Shattering the Ephemeral Storage Cost Barrier for Data-Intensive Serverless Workflows

Shyam Jesalpura*
University of Edinburgh
UK

Dmitrii Ustiugov*†
NTU Singapore
Singapore

Michal Baczun
Bloomberg
UK

Bora A. Malper
Stripe
Ireland

Rustem Feyzkhanov
Instrumental
USA

Edouard Bugnion
EPFL
Swizerland

Marios Kogias
Imperial College London
UK

Boris Grot
University of Edinburgh
UK

Abstract

Serverless computing enables developers to deploy applications as workflows of functions that invoke one another, with cloud providers handling autoscaling and routing. However, serverless platforms lack efficient mechanisms for cross-function data transfers, which hinders the performance of data-intensive applications. Current solutions rely on intermediary services like AWS S3 or ElastiCache(EC), leading to significant cost inefficiencies—storage costs can account for 24-99% of the total execution bill.

Zipline addresses this challenge with a fast, API-compatible data communication method enabling direct function-to-function transfers. Zipline buffers data in the sender function's memory and transmits only the references to the dynamically selected receiver, which pulls the data directly from the sender's memory. While eliminating the need for intermediary services, it also integrates seamlessly with existing autoscaling infrastructure, preserving function invocation semantics while significantly reducing costs and latency. In a vHive/Knative prototype on AWS EC2, Zipline achieves 2-5× lower costs & 1.3-3.4× faster execution times compared to S3. Against EC, Zipline cuts costs by 17-772× while improving performance by 2-5%. Zipline demonstrates a cost-effective and high-performance solution for data-intensive serverless applications.

CCS Concepts

• **Computer systems organization → Cloud computing**.

ACM Reference Format:

Shyam Jesalpura, Dmitrii Ustiugov, Michal Baczun, Bora A. Malper, Rustem Feyzkhanov, Edouard Bugnion, Marios Kogias, and Boris Grot. 2025. Shattering the Ephemeral Storage Cost Barrier for Data-Intensive Serverless Workflows. In *The 3rd Workshop on SErverless Systems, Applications and MEthodologies (SESAME' 25), March 30-April 3, 2025, Rotterdam, Netherlands.* ACM, New York, NY, USA, 9 pages. https://doi.org/10.1145/3721465.3721866

*Both authors contributed equally to this research.

†Corresponding author: dmitrii.ustiugov@ntu.edu.sg

SESAME' 25, Rotterdam, Netherlands

ACM ISBN 979-8-4007-1557-0/2025/03
https://doi.org/10.1145/3721465.3721866

1 Introduction

Serverless functions are stateless and ephemeral, requiring inter-function communication to pass intermediate state. Typically, a producer function invokes consumer functions, passing data inputs. However, the consumer instances are dynamically assigned by the cloud provider's load balancer and autoscaler, making direct communication challenging. Data transfers can be substantial, often tens of MBs, as seen in applications like video analytics [24, 25, 51, 52], data analytics [46, 49, 50], and ML [29].

The common programming model for data communication is object-centric, using an intermediate external service with a `put()/get()` interface. This service can be a storage service (e.g., AWS S3, Google Cloud Storage) or an in-memory cache (e.g., AWS ElastiCache). The producer stores the data, invokes the consumer, which then retrieves the data from storage. This indirection via *storage services* introduces latency and additional costs.

Researchers have proposed solutions to improve serverless communication efficiency. Some seek to improve the performance of storage-based transfers using tiered storage, such as combining an in-memory cache layer (e.g., EC) with a cold storage layer (e.g., S3) [42, 45, 51, 56]. While tiered storage can improve performance over a single storage layer (or cost over a single in-memory cache layer), the disadvantages of through-storage indirection remain.

We find that serverless architectures using through-storage transfers, such as AWS S3 or multi-tier services, incur prohibitively high costs for storing transmitted data. Even with perfect garbage collection, intermediate bookkeeping costs dominate execution costs for data-intensive applications. For instance, in a MapReduce shuffle phase, data transmission via S3 and EC can account for 70% to over 99% of total processing costs.

By studying the production traces from Azure Functions [51], we make the following key observation: 75% of data objects transmitted across functions must be buffered for only 30 seconds or less. In contrast, a function instance's lifetime (e.g., the minimum keep-alive period of an idle instance before serverless infrastructure tears it down) spans to many minutes [9, 53]. Hence, the function instances' lifetime significantly exceeds the transmitted data's lifetime.

We exploit the disparity between the data and instances' lifetimes and introduce Zipline[1] (ZL), a serverless communication substrate

[1]The Zipline code is available at https://github.com/ease-lab/zipline.

that allows direct communication between two function instances in a manner that is flexible and compatible with the autoscaling infrastructure used by cloud providers. ZL preserves the existing API and invocation semantics of serverless functions while avoiding the need for intermediate storage for arbitrarily-sized data transfers. At the heart of ZL is an explicit separation of the control plane used for function invocation, which is tightly integrated with the autoscaling infrastructure, from the data transfer itself. In simplest terms, with ZL, the producer function buffers the data that needs to be transferred in its memory and sends a reference to the data inlined with the invocation to the consumer function. The consumer then directly *pulls* the data from the producer's memory. More concretely, ZL defines a short-lived namespace of objects with the same lifetime as the function instance. Subsequent function instances can access this namespace through references that do not expose the underlying infrastructure to the user code.

ZL naturally supports a variety of inter-function communication patterns, including producer-consumer, scatter (map), gather (reduce), and broadcast. Compared to through-storage transfers, ZL avoids high-latency data copies to and from a storage layer and the associated monetary cost of storage usage. Critically, ZL is fully compatible with the autoscaling infrastructure and requires minimal modifications at the endpoints of the existing control plane.

We prototype ZL in Knative [3], by extending its queue-proxy components with ZL support. We evaluate our proposal by deploying a ZL-enabled vHive cluster in AWS EC2. Using real-world applications, we show that ZL delivers 2-5× lower cost *and* superior performance versus transfers via S3 storage (i.e., cheapest among existing solutions) and EC in-memory cache (i.e., fastest among prior works) for all the above communication patterns in serverless.

The main contributions of our work are as follows:

- We demonstrate that through-storage communication incurs high costs, making up 24-99% of total expenses for data-intensive serverless applications.
- We observe that function instances live significantly longer than the data they transmit, suggesting the use of instance memory for buffering the transmissions.
- We introduce ZL, which separates control and data paths, allowing direct data transfer from producer to consumer memory, supporting various communication patterns and compatible with autoscaling.
- We show ZL's efficiency and cost-effectiveness, outperforming S3 by 1.3-3.4× with 2-5× cost savings, and surpassing EC by 2-5% while reducing costs by 17-772×.

