

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

Dmitrii Ustiugov*
NTU Singapore

Shyam Jesalpura
University of Edinburgh

Mert Bora Alper*
Stripe

Michal Baczun*
Bloomberg

Rustem Feyzkhanov
Instrumental

Edouard Bugnion
EPFL

Marios Kogias
Imperial College London &
Microsoft Research

Boris Grot
University of Edinburgh

Abstract—Serverless computing has emerged as a popular cloud deployment paradigm. In serverless, the developers implement their application as a set of functions forming a workflow in which functions invoke each other. The cloud providers are responsible for automatically scaling the number of instances for each function on demand and forwarding the requests in a workflow to the appropriate function instance. Problematically, today’s serverless clouds lack efficient support for cross-function data transfers in a workflow, preventing the efficient execution of data-intensive serverless applications. Instead, functions transmit intermediate, i.e., ephemeral, data to other functions through third-party services, such as AWS S3 storage, AWS ElastiCache in-memory cache, or multi-tier solutions proposed by prior works.

We show that data-intensive application deployments in serverless platforms that rely on such *through-storage* data transfers are economically impractical, with the storage costs accounting for >24-99% of the total serverless execution bill. We introduce Zipline, an API-preserving fast data communication method for serverless that enables direct function-to-function transfers. With Zipline, the sender function’s runtime buffers the payload in its memory and sends a reference to the receiver, which the load balancer and autoscaler pick based on the current load. Using the reference, the receiver instance pulls the transmitted data directly from the sender’s memory. Zipline is natively compatible with existing autoscaling infrastructure, preserves function invocation semantics, and avoids the cost and performance overheads of using an intermediate service for data transfers. We prototype our system in vHive/Knative deployed on a cluster of AWS EC2 nodes, showing that Zipline reduces the overall cost and improves latency and bandwidth over AWS S3 (lowest cost state-of-the-art) and ElastiCache (highest performance state-of-the-art). On real-world applications, Zipline reduces the overall cost of 2-5× and application execution time by 1.3-3.4× compared to S3. Compared to ElastiCache, Zipline reduces the total cost by 17-772× while improving performance by 2-5%.

I. INTRODUCTION

Serverless computing has emerged as a pervasive cloud technology due to its scalability, resource- and cost-efficiency – factors that benefit both cloud providers and their customers. All major cloud providers have serverless offerings, including AWS Lambda [9], Azure Functions [49], and Google Cloud Functions [31] and Google Cloud Run [66]. In serverless, the application logic is organised as a set of *stateless functions* that

communicate with each other and with cloud storage services hosting the application state. Serverless computing is expressive enough to support many applications, e.g., video encoding [29], [61], compilation [28], [36] and machine learning [34].

The stateless and ephemeral nature of function instances mandates that functions communicate any intermediate and ephemeral state across the functions comprising the application logic. Inter-function communication generally happens when one function, the *producer*, invokes one or more *consumer* functions in the workflow and passes inputs to them. Crucially, the instances of the consumer functions are not known by the producer at invocation time because they are picked by the cloud provider’s load balancer and autoscaler components on demand. Also, for many applications, the amount of data transmitted across function instances can be large, measuring 10s of MBs or more; examples include video analytics [28], [29], [60], [61], data analytics [52], [56], [57], and ML [34].

Programming model for data communication is *object-centric*, with the functions passing data via an intermediate external service, typically using a `put()`/`get()` interface. This intermediate service can be a storage service (e.g., AWS S3 or Google Cloud Storage) or an in-memory cache service (e.g., AWS ElastiCache), which requires the producer function to first store the data, then invoke the consumer, and subsequently have the consumer retrieve the data from storage. The indirection via an intermediate service introduces large latency overheads and adds to the cost of the intermediate service. We further refer to such services as *storage services*.

