the top to the bottom of the tree mesh.

Receiver If the lambda is a receiver, it calls the *receive* method. This method blocks waiting for the lambda to receive the broadcast data. Once the lambda has received this data from one of its peers, the Zip runtime forwards the data to the lambda's peers and returns the data to the *receive* caller.

4.5.3.3 Reduce

Another primitive supported by Zip is *reduce*. In a distributed reduce, data from all lambdas is combined according to a user-provided reduce function.

4.5.3.4 Shuffle

A shuffle allows a group of lambdas (senders) to distribute data across a group of lambdas (receivers) according to a user provided function.

Zip's shuffle primitive works in three stages. First, lambdas create a handle to a *ShuffleChannel* and specify their role in the shuffle: receiver, sender or receiver+sender. This role indicates whether the lambda just sends data during the shuffle, just receives data, or does both. Second, when a shuffle starts, the list of all lambdas and their roles is broadcasted to all participating lambdas. To perform this broadcast, the root of the channel mesh contacts the controller to retrieve the list of lambdas currently in the channel and then broadcasts this information to all the lambdas in the channel. Third, once a lambda has received this list, it can start executing the shuffle.

4.5.4 Resource sharing and isolation

Serverless platforms are shared by many concurrent clients. For this reason, serverless platforms need to provide strong isolation between lambdas and share resources (CPU, network, memory, and disk) among all running lambdas.

Zip ensures lambdas are executed within containerized environments. Within each container, a lambda is assigned 1 virtual CPU and a limited amount of memory and disk (configured by the developer when the lambda function is created). Another resource that requires isolation is the per-server memory that Zip allocates for lambda-to-daemon communication. Our implementation currently does not isolate this shared memory between lambdas from different users. However, tampering can be prevented by creating a separate shared memory region per channel within each server. This memory region can be made only accessible to the containers of lambdas belonging to the same channel. This guarantees that only lambdas from the same user can read and write to the shared memory.

4.5.5 Fault tolerance

Zip can detect and recover from individual or multiple lambda failures to guarantee that subsequent communications within a channel can terminate successfully. Upon the detection of a lambda failure, Zip updates the connections between lambdas within a channel.

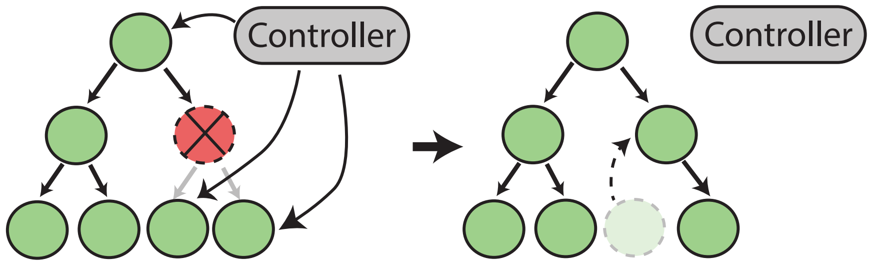


Figure 4.5: Zip's recovery of a node failure. When a node failure is detected, the controller updates the connections of the parent and children of the terminated worker.

Furthermore, periodically Zip checks it the mesh of connections is balanced. If not, Zip re-balances the mesh.

Updating channel connections The unexpected termination of a lambda is detected in Zip by one of its peers, by the controller or by the daemon. Upon the detection of the failure, a message is sent to the Zip controller to inform it of the failure. Once this message is received, the Zip controller starts the recovery process to ensure that all the lambdas in the channel mesh remain connected and that subsequent communications can still occur. To accomplish this, the controller removes the failed lambda from the mesh tree and sends a message to its neighbors informing them of their new parents and children (see Figure 4.5). While this recovery is happening, other lambda failures might occur that lead to the lambda having to reconstruct the tree. The Zip controller continuously updates the tree until there are no more failures. In order to make failure recovery scalable, the recovery algorithm of Zip only updates the connections of the lambdas directly connected to the terminated lambda, a small fraction of the total number of lambdas. This means that for each failure Zip sends at most $O(|\text{children}|)$ messages.

Re-balancing connections After a number of failures, the per-channel tree of connections might become unbalanced which can result in sub-optimal distributed communication. To mitigate this problem, the controller keeps track of how balanced the tree of each channel is during execution. To accomplish this, the Zip controller maintains the depth of every node. When the controller finds that the tree depth between two paths differs by more than a developer-configured value, it can start the re-balancing process. To re-balance the channel mesh, the controller communicates with all the lambdas to suspend the execution in the channel while the re-balancing occurs.

4.6 Implementation

The implementation of Zip consists of four components: the *frontend*, the *backend*, the *daemon*, and the *controller* (see Table 4.6). Both the frontend and backend run inside the

Component	**Language**	**LOC**
Frontend	Python	523
Backend	Python & C++	2430
Daemon	C++	1004
Controller	C++	403
Total		4360

Table 4.6: Zip components.

lambda.

Frontend The frontend is implemented as a thin Python layer that provides the Zip API for application developers. It is responsible for passing the input data and function (e.g., shuffle hash or reduce functions) parameters to the Zip C++ backend.

Backend The backend implements the bulk of the Zip primitives. It maintains the connections with other components (controller, daemon) and peer lambdas, and executes the communication calls received by the frontend library. The backend makes use of a background thread so that it can maintain the connections from the lambda to its peers and to achieve high performance by transferring data in the background. For instance, when one of the lambda's child terminates unexpectedly, the backend is notified to update its connections without the application involvement. Similarly, when a broadcast is initiated by a separate lambda on the same channel, a lambda can receive that data, store it locally and forward it even before the application calls the *receive* API.

The backend is implemented in C++ to avoid the performance and memory overheads of the Python runtime. Our experience revealed that this is crucial to achieve low end-to-end latency, specially for small messages, and to keep the lambda memory usage low. When needed, the backend can execute an upcall into the frontend to execute functions provided by the application. For instance, during a reduce the backend calls the application-specific reduce function asynchronously.

As an optimization, when transferring numpy arrays the backend transfers the raw bytes of the arrays along with metadata about the numpy array (shape and type of the array entries). This obviates the need for expensive serialization and deserialization steps, which can be up to 5x more expensive than just copying the array to the daemon's shared memory.

Daemon The daemon maintains socket connections to all lambdas within the same machine, its daemon parent and its daemon children in remote machines. The daemon implementation leverages shared memory between lambdas within the same machine to provide high data transfer speeds for large messages. Whenever a lambda wants to send data to the daemon, it asks the daemon for a range of available memory within the daemon's shared memory. The daemon then allocates the memory and sends its address back to the lambda. Once the lambda receives the position in the shared memory it copies its data to the shared memory. Finally, the lambda can send a message to the daemon with a pointer to this data. Upon receiving the message, the daemon can copy this data into its own local memory or use that data directly. As an optimization, Zip uses the data directly for numpy arrays to avoid data copies. Once the daemon no longer needs this data, it deallocates the corresponding range of memory.

Controller We implemented the controller as a multi-threaded server that listens for connections from lambdas and serves requests from those connections. The controller maintains a *ChannelMetadata* data structure for every channel in the system. This data structure contains the channel metadata, such as the topology of connections between all the lambdas, the type of channel, and its name. The controller updates this metadata when lambdas join or leave the channel. For such events, the controller determines which other lambdas need to update their connections and informs them.

