The Design of Stateful Serverless Infrastructure

by

Vikram Sreekanti

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

Committee in charge:

Professor Joseph M. Hellerstein, Chair
Professor Joseph E. Gonzalez
Professor Fernando Perez

Summer 2020

The Design of Stateful Serverless Infrastructure

Abstract

The Design of Stateful Serverless Infrastructure

by

Vikram Sreekanti

Doctor of Philosophy in Computer Science

University of California, Berkeley

Professor Joseph M. Hellerstein, Chair

Serverless computing has become increasingly popular in the last few years because it simplifies the developer's experience of constructing and deploying applications. Simultaneously, it enables cloud providers to pack multiple users' workloads into shared physical resources at a fine granularity, achieving higher resource efficiency. However, existing serverless Function-as-a-Service (FaaS) systems have significant shortcomings around state management—notably, high-latency IO, disabled point-to-point communication, and high function invocation overheads.

In this dissertation, we present a line of work in which we redesign serverless infrastructure to natively support efficient, consistent, and fault-tolerant state management. We first explore the architecture of a stateful FaaS system we designed called Cloudburst, which overcomes many of the limitations of commercial FaaS systems. We then turn to consistency and fault-tolerance, describing how we provide read atomic transactions in the context of FaaS applications. Finally, we describe the design and implementation of a serverless dataflow API and optimization framework specifically designed to support machine learning prediction serving workloads.

To my parents

Contents

List of Figures

List of Tables

Acknowledgments

This thesis would not have been possible without the support, advice, friendship, and guidance of a great many people. First and foremost is my advisor Joe Hellerstein, who hired me as a research engineer straight out of college and gave me plenty of freedom and support while I stumbled around blindly, working on the Ground project. He advised and shepherded me through the constant evolution of my interests and spent many a late night helping me rewrite paper introductions. Most importantly, he always encouraged me to do great work while reminding me that it is important to maintain balance and to make time for myself. Just as important is my *de facto* co-advisor, Joey Gonzalez, who was just as integral to my success. His infectious enthusiasm for research and for exploring new ideas always helped me get excited about new projects, and he constantly pushed me to improve ideas, arguments, papers, and talks. I am a better researcher, writer, thinker, and person thanks to their efforts.

Chenggang Wu was my partner-in-crime throughout grad school, and we collaborated on every paper either of us wrote for the past three years. His creativity, critical thinking, and dedication to research always motivated me to do better, and I am grateful for his help, advice, and friendship. Much of the work in this thesis has its origins in long conversations with him.

Many other grad students, postdocs, and undergrads helped improve ideas, commiserated about grad school, and played a key role in maintaining sanity over the last four years: Michael Whittaker, Johann Schleier-Smith, Rolando Garcia, Yifan Wu, Charles Lin, Jose Faleiro, Alexey Tumanov, Hari Subbaraj, Saurav Chhatrapati, Dan Crankshaw, Gabe Fierro, Nathan Pemberton, Anurag Khandelwal, Neeraja Yadwadkar, Zongheng Yang, Evan Sparks, Eyal Sela, Moustafa Abdelbaky. The support staff in the AMP and RISE labs have been amazing: Kattt, Boban, Shane, Jon, and Jey, we are all grateful for your hard work.

My friends outside of research were even more important for sanity: They celebrated paper deadlines, sympathized over paper reviews, and picked me up when I was the most frustrated. Saavan, Divya, Suhani, Praj, Tej, Eric, Anja, Roshan, Teeks, Mayur, Sahana, Shruti, Raghav, Nathan, Sri, Jason, Sri, Abishek, Abheek, Varun, Abhinav, thank you for everything.

None of this would have been possible without the bottomless well of love and support from my family. Vibhav and Shreya, thank you for advice and wisdom over the years, for being friends & confidants, and for trying to get Annika to play with me even when she would rather run away screaming. Mom and Dad, thank you for all of the sacrifices you have made over the last 30 years—nothing we have done would have been possible without your hard work, encouragement, tough love, and (most importantly) cooking. You have given us everything we could ask for, and we couldn't be more grateful.

Finally, much of the credit for my success goes to my best friend and my biggest fan. Nikita, your never-ending love and encouragement over the last few years have been constant sources of motivation and inspiration for me. Your family has treated me as one of their own, and I could not be more grateful for their love. You have put up with paper sprints, late nights, long hours spent refining writing, and frustrated rants about paper reviews—to name just a few of the many things I bother you with. Through it all, you have given me all the support anyone could hope for. Thank you.

Chapter 1

Introduction

Cloud computing has revolutionized the modern computing landscape. The original promise of the cloud was that it would enable developers to scale their applications with on-demand provisioning, avoiding traditional concerns surrounding infrastructure provisioning and management. Simultaneously, cloud providers like Amazon Web Services (AWS), Google Cloud, or Microsoft Azure can pack multiple users' workloads into the same physical infrastructure using virtualization techniques and multitenant services, satisfying user demand with significantly higher resource efficiency [39].

The cloud has succeeded in fulfilling these goals. It has drastically simplified the management of servers and has enabled easy access to resources for developers of all kinds. In a pre-cloud world, only large companies with years-long requisitioning processes would have access to resources at large scales. And with more users than ever, cloud providers are able to make smarter decisions about resource provisioning and management. These properties have made the cloud virtually ubiquitous in software: Social networks, internet of things applications, large-scale machine learning, media services, and scientific computing applications have all been natively built on cloud infrastructure.

There are roughly two categories of cloud services: virtualized hardware resources and higher-level, hosted service offerings. The core cloud services for the last decade have provided users with familiar server-like abstractions that allow them to rent (virtualized) CPUs, RAM, and disks. However, systems that provide higher-level APIs—e.g., AWS Redshift data warehouse—have become increasingly popular for a variety of reasons, including simplified software management, stronger security guarantees, and increased reliability.

These services have become the *de facto* standard infrastructure for modern software, and their success has significantly improved the experience of most software developers. Nonetheless, the model and success of the cloud have introduced new challenges around infrastructure management; here, we look at developer experience and resource efficiency.

Developer Experience Thanks to the flexible resource acquisition afforded by public clouds, modern applications can easily run on hundreds or thousands of servers. At this scale, individual machines fail routinely [31], networks can be partitioned for extended periods [19], and data and computation are spread across large physical distances.

Managing 1,000s of servers by hand is obviously intractable, and in recent years, a variety of tools have emerged to simplify this process—systems referred to as Developer Operations (or "DevOps") tools. In the late 2000s and early 2010s, tools like Chef [129] and Puppet [87] emerged to automate the process of configuring and deploying new application servers. Recently, Kubernetes [78] has gained popularity by almost completely abstracting away resource management, from allocating machines and monitoring faults to configuring networks and ensuring security.

DevOps tools have simplified the deployment of large scale applications—it is undoubtedly easier to run an application on a hundred-node Kubernetes cluster than it is to allocate, configure, and periodically update each machine manually. However, these tools have only abstracted the *process* of resource management, not the *decision*; in other words, developers still must decide when and how to scale their applications up and down, which can be a cumbersome and error-prone process.

Worse yet, these systems have increasingly complex and nuanced APIs, which comes at the cost of simplicity and usability. For example, as of this writing, Kubernetes has close to 170 different APIs in its reference documentation, many of which are overlapping or closely related. This API is rapidly changing, meaning users need to constantly be up-to-date with new releases and best practices or risk quickly incurring technical debt.

Resource Efficiency. As we have discussed, virtualized servers offered by services like AWS EC2 allow cloud providers to pack users into shared physical hardware while offering strong isolation. However, the server-based abstraction has led to most applications "renting" servers for indefinite periods of time. Often, large enterprises will sign discounted deals with cloud providers over the course of multiple years—not dissimilar to the process of acquiring and writing down the cost of physical servers.

Furthermore, allocating servers at fine granularity in order to respond to unpredictable workload changes [50, 115] is often not possible because the wait time for resource acquisition and configuration can be on the order of minutes. Ideally, a developer would want her application to have the minimal resources required for the current workload. Instead, applications with unpredictable workload changes might provision for peak load—they allocate as many resources as they believe they will need for their largest workload volume. During non-peak times, the majority of these resources will be idle, unnecessarily billing the user for unused servers.

In 2014, Amazon Web Services (AWS) introduced a new service called Lambda, and with it, the concept of *serverless computing*. Serverless computing promises solutions to both of these challenges.

The initial incarnation of serverless computing was Functions-as-a-Service (FaaS), in which a developer simply writes a function, uploads it to AWS Lambda, and configures a trigger event (e.g., an API Gateway call, an image upload to S3). The cloud provider is responsible for detecting the trigger event, allocating resources to the function, executing the function, and returning the result to the user. The user is *only* billed for the duration of the function execution—once the function finishes, the cloud provider is free to use those resources for other purposes.

This model removes operational concerns from the developer's purview, simplifying their lives and making them more effective. In essence, serverless raises the deployment abstraction

from a general-purpose virtual machine or a container to a specific function. It also improves resource efficiency: Applications are only billed for the duration of the execution of their code, so there are resources are only allocated as necessary. From the provider's perspective, removing idle servers that might be used for a future load spike means requests from multiple customers can be aggressively packed into the same physical hardware at a much finer granularity. The same user demand can thus be serviced with fewer resources. This increased efficiency translates into better profit margins and, ideally, lower costs for users. As a result, serverless computing has become an exciting area of focus in both research [37, 66, 4, 65, 55, 73, 10, 133, 41, 38] and industry [8].

While the advent of the concept of serverless computing coincided with the development of the first FaaS system, AWS Lambda was by no means the first cloud service that offered the benefits of serverless computing. Consider a large-scale cloud object storage service like AWS S3: Users are not concerned how many hard disks are allocated for their data or in which datacenters those disks live. Instead, they simply upload data to S3, retrieve it when needed, and are billed based on the volume of data stored and the volume of data moved. From these two examples, we can derive a general definition of serverless computing that underlies the work in this thesis. A *serverless* system has two properties: (1) Users do not manually allocate resources but request resources on demand; and (2) users are billed based on the time and volume of resources *used* rather than the volume of resources *allocated*.

1.1 The State of Serverless

Thus far, we have looked at the evolution of cloud computing and how serverless infrastructure followed naturally from the scale and complexity of developing and deploying modern applications. Now, we take a closer look at the properties of existing serverless systems—FaaS systems in particular—highlighting where they succeed and fail. The shortcomings of existing FaaS systems drive much of the research in this thesis.

As we have already described, the simplicity of existing serverless platforms is perhaps their biggest asset—they significantly simplify the process of constructing, deploying, and managing applications. The approach of existing systems has been to provide extremely straightforward APIs while significantly limiting what applications can do on those platforms. For a certain class of application, this works well.

For example, the pywren project [66, 116] has shown that for embarrassingly parallel tasks—e.g., `map`—AWS Lambda can provide significant performance benefits at extremely large scale. Even for tasks that are less straightforward but have low-frequency communication—e.g., complex linear algebra operations like QR factorization or Cholesky decomposition—existing serverless systems can help applications scale well beyond what was previously possible on commodity infrastructure.

Commercial FaaS offerings have also been successful at simplifying the use and operation of other services—often called "glue code." AWS' published case studies on the use of Lambda [8] has a number of examples in which Lambda functions are used to perform database writes, trigger

emails to be sent, and insert data into queues for processing elsewhere. Standing up individual VMs to manage each one of these operations, making these services fault-tolerant, and replicating them for scale is tiresome and error-prone. In situations like these, FaaS services are at their best.

Beyond this point, however, existing FaaS infrastructure is insufficient for a number of common tasks. The most commonly cited complaint about FaaS systems is their limited function runtimes (15 minutes on AWS Lambda, 10 minutes on Google Cloud Functions, etc.)—but these limits have been increasing steadily over the last 5 years, and we believe they will continue to improve. More problematically, FaaS systems today have three state fundamental limitations around state: (1) they force all IO through limited-bandwidth networks; (2) they disable point-to-point communication; and (3) they have no access to specialized hardware.

Hellerstein et al. [56] details the implications of each of these limitations, but at a high-level, it is straightforward to see that they make FaaS ill-suited to a variety of applications. An application requiring fine-grained messaging (e.g., a distributed consensus protocol) would have to write all messages to storage and poll on storage keys to receive updates. An application that repeatedly accesses subsets of a large dataset (e.g., machine learning model training) will be forced to move its input data over the network from storage to compute multiple times, incurring high latencies and data access costs. In sum, any applications with state—whether from a storage system or communicated between functions—will find today's FaaS infrastructure to be insufficient.

A noteworthy exception is applications that focus on high-bandwidth, latency-insensitive workloads. For example, numpywren [116], a linear algebra extension to the pywren project, demonstrates the ability to effectively run supercomputer-scale matrix operations by taking advantage of AWS Lambda's high parallelism and AWS S3's bandwidth. Similarly, Starling [102] layers large-scale OLAP queries with AWS Lambda and S3, again effectively leveraging these services' parallelism and high bandwidth.

Solutions have begun to emerge that incrementally adapt existing FaaS systems to a wider variety of applications, but these largely work around FaaS' limitations rather than improving on them. For example, ExCamera [37] enables massively-parallel video encoding but requires a separate, server-based task manager and coordinator, which quickly becomes a bottleneck in deployments that must scale. Even numpywren requires an external fixed-deployment Redis instance that it uses to manage tasks and communicate between Lambda-based workers. However, these workarounds architecturally reintroduce all of the scaling and management challenges endemic to traditional cloud deployment models.

The goal of the work in this dissertation is to re-imagine the design of serverless infrastructure from scratch and to design serverless systems that treat state management as a first-class citizen. This allows us to support new classes of applications while maintaining the simplicity, scalability, and generality of serverless infrastructure.

1.2 Designing Stateful Serverless Infrastructure

Serverless computing is an attractive model because of the simplicity of its abstractions and the economic opportunity it offers both developers and cloud providers. To simplify programming

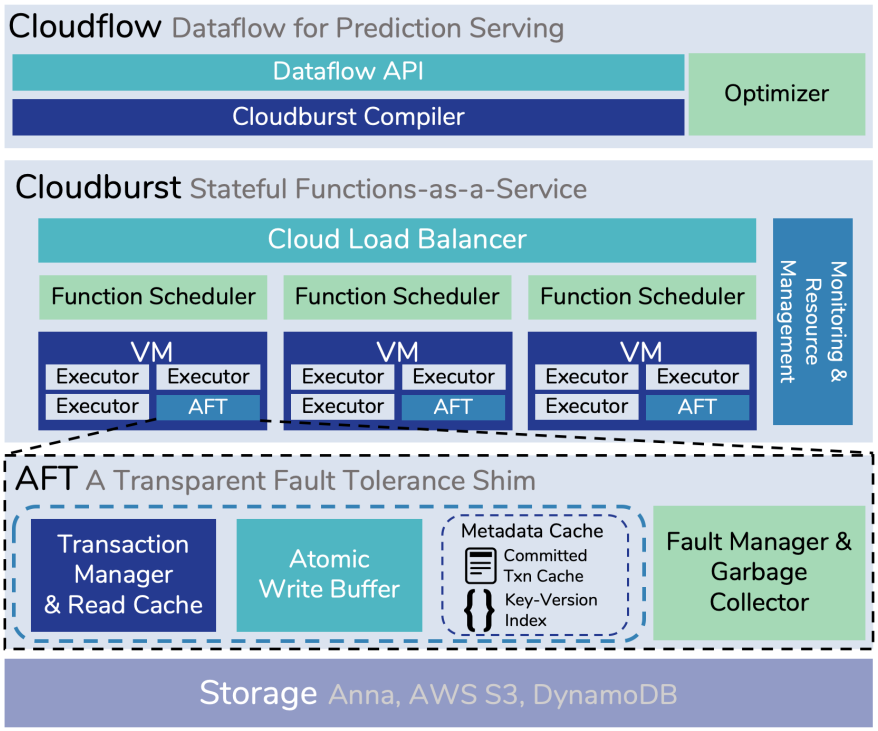


Figure 1.1: An overview of the systems that comprise the work in this thesis.

for as many developers as possible, serverless must evolve to support stateful abstractions like the patterns described in the previous section. The work in this dissertation is an initial set of designs and evaluations to tackle core challenges in making serverless infrastructure truly stateful. Figure 1.1 shows a high-level overview of the systems discussed in this thesis and how they fit together.

We begin in Chapter 2 by presenting the design of a new FaaS system, Cloudburst [121], that directly addresses the first two shortcomings of commercial FaaS offerings—high-latency, low-bandiwdth IO and no direct communication. At its core, the system accomplishes this by introducing caches that live on the same physical machines as function executors, which help enable low-latency data access and communication between workers. This design enables order-of-magnitude improvement over existing FaaS systems and matches the performance of state-of-the-art serverful distributed Python execution frameworks that require traditional reserved resource management.

Having introduced state into serverless functions, we turn to consistency and fault-tolerance in Chapter 3. Both for Cloudburst and for applications running on AWS Lambda, there are thorny questions around state consistency in the face of faults. FaaS systems by default blindly retry requests but will eagerly persist data into storage engines. This means that between retries of failed requests, parallel clients will read the persisted *partial* results of a function, and retried

functions may accidentally duplicate their results unless developers are careful to write idempotent programs. To solve this challenge, we introduce AFT [120], a shim layer that sits in between a serverless compute framework and storage engine. AFT assures fault tolerance by enforcing the *read atomic* consistency guarantee—a coordination-free consistency level that ensures that transactions only read from committed data in the order of commit. We show that AFT is able to prevent a significant number of anomalies while imposing a minimal overhead compared to standard architectures.

Finally, in Chapter 4, we study a modern, compute-intensive workload built on top of our serverless infrastructure: machine learning prediction serving [122]. We argue that prediction serving tasks should be modeled as simple dataflows, so they can be optimized with well-studied techniques like operator fusion. These dataflows are compiled down to execute on Cloudburst, and we extend Cloudburst to support key features specific to prediction dataflows. We also add support for GPU-based executors in Cloudburst, the third shortfall in commercial FaaS offerings described in the previous section. The dataflow model combined with serverless infrastructure allow us to beat state-of-the-art prediction serving systems by up to 2× on real-world tasks and—importantly—meet latency goals for compute-intensive tasks like video stream processing.

Taken together, the contributions of this thesis are as follows:

- Demonstrating that state management can and should be a key consideration in the design and implementation of serverless infrastructure.
- Significant performance improvements over state-of-the-art cloud services at both the compute and storage layers.
- Introducing consistency guarantees and programming frameworks that have familiar APIs which developers can easily leverage.
- Enabling a variety of new applications to run on serverless infrastructure, most notably real-time prediction serving.

Designing serverless infrastructure to support state management is not only possible but is in fact promising for a variety of real-world design patterns and applications. We believe that this line of work opens up possibilities for a variety of future work in the serverless space on topics such as strong consistency and programming interfaces, as well as enabling many new applications.

Chapter 2

Cloudburst: Stateful Functions-as-a-Service

As we have discussed, autoscaling is a hallmark feature of serverless infrastructure. The design principle that enables autoscaling in standard cloud infrastructure is the disaggregation of storage and compute services [48]. Disaggregation—the practice of deploying compute and storage services on separate, dedicated hardware—allows the compute layer to quickly adapt computational resource allocation to shifting workload requirements, packing functions into VMs while reducing data movement. Correspondingly, data stores (e.g., object stores, key-value stores) can pack multiple users' data storage and access workloads into shared resources with high volume and often at low cost. Disaggregation also enables allocation at multiple timescales: long-term storage can be allocated separately from short-term compute leases. Together, these advantages enable efficient autoscaling. User code consumes expensive compute resources as needed and accrues only storage costs during idle periods.

Unfortunately, today's FaaS platforms take disaggregation to an extreme, imposing significant constraints on developers. First, the autoscaling storage services provided by cloud vendors—e.g., AWS S3 and DynamoDB—are too high-latency to access with any frequency [135, 55]. Second, function invocations are isolated from each other: FaaS systems disable point-to-point network communication between functions. Finally, and perhaps most surprisingly, current FaaS offerings provide very slow nested function calls: argument- and result-passing is a form of cross-function communication and exhibits the high latency of current serverless offerings [4]. We return these points in §2.1, but in short, today's popular FaaS platforms only work well for isolated, *stateless* functions.

As a workaround, many applications—even some that were explicitly designed for serverless platforms—are forced to step outside the bounds of the serverless paradigm altogether. As discussed in Chapter 1, the ExCamera serverless video encoding system [37] depends upon a single server machine as a coordinator and task assignment service. Similarly, numpywren [116] enables serverless linear algebra but provisions a static Redis machine for low-latency access to shared state for coordination. These workarounds might be tenable at small scales, but they architecturally reintroduce the scaling, fault tolerance, and management problems of traditional

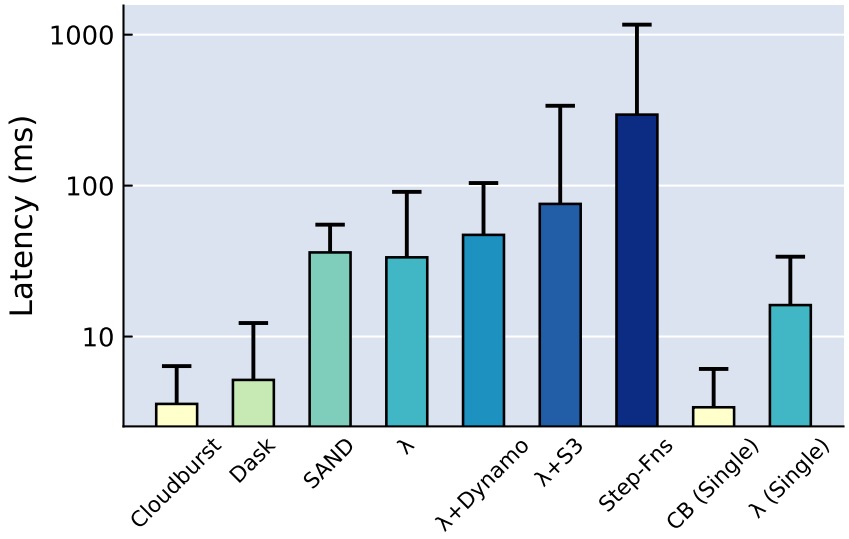


Figure 2.1: Median (bar) and 99th percentile (whisker) end-to-end latency for `square(increment(x: int))`. Cloudburst matches the best distributed Python systems and outperforms other FaaS systems by over an order of magnitude (§2.4).

server deployments.

Toward Stateful Serverless via LDPC

Given the simplicity and economic appeal of FaaS, we are interested in exploring designs that preserve the autoscaling and operational benefits of current offerings, while adding performant, cost-efficient, and consistent shared state and communication. This "stateful" serverless model opens up autoscaling FaaS to a much broader array of applications and algorithms. We aim to demonstrate that serverless architectures can support stateful applications while maintaining the simplicity and appeal of the serverless programming model.

For example, many low-latency services need to autoscale to handle bursts and also dynamically manipulate data based on request parameters. This includes webservers managing user sessions, discussion forums managing threads, ad servers managing ML models, and more. In terms of algorithms, a multitude of parallel and distributed protocols require fine-grained messaging, from quantitative tasks like distributed aggregation [72] to system tasks like membership [28] or leader election [7]. This class of protocols forms the backbone of parallel and distributed systems. As we see in §2.4, these scenarios are infeasible in today's stateless FaaS platforms.

To enable stateful serverless computing, we propose a new design principle: *logical disaggregation with physical colocation* (LDPC). Disaggregation is needed to provision, scale, and bill storage and compute independently, but we want to deploy resources to different services in close physical proximity. In particular, a running function's "hot" data should be kept physically

nearby for low-latency access. Updates should be allowed at any function invocation site, and cross-function communication should work at wire speed.

Cloudburst: A Stateful FaaS Platform

To that end, we present a new Function-as-a-Service platform called Cloudburst that removes the shortcomings of commercial systems highlighted above, without sacrificing their benefits. Cloudburst is unique in achieving logical disaggregation and physical colocation of computation and state, and in allowing programs written in a traditional language to observe consistent state across function compositions. Cloudburst is designed to be an autoscaling Functions-as-a-Service system—similar to AWS Lambda or Google Cloud Functions—but with new abstractions that enable performant, stateful programs.

Cloudburst achieves this via a combination of an autoscaling key-value store (providing state sharing and overlay routing) and mutable caches co-located with function executors (providing data locality). The system is built on top of Anna [140, 138], a low-latency autoscaling key-value store designed to achieve a variety of coordination-free consistency levels by using mergeable monotonic *lattice* data structures [117, 23]. For performant consistency, Cloudburst takes advantage of Anna's design by transparently encapsulating opaque user state in lattices so that Anna can consistently merge concurrent updates. We evaluate Cloudburst via microbenchmarks as well as two application scenarios using third-party code, demonstrating benefits in performance, predictable latency, and consistency. In sum, our contributions are:

1. The design and implementation of an autoscaling serverless architecture that combines logical disaggregation with physical co-location of compute and storage (LDPC) (§2.3).
2. The ability for programs written in traditional languages to enjoy coordination-free storage consistency for their native data types via *lattice capsules* that wrap program state with metadata that enables automatic conflict APIs supported by Anna (§2.2).
3. An evaluation of Cloudburst's performance on workloads involving state manipulation, fine-grained communication, and dynamic autoscaling (§2.4).

2.1 Motivation and Background

Although serverless infrastructure has gained traction recently, there remains significant room for improvement in performance and state management. In this section, we discuss common pain points in building applications on today's serverless infrastructure and explain Cloudburst's design goals.

Deploying Serverless Functions Today

Current FaaS offerings are poorly suited to managing shared state, making it difficult to build applications, particularly latency-sensitive ones. There are three kinds of shared state management

```
from cloudburst import *
cloud = CloudburstClient(cloudburst_addr, my_ip)
cloud.put('key', 2)
reference = CloudburstReference('key')
def sqfun(x): return x * x
sq = cloud.register(sqfun, name='square')

print('result: %d' % (sq(reference))
> result: 4

future = sq(3, store_in_kvs=True)
print('result: %d' % (future.get())
> result: 9
```

Figure 2.2: A script to create and execute a Cloudburst function.

that we focus on here: function composition, direct communication, and shared mutable storage.

Function Composition. For developers to embrace serverless as a general programming and runtime environment, it is necessary that function composition work as expected. Pure functions share state by passing arguments and return values to each other. Figure 2.1 (discussed in §2.4), shows the performance of a simple composition of side-effect-free arithmetic functions. AWS Lambda imposes a latency overhead of up to 40ms for a single function invocation, and this overhead compounds when composing functions. AWS Step Functions, which automatically chains together sequences of operations, imposes an even higher penalty. Since the latency of function composition compounds linearly, the overhead of a call stack as shallow as 5 functions saturates tolerable limits for an interactive service (∼200ms). Functional programming patterns for state sharing are not an option in current FaaS platforms.

Direct Communication. FaaS offerings disable inbound network connections, requiring functions to communicate through high-latency storage services like S3 or DynamoDB. While point-to-point communication may seem tricky in a system with dynamic membership, distributed hashtables (DHTs) or lightweight key-value stores (KVSs) can provide a lower-latency solution than deep storage for routing messages between migratory function instances [105, 124, 111, 110]. Current FaaS vendors do not offer autoscaling, low-latency DHTs or KVSs. Instead, FaaS applications resort to server-based solutions for lower-latency storage, like hosted Redis and memcached.

Low-Latency Access to Shared Mutable State. Recent studies [135, 55] have shown that latencies and costs of shared autoscaling storage for FaaS are orders of magnitude worse than underlying infrastructure like shared memory, networking, or server-based shared storage. Non-autoscaling caching systems like Redis and memcached have become standard for low-latency data access in the cloud. However these solutions are insufficient as they still require moving data over networks: As [135] shows, networks quickly become performance bottlenecks for existing FaaS systems. Furthermore, caches on top of weakly consistent storage systems like AWS S3 introduce thorny consistency challenges—these challenges are out of scope for this thesis but are discussed further in [121] and [139].

Towards Stateful Serverless

As a principle, LDPC leaves significant latitude for designing mechanisms and policy that co-locate compute and data while preserving correctness. We observe that many of the performance bottlenecks described above can be addressed by an architecture with distributed storage and local caching. A low-latency autoscaling KVS can serve as both global storage and a DHT-like overlay network. To provide better data locality to functions, a KVS cache can be deployed on every machine that hosts function invocations. Cloudburst's design includes consistent mutable caches in the compute tier (§2.3).

Regarding programmability, we would like to provide consistency without imposing undue burden on programmers, but Anna can only store values that conform to its lattice-based type system. To address this, Cloudburst introduces *lattice capsules* (§2.2), which transparently wrap opaque program state in lattices chosen to support Cloudburst's consistency protocols.

2.2 Programming Interface

Cloudburst accepts programs written in vanilla Python[1]. An example client script to execute a function is shown in Figure 2.2. Cloudburst functions act like regular Python functions but trigger remote computation in the cloud. Results by default are sent directly back to the client (line 8), in which case the client blocks synchronously. Alternately, results can be stored in the KVS, and the response key is wrapped in a `CloudburstFuture` object, which retrieves the result when requested (line 11-12).

Function arguments are either regular Python objects (line 11) or KVS references (lines 3-4). KVS references are transparently retrieved by Cloudburst at runtime and deserialized before invoking the function. To improve performance, the runtime attempts to execute a function call with KVS references on a machine that might have the data cached. We explain how this is accomplished in §2.3. Functions can also dynamically retrieve data at runtime using the Cloudburst communication API described below.

To enable stateful functions, Cloudburst allows programmers to `put` and `get` Python objects via the Anna KVS' API. Object serialization and encapsulation for consistency (§2.2) is handled transparently by the runtime. The latency of `put` and `get` is very low in the common case due to the presence of caches at the function executors.

For repeated execution, Cloudburst allows users to register arbitrary compositions of functions. We model function compositions as DAGs in the style of systems like Apache Spark [145], Dryad [62], Apache Airflow [2], and Tensorflow [1]. This model is also similar in spirit to cloud services like AWS Step Functions that automatically chain together functions in existing serverless systems.

Each function in the DAG must be registered with the system (line 4) prior to use in a DAG. Users specify each function in the DAG and how they are composed—results are automatically

[1]There is nothing fundamental in our choice of Python—we simply chose to use it because it is a commonly used high-level language.

API Name	Functionality
`get(key)`	Retrieve a key from the KVS.
`put(key, value)`	Insert or update a key in the KVS.
`delete(key)`	Delete a key from the KVS.
`send(recv, msg)`	Send a message to another executor.
`recv()`	Receive outstanding messages for this function.
`get_id()`	Get this function's unique ID

Table 2.1: The `Cloudburst` object communication API. Users can interact with the key value store and send and receive messages.

passed from one DAG function to the next by the Cloudburst runtime. The result of a function with no successor is either stored in the KVS or returned directly to the user, as above. Cloudburst's resource management system (§2.3) is responsible for scaling the number of replicas of each function up and down.