Researchers have identified the problem of efficient serverless communication and have proposed solutions. Some seek to improve the performance of storage-based transfers using tiered storage, such as combining an in-memory cache layer (e.g., ElastiCache) with a cold storage layer (e.g., S3) [47], [51], [60], [65]. 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 observe that serverless architectures that use through-storage transfers, whether via AWS S3 or previously proposed intelligent multi-tier services, are economically unattractive due to the prohibitively high costs of storing transmitted

* The work was done when the author was at the University of Edinburgh

data in storage services. We find that even if these services implemented perfect garbage collection, i.e., deallocating transmitted objects immediately after the last retrieval, the costs of intermediate bookkeeping would still dominate the overall execution cost for data-intensive applications. For example, we show that transmitting data via S3 and ElastiCache in a MapReduce application’s shuffle phase can account for 70% to over 99% of the total processing cost.

By studying the production traces from Azure Functions [60], 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], [62]. Hence, the function instances’ lifetime significantly exceeds the transmitted data’s lifetime.

We exploit the disparity between the data and instances’ lifetimes and introduce Zipline¹, a serverless communication substrate that allows direct communication between two function instances in a manner that is flexible and compatible with the autoscaling infrastructure used by cloud providers. Zipline 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 Zipline 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 Zipline, 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, Zipline 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.

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

We prototype Zipline in Knative [4], which is used in many commercial clouds [38], by extending its queue-proxy components with Zipline support. We evaluate our proposal by deploying a Zipline-enabled vHive cluster in AWS EC2. Using real-world applications, we show that Zipline delivers 2-5 \times lower cost *and* superior performance versus transfers via S3 storage (i.e., cheapest among existing solutions) and ElastiCache in-memory cache (i.e., fastest among prior works) for all the above communication patterns in serverless computing.

The main contributions of our work are as follows:

- We show that through-storage inter-function communication imposes prohibitively high storage-related costs, accounting for 24-99% of the total expenses, dominating the overall serverless execution bill for data-intensive applications.
- We find that the lifetime of serverless function instances significantly exceeds the lifetime of the data objects transmitted across functions, indicating the opportunity for using the instances’ memory for buffering the transmissions.
- We introduce Zipline, which uses control/data path separation to pass an object reference to a consumer function instance as part of an invocation request, and delegates to the consumer, pulling the data from the producer’s memory. Zipline supports various inter-function communication patterns and is fully compatible with serverless autoscaling infrastructure.
- We demonstrate that Zipline is flexible and fast. On real-world applications, Zipline outperforms S3 by 1.3-3.4 \times with cost savings of 2-5 \times . Furthermore, Zipline also consistently outperforms ElastiCache (the highest-performance data transfer option available today) by 2-5% while slashing the overall application execution cost by 17-77 \times .

II. 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.

A. Serverless Computing and Autoscaling

The division of labor between the programmer and the underlying infrastructure is central to the serverless paradigm, defining its programming model and infrastructure management philosophy. The programming model defines *a function* as the key abstraction for programming and *a function instance* as a unit of placement and scaling from the infrastructure perspective. Using these abstractions, developers can write and deploy arbitrary applications without considering system configuration and cloud resource management. Instead, serverless infrastructure transparently manages cloud resources, continuously adjusting the number of function instances based on the current function invocation traffic.

We describe the operation of serverless autoscaling infrastructure (Figure 1) using the Knative [4] terminology since it is used widely in production [38] and representative of the leading clouds [45]. The autoscaling infrastructure of serverless aims to achieve two objectives. The first objective is responding to load changes by spawning new function instances when the load increases and shutting down idle instances once the load drops. The second objective is minimizing queuing latency by balancing the load across the active instances.

Instance scaling and load-balancing decisions inherently rely on utilization metrics from the active function instances, gathered and stored with the help of the following two components. Each function invocation traverses a provider-managed *queue-proxy* component, which is in charge of forwarding incoming requests to the function instance with which the queue proxy is co-located. Queue proxy also collects and reports utilization metrics of that instance to the *autoscaler*

¹We plan to release the Zipline’s source code by the time of publication.