4.7 Evaluation

This section evaluates Zip using macro and micro-benchmarks and compares with other existing storage-based approaches. First, it evaluates the performance and cost for a distributed sorting application. Second, it analyzes the performance and cost of individual Zip operations: broadcast, reduce, and shuffle.

Evaluation setup All experiments ran on AWS EC2 in the *us-west-2* AWS availability zone. We used AWS ElastiCache [13] managed Redis service for the Redis experiments. Zip's lambdas were deployed within docker containers (Docker version 19.03.8) configured with limits of 1 CPU, and 3GB of DRAM. These resource limits are consistent with those of traditional serverless platforms such as AWS Lambda and Google Functions. All the experiments in the evaluation run on m5.2xlarge, including Redis and Pocket, AWS EC2 instances unless otherwise noted.

For the experiments that use the shuffle communication pattern we ran Zip without the daemon because Zip does not use the daemon for shuffle, since it does not improve or degrade the shuffle performance. For the Zip experiments with the daemon, we launched the daemon process on every EC2 server. We set up the daemon to create a 1GB shared memory region that was shared among all the lambdas to accelerate communication.

To make our experimental results reproducible and avoid the high time variance of launching docker containers, we started measuring the experiments time only after containers were all running. For all Zip experiments, a Zip controller was deployed on a separate *t2.xlarge* instance. In our evaluation we do not consider the cost of this instance because it is only used for the control path, hence it uses very little resources, and it can scale to serve many clients simultaneously (see Section 4.7.2.3). For similar reasons, we also do not consider the costs of the controller and metadata nodes in the Pocket experiments.

For experiments involving external storage systems, i.e., AWS S3, Redis, and Pocket,

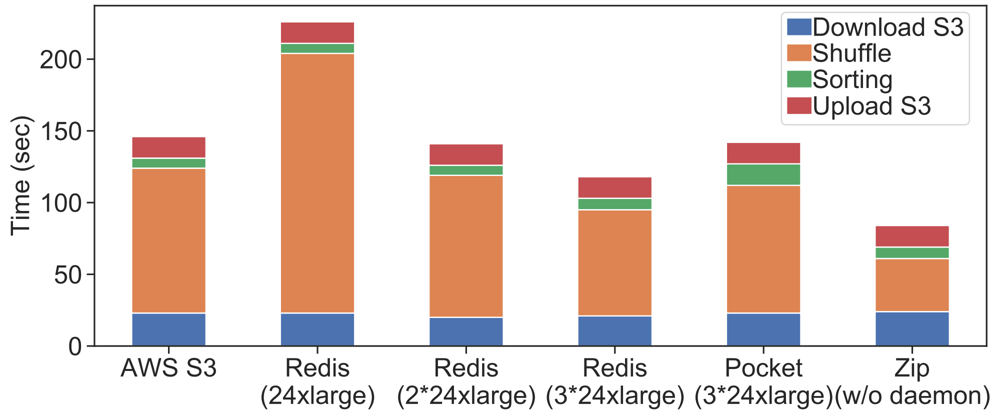


Figure 4.6: Sorting of 100GB dataset with 120 lambdas using different communication methods.

we implemented a separate Python communication library. This library provides the same API as Zip but internally uses the APIs of the respective storage service to perform lambda communication. For the Redis and Pocket experiments, we varied the number and type of VM instances to assess the impact of the distributed store performance in the end-to-end performance of the workload.

4.7.1 Distributed Sorting

Distributed sorting is the process of sorting a dataset by leveraging multiple machines. Distributed sorting requires a mix of compute and communication heavy phases and can scale to many workers. For these reasons it is a standard benchmark for distributed systems [34, 34, 83, 84]. This workload has roughly 3 stages: (a) downloading the dataset to the lambdas, (b) parsing and shuffling the dataset across the lambdas, and (c) sorting the data within each lambda. Step (b) takes the largest fraction of the end-to-end time because it requires the coordination of all the workers to perform an all-to-all shuffle of the dataset between lambdas.

Experiment setup To evaluate this workload we used a dataset of randomly generated 100-byte strings, totalling 100GB (the same setup used in the evaluation of Pocket [83]). We split the dataset in blocks of 100MB and stored them in AWS S3. We also used S3 to store all the final results of the experiment, but vary the system used for the intermediate data shuffle (i.e., Zip, S3, Redis, Pocket). To accelerate the hashing computations during the shuffle for all the experiments, we implemented the hashing function in C++. This increased the performance of hashing by approximately 3x.

Results Our experiment confirms that most of the lambda execution time on the sort workload is spent on shuffling data between lambdas (see Figure 4.6). For instance, for the Pocket

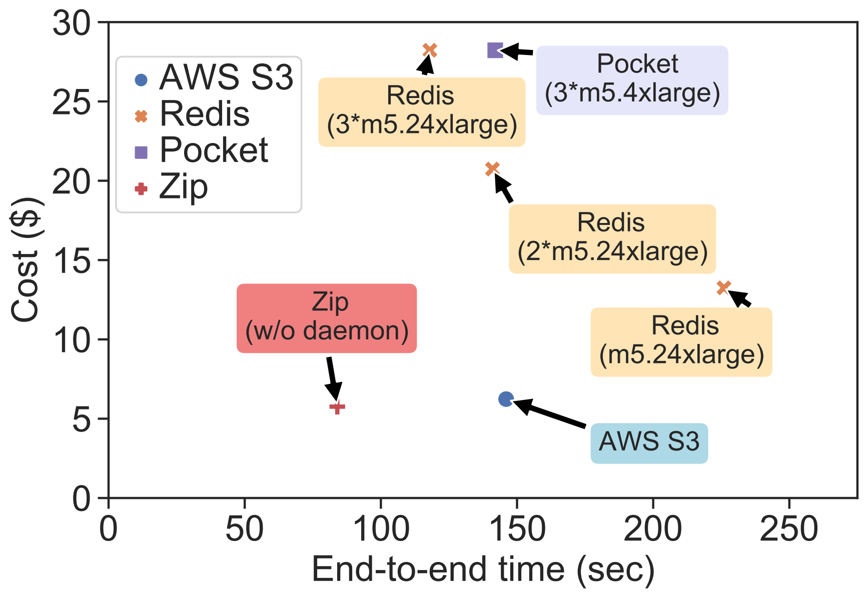


Figure 4.7: Sorting end-to-end performance and cost.

experiment shuffling takes 63% of the total time, substantially higher than downloading the dataset to the lambdas (16%), sorting (11%) and uploading the dataset (10%).

We found that Zip reduces the end-to-end sorting time by 28% (best Redis configuration), 40% (AWS S3), and 42% (Pocket). Zip outperforms Redis, AWS S3, and Pocket because data transfers are made directly between lambdas and do not have to pass through a centralized system. In fact we observe that with Zip on average each lambda transfers data at 299 MB/s, which is higher than with AWS S3 (87 MB/s), Redis (152 MB/s) and Pocket (95MB/s). Even though we configured Pocket to store all the data in memory, its performance is lower than Redis. Upon code inspection, we found that Pocket sends an RPC to the metadata node for every 65K data block that is stored which results in a lower end-to-end throughput. Our results for Pocket and Redis are 2x the results in the Pocket paper [83] when using 250 lambdas (60s of end-to-end time). We attribute this difference to the fact that we used roughly half the number of lambdas (120 in our experiment).