Cloudburst System API. Cloudburst provides developers an interface to system services— Table 2.1 provides an overview. The API enables KVS interactions via `get` and `put`, and it enables message passing between function invocations. Each function invocation is assigned a unique ID, and functions can advertise this ID to well-known keys in the KVS. These unique IDs are translated into physical addresses and used to support direct messaging.

Note that this process is a generalization of the process that is used for function composition, where results of one function are passed directly to the next function. We expose these as separate mechanisms because we aimed to simplify the common case (function composition) by removing the hassle of communicating unique IDs and explicitly sharing results.

In practice this works as follows. First, one function writes its unique ID to a pre-agreed upon key in storage. The second function waits for that key to be populated and then retrieves the first thread's ID by reading that key. Once the second function has the first function's ID, it uses the `send` API to send a message. When `send` is invoked, the executor thread uses a deterministic mapping to convert from the thread's unique ID to an IP-port pair. The executor thread opens a TCP connection to that IP-port pair and sends a direct message. If a TCP connection cannot be established, the message is written to a key in Anna that serves as the receiving thread's "inbox". When `recv` is invoked by a function, the executor returns any messages that were queued on its local TCP port. On a `recv` call, if there are no messages on the local TCP port, the executor will read its inbox in storage to see if there are any messages stored there. The inbox is also periodically read and cached locally to ensure that messages are delivered correctly.

Lattice Encapsulation

Mutable shared state is a key tenet of Cloudburst's design. Cloudburst relies on Anna's lattice data structures to resolve conflicts from concurrent updates. Typically, Python objects are not

lattices, so Cloudburst transparently encapsulates Python objects in lattices. This means every program written against Cloudburst automatically inherits Anna's consistency guarantees in the face of conflicting updates to shared state.

By default, Cloudburst encapsulates each bare program value into an Anna *last writer wins* (LWW) lattice—a composition of an Anna-provided global timestamp and the value. The global timestamp is generated by each node in a coordination-free fashion by concatenating the local system clock and the node's unique ID. Anna merges two LWW versions by keeping the value with the higher timestamp. This allows Cloudburst to achieve eventual consistency: All replicas will agree on the LWW value that corresponds to the highest timestamp for the key [134].

Depending on the input data type Cloudburst can also encapsulate state in a different data type that supports more intelligent merge functions. For example, if the input is a `map` or a `set`, Anna has default support for merging those data types via set union; in such cases, we use the corresponding `MapLattice` or `SetLattice` data structures instead of a last-writer wins lattice.

2.3 Architecture

Cloudburst implements the principle of logical disaggregation with physical colocation (LDPC). To achieve disaggregation, the Cloudburst runtime autoscales independently of the Anna KVS. Colocation is enabled by mutable caches placed in the Cloudburst runtime for low latency access to KVS objects.

Figure 2.3 provides an overview of the Cloudburst architecture. There are four key components: function executors, caches, function schedulers, and a resource management system. User requests are received by a scheduler, which routes them to function executors. Each scheduler operates independently, and the system relies on a standard stateless cloud load balancer (AWS Elastic Load Balancer). Function executors run in individual processes that are packed into VMs along with a local cache per VM. The cache on each VM intermediates between the local executors and the remote KVS. All Cloudburst components are run in individual Docker [34] containers. Cloudburst uses Kubernetes [78] simply to start containers and redeploy them on failure. Cloudburst system metadata, as well as persistent application state, is stored in Anna which provides autoscaling and fault tolerance.

Function Executors

Each Cloudburst executor is an independent, long-running Python process. Schedulers (§2.3) route function invocation requests to executors. Before each invocation, the executor retrieves and deserializes the requested function and transparently resolves all KVS reference function arguments in parallel. DAG execution requests span multiple function invocations, and after each DAG function invocation, the runtime triggers downstream DAG functions. To improve performance for repeated execution (§2.2), each DAG function is deserialized and cached at one or more function executors. Each executor also publishes local metrics to the KVS, including the

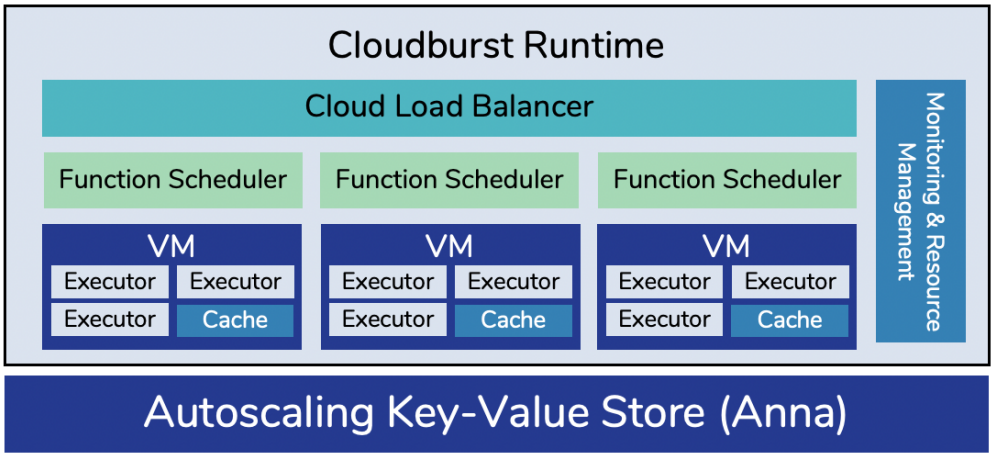


Figure 2.3: An overview of the Cloudburst architecture.

executor's cached functions, stats on its recent CPU utilization, and the execution latencies for finished requests. We explain in the following sections how this metadata is used.

Caches

To ensure that frequently-used data is locally available, every function execution VM has a local cache process, which executors contact via IPC. Executors interface with the cache, not directly with Anna; the cache issues requests to the KVS as needed. When a cache receives an update from an executor, it updates the data locally, acknowledges the request, then asynchronously sends the result to the KVS to be merged. If a cache receives a request for data that it does not have, it makes an asynchronous request to the KVS.

Cloudburst must ensure the freshness of data in caches. A naive (but correct) scheme is for the Cloudburst caches to poll the KVS for updates, or for the cache to blindly evict data after a timeout. In a typical workload where reads dominate writes, this generates unnecessary load on the KVS. Instead, each cache periodically publishes a snapshot of its cached keys to the KVS. We modified Anna to accept these cached keysets and incrementally construct an index that maps each key to the caches that store it; Anna uses this index to periodically propagate key updates to caches. Lattice encapsulation enables Anna to correctly merge conflicting key updates (§2.2). The index itself is partitioned across storage nodes following the same scheme Anna uses to partition the key space, so Anna takes the index overhead into consideration when making autoscaling decisions.

Function Schedulers

A key goal of Cloudburst's architecture is to enable low latency function scheduling. However, policy design is not a main goal of our work; Cloudburst's scheduling mechanisms allow pluggable policies to be explored in future work. In this section, we describe Cloudburst's scheduling mechanisms, illustrating their use with policy heuristics that enable us to demonstrate benefits from data locality and load balancing.

Scheduling Mechanisms. All user requests to register or invoke functions and DAGs are routed to a scheduler. Schedulers register new functions by storing them in Anna and updating a shared KVS list of registered functions. For new DAGs, the scheduler verifies that each function in the DAG exists and picks an executor on which to cache each function.

For single function execution requests, the scheduler picks an executor and forwards the request to it. DAG requests require more work: The scheduler creates a schedule by picking an executor for each DAG function—which is guaranteed to have the function stored locally—and broadcasts this schedule to all participating executors. The scheduler then triggers the first function(s) in the DAG and, if the user wants the result stored in the KVS, returns a `CloudburstFuture`.

DAG topologies are the scheduler's only persistent metadata and are stored in the KVS. Each scheduler tracks how many calls it receives per DAG and per function and stores these statistics in the KVS. Finally, each scheduler constructs a local index that tracks the set of keys stored by each cache; this is used for the scheduling policy described next.

Scheduling Policy. Our scheduling policy makes heuristic-based decisions using metadata reported by the executors, including cached key sets and executor load. We prioritize data locality when scheduling both single functions and DAGs. If the invocation's arguments have KVS references, the scheduler inspects its local cached key index and attempts to pick the executor with the most data cached locally. Otherwise, the scheduler picks an executor at random.

Hot data and functions get replicated across many executor nodes via backpressure. The few nodes initially caching hot keys will quickly become saturated with requests and will report high utilization (above 70%). The scheduler tracks this utilization to avoid overloaded nodes, picking new nodes to execute those requests. The new nodes will then fetch and cache the hot data, effectively increasing the replication factor and hence the number of options the scheduler has for the next request containing a hot key.

Monitoring and Resource Management

An autoscaling system must track system load and performance metrics to make effective policy decisions. Cloudburst uses Anna as a substrate for tracking and aggregating metrics. Each executor and scheduler independently tracks an extensible set of metrics (described above) and publishes them to the KVS. The monitoring system asynchronously aggregates these metrics from storage and uses them for its policy engine.

For each DAG, the monitoring system compares the incoming request rate to the number of requests serviced by executors. If the incoming request rate is significantly higher than the

request completion rate of the system, the monitoring engine will increase the resources allocated to that DAG function by pinning the function onto more executors. If the overall CPU utilization of the executors exceeds a threshhold (70%), then the monitoring system will add nodes to the system. Similarly, if executor utilization drops below a threshold (20%), we deallocate resources accordingly. We rely on Kubernetes to manage our clusters and efficiently scale the cluster. This simple approach exercises our monitoring mechanisms and provides adequate behavior (see §2.4).

When a new thread is allocated, it reads the relevant data and metadata (e.g., functions, DAG metadata) from the KVS. This allows Anna to serve as the source of truth for system metadata and removes concerns about efficiently scaling the system. The heuristics that we described here are based on the existing dynamics of the system (e.g., node spin up time). There is an opportunity to explore more sophisticated autoscaling mechanisms and policies which draw more heavily on understanding how workloads interact with the underlying infrastructure.

Fault Tolerance

At the storage layer, Cloudburst relies on Anna's replication scheme for k-fault tolerance. For the compute tier, we adopt the standard approach to fault tolerance taken by many FaaS platforms. If a machine fails while executing a function, the whole DAG is re-executed after a configurable timeout. The programmer is responsible for handling side-effects generated by failed programs if they are not idempotent. In the case of an explicit program error, the error is returned to the client. This approach should be familiar to users of AWS Lambda and other FaaS platforms, which provides the same guarantees.

2.4 Evaluation

We now present a detailed evaluation of Cloudburst. We first study the individual mechanisms implemented in Cloudburst (§2.4), demonstrating orders of magnitude improvement in latency relative to existing serverless infrastructure for a variety of tasks. Then, we implement and evaluate two real-world applications on Cloudburst: machine learning prediction serving and a Twitter clone (§2.4).

All experiments were run in the `us-east-1a` AWS availability zone (AZ). Schedulers were run on AWS `c5.large` EC2 VMs (2 vCPUs and 4GB RAM), and function executors were run on `c5.2xlarge` EC2 VMs (8 vCPUs and 16GB RAM); 2 vCPUs correspond to one physical core. Our function execution VMs used 4 cores—3 for Python execution and 1 for the cache. Clients were run on separate machines in the same AZ. All Redis experiments were run using AWS Elasticache, using a cluster with two shards and three replicas per shard.

Mechanisms in Cloudburst

In this section, we evaluate the primary individual mechanisms that Cloudburst enables—namely, low-latency function composition (§2.4), local cache data accesses (§2.4), direct communication

(§2.4), and responsive autoscaling (§2.4).

Function Composition

To begin, we compare Cloudburst's function composition overheads with other serverless systems, as well as a non-serverless baseline. We chose functions with minimal computation to isolate each system's overhead. The pipeline was composed of two functions: `square(increment(x:int))`. Figure 2.1 shows median and 99th percentile measured latencies across 1,000 requests run in serial from a single client.

First, we compare Cloudburst and Lambda using a "stateless" application, where we invoke one function—both bars are labelled stateless in Figure 2.1. Cloudburst stored results in Anna, as discussed in Section 2.2. We ran Cloudburst with one function executor (3 worker threads). We find that Cloudburst is about 5× faster than Lambda for this simple baseline.

For a composition of two functions—the simplest form of statefulness we support—we find that Cloudburst's latency is roughly the same as with a single function and significantly faster than all other systems measured. We first compared against `SAND` [4], a new serverless platform that achieves low-latency function composition by using a hierarchical message bus. We could not deploy `SAND` ourselves because the source code is unavailable, so we used the authors' hosted offering [113]. As a result, we could not replicate the setup we used for the other experiments, where the client runs in the same datacenter as the service[2]. To compensate for this discrepancy, we accounted for the added client-server latency by measuring the latency for an empty HTTP request to the `SAND` service. We subtracted this number from the end-to-end latency for a request to our two-function pipeline running `SAND` to estimate the in-datacenter request time for the system. In this setting, `SAND` is about an order of magnitude slower than Cloudburst both at median and at the 99th percentile.

To further validate Cloudburst, we compared against Dask, a "serverful" open-source distributed Python execution framework. We deployed Dask on AWS using the same instances used for Cloudburst and found that performance was comparable to Cloudburst's. Given Dask's relative maturity, this gives us confidence that the overheads in Cloudburst are reasonable.

We compared against four AWS implementations, three of which used AWS Lambda. Lambda (Direct) returns results directly to the user, while Lambda (S3) and Lambda (Dynamo) store results in the corresponding storage service. All Lambda implementations pass arguments using the user-facing Lambda API. The fastest implementation was Lambda (Direct) as it avoided high-latency storage, while DynamoDB added a 15ms latency penalty and S3 added 40ms. We also compared against AWS Step Functions, which constructs a DAG of operations similar to Cloudburst's and returns results directly to the user in a synchronous API call. The Step Functions implementation was 10× slower than Lambda and 82× slower than Cloudburst.

***Takeaway**: Cloudburst's function composition matches state-of-the-art Python runtime latency and outperforms commercial serverless infrastructure by 1-3 orders of magnitude.*

[2]The `SAND` developers recently released an open-source version of their system but not in time for exploration in this dissertation.

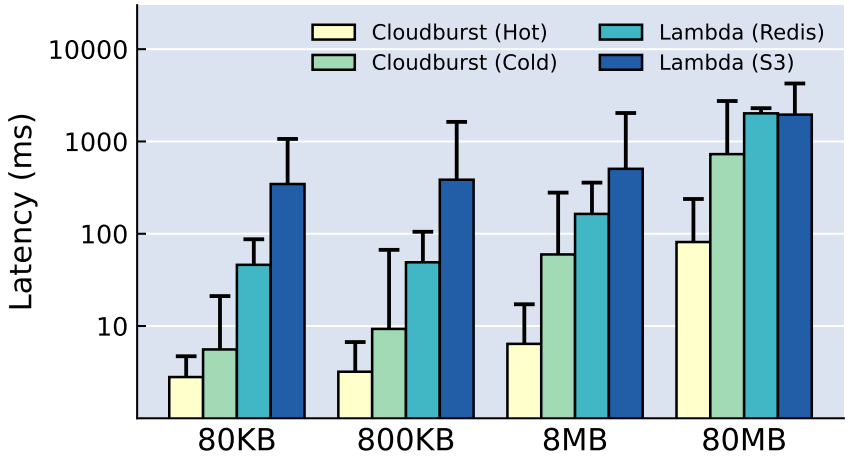


Figure 2.4: Median and 99th percentile latency to calculate the sum of elements in 10 arrays, comparing Cloudburst with caching, without caching, and AWS Lambda over AWS ElastiCache (Redis) and AWS S3. We vary array lengths from 1,000 to 1,000,000 by multiples of 10 to demonstrate the effects of increasing data retrieval costs.

Data Locality

Next, we study the performance benefit of Cloudburst's caching techniques. We chose a representative task, with large input data but light computation: our function returns the sum of all elements across 10 input arrays. We implemented two versions on AWS Lambda, which retrieved inputs from AWS ElastiCache (using Redis) and AWS S3 respectively. ElastiCache is not an autoscaling system, but we include it in our evaluation because it offers best-case latencies for data retrieval for AWS Lambda. We compare two implementations in Cloudburst. One version, Cloudburst (Hot) passes the *same* array in to every function execution, guaranteeing that every retrieval after the first is a cache hit. This achieves optimal latency, as every request after the first avoids fetching data over the network. The second, Cloudburst (Cold), creates a new set of inputs for each request; every retrieval is a cache miss, and this scenario measures worst-case latencies of fetching data from Anna. All measurements are reported across 12 clients issuing 3,000 requests each. We run Cloudburst with 7 function execution nodes.

The Cloudburst (Hot) bars in Figure 2.4 show that system's performance is consistent across the first two data sizes for cache hits, rises slightly for 8MB of data, and degrades significantly for the largest array size as computation costs begin to dominate. Cloudburst performs best at 8MB, improving over Cloudburst (Cold)'s median latency by about 10×, over Lambda on Redis' by 25×, and over Lambda on S3's by 79×.

While Lambda on S3 is the slowest configuration for smaller inputs, it is more competitive at 80MB. Here, Lambda on Redis' latencies rise significantly. Cloudburst (Cold)'s median latency is the second fastest, but its 99th percentile latency is comparable with S3's and Redis'. This

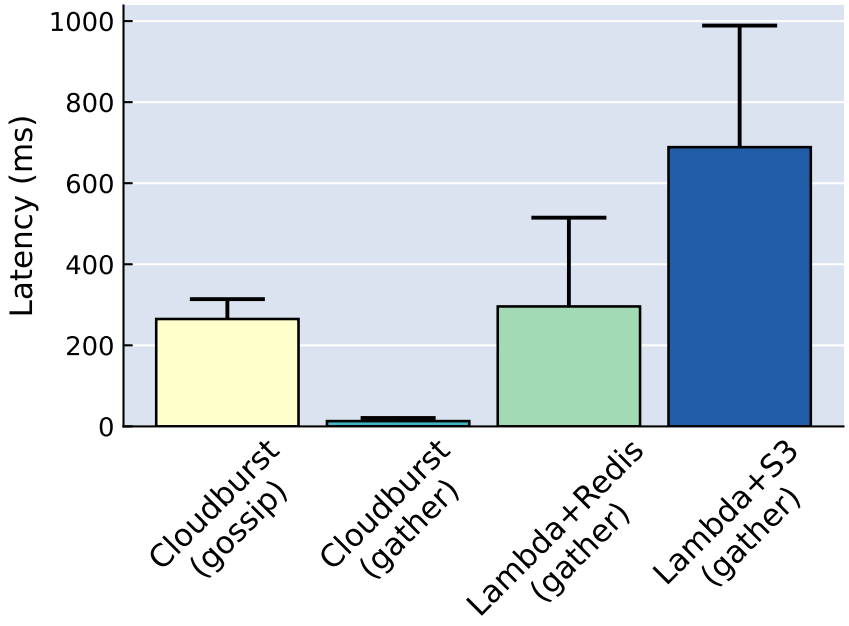


Figure 2.5: Median and 99th percentile latencies for distributed aggregation. The Cloudburst implementation uses a distributed, gossip-based aggregation technique, and the Lambda implementations share state via the respective key-value stores. Cloudburst outperforms communication through storage, even for a low-latency KVS.

validates the common wisdom that S3 is efficient for high bandwidth tasks but imposes a high latency penalty for smaller data objects. However, at this size, Cloudburst (Hot)'s median latency is still 9× faster than Cloudburst (Cold) and 24× faster than S3's.

***Takeaway**: While performance gains vary across configurations and data sizes, avoiding network roundtrips to storage services enables Cloudburst to improve performance by 1-2 orders of magnitude.*

Low-Latency Communication

Another key feature in Cloudburst is low-latency communication, which allows developers to leverage distributed systems protocols that are infeasibly slow in other serverless platforms [55].

As an illustration, we consider distributed aggregation, the simplest form of distributed statistics. Our scenario is to periodically average a floating-point performance metric across the set of functions that are running at any given time. Kempe et al. [72] developed a simple gossip-based protocol for approximate aggregation that uses random message passing among the current participants in the protocol. The algorithm is designed to provide correct answers even as the membership changes. We implemented the algorithm in 60 lines of Python and ran it over Cloudburst

with 4 executors (12 threads). We compute 1,000 rounds of aggregation with 10 actors each in sequence and measure the time until the result converges to within 5% error.

The gossip algorithm involves repeated small messages, making it highly inefficient on stateless platforms like AWS Lambda. Since AWS Lambda disables direct messaging, the gossip algorithm would be extremely slow if implemented via reads/writes from slow storage. Instead, we compare against a more natural approach for centralized storage: Each lambda function publishes its metrics to a KVS, and a predetermined leader gathers the published information and returns it to the client. We refer to this algorithm as the "gather" algorithm. Note that this algorithm, unlike [72], requires the population to be fixed in advance, and is therefore not a good fit to an autoscaling setting. But it requires less communication, so we use it as a workaround to enable the systems that forbid direct communication to compete. We implement the centralized gather protocol on Lambda over Redis for similar reasons as in § 2.4—although serverful, Redis offers best-case performance for Lambda. We also implement this algorithm over Cloudburst and Anna for reference.

Figure 2.5 shows our results. Cloudburst's gossip-based protocol is 3× faster than the gather protocol using Lambda and DynamoDB. Although we expected gather on serverful Redis to outperform Cloudburst's gossip algorithm, our measurements show that gossip on Cloudburst is actually about 10% faster than the gather algorithm on Redis at median and 40% faster at the 99th percentile. Finally, gather on Cloudburst is 22× faster than gather on Redis and 53× faster than gather on DynamoDB. There are two reasons for these discrepancies. First, Lambda has very high function invocation costs (see §2.4). Second, Redis is single-mastered and forces serialized writes, creating a queuing delay for writes.

***Takeaway**: Cloudburst's low latency communication mechanisms enable developers to build fast distributed algorithms with fine-grained communication. These algorithms can have notable performance benefits over workarounds involving even relatively fast shared storage.*

Autoscaling

Finally, we validate Cloudburst's ability to detect and respond to workload changes. The goal of any serverless system is to smoothly scale program execution in response to changes in request rate. As described in § 2.3, Cloudburst uses a heuristic policy that accounts for incoming request rates, request execution times, and executor load. We simulate a relatively computationally intensive workload with a function that sleeps for 50ms. The function reads in two keys drawn from a Zipfian distribution with coefficient of 1.0 from 1 million 8-byte keys stored in Anna, and it writes to a third key drawn from the same distribution.

The system starts with 60 executors (180 threads) and one replica of the function deployed—the remaining threads are all idle. Figure 2.6 shows our results. At time 0, 400 client threads simultaneously begin issuing requests. The jagged curve measures system throughput (requests per second), and the dotted line tracks the number of threads allocated to the function. Over the first 20 seconds, Cloudburst takes advantage of the idle resources in the system, and throughput reaches around 3,300 requests per second. At this point, the management system detects that all nodes are saturated and adds 20 EC2 instances, which takes about 2.5 minutes; this is seen in the

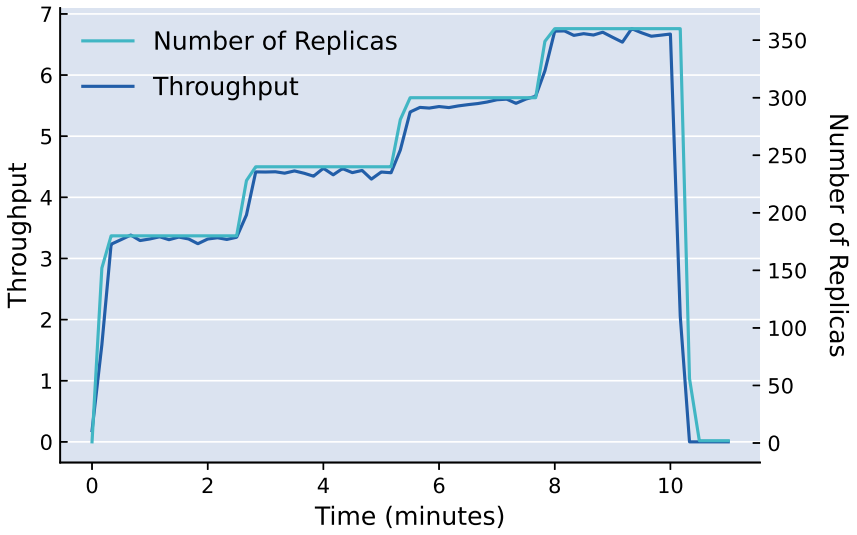


Figure 2.6: Cloudburst's responsiveness to load increases. We start with 30 executor threads and issue simultaneous requests from 60 clients and measure throughput. Cloudburst quickly detects load spikes and allocate more resources. Plateaus in the figure are the wait times for new EC2 instances to be allocated.

plateau that lasts until time 2.67. As soon as resources become available, they are allocated to our task, and throughput rises to 4.4K requests a second.

This process repeats itself twice more, with the throughput rising to 5.6K and 6.7K requests per second with each increase in resources. After 10 minutes, the clients stop issuing requests, and by time 10.33, the system has drained itself of all outstanding requests. The management system detects the sudden drop in request rate and, within 20 seconds, reduces the number of threads allocated to the sleep function from 360 to 2. Within 5 minutes, the number of EC2 instances drops from a max of 120 back to the original 60. Our current implementation is bottlenecked by the latency of spinning up EC2 instances; tools like Firecracker [36] and gVisor [47] might help improve these overheads, but we have not yet explored them.

We also measured the per-key storage overhead of the index in Anna that maps each key to the caches it is stored in. We observe small overheads even for the largest deployment (120 function execution nodes). For keys in our working set, the median index overhead is 24 bytes and the 99th percentile overhead is 1.3KB, corresponding to keys being cached at 1.6% and 93% of the function nodes, respectively. Even if all keys had the maximum overhead, the total index size would be around 1 GB for 1 million keys.

***Takeaway**: Cloudburst's mechanisms for autoscaling enable policies that can quickly detect and react to workload changes. We are mostly limited by the high cost of spinning up new EC2 instances. The policies and cost of spinning up instances can be improved in future without changing Cloudburstś architecture.*

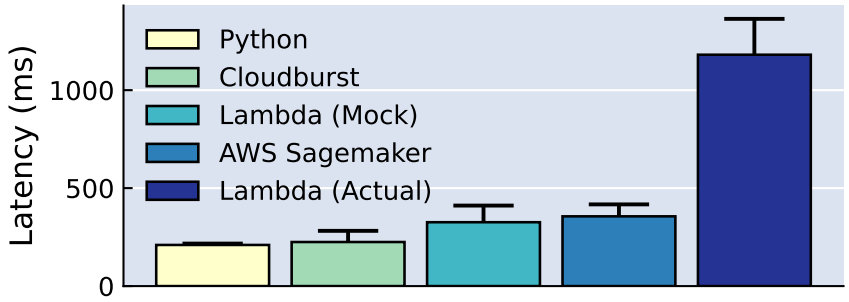


Figure 2.7: A comparison of Cloudburst against native Python, AWS Sagemaker, and AWS Lambda for serving a prediction pipeline.

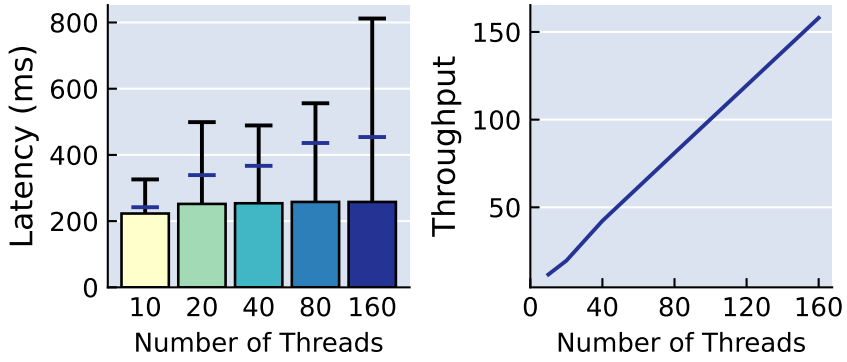


Figure 2.8: A measure of the Cloudburst's ability to scale a simple prediction serving pipeline. The blue whiskers represent 95th percentile latencies.

Case Studies

In this section, we discuss the implementation of two real-world applications on top of Cloudburst. We first consider low-latency prediction serving for machine learning models and compare Cloudburst to a purpose-built cloud offering, AWS Sagemaker. We then implement a Twitter clone called Retwis, which takes advantage of our consistency mechanisms, and we report both the effort involved in porting the application to Cloudburst as well as some initial evaluation metrics.

Prediction Serving

ML model prediction is a computationally intensive task that can benefit from elastic scaling and efficient sparse access to large amounts of state. For example, the prediction serving infrastructure at Facebook [50] needs to access per-user state with each query and respond in real time,

with strict latency constraints. Furthermore, many prediction pipelines combine multiple stages of computation—e.g., clean the input, join it with reference data, execute one or more models, and combine the results [24, 82].

We implemented a basic prediction serving pipeline on Cloudburst and compare against a fully-managed, purpose-built prediction serving framework (AWS Sagemaker) as well as AWS Lambda. We also compare against a single Python process to measure serialization and communication overheads. Lambda does not support GPUs, so all experiments are run on CPUs.

We use the MobileNet [60] image classification model implemented in Tensorflow [1] and construct a three-stage pipeline: resize an input image, execute the model, and combine features to render a prediction. Qualitatively, porting this pipeline to Cloudburst was easier than porting it to other systems. The native Python implementation was 23 lines of code (LOC). Cloudburst required adding 4 LOC to retrieve the model from Anna. AWS SageMaker required adding serialization logic (10 LOC) and a Python web-server to invoke each function (30 LOC). Finally, AWS Lambda required significant changes: managing serialization (10 LOC) and manually compressing Python dependencies to fit into Lambda's 512MB container limit[3]. Since the pipeline does not involve concurrent modification to shared state, we use the default last-writer-wins for this workload. We run Cloudburst with 3 executors (9 threads) and 6 clients issuing requests.

Figure 2.7 reports median and 99th percentile latencies. Cloudburst is only about 15ms slower than the Python baseline at the median (210ms vs. 225ms). AWS Sagemaker, ostensibly a purpose-built system, is 1.7× slower than the native Python implementation and 1.6× slower than Cloudburst. We also measure two AWS Lambda implementations. One, AWS Lambda (Actual), computes a full result for the pipeline and takes over 1.1 seconds. To better understand Lambda's performance, we isolated compute costs by removing all data movement. This result (AWS Lambda (Mock)) is much faster, suggesting that the latency penalty is incurred by the Lambda runtime passing results between functions. Nonetheless, AWS Lambda (Mock)'s median is still 44% slower than Cloudburst's median latency and only 9% faster than AWS Sagemaker.