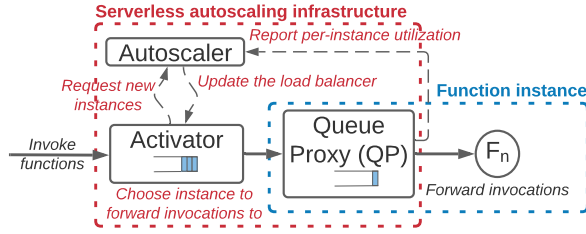


Figure 1: Operation of serverless autoscaling infrastructure.

control plane component. The *autoscaler* monitors the load in front of active instances and implements the scaling policy.

To balance the load among all active instances of a function, serverless clouds employ a *load balancer* whose job is to steer each request to one of the instances. Every request must traverse the load balancer, referred to as the *activator* in Knative. The autoscaler periodically updates the activator about active instances and their load. If there is an incoming request for a function and no active instances are available, or all of them are busy, the activator needs to request new instances of that function from the autoscaler. The autoscaler makes a placement decision and spawns a new instance while the activator buffers the pending request. Once the instance is up, the activator steers the invocation to the instance via its corresponding queue proxy.

Together, queue proxy, autoscaler, and load balancer drive serverless functions autoscaling. The rest of the system is designed around this triplet to deliver scalability to application developers and resource efficiency to cloud providers.

B. Programming Data-Intensive Applications in Serverless

Data-intensive applications are ubiquitous in today’s serverless clouds [28], [29], [34], [52], [56], [57], [60], [61]. However, these applications are challenging to support with contemporary serverless infrastructure because they need to rapidly communicate state from one processing stage (i.e., function) to another in an application workflow. Typically, serverless functions represent a single stage (e.g., map and reduce functions) in such an application’s data-processing workflow while each function’s instances can be in charge of a single piece of state [28], [36], [37]. This way the developer is completely free of any autoscaling or resource management decision since the application can leverage any available compute resource offered to it depending on the combination of pieces of state and workflow stages.

The challenge is to devise a solution for fast communication of these pieces of state, further referred to as *objects*, across different functions (i.e., their instances) while keeping the benefits of serverless computing, i.e. elasticity, and without substantially increasing cost. Establishing direct communication between the instances using traditional POSIX APIs, e.g., via sockets, could enable cheap and high-performance communication. Still, it would require the developer to devise custom autoscaling and data-partitioning solutions in the application code. Thus, such solutions forego serverless computing’s main advantage, which is its obviating the need for infrastructure management by application developers. Instead, existing data-intensive applications on serverless pass objects across functions via

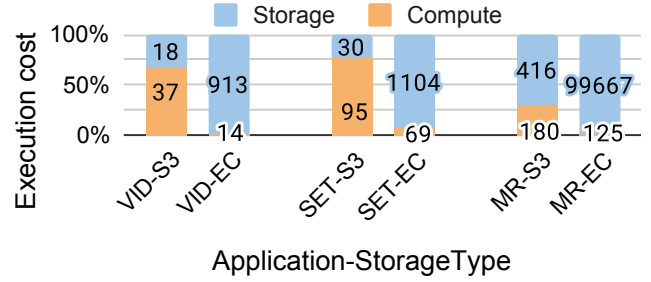


Figure 2: The cost breakdown for real-world data-intensive multi-function applications (§V-5), namely Video Analytics (VID), Stacking Ensemble Training (SET), and Map-Reduce (MR), when performing data transfers through AWS S3 and ElastiCache (EC). The numbers show the cost values in (in $USD \times 10^{-6}$) for compute and storage expenses.

storage services, such as cloud storage or in-memory cache, typically using a `put()`/`get()` API [27], [60].² We refer to these communication methods as *through-storage* transfers.

C. Through-Storage Transfers and Their Cost

Through-storage communication imposes a performance overhead as well as a financial cost. The financial aspect includes the cost for each `Get()` or `Put()` operation plus the cost of storing data objects in remote storage in GB-hour. We refer to the latter as the *storage lease* cost, proportional to the time and space the objects take up in the remote storage.