We further analyzed the cost-performance for all the systems (see Figure 4.7). Our measurements demonstrate that Zip attains the best cost of all the configurations. Unlike the other systems, Zip does not incur any extra cost on top of that necessary for the VMs that are used to deploy the lambdas. AWS S3 incurs the cost of PUT/GET calls and data transfers (depending on size and destination of the transfer). This leads to a small extra cost (+8% compared to Zip) for the S3 PUT/GETs request issued during the shuffle during the workload. However, with Redis the end cost of the workload can be substantially higher due to the cost of provisioning the Redis servers (up to 4x in our experiments).

Last, we evaluated the impact of varying the Redis instances on end-to-end performance.

We found that from 1 to 2 instances, the shuffle time reduced by roughly 2x which indicates that the network capacity of the Redis cluster was a bottleneck. Adding a third Redis instance reduced the shuffle time by another 33% which indicates diminishing returns. We found that even with 3 instances, Zip is 40% faster than Redis and costs 4.8x less.

4.7.2 Micro-benchmarks

This sections provides a detailed evaluation of the performance, cost, and scalability of each individual communication primitive provided by Zip (i.e., reduce, broadcast, and shuffle). It evaluates the scalability of the Zip controller.

4.7.2.1 Reduce

Experiment setup We measured the end-to-end completion time of a distributed reduce of a numpy array (Figure 4.8 and 4.9). This experiment ran with a small, 100 bytes, and a large, 100MB, array to determine Zip's performance and the other systems' performance.

In this experiment, each lambda generates an array of 8-byte integers with the predetermined size and subsequently all lambdas perform a distributed sum of all the arrays using a reduce operation. Most of the time is spent in the data communication phase in this workload because the reduce computation is relatively quick to complete. In our reduce implementation with Redis, Pocket, and AWS S3 the lambdas initially store their input data in the respective system. Then one of the lambdas, assigned to do the reduce, is responsible for iteratively downloading the data to its local memory and reducing it. As an optimization, our implementation keeps track of which objects are already available to be downloaded, to start the download phase as soon as possible (polling). To achieve this, we use storage APIs that allow us to list the objects in the store.

Results We find that Zip achieves the best performance for both the 100 byte and 100MB experiments. For the experiment with 100 bytes, Zip with and without daemon has similar performance. For small data sizes, the faster data-path and local reduce of the daemon does not provide a speedup. However, for 100MB the daemon takes 40% less time (2.8s vs 4.6s) than the experiment without daemon. Here we observe a 2-3x higher transfer throughput between a lambda and a local daemon when compared to two co-located lambdas communicating through sockets (Zip without daemon).

The reduce experiment with Pocket performs the worse, even when compared to AWS S3. We attribute Pocket's poor performance to the fact that it sends an RPC to the metadata node to allocate storage space, for every 65K data block. Secondly, AWS S3 also scales poorly. This occurs because (a) AWS S3 has a lower per-lambda transfer throughput, and (b) all accesses to the store (including listing the store contents) have a higher overhead than the other systems. For instance, we find that listing an AWS S3 bucket takes on average 40ms, as opposed to under 2ms for Redis. On Redis the 100-byte reduce takes 2.4x and 1.7x longer than Zip with daemon and without daemon, respectively. This difference can be

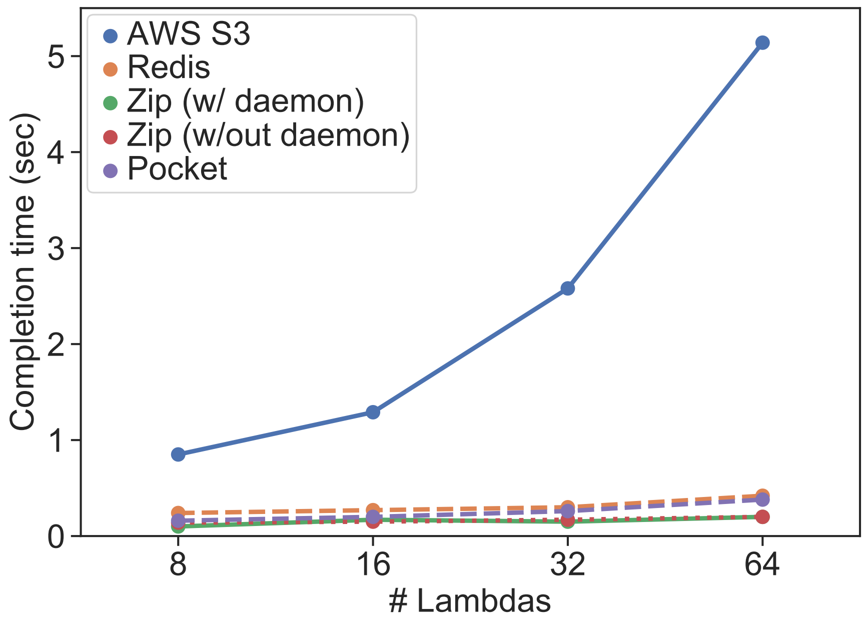


Figure 4.8: Reduce of 100 bytes numpy array across varying number of lambdas.

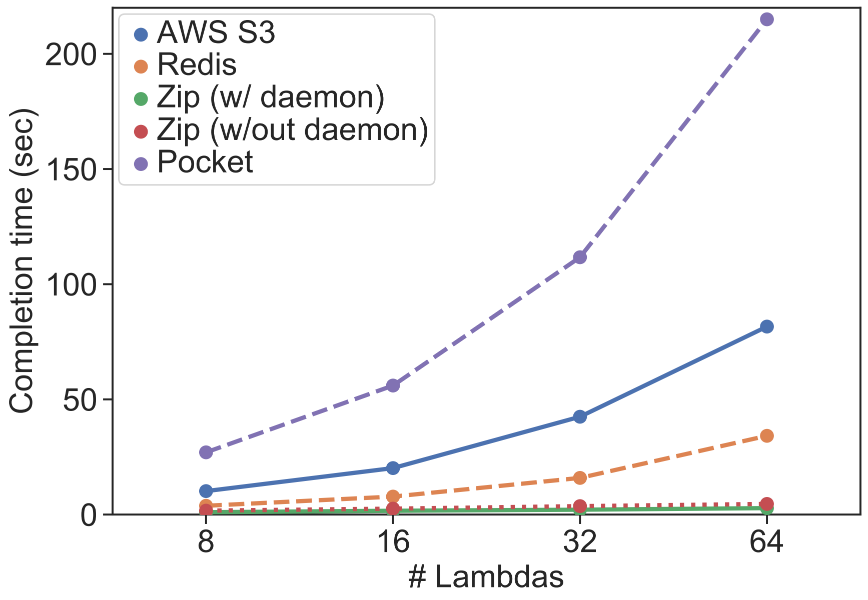


Figure 4.9: Reduce of 100MB numpy array across varying number of lambdas.