2 Background and Motivation

Below we describe the modern serverless cloud architectures and programming models for data-intensive applications and evaluate the associated performance and cost overheads.

2.1 Serverless Computing and Autoscaling

The serverless paradigm divides responsibilities between the programmer and the infrastructure. The programming model uses *functions* as the core abstraction and *function instances* as units of scaling. Developers can deploy applications without managing system configuration or cloud resources, as the serverless infrastructure automatically adjusts the number of function instances based on traffic.

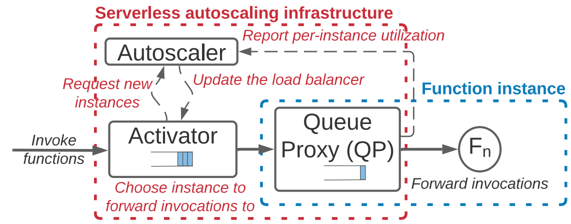


Figure 1: Operation of serverless autoscaling infrastructure.

We describe the serverless autoscaling infrastructure (Fig. 1) using the Knative [3] terminology. The autoscaling infrastructure aims to achieve two objectives: (1) respond to load changes by spawning new function instances when the load increases and shutting down idle instances when the load drops, and (2) minimize queuing latency by balancing the load across active instances.

Instance scaling and load-balancing decisions rely on utilization metrics from active function instances, gathered by the *queue-proxy* component, which forwards incoming requests and reports metrics to the *autoscaler*. The autoscaler monitors the load and implements the scaling policy. To balance the load, serverless clouds use a *load balancer* (referred to as the *activator* in Knative) to steer requests to instances. If no active instances are available or all are busy, the activator requests new instances from the autoscaler, which then spawns new instances while the activator buffers the requests. Once the instances are up, the activator directs the requests to them.

Together, the queue proxy, autoscaler, and load balancer enable serverless function autoscaling, ensuring scalability for developers and resource efficiency for cloud providers.

2.2 Data-Intensive Applications in Serverless

Data-intensive applications are prevalent in today's serverless clouds [24, 25, 29, 46, 49–52]. These applications require rapid state communication between processing stages (i.e., functions) in a workflow. Typically, serverless functions handle single stages (e.g., map and reduce) with instances managing individual state pieces [24, 31, 32]. This allows developers to leverage available compute resources without managing autoscaling or resource allocation.

The challenge lies in enabling fast state communication (referred to as *objects*) across functions while maintaining serverless benefits like elasticity and cost-efficiency. Direct communication via traditional POSIX APIs (e.g., sockets) could offer high performance but would require custom autoscaling and data-partitioning solutions, negating serverless advantages. Instead, existing data-intensive serverless applications use storage services (e.g., cloud storage or in-memory cache) for object transfers via `put()`/`get()` APIs [23, 51].[2] We term these methods as *through-storage* transfers.

[2] Small data objects can be passed inline, such as AWS Lambda's support for objects smaller than 256KB and 6MB for asynchronous and synchronous invocations, respectively. However, through-storage transfers are more common in practice.

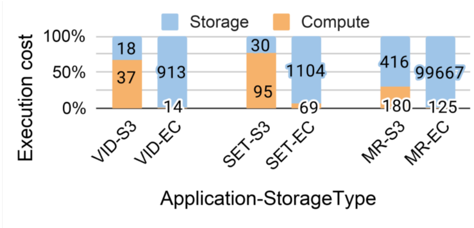


Figure 2: The cost breakdown for real-world data-intensive multi-function applications (§5.0.5), namely Video Analytics (VID), Stacking Ensemble Training (SET), and Map-Reduce (MR), when performing data transfers through AWS S3 and ElastiCache (EC). The numbers show the cost values in (in $USD \times 10^{-6}$) for compute and storage expenses.

2.3 Through-Storage Transfers and Their Cost

Through-storage communication in serverless architectures incurs performance and financial overhead. The financial cost includes charges for each `Get()/Put()` operation and the storage lease cost, proportional to the duration and size of data stored remotely.

Prior research has explored cost-performance trade-offs in storage solutions for data-intensive serverless applications. These include using conventional storage for cost efficiency, in-memory cache for high performance, or multi-tier systems combining both. For example, Locus [50] utilizes AWS ElastiCache (EC) for shuffling and S3 for cold storage. Solutions like Pocket [32] and SONIC [42] adopt control-plane mechanisms to dynamically multiplex storage tiers based on application needs. Other systems, such as Faa$T [51], Cloudburst [56], and OFC [45], employ key-value stores for distributed caching. These approaches introduce direct storage costs (operations and leases) and indirect computational costs due to storage latency impacting function execution time.

Even with the cheapest storage solutions, data-intensive serverless applications often face disproportionate storage costs. Using AWS pricing models, we estimate the cost of buffering data in S3 and EC under conservative assumptions: immediate deallocation after retrieval and no overprovisioning. However, practical constraints—like S3's minimum expiration time of one day and EC's 1 GB metering minimum—result in higher real-world costs. Fig. 2 illustrates that storage accounts for 24–70% and 94–99% of overall costs when using S3 and EC, respectively.

These findings demonstrate that through-storage architectures are economically impractical for data-intensive applications. While multi-tier systems narrow the performance gap between in-memory and cloud storage, the latter remains the cost-optimal but slowest tier, contributing significantly to both cost and tail latency, as highlighted in prior studies [58].

3 Zipline Communication

3.1 Design Insights

We exploit three insights enabling a serverless communication model, which, in the common case, obviates the need for through-storage transfers.

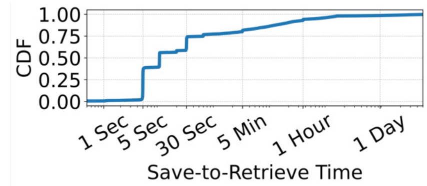


Figure 3: CDFs of the time duration between saving a data object in storage and its last retrieval, based on Azure Blob Traces [51]. Note the logarithmic scale on the horizontal axis.

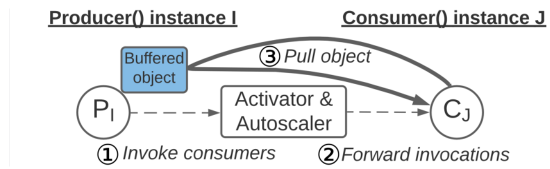


Figure 4: Zipline architecture overview.

API Call	Description
`rsp := invoke(URL, obj)`	Invoke a function
`ref := put(obj, N)`	Buffer an object locally
`obj := get(ref)`	Fetch a remote object

Table 1: Zipline API description.

Our first insight is to separate control (function invocation) and data (transfer) paths without impacting the autoscaling infrastructure. The challenge is doing so without resorting to a storage service, which is what curent through-storage transfers rely on. We address this challenge with the help of the second insight.