Prior works have rigorously explored the cost-performance trade-offs in storage architectures for data-intensive serverless applications using a conventional storage service as the cheapest option, an in-memory cache service as the highest-performance alternative, or a multi-tier combination of the two. For example, Locus [57] uses different storage tiers for specific purposes, namely AWS ElastiCache for shuffling and S3 for cold storage. Pocket [37] and SONIC [47] employ a similar idea and develop a control-plane solution to multiplex different storage services based on inferred application needs. FaaS\$T [60], Cloudburst [65], and OFC [51] propose using key-value stores to cache objects in the main memory or disk distributed across the serverless cluster deployment. Since all these solutions require additional bookkeeping, their usage adds to the bill of serverless application execution in two ways. The first cost type is associated directly with the storage usage, namely the cost of operations and storage lease. The second cost type is indirect, as the latency of saving and retrieving objects from storage is a part of the billed serverless function execution time, i.e., the computational cost. Hence, transferring data through slow storage would result in higher computational costs.

Surprisingly, even using the cheapest storage technologies may dominate the overall execution cost for data-intensive

²In serverless, it is possible to pass small data objects inline in simple workflows such as a pipeline. For example, AWS Lambda supports inline transfers for objects smaller than 256KB and 6MB for asynchronous and synchronous function invocations, respectively. Thus, through-storage transfers are dominant in practice.

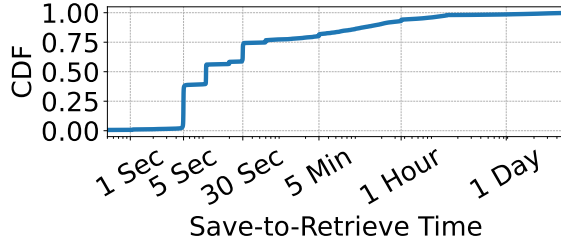


Figure 3: Cumulative Distribution Functions (CDFs) of the time duration between saving a data object in storage and its last retrieval, based on Azure Blob Traces [60]. Note the logarithmic scale on the horizontal axis.

serverless applications. We estimate the costs from an application developer’s perspective based on the applications’ execution time measured in a real-world deployment and using the AWS pricing model to determine the billed execution time and storage capacity [11]–[13] (details are in §V). In AWS Lambda, compute time is billed proportionally to execution time and maximum function memory usage. We estimate the lower-bound costs for buffering transferred data objects in AWS S3 and ElastiCache, making two conservative assumptions: (1) data objects are de-allocated immediately after the last retrieval and (2) storage space is never overprovisioned.³ Figure 2 shows the cost breakdown for three widely-used applications, accounting for 24-70% and 94-99% of the overall cost when using S3 and ElastiCache for transfers, respectively.

Given the above results, it is clear that serverless architectures that rely on through-storage transfers are economically infeasible for data-intensive applications. Although multi-tier proposals [37], [47], [51], [60] have narrowed the performance gap between in-memory caches and cloud storage, the latter still underpins their most cost-optimal tier, hence such systems would impose much storage costs in between our estimates for S3 and ElastiCache based transfers. Finally, the slowest tier in multi-tier systems inevitably becomes the main source of the tail latency increase for data-intensive applications, as discovered by prior work [68].

III. ZIPLINE COMMUNICATION

A. Design Insights

We exploit three insights that enable a serverless communication model, which, in the common case, obviates the need for through-storage transfers. Our first insight is to separate control (function invocation) and data (transfer) paths without impacting the functioning of the autoscaling infrastructure. The challenge is doing so without resorting to a storage service, which is what existing 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

³In practice, services rarely satisfy these assumptions. For example, as of 2024, AWS S3 lacks support for expiration time below one day. Newly added AWS Serverless ElastiCache is metered for a minimum of 1 GB of data stored [11]. Hence, storage costs would account for a larger fraction of expenses in real-world deployments.