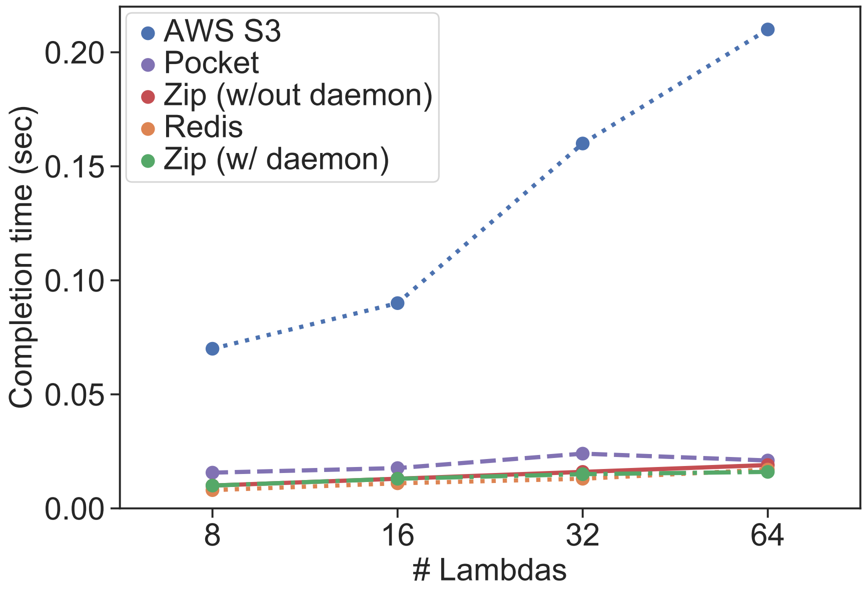


Figure 4.10: Broadcast of 100 bytes numpy array across varying number of lambdas. The amount of broadcast data transmitted increases proportionally with the number of lambdas.

attributed to the extra hop required to move data from the senders to the final reducer. For the 100MB experience Redis takes 3.7x longer than Zip with daemon with 8 lambdas and 12.3x longer with 64 lambdas. This sharp increase is caused by the increase in the number of simultaneous writes to Redis. With a large number of lambdas, Redis gets congested. This shows the additional effort required when using external storage systems for communication – developers need to carefully provision the storage system to meet the performance demands of the application.

4.7.2.2 Broadcast

Experiment setup We evaluated the end-to-end completion time of a distributed broadcast (Figure 4.10 and 4.12). In this workload, a single lambda transmits a numpy array to all other lambdas, i.e., every lambda receives the full data broadcasted. We varied the array sizes by running experiments with 100 byte and 100MB arrays. For the non-Zip experiments, one of the lambdas initially stores the broadcast data into the respective data store. Next, all other lambdas download that data. For these experiments lambdas continuously poll the data store to start the transfer.

Results We found that for the 100-byte broadcast, all the systems except AWS S3 have similar performance (within 30% of each other). Similarly to the reduce micro-benchmark, AWS S3's end-to-end time is considerably worse due to its high fixed cost for each data access (5-10ms). For the 100MB broadcast, at 64 lambdas, Zip is up to 3.4x faster than Redis, up to

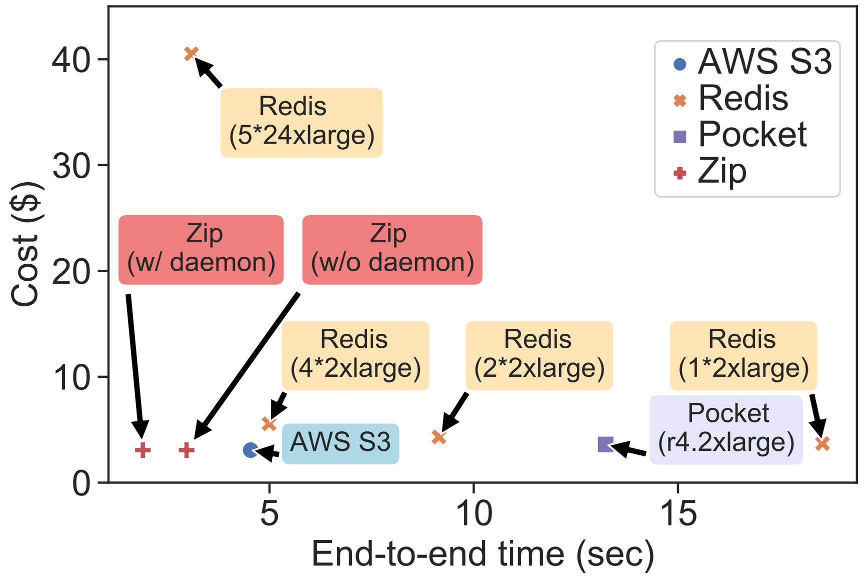


Figure 4.11: Performance-cost trade-off of 100MB broadcast between 64 lambdas.

6.3x faster than S3, and up to 7x faster than Pocket. As the number of servers increases, the end-to-end times increase except for AWS S3. AWS S3 maintains its end-to-end time even at 64 lambdas because it can auto-scale to meet the read throughput required by the lambdas. With Zip (with and without daemon) we observe a slight increase in completion time with the number of lambdas due to the extra number of hops required to traverse the channel communication tree. In contrast, Redis and Pocket run out of available bandwidth and their performance drops as the number of lambdas increases. This problem illustrates the importance of accurately estimating and provisioning the number of Redis servers required when using traditional approaches. In contrast, the Zip approach obviates the need for this complex and time-consuming task and additionally offers good end-to-end performance.

4.7.2.3 Shuffle

Experiment setup We also evaluated the performance of data shuffles with Zip, AWS S3, Pocket, and Redis. In this workload, a group of lambdas shuffles a list of 1M integers with each other according to a hash function. The hash function we used for this workload performs the modulo of a number in the list by the number of workers in the workload; hence, each worker sends a similar fraction of its input to other workers.

Results We measured the end-to-end time of this workload for Zip (without daemon), AWS S3, Pocket, and Redis. We found that Zip and Redis perform similarly with varying numbers of lambdas we used. In contrast, Pocket's performance is lower. This results from the fact that the Pocket API only receives Python strings, which requires an extra step to transform

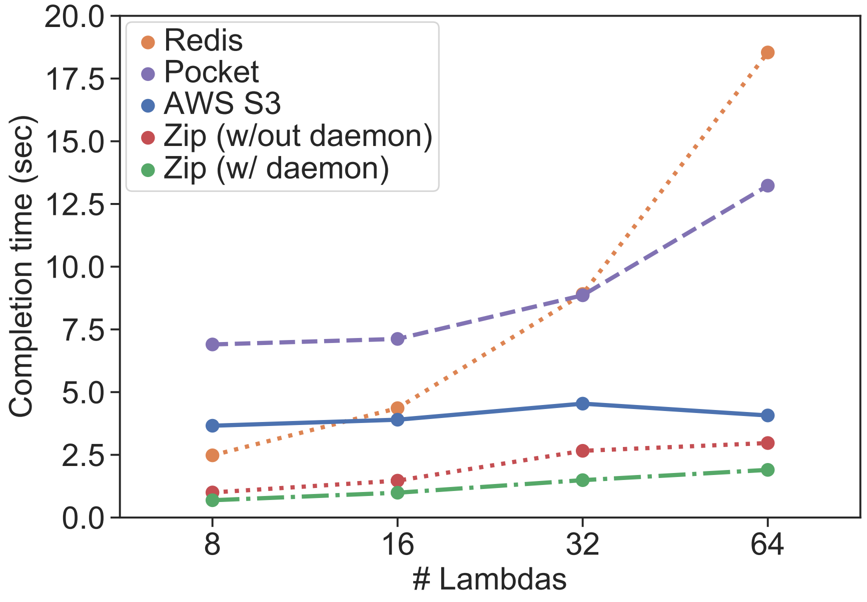


Figure 4.12: Broadcast of 100MB numpy array across varying number of lambdas. The amount of broadcast data transmitted increases proportionally with the number of lambdas.