The second insight is that the data transferred between instances are ephemeral, with lifetimes on the order of a few seconds. Using the Azure Blob Traces [51], we analyze the time between an object produced by one function and its *last* retrieval by another function of the same application. Fig. 3 shows that 75% of the data objects transferred across functions are consumed within 30 seconds. Hence, the data lifetime is much shorter than the keep-alive period of serverless functions (which is typically in the order of minutes to maximize the likelihood of a warm invocation [9, 53]).

Based on the above, we draw one final insight: instead of using a storage service to communicate data across function instances, a producer instance can simply buffer the data in its own memory and have the consumer instance pull from it. We note that most language runtimes require buffering the transmitted object in a memory buffer before calling the `Put()` API of the storage service, so the system only needs to provide a way to pass a pointer to that buffer to the target consumer instance. This insight forms the foundation for Zipline, presented next.

3.2 Design Overview

We introduce Zipline (ZL), a serverless-native data communication fabric that meets all five serverless communication requirements: high performance, compliance with existing serverless function invocation semantics, compatibility with autoscaling, and standard data-transfer APIs.

Following the insights from Sec. 3.1, ZL separates function invocation into control and data planes. The control plane, unchanged, matches the existing serverless architecture (Fig. 1), allowing autoscaling to balance loads by directing invocations to the least-loaded instances. It carries only control messages. The data plane handles object transfers. In essence, a producer function instance in ZL buffers data in its memory and sends a reference to the consumer function(s). The consumer(s) then pull the data directly from the producer's memory. This replaces push-based data transfers with a pull-based approach after the control plane has made its decisions.

Fig. 4 illustrates ZL operation. Consider two serverless functions, a producer and a consumer, each with multiple instances. The producer invokes the consumer function, passing a data object as an argument. Unlike existing systems, in ZL, consumer function invocations go to the activator separately from their corresponding objects (1), which remain buffered at the source. The activator, after consulting the autoscaler, selects a consumer instance and forwards the invocation (2). Upon receiving the invocation, the consumer instance pulls the object from the producer instance (3) using the reference in the invocation message.

3.2.1 Zipline Programming Model. The ZL programming model features a minimalist yet expressive API (Table 1) that supports essential communication patterns: invoking a function, scattering and broadcasting objects to multiple consumers, and gathering outputs from several functions. The API is compatible with production cloud APIs like AWS Lambda and S3's Boto3 [15].

ZL supports both blocking and non-blocking interfaces. The `invoke()` call invokes a function by its `URL`, passing a binary data object `obj` by value, with the object buffered at the producer side until the consumer instance pulls it. For non-blocking transfers, ZL uses `put()` and `get()` calls, similar to a key-value store interface. The producer can finish the invocation before the consumer retrieves the object.

ZL introduces references as first-class primitives to decouple function invocation and data transfer. When `put()` is called, the runtime returns a reference to the object, which the consumer can retrieve using `get()`. Each reference is associated with a specified number of retrievals `N` and includes a ZL ID to uniquely identify objects within the same workflow.

This model allows seamless porting of serverless applications, such as those for AWS Lambda or Knative, with corresponding wrapper functions. We implemented ZL SDKs for Python and Golang and deployed them in a Knative cluster to demonstrate the API's portability.

3.2.2 Zipline Semantics. ZL complies with *at-most-once* semantics and can be extended to at-least-once semantics using existing serverless infrastructure with minimal modifications. Modern serverless platforms like AWS Lambda and Azure Functions provide at-most-once invocation semantics [26, 34, 36], ensuring an invocation executes no more than once, even in case of failures.[3] Providers

[3]Users can achieve at-least-once semantics by combining at-most-once primitives with retry logic. Prior work also demonstrates constructing exactly-once semantics [36].

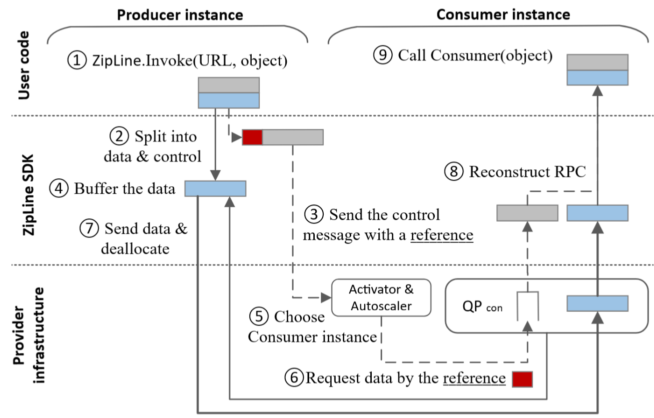


Figure 5: Zipline operation in a one producer one consumer scenario (only the request path is shown). Dashed arrows show the control plane, solid lines show the data plane, and the thick solid lines show data streaming in the data plane.

expose runtime errors to user logic for handling [16, 17, 19, 44]. Error handling varies based on function composition, either as direct chains or asynchronous workflows managed by orchestrators like AWS Step Functions [14] or Azure Durable Functions [43]. Failures may require re-executing several functions, necessitating the user to pass context throughout the workflow.

3.2.3 Zipline Error Handling . In a two-function workflow, the lifetime of a ZL object is tied to the producer instance. If the producer shuts down, all its objects are de-allocated immediately. For blocking invocations (`invoke()`), the producer waits for the consumer's response and may re-invoke if an error occurs. For non-blocking invocations, a ZL transfer may fail if the producer instance is terminated before the consumer retrieves the object. The consumer receives an error on `get()` and must re-invoke the workflow from the producer. The consumer should forward this error to the orchestrator or driver function to re-invoke the producer with the original arguments. For instance, AWS Step Functions allows defining custom fallback functions for error handling [17]. Providers could enhance ZL error handling by backing up non-retrieved objects to cloud storage before instance shutdown, converting ZL references to storage service keys. The consumer would first attempt a regular ZL retrieval, followed by a storage service retrieval if needed. If ZL errors persist, the infrastructure can disable ZL for those functions.

4 Implementation

We prototype Zipline (ZL) in vHive [59] using the Knative model [3].

4.1 Zipline Prototype in vHive/Knative

4.1.1 ZL Software Development Kit (SDK). ZL uses an SDK to bridge user logic with provider components for data transfer. At the producer side, the SDK splits the invocation request into a control message and an object (the data). It creates a ZL reference, an encrypted string containing the pod's IP address and a unique object key, and adds it to the gRPC request header. This encryption ensures IP addresses remain hidden from user code.

At the consumer side, the SDK reconstructs the original request by combining the control message and the retrieved object, then invokes the consumer function as with the standard serverless API.