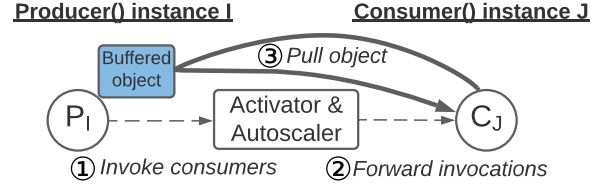


Figure 4: Zipline architecture overview.

seconds. Using the Azure Blob Traces [60], we analyze the time between an object produced by one function and its *last* retrieval by another function of the same application. Figure 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], [62]).

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.

B. Design Overview

We introduce Zipline, a serverless-native data communication fabric that meets all five serverless communication requirements: high performance, compliance with the existing semantics of serverless function invocations, compatibility with autoscaling, and the standard data-transfer API in serverless.

Following the insights developed in Sec. III-A, Zipline splits the function invocation plane into control and data planes. Crucially, the control plane is unchanged, matching the existing serverless architecture (Figure 1), thus allowing the autoscaling infrastructure to take the load balancing decisions for each incoming invocation by steering the invocation to the least-loaded instances of a function. The control plane carries only the function invocation control messages, i.e., RPCs. The data plane is responsible for transferring the objects.

In simplest terms, a producer function instance in Zipline buffers the data to be communicated to the consumer(s) in its own memory and sends a reference to the data inline with the invocation to the consumer function(s). The consumer(s) then directly *pull* the data from the producer’s memory. Zipline fundamentally replaces push-based data transfers, in which the producer pushes the data through the activator or through a storage layer, with an approach in which the consumer directly pulls the data after the control plane has made its decisions.

Figure 4 describes Zipline operation. Let us assume two serverless functions, a producer and a consumer, each of which may have any number of instances at any point in time. As in the case with existing communication methods, the producer

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

Table I: Zipline API description.

logic invokes the consumer function while passing a data object as an argument. However, in contrast to the existing systems, in Zipline, consumer function invocations travel to the activator separately from their corresponding objects ①, which remain buffered at their source. After contacting the autoscaler as needed, the activator chooses the instance of the consumer function, to which the activator forwards the invocation for processing ②. Once the invocation arrives at the target instance, the instance can pull the object from the producer instance ③, using the reference enclosed in the invocation message.

1) *Zipline Programming Model*: The Zipline programming model features a minimalist yet expressive API (Table I) that supports all three essential communication patterns, namely invoking a function, scattering and broadcasting objects to several consumers, and gathering the output of several functions. The Zipline API is fully compatible with the API supported by production clouds, such as AWS Lambda and S3’s Boto3 [15].

First, Zipline supports the standard blocking API, as in AWS Lambda [18], which is the `invoke()` call that invokes a function by its URL, passing a binary data object `obj` by value. Upon invocation, the API of the Zipline SDK is responsible for buffering the object at the producer side until the consumer function instance, chosen by the autoscaling infrastructure, pulls it. In this case, the consumer function starts processing *after* the object is transferred to the consumer instance.

Zipline also supports the standard non-blocking (asynchronous) interface, which is similar to a common key-value store interface like in AWS S3 [15], namely `get()` and `put()` calls. In contrast to using a storage service, with Zipline, the sender instance of the producer function can finish the invocation before one of the consumer instances retrieves the transmitted object.

To de-couple the function invocation and the data transfer interfaces, Zipline introduces *Zipline references* as a first-class primitive. When the producer function calls `put()`, the runtime returns a Zipline reference to a specific object while retaining an immutable copy of the object.⁴ When the consumer needs to read this object, it calls `get()` that pulls the object from the remote server. Each reference is associated with a user-specified number of retrievals `N` of that object, which complete before the producer instance’s runtime de-allocates the object. From the user perspective, references are just opaque hashes that do not expose any information regarding the underlying provider infrastructure, and that can be neither generated nor manipulated by user code.

⁴Note that during non-blocking transfers, the producer function’s user code allocates the object, with the Zipline SDK only holding references to it.