Figure 2.8 measures throughput and latency for Cloudburst as we increase the number of worker threads from 10 to 160 by factors of two. The number of clients for each setting is set to $\lfloor \frac{workers}{3} \rfloor$ because there are three functions executed per client. We see that throughput scales linearly with the number of workers. We see a climb in median and 99th percentile latency from 10 to 20 workers due to increased potential conflicts in the scheduling heuristics. From this point on, we do not a significant change in either median or tail latency until 160 executors. For the largest deployment, only one or two executors need to be slow to significantly raise the 99th percentile latency—to validate this, we also report the 95th percentile latency in Figure 2.8, and we see that there is a minimal increase between 80 and 160 executors.

The prediction serving task here is illustrative of Cloudburst's capabilities but our exploration is not comprehensive. In Chapter 4, we study prediction serving in significantly more detail—there we develop an optimized dataflow DSL that runs on top of Cloudburst and examine a number of more complex prediction serving pipelines.

[3] AWS Lambda allows a maximum of 512MB of disk space. Tensorflow exceeds this limit, so we removed unnecessary components. We do not quantify the LOC changed here as it would artificially inflate the estimate.

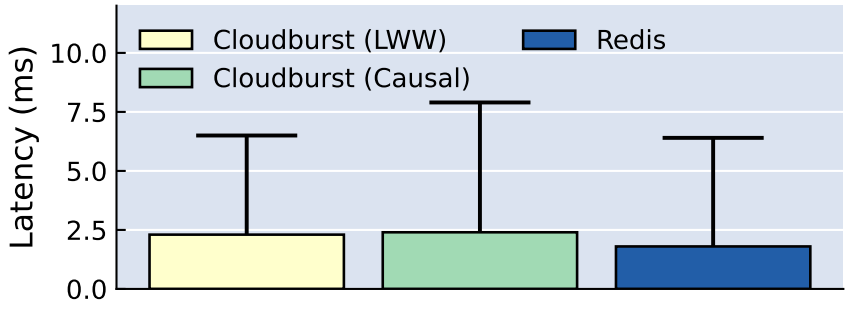


Figure 2.9: Median and 99%-ile latencies for Cloudburst in LWW and causal modes, in addition to Retwis over Redis.

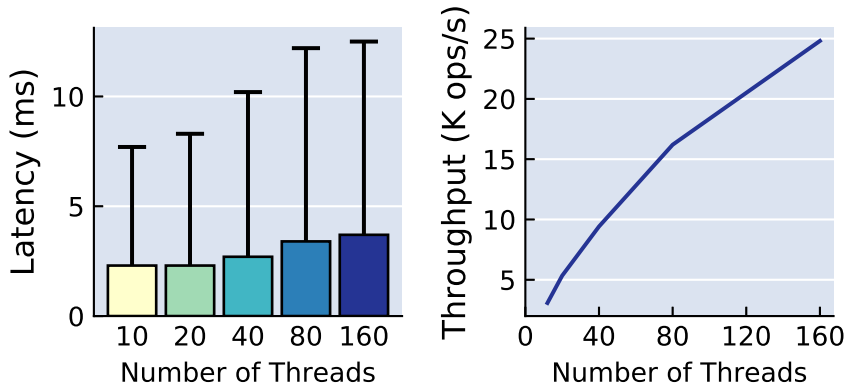


Figure 2.10: Cloudburst's ability to scale the Retwis workload up to 160 worker threads.

Takeaway: *An ML algorithm deployed in Cloudburst delivers low, predictable latency comparable to a single Python process in addition to smooth scaling, and we out-perform a purpose-built commercial service.*

Retwis

Web serving workloads are closely aligned with Cloudburst's features. For example, Twitter provisions server capacity of up to 10x the typical daily peak in order to accommodate unusual events such as elections, sporting events, or natural disasters [49]. To this end, we considered an example web serving workload. Retwis [108] is an open source Twitter clone built on Redis and is often used to evaluate distributed systems [119, 59, 147, 27, 143].

Causal consistency is a natural requirement for these kinds of workloads: It is confusing to read the response to a post in a conversational thread on Twitter (e.g., "lambda!") before you have read the post it refers to ("what comes after kappa?"). My colleague Chenggang Wu

has investigated efficient implementations of causality in the context of Cloudburst [121, 139] and demonstrated that these protocols do not significantly hinder performance while preventing many classes of common anomalies. Hence for the rest of our experiments, we measure both a causal and non-causal version of Cloudburst.

We adapted a Python Retwis implementation called `retwis-py` [109] to run on Cloudburst and compared its performance to a vanilla "serverful" deployment on Redis. We ported Retwis to our system as a set of six Cloudburst functions. The port was simple: We changed 44 lines, most of which were removing references to a global Redis variable.

We created a graph of 1000 users, each following 50 other users (zipf=1.5, a realistic skew for online social networks [93]) and prepopulated 5000 tweets, half of which were replies to other tweets. We compare Cloudburst in LWW mode, Cloudburst in causal consistency mode, and Retwis over Redis; all configurations used 6 executor threads (webservers for Retwis) and 1 KVS node. We run Cloudburst in LWW mode and Redis with 6 clients and stress test Cloudburst in causal mode by running 12 clients to increase causal conflicts. Each client issues 1000 requests—20% `PostTweet` (write) requests and 80% `GetTimeline` (read) requests.

Figure 2.9 shows our results. Median and 99th percentile latencies for LWW mode are 27% and 2% higher than Redis', respectively. This is largely due to different code paths; for Retwis over Redis, clients communicate directly with web servers, which interact with Redis. Each Cloudburst request interacts with a scheduler, a function executor, a cache, and Anna. Cloudburst's causal mode adds a modest overhead over LWW mode: 4% higher at the median and 20% higher at the tail. However, causality prevents anomalies on over 60% of requests—when a timeline returns a reply without the original tweet—compared to LWW mode.

Figure 2.10 measures throughput and latency for Cloudburst's causal mode as we increase the number of function executor threads from 10 to 160 by factors of two. For each setting, the number of clients is equal to the number of executors. From 10 threads to 160 threads, median and the 99th percentile latencies increase by about 60%. This is because increased concurrency means a higher volume of new tweets. With more new posts, each `GetTimeline` request forces the cache to query the KVS for new data with higher probability in order to ensure causality—for 160 threads, 95% of requests incurred cache misses. Nonetheless, these latencies are well within the bounds for interactive web applications [53]. Throughput grows nearly linearly as we increase the executor thread count. However, due to the increased latencies, throughput is about 30% below ideal at the largest scale.

Takeaway: *It was straightforward to adapt a standard social network application to run on Cloudburst. Performance was comparable to serverful baselines at the median and better at the tail, even when using causal consistency.*

2.5 Related Work

Serverless Execution Frameworks. In addition to commercial offerings, there are many open-source serverless platforms [57, 98, 97, 77], all of which provide standard stateless FaaS guarantees. Among platforms with new guarantees [58, 4, 88], `SAND` [4] is most similar to Cloudburst,

reducing overheads for low-latency function compositions. Cloudburst achieves better latencies (§2.4) and adds shared state and communication abstractions that enable a broader range of applications.

Recent work has explored faster, more efficient serverless platforms. SOCK [96] introduces a generalized-Zygote provisioning mechanism to cache and clone function initialization; its library loading technique could be integrated with Cloudburst. Also complementary are low-overhead sandboxing mechanisms released by cloud providers—e.g., gVisor [47] and Firecracker [36].

Other recent work has demonstrated the ability to build data-centric services on top of commodity serverless infrastructure. Starling [102] implements a scalable database query engine on AWS Lambda. Similarly, the PyWren project [66] and follow-on work [116] implemented highly parallel map tasks on Lambda, with a focus on linear algebra applications. The ExCamera project [37] enabled highly efficient video encoding on FaaS. These systems explored what is possible to build on stateless serverless infrastructure By contrast, our work explores the design of a new architecture, which raises new challenges for systems issues in distributed consistency.

Perhaps the closest work to ours is the Archipelago system [118], which also explores the design of a new serverless infrastructure. They also adopt the model of DAGs of functions, but their focus is complementary to ours. They are interested in scheduling techniques to achieve latency deadlines on a per-request basis.

Serverless IO Infrastructure. Pocket [73] is a serverless storage system that improves the efficiency of analytics (large read oriented workloads). Locus [103] provides a high-bandwidth shuffle functionality for serverless analytics. Neither of these systems offers a new serverless programming framework, nor do they consider issues of caching for latency or consistent updates.

Finally, Shredder [148] enables users to push functions into storage. This raises fundamental security concerns like multi-tenant isolation, which are the focus of the research. Shredder is currently limited to a single node and does not address autoscaling or other characteristic features of serverless computing.

Language-Level Consistency Programming languages also offer solutions to distributed consistency. One option from functional languages is to *prevent* inconsistencies by making state *immutable* [22]. A second is to constrain updates to be *deterministically mergeable*, by requiring users to write associative, commutative, and idempotent code (ACID 2.0 [54]), use special-purpose types like CRDTs [117] or DSLs for distributed computing like Bloom [23]. As a platform, Cloudburst does not prescribe a language or type system, though these approaches could be layered on top of Cloudburst.

2.6 Conclusion and Takeaways

The design and evaluation of Cloudburst demonstrate the feasibility of general-purpose stateful serverless computing. We enable autoscaling via logical disaggregation of storage and compute and achieve performant state management via physical colocation of caches with compute ser-

vices. Cloudburst shows that disaggregation and colocation are not inherently in conflict. In fact, the LDPC design pattern is key to our solution for stateful serverless computing.

Using this architecture, we are able to enable orders-of-magnitude performance improvements over commodity FaaS infrastructure for common programming patterns like function composition and accessing shared, mutable state. However, introducing stateful paradigms into serverless programming raises a number of new concerns that we have not addressed here. For example, [121] and [139] discuss our approaches to providing consistency across functions composed in Cloudburst. Other concerns raised by introducing state include cache residency in multi-tenanted settings, intelligent policies for scheduling and scaling resources, and fault tolerance. While we do not address all of these concerns, we next turn our attention to a detailed study of serverless fault tolerance.

Chapter 3

A Fault-Tolerance Shim for Serverless Computing

As we have discussed, the standard for state management in serverless infrastructure today is to manage data stored in an external storage engine like AWS S3 or Google Cloud Storage—a pattern that significantly compromises on performance. Cloudburst improves on this model with its physically colocated caches and direct communication techniques. However, both these architectures are susceptible to key concerns around fault tolerance—the need to ensure consistency and correctness in the face of unexpected system- and infrastructure-level failures.

All FaaS platforms we are aware of provide a minimal measure of fault tolerance with retries—they automatically retry functions if they fail, and clients typically will re-issue requests after a timeout. Retries ensure that functions are executed *at-least once*, and cloud providers encourage developers to write idempotent programs [79] because idempotence logically ensures *at-most once* execution. Combining retry-based at-least once execution and idempotent at-most once execution would seem to guarantee *exactly once* execution, a standard litmus test for correct fault handling in distributed systems.

This idempotence requirement is both unreasonable for programmers—as programs with side-effects are ubiquitous—and also insufficient to guarantee exactly once serverless execution. To see why idempotence is insufficient, consider a function f that writes two keys, k and l, to storage. If f fails between its writes of k and l, parallel requests might read the new version of k while reading an old version of l. Even if f is idempotent—meaning that a retry of f would write the same versions of k and l—the application has exposed a *fractional* execution in which some updates are visible and others are not. As a result, developers are forced to explicitly reason about the *correctness* of their reads in addition to the idempotence of their applications—an already difficult task.

We propose that in a retry-based fault-tolerance model, *atomicity* is necessary to solve these challenges: Either all the updates made by an application should be visible or none of them should. Returning to the example above, if f fails between the writes of k and l, an atomicity guarantee would ensure that f's version of k would not be visible to other functions without f's version of l. Atomicity thus prevents fractional executions from becoming visible.

A simple solution here is to use serializable transactions. Functions have well-defined beginnings and endings, making a transactional model a natural fit to guarantee atomicity for FaaS platforms. In reality, each logical request in an application will likely span multiple functions—as it would be difficult to pack a whole application into one function—meaning transactions should span *compositions* of functions. However, strongly consistent transactional systems have well-known scaling bottlenecks [19, 21] that make them ill-suited for serverless settings, which must accommodate unbounded numbers of client arrivals.

Instead, we turn to *read atomicity*, a coordination-free isolation level introduced in [9]. Intuitively, read atomic isolation guarantees that transactions do not leak partial side effects to each other, which is the requirement described above. However, the original implementation of read atomic isolation—termed RAMP in [9]—makes key limiting assumptions that are unreasonable in our setting. First, their storage system forbids replication, which limits locality and scalability of reads. Second, they assume that every transaction's read and write sets are predeclared, which is unreasonable for interactive applications that make dynamic decisions.

A Fault-Tolerance Shim

Our goal is to provide fault tolerance in the context of widely-used FaaS platforms and storage systems. To that end, we present AFT, an Atomic Fault Tolerance shim for serverless computing. AFT provides fault-tolerance for FaaS applications by interposing between a FaaS platform (e.g., AWS Lambda, Azure Functions, Cloudburst) and a cloud storage engine (e.g., AWS S3, Google Cloud BigTable, Anna). Updates written to storage during a single logical request—which may span multiple functions—are buffered by AFT and atomically committed at request end. AFT enforces the read atomic isolation guarantee, ensuring that transactions never see partial side effects and only read data from sets of atomic transactions. Given the unsuitability of the original RAMP protocols for serverless infrastructure, we develop new protocols to support read atomic isolation over shared storage backends. Importantly, AFT maintains a high measure of flexibility by only relying on the storage engine for durability.

Our contributions are the following:

- The design of AFT, a low-overhead, transparent fault tolerance shim for serverless applications that is flexible enough to work with many combinations of commodity compute platforms and storage engines.
- A new set of protocols to guarantee read atomic isolation for shared, replicated storage systems.
- A garbage collection scheme for our protocols that significantly reduces the storage overheads of read atomic isolation.
- A detailed evaluation of AFT, demonstrating that it imposes low latency penalties and scales smoothly to hundreds of clients and thousands of requests per second, while preventing consistency anomalies.

3.1 Background and Motivation

In this section, we describe prior work on read atomicity and its guarantees (§3.1), and we explain the technical challenges in providing read atomicity for serverless applications (§3.1).

Read Atomic Isolation

The read atomic isolation guarantee, introduced by Bailis et al. in [9], aims to ensure that transactions do not view partial effects of other transactions. Bailis et al. provide the following definition: "A system provides Read Atomic isolation (RA) if it prevents fractured reads anomalies and also prevents transactions from reading uncommitted, aborted, or intermediate data." We refer to reads of "uncommitted, aborted, or intermediate" data as *dirty reads*. A *fractured read* happens when, "... transaction T_i writes versions x_m and y_n (in any order, with x possibly but not necessarily equal to y), [and] T_j [later] reads version x_m and version y_k, and $k < n$." The read atomic isolation level is a good fit for the serverless setting because it enforces atomic visibility of updates without strong consistency or coordination.

Challenges and Motivation

The protocols introduced in [9] make two assumptions that are unreasonable for serverless applications: pre-declared read/write sets and a linearizable, unreplicated, and sharded storage backend. Relaxing these assumptions enables us to bring read atomic isolation to the serverless setting but raises new challenges around consistency and visibility.

Read and Write Sets. [9] requires that each transaction declares its read and write sets in advance, in order to correctly ensure that the transaction's operations adhere to the definition of read atomicity above. We relax this assumption, allowing requests to issue reads and writes without restriction, and we develop new read atomic protocols that allow us to dynamically construct atomic read sets (§3.2). The drawback of this flexibility is that clients may be forced to read data that is more stale than they would have under the original RAMP protocol, and in rare cases, a request may be forced to abort because there are no valid key versions for it to read. We explain this tradeoff in more detail in §3.2.

Shared Storage Backends. The RAMP protocols assume linearizable, unreplicated, and shared-nothing storage shards, each of which operate independently. Each shard is the sole "source of truth" for the set of keys it stores, which can lead to scaling challenges in skewed workloads. This design is incompatible with standard cloud applications, where workloads can often be highly skewed [128, 13] and require strong durability. To that end, AFT offers read atomic isolation in a shim layer over a durable shared storage backend without requiring the storage layer to provide consistency guarantees and partitioning. We avoid the scaling pitfalls of RAMP by letting all nodes commit data for all keys. This requires carefully designing commit protocols for individual nodes (§3.2) and ensuring that nodes are aware of transactions committed by their peers (§3.3).

Our coordination-free and fungible node design leads to a potential ballooning of both data and metadata, which we address in §3.4.

Serverless Applications. Serverless applications are typically compositions of multiple functions. We model each request as a linear composition of one or more functions executing on a FaaS platform. AFT must ensure atomic reads and writes across all functions in the composition, each of which might be executed on a different machine—this informs our design. In effect, we must support a "distributed" client session as the transaction moves across functions, and we must ensure that retries upon failure guarantee idempotence. We discuss these issues in more detail in §3.2.

3.2 Achieving Atomicity

In this section, we describe how AFT achieves atomic writes and reads at a single node. We first discuss AFT's API, architecture, and components (§3.2). We then formally state the guarantees we make (§3.2); we describe AFT's protocol for each of the guarantees in §3.2-§3.2. In §3.3, we discuss howAFT operates in distributed settings.

Architecture and API

AFT offers transactional key-value store API, shown in Table 3.1. We refer to each logical request (which might span multiple FaaS functions) as a transaction. A new transaction begins when a client calls `StartTransaction`, and the transaction is assigned a globally-unique UUID. At commit time, each transaction is given a commit timestamp based on the machine's local system clock. This $\langle timestamp, uuid \rangle$ pair is the transaction's ID. AFT uses each transaction's ID to ensure that its updates are only persisted once, assuring idempotence in the face of retries—as discussed earlier, guaranteeing idempotence and atomicity results in exactly once execution semantics. We do not rely on clock synchronization for the correctness of our protocols, and we only use system time to ensure relative freshness of reads. As a result, we need not coordinate to ensure timestamp uniqueness—ties are broken by lexicographically comparing transactions' UUIDs.

Each transaction sends all operations to a single AFT node. Within the bounds of a transaction, clients interact with AFT like a regular key-value store, by calling `Get(txid, key)` and `put(txid, key, value)`. When a client calls `CommitTransaction`, AFT assigns a commit timestamp, persists all of the transaction's updates, and only acknowledges the request once the updates are durable. At any point during its execution, if a client calls `AbortTransaction`, none of its updates are made visible, and the data is deleted.

Figure 3.1 shows a high-level overview of the system architecture. Each AFT node is composed of a transaction manager, an atomic write buffer, and a local metadata cache. The atomic write buffer gathers each transaction's writes and is responsible for atomically persisting them at commit time. The transaction manager tracks which key versions each transaction has read thus far and is responsible for enforcing the read atomicity guarantee described later in this section. AFT

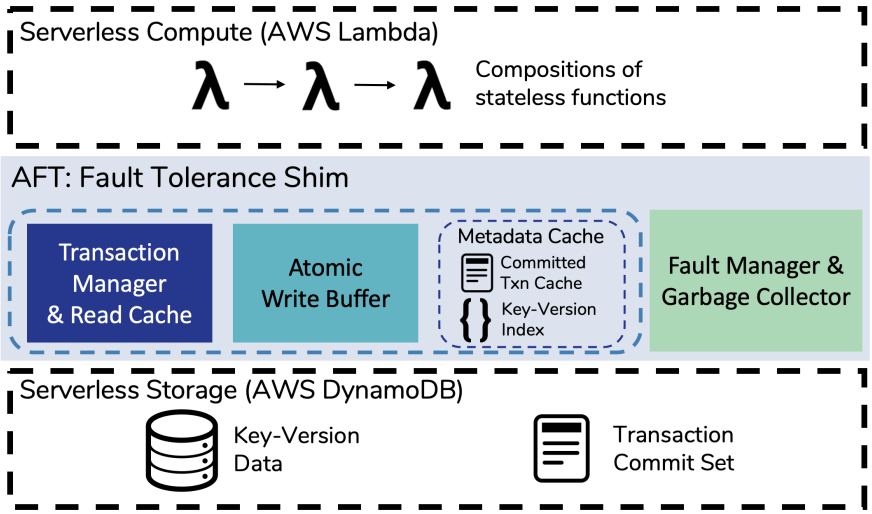


Figure 3.1: A high-level overview of the AFT shim in context.

API	Description
`StartTransaction()->txid`	Begins a new transaction and returns a transaction ID.
`Get(txid, key)->value`	Retrieves `key` in the context of the transaction keyed by `txid`.
`Put(txid, key, value)`	Performs an update for transaction `txid`.
`AbortTransaction(txid)`	Aborts transaction `txid` and discards any updates made by it.
`CommitTransaction(txid)`	Commits transcation `txid` and persists its updates; only acknowledges after all data and metadata has been persisted.

Table 3.1: AFT offers a simple transactional key-value store API. All `get` and `put` operations are keyed by the ID transaction within which they are executing.

maintains a Transaction Commit Set storage, which holds the ID of each committed transaction and its corresponding write set. We defer discussion of fault management and garbage collection to §3.3 and §3.4, respectively.

Algorithm 1 (§3.2) requires access to the list of recently committed transactions. To avoid metadata fetches on each read, AFT caches the IDs of recently committed transactions and locally maintains an index that maps from each key to the recently created versions of that key. When an AFT node starts (e.g., after recovering from failure), it bootstraps itself by reading the latest records in the Transaction Commit Set to warm its metadata cache. We discuss how the metadata cache is pruned by the garbage collection processes in §3.4.

In addition to a metadata cache, AFT has a data cache, which stores values for a subset of the key versions in the metadata cache. The data cache improves performance by avoiding storage

lookups for frequently accessed versions; we measure its effects in §3.5.

AFT make no assumptions about the serverless compute layer it is interacting with—it simply responds to the API calls described in Table 3.1. The only assumption AFT makes about the underlying storage engine is that updates are durable once acknowledged. AFT does not rely on the storage engine to enforce any consistency guarantees or to immediately make keys visible after they are written.

Definitions

A transaction T's ID is denoted by its subscript: transaction T_i has ID i. We say that T_i is newer than T_j ($T_i > T_j$) if $i > j$—we describe how IDs are compared in §3.2. A key without a subscript, k, refers to any version of that key; k_i is the version of key k written by transaction T_i. Each key has a `NULL` version, followed by many non-null versions. Importantly, key versions are hidden from users: Clients requests reads and writes of keys, and AFT automatically determines which versions are compatible with each request.

We define a transaction T_i's write set, $T_i.writeset$, as the set of key versions written by this transaction. Similarly, each key version has a cowritten set: If T_i writes k_i, $k_i.cowritten$ is the same as $T_i.writeset$.

As described in §3.1, read atomic isolation requires preventing *dirty reads* and preventing *fractured reads*. AFT ensures two simple properties in addition to read atomicity that most programmers are familiar with: *read-your-writes* and *repeatable read*. We describe these four properties briefly.

Dirty Reads. To prevent dirty reads, AFT guarantees that if transaction T_i reads key version k_j written by transaction T_j, T_j must have successfully committed.

Fractured Reads. To avoid fractured reads, each transaction's read set must form an *Atomic Readset* defined below:

Definition 1 (Atomic Readset)**.** *Let R be a set of key versions. R is an* Atomic Readset *if* $\forall k_i \in R, \forall l_i \in k_i.cowritten, l_j \in R \Rightarrow j \geq i$.

In other words, for each key version k_i read by transaction T_j, if T_j also reads a version of key l that was cowritten with k_i (i.e., T_j reads l_i), we return a version of l that is no older than l_i. Consider a scenario where there are two committed transactions in storage, $T_1 : \{l_1\}$ and $T_2 : \{k_2, l_2\}$, where the set following the colon denotes the set of key versions written by the corresponding transaction. If a new transaction, T_n first requests k and reads k_2, a subsequent read of l must return a version of l that is at least as new as l_2 (i.e., l_2 or some newer version). If T_n were to read l_1, that would violate Definition 1 because l is in the cowritten set of k_2, and $l_1 < l_2$.

Read Your Writes. Guaranteeing read your writes requires ensuring that a transaction reads the most recent version of a key it previously wrote. Say a transaction T_i writes key version k_{i_1}; a following read of k should return k_{i_1}. If T_i then proceeds to write k_{i_2}, future reads will return k_{i_2}.

Repeatable Read. Repeatable read means that a transaction should view the same key version if it requests the same key repeatedly: If T_i reads version k_j then later requests a read of k again, it should read k_j, *unless* it wrote its own version of k, k_i, in the interim. Note that this means the read-your-writes guarantee will be enforced at the expense of repeatable reads.

Preventing Dirty Reads

AFT implements atomic updates via a simple write-ordering protocol. AFT's Atomic Write Buffer sequesters all the updates for each transaction. When `CommitTransaction` is called, AFT first writes the transaction's updates to storage. Once all updates have successfully been persisted, AFT writes the transaction's write set, timestamp, and UUID to the Transaction Commit Set in storage. Only after the Commit Set is updated does AFT acknowledge the transaction as committed to the client and make the transaction's data visible to other requests. If a client calls `AbortTransaction`, its updates are simply deleted from the Atomic Write Buffer, and no state is persisted in the storage engine. This protocol is carefully ordered to ensure that dirty data is never visible.

Crucially, to avoid coordination, AFT does not overwrite keys in place: Each key version is mapped to a unique storage key, determined by its transaction's ID. This naturally increases AFT's storage and metadata footprints; we return to this point in §3.4, where we discuss garbage collection.

AFT prevents dirty reads by only making a transaction's updates visible to other transactions *after* the correct metadata has been persisted. As we will see in Section 3.2, AFT only allows reads from transactions that have already committed by consulting the local committed transaction cache (see Section 3.2).

For long-running transactions with large update sets, users might worry that forcing all updates to a single Atomic Write Buffer means that the update set must fit in memory on a single machine. However, when an Atomic Write Buffer is saturated, it can proactively write intermediary data to storage. The protocols described in this section guarantee that this data will not be made visible until the corresponding commit record is persisted. If a AFT node fails after such a write happens but before a commit record is written, the intermediary writes must be garbage collected; we discuss this in Section 3.4.

Atomic Fault Tolerance

This write-ordering protocol is sufficient to guarantee fault tolerance in the face of application and AFT failures. When an application function fails, none of its updates will be persisted, and its transaction will be aborted after a timeout. If the function is retried, it can use the same transaction ID to continue the transaction, or it can begin a new transaction.

There are two cases for AFT failures. If a transaction had not finished when an AFT node fails, we consider its updates lost, and clients must redo the whole transaction. If the transaction had finished and called `CommitTransaction`, we will read its commit metadata (if it exists) during the AFT startup process (§3.2). If we find commit metadata, our write-ordering protocol ensures

that the transaction's key versions are persisted; we can declare the transaction successful. If no commit record is found, the client must retry.

Preventing Fractured Reads

In this section, we introduce AFT's atomic read protocol. We guarantee that after every consecutive read, the set of key versions read thus far forms an Atomic Readset (Definition 1). Unlike in [9], we do not require that applications declare their read sets in advance. We enable this flexibility via versioning—the client may read older versions because prior reads were from an earlier timestamp.

AFT uses the metadata mentioned in Section 3.2 to ensure that reads are correct. As discussed in Section 3.2, a transaction's updates are made visible only after its commit metadata is written to storage. Algorithm 1 uses the local cache of committed transaction metadata and recent versions of keys to ensure that reads are only issued from already-committed transactions.

Algorithm 1 shows our protocol for guaranteeing atomic reads. If a client requests key k, two constraints limit the versions it can read. First, if k_i was cowritten with an earlier read l_i, we must return a version, k_j, that is at least as new as k_i ($j \geq i$). Second, if we previously read some version l_i, the k_j return cannot be cowritten with $l_j \mid j > i$; if it was, then we should have returned l_j to the client earlier, and this read set would violate read atomicity.

Theorem 1. *Given k and R, the R_{new} produced by Algorithm 1 is an Atomic Readset, as defined in Definition 1.*

Proof. We prove by induction on the size of R.

Base Case. Before the first read is issued, R is empty and is trivially an Atomic Readset. After executing Algorithm 1 for the first read, R_{new} contains a single key, k_{target}, so Theorem 1 holds; R_{new} is an Atomic Readset.

Inductive hypothesis: Let R be the Atomic Readset up to this point in the transaction, and let k_{target} be the key version returned by Algorithm 1. We show that R_{new} is also an Atomic Readset. From the constraints described above, we must show that (1) $\forall l_i \in R, k_i \in l_i.cowritten \Rightarrow target \geq i$, and (2) $\forall l_{target} \in k_{target}.cowritten, l_i \in R \Rightarrow i \geq target$.

Lines 3-5 of Algorithm 1 ensure (1) by construction, as the lower bound of $target$ is computed by selecting the largest transaction ID in R that modified k—we never return a version that is older than $lower$. We iterate through versions of k that are newer than $lower$, starting with the most recent version first. Lines 13-23 check if each version satisfies case (2) We iterate through all the cowritten keys of each candidate version. If any cowritten key is in R, we declare the candidate version valid if and only if the cowritten key's version is *not newer* than the version in R. If there are no valid versions, we return `NULL`.

In summary, k_j satisfies case (1) because we consider no versions older than $lower$, and it is satisfies case (2) because we discard versions of k that conflict with previous reads. □

Algorithm 1 AtomicRead: For a key k, return a key version k_j such that the read set R combined with k_j does not violate Definition 1.

Input: $k, R, WriteBuffer, storage, KeyVersionIndex$

1: $lower \coloneqq 0$ // Transaction ID lower bound.
2: // Lines 3-5 check case (1) of the inductive proof.
3: **for** $l_i \in R$ **do**
4: **if** $k \in l_i.cowritten$ **then** // We must read k_j such that $j \geq i$.
5: $lower = max(lower, i)$
6: // Get the latest version of k that we are aware of.
7: $latest \coloneqq max(KeyVersionIndex[k])$
8: **if** $latest == None \wedge lower == 0$ **then**
9: return `NULL`
10: $target \coloneqq None$ // The version of k we want to read.
11: $candidateVersions \coloneqq sort(filter(KeyVersionIndex[k], kv.tid \geq lower))$ // Get all versions of k at least as new as $lower$.
12: // Loop through versions of k in reverse timestamp order — lines 13-23 check case (2) of the inductive proof.
13: **for** $t \in candidateVersions.reverse()$ **do**
14: $valid \coloneqq True$
15: **for** $l_i \in k_t.cowritten$ **do**
16: **if** $l_j \in R \wedge j < t$ **then**
17: $valid \coloneqq False$
18: break
19: **if** $valid$ **then**
20: $target = t$ // The most recent valid version.
21: break
22: **if** $target == None$ **then**
23: return `NULL`
24: $R_{new} \coloneqq R \cup \{k_{target}\}$
25: return $storage.get(k_{target}), R_{new}$

Other Guarantees

AFT makes two other guarantees, defined in §3.2: read-your-writes and repeatable reads. We briefly discuss each.