4.1.2 Control and Data Planes. ZL uses gRPC [2] for the control plane and for the data plane, we choose the high-performance Cap'n Proto [1] RPC fabric. This fabric runs directly on top of TCP, delivering higher performance when compared to gRPC, whose performance is limited by HTTP compatibility.

4.1.3 Provider Components Extension. We extend the Knative queue proxy (QP) for object buffering (§2.1). QP is an auxiliary per-function provider container deployed in the same pod as the function server. The added logic increases the QP memory footprint by 2MB.

The ZL architecture is general and can be easily ported to popular serverless platforms like AWS Lambda and Google Cloud Run. Google Cloud Run is based on Knative and shall support ZL with minimal changes [7]. AWS offers Lambda Extensions that allow tools to execute custom logic before and after function invocations, enabling use cases like diagnostics, instrumentation, and configuration [6]. In the future, AWS could potentially release extensions that provide functionality similar to Zipline.

4.2 Zipline Operation

4.2.1 ZL `invoke()` Operation. Fig. 5 illustrates the ZL request path during an `invoke()` call: (1) The caller function invokes the SDK. (2) The SDK splits the request into a ZL object and a control plane message containing the object reference. (3) The SDK sends the control message to the activator and (4) stores the object in a buffer for the consumer's QP (QP_{con}). (5) The activator selects the consumer instance and forwards the control message to the consumer's QP (QP_{con}). (6) QP_{con} decrypts the reference, extracts the IP address and object key, and requests the data from the producer's SDK via Cap'n Proto RPC. (7) The producer's SDK sends the data to QP_{con} and deallocates the object. (8) QP_{con} forwards the object to the SDK, which reconstructs the original request, and (9) invokes the function handler. If the response is small, it follows the reverse control plane path through the QPs and the activator.

4.2.2 ZL `get()` / `put()` Operation. While `invoke()` is synchronous, ZL's `put()` and `get()` are asynchronous. The key difference is that `put()` returns a ZL reference for the object, which the producer can pass to any function within the same user domain. The consumer retrieves the object by calling `get()` with the reference, prompting the SDK to fetch the object via a Cap'n Proto RPC request directly from the producer instance. The same technique is used for large responses, where the consumer sends a reference and the producer fetches the object.

4.2.3 ZL Flow Control. Cap'n Proto RPC, built on TCP, inherently supports flow control, requiring no changes to ZL logic. If transmitted objects exceed available buffers, transfers pause, causing the user code to block in the ZL API call.

5 Methodology

5.0.1 Evaluation Platform. We evaluate Zipline (ZL) on a cluster of AWS EC2 `m5.16xlarge` instances in 'us-west-1', ensuring low access time to AWS S3 similar to [32, 42, 63]. Each instance features a 64 core Intel Xeon Platinum 8000 series 3 processor, 256GB RAM, and a 20Gb/s NIC. Pods are scheduled to ensure all data transfers occur over the network, with each function placed on a separate EC2 node. All experiments emulate a stable serverless workflow with no cold starts.

5.0.2 Measurement. Unless specified otherwise, we report average end-to-end latency based on 10 measurements. For microbenchmarks, which do not have any computational overheads except network processing, we calculate *effective bandwidth* by dividing the transferred object size by the measured end-to-end latency.

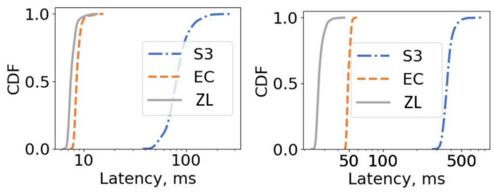
5.0.3 Baseline and ZL Configurations. Our baseline employs two storage options for through-storage communication, representing the performance and cost extremes of multi-tier ephemeral storage [32, 42, 50, 51, 56]. The first is AWS S3, the most economical but slowest option. The second is ElastiCache (EC), a high-performance in-memory store, priced over 100× more than S3. We use EC in on-demand mode, which is four times cheaper than its serverless mode [11]. Prior studies [31, 32] highlight EC's superior performance for inter-function communication at a premium cost. For EC, we configure a single-node Redis cache of type `cache.m6g.16xlarge` with 64 vCPUs and a 25 Gb/s NIC, costing $4.7 per hour.

5.0.4 Microbenchmarks. We implement microbenchmarks in Golang to evaluate common serverless data transfer patterns (§3.2.1): producer-consumer (1-1), scatter, gather, and broadcast. Each pattern involves varying numbers of producer and consumer function instances, transferring one or more objects between them. Hereafter, a producer (consumer) refers to a producer (consumer) function instance.

5.0.5 Real-World Workloads. We use three data-intensive applications from the vSwarm suite [5]. Each workload consists of multiple functions deployed with Knative Serving [4], using a blocking interface for inter-function communication. We modify the workloads to support ZL alongside S3 and ElastiCache (EC) baselines using the same communication API: `invoke()`, `get()`, and `put()` (§3.2.1).

The workloads demonstrate various communication patterns. *Video Analytics (VID)* involves 1-1 and scatter patterns with functions for video streaming, frame decoding, and object recognition. The frame decoder invokes the object recognition function for several frames in a scatter pattern. *Stacking Ensemble Training (SET)* is a distributed ML training application utilizing broadcast and gather patterns. The initial function broadcasts the training dataset to parallel training tasks, and the final function gathers and reconciles the trained models. *MapReduce (MR)* implements the Aggregation Query from the AMPLab Big Data Benchmark [48], with the gather pattern being crucial during the data-intensive shuffling phase between mapper and reducer functions.

5.0.6 Cost Model. We estimate the cost of executing the studied applications from the developer's perspective using AWS pricing models [11–13]. The cost of a function invocation includes a fixed



(a) Latency CDFs for 10KB obj.(b) Latency CDFs for 10MB obj.

Figure 6: Transfer latency CDFs for S3, ElastiCache (EC) and ZL in the *1-1* workflow. Note the log scale on the latency axis.

fee, a fee based on the `processing time` and `maximum memory footprint` (assumed to be 512MB), and storage costs for data transfer. Storage costs are calculated on a `GB/month` basis for AWS S3 [12] and `GB/hour` for on-demand AWS EC [11]. We assume ephemeral storage de-allocates data immediately after the last retrieval, although in practice, services like AWS S3 do not support expiration times below one day as of 2024.

6 Evaluation

We compare Zipline (ZL) to through-storage transfers based on AWS S3 and ElastiCache (EC). We first study the performance of the communication mechanisms on microbenchmarks featuring 1-1, gather, scatter, and broadcast patterns. We then assess the performance and cost of real-world serverless applications in cloud.

6.1 Microbenchmarks

6.1.1 Producer-Consumer Communication. We focus on the 1-1 (producer-consumer) pattern to study the latency characteristics of the communication methods.