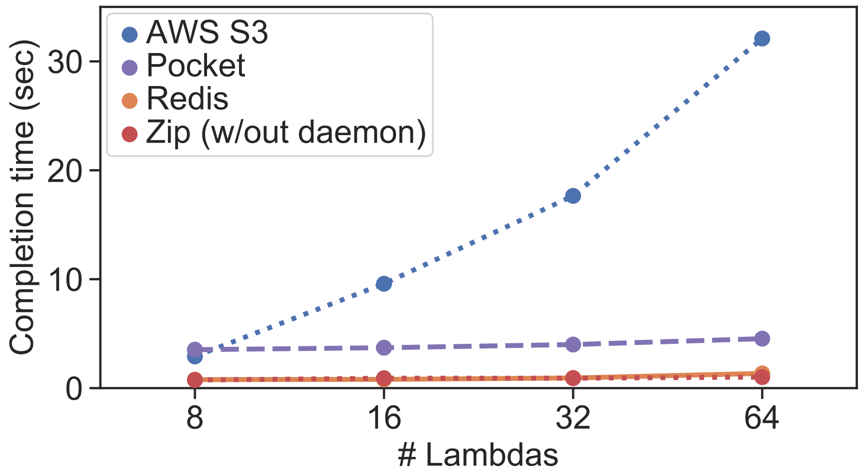


Figure 4.13: Shuffle of 1M integers between varying number of workers with different communication methods.

the shuffle data into this format before storing it into Pocket. Nonetheless, Pocket also maintains a constant end-to-end time up to 64 lambdas. Lastly, AWS S3 end-to-end time increases roughly linearly with the number of lambdas. We attribute this to the high fixed overhead for each PUT operation into AWS S3.

4.7.3 Controller scalability

Experiment setup To evaluate the scalability of the controller we measured the attained throughput when workers make many requests to the controller to join channels. In this situation, for every channel a lambda wants to join it sends a request to the controller. In turn, the controller adds the lambda to the channel (or creates one if it does not exist) and replies with metadata information about the channel (e.g., address of the channel root lambda) and pertaining to the lambda requesting to join (e.g., addresses of parent and children). To this end, we deployed 8 lambdas in one server and in another we deployed the controller. We made each lambda join 20K uniquely named channels and recorded the throughput each lambda was able to achieve.

Results We observed that each lambda on average was able to join 5.5K channels per second. Similarly, we measured the same throughput but this time in the case when all the lambdas create 20K separate handles to the same channel, which generates a channel with 160K members. In this case the overall throughput dropped to an average of 3.5K joins per second. As the number of members in the channel increases, the controller has to traverse a longer path until it can create a leaf node for the new member.

4.8 Related Work

In this section we discuss and revisit related work in the areas of distributed communication in serverless. Here we try to explain the differences between our system Zip, and previous approaches in this space.

Serverless platforms There is a large number of commercial [17, 20, 48, 30, 58, 6] and open-source [54, 79, 10] serverless platforms. These platforms provide similar APIs for specifying and launching lambdas. Zip design is applicable to existing platforms. Other work focuses on optimizing different aspects of existing serverless platforms (e.g.,[78, 4, 23]). For instance, SOCK [78] and SAND [4] propose ways for improving the boot time of lambdas. SAND also improves on the performance of message passing between lambdas. Zip extends, rather than replaces, this work.

Storage systems Developers have in the past leveraged different cloud storage systems for storing application state and lambda communication, such as AWS S3[18], Redis[87], and Pocket/Apache Crail [83]. These systems are critical to today's serverless architectures because they enable scalable storage for the state of serverless applications. However, when used for lambda communication they can suffer from several issues, such as lack of scalability, need for manual resource provisioning and higher message passing latency.

Data and compute locality Previous work [105, 69] have proposed shipping computations closer to the data. For instance, the Shredder storage system can receive JavaScript functions which can interact with data locally without incurring network overheads. To achieve the goal of reducing network overhead for communication, Zip leverages data locality within each server through the Zip daemon.

Serverless network communication The ability to perform direct lambda-to-lambda communication in the AWS cloud by using off-the-shelf NAT traversal techniques has been previously discussed in GG [43] and instantiated in an open-source framework library [99]. Similarly, SAND [4] proposes a hierarchical message bus to provide higher-performance function chaining. Compared to these approaches, Zip also leverages direct lambda-to-lambda communication for better distributed communication performance. However, unlike previous work, Zip provides an entire API-to-backend stack.

Serverless frameworks Other work has proposed frameworks for running serverless workloads in a wide range of domains, such as map reduce [64, 40, 31], distributed sorting [84], linear algebra [93], machine learning workflows [24], and code compilation [43]. In designing Zip we leveraged the lessons of this previous work. For instance, PyWren's [64] scalability is limited by the scalability of AWS S3, which throttles traffic to just a few thousands of writes/second. Similarly, Cirrus [24] and Locus both require manually provisioned VMs to achieve scalability and performance. Cirrus deploys the parameter server on VMs during ML training because parameter servers have stringent requirements in terms of network (high bandwidths) and the ability to perform specialized optimization methods (e.g., Adagrad) on the data. These requirements are hard to achieve with other systems such as AWS S3, Redis or Pocket. Similarly, Locus leverages a deployment of VMs running Redis to provide a scalable and efficient data shuffle.

4.9 Summary

This chapter presents Zip, a system that addresses the lack of a high-performance and scalable abstraction for distributed communication in serverless. Zip provides developers with a simple high-level API, backed by a high-performance backend that seamlessly optimizes communication and recovers from failures. Zip's design outperforms approaches that use external storage systems for communication in terms of performance, ease of development, and deployment cost.

Our evaluation demonstrates that for a distributed sorting application, Zip provides up to 1.4x speedup and 4.8x lower cost compared to the fastest memory-based store configuration, and 1.73x speedup and 8% lower cost compared to using AWS S3. For specific communication patterns, we show that Zip provides up to 1.33x (shuffle), 4.4x (broadcast), and 12x (reduce) speedups when compared to the second fastest alternative.

In the next chapter, we discuss open challenges in serverless computing and propose research directions to address them.

Chapter 5

Open Research Challenges in Serverless

Just like in the first years of infancy of cloud computing, serverless computing is rapidly evolving and claiming its space within the cloud ecosystem. To win the adoption of users and organizations, the current generation of serverless platforms needs to (a) provide performance that more closely matches that of traditional VM-based platforms, and (b) provide significantly better support for use cases and workloads that are challenging to execute with traditional VM-based platforms.

In this chapter we discuss two concrete open challenges, and possible research directions to address them, that illustrate this. In Section 5.1 we discuss the need for serverless platforms to take advantage of language runtimes capabilities for compiling code in order to provide faster code execution. In Section 5.2 we discuss how interactive environments such as Jupyter Notebooks [65] can evolve to provide a more seamless integration with serverless computing platforms for interactive tasks.

5.1 Faster Execution with Platforms-Runtimes Co-design

JavaScript and Python are the two most used languages in today's serverless platforms. These two interpreted languages alone account for almost 90% of the total number of functions submitted to AWS Lambda according to estimates from Dashbird [96]. This class of languages are supported by the vast majority of serverless platforms today (see Table 2.1 in Chapter 2). This support comes in the form of up-to-date runtimes, large and well-maintained set of dependencies and libraries, and native APIs for all the BaaS services. Such languages provide a number of benefits when compared to unmanaged languages, such as higher portability and ease of development.