A Zipline reference also includes a Zipline ID that unambiguously identifies objects created in the same workflow, even when an application’s functions process many invocations concurrently. For example, when multiple instances of the same function in a multi-stage (multi-function) video processing pipeline produce objects, the objects will be guaranteed to have different Zipline IDs. This mechanism is similar to the tracing implementation in distributed tracing frameworks [54].

This programming model allows the seamless porting of serverless applications, e.g., those implemented for AWS Lambda or Knative serverless platforms, with corresponding wrapper functions. To demonstrate the API’s portability, we implemented Zipline SDKs for applications written in Python and Golang, and deployed them in a Knative cluster.

2) *Zipline Semantics & Error Handling*: Function invocations in modern serverless offerings, like AWS Lambda and Azure Functions, provide the *at-most-once* semantics [30], [40], [41], i.e., an invocation may execute not more than once even in the presence of a failure.⁵ Hence, the provider is responsible for exposing the runtime errors to the user logic to handle them [16], [17], [21], [50]. Error handling logic in today’s applications varies based on the function composition method. The user can compose the functions as a direct chain (e.g., the producer makes a blocking call to the consumer) or chain the functions in an asynchronous workflow. In the latter case, an *orchestrator* invokes the functions within the workflow. The orchestrator can be provider-based (e.g., AWS Step Functions [14] and Azure Durable Functions [48]) or an auxiliary function that drives other functions. Handling failures may require re-execution of several functions. In this case, the first function of the sub-workflow must be re-invoked with the same arguments as the original invocation. Hence, the user is responsible for passing the first function’s context throughout the sub-workflow to the function that can detect its failure.

Handling of Zipline-related failures follows the same approach. We describe a Zipline failure scenario in a two-function workflow with one producer function and one consumer function, which can be generalized to an arbitrary workflow. Crucially, the lifetime of a Zipline object is connected to the lifetime of the producer instance, thus a shutdown of a producer instance leads to immediate de-allocation of all the objects, retrievals of which have not completed.

For blocking invocations, i.e., the ones invoked with the `invoke()` call, the producer instance stays alive waiting for the response from the consumer, and may decide to re-invoke the consumer invocation if the previous invocation returns an error. For the non-blocking invocations, a Zipline transfer may fail if the producer instance is killed (e.g., due to exceeding the maximum invocation processing time) before a consumer instance retrieves the transmitted object. For example, the producer function may return success before the transfer is complete, followed by the instance shutdown. However, in this case, the consumer function receives the corresponding error

⁵The user can construct primitives with at-least-once semantics by combining primitives with at-most-once semantics and re-try logic. Prior work also shows constructing primitives with the exactly-once semantics [41].

when executing Zipline `get()`. The invocation of the consumer can follow the at-least-once semantics approach. To guarantee the correct execution of the entire workflow, the consumer must re-invoke the workflow starting from the producer function. Hence, the user code in the consumer function should forward this error to the corresponding entity (i.e., the orchestrator or the driver function) that can re-invoke the producer with the same original arguments. For example, with the AWS Step Functions orchestrator, the user can define a custom fallback function to handle particular errors [17].

As an alternative to offloading Zipline timeout error handling, providers may consider modifying their runtime (which is aware of the grace period before the instance shutdown) to back up the remaining non-retrieved objects to cloud storage transparently to the application code, by converting Zipline references into the provider’s storage service keys. For example, when the cluster manager asks the producer instance to shut down (e.g., Knative instances have at least one minute to shut down [59]), the instance can store remaining, non-consumed Zipline objects in the storage service before gracefully terminating. Unaware of the producer instance shutdown, the consumer instance can first attempt a regular Zipline retrieval followed by a retrieval from the storage service using the same key if the first retrieval returns an error. If an application repeatedly faces Zipline errors, the infrastructure can disable Zipline for these functions.

To summarize, Zipline is fully compliant with the existing at-most-once semantics of serverless function invocations and can be enhanced to at-least-once semantics using existing serverless infrastructure by introducing error handling in application logic. Alternatively, these errors can be handled by the runtime with minimal modifications.