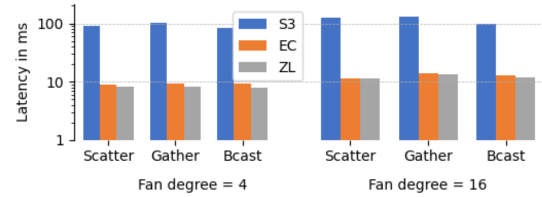
Fig. 6 shows the median and 99th percentile latencies for S3, EC, and ZL transfers for 10KB and 10MB objects. For 10KB objects, EC offers significantly lower latency than S3, with median and tail latencies reduced by 89% and 92%, respectively. ZL further improves on EC, reducing median and tail latencies by 12% and 10%. For 10MB objects, EC reduces median and tail latencies by 87% and 90% compared to S3, while ZL achieves 45% and 34% lower latencies than EC. ZL's advantage comes from avoiding intermediate writes and reads, which are more pronounced with larger objects.

6.1.2 Collective Communication. We evaluate the latency and effective bandwidth[4] of collective communication patterns (gather, scatter, broadcast) for fan-in and fan-out degrees of 4 and 16, using 10KB and 10MB transfer sizes.

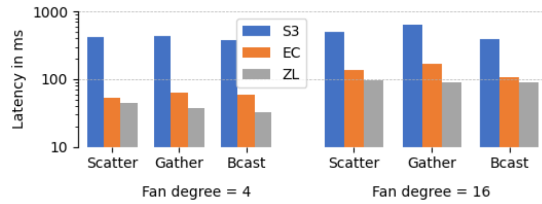
Fig. 7a shows results for 10KB transfers. EC outperforms S3, with 9.2-11.0× lower latency at a fan degree of 4 and 7.8-10.8× lower at a fan degree of 16. ZL matches or exceeds EC, achieving up to 1.16× lower latency.

For 10MB transfers (Fig. 7b), EC maintains its advantage over S3 with up to 7.7× lower latency. ZL further improves on EC, delivering 1.2-1.9× lower latency. ZL also achieves higher effective bandwidth. For 10MB transfers with a fan degree of 32, ZL reaches 16.4Gb/s (82% of NIC peak bandwidth), compared to EC's 14.0Gb/s (70%) and S3's 5.5Gb/s (28%).

[4]The latency and bandwidth metrics are calculated as the size of the transferred objects divided by the end- to-end transfer time.



(a) 10KB object transfers



(b) 10MB object transfers

Figure 7: Transfer latency of the scatter, gather, and broadcast patterns with the fan degrees of 4 and 16. Note that both subfigures use a log scale for latency, but the scales differ.

	S3			ElastiCache(EC)			**ZL**
App	Comp.	Stor.	**Total**	Comp.	Stor.	**Total**	**Total (comp.)**
VID	37	18	**55**	14	913	**928**	**17**
SET	95	30	**125**	69	1104	**1172**	**70**
MR	180	416	**595**	125	99667	**99792**	**129**

Table 2: Cost estimation (in $USD \times 10^{-6}$) for compute and storage spending when executing a single invocation for S3, EC, and ZL based configurations based on AWS Lambda [13], AWS S3 [12], and AWS EC [11] prices as of 1/1/2023.

6.2 Real-World Workloads

Next, we study three data-intensive applications (§5.0.5), presenting their end-to-end latency along with a detailed breakdown (Fig. 8) and estimating the associated cost (Table 2) of executing a request.

6.2.1 Performance Analysis.

Video Analytics (VID). The workload spends 39% and 5% of its execution time transferring the video fragment and the frames in the S3-based and EC-based configurations, respectively. With ZL, this fraction decreases to 4%, reducing the overall processing time by 36% and 2% vs. the S3 and EC baselines, respectively. This speedup comes from 9.5× and 1.2× faster transmission of video and frames, respectively.

Stacking Ensemble Training (SET). SET spends 76% and 14% of execution time in communication in the S3-based and EC-based configuration, respectively. The largest fraction of data communication is the *gather trained models* latency component, accounting for

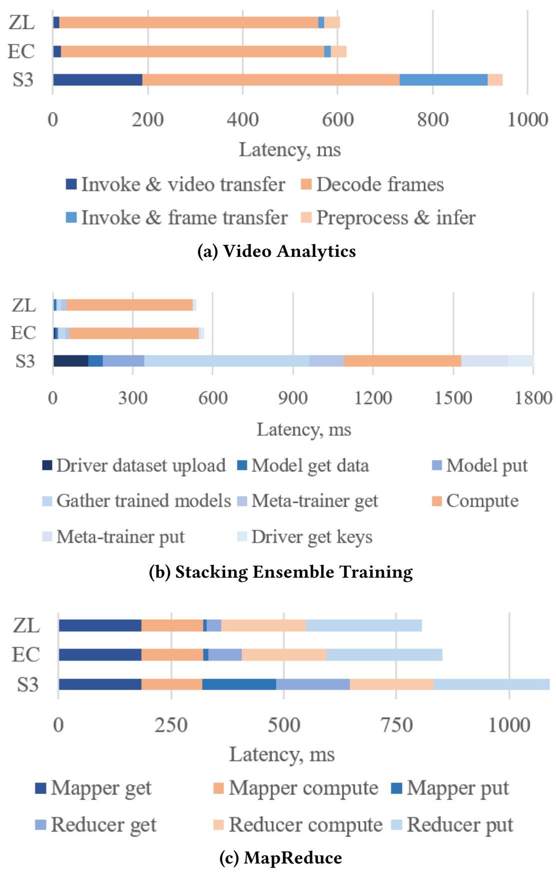


Figure 8: Latency breakdown of real-world workloads, deployed in Zipline, ElastiCache (EC) and S3 based systems.

34% and 4% of the overall execution time in the S3-based and EC-based configurations, respectively. Using ZL decreases the gather fraction to 3% of the end-to-end latency, driving the communication fraction down to 12%. Thus, ZL delivers a 3.4× speedup over the S3 baseline and 1.05× vs. EC.

MapReduce (MR). The workload shows 70% and 62% of execution time spent in communication for the S3 and EC configurations respectively. Moreover, 40% of the overall time in S3 baseline is spent retrieving the original input from S3 and writing back the results to S3, which we do not optimize with ZL. The rest, i.e., 30% of time, are subject to ZL optimization. ZL delivers 1.26× overall speedup over the S3 baseline and 1.05× over EC. ZL's speedup is due to a significant decrease in data shuffling, namely mapper-put and the reducer-get phases, which are reduced by 23.4× and 4.8×, respectively, compared to the S3 baseline, and by 30% and 55%, respectively, compared to EC.