Python and JavaScript developers do not have to compile their code to execute it. To mitigate the performance overheads due to interpreting code, modern runtimes are equipped

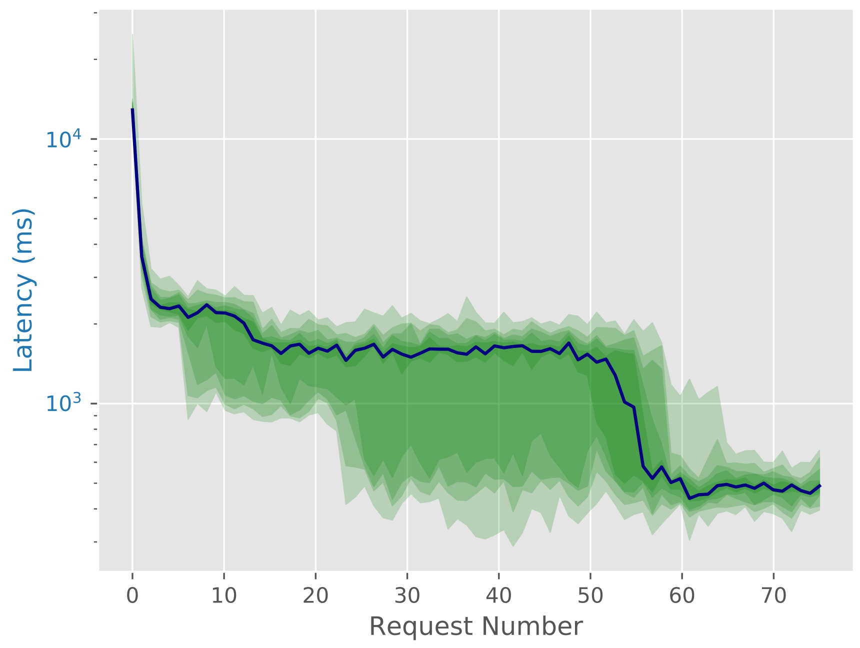


Figure 5.1: Execution time of a sequence of identical serverless requests. Each request constructs and evaluates the same math expression. The code is written in JavaScript running on GraalVM.

with a just-in-time (JIT) compiler that can identify pieces of code that are executed frequently and compile them down to the underlying hardware to produce more efficient code. Furthermore, some runtimes, such as the JVM [73] and GraalVM [21], can leverage information gathered during the execution of a program – code profiling – to generate further optimizations.

We found these two compilation stages to produce significant improvements in code efficiency when compared to just running interpreted code. For instance, when running a simple JavaScript program on top of GraalVM (see Figure 5.1) and without restarting the environment, the execution time of each individual request can vary significantly. We observe that the first few requests take significantly long because the code is being interpreted. After few requests, once the runtime has been given enough time to compile the code down to binary, the execution time decreases by 6x. After roughly 50 requests, we observe another significant decrease (2x) in the execution time of each request, this time due to the profiling compiler, generating additional optimizations on the code of our function. These results illustrate the importance of JIT and profiling compilers for developers of managed languages such as Python and JavaScript.

Unfortunately, today's serverless computing execution model is not well suited to take advantage of these optimizations. For instance, when running a MapReduce map stage that requires many lambdas, each lambda is going to be executed in a newly constructed container. Since the runtime has just been constructed for this execution, the performance of this task will be running at the lowest possible performance (left side of the graph in Figure 5.1). Because this map stage is going to be run only once, the runtime will never have the chance to compile the code, firstly using the JIT compiler and secondly using the profiler compiler, and achieve the optimal execution time.

This problem suggests that serverless platforms need to be redesigned to take advantage of the runtimes compilation features for better performance.

5.2 Interactive Environments for Serverless

In Chapter 3 we showed how Cirrus leverages serverless computing for interactive ML workflows spanning 3 types of tasks: dataset preprocessing, ML training and ML hyperparameter tuning. Cirrus provides better interactivity than other approaches because it leverages the fast invocation of lambdas for the different ML stages, which saves time for users. Furthermore, Cirrus provides a dashboard that provides some introspection into the computations being performed in the serverless platform.

Cirrus is a first step towards interactive workflows backed by serverless. Here we discuss ways in which interactivity with serverless could be improved, in particular by augmenting interactive environments, such as Jupyter Notebooks, to provide a more productive experience for users. Our discussion is agnostic to the particular programming model being used – it could be a specialized API such as the one provided by Cirrus, MapReduce provided PyWren, or other – and focus instead on the integration between the interactive environment and the serverless platform.

We start with the premise that serverless platforms should feel more like an extension of the data scientists laptop, rather than a distant separate system. This mental model can help simplify the way data scientists use serverless platforms. Since it provides an experience that is familiar for data scientists, it will help the transition towards shifting some of their computations to serverless platforms. Within this model, we identify 3 goals that interactive environments should move towards to provide simpler and more efficient utilization of serverless: state introspection, interactivity and seamless connectivity.

First, data scientists should be able to access the program state on the serverless side in the same way they do when accessing the state within their local Python runtime, even when the state is being modified. This allows data scientists to inspect, and debug the state of computations. In the current version of Cirrus users have to stop or wait until termination to see the result of the computation, and even then users cannot inspect other variables of interest. Second, environments should provide first-class support for asynchronous remote tasks. For instance, consider a ML training task running on serverless. In this situation, the data scientist might want to inspect the current version of the model being trained and test its performance on a separate test set while training has not yet terminated. In this case, the environment needs to allow temporarily pushing to the training task to the background while the data scientist proceeds with the performance analysis. Lastly, environments should be designed to gracefully tolerate failures on serverless. Interactive workloads are more prone to failures because the number of potential failures is greater. Potential sources of failures are: local to remote disconnections, hardware and software failures arising from the serverless platform or from the users computations.

5.3 Summary

In this chapter we proposed and discussed two research directions aimed at improving the performance and usability of serverless computing platforms. First, we proposed designing co-designing platforms and runtimes in a way that allows platforms to take advantage of the decades of research on runtime compilation and optimization of code. Our preliminary results show that runtimes can be very effective at optimizing user's code by (a) compiling high-level code down to machine code (JIT), and (b) gathering statistics about the code execution and generating optimizations from those statistics (profile-guided optimizations). Second, we proposed redesigning interactive environments currently used by data scientists to provide tighter integration between those environments and the serverless platforms. This can simplify the adoption of serverless computing for interactive and exploratory workloads.

Chapter 6

Conclusion

The history of the Cloud is largely a history of relentless simplification of the process of software development and deployment. The Cloud has significantly simplified the process of planing, purchasing and setting up hardware. As a consequence, software developers no longer have to align their application development processes with the slow and expensive cycles of hardware purchasing, shipping, installation and testing. This has dramatically accelerated software development cycles, and software innovation.

However, software developers still face the burden of managing cloud resources for their applications. For instance, developers today still have to choose, deploy and manage VMs. Aspects such as fault-tolerance, scalability, configuration and security are still largely the responsibility of developers, on top of the already time-consuming aspects of application development.

Serverless computing provides a step towards abstracting these responsibilities from developers. The serverless computing model based on a FaaS+BaaS has the potential to simplify many important classes of applications. In this thesis, we focused our attention on the class of highly distributed applications because they are one of the most compelling cases for serverless computing. Distributed applications are inherently complex, and have strong demands in terms of scalability and performance.