Read-Your-Writes When a transaction requests a key that is currently stored in its own write set, we simply return that data immediately; this guarantees the read-your-writes property described above. Note that this process operates outside of the scope of Algorithm 1. Key versions that are stored in the Atomic Write Buffer are not yet assigned a commit timestamp, so they cannot correctly participate in the algorithm. We make this exception because read-your-writes is a useful guarantee for programmers.

Repeatable Read. Ensuring repeatable read is a corollary of Theorem 1. As mentioned in 3.2,

read-your-writes takes precedence over repeatable read; therefore, Corollary 1.1 only applies in transactions without intervening writes of k.

Corollary 1.1. *Let R be an Atomic Readset for a transaction T_i, and let k_j, R_{new} be the results of Algorithm 1. $k \notin T_i.writeset \wedge k_i \in R \Rightarrow k_i == k_j$.*

Proof. Given R_{new} and k_j as the results of Algorithm 1, Theorem 1 tells us that $\forall l_i \in R, k_i \in l_i.cowritten \Rightarrow j \geq i$. Since $k_i \in R$ (and trivially, $k_i \in k_i.cowritten$), we know that $j \geq i$ for the k_j returned by Algorithm 1.

Theorem 1 further guarantees that $\forall l_j \in k_j.cowritten, l_i \in R \Rightarrow i \geq j$. Once more, since $k_j \in k_j.cowritten$ and $k_i \in R$, we know that $i \geq j$. Combining these two cases, $i == j$, meaning Algorithm 1 guarantees repeatable read. □

Staleness

Our protocol is more flexible than RAMP because it allows interactively defined read sets. However, it increases the potential for reading stale data because of restriction (2) in the proof of Theorem 1. If T_r reads l_i, it cannot later read k_j if $l_j \in k_j.cowritten, j > i$, because this violates the Definition 1. The RAMP protocol in [9] avoided this pitfall with pre-declared read sets—l and k are read at the same time. If the storage engine returns the same l_i and k_j as above, [9] repairs this mismatch by forcing a read of l_j. However, in our system, if k_i is exposed to the client without knowledge of the later read of l, we cannot repair the mismatch.

In extreme cases, this can cause transaction aborts. Continuing our example, if T_r reads l_i and then requests k, Algorithm 1 will determine that k_j is not a valid version for T_r. If k_j is the only version of k, we return `NULL` because $\{l_i, k_j\}$ does not form Atomic Readset. Note that this equivalent to reading from a fixed database snapshot—if no versions of l exist as of time i, a client would read `NULL`, abort, and retry.

3.3 Scaling AFT

A key requirement for any serverless system is the ability to scale to hundreds or thousands of parallel clients. The protocols described in §3.2 ensure read atomicity for a single AFT replica. In this section, we discuss how we scale AFT while maintaining distributed consistency.

As discussed earlier, we explicitly chose to avoid coordination-based techniques, as they have well-known issues with performance and scalability [19, 21]. AFT nodes do not coordinate on the critical path of each transactions. The write protocol described in §3.2 allows each transaction to write to separate storage locations, ensuring that different nodes do not accidentally overwrite each others' updates.

Allowing each node to commit transactions without coordination improves performance but requires ensuring nodes are aware of transactions committed by other nodes. Each machine has a background thread that periodically—every 1 second—gathers all transactions committed recently on this node and broadcasts them to all other nodes. This thread also listens for messages

Algorithm 2 IsTransactionSuperseded: Check whether transaction T_i has been superseded—if there is a newer version of every key version written by T_i.

Input: $T_i, keyVersionIndex$

```
for k_i ∈ T_i.writeset do
    latest := k.latest_tid()
    if latest == i then
        return False
return True
```

from other replicas. When it receives a new commit set, it adds all those transactions to its local Commit Set Cache and updates its key version index.

In a distributed setting where we might be processing thousands of transactions a second, the cost of communicating this metadata can be extremely high. We now describe an optimization that reduces AFT's communication overheads. §3.3 discusses fault-tolerance for distributed deployments. In §3.3, we describe AFT's deployment model.

Pruning Commit Sets

To avoid communicating unnecessary metadata, we proactively prune the set of transactions that each node multicasts. In particular, any transaction that is locally *superseded* does not need to be broadcast. A transaction T_i is locally superseded if, $\forall k_i \in T_i.writeset, \exists k_j \mid j > i$—that is, for every key written by T_i, there are committed versions of those keys written by transactions newer than T_i. For highly contended workloads in particular—where the same data is likely to be written often—this significantly reduces the volume of metadata that must be communicated between replicas.

Algorithm 2 how we determine if a transaction is superseded. Each node's background multicast protocol checks whether a recently committed transaction is superseded before sending it to other replicas. If the transaction is superseded, it is omitted entirely from the multicast message. Similarly, for each transaction received via multicast, the receiving node checks to see if it is superseded by transactions stored locally; if it is, we do not merge it into our the metadata cache. Note that we can safely make decisions about transaction supersedence without coordination because each transaction receives monotonically increasing sets of keys; once a transaction is superseded on a particular node, that node can safely delete the transaction metadata.

Fault Tolerance

We now turn to guaranteeing fault tolerance for the distributed protocols described in this section; we consider both safety and liveness. To guarantee safety, we rely on our write-ordering protocol, which ensures that each node does not persist dirty data and that commits are correctly acknowledged. To guarantee liveness, we must further ensure that if a replica commits a transaction, acknowledges to a client, and fails before broadcasting the commit, other replicas are still made aware of the committed data. If other nodes do not know about the committed transaction,

the new data will be in storage but will never be visible to clients, which is equivalent to not having committed.

To this end, distributed deployments of AFT have a fault manager (see Figure 3.1) that lives outside of the request critical path. The fault manager receives every node's committed transaction set without our pruning optimization applied. It periodically scans the Transaction Commit Set in storage and checks for persisted commit records that it has not received via broadcast. It notifies all AFT nodes of any such transactions, ensuring that data is never lost once it has been committed. Thus, if a AFT node acknowledges a commit to the client but fails before broadcasting it to other AFT nodes, the fault manager will read that transaction's commit record and ensure that other nodes are aware of the transaction's updates. The fault manager is itself stateless and fault-tolerant: If it fails, it can simply scan the Commit Set again.

Deployment and Autoscaling

AFT is deployed using Kubernetes [78], a cluster management tool that deploys applications running in Docker containers [34]. Each AFT replica as well as the fault manager run in separate Docker containers and on separate machines. The fault manager described above is responsible for detecting failed nodes and configuring their replacements.

An important part of any serverless system is the ability to autoscale in response to workload changes. Our protocols for achieving distributed fault tolerance and read atomicity do not require coordination, and distributed deployments of AFT scale with low overhead, which we will demonstrate in §3.5. The second aspect of this challenge is making accurate scaling decisions without user intervention. This is a question of designing an efficient policy for choosing to add and remove nodes from the system. That policy is pluggable in AFT out of scope for our work.

AFT and Cloudburst

While we have primarily discussed the implementation of AFT in the context of AWS Lambda and other commodity cloud services thus far, the AFT architecture is also well-suited to working within the Cloudburst stack. In this section, we describe how we modify the protocols described in this section and Section 3.2 to work with Cloudburst.

As we discussed in Section 2.3, Cloudburst deploys a cache on each node in the function execution layer that intermediates on all KVS accesses. Given Cloudburst's on-machine cache design, it makes sense to deploy AFT as a substitute for these caches rather than as a completely separate shim layer. We model each Cloudburst DAG as a single AFT transaction.

In addition to some engineering tasks (e.g., adding an IPC interface, implementing an Anna backend), porting AFT to Cloudburst required support for distributed transactions. This is because Cloudburst DAGs are scheduled across multiple nodes for reasons of load balancing and resource allocation. We would like each node to interact with the AFT cache instance running on the same node, which means the metadata for a single transaction might be spread across multiple AFT instances. Note that, as before, we only support *linear* chains of functions.

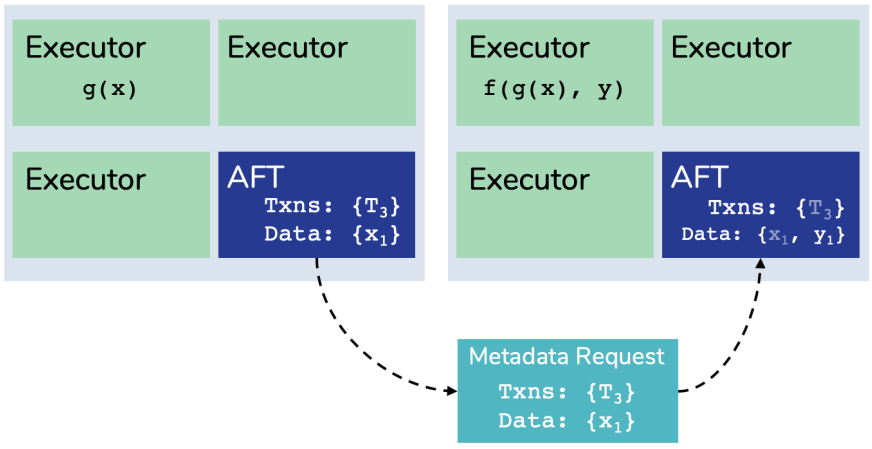


Figure 3.2: An illustration of the data and metadata movement between AFT caches deployed on separate Cloudburst machines.

Metadata Transfer. To correctly enforce AFT's guarantees, whenever a transaction interacts with a new cache, that cache must have the correct metadata about the transaction's reads and writes. This metadata is used to enforce both the read atomicity and the read your writes guarantees. We modified the Cloudburst executors to notify AFT when a new function starts, including the ID of the enclosing transaction. As illustrated in Figure 3.2, if this AFT instance has not interacted with this transaction before, it queries the AFT cache upstream from itself in the DAG's execution for the transaction's metadata.

As we have already discussed, previously read keys affect which key versions are valid for future reads, so each cache must have the metadata for all keys read in the transaction (see Section 3.2). When a client requests a key that was written earlier in the transaction, that key cannot be read from storage because the transaction has not yet committed. Instead, the cache detects the key was previously written using the transaction's metadata and queries the upstream cache(s) to find and return the correct key version.

Distributed Commit. At commit time, AFT executes a version of two-phrase commit where the last function in the DAG acts as the coordinator. After the last function has finished executing, the Cloudburst executor calls `commit` on the DAG's transaction. The AFT cache on the last function's executor node broadcasts an asynchronous commit message to all upstream caches in the DAG's execution, performs its own writes (if any), and waits for responses from all upstream caches. Only after all other caches have responded is the commit record written. This is the distributed equivalent of the commit protocol described in the previous section.

3.4 Garbage Collection

In the protocols described thus far, there are two kinds of data that would grow monotonically if left unchecked. The first is transaction commit metadata—the list of all transactions committed by AFT thus far. The second is set of key versions. As described in §3.2, each transaction's updates are written to unique keys in the storage engine and are never overwritten. Over time, the overheads incurred from these sources can grow prohibitive, in terms of both performance and cost. In §3.4, we describe how each node clears its local metadata cache, and in §3.4, we describe how we reduce storage overheads by deleting old data globally.

Local Metadata Garbage Collection

In §3.3, we introduced Algorithm 2, which enables each node to locally determine whether a particular transaction has been superseded because there are newer versions of all keys the transaction wrote. To locally garbage collect transaction metadata, a background garbage collection (GC) process periodically sweeps through all committed transactions in the metadata cache. For each transaction, the background process executes Algorithm 2 to check if it is superseded and ensures that no currently-executing transactions have read from that transaction's write set.

If both conditions are met, we remove that transaction from the Commit Set Cache and evict any cached data from that transaction. This significantly reduces our metadata overheads because old transactions are frequently discarded as new transactions arrive. While supersedence can safely be decided locally, individual nodes *cannot* make decisions about whether to delete key versions because a transaction running at another node might read the superseded transaction's writes. As a result, we next describe a global protocol that communicates with all replicas to garbage collect key versions. Each individual replica maintains a list of all locally deleted transaction metadata to aid in the global protocol.

Global Data Garbage Collection

The fault manager discussed in §3.3 also serves as a global garbage collector (GC). We combine these processes because the fault manager already receives commit broadcasts from AFT nodes, which allows us to reduce communication costs. The global GC process executes Algorithm 2 to determine which transactions have been superseded. It generates a list of transactions it considers superseded and asks all nodes if they have locally deleted those transactions. If all nodes have deleted a transaction's metadata, we can be assured that no running transactions will attempt to read the deleted items. Nodes respond with the transactions they have deleted, and when the GC process receives acknowledgements from all nodes[1], it deletes the corresponding transaction's writes and commit metadata. We allocate separate cores for the data deletion process, which allows us to batch expensive delete operations separate from the GC process.

[1]Note that knowing *all* nodes present in the system is a traditional distributed systems membership problem, which requires coordination; we currently rely on Kubernetes to provide this information.

Limitation: Missing Versions

There is one key shortcoming to this protocol. Recall from Section 3.2 that AFT returns `NULL` if there are no key versions in the valid timestamp range from Algorithm 1. This problem can be exacerbated by our garbage collection protocol.

As mentioned earlier, our local metadata GC protocol will not delete a transaction T_i if a running transaction, T_j, has read from T_i's write set. However, since we do not know each running transaction's full read set, we might delete data that would be required by a running transaction in the future. Consider the following transactions and write sets: $T_a : \{k_a\}, T_b : \{l_b\}, T_c : \{k_c, l_c\}, a < b < c$. Say a transaction T_r first reads k_a then requests key l. The GC process will not delete T_a because T_r has read from it and is uncommitted. However, it might delete T_b if there are no restrictions on it; when T_r attempts to read l, it will find no valid versions since l_c is invalid, and Algorithm 1 will return `NULL`.

Long-running transactions accessing frequently updated keys are particularly susceptible to this pitfall. These transactions might be forced to repeatedly retry because of missing versions, significantly increasing latencies. In practice, we mitigate this issue by garbage collecting the oldest transactions first. In our evaluation, we did not encounter valid versions of keys had been deleted by the GC process.

3.5 Evaluation

In this section, we present a detailed evaluation of AFT. A key design goal for AFT is the flexibility to run on a variety of cloud storage backends, so we implemented AFT over AWS S3, a large scale object storage system, and AWS DynamoDB, a cloud-native key-value store. We also run AFT over Redis (deployed via AWS ElastiCache) because it offers best-case performance for a memory-speed KVS despite the fact that it is not an autoscaling storage engine. All experiments use Redis in cluster mode with 2 shards and 2 nodes per shard, unless stated otherwise.

First, we measure AFT's performance overheads and consistency benefits by comparing it to a variety of other serverless storage architectures (§3.5). We then evaluate aspects of the system design: read caching (§3.5), scalability (§3.5), and garbage collection (§3.5). Finally, we measure AFT's ability to tolerate and recover quickly from faults (§3.5).

AFT is implemented in just over 2,500 lines of Go and 700 lines of Python and runs on top of Kubernetes [78]. The majority of the AFT protocols described here are implemented in Go. We use Python to spin up and configure Kubernetes clusters as well as to detect node failures in Kubernetes and to reconfigure the cluster in such an event. We use a simple stateless load balancer implemented in Go to route requests to AFT nodes in a round-robin fashion. All experiments were run in the `us-east-1a` AWS availability zone (AZ). Each AFT node ran on a `c5.2xlarge` EC2 instance with 8vCPUs (4 physical cores) and 16GB of RAM.

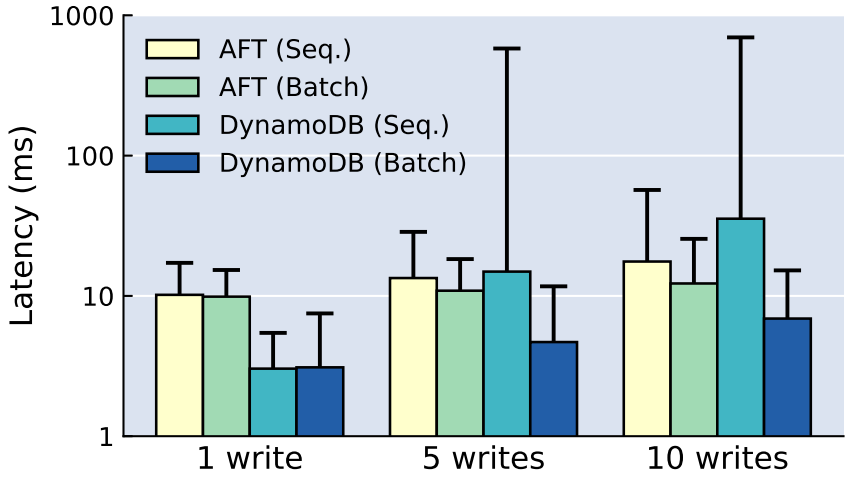


Figure 3.3: The median (box) and 99th percentile (whisker) latencies across 1,000 sequential requests for performing 1, 5, and 10 writes from a single client to DynamoDB and AFT with and without batching. AFT's automatic batching allows it to significantly outperform sequential writes to DynamoDB, while its commit protocol imposes a small fixed overhead relative to batched writes to DynamoDB.

AFT Overheads

We first measure the performance overheads introduced by AFT relative to interacting with cloud storage engines without AFT. To isolate the overheads introduced by Function-as-a-Service platforms, we first measure IO cost with and without AFT interposed (§3.5); we then measure end-to-end latencies and consistency anomalies for transactions running on AWS Lambda over a variety of storage engines (§3.5).

IO Latency

We first compare the cost of writing data directly to DynamoDB and to the cost of the same set of writes using AFT's commit protocol (§3.2). To isolate our performance overheads, we issue writes from a single thread in a VM rather than using a FaaS system. We measure four configurations. Our two baselines write directly to DynamoDB, one with sequential writes and the other with batching. Batching provides best-case performance, but interactive applications can rarely batch multiple writes. AFT takes advantage of batched writes by default in its commit protocol—all client updates sent to the Atomic Write Buffer are written to storage in a single batch when possible. We also measure two configurations over AFT—one where the client sends sequential writes to AFT, and one where the client sends a single batch. We focus specifically on DynamoDB in this experiment because it is the only one of the storage systems we support which enables batching—this allows us to more effectively highlight AFT's overheads.

Figure 3.3 shows the latency for performing 1, 5, and 10 writes. As expected, the latency of

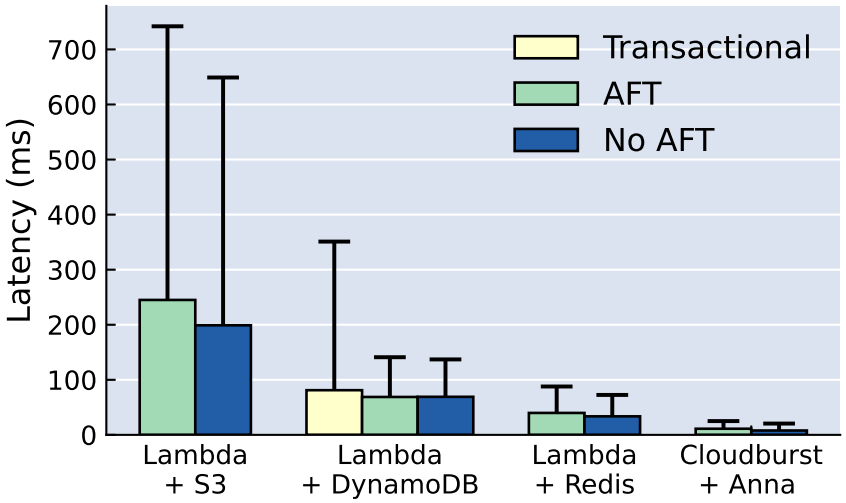


Figure 3.4: The end-to-end latency for executing a transaction with two sequential functions, each of which does 1 write and 2 reads (6 IOs total) on Cloudburst with Anna and Lambda with AWS S3, AWS DynamoDB, and AWS ElastiCache (Redis). Numbers are reported from 10 parallel clients, each running 1,000 transactions.

sequential writes to DynamoDB increases roughly linearly with the number of writes, and the 99th percentile latency increases super-linearly. The latency of batched writes to DynamoDB on the other hand scales much better, increasing by about 2× from 1 write to 10.

Both AFT configurations take advantage of batched writes. AFT Sequential's performance scales better than DynamoDB Sequential's, but latency increases about 70% from 1 write to 10 writes—this is primarily due to the cost and network variance incurred by issuing sequential requests from the client to AFT. The AFT Batch bar measures a client who sends all writes in a single request to AFT, leading to much more consistent performance. We observe a fixed difference of about 6ms between the DynamoDB Batch and the AFT Batch measurements—this is (1) the cost of the extra network overhead imposed by shipping data to AFT, and (2) the cost of writing the extra commit record our protocol requires.

***Takeaway**: For interactive applications that perform sequential writes, AFT significantly improves IO latency by automatically batching updates.*

End-to-End Latency

Next, we measure the end-to-end latency of executing transactions on AWS Lambda with and without AFT interposed between the compute and storage layers. We evaluate three storage systems: AWS S3, AWS DynamoDB, and Redis. We also measure a deployment in which AFT is deployed as the Cloudburst caching layer and compare to the standard Cloudburst architecture. Each transaction is composed of 2 functions, which perform one write and two reads each; all

Storage Engine	Consistency Level	RYW Anomalies	FR Anomalies
AFT	Read Atomic	0	0
Lambda + S3	None	595	836
Lambda + DynamoDB	None	537	779
Lambda + DynamoDB	Serializable	0	115
Lambda + Redis	Shard Linearizable	215	383
Cloudburst + Anna	Last Writer Wins	911	3628

Table 3.2: A count of the number of anomalies observed under Read Atomic consistency for DynamoDB, S3, and Redis over the 10,000 transactions run in Figure 3.4. Read-Your-Write (RYW) anomalies occur when transactions attempt to read keys they wrote and observe different versions. Fractured Read (FR) anomalies occurs when transactions read fractured updates with old data (see §3.1). AFT's read atomic isolation prevents up to 13% of transactions from observing anomalies otherwise allowed by DynamoDB and S3.

reads and writes are of 4KB objects. We chose small transactions because they reflect many real-world CRUD applications and because they highlight AFT's overheads; we use this workload in the rest of our evaluation. We measure performance and consistency anomalies with a lightly skewed workload (a Zipfian coefficient of 1.0).

Figure 3.4 and Table 3.2 show our results. The bars labeled "Plain" in Figure 3.4 represent end-to-end transaction latencies measured by running functions which write data directly to the respective storage engines. When executing requests without AFT, we detect consistency anomalies by embedding the same metadata AFT uses—a timestamp, a UUID, and a cowritten key set—into the key-value pairs; this accounts for about an extra 70 bytes on top of the 4KB payload.

Consistency. AFT's key advantage over DynamoDB, Redis, and S3 is its read atomic consistency guarantee. We measure two types of anomalies here. Read-Your-Write (RYW) anomalies occur when a transaction writes a key version and does not later read the same data; Fractured Read (FR) anomalies occur when a read violates the properties described in §3.1—these encompass repeatable read anomalies (see §3.2). Table 3.2 reports the number of inconsistencies observed in the 10,000 transactions from Figure 3.4. Vanilla DynamoDB and S3 have weak consistency guarantees and incur similar numbers of anomalies—6% of transactions experience RYW anomalies, and 8% experience FR anomalies.

Each Redis shard is linearizable but no guarantees are made across shards. This consistency model combined with low latency IO enables it to eliminate many anomalies by chance, as reads and writes interfere with each other less often. Nonetheless, we still found anomalies on 6% of requests.

Cloudburst's standard architecture incurs the *most* anomalies out of all systems measured. This is because the caching system in Cloudburst by default relies on Anna to periodically propagate updates, and the default interval between updates is 10 seconds—in the interim, the cache

defaults to returning whatever data is stored locally. Thus, updates written earlier in the transaction are unlikely to be stored in the cache, increasing read-your-writes anomalies. This is because each transaction takes a few milliseconds, which means many transactions execute in the default 10 second interval between updates from Anna. Similarly, Cloudburst's default caching scheme exhibits significantly more fractured reads anomalies than other systems because the caches are not kept synchronously updated. Other systems overwrite data in place, meaning there is a higher chance a new, valid version is read by the transaction, while Cloudburst's cache returns old data without respect to AFT's transaction versions.

Finally, we evaluate DynamoDB's transaction mode [30], which provides stronger consistency than vanilla DynamoDB. In contrast to AFT's general purpose transactions, DynamoDB supports transactions that are either *read-only* or *write-only*. All operations in a single transaction either succeed or fail as a group; however, DynamoDB's transaction mode *does not* to guarantee atomicity for transactions that span multiple functions—each transaction is only a single API call. To accommodate this model, we modified the workload slightly: The first function in the request does a two-read transaction, and the second function does a two-read transaction followed by a two-write transaction. We grouped all writes into a single transaction to guarantee that the updates are installed atomically rather than being spread across two separate transactions—this reduces the flexibility of the programming model but is more favorable to DynamoDB's transactional guarantees.

This setup avoids RYW anomalies because all of a request's writes are done in a single transaction. However, reads are spread across two transactions in two different functions, so we still encounter FR anomalies on over 1% of requests. DynamoDB serializes all transactions, so many writes will be executed between the two read transactions, making our application likely to read newer data in the second read transaction that conflicts with data read in the first.

Performance. AFT imposes the highest overhead over S3—25% slower than baseline at median and 14% at 99th percentile. S3 is a throughput-oriented object store that has high write latency variance, particularly for small objects [17, 142], and even in the AFT-less configuration, S3 is 4 to 10× slower than other storage systems. Our design writes each key version to a separate storage key; this is poorly suited to S3, which has high random IO latencies and is sensitive to data layouts. For this reason, we do not consider S3 in the rest of our evaluation. We return to this point in §3.7.

AFT is able to match DynamoDB Plain's performance. In this workload, each function does one write, so when writing directly from Lambda to DynamoDB, function cannot take advantage of write batching. On the other hand, AFT's use of batching offsets the cost of its extra commit metadata write. With transaction mode enabled, DynamoDB proactively aborts transactions in the case of conflict, so the reported latencies include retries. AFT improves median performance over DynamoDB's transaction mode by 18% at median and by 2.5× at the 99th percentile.

Finally, AFT imposes a 20% latency penalty compared to Redis Plain because we are not able to take advantage of batching. While Redis supports a `MSET` operation to write multiple keys at once, that operation can only modify keys in a single shard. Since requests can write arbitrary data, we are not guaranteed to modify keys in a single shard and cannot consistently batch updates.

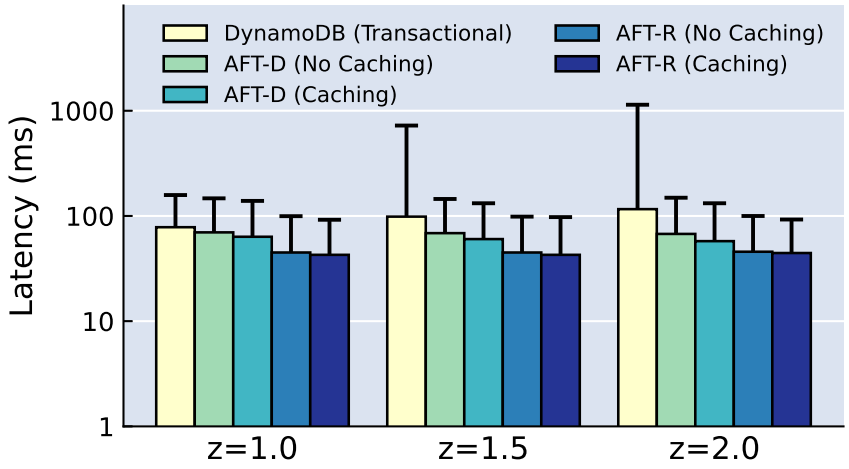


Figure 3.5: End-to-end latency for AFT over DynamoDB (AFT-D) and Redis (AFT-R) with and without read caching enabled, as well as DynamoDB's transaction mode. We vary the skew of the data access distribution to demonstrate the effects of contended workloads. Caching improves AFT-D's performance by up to 15%, while it has little effect on AFT-R's performance. DynamoDB's transaction mode suffers under high contention due to large numbers of repeated retries.

Cloudburst in general exhibits better performance than all of the Lambda-based deployments, as validated in Chapter 2. AFT introduces a small, fixed overhead of about 3ms at the median and 5ms at the tail compared to the standard Cloudburst architecture without AFT. This is primarily due to the latency to move transaction metadata between AFT nodes and the cost of the distributed commit protocol (see Section 3.3). However, as highlighted above, this overhead prevents over 4,500 anomalies.

Here, we establish the low overheads AFT achieves on all systems; in the rest of our evaluation, we focus on the Lambda-based deployments and exclude the Cloudburst architecture in order to demonstrate the generality of the AFT approach.

***Takeaway**: AFT offers performance that is competitive with state-of-the-art cloud storage engines while also eliminating a significant number of anomalies.*

Read Caching & Data Skew

We now turn our attention to AFT's data caching, its effect on performance, and its interaction with access distribution skewness. In this experiment, we use the same workload as in Section 3.5—a 2-function transaction with 2 reads and 1 write per function. Figure 3.5 shows the median and 99th percentile latencies for AFT deployed over DynamoDB (AFT-D) and Redis (AFT-R) with and without caching. We also measure DynamoDB's transaction mode; we omit any configurations that do not provide transactions in some form. We measured 3 Zipfian distributions: 1.0 (lightly contended), 1.5 (moderately contended), and 2.0 (heavily contended).

Interestingly, we find that AFT-R's performance varies very little across all configurations. Read caching does not improve performance because the cost of fetching a 4KB payload from Redis is negligible compared to invoking a Lambda function and executing our read and write protocols. While recent work has shown that Redis' performance can suffer under high contention [140], this experiment measures latencies and thus does not saturate Redis's capacity.

With caching, AFT-D's performance improves by 10% for the lightly contended workload and up to 17% for the heavily contended workload. As distribution skew increases, so does the likelihood that we have a valid key version cached, thus improving performance.

Finally, we measure DynamoDB's transaction mode. Interestingly, we find that for Zipf=1.0, DynamoDB's transactional performance improves relative to Figure 3.4 in §3.5. This is because we use a larger dataset in this experiment (100,000 keys vs 1,000 keys), so the lightly contended workload is less likely to encounter data access conflicts. As we increase contention, performance degrades significantly, and for the most contended workload, AFT-D is 2× faster at median and 7.6× better at the tail.

***Takeaway**: Introducing read caching unlocks significant performance improvements for* AFT, *particularly for skewed access distributions.*

Read-Write Ratios

Next, we look at the effects of read-write ratios within a single transaction. Thus far, our workload has consisted of transactions with only 4 writes and 2 reads split across 2 functions. In this section, we will use longer transactions with 10 total IOs and vary the percentage of those IOs that are reads from 0% to 100%.