6.2.2 Cost Analysis.

A single invocation processed in a ZL-enabled system lowers the cost by 2-5× and 56×, compared to S3 based configurations. With EC, the cost reduction is 56× in the case of VID, 17× for SET, and 772× for MR. This large cost reduction associated with MR is due to the large amount of ephemeral data transferred during the shuffle phase, making through-storage transfers particularly expensive.

7 Related Work

Prior works [32, 42, 50, 51, 56] explore ephemeral storage services for high-performance transfers at reasonable costs. However, as shown in §2.3, even the cheapest tier (e.g., AWS S3) can significantly impact the overall cost of data-intensive serverless applications. Other studies [22, 30, 45, 51, 54, 55, 61] propose extending serverless with a distributed shared memory tier to pass references instead of data objects. Unlike these, Zipline transmits immutable data objects, simplifying data consistency. YuanRong [21], a Huawei system, also passes data by references but lacks implementation details.

Other works [47, 57, 62, 63] explore connection-based communication for serverless applications, aiding in porting microservices and monoliths but difficult to apopt in serverless-native applications [23, 51] as these optimizations conflict with the core serverless principle of transparent autoscaling by the cloud provider, as they require reconfiguring workflow topology when scaling instances. In contrast, Zipline uses an object-centric `get()/put()` API, ensuring compatibility with existing cloud autoscaling infrastructure.

Similar to Zipline, prior works propose separating control and data planes to avoid bottlenecks and enhance performance. Crab [33] and Prism [27] reduce load on L4 and L7 load balancers, respectively. Dataflower [38] and FUYAO [39] use asynchronous transfers to decouple data from the control plane. FractOS [60] uses RPCs to orchestrate distributed OS-level operations across disaggregated datacenter hardware, associating references (capabilities) with each call. RMMAP [40] utilizes RDMA to expose remote memory to the receiving function to support serialization-free data transfers.

Zipline achieves high-performance transfers without relying on function instance co-location or data locality, distinguishing it from prior works. SAND [10] uses a hierarchical messaging bus for co-located function communication. FaaSFlow [37], Sledge [41], and Wukong [20] leverage locality for faster serverless multi-function execution. Faastlane [35] adopts a similar approach to execute functions in the same container as lightweight threads, isolating them using Intel MPK. [8] Nightcore [28] exchanges messages via OS pipes for co-located functions. However, commercial systems like AWS Lambda avoid function co-location to prevent hotspots [9, 18], favoring statistical multiplexing across a large server fleet.

8 Conclusion

The cost and performance of data-intensive serverless applications depend on efficient inter-function data transfers. Current methods fall short of these demands. We introduce Zipline (ZL), a high-speed, API-preserving direct function-to-function communication method that integrates with existing autoscaling infrastructure. ZL uses control/data separation and references to provide low latency and high bandwidth. A prototype ZL reduces the cost of end-to-end applications by 2-5× and accelerates real-world serverless applications by 1.3-3.4×. Compared to through-cache transfers, ZL cuts costs by 17-772× while achieving speedups of 2-5%.

9 Acknowledgements

The authors thank the anonymous reviewers and the members of the EASE lab at the University of Edinburgh and the HyScale lab at NTU Singapore for their valuable feedback and thoughtful discussions about this work. The work was partially supported by the NTU start-up grant, University of Edinburgh, Huawei and by the EASE Lab's industry partners and sponsors including ARM, Cisco, and Intel.