We have shown through analysis and experimentation that highly distributed applications are challenging to implement in serverless largely due to complexities inherent to the serverless model, such as limited resources, and due to the lack of support for distributed communication.

To address the challenge of serverless computing complexities, we proposed a system, Cirrus, that allows data scientists and ML practitioners to easily run ML workflows by abstracting them from the underlying cloud resources. At the same time, Cirrus can leverage serverless computing to elastically scale its resources to the needs of each stage of the ML workflow.

Designing and developing Cirrus was our gateway to understand the complexities of developing data and compute-intensive workloads in serverless. On one hand, it was with Cirrus that we understood how serverless computing can satisfy the need for highly interactive and scalable systems. The very fast invocation times for thousands of functions, the easy process for developing and submitting a single function and the scalability of the cloud storage systems were some of the ingredients we early on identified as critical for the future of serverless. On the other hand, developing Cirrus was also challenging due to the limited hardware resources in many important aspects such as CPUs, memory and network. We also found modern serverless platforms to lack good tools for debugging and monitoring the execution in lambdas. We expect to see significant improvements in many of these operational aspects in the next couple of years.

The Cirrus design provides a blueprint for future interactive frameworks for serverless. The combination of a stateful frontend and a stateless backend, per-stage provisioning, and simple APIs with a dashboard are key ingredients for providing future interactive computing on serverless.

It was during the design and development of Cirrus that we realized the pressing need of better support for distributed communication in serverless platforms for workloads such as ML training. This led us to design and implement Zip. Zip tackles the problem of lack of support for distributed communication in serverless, one of the most critical shortcomings of serverless. Zip provides an API for widely used communication patterns and is backed by a high-performance and scalable backend. Zip's architecture nicely extends the design of existing serverless platforms to provide significantly better performance than current approaches for serverless distributed communication.

The problem of serverless communication, the focus of Zip, is a major barrier to the adoption of serverless for many workloads. For this reason, we expect Zip to be a significant contribution to serverless by bringing its benefits to workloads such as ML training, data analytics, scientific computing, and others. We hope that serverless providers will move towards platform redesigns that provide such support.

To conclude, in this dissertation we tackled major problems of serverless computing and modern serverless platforms. We designed and developed two systems, Cirrus and Zip, that make serverless distributed computing for ML, data analytics and workloads more easily accessible and efficient. We hope that our work can pave the way for future work in these areas.

Bibliography

[1] Martıén Abadi et al. “Tensorflow: A system for large-scale machine learning”. In: *OSDI*. 2016.

[2] Alekh Agarwal et al. “A reliable effective terascale linear learning system”. In: *The Journal of Machine Learning Research* 15.1 (2014).

[3] Marcos K. Aguilera et al. “Designing Far Memory Data Structures: Think Outside the Box”. In: *HotOS*. New York, NY, USA: ACM, 2019.

[4] Istemi Ekin Akkus et al. “SAND: Towards High-Performance Serverless Computing”. In: *2018 USENIX Annual Technical Conference (USENIX ATC 18)*. 2018.

[5] Ahsan Ali et al. “Batch: Machine Learning Inference Serving on Serverless Platforms with Adaptive Batching”. In: *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*. SC ’20. 2020.

[6] *Alibaba Functions*. `https://www.alibabacloud.com/products/function-compute`.

[7] *Amazon*. `www.amazon.com`. 2020.

[8] *Amazon AWS*. `https://aws.amazon.com/`. 2020.

[9] Apache. *Apache Hadoop*. `http://hadoop.apache.org`.

[10] *Apache OpenWhisk*. `https://openwhisk.apache.org/`.

[11] Michael Armbrust et al. “Spark sql: Relational data processing in spark”. In: *SIGMOD*. 2015.

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

[13] AWS. *AWS ElasticCache*. `https://aws.amazon.com/elasticache/`.

[14] *AWS DynamoDB*. `https://aws.amazon.com/dynamodb/`.

[15] *AWS EMR*. `https://aws.amazon.com/emr/`.

[16] *AWS Greengrass*. `https://aws.amazon.com/greengrass/`.

[17] *AWS Lambda*. `https://aws.amazon.com/lambda/`.

[18] *AWS S3*. `https://aws.amazon.com/s3/`.

[19] *AWS SageMaker*. `https://aws.amazon.com/sagemaker/`.

[20] *Azure Functions*. `https://azure.microsoft.com/en-us/services/functions/`.

[21] Daniele Bonetta. "GraalVM: Metaprogramming inside a Polyglot System (Invited Talk)". In: META 2018. Boston, MA, USA, 2018.

[22] Léon Bottou, Frank E. Curtis, and Jorge Nocedal. *Optimization Methods for Large-Scale Machine Learning*. 2018.

[23] James Cadden et al. "SEUSS: Skip Redundant Paths to Make Serverless Fast". In: *EuroSys*. 2020.

[24] Joao Carreira et al. "Cirrus: A Serverless Framework for End-to-End ML Workflows". In: *SoCC*. 2019.

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

[26] Tianqi Chen et al. "Mxnet: A flexible and efficient machine learning library for heterogeneous distributed systems". In: *arXiv preprint* (2015).

[27] *Cloud Dataflow*. `https://cloud.google.com/dataflow`.

[28] *Cloud Firestore*. `https://firebase.google.com/docs/firestore`.

[29] *Cloud Pub/Sub*. `https://cloud.google.com/pubsub`.

[30] *Cloudflare Workers*. `https://www.cloudflare.com/products/cloudflare-workers/`.

[31] *Corral*. `https://github.com/bcongdon/corral`.

[32] Daniel Crankshaw et al. "Clipper: A Low-Latency Online Prediction Serving System". In: *NSDI*. 2017.

[33] *Criteo Dataset*. `http://labs.criteo.com/2014/02/kaggle-display-advertising-challenge-dataset/`.

[34] Databricks. *Apache Spark the fastest open source engine for sorting a petabyte*. `https://databricks.com/blog/2014/10/10/spark-petabyte-sort.html`. 2014.

[35] Christina Delimitrou and Christos Kozyrakis. "Quasar: Resource-efficient and QoS-aware Cluster Management". In: *ASPLOS*. 2014.

[36] *Disaggregated Rack*. `http://www.opencompute.org/wp/wp-content/uploads/2013/01/OCP_Summit_IV_Disaggregation_Jason_Taylor.pdf`. OpenCompute Summit. 2013.

[37] *Display Advertising Challenge*. `https://www.kaggle.com/c/criteo-display-ad-challenge`.

[38] *Dropbox*. `https://www.dropbox.com/`.

[39] Vojislav Dukic et al. "Photons: Lambdas on a Diet". In: *SoCC*. 2020.

[40] *FaastJS: Serverless batch computing made simple*. `https://faastjs.org/`.

[41] Message P Forum. *MPI: A Message-Passing Interface Standard.* Tech. rep. Knoxville, TN, USA, 1994.

[42] Sadjad Fouladi et al. “Encoding, Fast and Slow: Low-Latency Video Processing Using Thousands of Tiny Threads”. In: *NSDI.* 2017.

[43] Sadjad Fouladi et al. “From Laptop to Lambda: Outsourcing Everyday Jobs to Thousands of Transient Functional Containers”. In: *USENIX ATC.* 2019.

[44] Armando Fox et al. “Above the clouds: A berkeley view of cloud computing”. In: (2009).