Figure 3.6 shows our results. With AFT running over DynamoDB, performance is largely consistent with less than 10% variance. There is a slight increase in median latency from 0% reads to 80% reads. At 0% reads, there are two API calls to DynamoDB—one to write the batch of updates, and the other to write the transaction's commit record. As we add reads, each individual read results in a separate API call, resulting in a slight increase in latency. At 100% reads, we remove the batch write API call, leading to a small dip in latency. 0% and 20% reads (i.e., 10 and 8 writes, respectively) have higher tail latencies because the larger numbers of writes increase the chance that the batch writes hit slower shards in the underlying storage engine—this is similar to the differences in tail latencies seen in Figure 3.3.

AFT over Redis shows very little variation in performance across all read-write ratios. This is because Redis (in cluster mode) does not support writing multiple objects at once, and the system treats reads and writes roughly uniformly. As a result, for all configurations, we make 11 API calls—10 for the IOs and 1 for the final commit record.

***Takeaway**:* AFT *maintains consistent performance across a variety of read-write ratios with minor variations based on the characteristics of the underlying storage system.*

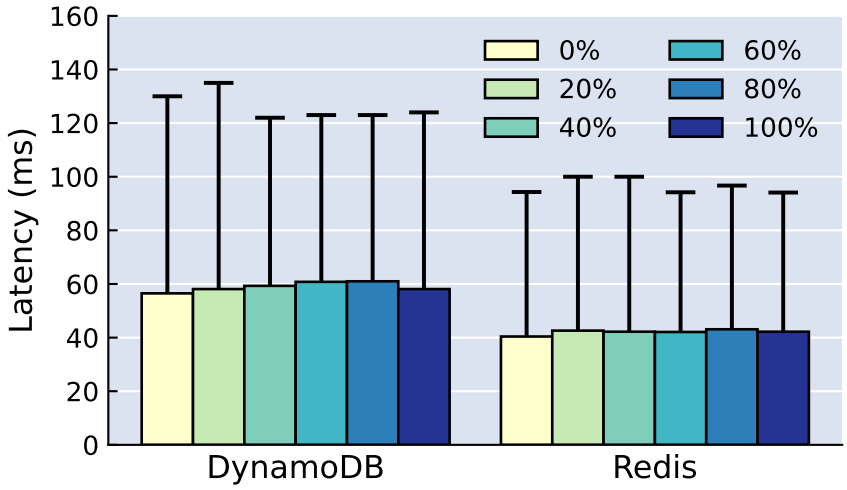


Figure 3.6: Median and 99th percentile latency for AFT over DynamoDB and Redis as a function of read-write ratio, from transactions with 0% reads to transactions with 100% reads. AFT over Redis should little variation, while our use of batching over DynamoDB leads to small effects based on read-write ratios.

Transaction Length

In this experiment, we study the performance impact of transaction length. We vary transactions from 1 function to 10 functions, where each function consists of 2 reads and 1 write. Figure 3.7 shows our results.

AFT scales roughly linearly with transaction length over both DynamoDB and Redis. As in previous experiments, AFT's use of DynamoDB's batching capabilities means that the overhead of increased writes is masked with the batch update operation. As a result, 10-function transactions are only 6.2× slower than 1-function transactions—intuitively, this corresponds to the fact that $\frac{2}{3}$ (or 67%) of our API calls are reads, which scale linearly, while the remaining writes are batches into one API call.

As before, Redis requires separate API calls for each write operation and thus 10-function transactions are 8.9× slower than 1-function transactions. The remaining difference is because the cost of writing an extra commit record to storage as a fixed cost is a large portion of the operating time of a 1-function transaction but a much smaller portion for 10-function transactions. DynamoDB is 59% slower than Redis for 1-function transactions but only 13% slower for 10-function transactions. In the rest of our experiments, we use 2-function transactions because they more clearly highlight our overheads than longer transactions do.

Takeaway: *AFT scales linearly with transaction length and is able to mask update and commit overheads for longer transactions.*

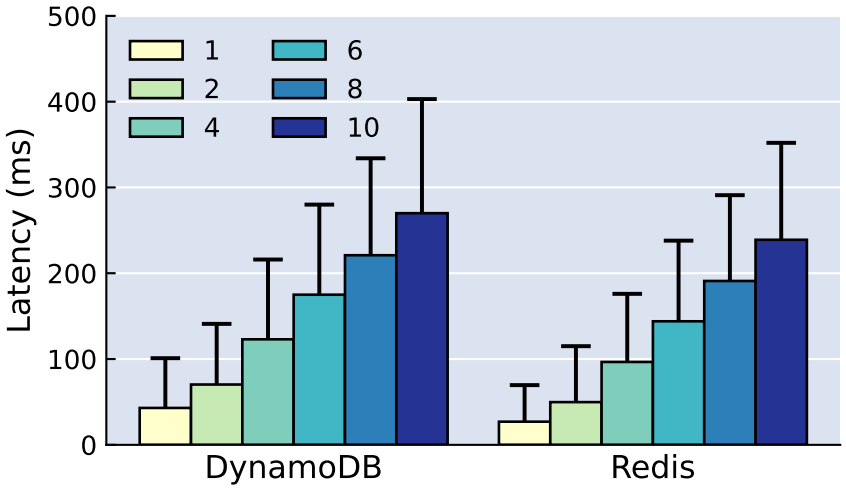


Figure 3.7: Median and 99th percentile latency for AFT over DynamoDB and Redis as a function of transaction length, from 1 function (3 IOs) to 10 functions (30 IOs). Longer transactions mask the overheads of AFT's protocols, which play a bigger role in the performance of the shorter transactions.

Scalability

In this section, we evaluate AFT's scalability relative to the number of parallel clients. We first measure the number of clients a single AFT node can support in §3.5. We then measure the overhead of the distributed protocols in §3.3 by measuring throughput in distributed deployments.

Single-Node Scalability

In this section, we measure the number of parallel clients that a single AFT node can support. We run the same 2-function, 6-IO transactions as before under the moderate contention level (Zipf=1.5), and we we vary the number of clients from 1 to 50. Each client makes 1,000 requests by synchronously invoking the transaction, waiting for a response, and then triggering another transaction. Figure 3.8 shows our results with AFT deployed over DynamoDB and Redis.

We find that AFT scales linearly until 40 and 45 clients for DynamoDB and Redis, respectively. At this point, contention for shared data structures causes AFT's throughput to plateau. Similar to previous experiments, AFT over Redis achieves better performance than AFT over DynamoDB. Because Redis offers significantly lower IO latencies, each transaction completes faster (see Figure 3.4). Our clients synchronously invoke each Lambda, so the reduced latency directly translates to better throughput. At peak, AFT over Redis is able to achieve 900 transaction per second, while AFT over DynamoDB achieves just under 600 transaction per second.

***Takeaway**: A single AFT node is able to scale linearly to over 40 clients (600 tps), demonstrating the low overhead of our read atomic protocols.*

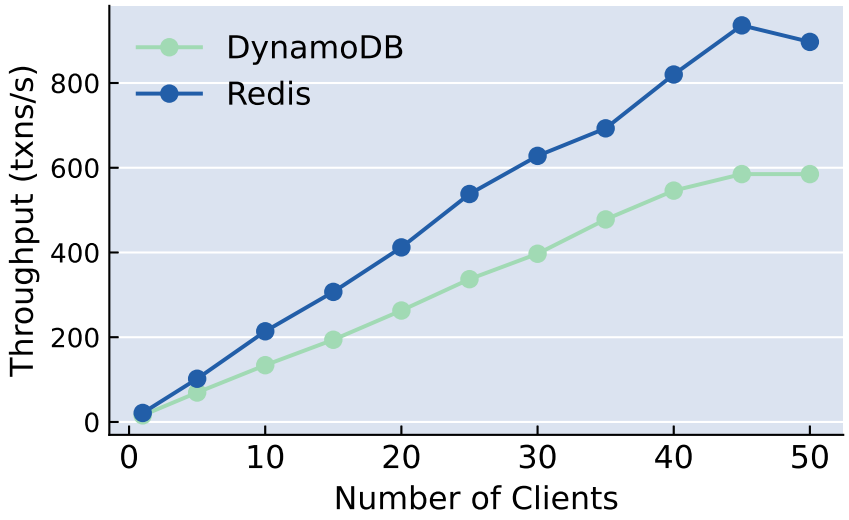


Figure 3.8: The throughput of a single AFT node as a function of number of simultaneous clients issues requests to it. We can see a single node scales linearly until about 40 clients for DynamoDB and 45 clients for Redis, at which point, the throughput plateaus.

Distributed Scalability

We now turn to measuring AFT's ability to scale smoothly for multi-node deployments. Based on the results in §3.5, we ran 40 clients per AFT node and progressively scaled up the number of nodes. Figure 3.9 shows our results.

Deployed over DynamoDB, we find that AFT is able to scale seamlessly to 8,000 transactions per second for 640 parallel clients. AFT scales at slope that is within 90% of the ideal slope, where the ideal throughput is the number of nodes multiplied by a single node's throughput. We originally intended to demonstrate scalability to a thousand parallel clients, but we were restricted by AWS DynamoDB's resource limits, which would not let us scale beyond what is shown.

With Redis, we observe that AFT is similarly able to scale linearly. Similar to §3.5, AFT over Redis has a higher aggregate throughput due to lower IO latencies. Nonetheless, throughput remains within 90% of ideal. Throughput for 640 clients plateaus not because of AFT overheads but because we were limited by the number of concurrent function invocations supported by AWS Lambda.

Note that we manually configured both storage systems with the appropriate resources. We chose to disable DynamoDB's autoscaling because the goal of this experiment was not to measure efficacy of their autoscaling policy. In general, however, DynamoDB's autoscaling support makes it well-suited to serverless applications, while Redis is a fixed-deployment system with high reconfiguration overheads.

***Takeaway**: AFT is able to efficiently scale to thousands of transactions per second and hundreds of parallel clients within 90% of ideal throughput.*

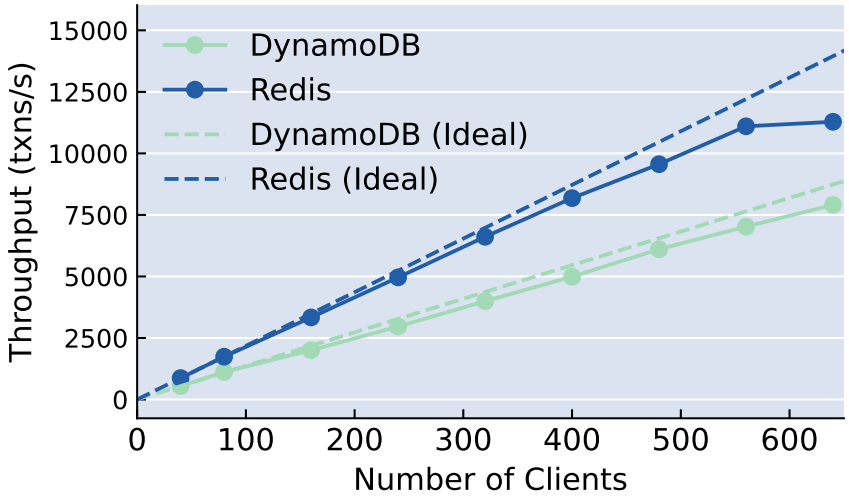


Figure 3.9: AFT is able to smoothly scale to hundreds of parallel clients and thousands of transactions per second while deployed over both DynamoDB and Redis. We saturate either DynamoDB's throughput limits or AWS Lambda's concurrent function inovcation limit while scaling within 90% of ideal throughput.

Garbage Collection Overheads

In this experiment, we quantify the overhead of enabling AFT's global garbage collection (§3.4). We run a single AFT node with 40 clients and measure throughput with garbage collction enabled and disabled. We also measured the number of transactions deleted per second when garbage collection is enabled. Figure 3.10 shows our results.

There is no discernible difference between throughput with garbage collection enabled and disabled. The bulk of the work related to determining transaction supersedence happens periodically on each node to reduce metadata overheads (see §3.3). As a result, the global GC protocol simply collects lists of superseded transactions from all AFT nodes and deletes transactions that all nodes consider superseded.

The cost of this garbage collection process is that we require separate cores that are dedicated to deleting old data. However, the resources allocated to garbage collection are much smaller than the resources required to run the system: For the four cores we used to run the AFT node, we only required 1 core to delete transactions.

Takeaway: AFT's *local metadata garbage collection enables efficient global deletion of superseded data with no effect on system throughput and at reasonable added cost.*

Fault Tolerance

Finally, in this experiment, we measure AFT's performance in the presence of failures. We know from our protocols and prior experiments that the behavior and performance of AFT nodes is

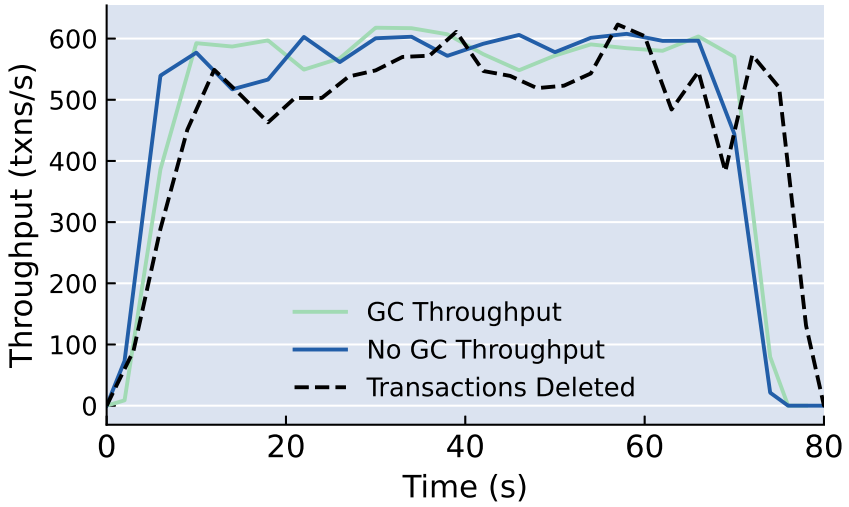


Figure 3.10: Throughput for AFT over DynamoDB with and without global data garbage collection enabled. The garbage collection process has no effect on throughput while effectively deleting transactions at the same rate AFT processes them under a moderately contended workload (Zipf=1.5).

independent. In this experiment, we are looking to measure the effect of a node failure and the cost of recovering from failure—that is how long it takes for a node to cold start and join the system. Figure 3.11 shows our results.

We run AFT with 4 nodes and 200 parallel clients and terminate a node just before 10 seconds. We see that throughput immediately drops about 16%. Within 5 seconds, the AFT management process determines that a node has failed, and it adds assigns a new node to join the cluster. Note that we pre-allocate "standby" nodes to avoid having to wait for new EC2 VMs to start, which can take up to 3 minutes.

Over the next 45 seconds, this new node downloads the AFT Docker container and updates its local metadata cache. Just after 60 seconds, the node joins the cluster, and throughput returns to pre-failure peaks within a few seconds. Note that throughput is on a slight downward slope between 10 and 60 seconds—this is because the remaining three nodes are saturated, and the queue of pending requests grows, causing throughput to decrease.

Note that the overheads involved in starting a new node can be further mitigated by downloading containers in advance and by maintaining standbys nodes with warm metadata caches. These are engineering tasks that are not fundamental to the design of AFT.

Takeaway: AFT's *fault management system is able to detect and recover from faults in a timely fashion.*

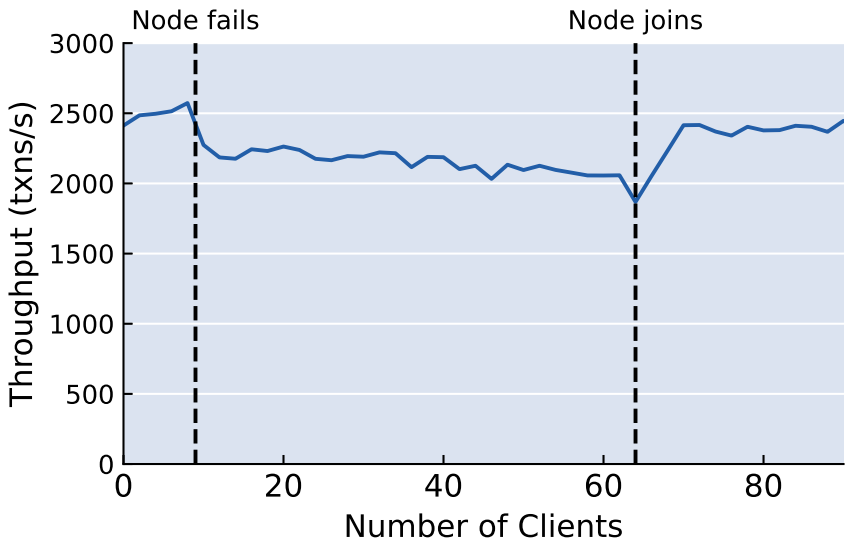


Figure 3.11: AFT's fault manager is able to detect faults and allocate new resources within a reasonable time frame; the primary overheads we observe are due to the cost of downloading Docker containers and warming up AFT's metadata cache. AFT's performance does not suffer significantly in the interim.

3.6 Related Work

Atomic Reads. The RAMP protocols in [9] are the only prior work we are aware of that explicitly addresses read atomic isolation. However, another coordination-free consistency model that guarantees atomic reads atomic is transactional causal consistency (TCC), implemented in systems like Occult [89] and Cure [3]. In addition to atomic reads, causal consistency guarantees that reads and writes respect Lamport's "happens-before" relation [80]. In general, causal systems must track each transaction's read set in addition to its write set, which adds a metadata overhead that read atomicity does not require.

Two key aspects distinguish our approach from prior work. First, [9], [89], and [3] achieve read atomicity at the storage layer, whereas AFT is a shim above the storage engine. This allows for more flexibility as AFT users can pick any key-value storage system, while still maintaining consistency. Second, the mechanisms used to achieve read atomic isolation in [89] and [3] rely on fixed node membership at the storage layer, which we cannot assume in an autoscaling serverless environment. The read atomic isolation protocols in this paper do not require knowledge of node membership. AFT only requires membership for garbage collection of data, which happens off the critical path of transaction processing.

Multi Versioning. The idea of multi-version concurrency control and storage dates back to the 1970s [107, 14]. The original Postgres storage manager [125] introduced the notion of maintaining a commit log separate from data version storage, an idea we adopted in AFT. More recently, a

variety of systems have proposed and evaluated techniques for multi-version concurrency [141, 95, 35]. These systems all offer some form of strong transactional consistency (e.g., snapshot isolation, serializability), which AFT and the RAMP protocols do not. Similar to the causal approaches, these systems also enforce consistency in the storage system, while AFT offers consistency over a variety of storage backends.

Fault Tolerance. There is a rich literature on fault tolerance for distributed systems (see, e.g., [104, 123]). Many techniques are conceivably applicable in the FaaS context, from checkpoint/restart of containers or virtual machines (e.g. [83]) to log- (or "lineage-") based replay (e.g. [136]) to classical process pairs (e.g., [11]). Our approach is based on the simple retry-from-scratch model used by existing serverless platforms such as AWS Lambda and Google Cloud Functions, and appropriate to short-lived interactive ("transactional") tasks. As noted above, existing FaaS platforms attempt to offer at-least-once execution semantics, but may expose fractional writes from failed function attempts. AFT's built-in atomicity and idempotence guarantees allow a simple retry scheme to achieve exact-once semantics that provides both safety and liveness in the face of failures.

3.7 Conclusion and Takeaways

In this chapter, we discussed AFT, a low-overhead fault-tolerance shim for serverless computing. AFT interposes between commodity FaaS platforms and key-value stores to achieve fault-tolerance by transparently guaranteeing read atomic isolation [9]. We develop new distributed protocols for read atomicity, which build on shared storage for high throughput and do not require pre-declared read and write sets. AFT supports read atomic transactions in the context of the Cloudburst stack but, importantly, is general purpose enough to interface with commodity serverless infrastructure like AWS Lambda and AWS DynamoDB. The system adds minimal overhead to existing serverless architectures and scales linearly with the size of the cluster, while offering exactly-once execution in the face of failures.

The success of a consistency- and fault tolerance-oriented shim layer suggests that serverless developers need not be limited to the general-purpose, commodity guarantees in systems like AWS DynamoDB and S3. These systems are architected to provide the minimum guarantees required in the most scalable fashion—while this is natural for an extremely general-purpose service, it is often insufficient for applications that require more from their infrastructure. Interposing between the application and storage to extend the guarantees provided at the lower layers without necessarily modifying the infrastructure in line is a promising direction—for example, AFT could be modified to support strong consistency or different coordination-free guarantees.

Chapter 4

Low-Latency Serverless Dataflow for Prediction Serving

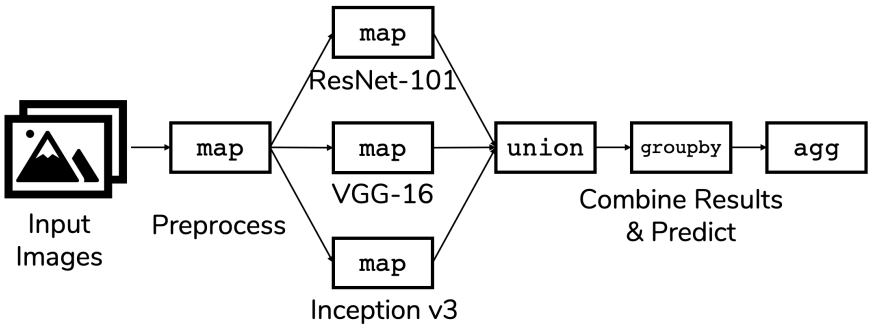
Thus far, we have focused on the serverless infrastructure layer, its guarantees, and its performance, largely using synthetic workloads. In this chapter, we turn our attention to more complex compute workloads that are well-suited to run on the infrastructure described thus far. As an illustrative case study, we focus on machine learning prediction serving.

Machine learning has become ubiquitous over the last decade in an increasingly broad set of fields, ranging from manufacturing to medicine and sports. Much of the systems research surrounding machine learning has focused on improving the infrastructure that supports the creation of models—tools like Tensorflow [1], PyTorch [100], and MLLib [91] have greatly simplified the process of developing and training models at scale. The models trained in these systems are deployed in numerous applications—generating social media feeds, enabling chat bots, designing video games, and so on.

This process of deploying a trained model for use in a larger application is often called *prediction serving*. Prediction serving is a particularly interesting task because it combines the complex computations of machine learning with the performance requirements of interactive applications. More specifically, it has three key properties: (1) it is *computationally intensive*; (2) it is a part of an interactive application, meaning it has *low latency requirements*; and (3) it is *compositional*, meaning a single request passes through multiple stages. Figure 4.1 shows an example prediction serving pipeline, which normalizes an input image, runs three image classification models in parallel, and combines the results to make a prediction (often called a *model ensemble*).

Commercial services like AWS Sagemaker and Azure ML have emerged in recent years to attempt to fill this gap. These systems deploy individual models as separate microservices which enables neat modularity of pipeline stages. However, this approach gives the system no visibility into the *structure* of the computation and how the individual microservices relate to each other, which significantly complicates debugging and limits end-to-end performance optimization.

Alternately, Pretzel [81] is a recent research system which explored fine-grained optimizations of *white-box*, dataflow-style prediction pipelines that leverage full visibility into the pipeline and the semantics of each stage. However, this approach also requires the developer to rewrite their




```
fl = new cloudflow.Dataflow([('url', str)])
img = fl.map(img_preproc)
p1 = img.map(resnet_101)
p2 = img.map(vgg_16)
p3 = img.map(inception_v3)
fl.output = p1.union(p2,p3).groupby(rowID).agg(max,'conf')
```

Figure 4.1: An example prediction serving pipeline to classify a set of images using an ensemble of three models, and the Cloudflow code to specify it. The models are run in parallel; when all finish, the result with the highest confidence is output.

individual models (i.e., pipeline stages) in Pretzel's DSL, which is cumbersome and limits model development. It also makes Pretzel inflexible to innovations in machine learning frameworks.

We argue for a natural middle ground: a simple dataflow API for prediction pipelines based on arbitrary (black-box) operators, which are typically models trained by users in their library of choice (e.g., TensorFlow, PyTorch, Scikit-Learn). Rather than requiring models to be rewritten in a white-box fashion, a graph of familiar dataflow operators (e.g., `map`, `filter`, `join`) can be used to wrap black-box models. The dataflow API also provides visibility into the structure of the pipeline, which allows for the end-to-end computation to be optimized—e.g., two operators that have large data dependencies might be fused together.

Existing large-scale distributed dataflow systems are not well suited for prediction serving for two reasons. First, prediction serving tasks have low latency requirements. Systems like Apache Spark [145] and Apache Flink [20], however, are optimized for throughput rather than for minimizing tail-latency.

Second, prediction serving systems—like all interactive applications—must operate in the presence of bursty and unpredictable workloads [50, 115], which requires fine-grained resource allocation. Unfortunately, efforts [71, 42, 85, 40, 84, 70] to address operator-level auto-scaling have primarily focused on meeting throughput requirements.

We present Cloudflow, a dataflow system for prediction serving pipelines, built on top of Cloudburst. Layering on top of a stateful serverless platform enables fine-grained resource allocation, provides a simple functional programming model to match dataflow, and supports efficient state retrieval.

Cloudflow exposes a dataflow API that enables simple composition patterns common in pre-

diction serving. The dataflow model enables us to apply both dataflow and prediction serving optimizations (e.g., operator fusion, competitive execution) to optimize those pipelines, even while treating models as black boxes. We demonstrate that these optimizations can be applied without user intervention. In practice, Cloudflow is able to outperform prior systems for prediction serving by as much as 2×, and—critically—meets latency goals of demanding applications like real-time video analysis.

In sum, our contributions are as follows:

- The modeling and implementation of prediction serving pipelines as dataflow graphs using a familiar API of functional operators. This includes the implementation of common patterns like ensembles and cascades.
- Leveraging and extending a stateful serverless platform to enable efficient dataflows without changes to user programs, as discussed in Sections 4.2 and 4.3.
- The automatic application of well-studied optimizations from the dataflow and prediction serving domains to Cloudflow dataflows.
- A thorough evaluation of Cloudflow, demonstrating the benefits of optimized dataflow on both synthetic benchmarks and on real-world prediction pipelines.

4.1 Background and Motivation

In this section, we discuss challenges posed by prediction serving (Section 4.1). We then argue that dataflow is well-suited to address these challenges and discuss how we optimize prediction pipelines (Section 4.1), and we motivate the use of a serverless runtime and our choice, Cloudburst (Section 4.1).

Prediction Serving

Prediction serving has become a common part of many tasks, such as organizing social media news feeds, responding to voice assistant queries (Alexa, Siri, etc.), detecting fraud in financial transactions, and analyzing radiological scans. The models used in these applications, particularly with the advent of deep neural networks, can be incredibly computationally intensive [64, 51]. As a result, it has become commonplace for these models to run on specialized hardware, like GPUs or Google's TPU (Tensor Processing Unit) [69].

Furthermore, these applications are often interactive and require predictions to be made within tight latency bounds. Facebook published a detailed case study [50] in which they describe models that query user-specific state and generate predictions in real-time—about a few hundred milliseconds. Predictions that miss their latency deadlines are usually discarded in favor of a default response [149, 53], making it critical to minimize both median *and* tail (99th percentile) latencies.

Importantly, it is typical for applications to compose multiple models to make a single prediction [81]. Consider the example in Figure 4.1. This image classification pipeline first preprocesses and normalizes an image, and then runs three different models in parallel—each has different specializations. The results of all three models are aggregated to generate a prediction. This particular pattern is called a *model ensemble*.

Dataflow and Optimizations

The sequence of tasks in a prediction pipeline forms a natural dataflow graph: Each stage of the pipeline receives an input, applies a transformation, and passes the result downstream. The output of the last stage is the prediction rendered by the whole pipeline. Using a traditional dataflow model with operators like `map`, `filter`, and `join` makes constructing pipelines easy—the ensemble in Figure 4.1 being a case in point. On inspection this may seem straightforward, but it is in stark contrast with existing systems: AWS Sagemaker forces users to manually construct containers for each model stage, while Pretzel [81] requires users to rewrite their models in a custom API to leverage the system's optimizations. We argue that a traditional dataflow API enables us to implement a number of pipeline optimizations *without* requiring any modification of black-box ML models.

We focus on five optimizations from the data processing and prediction serving literature that apply here [6, 18, 25, 81, 63, 75], but this list is not meant to be exhaustive.

Operator Fusion. Separate stages in a dataflow may pass significant data like videos or images. To minimize communication, it can be beneficial to fuse these operators to avoid data movement.

Competitive Execution. Machine learning models can have highly variable execution times [76], depending on model structure and input complexity. Similar to straggler mitigation in MapReduce [31], competitive execution of inference has been shown to improve tail latencies for ML models [126, 75].

Operator Autoscaling and Placement. Given the diversity of compute requirements in a pipeline (e.g., simple relational operations vs neural-net inference), we often want to devote more—and more specialized—resources to bottleneck tasks.

Data Locality. While prediction pipelines are typically computationally intensive tasks, they also often involve significant data access [46]. Remote data access can easily become a bottleneck unless we optimize for data locality by placing computations near cached data.

Batching. Large models like neural nets can benefit greatly from batched execution on vectorized processors like GPUs [15, 43], resulting in better throughput at the cost of higher latency. The improved throughput translates into better resource utilization and thereby reduces costs.

In Section 4.3 we explore each of these optimizations in more detail, and explain how we deliver on them in Cloudflow.

Deploying Prediction Pipelines

Having motivated the use of dataflow to construct and optimize prediction pipelines, we now consider the runtime on which they are executed. Two key concerns affect our decision: (1) tight latency constraints, and (2) nimble responses to unpredictable workload changes. Unfortunately, batch streaming dataflow systems (e.g., Apache Flink [20], Apache Spark [145]) are unsuitable on both counts—they are throughput-oriented systems that do not scale easily.

Cloudburst is well-suited to to running these workloads because—as a FaaS platform—it natively support fine-grained elasticity, which is an integral part of operator autoscaling. Additionally, the functional programming model in FaaS neatly matches the operator-based abstractions of dataflow—each operator can be deployed as a separate function.

Commercial FaaS offerings are ill-suited to this task for all of the reasons described in Chapter 1—high latencies, lack of support for stateful workloads, and unpredictable performance. As we have already detailed, Cloudburst avoids these pitfalls and is particularly well-suited to running prediction dataflows: In addition to best-in-class performance, its DAG programming model is an obvious match for dataflows.

4.2 Architecture and API

In this section, we describe the Cloudflow API for composing dataflows, how that API captures common prediction pipelines, and how Cloudflow pipelines are compiled down to Cloudburst serverless functions.

Dataflow API

The Cloudflow API centers around three main concepts: a simple `Table` type for data, computational `Operators` that compute over `Table`s, and a functional-style `Dataflow` to author specifications of DAGs of `Operator`s. A Python program can specify a `Dataflow` and execute it on a `Table`; the Cloudflow runtime is responsible for taking that specification and invoking the `Operator`s to generate an output `Table`.