References

[1] Cap'n Proto. Available at https://capnproto.org.
[2] gRPC: A High-Performance, Open Source Universal RPC Framework. Available at https://grpc.io.
[3] Knative. Available at https://knative.dev.
[4] Knative Serving. Available at https://knative.dev/docs/serving.
[5] vSwarm - Serverless Benchmarking Suite. Available at https://github.com/vhive-serverless/vSwarm/tree/main/benchmarks.
[6] Augment Lambda functions using Lambda extensions - AWS Lambda, March 2025. [Online; accessed 3. Mar. 2025].
[7] Cloud Run sidecars enable advanced multi-container patterns | Google Cloud Blog, March 2025. [Online; accessed 3. Mar. 2025].
[8] Memory Protection Keys — The Linux Kernel documentation, March 2025. [Online; accessed 7. Mar. 2025].
[9] Alexandru Agache, Marc Brooker, Alexandra Iordache, Anthony Liguori, Rolf Neugebauer, Phil Piwonka, and Diana-Maria Popa. Firecracker: Lightweight Virtualization for Serverless Applications. In *Proceedings of the 17th Symposium on Networked Systems Design and Implementation (NSDI)*, pages 419–434, 2020.
[10] Istemi Ekin Akkus, Ruichuan Chen, Ivica Rimac, Manuel Stein, Klaus Satzke, Andre Beck, Paarijaat Aditya, and Volker Hilt. SAND: Towards High-Performance Serverless Computing. In *Proceedings of the 2018 USENIX Annual Technical Conference (ATC)*, pages 923–935, 2018.
[11] Amazon. Amazon ElastiCache Pricing. Available at https://aws.amazon.com/elasticache/pricing.
[12] Amazon. Amazon S3 Pricing. Available at https://aws.amazon.com/s3/pricing.
[13] Amazon. AWS Lambda Pricing. Available at https://aws.amazon.com/lambda/pricing.
[14] Amazon. AWS Step Functions. Available at https://aws.amazon.com/step-functions.
[15] Amazon. Boto3 documentation. Available at https://boto3.amazonaws.com/v1/documentation/api/latest/index.html.
[16] Amazon. Error Handling and Automatic Retries in AWS Lambda. Available at https://docs.aws.amazon.com/lambda/latest/dg/invocation-retries.html.
[17] Amazon. Error Handling in Step Functions. Available at https://docs.aws.amazon.com/step-functions/latest/dg/concepts-error-handling.html.
[18] Bharathan Balaji, Christopher Kakovitch, and Balakrishnan Narayanaswamy. FirePlace: Placing Firecraker Virtual Machines with Hindsight Imitation. *Proceedings of the 34th Workshop on Machine Learning for Systems at NeurIPS 2020*, 3, 2021.
[19] Sebastian Burckhardt, Chris Gillum, David Justo, Konstantinos Kallas, Connor McMahon, Christopher S. Meiklejohn, and Xiangfeng Zhu. Netherite: Efficient Execution of Serverless Workflows. *Proc. VLDB Endow.*, 15(8):1591–1604, 2022.
[20] Benjamin Carver, Jingyuan Zhang, Ao Wang, Ali Anwar, Panruo Wu, and Yue Cheng. Wukong: A Scalable and Locality-enhanced Framework for Serverless Parallel Computing. In *Proceedings of the 2020 ACM Symposium on Cloud Computing (SOCC)*, pages 1–15, 2020.
[21] Qiong Chen, Jianmin Qian, Yulin Che, Ziqi Lin, Jianfeng Wang, Jie Zhou, Licheng Song, Yi Liang, Jie Wu, Wei Zheng, Wei Liu, Linfeng Li, Fangming Liu, and Kun Tan. Yuanrong: A production general-purpose serverless system for distributed applications in the cloud. In *SIGCOMM Conference*, pages 843–859, 2024.
[22] Dong Du, Qingyuan Liu, Xueqiang Jiang, Yubin Xia, Binyu Zang, and Haibo Chen. Serverless Computing on Heterogeneous Computers. In *Proceedings of the 27th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS-XXVII)*, pages 797–813, 2022.
[23] Simon Eismann, Joel Scheuner, Erwin van Eyk, Maximilian Schwinger, Johannes Grohmann, Nikolas Herbst, Cristina L. Abad, and Alexandru Iosup. The state of serverless applications: Collection, characterization, and community consensus. *IEEE Transactions on Software Engineering*, 48(10), 2022.
[24] Sadjad Fouladi, Francisco Romero, Dan Iter, Qian Li, Shuvo Chatterjee, Christos Kozyrakis, Matei Zaharia, and Keith Winstein. From Laptop to Lambda: Outsourcing Everyday Jobs to Thousands of Transient Functional Containers. In *Proceedings of the 2019 USENIX Annual Technical Conference (ATC)*, pages 475–488, 2019.
[25] Sadjad Fouladi, Riad S. Wahby, Brennan Shacklett, Karthikeyan Balasubramaniam, William Zeng, Rahul Bhalerao, Anirudh Sivaraman, George Porter, and Keith Winstein. Encoding, Fast and Slow: Low-Latency Video Processing Using Thousands of Tiny Threads. In *Proceedings of the 14th Symposium on Networked Systems Design and Implementation (NSDI)*, pages 363–376, 2017.
[26] Armando Fox and Eric A. Brewer. Harvest, Yield and Scalable Tolerant Systems. In *Proceedings of The 7th Workshop on Hot Topics in Operating Systems (HotOS-VII)*, pages 174–178, 1999.
[27] Yutaro Hayakawa, Michio Honda, Douglas Santry, and Lars Eggert. Prism: Proxies without the Pain. In *Proceedings of the 18th Symposium on Networked Systems Design and Implementation (NSDI)*, pages 535–549, 2021.
[28] Zhipeng Jia and Emmett Witchel. Nightcore: Efficient and Scalable Serverless Computing for Latency-Sensitive, Interactive Microservices. In *Proceedings of the 26th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS-XXVI)*, pages 152–166, 2021.
[29] Jiawei Jiang, Shaoduo Gan, Yue Liu, Fanlin Wang, Gustavo Alonso, Ana Klimovic, Ankit Singla, Wentao Wu, and Ce Zhang. Towards Demystifying Serverless Machine Learning Training. In *SIGMOD Conference*, pages 857–871, 2021.
[30] Anurag Khandelwal, Yupeng Tang, Rachit Agarwal, Aditya Akella, and Ion Stoica. Jiffy: Elastic Far-Memory for Stateful Serverless Analytics. In *Proceedings of the 2022 EuroSys Conference*, pages 697–713, 2022.
[31] Ana Klimovic, Yawen Wang, Christos Kozyrakis, Patrick Stuedi, Jonas Pfefferle, and Animesh Trivedi. Understanding Ephemeral Storage for Serverless Analytics. In *Proceedings of the 2018 USENIX Annual Technical Conference (ATC)*, pages 789–794, 2018.
[32] Ana Klimovic, Yawen Wang, Patrick Stuedi, Animesh Trivedi, Jonas Pfefferle, and Christos Kozyrakis. Pocket: Elastic Ephemeral Storage for Serverless Analytics. In *Proceedings of the 13th Symposium on Operating System Design and Implementation (OSDI)*, pages 427–444, 2018.
[33] Marios Kogias, Rishabh Iyer, and Edouard Bugnion. Bypassing the Load Balancer without Regrets. In *Proceedings of the 2020 ACM Symposium on Cloud Computing (SOCC)*, pages 193–207, 2020.
[34] Marios Kogias, George Prekas, Adrien Ghosn, Jonas Fietz, and Edouard Bugnion. R2P2: Making RPCs First-Class Datacenter Citizens. In *Proceedings of the 2019 USENIX Annual Technical Conference (ATC)*, pages 863–880, 2019.
[35] Swaroop Kotni, Ajay Nayak, Vinod Ganapathy, and Arkaprava Basu. Faastlane: Accelerating Function-as-a-Service workflows. In *2021 USENIX Annual Technical Conference (USENIX ATC 21)*, pages 805–820. USENIX Association, July 2021.
[36] Collin Lee, Seo Jin Park, Ankita Kejriwal, Satoshi Matsushita, and John K. Ousterhout. Implementing Linearizability at Large Scale and Low Latency. In *Proceedings of the 25th ACM Symposium on Operating Systems Principles (SOSP)*, pages 71–86, 2015.
[37] Zijun Li, Yushi Liu, Linsong Guo, Quan Chen, Jiagan Cheng, Wenli Zheng, and Minyi Guo. FaaSFlow: Enable Efficient Workflow Execution for Function-as-a-Service. In *Proceedings of the 27th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS-XXVII)*, pages 782–796, 2022.
[38] Zijun Li, Chuhao Xu, Quan Chen, Jieru Zhao, Chen Chen, and Minyi Guo. Dataflower: Exploiting the data-flow paradigm for serverless workflow orchestration. In *Proceedings of the 29th ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS-XXIX)*, 2024.
[39] Guowei Liu, Laiping Zhao, Yiming Li, Zhaolin Duan, Sheng Chen, Yitao Hu, Zhiyuan Su, and Wenyu Qu. Fuyao: Dpu-enabled direct data transfer for serverless computing. In *Proceedings of the 29th ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS-XXIX)*, 2024.
[40] Fangming Lu, Xingda Wei, Zhuobin Huang, Rong Chen, Minyu Wu, and Haibo Chen. Serialization/deserialization-free state transfer in serverless workflows. In *Proceedings of the Nineteenth European Conference on Computer Systems*, EuroSys '24, page 132–147, New York, NY, USA, 2024. Association for Computing Machinery.
[41] Xiaosu Lyu, Ludmila Cherkasova, Robert C. Aitken, Gabriel Parmer, and Timothy Wood. Towards Efficient Processing of Latency-Sensitive Serverless DAGs at the Edge. In *Proceedings of the 5th International Workshop on Edge Systems, Analytics and Networking (EdgeSys)*, pages 49–54, 2022.
[42] Ashraf Mahgoub, Karthick Shankar, Subrata Mitra, Ana Klimovic, Somali Chaterji, and Saurabh Bagchi. SONIC: Application-aware Data Passing for Chained Serverless Applications. In *Proceedings of the 2021 USENIX Annual Technical Conference (ATC)*, pages 285–301, 2021.
[43] Microsoft. What are Durable Functions? Available at https://aws.amazon.com/step-functions.
[44] Mikhail Shilkov. Making Sense of Azure Durable Functions. Available at https://mikhail.io/2018/12/making-sense-of-azure-durable-functions.
[45] Djob Mvondo, Mathieu Bacou, Kevin Nguetchouang, Lucien Ngale, Stéphane Pouget, Josiane Kouam, Renaud Lachaize, Jinho Hwang, Tim Wood, Daniel Hagimont, Noël De Palma, Bernabé Batchakui, and Alain Tchana. OFC: An Opportunistic Caching System for FaaS Platforms. In *Proceedings of the 2021 EuroSys Conference*, pages 228–244, 2021.
[46] Ingo Müller, Renato Marroquin, and Gustavo Alonso. Lambada: Interactive Data Analytics on Cold Data Using Serverless Cloud Infrastructure. In *SIGMOD*