[45] *Google.* `www.google.com`. 2020.

[46] *Google AppEngine.* `https://cloud.google.com/appengine`.

[47] *Google BigQuery.* `https://cloud.google.com/bigquery`.

[48] *Google Cloud Functions.* `https://cloud.google.com/functions/`.

[49] *Google cloudml-samples.* `https://github.com/GoogleCloudPlatform/cloudml-samples`.

[50] *Google Firebase.* `https://firebase.google.com/`.

[51] *Google Workspace.* `https://workspace.google.com/`.

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

[53] Joseph M Hellerstein et al. “Serverless computing: One step forward, two steps back”. In: *arXiv preprint arXiv:1812.03651* (2018).

[54] Scott Hendrickson et al. “Serverless computation with openlambda”. In: *Elastic* ().

[55] *Heroku.* `https://www.heroku.com/`.

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

[57] *Huawei DC 3.0.* `www.huawei.com/ilink/en/download/HW_349607&usg=AFQjCNEOm-KD71dxJeRf1cJSkNaJbpNgnw&sig2=opyc-KxWX3Vb7Jj11dyaMA`. [Online; accessed 20-Jan-2017]. 2017.

[58] *IBM Functions.* `https://console.bluemix.net/openwhisk/`.

[59] Plotly Technologies Inc. *Collaborative data science.* 2015. URL: `https://plot.ly`.

[60] *Intel Rack Scale Architecture.* `http://www.intel.com/content/www/us/en/architecture-and-technology/rack-scale-design-overview.html`. 2017.

[61] Vatche Ishakian, Vinod Muthusamy, and Aleksander Slominski. “Serving deep learning models in a serverless platform”. In: *IC2E.* 2018.

[62] Yangqing Jia et al. “Caffe: Convolutional Architecture for Fast Feature Embedding”. In: *arXiv preprint arXiv:1408.5093* (2014).

[63] Eric Jonas et al. "Cloud Programming Simplified: A Berkeley View on Serverless Computing". In: *arXiv preprint arXiv:1902.03383* (2019).

[64] Eric Jonas et al. "Occupy the Cloud: Distributed Computing for the 99%". In: *CoRR* abs/1702.04024 (2017). URL: `http://arxiv.org/abs/1702.04024`.

[65] *Jupyter Notebook*. `http://www.https://jupyter.org/`.

[66] Ana Klimovic et al. "Flash Storage Disaggregation". In: *EuroSys*. 2016.

[67] Ana Klimovic et al. "Understanding Ephemeral Storage for Serverless Analytics". In: *2018 USENIX Annual Technical Conference (USENIX ATC 18)*. Boston, MA: USENIX Association, 2018, pp. 789–794.

[68] *Kubeless*. `https://kubeless.io/`.

[69] Chinmay Kulkarni et al. "Splinter: Bare-Metal Extensions for Multi-Tenant Low-Latency Storage". In: *OSDI*. 2018.

[70] John Langford. *Allreduce (or MPI) vs. Parameter server approaches*. `https://hunch.net/?p=151364`. 2014.

[71] Mu Li et al. "Communication efficient distributed machine learning with the parameter server". In: *NIPS*. 2014.

[72] Mu Li et al. "Scaling Distributed Machine Learning with the Parameter Server". In: 2014.

[73] Tim Lindholm et al. *The Java Virtual Machine Specification, Java SE 8 Edition*. 1st. Addison-Wesley Professional, 2014. ISBN: 013390590X.

[74] *Microsoft*. `www.microsoft.com`. 2020.

[75] Ingo Müller, Renato Marroquín, and Gustavo Alonso. "Lambada: Interactive Data Analytics on Cold Data Using Serverless Cloud Infrastructure". In: *SIGMOD*. 2020.

[76] *Multiverso*. `https://github.com/Microsoft/Multiverso`.

[77] *Netflix Dataset*. `https://www.kaggle.com/netflix-inc/netflix-prize-data`.

[78] Edward Oakes et al. "SOCK: Rapid Task Provisioning with Serverless-Optimized Containers". In: *USENIX ATC*. 2018.

[79] *OpenFaaS*. `https://www.openfaas.com/`.

[80] *Parse*. `https://parseplatform.org/`.

[81] Adam Paszke et al. "PyTorch: An Imperative Style, High-Performance Deep Learning Library". In: *NeurIPS*. 2019.

[82] Pitch Patarasuk and Xin Yuan. "Bandwidth optimal all-reduce algorithms for clusters of workstations". In: *Journal of Parallel and Distributed Computing* (2009).

[83] *Pocket: Elastic Ephemeral Storage for Serverless Analytics*. `https://web.stanford.edu/~anakli/pdf/pocket.pdf`.

[84] Qifan Pu, Shivaram Venkataraman, and Ion Stoica. “Shuffling, Fast and Slow: Scalable Analytics on Serverless Infrastructure”. In: *NSDI*. 2019.

[85] *Qubole Announces Apache Spark on AWS Lambda*. `https://www.qubole.com/blog/spark-on-aws-lambda/`.

[86] Benjamin Recht et al. “Hogwild: A lock-free approach to parallelizing stochastic gradient descent”. In: *NIPS*. 2011.

[87] Redis. *Redis*. `https://redis.io/`.

[88] Charles Reiss et al. “Heterogeneity and dynamicity of clouds at scale: Google trace analysis”. In: *SoCC*. 2012.

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

[90] Josep Sampé et al. “Serverless data analytics in the ibm cloud”. In: *Proceedings of the 19th International Middleware Conference Industry*.

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

[92] Mohammad Shahrad et al. “Serverless in the Wild: Characterizing and Optimizing the Serverless Workload at a Large Cloud Provider”. In: *USENIX ATC*. 2020.

[93] Vaishaal Shankar et al. “Serverless Linear Algebra”. In: *SoCC*. 2020.

[94] Simon Shillaker and Peter Pietzuch. “Faasm: Lightweight Isolation for Efficient Stateful Serverless Computing”. In: *USENIX ATC*. 2020.

[95] *Spotinst Functions*. `https://spotinst.com/products/spotinst-functions/`.

[96] *State of Lambda functions in 2019 by Dashbird*. `https://dashbird.io/blog/state-of-lambda-functions-2019/`. 2019.

[97] Yang Tang. “Lambdata: Optimizing Serverless Computing by Making Data Intents Explicit”. In: 2020.

[98] Shelby Thomas et al. “Particle: Ephemeral Endpoints for Serverless Networking”. In: *SoCC*. 2020.

[99] Tim Wagner. *ServerlessNetworkingClients - Client SDKs for ServerlessNetworking*. `https://networkingclients.serverlesstech.net/`. 2020.

[100] Liang Wang et al. “Peeking Behind the Curtains of Serverless Platforms”. In: *USENIX ATC*. 2018.

[101] *Weblab*. `https://weblab.io/`.

[102] Jinliang Wei et al. “Managed communication and consistency for fast data-parallel iterative analytics”. In: *SoCC*. 2015.

[103] Eric P Xing et al. “Petuum: A new platform for distributed machine learning on big data”. In: *IEEE Transactions on Big Data* (2015).

[104] Matei Zaharia et al. "Spark: Cluster Computing with Working Sets". In: *HotCloud*. 2010.

[105] Tian Zhang et al. "Narrowing the Gap Between Serverless and Its State with Storage Functions". In: *SoCC*. 2019.