The core data structure in Cloudflow is a simple in-memory relational `Table`. A `Table` has a *schema*, which is a list of column descriptors, each consisting of a name and an associated data type (e.g., `str`, `int`). It also has an optional *grouping column*[1]. We briefly defer our description of grouping columns to the discussion below of the dataflow operators in Table 4.1.

A Cloudflow `Dataflow` represents a *specification* for a DAG of dataflow operators with a distinguished input and output, as in Figure 4.1. One can think of a `Dataflow` as a declarative query, or a lazy functional expression, describing how to compute an output table from an input table. We step through Figure 4.2 to illustrate. A `Dataflow` instance is instantiated with an input schema (Figure 4.2, line 1). To construct pipelines, the `Dataflow` class includes a set of method

[1] Our syntax naturally extends to support a list of grouping columns, but we omit that detail here for descriptive simplicity. Even without that syntax, one can form composite grouping columns with a `map` operator.

API	Inputs	Output Type	Description
map	fn: $(c_1, ..., c_n) \rightarrow (d_1, ..., d_m)$, table: Table$[c_1, ..., c_n][column?]$	Table$[d_1, ..., d_n][column?]$	Apply a function fn to each row in table
filter	fn: $(c_1, ..., c_n) \rightarrow bool$, table: Table$[c_1, ..., c_n][column?]$	Table$[c_1, ..., c_n][column?]$	Apply Boolean function to each row in table and include only rows with true results
groupby	column: str, table: Table$[c_1, ..., c_n][]$	Table$[c_1, ..., c_n][column]$	Group rows in an ungrouped table by the value in column
agg	agg_fn: $\{c_i\} \rightarrow d$, table: Table$[c_1, ..., c_n][column?]$	Table$[column?, d][]$	Apply a predefined aggregate function agg_fn (count, sum, min, max, avg) to column c_i of table
lookup	key: str? c_k?, Table$[c_1, ..., c_n][column?]$	Table$[c_1, ..., c_n, key][column?]$	Retrieve an object from the underlying KVS and insert into the table
join	left: Table$[c_1, ..., c_n][]$, right: Table$[d_1, ..., d_m][]$, key: str? how: left?, outer?	Table$[c_1, ..., c_n, d_1, ...d_m][]$	Join two Tables on key, using the automatically assigned query ID as a default. Optionally specify left join or (full) outer join mode.
union	{Table$[c_1, ..., c_n][column?]$, ...}	Table$[c_1, ..., c_n][column?]$	Form union of many Tables with matching schemas.
anyof	{Table$[c_1, ..., c_n][column?]$, ...}	Table$[c_1, ..., c_n][column?]$	Pick any one of many Tables with matching schemas.
fuse	sub_dag: Flow, table: Table$[c_1, ..., c_n][column?]$	Table$[d_1, ..., d_m][column?]$	An encapsulated chain of operators (see Section 4.3)

Table 4.1: The core Operators supported by Cloudflow. Each accepts a Table as input and and returns a Table as output. Our table type notation here is Table$[c_1, ..., c_n][column]$, where $c_1, \ldots, c_n$ is the schema, and $column$ is the grouping column. Optional items are labeled with a ?.

```
1  fl = cloudflow.Dataflow([('jpg_url', str)])
2  img = fl.map(img_preproc)
3  pred = img.map(resnet)
4  label = pred.map(convert_to_label)
5  fl.output = label
6  fl.deploy()
7
8  input_tbl = \
9    Table([('jpg_url', str)],
10         ['s3://mybucket/cats.jpg', 's3://mybucket/dogs.jpg'])
11 out = fl.execute(input_tbl)
12 out.result()
```

Figure 4.2: A script to create a Cloudflow dataflow and execute it once.

names that correspond one-to-one with the Operator names in Table 4.1; these methods generate new Dataflow objects that append the specification of the corresponding operator onto their input dataflow. So, for example line 2 represents a dataflow that has a single-column table as its input, followed by the expression map(img_preproc); line 3 represents a dataflow equivalent to flow.map(img_preproc).map(resnet), and so on. Dataflow fl is *valid* after its output member is assigned (line 5). This assignment requires its right-hand-side to be derived from the same Dataflow fl, representing a connected flow from the input to the output of fl.

To prepare a Dataflow for execution, we use the deploy method (line 6), which compiles the

`Dataflow` and registers it with the Cloudburst runtime. Once deployed, we can repeatedly execute the `Dataflow` by passing in a `Table` with the corresponding schema. `execute` immediately returns a Python future that represents the result table of the execution (line 11). Cloudburst schedules and executes the DAG as described in [121], and results are stored in Anna. The result is accessed by calling the `result` method of the future returned by `execute` (line 12). During execution of a `Dataflow`, each row in the input is automatically assigned a unique row ID, which stays with the row throughout execution.

Dataflow Operators. The set of Cloudflow `Operator`s is similar to the relational algebra, or the core of a dataframe API. It consists of a small set of dataflow operators, each of which takes in `Table`s and produces a `Table`. Table 4.1 provides an overview of the input/output types of each operator. The operators look familiar from batch processing and dataframe libraries, but recall that our focus here is on lightweight, interactive predictions over small in-memory request `Table`s.

The `map`, `filter`, `join` and `union` operators are fairly standard. `groupby` takes an ungrouped `Table` and returns a grouped `Table`. `map`, `filter`, `union`, `anyof` and `fuse` all accept `Table`s that can be grouped or ungrouped (denoted by $[column?]$) and return `Table`s with the same grouping. `anyof` passes exactly one of its input tables to the output; the decision is left up to the runtime (Section 4.3). `fuse` is an internal operator that executes multiple other operators over a `Table`—we discuss its use in Section 4.3.

The `agg` operator supports basic aggregations: counts, sums, averages, min, and max. When `agg` is applied over an ungrouped table, it returns a table with a single row; when applied over a grouped table, `agg` returns an ungrouped table with one row per input group containing the group value and aggregate result. `join` takes two input tables, both of which must be ungrouped, and joins them on a user-specified key; if no key is specified, the two tables are joined on the row ID. Cloudflow also supports left joins, where rows from the left table are included in the output even if there is no match in the other table, and outer joins, in which rows from each table are included in the output even if there are no matches.

Finally, `lookup` allows dataflows to interact with data outside the input table. Specifically it allows for reads from the Anna KVS that Cloudburst uses for storage. `lookup`s can take either a constant or a column reference. If the input is a column reference c_k, the effect is akin to a join with the KVS: for each input row, Cloudflow retrieves the matching object in the KVS. As we discuss in Section 4.3, we use the information in the `lookup` operator to take advantage of Cloudburst's locality-aware scheduling. In practice, many of the API calls in Table 4.1 have other arguments for user convenience (e.g., naming output columns) and to give Cloudflow hints about resource requirements and optimizations (e.g., requires a GPU, supports batching). We omit describing these for brevity.

Typechecking and Constraints. Similar to TFX [12], Cloudflow supports basic typechecking to ensure the correctness of pipelines. If the input type of an operator does not match the output type of the previous operator, Cloudflow raises an error.

To this end, we require programmers to provide Python type annotations for the functions they pass into `map` and `filter`. Since Python is a weakly-typed language, these type annotations

```
1 flow = cloudflow.Dataflow(['jpg_url', str])
2 img = flow.map(preproc)
3 simple = img.map(simple_model)
4 complex = simple.filter(low_confidence).map(complex_model)
5
6 flow.output = simple.join(complex, how='left').map(max_conf)
```

Figure 4.3: A simple two-model cascade specified in Cloudflow.

do not guarantee that the data returned by a function matches the annotation. At runtime, the type of each function's output is inspected using Python's `type` operator. Once again, if the type does not match the function's annotation, Cloudflow raises an error. This prevents Python from arbitrarily coercing types, causing pipelines to fail silently—generating incorrect or nonsensical results without surfacing an error.

Prediction Serving Control Flow

Having described Cloudflow's dataflow model, we show how it simplifies the implementation of control flow constructs common in prediction pipelines. We describe some patterns and highlight their simplicity with code snippets. This section is not meant to be exhaustive—rather we illustrate Cloudflow's fitness for standard prediction serving control flow.

Ensembles. In an ensemble pattern (e.g., Figure 4.1), an input is evaluated by multiple models in parallel. After all models finish evaluating, the results are combined, either by picking the prediction with the highest confidence or by taking a weighted vote of the results. Figure 4.1 shows a Cloudflow implementation of a model ensemble. After a preprocessing stage (line 2), three models are evaluated in parallel (lines 3-5), the results are unioned, and a final prediction is selected by using `agg` to pick the prediction with the maximum confidence (line 6).

Cascades. In a cascade, models of increasing complexity are executed in sequence. If an earlier model returns a prediction of sufficiently high confidence, then later models are skipped. Figure 4.3 shows the Cloudflow implementation of a model cascade. After the simple model is executed (line 3), we retain rows with low confidence and apply the complex model to them (line 4). We then join the simple and complex models' results using a left join to include rows that were filtered out from the simple model (line 6). We then apply a `max_conf` function to report the prediction with the highest confidence.

Discussion

In addition to the dataflow operators highlighted in Table 4.1, Cloudflow also has an `extend` method. `extend` takes in another valid flow as an argument and appends its DAG to the existing flow, creating a dataflow that chains the pair. This allows for the easy composition of two flows. For example, multiple users in an organization might share an image preprocessing flow which they each extend with a custom image classification flow.

In sum, Cloudflow's API significantly simplifies the process of constructing prediction pipelines as compared to other systems. Our dataflow model provides natural abstractions for *composition*—e.g., chains of user-defined code in Figure 4.2, parallel executions with joins in Figures 4.1 and Figure 4.3. AWS Sagemaker has limited support for prediction pipelines—all operators must be in a sequence—and AzureML and Clipper have no support for multi-stage pipelines to our knowledge.

Additionally, Cloudflow takes simple Python specs and automatically deploys pipelines for the user in a serverless framework. AzureML and Sagemaker require users to create custom containers for each stage[2]. As we discuss in Section 4.4, porting pipelines to other systems required writing and deploying long-lived driver programs to manage each request as it moved through a pipeline—something we avoid altogether in Cloudflow.

Finally, using a dataflow abstraction makes the resulting DAG of computation amenable to a variety of optimizations. We turn to that topic next.

4.3 Optimizing Dataflows

In this section, we describe the Cloudflow implementation of each of the optimizations from Section 4.1. All the techniques described in this section are automatic optimizations; the user only needs to select *which* optimizations to enable. We return to the topic of automating optimization selection in Section 4.6. Automatic optimization is a key benefit of the dataflow model, allowing users to focus on pipeline logic, while Cloudflow takes care of operational concerns including deployment and scheduling.

The Cloudflow dataflow API produces a DAG of operators selected from Table 4.1. Cloudburst provides a lower-level DAG-based DSL for specifying compositions of black-box Python functions [121]. A naive 1-to-1 mapping of a user's Cloudflow DAG into an isomorphic Cloudburst DAG will produce correct results, but we show in this section that we can do much better.

Our compilation occurs at two levels. **Dataflow rewrites** capture static Cloudflow-to-Cloudflow optimizations within the dataflow topology, including operator fusion and competitive execution. **Dataflow-to-FaaS compilation** translates Cloudflow dataflows to Cloudburst DAGs that expose opportunities for dynamic runtime decisions in the FaaS infrastructure. These include *wait-for-any* DAG execution, autoscaling and hardware-aware function placement, locality-aware scheduling and batching.

In many cases Cloudburst was not designed to leverage optimizations apparent at the Cloudflow layer. As a result we had to extend Cloudburst in terms both of its API and its runtime mechanisms; we highlight these changes below.

Operator Fusion In operator fusion, a chain of `Operator`s within a Cloudflow DAG is encapsulated into a single `fuse` operator. The benefit of `fuse` is that it is compiled into a single Cloudburst function; as a result, the Cloudburst DAG scheduler places all logic within `fuse` at a single

[2]These systems will automatically generate a containerized version of the model only if you use their development infrastructure end-to-end.

physical location for execution. With fusion enabled, Cloudflow will greedily fuse together all operators in a chain, but will optionally avoid fusing operators with different resource requirements (CPUs vs GPUs).

Competitive Execution Competitive execution is used to reduce the tail latency (usually 95th or 99th percentile) of operators with highly variable execution times. Executing multiple replicas of such operators in parallel significantly reduces perceived latency [131, 75, 126]. This is a straightforward dataflow rewrite in Cloudflow: we create redundant parallel replicas of the operator in question and add an `anyof` to consume the results. Then the runtime can choose the replica that completes first.

Cloudburst's DAG execution API was ill-suited to `anyof`: It waits for every upstream function in the DAG before executing a function (*wait-for-all* semantics). We modified Cloudburst to enable functions to execute in *wait-for-any* mode.

Operator Autoscaling and Placement Each operator in a dataflow may have different performance needs—memory consumption, compute requirements and so on. This heterogeneity is particularly pronounced in AI prediction serving. For example, an image processing flow might use a CPU-intensive preprocessing stage before a GPU-based neural net.

We seek two optimizations: per-operator autoscaling decisions, and the scheduling of each operator to run on well-chosen hardware. Continuing our example, if the preprocessing stage was serialized and slow, while the neural net stage was efficient and supported batching, it makes sense to deploy preprocessing functions onto many CPU nodes, and the neural net onto fewer GPU nodes.

Cloudflow's dataflow model makes it easy to autoscale dataflow pipelines in a fine-grained way. As described above, the Cloudflow compiler translates each operator into a separate Cloudburst function (unless fused). Like other FaaS systems, Cloudburst natively autoscales function individually: it adds and removes replicas as load (across many execution requests) changes.

However, Cloudburst—like most FaaS systems—does not support specialized hardware today. We extended Cloudburst's runtime to add support for GPUs in addition to regular CPU-based executors. Then we extended the Cloudburst API to allow functions to be annotated with different resource class labels, and enhanced the Cloudburst scheduler to partition its task pool by label. The default Cloudburst scheduling policy is applied within each class. It is interesting to consider new policies in this context, but beyond the scope of our work.

Data Locality via Dynamic Dispatch Data retrieval is an important part of many prediction pipelines. For example, a recommender system might look at a user's recent click history, then query a database for a set of candidate products before returning a set of recommended items. Ideally, we want to avoid fetching data over the network, as this can quickly become a latency bottleneck. Cloudburst, by default, performs locality-aware scheduling—it attempts to schedule computation on machines where the task's data dependencies are likely to be cached. However, its API requires that all of a request's data dependencies be *pre-declared*. This conflicts with the dynamic nature of many prediction pipelines—in the example above, the candidate product set is determined by the user's recent actions, so the Cloudburst scheduler will not know the request's dependencies *a priori*. Avoiding this pitfall requires both Cloudflow rewrites and a Cloudburst

modification.

We implement two Cloudflow rewrites. First, Cloudflow fuses each `lookup` operator with the operator downstream from it, so that the processing is colocated with the `lookup`. Second, Cloudflow rewrites `lookup` operators to enable Cloudburst to perform *dynamic dispatch* of the (fused) operator at a machine that has cached the column value. To support this, the column argument to `lookup` is converted to a reference ref that Cloudburst can process at runtime. The Cloudflow compiler then splits the dataflow just before the `lookup`, and generates *two* Cloudburst DAGs, flagging the first DAG with a *to-be-continued*(d, ref) annotation, where d is a pointer to the second DAG.

We then modified the Cloudburst runtime to support *to-be-continued* annotations. When a *to-be-continued* DAG finishes executing in Cloudburst, the result—rather than being returned to the user—is sent back to the Cloudburst scheduler along with a resolved ref (a KVS key) and the DAG ID d. Given the ref, the scheduler can place the DAG d on a machine where the KVS entry is likely to already be cached.

Batching Batching is a well-known optimization for prediction serving, taking advantage of the parallelism afforded by hardware accelerators like GPUs Many popular ML libraries like PyTorch offer APIs that perform a batch of predictions at once. Because our dataflow model is based on small request `Table`s, we augment our runtime to form batches across invocations of `execute`. To support this, Cloudflow's API provides a flag for the function arguments to `map` and `filter` to declare batch-awareness. We then modified the Cloudburst API to recognize that flag as we pass it down in compilation. Finally we modified the Cloudburst executor, so that when it is running a batch-enabled function, it dequeues multiple execution requests. and executes the entire batch in a single invocation of the function. The maximum batch size is configurable (defaults to 10). The executor demultiplexes the results, and the API returns results for each invocation separately.

Tradeoffs and Limitations It is worth noting that the optimizations we discuss here have clear tradeoffs. For example, operator fusion is at odds with fine-grained autoscaling: If a slow function and a fast function are fused, an autoscaler cannot selectively allocate more resources to each function. However, automated strategies for trading off these optimizations, such as a cost-based optimizer or a planner, are out of scope here. We focus on the architecture and mechanisms that enable optimized dataflows; we return to the idea of automated optimization in Section 4.6.

4.4 Evaluation

In this section, we present a detailed evaluation of Cloudflow. Section 4.4 studies each of the optimizations from Section 4.3 in isolation using synthetic workloads. We then compare Cloudflow against state-of-the-art industry and research systems on real-world pipelines. We ran all experiments on AWS in the `us-east-1a` availability zone. Cloudburst CPU nodes used `c5.2xlarge` instances (2 executors per machine); GPU nodes used `g4dn.xlarge` instances with Tesla T4 GPUs.

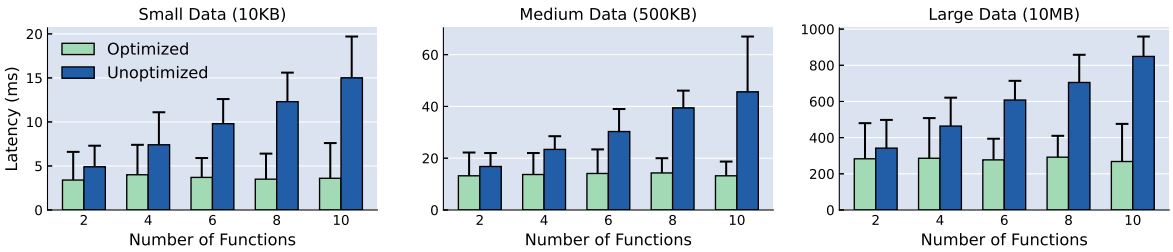


Figure 4.4: A study of the benefits of operator fusion as a function of chain length (2 to 10 functions) and data size (10KB to 10MB). We report median (bar) and 99th percentile (whisker) each configuration. In brief, operator fusion improves performance in all settings and achieves speedups of 3-5× for the longest chains of functions.

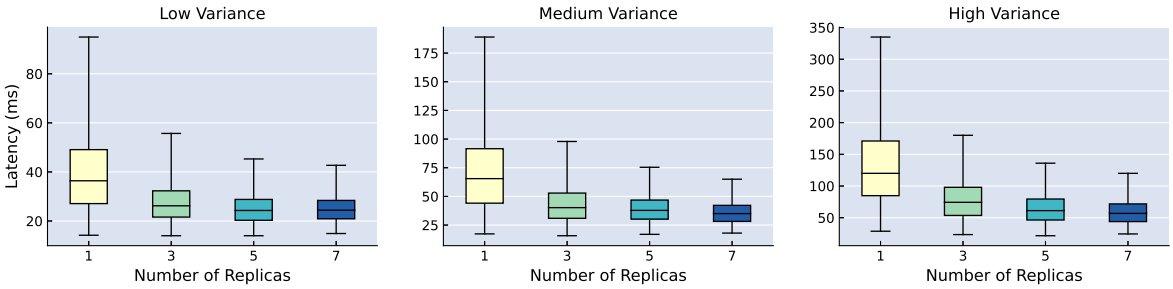


Figure 4.5: Latencies (1st, 25th, 50th, 75th, and 99th percentile) as a function of the number of additional replicas computed of a high-variance function. Adding more replicas reduces both median and tail latencies, especially for the high variance function.

Optimization Microbenchmarks

We evaluate each of the proposed optimizations in isolation on synthetic workloads.

Operator Fusion

We first study the benefits of operator fusion on linear chains of functions. The main benefit of fusion is avoiding the cost of data movement between compute locations. Correspondingly, this experiment varies two parameters: the length of the function chain and the size of data passed between functions. The functions themselves do no computation—they take an input of the given size and return it as an output. The output is passed downstream to the next function in the chain.

For each combination of chain length and size, we measure an optimized (fused) pipeline and an unoptimized pipeline. The fused pipelines execute all operators in a single Cloudburst function, while the unfused pipelines execute every stage in a separate Cloudburst function. Figure 4.4 reports median (bar) and 99th percentile (whisker) latencies for each setting.

As expected, for each input data size, the median latency of the optimized pipelines is roughly constant. The 99th percentile latencies have slightly more variation—generally expected of tail

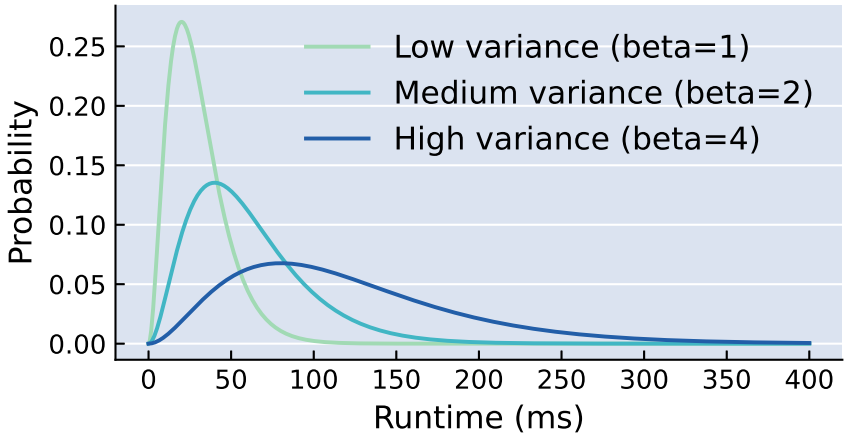


Figure 4.6: Three gamma distributions with different levels of variance from which we draw function runtimes for the experimental results shown in Figure 4.5.

latencies—but there is no discernible trend. The median latencies of the unoptimized pipelines increase linearly with the length of the function chain—primarily because the cost of data movement increases with the length of the chain. For smaller chains, we see that fusion only imporoves latencies by 20-40%, depending on data size; however, fusing longer chains leads to improvements up to 4×.

***Takeaway**: Operator fusion in* Cloudflow *yields up to a 4× latency decrease by avoiding the overheads of data serialization and data movement between function executors.*

Competitive Execution

We proposed competitive execution to reduce latencies for operators with variable runtimes. As discussed in Section 4.3, Cloudflow executes multiple replicas of a high-variance operator in parallel and selects the result of the replica that finishes first. Here, we construct a 3-stage pipeline; the first and third operators do no computation. The second function draws a sample from one of three Gamma distributions and sleeps for the amount of time returned by the sample. For each Gamma distribution the shape parameter is $k = 3$ and the scale parameter is $\theta \in \{1, 2, 4\}$ corresponding to low, medium, and high variances.

We expect that increasing the number of replicas, particularly for high variance distributions, will noticeably reduce tail latencies. Figure 4.5 shows our results—the boxes show the interquartile range, and the whiskers show the 1st and 99th percentiles. In all cases, increasing from 1 to 3 replicas reduces *all* latencies significantly: tail latencies decreased 71%, 94%, and 86% for low, medium, and high variances, and median latencies decreased 39%, 63%, and 62%.

Beyond 3 replicas, higher variance models see larger improvements. For the low variance dataflow, increasing from 3 to 7 replicas yields a 30% improvement in tail latency and only a 7%

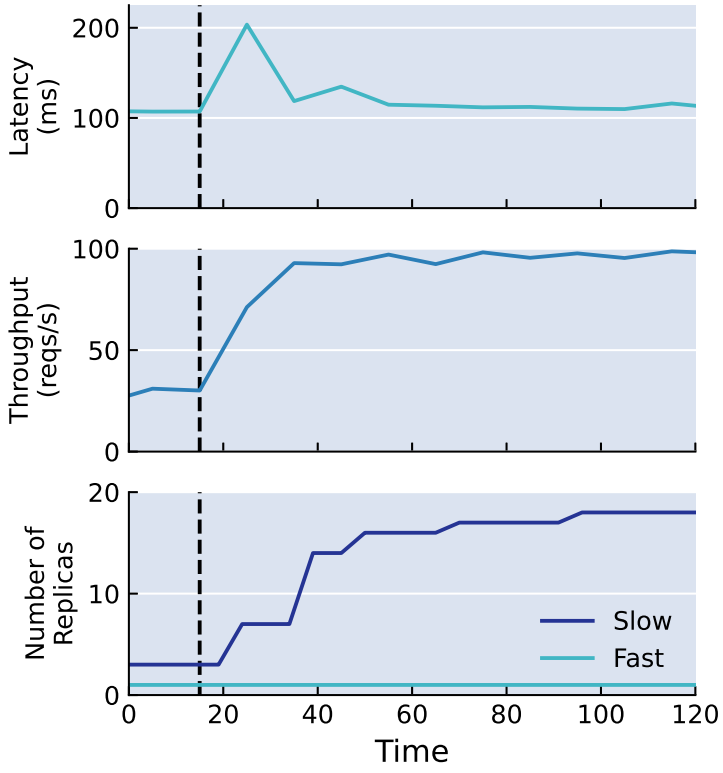


Figure 4.7: Median latency, throughput, and resource allocation in response to a load spike for a pipeline with a fast and a slow function. Cloudflow's dataflow model enables a fine-grained resource allocation in Cloudburst, and we scale up only the bottleneck (the slow function) without wasting resources on the fast function.

improvement at median. However, for the high variance setting, we observe a 50% improvement at the tail and a 31% improvement in median latency.

***Takeaway**: Increasing the number of replicas reduces both tail and median latencies, with particularly significant improvements for extremely highly-variable dataflows.*

Operator Autoscaling

In this section, we look at Cloudflow's and Cloudburst's ability to respond to load changes with fine-grained operator scaling. We study a workload with two functions—one fast and one slow—similar to the example from Section 4.3. We introduce a sudden load spike and measure latency, throughput, and resource allocation. Figure 4.7 shows our results. We begin with 4 client threads issuing requests simultaneously; latency and throughput are steady. There are 3 threads allocated to the slow function and 1 thread allocated to the fast function.

At 15 seconds, we introduce a 4× load spike. Latency immediately increases as the allocated

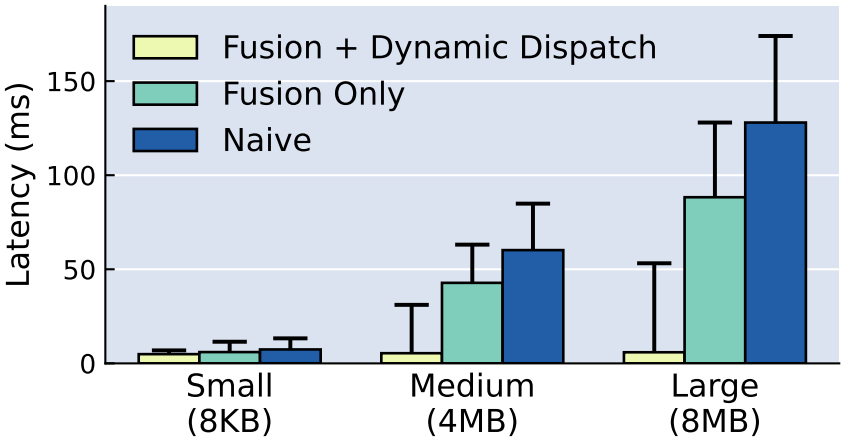


Figure 4.8: Median latency and 99th percentile latencies for a data-intensive pipeline on Cloudflow with the fusion and dynamic dispatch optimizations enabled, only fusion enabled, and neither enabled. The pipeline retrieves large objects from storage and returns a small result; Cloudflow's optimizations reduce data shipping costs by scheduling requests on machines where the data is likely to be cached. For small data, the data shipping cost is only a milliseconds, but for the medium and large inputs, Cloudflow's optimizations enable orders of magnitude faster latencies.

resources are saturated. The autoscaler responds by adding 16 replicas of the slow function over 15 seconds, and after time 40, latency returns to pre-spike levels (with a minor blip); throughput stabilizes at a new high. In the meantime, there is no change in the resource allocation of the fast function—it remains at 1 replica.

Over the remaining minute, the autoscaler adds 2 more replicas of the slow function to introduce slack into the system. This is because the existing resource allocation matches the incoming request rate exactly, and the autoscaler creates a small amount of excess capacity to account for potential future load spikes. There is no increase in throughput because the existing resources were sufficient to service the incoming request rate. This type of fine-grained resource allocation is especially useful for pipelines with heterogeneous resource requirements—a user would not want to scale up GPU allocation beyond necessary for a pipeline is bottlenecked on a CPU task.

***Takeaway**: Cloudflow's dataflow model allows for fine-grained resource allocation in the underlying serverless runtime, enabling nimble responses to load changes while ensuring efficient use of resources.*

Locality

Next, we look at the benefits of data locality. We picked a representative task in which we access each of a small set of objects (100) a few times (10) in a random order. The pipeline consists of a `map` that pick which object to access, a `lookup` of the object, and a second `map` that computes a result (the sum of elements in an array). We vary the size of the retrieved data from 8KB to 8MB. Figure 4.8 shows our results. In all settings, we warm up the Cloudburst's caches by issuing a

request for each data item once before starting the benchmark.

As described in Section 4.3, Cloudflow's locality optimization has two components: (1) fusing `lookup`s with downstream operators and (2) enabling dynamic dispatch to take advantage of Cloudburst's scheduling heuristics. We measure the incremental benefits of both these rewrites. The Naive bar Figure 4.8 implements neither optimization—data is retrieved from the KVS in the execution of the `lookup` operator and shipped downstream to the next operator. The Fusion Only bar merges the `lookup` with the second `map` but does not use dynamic dispatch. The Fusion + Dispatch bar uses both optimizations.

For small data sizes, our optimizations make little difference—the cost of shipping 8KB of data is low, and the Naive implementation is only 2.5ms slower than having both optimizations implemented. As we increase data size, however, Naive performance significantly worsens—for each request, data is moved once from the KVS to the `lookup` operator and again from the `lookup` to the second `map`. The Fusion Only operator avoids *one* hop of data movement (between operators), but relies on random chance to run on a machine where its input data is cached—this does not happen very often. With the dynamic dispatch optimization implemented, Cloudflow takes advantage of locality-aware scheduling and for the largest data size (8MB) is 15× faster than Fusion Only and 22× faster than Naive. Tail latencies, however, do increase noticeably with data size, as the requests that incur cache misses will still pay data shipping costs.