Conference, pages 115–130, 2020.

[47] Anna Maria Nestorov, Josep Lluís Berral, Claudia Misale, Chen Wang, David Carrera, and Alaa Youssef. Floki: A Proactive Data Forwarding System for Direct Inter-Function Communication for Serverless Workflows. In *Proceedings of the 8th International Workshop on Container Technologies and Container Clouds (WoC)*, pages 13–18, 2022.

[48] Andrew Pavlo, Erik Paulson, Alexander Rasin, Daniel J. Abadi, David J. DeWitt, Samuel Madden, and Michael Stonebraker. A Comparison of Approaches to Large-Scale Data Analysis. In *SIGMOD Conference*, pages 165–178, 2009.

[49] Matthew Perron, Raul Castro Fernandez, David J. DeWitt, and Samuel Madden. Starling: A Scalable Query Engine on Cloud Functions. In *SIGMOD Conference*, pages 131–141, 2020.

[50] Qifan Pu, Shivaram Venkataraman, and Ion Stoica. Shuffling, Fast and Slow: Scalable Analytics on Serverless Infrastructure. In *Proceedings of the 16th Symposium on Networked Systems Design and Implementation (NSDI)*, pages 193–206, 2019.

[51] Francisco Romero, Gohar Irfan Chaudhry, Iñigo Goiri, Pragna Gopa, Paul Batum, Neeraja J. Yadwadkar, Rodrigo Fonseca, Christos Kozyrakis, and Ricardo Bianchini. Faa$T: A Transparent Auto-Scaling Cache for Serverless Applications. In *Proceedings of the 2021 ACM Symposium on Cloud Computing (SOCC)*, pages 122–137, 2021.

[52] Francisco Romero, Mark Zhao, Neeraja J. Yadwadkar, and Christos Kozyrakis. Llama: A Heterogeneous & Serverless Framework for Auto-Tuning Video Analytics Pipelines. In Carlo Curino, Georgia Koutrika, and Ravi Netravali, editors, *Proceedings of the 2021 ACM Symposium on Cloud Computing (SOCC)*, pages 1–17, 2021.

[53] Mohammad Shahrad, Rodrigo Fonseca, Iñigo Goiri, Gohar Chaudhry, Paul Batum, Jason Cooke, Eduardo Laureano, Colby Tresness, Mark Russinovich, and Ricardo Bianchini. Serverless in the Wild: Characterizing and Optimizing the Serverless Workload at a Large Cloud Provider. In *Proceedings of the 2020 USENIX Annual Technical Conference (ATC)*, pages 205–218, 2020.

[54] Simon Shillaker and Peter R. Pietzuch. Faasm: Lightweight Isolation for Efficient Stateful Serverless Computing. In *Proceedings of the 2020 USENIX Annual Technical Conference (ATC)*, pages 419–433, 2020.

[55] Qi Shixiong, Monis Leslie, Zeng Ziteng, Wang Ian-chin, and Ramakrishnan K. K. Spright: Extracting the server from serverless computing! high-performance ebpf-based event-driven, shared-memory processing. In *SIGCOMM Conference*, pages 780–794, 2022.

[56] Vikram Sreekanti, Chenggang Wu, Xiayue Charles Lin, Johann Schleier-Smith, Joseph Gonzalez, Joseph M. Hellerstein, and Alexey Tumanov. Cloudburst: Stateful Functions-as-a-Service. *Proc. VLDB Endow.*, 13(11):2438–2452, 2020.

[57] Shelby Thomas, Lixiang Ao, Geoffrey M. Voelker, and George Porter. Particle: Ephemeral Endpoints for Serverless Networking. In *Proceedings of the 2020 ACM Symposium on Cloud Computing (SOCC)*, pages 16–29, 2020.

[58] Dmitrii Ustiugov, Theodor Amariucai, and Boris Grot. Analyzing tail latency in serverless clouds with stellar. In *Proceedings of the 2021 IEEE International Symposium on Workload Characterization (IISWC)*, 2021.

[59] Dmitrii Ustiugov, Plamen Petrov, Marios Kogias, Edouard Bugnion, and Boris Grot. Benchmarking, analysis, and optimization of serverless function snapshots. In *Proceedings of the 26th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS-XXVI)*, pages 559–572, 2021.

[60] Lluís Vilanova, Lina Maudlej, Shai Bergman, Till Miemietz, Matthias Hille, Nils Asmussen, Michael Roitzsch, Hermann Härtig, and Mark Silberstein. Slashing the disaggregation tax in heterogeneous data centers with fractos. In *Proceedings of the Seventeenth European Conference on Computer Systems*, EuroSys '22, page 352–367, New York, NY, USA, 2022. Association for Computing Machinery.

[61] Stephanie Wang, Benjamin Hindman, and Ion Stoica. In Reference to RPC: It's Time to Add Distributed Memory. In *Proceedings of The 17th Workshop on Hot Topics in Operating Systems (HotOS-XVIII)*, pages 191–198, 2021.

[62] Michael Wawrzoniak, Gianluca Moro, Rodrigo Bruno, Ana Klimovic, and Gustavo Alonso. Off-the-shelf Data Analytics on Serverless. In *Proceedings of the Conference on Innovative Data Systems Research (CIDR)*, 2024.

[63] Michal Wawrzoniak, Ingo Müller, Gustavo Alonso, and Rodrigo Bruno. Boxer: Data Analytics on Network-enabled Serverless Platforms. In *Proceedings of the 11th Biennial Conference on Innovative Data Systems Research (CIDR)*, 2021.