IV. IMPLEMENTATION

We prototype Zipline in vHive [69], an open-source framework for serverless experimentation that is representative of production clouds. vHive features the Knative programming model [4] where a function is deployed as a containerized HTTP server’s handler (further referred to as *function server*), which is triggered upon receiving an HTTP request, i.e., RPC, sent to a URL assigned to the function by Knative. Each function instance runs in a separate Kubernetes pod atop a worker host (bare-metal or virtualized) in a serverless cluster.

A. Zipline Prototype in vHive/Knative

We start by describing the implementation of the different software layers of the prototype, required to support blocking function invocations with Zipline, followed by a discussion of support for the non-blocking Zipline API.

1) *Zipline Software Development Kit (SDK)*: Zipline relies on an SDK to implement the API, bridging the user logic and the provider components that perform the transfer. At the producer instance’s side, the SDK splits the original invocation request into two messages, namely a control message and an object, which comprises the transferred data. The SDK creates and adds a Zipline reference to the gRPC request as

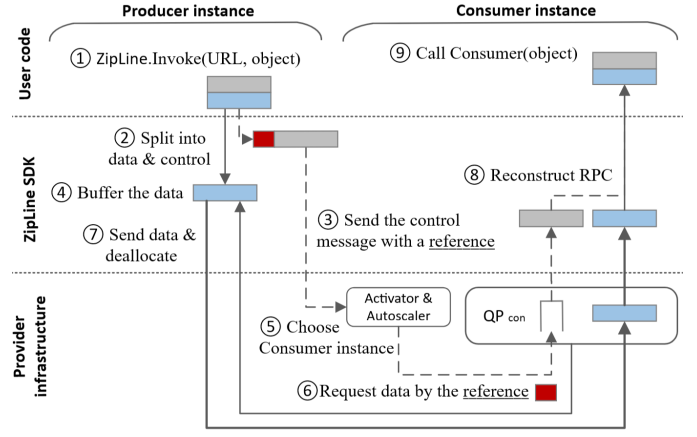


Figure 5: Zipline operation in a single producer single consumer scenario (only the request path is shown). Dashed arrows show the control plane, solid lines show the data plane, and the thick solid lines show data streaming in the data plane.

an HTTP header. The reference comprises an encrypted string, containing the IP address of the pod where instance’s function server is running, and the object key, which is unique for that pod. Encryption prevents the user code from obtaining the IP addresses of function instances.

At the consumer instance’s side, the SDK reconstructs the original request, joining the control message and the object (after the latter has been pulled), before invoking the consumer function in the same way as with the vanilla serverless API.

2) *Control and Data Planes*: Zipline uses gRPC [2] for the control plane, a common industry choice that retains compatibility to the rest of HTTP-based control-plane components. 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. Both protocols support a wide range of programming languages.

3) *Provider Components Extension*: We extend the Knative queue proxy (QP) for object buffering (§II-A). QP is an auxiliary provider container. It is deployed per function instance and shares the pod with the function server. The added logic increases the QP memory footprint by 2MB.

Because a QP, being a minimal provider container, might be online long before the function server during a cold start, we deploy the following performance optimization. We let the QP retrieve the object on behalf of the consumer function server, instead of the consumer SDK, to overlap retrieving the request with booting the function instance.

B. Zipline Operation

1) *Zipline `invoke()` Operation*: Figure 5 shows the request path in the Zipline infrastructure following an `invoke()` call. ① when the caller function needs to call another function it invokes the SDK. ② the SDK splits the request into two parts, the Zipline object and the control plane message that carries the reference to the object. ③ the SDK sends the control message to the activator and ④ stores the object into

a buffer to be fetched later by the consumer's QP (QP_{con}).
 ⑤ the activator chooses the instance of the consumer and forwards the control message to the consumer's QP (QP_{con}).
 ⑥ QP_{con} extracts the reference from the header, decrypts the reference to extract IP address and the object key, and requests the data by sending a Cap'n Proto RPC request to the producer function's SDK, requesting the data by the object key. ⑦ SDK at the producer function sends the data to the QP_{con} and de-allocates the object when they are dispatched. ⑧ QP_{con} forwards the object to the SDK that reconstructs the original request, and ⑨ invokes the function handler. If the response is small, it follows the reverse control plane path through the two QPs and the activator.