Takeaway: Cloudflow *'s locality optimizations enable it to avoid data shipping costs and lead to an order of magnitude improvement in latencies for non-trivial data accesses.*

Batching

Finally, we look at the benefits of batching. In this experiment, we introduce GPUs, since batching primarily benefits with highly parallel hardware. We execute a pipeline with a single model (the ResNet-50 [52] image classification model in PyTorch) and no other operators. Enabling batching required changing only two lines of user code—using `torch.stack` to combine inputs into a single tensor. Our experiment varies the batch size from 1 to 40 in increments of 10. We asynchronously issue k requests (where k is the batch size) from a single client in order to control the batch size and measure the time until all results are returned. Figure 4.9 reports latency (on a log-scale) and throughput for each batch size.

At baseline (batch size 1), the GPU has a roughly 4× better latency and throughput than the CPU. Increasing the batch size for the CPU from 1 to 10 yields a 20% increase in throughput (18 to 22 requests) with an 8× increase in latency. Beyond that point, latency increases linearly and throughput plateaus—standard CPUs do not support parallel execution.

On a GPU, we see a 4.5× jump in latency and 2.2× increase in throughput from size 1 to 10. Between 10 and 20, we see a further 70% increase in latency with only an 18% increase in throughput. Past size 20, the GPU is saturated; throughput plateaus and latency increases linearly with batch size. Note that between batch sizes 1 and 20, there is only an 8× increase in latency (1̃2ms to 100ms)—well within the bounds for an interactive web application [53]—while significantly improving the throughput (3×).

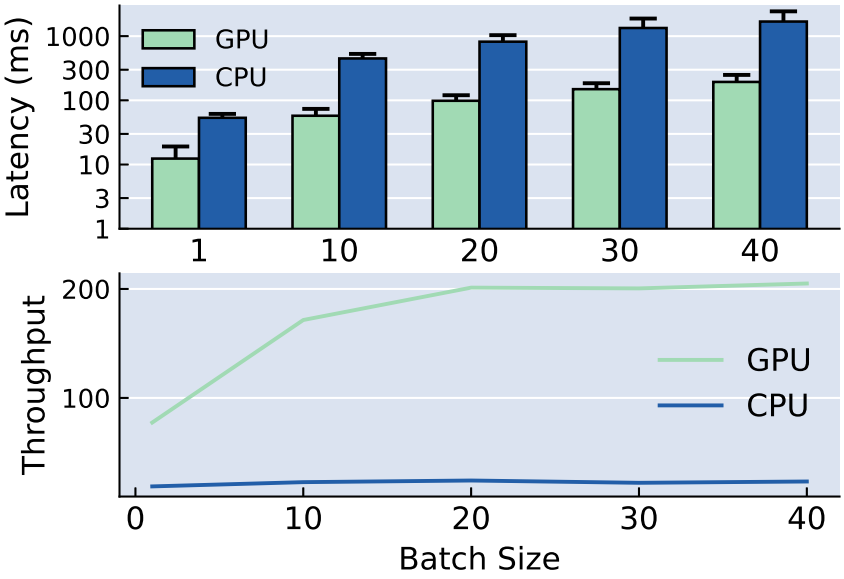


Figure 4.9: A comparison of CPUs and GPUs on Cloudflow, measuring latency and throughput while varying the batch size for the ResNet-101 computer vision model.

***Takeaway**: Cloudflow's batching optimization enables a 3× increase in throughput while retaining interactive latency.*

Prediction Serving Pipelines

Having measured Cloudflow in isolation, we now look at real prediction serving pipelines and compare Cloudflow to state-of-the-art commercial and research systems: AWS Sagemaker and Clipper [25]. AWS Sagemaker is a hosted, end-to-end platform for managing machine learning models that includes a containerized deployment tool. We selected it as a representative of a number of hosted model management tools with prediction serving capabilities. Clipper is an open-source research platform that automatically containerizes and deploys models while maintaining strict latency goals. We measure Cloudflow against both systems on four real world pipelines: an image cascade, a video processing pipeline, a neural machine translation task, and a recommender system.

For each pipeline, we sampled multiple optimization strategies on Cloudflow. Due to space limits, we do not describe results from each configuration; we limit ourselves to the optimization strategies that worked best and the corresponding results. We return to automating the optimzation process in Section 4.6.

Pipelines

Image Cascade. A cascade pipeline (described in Section 4.2) chains together models of increasing complexity, executing later models only if the earlier models are unsure of their predictions. We use a cascade with two models: ResNet-101 [52] and Inception v3 [127]. We preprocess an input image and execute ResNet; if ResNet's confidence is below 85%, we then execute Inception and pick the result with the higher confidence. We run this pipeline with a random sample of images from the ImageNet dataset [112].

Video Streams. Processing video streams is an extremely data- and compute-intensive task. We use a standard video processing flow [146] that uses YOLOv3 [106] to classify the frames of a short video clip (1 second). Based on the YOLO classifications, we select frames that contain people and vehicles (representative of a pipeline for an autonomous vehicle) and send those frames to one of two ResNet-101 models. The two ResNet models are executed in parallel and more specifically classify people and vehicles, respectively. We union the results of the two models, group by the ResNet classifications, and `count` the occurrences of each class per input clip.

Neural Machine Translation. Models that translate text between languages are becoming increasingly widely used (e.g., chat applications, social networks). These models are also notoriously large and computationally intensive. We model a translation task by first classifying an input text's language using fastText, an open-source language modeling library [67, 68]. We limit our inputs to French and German. Based on the language classification, we then feed a second input into one of two models that translates between English and the classified language. The translation models are implemented in the PyTorch FAIRSEQ library [99] based on sequence learning translation models [44].

Recommender System. Recommender systems are ubiquitous among internet services. Facebook recently published a case study of their DNN-based personalized recommenders [46]; we implement a pipeline modeled after their architecture. A request comes in with a user ID and recent items the user has clicked on; based on the set of recently clicked items, we generate a product category recommendation. The pipeline then computes a top-k set of items from that product category, using the user's pre-calculated weight vector. This is done with a simple matrix

```
flow = cloudflow.Dataflow([('img', numpy.ndarray)])
img = flow.map(preprocess)
resnet = img.map(resnet_101)
inception = resnet.filter(low_confidence).map(inception_v3)
label = resnet.join(inception).map(pick_best_prediction)

category_counts = people.union(vehicles).groupby('category').agg('count')
label = category_counts

flow.output = label
```

Figure 4.10: The Cloudflow implementation of the image cascade pipeline.

multiplication-based scoring scheme. While the user vectors are small (length 512), the product category sets can be large (roughly 10MB)—as a result, enabling locality and minimizing data movement is an important part of implementing these pipelines efficiently.

Benchmark Setup

Cloudflow. For each pipeline, we start with one replica per operator and run a 200 request warm-up phase that allows the Cloudburst autoscaler to settle on a resource allocation. We then run 1,000 requests from 10 benchmark clients in parallel and report median latency, 99th percentile latency, and throughput below. Unless stated otherwise, we copied the exact resource allocation from Cloudflow to each of the other systems.

Sagemaker. Sagemaker requires users to provide a Docker container per pipeline stage—a cumbersome and time-consuming process. We wrap each pipeline stage in a bespoke container and deploy it as a Sagemaker endpoint. We do not rely on Sagemaker's built-in inference pipelines feature, as it does not support parallelism, which is required in our workloads. Instead, we built a proxy service that manages each client request as it passes through the pipeline. This enables us to invoke multiple endpoints in parallel when necessary. CPU workers in Sagemaker used `ml.c5.2xlarge` instances, and GPU workers used `ml.g4dn.xlarge` instances.

Clipper. Clipper is a state-of-the-art research system that, similar to Sagemaker, deploys models as microservices; Clipper does not support pipelines in any form. Similar to our Sagemaker deployments, we use custom code to move each request through the pipeline. Deploying models

```
flow = cloudflow.Dataflow([('frame', numpy.ndarray)])
yolo_preds = flow.map(yolov3)
people = yolo_preds.map(resnet101_people)
vehicle = yolo_preds.map(resnet101_vehicles)
category_counts = people.union(vehicles).groupby('category').agg('count')
label = category_counts

flow.output = label
```

Figure 4.11: The Cloudflow implementation of the video stream processing pipeline.

```
flow = cloudflow.Dataflow([('english', str), ('other', str)])
language_type = flow.map(yolov3)
french = language_type.filter(is_french).map(translate_from_french)
german = language_type.filter(is_german).map(translate_from_german)
result = german.join(french)

flow.output = result
```

Figure 4.12: The Cloudflow implementation of the neural machine translation pipeline.

using Clipper's API was easier than with Sagemaker, but certain pipelines with bespoke dependencies required us to build custom Docker containers. The simplest way to deploy model replicas on Clipper was to create multiple replicas of the model on a large machine—however, we ensured that the number of resources (vCPUs, RAM, and GPUs) per worker was the same as on Cloudflow and Sagemaker. Note that Clipper also supports batching, which we enabled for GPU workloads.

Results

We report results for each pipeline in Section 4.4 on both CPU and GPU deployments, except for the recommender system, which we only run on CPU. We omit GPU results for this pipeline because, as noted by [46], recommendation models have low arithmetic intensity and do not usually benefit from GPU acceleration. Unless stated otherwise, we enabled batching on Cloudflow and Clipper for all GPU workloads and disabled it for all CPU workloads, as per our results in Section 4.4.

Image Cascade. The leftmost graph in Figure 4.14 shows results for the image cascade. We enabled Cloudflow's fusion optimization and merged the whole pipeline into a single operator. Despite having a mix of CPU and GPU operators in the pipeline, we found that the CPU execution costs were low (10-15ms) and that reducing data movement best improved Cloudflow's performance.

On both CPU and GPU deployments, Cloudflow has a roughly 2× better median latency than the other systems. This translates into a 2× increase in throughput for both hardware configurations. These improvements are largely due to reduced data movement costs enabled by operator fusion. We also note the benefits of batching by comparing Clipper and Sagemaker's performance. In CPU mode where neither system uses batching, Sagemaker outperforms Clipper by 20%; however, in GPU mode, where Clipper supports batching and Sagemaker does not, Clipper is able to close the performance gap and *match* Sagemaker's throughput.

Video Streams. The video streaming pipeline was the most data intensive: Each video input had 30 frames, which was about 20MB of data. In this pipeline, there were only GPU models (YOLOv3 and ResNet-101) without any CPU stages—we again apply operator fusion to the whole pipeline to generate a single Cloudburst function. We found that the cost of executing the two parallel ResNet instances in serial was lower than the cost of shipping the input over the network.

```
flow = cloudflow.Dataflow([('uid', int), ('recent_clicks', numpy.ndarray)])
topk_for_user = flow.lookup('uid', type='column_ref') \
                    .map(pick_category) \
                    .lookup('category', type='column_ref')
                    .map(get_topk)

flow.output = topk_for_user
```

Figure 4.13: The Cloudflow implementation of the recommender system pipeline.

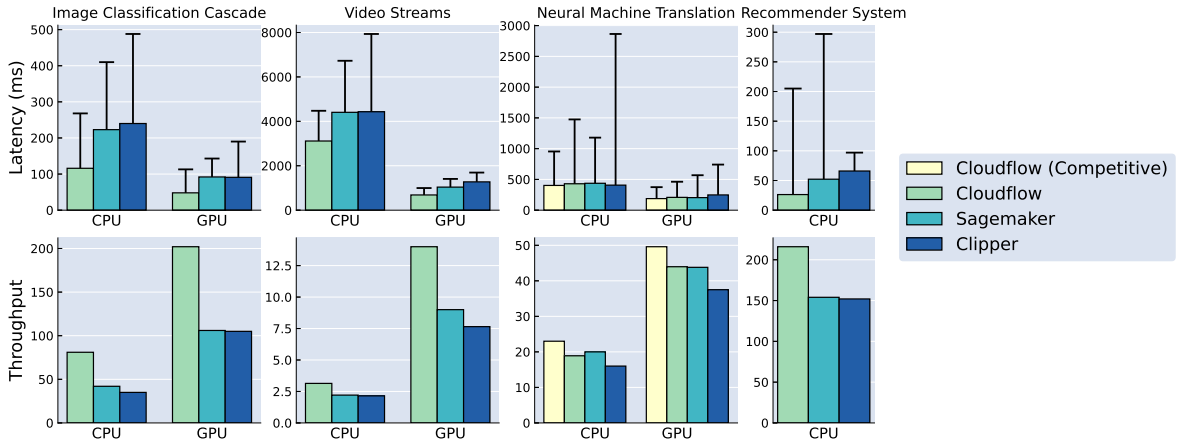


Figure 4.14: Latencies and throughputs for each of the four pipelines described in Section 4.4 on Cloudflow, AWS Sagemaker, and Clipper.

CPU deployments are expectedly slow—Cloudflow's median latency was 3.1 seconds, while Sagemaker's and Clipper's were around 4.4 seconds; tail latencies were 4.4, 6.7, and 7.9 seconds respectively. Cloudflow's reduced data movement enables significant improvements (41% at median and up to 80% at the 99th percentile) but was still capped at 3.1 requests per second.

Results are much more promising for GPU deployments. Cloudflow's median latency was 685ms, and its 99th percentile latency was about 1 second (996ms)—meaning that Cloudflow can consistency process video in real-time. Sagemaker and Clipper were both slower, with median latencies of 1 and 1.3 seconds, respectively, and 99th percentile latencies at 1.4 and 1.7s seconds. Cloudflow is able to process 14 requests per second, while the other systems were both below 9 requests a second.

Neural Machine Translation. The neural machine translation task was the best case scenario for both Clipper and Sagemaker as it had the least input data (strings to translate) while being very computationally intensive. We found that the NMT models—particularly on CPU—had high variance in runtimes, so we enabled competitive execution in Cloudflow. However, since competitive execution required allocating extra resources not available to Sagemaker and Clipper, we report Cloudflow measurements with and without competitive execution.

On CPUs, the median latencies for all configurations are similar—Clipper was about 7% faster than Sagemaker and Cloudflow without competition. At the tail, Clipper was the slowest with a 99th percentile of 2.8 seconds; Cloudflow's tail latency was 1.5 seconds, and Sagemaker's was 1.2 seconds. Despite a 25% lower tail latency, Sagemaker's throughput was only 5% higher than Cloudflow's. Adding two competitive replicas reduced Cloudflow's median latency by 7% and its 99th percentile by over 50%. This led to a 20% improvement in throughput compared to Cloudflow without competition and 15% improvement over Sagemaker.

On the GPU deployment, Cloudflow without competition and Sagemaker had similar performance. Clipper's median latency was about 20% higher, likely because its scheduler constructs

batches more aggressively than Cloudflow does. Cloudflow without competition and Sagemaker had the same throughput, while Clipper's was 15% lower. Introducing competitive GPU replicas lowered median latencies by 10% and lowered tail latency by almost 25% relative to Cloudflow without competition (and by over 50% relative to Sagemaker). This translated into a 13% improvement in throughput.

Recommender System. Finally, we look at the recommender pipeline. This workload was different from the others because it was data-intensive and computationally light—each request required only a matrix multiplication followed by a few simple operations. The user weight vectors and product categories in this pipeline are pre-generated. That provides an opportunity for us to exercise Cloudflow's locality aware scheduling.

Running on Cloudburst allows us to automatically access data in Anna; however, Sagemaker blocks inbound network connections, which Anna requires, as it is an asynchronous KVS. As a workaround we used AWS ElastiCache's hosted Redis offering, which has similar performance to Anna [140], as the storage engine for Clipper and Sagemaker. To simulate the caches in Cloudburst, we also added in-memory caches of size 2GB to Sagemaker and Clipper, that allowed them to cache user weight vectors and product categories—however, these systems do not have Cloudflow's dynamic dispatch, so the likelihood of a cache miss is much higher. We pre-generate 100,000 user weight vectors (length 512, 4KB each) and 1,000 product categories (2,500 products per category, 10MB each) and put them in storage. For each request, we input a random user ID and a random set of recent clicks—this translates to a random product category per request.

Note that an alternative solution for Sagemaker and Clipper would have been to pre-load the *whole* model (user vectors and product categories) into memory at container start. We did not implement this solution for two reasons. First, it is incredibly memory inefficient—our modest dataset was 14GB, but for most access patterns (e.g., random, Zipfian), a large majority of that data will be unused. Second, it is inelastic: Loading the whole model from S3 takes over 2 minutes, making it expensive to scale the system up.

Cloudflow's locality optimizations lead to a 2× improvement in median latency over Sagemaker's and 2.5× improvement over Clipper's. Cloudflow reliably places requests on machines with cached inputs, whereas Sagemaker and Clipper are much less likely to see cache hits. Tail latencies for all systems are high due to the cost of retrieving data over the network Nonetheless, the lower median latencies translate to a roughly 40% increase in throughput over Sagemaker and Clipper.

***Takeaway**:Cloudflow's optimizations enable it to outperform state-of-the-art prediction serving systems up to 2× on real-world tasks; importantly, it is able to meet real-time latency goals for a data- and compute-intensive task like video streaming.*

4.5 Related Work

Dataflow Abstractions. Dataflow abstractions have deep roots in both software and hardware architectures (e.g., [32, 33]). In the last decade, dataflow emerged as a core programming ab-

straction for systems such as Spark [144] and Dryad [61] that excel in high throughput big data processing. Other systems like Naiad [94], Flink [20], and Noria [45, 114] implement stateful dataflow that targets scenarios where the input data continuously streams into the system. Cloudflow, in comparison, is designed for handling interactive model serving queries at low latency, and the input of each query is bounded. In addition to standard dataflow optimizations such as operator fusion, Cloudflow implements a variety of custom optimizations that are well-suited to interactive model serving, such as competitive execution and fine-grained autoscaling.

Packet Processing and Distributed Systems. Programmable packet processors for networks share our latency focus, but target much less compute-intensive tasks than model serving, using lower-level optimizations. The Click router [74] is an influential example; subsequent work in declarative networking [86] and packet processing languages [16] provide higher-level abstractions that often compile down to dataflow. Similar languages have targeted distributed systems [5, 90, 92], with a focus on orthogonal concerns like coordination avoidance.

Lightweight Event Flows. Systems such as SEDA [137] adapt the resource allocation of different stages of a pipeline based on demand, but do not have a semantic notion of the tasks' meanings and thus cannot perform logical rewrites of the pipeline. Additionally, SEDA is focused on single-server rather than distributed executions and has no autoscaling.

ML Optimizations. TASO [63] and ParM [75] introduce optimizations on individual models that can be integrated into prediction serving systems. TASO performs low-level rerwites of DNN computation graphs to improve model performance. ParM reduces the tail latencies and enables failure recovery for variable models by using erasure coding to reconstruct slow and failed predictions. Neither system offers end-to-end optimizations similar to Cloudflow's.

Prediction Serving Systems. Clipper [25], TFServe [12], Azure ML [130], and AWS Sagemaker either have no support for pipelines or impose restrictions on the topology of the pipelines. As such, none of these systems perform pipeline optimizations as in Cloudflow. Other systems such as Inferline [26] and Pretzel [81] are complementary to Cloudflow; Inferline focuses on efficient scheduling to meet latency goals, and Pretzel exploits white-box pipelines to optimize individual stages.

4.6 Conclusion and Takeaways

Cloudflow is a framework for serverless prediction serving pipelines that offers a familiar dataflow programming API, allowing developers to easily construct prediction pipelines in a few lines of code. We transparently implement two kinds of optimizations over the resulting dataflow graph: dataflow rewrites and dataflow-to-FaaS compilation. Cloudflow deploys pipelines on top of Cloudburst, and we extended Cloudburst with key features to support our workloads. Our evaluation shows that Cloudflow's automated optimizations enable us to outperform state-of-the-art prediction serving systems by up to 2× for real-world tasks like image cascades and video processing.

The early promise of dataflow for modern, compute-intensive applications points to an exciting future for using serverless dataflow for a variety of applications. Many modern applications—not just prediction serving—can be adapted to fit this mold. For example, a social media application generating a news feed might receive a REST request, use a model to generate its news feed, retrieve posts from a database (e.g., using a `lookup`), render HTML, and return a webpage to the user—all of which could be written using the operators detailed here. There is a large space of research to be explored here, simplifying the development and deployment of modern applications—and simultaneously improving performance—with low-latency serverless dataflow.

Chapter 5

Discussion and Lessons Learned

Cloud computing has been the defining computing infrastructural innovation of this century. The push-button availability of resources at large scales has led to the success of companies providing and improving upon cloud services and has also enabled seminal systems research. In many ways, the most immediate success of the cloud has been the democratization of access to resources. Any developer can rent a handful of servers for their application and pay a few dollars a day rather than having to pay thousands of dollars up front, manually install the servers, and manage networking and cooling infrastructure—to name just a few concerns.

As we have already discussed, however, the model of renting resources or fixed-deployment services from cloud providers also has its limitations, including cumbersome and slow deployment processes, manual resource allocation, and resource inefficiency. As we think about the next decade, we must simplify these processes in order to enable the millions of new programmers who don't have deep experience in systems infrastructure—data scientists, social scientists, or web developers for example.

The advent of serverless computing has begun to address some of these challenges. The model simplifies the lives of developers while giving cloud providers the needed flexibility to achieve higher utilization. From this perspective, serverless computing has the opportunity to be the default mode in which programmers interact with the cloud, focusing on the code to achieve their task without worrying about servers, consistency, or fault-tolerance. However, existing FaaS systems have stopped short of enabling truly general purpose cloud programming.

The research in this dissertation improves the state-of-the-art in one direction: designing serverless infrastructure that supports state management natively, rather than as an afterthought. We have shown orders-of-magnitude improvements in performance over existing FaaS systems while providing stronger consistency guarantees and supporting compute-intensive real-world applications. This represents a significant advancement over the state-of-the-art, but there is significant work left to be done if serverless computing is to become the cloud default. The work here has primarily focused on applying systems and database techniques to serverless computing, but there are opportunities for a much broader set of research directions, inspired by techniques from programming language, AI, and economics. We briefly highlight a few of them here.

Efficient Autoscaling Policies. Throughout this work, we have made a point of separating

out the autoscaling mechanisms from policies, and we have paid minimal attention to policies. However, to make serverless infrastructure (and the systems described here) more generally applicable, the correct policy (or policies) for autoscaling must be carefully considered. Different applications will likely have different characteristics and varying needs, which means truly general infrastructure might have pluggable policies—however, this raises thorny questions around who bears the cost burden of bad autoscaling decisions. A user can't be allowed to over-allocate resources and only be billed for requests run—the cloud provider would lose money on allocated, "free" resources. But if a user is billed for resources allocated, we return to a serverful world.

Perhaps, domain specific infrastructure with stronger assumptions and better semantic knowledge of workloads will emerge on top of general purpose serverless infrastructure. The lowest, general-purpose layer will provide pluggable policies with sensible defaults. Meanwhile, the intermediary layers will have better visibility into application needs and will be able to make smarter policy decisions—once more reinforcing the need for separated policies and mechanisms.

Higher Level APIs. Commercial FaaS systems and Cloudburst both provide extremely general APIs—the user brings arbitrary code, and the infrastructure executes it without any notion of what the program might be doing. Cloudburst optionally allows users to provide visibility into data access patterns, which are used to improve performance, but pre-declaration is cumbersome and, as we discussed in Chapter 4, not always possible. A higher level API, like Cloudflow's dataflow operators, allows more visibility into program semantics and allows the system to optimize programs. Cloudflow is just one example of such an API, and we believe others will emerge as the area matures.

There is, however, a concerning pitfall here, highlighted by Jonas et al. [65]. In recent years, cloud providers have begun to offer domain-specific serverless offerings—like Google Cloud Dataflow (serverless bulk-synchronous processing) or AWS Athena (serverless SQL queries). If this model becomes standard, it will stifle the innovation of general-purpose serverless systems because users will default to using the domain-specific cloud system. In that world, however, only cloud providers could introduce new serverless infrastructure. Independent researchers with new ideas would have to construct their application from the virtual machine up, rather than taking advantage of an existing, general-purpose system. The success of the serverless paradigm is contingent on the continued development and use of general-purpose systems—on top of which domain-specific APIs can build (e.g., Cloudflow and Cloudburst)—rather than vertically integrated offerings.

Pricing & Marketplaces. Existing FaaS systems allow users to select the amount of RAM allocated to a function—this determines the CPU timeslice given to that function as well as the network bandwidth it can use [135]. These restrictions will be loosened as serverless infrastructure matures—it would be unreasonable for a general-purpose system to have fixed ratios of RAM, CPU, and network bandwidth, not to mention disk space or hardware accelerators. For comparison, there are over 200 different EC2 instance types, each of which has different resource configurations. And as datacenters become disaggregated, it will be easier for applications to acquire only the resources they need [101].

This, however, is not as straightforward as it sounds. One solution would be to have users

select the quantity of each resource—which again reintroduces many of the configuration and scaling challenges present in non-serverless infrastructure. Users would be buying serverless servers, an oxymoron at best. Allocating resources based on need and use raises interesting questions around the timescale of resource acquisition and how it is priced. Consider memory usage: If a user doesn't allocate a fixed amount of RAM, who is responsible for deciding what data is cache-resident, and what data is written to slow storage? Either the user must explicitly pin and unpin memory objects, or the cloud provider must provide a caching policy, both of which can lead to poor performance depending on application variability. The incentive structure around these decisions must be structured such that neither party is able to exploit the other, leading to mismanaged prices or poor performance.

Lessons Learned

Serverless computing has undoubtedly become more popular in the last few years, particularly in academia. When our attention was first drawn to serverless computing in 2017, there were fewer than 50 published papers on the subject at the time [29]—in 2019, there were over 130, and 2020 is on pace for close to 200 such papers. Our first attempts at publishing in the space were met with significant criticism—complaints that our work was mostly engineering and that we had no novelty, repackaging old ideas into a new form both with and without the serverless label. That stance has quickly changed, and the consensus today seems to be that porting an existing system or feature to a serverless system *is* a contribution, given the characteristics of serverless systems and the requirements of the ecosystem.

More broadly, this is reflective of the incentives in academia. There is a high premium on novelty in publishing research, which means that research needs to be distinguishable from past ideas and implementations, particularly in systems work. This was a point of much criticism for our work. Many of the ideas in this thesis—for example, the notion of read atomicity—were not inherently novel, but were well-studied ideas applied in a novel context. I believe the distinction is not important: Applying an existing idea in a new space is worth exploring because it demonstrates the success of the idea and can likely enable new applications. As Mark Twain said, "There is no such thing as a new idea. It is impossible. We simply take a lot of old ideas and put them into a sort of mental kaleidoscope..." [132].

The other implication of the need to explicitly distinguish work is the focus—sometimes to the detriment in the research—on quantifiable gains, usually in terms of performance. That focus is reflected in much of the work here. The primary contribution of systems like Cloudburst and Cloudflow is that they improve performance by some large margin over the existing state-of-the-art. Performance is undoubtedly important and should not be neglected, but it is not the only metric that matters.

Equally important is the experience of the user, which can be seen in the increasing popularity of serverless computing in general. This can take many forms—removing consistency anomalies (as AFT does) is a form of improved user experience because the developer does not have to spend hours debugging missing or inconsistent data. Or better yet, a system like Spark [145] leveraged an existing set of familiar APIs to significantly simplify the development of programs

previously written in MapReduce (while also, of course, improving performance). User studies are not common in systems research and are generally not prioritized, but it is also received wisdom that the fastest system in the world with an unuititive API will not garner many users. Perhaps the usability of systems should receive as much attention as their performance.

The performance of serverless computing has been—and will continue to be—a major focus in the research community. However, that focus should not come at the expense of maintaining or improving the usability of these systems, which is often a primary attraction for users. In order to make serverless computing the future of the cloud, the new developers—the data scientists and social scientists—whom we target as users should continue to enjoy the benefits of simplified operations and autoscaling without having to program against complex APIs. Whether or not system developers are able to strike this balance will determine the scope of the impact of serverless infrastructure.

Bibliography

[1] Martìn Abadi et al. “Tensorflow: A system for large-scale machine learning”. In: *12th {USENIX} Symposium on Operating Systems Design and Implementation ({OSDI} 16)*. 2016, pp. 265–283.

[2] *Apache Airflow*. https://airflow.apache.org.

[3] Deepthi Devaki Akkoorath et al. “Cure: Strong semantics meets high availability and low latency”. In: *2016 IEEE 36th International Conference on Distributed Computing Systems (ICDCS)*. IEEE. 2016, pp. 405–414.

[4] Istemi Ekin Akkus et al. “SAND: Towards High-Performance Serverless Computing”. In: *2018 USENIX Annual Technical Conference (USENIX ATC 18)*. 2018, pp. 923–935.

[5] Peter Alvaro et al. “Consistency Analysis in Bloom: a CALM and Collected Approach.” In:

[6] Gennady Antoshenkov. “Dynamic query optimization in Rdb/VMS”. In: *Proceedings of IEEE 9th International Conference on Data Engineering*. IEEE. 1993, pp. 538–547.

[7] Baruch Awerbuch. “Optimal distributed algorithms for minimum weight spanning tree, counting, leader election, and related problems”. In: *Proceedings of the nineteenth annual ACM symposium on Theory of computing*. ACM. 1987, pp. 230–240.

[8] *AWS Lambda - Case Studies*. https://aws.amazon.com/lambda/resources/customer-case-studies/.

[9] Peter Bailis et al. “Scalable Atomic Visibility with RAMP Transactions”. In: *Proceedings of the 2014 ACM SIGMOD International Conference on Management of Data*. SIGMOD ’14. Snowbird, Utah, USA: ACM, 2014, pp. 27–38. ISBN: 978-1-4503-2376-5. DOI: 10.1145/2588555.2588562. URL: http://doi.acm.org/10.1145/2588555.2588562.

[10] Ioana Baldini et al. “Serverless computing: Current trends and open problems”. In: *Research Advances in Cloud Computing*. Springer, 2017, pp. 1–20.

[11] Joel Bartlett, Jim Gray, and Bob Horst. “Fault tolerance in tandem computer systems”. In: *The Evolution of Fault-Tolerant Computing*. Springer, 1987, pp. 55–76.

[12] Denis Baylor et al. “Tfx: A tensorflow-based production-scale machine learning platform”. In: *Proceedings of the 23rd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*. 2017, pp. 1387–1395.

[13] Paul Beame, Paraschos Koutris, and Dan Suciu. "Skew in parallel query processing". In: *Proceedings of the 33rd ACM SIGMOD-SIGACT-SIGART symposium on Principles of database systems*. ACM. 2014, pp. 212–223.

[14] Philip A. Bernstein and Nathan Goodman. "Multiversion Concurrency Control&Mdash;Theory and Algorithms". In: *ACM Trans. Database Syst.* 8.4 (Dec. 1983), pp. 465–483. ISSN: 0362-5915. DOI: 10.1145/319996.319998. URL: http://doi.acm.org/10.1145/319996.319998.

[15] Simone Bianco et al. "Benchmark analysis of representative deep neural network architectures". In: *IEEE Access* 6 (2018), pp. 64270–64277.

[16] Pat Bosshart et al. "P4: Programming protocol-independent packet processors". In: *ACM SIGCOMM Computer Communication Review* 44.3 (2014), pp. 87–95.

[17] Matthias Brantner et al. "Building a Database on S3". In: *Proceedings of the 2008 ACM SIGMOD International Conference on Management of Data*. SIGMOD '08. Vancouver, Canada: ACM, 2008, pp. 251–264. ISBN: 978-1-60558-102-6. DOI: 10.1145/1376616.1376645. URL: http://doi.acm.org/10.1145/1376616.1376645.

[18] Sebastian Breß and Gunter Saake. "Why it is time for a HyPE: A hybrid query processing engine for efficient GPU coprocessing in DBMS". In: *Proceedings of the VLDB Endowment* 6.12 (2013), pp. 1398–1403.

[19] E. Brewer. "CAP twelve years later: How the "rules" have changed". In: *Computer* 45.2 (Feb. 2012), pp. 23–29. ISSN: 0018-9162. DOI: 10.1109/MC.2012.37.

[20] Paris Carbone et al. "Apache flink: Stream and batch processing in a single engine". In: *Bulletin of the IEEE Computer Society Technical Committee on Data Engineering* 36.4 (2015).

[21] Tushar D Chandra, Robert Griesemer, and Joshua Redstone. "Paxos made live: an engineering perspective". In: *Proceedings of the twenty-sixth annual ACM symposium on Principles of distributed computing*. ACM. 2007, pp. 398–407.

[22] Michael Coblenz et al. "Exploring language support for immutability". In: *Proceedings of the 38th International Conference on Software Engineering*. ACM. 2016, pp. 736–747.

[23] Neil Conway et al. "Logic and Lattices for Distributed Programming". In: *Proceedings of the Third ACM Symposium on Cloud Computing*. SoCC '12. San Jose, California: ACM, 2012, 1:1–1:14. ISBN: 978-1-4503-1761-0. DOI: 10.1145/2391229.2391230. URL: http://doi.acm.org/10.1145/2391229.2391230.

[24] Daniel Crankshaw et al. "Clipper: A Low-Latency Online Prediction Serving System". In: *14th USENIX Symposium on Networked Systems Design and Implementation (NSDI 17)*. Boston, MA: USENIX Association, 2017, pp. 613–627. ISBN: 978-1-931971-37-9. URL: https://www.usenix.org/conference/nsdi17/technical-sessions/presentation/crankshaw.

[25] Daniel Crankshaw et al. “Clipper: A low-latency online prediction serving system”. In: *14th {USENIX} Symposium on Networked Systems Design and Implementation ({NSDI} 17)*. 2017, pp. 613–627.

[26] Daniel Crankshaw et al. “InferLine: ML Inference Pipeline Composition Framework”. In: *CoRR* abs/1812.01776 (2018). arXiv: 1812.01776. URL: http://arxiv.org/abs/1812.01776.

[27] Natacha Crooks et al. “Tardis: A branch-and-merge approach to weak consistency”. In: *Proceedings of the 2016 International Conference on Management of Data*. ACM. 2016, pp. 1615–1628.

[28] Abhinandan Das, Indranil Gupta, and Ashish Motivala. “Swim: Scalable weakly-consistent infection-style process group membership protocol”. In: *Proceedings International Conference on Dependable Systems and Networks*. IEEE. 2002, pp. 303–312.

[29] *dblp: Search for serverless*. https://dblp.uni-trier.de/search?q=serverless.

[30] *Amazon DynamoDB Transactions: How It Works - Amazon DynamoDB*. https://docs.aws.amazon.com/amazondynamodb/latest/developerguide/transaction-apis.html.

[31] Jeffrey Dean and Sanjay Ghemawat. “MapReduce: simplified data processing on large clusters”. In: *Communications of the ACM* 51.1 (2008), pp. 107–113.

[32] Jack B Dennis. “First version of a data flow procedure language”. In: *Programming Symposium*. Springer. 1974, pp. 362–376.

[33] Jack B Dennis and David P Misunas. “A preliminary architecture for a basic data-flow processor”. In: *Proceedings of the 2nd annual symposium on Computer architecture*. 1974, pp. 126–132.

[34] *Enterprise Application Container Platform | Docker*. https://www.docker.com.

[35] Jose M. Faleiro and Daniel J. Abadi. “Rethinking Serializable Multiversion Concurrency Control”. In: *Proc. VLDB Endow.* 8.11 (July 2015), pp. 1190–1201. ISSN: 2150-8097. DOI: 10.14778/2809974.2809981. URL: https://doi.org/10.14778/2809974.2809981.

[36] *Announcing the Firecracker Open Source Technology: Secure and Fast microVM for Serverless Computing*. https://aws.amazon.com/blogs/opensource/firecracker-open-source-secure-fast-microvm-serverless/.

[37] Sadjad Fouladi et al. “Encoding, Fast and Slow: Low-Latency Video Processing Using Thousands of Tiny Threads”. In: *14th USENIX Symposium on Networked Systems Design and Implementation (NSDI 17)*. Boston, MA: USENIX Association, 2017, pp. 363–376. ISBN: 978-1-931971-37-9. URL: https://www.usenix.org/conference/nsdi17/technical-sessions/presentation/fouladi.

[38] Sadjad Fouladi et al. “From Laptop to Lambda: Outsourcing Everyday Jobs to Thousands of Transient Functional Containers”. In: *2019 USENIX Annual Technical Conference (USENIX ATC 19)*. 2019, pp. 475–488.

[39] Armando Fox et al. "Above the clouds: A berkeley view of cloud computing". In: *Dept. Electrical Eng. and Comput. Sciences, University of California, Berkeley, Rep. UCB/EECS* 28.13 (2009), p. 2009.

[40] Tom Z. J. Fu et al. "DRS: Auto-Scaling for Real-Time Stream Analytics". In: *IEEE/ACM Trans. Netw.* 25.6 (Dec. 2017), pp. 3338–3352. ISSN: 1063-6692. DOI: 10.1109/TNET.2017.2741969. URL: https://doi.org/10.1109/TNET.2017.2741969.

[41] Yu Gan et al. "An Open-Source Benchmark Suite for Microservices and Their Hardware-Software Implications for Cloud & Edge Systems". In: *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*. ACM. 2019, pp. 3–18.

[42] Anshul Gandhi et al. "AutoScale: Dynamic, Robust Capacity Management for Multi-Tier Data Centers". In: *ACM Trans. Comput. Syst.* 30.4 (Nov. 2012), 14:1–14:26. ISSN: 0734-2071. DOI: 10.1145/2382553.2382556. URL: http://doi.acm.org/10.1145/2382553.2382556.

[43] Pin Gao et al. "Low latency RNN inference with cellular batching". In: *Proceedings of the Thirteenth EuroSys Conference*. 2018, pp. 1–15.

[44] Jonas Gehring et al. "Convolutional Sequence to Sequence Learning". In: *CoRR* abs/1705.03122 (2017). arXiv: 1705.03122. URL: http://arxiv.org/abs/1705.03122.

[45] Jon Gjengset et al. "Noria: dynamic, partially-stateful data-flow for high-performance web applications". In: *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18)*. Carlsbad, CA: USENIX Association, Oct. 2018, pp. 213–231. ISBN: 978-1-939133-08-3. URL: https://www.usenix.org/conference/osdi18/presentation/gjengset.

[46] Udit Gupta et al. "The architectural implications of Facebook's DNN-based personalized recommendation". In: *2020 IEEE International Symposium on High Performance Computer Architecture (HPCA)*. IEEE. 2020, pp. 488–501.

[47] *Open-sourcing gVisor, a sandboxed container runtime*. https://cloud.google.com/blog/products/gcp/open-sourcing-gvisor-a-sandboxed-container-runtime.

[48] Sangjin Han et al. "Network support for resource disaggregation in next-generation datacenters". In: *Proceedings of the Twelfth ACM Workshop on Hot Topics in Networks*. ACM. 2013, p. 10.

[49] Mazdak Hashemi. *The Infrastructure Behind Twitter: Scale*. https://blog.twitter.com/engineering/en_us/topics/infrastructure/2017/the-infrastructure-behind-twitter-scale.html.

[50] K. Hazelwood et al. "Applied Machine Learning at Facebook: A Datacenter Infrastructure Perspective". In: *2018 IEEE International Symposium on High Performance Computer Architecture (HPCA)*. Feb. 2018, pp. 620–629. DOI: 10.1109/HPCA.2018.00059.

[51] Kim Hazelwood et al. "Applied machine learning at facebook: A datacenter infrastructure perspective". In: *2018 IEEE International Symposium on High Performance Computer Architecture (HPCA)*. IEEE. 2018, pp. 620–629.

[52] Kaiming He et al. "Deep residual learning for image recognition". In: *Proceedings of the IEEE conference on computer vision and pattern recognition*. 2016, pp. 770–778.

[53] Yuxiong He et al. "Zeta: Scheduling interactive services with partial execution". In: *Proceedings of the Third ACM Symposium on Cloud Computing*. 2012, pp. 1–14.

[54] Pat Helland and David Campbell. "Building on Quicksand". In: *CoRR* abs/0909.1788 (2009).

[55] Joseph M. Hellerstein et al. "Serverless Computing: One Step Forward, Two Steps Back". In: *CIDR 2019, 9th Biennial Conference on Innovative Data Systems Research, Asilomar, CA, USA, January 13-16, 2019, Online Proceedings*. 2019. URL: http://cidrdb.org/cidr2019/papers/p119-hellerstein-cidr19.pdf.

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

[57] Scott Hendrickson et al. "Serverless Computation with OpenLambda". In: *8th USENIX Workshop on Hot Topics in Cloud Computing (HotCloud 16)*. Denver, CO: USENIX Association, 2016. URL: https://www.usenix.org/conference/hotcloud16/workshop-program/presentation/hendrickson.

[58] Scott Hendrickson et al. "Serverless computation with OpenLambda". In: *Elastic* 60 (2016), p. 80.

[59] Brandon Holt et al. "Claret: Using data types for highly concurrent distributed transactions". In: *Proceedings of the First Workshop on Principles and Practice of Consistency for Distributed Data*. ACM. 2015, p. 4.

[60] Andrew G. Howard et al. "MobileNets: Efficient Convolutional Neural Networks for Mobile Vision Applications". In: *CoRR* abs/1704.04861 (2017). arXiv: 1704.04861. URL: http://arxiv.org/abs/1704.04861.

[61] Michael Isard et al. "Dryad: Distributed Data-Parallel Programs from Sequential Building Blocks". In: *Proceedings of the 2nd ACM SIGOPS/EuroSys European Conference on Computer Systems 2007*. EuroSys '07. Lisbon, Portugal: Association for Computing Machinery, 2007, pp. 59–72. ISBN: 9781595936363. DOI: 10.1145/1272996.1273005. URL: https://doi.org/10.1145/1272996.1273005.

[62] Michael Isard et al. "Dryad: Distributed Data-parallel Programs from Sequential Building Blocks". In: *Proceedings of the 2Nd ACM SIGOPS/EuroSys European Conference on Computer Systems 2007*. EuroSys '07. Lisbon, Portugal: ACM, 2007, pp. 59–72. ISBN: 978-1-59593-636-3. DOI: 10.1145/1272996.1273005. URL: http://doi.acm.org/10.1145/1272996.1273005.

[63] Zhihao Jia et al. "TASO: Optimizing Deep Learning Computation with Automatic Generation of Graph Substitutions". In: *Proceedings of the 27th ACM Symposium on Operating Systems Principles*. SOSP '19. Huntsville, Ontario, Canada: Association for Computing Machinery, 2019, pp. 47–62. ISBN: 9781450368735. DOI: 10.1145/3341301.3359630. URL: https://doi.org/10.1145/3341301.3359630.

[64] Melvin Johnson et al. "Google's multilingual neural machine translation system: Enabling zero-shot translation". In: *Transactions of the Association for Computational Linguistics* 5 (2017), pp. 339–351.

[65] Eric Jonas et al. *Cloud Programming Simplified: A Berkeley View on Serverless Computing*. Tech. rep. UCB/EECS-2019-3. EECS Department, University of California, Berkeley, Feb. 2019. URL: http://www2.eecs.berkeley.edu/Pubs/TechRpts/2019/EECS-2019-3.html.

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

[67] Armand Joulin et al. "Bag of Tricks for Efficient Text Classification". In: *arXiv preprint arXiv:1607.01759* (2016).

[68] Armand Joulin et al. "FastText.zip: Compressing text classification models". In: *arXiv preprint arXiv:1612.03651* (2016).

[69] Norman P Jouppi et al. "In-datacenter performance analysis of a tensor processing unit". In: *Proceedings of the 44th Annual International Symposium on Computer Architecture*. 2017, pp. 1–12.

[70] Vasiliki Kalavri et al. "Three Steps is All You Need: Fast, Accurate, Automatic Scaling Decisions for Distributed Streaming Dataflows". In: *Proceedings of the 12th USENIX Conference on Operating Systems Design and Implementation*. OSDI'18. Carlsbad, CA, USA: USENIX Association, 2018, pp. 783–798. ISBN: 978-1-931971-47-8. URL: http://dl.acm.org/citation.cfm?id=3291168.3291226.

[71] Vasiliki Kalavri et al. "Three steps is all you need: fast, accurate, automatic scaling decisions for distributed streaming dataflows". In: *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18)*. Carlsbad, CA: USENIX Association, Oct. 2018, pp. 783–798. ISBN: 978-1-939133-08-3. URL: https://www.usenix.org/conference/osdi18/presentation/kalavri.

[72] David Kempe, Alin Dobra, and Johannes Gehrke. "Gossip-based computation of aggregate information". In: *44th Annual IEEE Symposium on Foundations of Computer Science, 2003. Proceedings.* IEEE. 2003, pp. 482–491.

[73] Ana Klimovic et al. "Pocket: Elastic ephemeral storage for serverless analytics". In: *13th {USENIX} Symposium on Operating Systems Design and Implementation ({OSDI} 18)*. 2018, pp. 427–444.

[74] Eddie Kohler et al. "The Click modular router". In: *ACM Transactions on Computer Systems (TOCS)* 18.3 (2000), pp. 263–297.

[75] Jack Kosaian, K. V. Rashmi, and Shivaram Venkataraman. "Parity Models: Erasure-Coded Resilience for Prediction Serving Systems". In: *Proceedings of the 27th ACM Symposium on Operating Systems Principles*. SOSP '19. Huntsville, Ontario, Canada: Association for Computing Machinery, 2019, pp. 30–46. ISBN: 9781450368735. DOI: 10.1145/3341301.3359654. URL: https://doi.org/10.1145/3341301.3359654.

[76] Jack Kosaian, KV Rashmi, and Shivaram Venkataraman. "Parity models: erasure-coded resilience for prediction serving systems". In: *Proceedings of the 27th ACM Symposium on Operating Systems Principles*. 2019, pp. 30–46.

[77] *Kubeless*. http://kubeless.io.

[78] *Kubernetes: Production-Grade Container Orchestration*. http://kubernetes.io.

[79] *Make a Lambda Function Idempotent*. https://aws.amazon.com/premiumsupport/knowledge-center/lambda-function-idempotent/.

[80] Leslie Lamport. "The part-time parliament". In: *ACM Transactions on Computer Systems (TOCS)* 16.2 (1998), pp. 133–169.

[81] Yunseong Lee et al. "{PRETZEL}: Opening the Black Box of Machine Learning Prediction Serving Systems". In: *13th {USENIX} Symposium on Operating Systems Design and Implementation ({OSDI} 18)*. 2018, pp. 611–626.

[82] Yunseong Lee et al. "PRETZEL: Opening the Black Box of Machine Learning Prediction Serving Systems". In: *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18)*. Carlsbad, CA: USENIX Association, 2018, pp. 611–626. ISBN: 978-1-931971-47-8. URL: https://www.usenix.org/conference/osdi18/presentation/lee.

[83] Wubin Li and Ali Kanso. "Comparing containers versus virtual machines for achieving high availability". In: *2015 IEEE International Conference on Cloud Engineering*. IEEE. 2015, pp. 353–358.

[84] B. Lohrmann, P. Janacik, and O. Kao. "Elastic Stream Processing with Latency Guarantees". In: *2015 IEEE 35th International Conference on Distributed Computing Systems*. June 2015, pp. 399–410. DOI: 10.1109/ICDCS.2015.48.

[85] Björn Lohrmann, Daniel Warneke, and Odej Kao. "Nephele Streaming: Stream Processing Under QoS Constraints At Scale". In: *Cluster Computing* 17 (Aug. 2013). DOI: 10.1007/s10586-013-0281-8.

[86] Boon Thau Loo et al. "Implementing Declarative Overlays". In: *Proceedings of the Twentieth ACM Symposium on Operating Systems Principles*. SOSP '05. Brighton, United Kingdom: Association for Computing Machinery, 2005, pp. 75–90. ISBN: 1595930795. DOI: 10.1145/1095810.1095818. URL: https://doi.org/10.1145/1095810.1095818.

[87] James Loope. *Managing infrastructure with puppet: configuration management at scale*. " O'Reilly Media, Inc.", 2011.

[88] Garrett McGrath and Paul R Brenner. "Serverless computing: Design, implementation, and performance". In: *2017 IEEE 37th International Conference on Distributed Computing Systems Workshops (ICDCSW)*. IEEE. 2017, pp. 405–410.

[89] Syed Akbar Mehdi et al. "I Can't Believe It's Not Causal! Scalable Causal Consistency with No Slowdown Cascades". In: *14th USENIX Symposium on Networked Systems Design and Implementation (NSDI 17)*. Boston, MA: USENIX Association, Mar. 2017, pp. 453–468. ISBN: 978-1-931971-37-9. URL: https://www.usenix.org/conference/nsdi17/technical-sessions/presentation/mehdi.

[90] Christopher Meiklejohn and Peter Van Roy. "Lasp: A language for distributed, coordination-free programming". In: *Proceedings of the 17th International Symposium on Principles and Practice of Declarative Programming*. 2015, pp. 184–195.

[91] Xiangrui Meng et al. "Mllib: Machine learning in apache spark". In: *The Journal of Machine Learning Research* 17.1 (2016), pp. 1235–1241.

[92] Matthew Milano et al. "A Tour of Gallifrey, a Language for Geodistributed Programming". In: *3rd Summit on Advances in Programming Languages (SNAPL 2019)*. Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik. 2019.

[93] Alan Mislove et al. "Measurement and Analysis of Online Social Networks". In: *Proceedings of the 7th ACM SIGCOMM Conference on Internet Measurement*. IMC '07. San Diego, California, USA: ACM, 2007, pp. 29–42. ISBN: 978-1-59593-908-1. DOI: 10.1145/1298306.1298311. URL: http://doi.acm.org/10.1145/1298306.1298311.

[94] Derek G. Murray et al. "Naiad: A Timely Dataflow System". In: *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*. SOSP '13. Farminton, Pennsylvania: Association for Computing Machinery, 2013, pp. 439–455. ISBN: 9781450323888. DOI: 10.1145/2517349.2522738. URL: https://doi.org/10.1145/2517349.2522738.

[95] Thomas Neumann, Tobias Mühlbauer, and Alfons Kemper. "Fast Serializable Multi-Version Concurrency Control for Main-Memory Database Systems". In: *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data*. SIGMOD '15. Melbourne, Victoria, Australia: ACM, 2015, pp. 677–689. ISBN: 978-1-4503-2758-9. DOI: 10.1145/2723372.2749436. URL: http://doi.acm.org/10.1145/2723372.2749436.

[96] Edward Oakes et al. "SOCK: Rapid Task Provisioning with Serverless-Optimized Containers". In: *2018 USENIX Annual Technical Conference (USENIX ATC 18)*. Boston, MA: USENIX Association, 2018, pp. 57–70. ISBN: 978-1-931971-44-7. URL: https://www.usenix.org/conference/atc18/presentation/oakes.

[97] *Home | OpenFaaS - Serverless Functions Made Simple*. https://www.openfaas.com.

[98] *Apache OpenWhisk is a serverless, open source cloud platform*. https://openwhisk.apache.org.

[99] Myle Ott et al. "fairseq: A Fast, Extensible Toolkit for Sequence Modeling". In: *CoRR* abs/1904.01038 (2019). arXiv: 1904.01038. URL: http://arxiv.org/abs/1904.01038.

[100] Adam Paszke et al. "Automatic differentiation in PyTorch". In: *NIPS-W*. 2017.

[101] Nathan Pemberton and Johann Schleier-Smith. "The Serverless Data Center: Hardware Disaggregation Meets Serverless Computing". In: *The First Workshop on Resource Disaggregation*. Vol. 4. 2019.

[102] Matthew Perron et al. "Starling: A Scalable Query Engine on Cloud Functions". In: *Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data*. 2020, pp. 131–141.

[103] Qifan Pu, Shivaram Venkataraman, and Ion Stoica. "Shuffling, fast and slow: Scalable analytics on serverless infrastructure". In: *16th {USENIX} Symposium on Networked Systems Design and Implementation ({NSDI} 19)*. 2019, pp. 193–206.

[104] Brian Randell. "System structure for software fault tolerance". In: *Ieee transactions on software engineering* (1975), pp. 220–232.

[105] Sylvia Ratnasamy et al. "A Scalable Content-addressable Network". In: *SIGCOMM Comput. Commun. Rev.* 31.4 (Aug. 2001), pp. 161–172. ISSN: 0146-4833. DOI: 10.1145/964723.383072. URL: http://doi.acm.org/10.1145/964723.383072.

[106] Joseph Redmon and Ali Farhadi. "Yolov3: An incremental improvement". In: *arXiv preprint arXiv:1804.02767* (2018).

[107] David Patrick Reed. "Naming and synchronization in a decentralized computer system." PhD thesis. Massachusetts Institute of Technology, 1978.

[108] *Tutorial: Design and implementation of a simple Twitter clone using PHP and the Redis key-value store | Redis*. https://redis.io/topics/twitter-clone.

[109] *pims/retwis-py: Retwis clone in Python*. https://github.com/pims/retwis-py.

[110] R Rodruigues, A Gupta, and B Liskov. "One-hop lookups for peer-to-peer overlays". In: *Proceedings of the 11th Workshop on Hot Topics in Operating Systems (HotOS'03)*. 2003.

[111] Antony Rowstron and Peter Druschel. "Pastry: Scalable, decentralized object location, and routing for large-scale peer-to-peer systems". In: *IFIP/ACM International Conference on Distributed Systems Platforms and Open Distributed Processing*. Springer. 2001, pp. 329–350.

[112] Olga Russakovsky et al. "Imagenet large scale visual recognition challenge". In: *International journal of computer vision* 115.3 (2015), pp. 211–252.

[113] *Nokia Bell Labs Project SAND*. https://sandserverless.org.

[114] Malte Schwarzkopf. *The Remarkable Utility of Dataflow Computing*. 2020. URL: https://www.sigops.org/2020/the-remarkable-utility-of-dataflow-computing/.

[115] Mohammad Shahrad et al. "Serverless in the Wild: Characterizing and Optimizing the Serverless Workload at a Large Cloud Provider". In: *arXiv preprint arXiv:2003.03423* (2020).

[116] Vaishaal Shankar et al. "numpywren: serverless linear algebra". In: *CoRR* abs/1810.09679 (2018). arXiv: 1810.09679. URL: http://arxiv.org/abs/1810.09679.

[117] Marc Shapiro et al. “Conflict-free replicated data types”. In: *Symposium on Self-Stabilizing Systems*. Springer. 2011, pp. 386–400.

[118] Arjun Singhvi et al. “Archipelago: A Scalable Low-Latency Serverless Platform”. In: *CoRR* abs/1911.09849 (2019). arXiv: 1911.09849. URL: http://arxiv.org/abs/1911.09849.

[119] Yair Sovran et al. “Transactional storage for geo-replicated systems”. In: *Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles*. ACM. 2011, pp. 385–400.

[120] Vikram Sreekanti et al. “A fault-tolerance shim for serverless computing”. In: *Proceedings of the Fifteenth European Conference on Computer Systems*. 2020, pp. 1–15.

[121] Vikram Sreekanti et al. “Cloudburst: Stateful Functions-as-a-Service”. In: *PVLDB* 13.11 (2020), pp. 2438–2452.

[122] Vikram Sreekanti et al. “Optimizing Prediction Serving on Low-Latency Serverless Dataflow”. In: *arXiv preprint arXiv:2007.05832* (2020).

[123] Maarten van Steep and Andrew S. Tanenbaum. *Distributed Systems*. 3.01. CreateSpace independen Publishing Platform, 2018.

[124] Ion Stoica et al. “Chord: A scalable peer-to-peer lookup service for internet applications”. In: *ACM SIGCOMM Computer Communication Review* 31.4 (2001), pp. 149–160.

[125] Michael Stonebraker. “The Design of the POSTGRES Storage System”. In: *Proceedings of the 13th International Conference on Very Large Data Bases*. VLDB ’87. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 1987, pp. 289–300. ISBN: 0-934613-46-X. URL: http://dl.acm.org/citation.cfm?id=645914.671639.

[126] Lalith Suresh et al. “C3: Cutting Tail Latency in Cloud Data Stores via Adaptive Replica Selection”. In: *12th USENIX Symposium on Networked Systems Design and Implementation (NSDI 15)*. Oakland, CA: USENIX Association, May 2015, pp. 513–527. ISBN: 978-1-931971-218. URL: https://www.usenix.org/conference/nsdi15/technical-sessions/presentation/suresh.

[127] Christian Szegedy et al. “Rethinking the Inception Architecture for Computer Vision”. In: *CoRR* abs/1512.00567 (2015). arXiv: 1512.00567. URL: http://arxiv.org/abs/1512.00567.

[128] Rebecca Taft et al. “E-store: Fine-grained Elastic Partitioning for Distributed Transaction Processing Systems”. In: *Proc. VLDB Endow.* 8.3 (Nov. 2014), pp. 245–256. ISSN: 2150-8097. DOI: 10.14778/2735508.2735514. URL: http://dx.doi.org/10.14778/2735508.2735514.

[129] Mischa Taylor and Seth Vargo. *Learning Chef: A Guide to Configuration Management and Automation*. " O’Reilly Media, Inc.", 2014.

[130] AzureML Team. "AzureML: Anatomy of a machine learning service". In: *Proceedings of The 2nd International Conference on Predictive APIs and Apps*. Ed. by Louis Dorard, Mark D. Reid, and Francisco J. Martin. Vol. 50. Proceedings of Machine Learning Research. Sydney, Australia: PMLR, Aug. 2016, pp. 1–13. URL: http://proceedings.mlr.press/v50/azureml15.html.

[131] Oliver Trachsel and Thomas R. Gross. "Variant-Based Competitive Parallel Execution of Sequential Programs". In: *Proceedings of the 7th ACM International Conference on Computing Frontiers*. CF '10. Bertinoro, Italy: Association for Computing Machinery, 2010, pp. 197–206. ISBN: 9781450300445. DOI: 10.1145/1787275.1787325. URL: https://doi.org/10.1145/1787275.1787325.

[132] Mark Twain. *Mark Twain's own autobiography: the chapters from the North American review*. Univ of Wisconsin Press, 2010.

[133] Erwin Van Eyk et al. "The SPEC cloud group's research vision on FaaS and serverless architectures". In: *Proceedings of the 2nd International Workshop on Serverless Computing*. ACM. 2017, pp. 1–4.

[134] Werner Vogels. "Eventually consistent". In: *Communications of the ACM* 52.1 (2009), pp. 40–44.

[135] Liang Wang et al. "Peeking behind the curtains of serverless platforms". In: *2018 {USENIX} Annual Technical Conference ({USENIX}{ATC} 18)*. 2018, pp. 133–146.

[136] Stephanie Wang et al. "Lineage stash: fault tolerance off the critical path". In: *Proceedings of the 27th ACM Symposium on Operating Systems Principles*. ACM. 2019, pp. 338–352.

[137] Matt Welsh, David Culler, and Eric Brewer. "SEDA: an architecture for well-conditioned, scalable internet services". In: *ACM SIGOPS Operating Systems Review* 35.5 (2001), pp. 230–243.

[138] Chenggang Wu, Vikram Sreekanti, and Joseph M Hellerstein. "Autoscaling tiered cloud storage in Anna". In: *Proceedings of the VLDB Endowment* 12.6 (2019), pp. 624–638.

[139] Chenggang Wu, Vikram Sreekanti, and Joseph M Hellerstein. "Transactional Causal Consistency for Serverless Computing". In: *Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data*. 2020, pp. 83–97.

[140] Chenggang Wu et al. "Anna: A kvs for any scale". In: *IEEE Transactions on Knowledge and Data Engineering* (2019).

[141] Yingjun Wu et al. "An Empirical Evaluation of In-memory Multi-version Concurrency Control". In: *Proc. VLDB Endow.* 10.7 (Mar. 2017), pp. 781–792. ISSN: 2150-8097. DOI: 10.14778/3067421.3067427. URL: https://doi.org/10.14778/3067421.3067427.

[142] Zhe Wu, Curtis Yu, and Harsha V. Madhyastha. "CosTLO: Cost-Effective Redundancy for Lower Latency Variance on Cloud Storage Services". In: *12th USENIX Symposium on Networked Systems Design and Implementation (NSDI 15)*. Oakland, CA: USENIX Association, May 2015, pp. 543–557. ISBN: 978-1-931971-218. URL: https://www.usenix.org/conference/nsdi15/technical-sessions/presentation/wu.

[143] Xinan Yan et al. "Carousel: low-latency transaction processing for globally-distributed data". In: *Proceedings of the 2018 International Conference on Management of Data*. ACM. 2018, pp. 231–243.

[144] Matei Zaharia et al. "Apache Spark: A Unified Engine for Big Data Processing". In: *Commun. ACM* 59.11 (Oct. 2016), pp. 56–65. ISSN: 0001-0782. DOI: 10.1145/2934664. URL: https://doi.org/10.1145/2934664.

[145] Matei Zaharia et al. "Resilient distributed datasets: A fault-tolerant abstraction for in-memory cluster computing". In: *Proceedings of the 9th USENIX conference on Networked Systems Design and Implementation*. USENIX Association. 2012, pp. 2–2.

[146] Haoyu Zhang et al. "Live Video Analytics at Scale with Approximation and Delay-Tolerance". In: *14th USENIX Symposium on Networked Systems Design and Implementation (NSDI 17)*. Boston, MA: USENIX Association, Mar. 2017, pp. 377–392. ISBN: 978-1-931971-37-9. URL: https://www.usenix.org/conference/nsdi17/technical-sessions/presentation/zhang.

[147] Irene Zhang et al. "Diamond: Automating data management and storage for wide-area, reactive applications". In: *12th {USENIX} Symposium on Operating Systems Design and Implementation ({OSDI} 16)*. 2016, pp. 723–738.

[148] Tian Zhang et al. "Narrowing the Gap Between Serverless and its State with Storage Functions". In: *Proceedings of the ACM Symposium on Cloud Computing*. 2019, pp. 1–12.

[149] Guorui Zhou et al. "Deep interest network for click-through rate prediction". In: *Proceedings of the 24th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining*. 2018, pp. 1059–1068.