2) *Zipline `get()` / `put()` Operation:* Whereas `invoke()` is a synchronous call, the two other calls of the Zipline API – `put()` and `get()` – are asynchronous. While the operation of `put()` and `get()` is similar to `invoke()`, there are a few important differences. The first difference is that `put()` returns a Zipline reference for the object to the user logic. The producer function may pass this reference, like any other string field, to any function that belongs to the same user. Once the consumer function calls `get()` using the delegated reference, the SDK retrieves the object by sending a Cap'n Proto RPC request directly to the producer instance (i.e., to a Cap'n Proto RPC server inside the SDK), using the IP address and the key in the reference.

The asynchronous `get()` / `put()` API can be used not only for invocations but also for large responses as well. The response path follows the control plan path in the reverse order and is used only with small (inline) replies, i.e., <6MB in AWS Lambda. In the case of a large reply, the Zipline-enabled consumer creates a reference to the response object through a `put()` call and includes the reference in the response. Upon receiving the response, the producer can retrieve the response payload through a `get()` call.

C. Flow Control

The Zipline design relies on the availability of the pre-allocated buffer in the QP_{con} component to offer high-performance data transfers. If buffers are unavailable, the system needs to engage a flow control mechanism to pace the sending components before the downstream buffers free up. Fortunately, a Cap'n Proto RPC works on top of TCP and can rely on its flow control without any changes to the Zipline logic, which only needs to buffer and forward the object's chunks along the component chain. Hence, if the number of transmitted objects exceeds the number of available buffers, the subsequent transfers are paused, resulting in the user code blocking in the corresponding Zipline API call.

V. METHODOLOGY

1) *Evaluation Platform:* We prototype and evaluate Zipline in Knative [4], which is widely used in commercial offerings [38] and also representative of the leading close-source serverless platforms [45]. We deploy a Knative cluster that features Zipline-enabled queue-proxy containers on AWS EC2

nodes, similarly to prior work [37], [47], [72], thus ensuring low access time to AWS S3. We use a multi-node cluster of `m5.16xlarge` instances in the 'us-west-1' availability zone, to evaluate the baseline and the Zipline-enabled settings. This instance features Intel Xeon Platinum 8000 series 3.1GHz with 64 SMT cores, 256GB RAM, EBS storage, and a 20Gb/s NIC.

Using the vHive experimentation framework v1.4.2 [69], we set up Knative 1.3 in a multi-node Kubernetes 1.23 cluster [6], running all deployed functions, Knative autoscaling components, and Istio ingress [3]. The pods are scheduled on nodes to ensure all the data transfers happen across the network (i.e., no local communication), placing each function on a separate AWS EC2 node. In all experiments, we emulate a stable serverless workflow where active instances are always present – i.e., there are no cold starts during the measurements.

2) *Measurement Framework:* We use the measurement infrastructure integrated with the vHive framework [7], which supports end-to-end benchmarking. The vHive framework features a service, called invoker, that injects requests in a common format for all of the studied workloads and waits for the responses from the corresponding workflows, reporting the end-to-end delays. The user code of workloads is annotated with logs, which are then aggregated to determine the end-to-end latency breakdown. 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* of a data transfer by dividing the transferred object size by the measured end-to-end latency.

3) *Baseline and Zipline Configurations:* Our baseline is the through-storage communication approach. We evaluate two options for the storage service that represent the lower and higher performance *and* cost bounds of the previously proposed multi-tier ephemeral storage solutions [37], [47], [57], [60], [65]. The first is Amazon S3, which is the baseline configuration used in prior work and represents the cheapest, albeit slowest, storage option in today's clouds. The second is ElastiCache, a cloud-native in-memory data store that offers the fastest, albeit most expensive, storage solution today, $\sim 100\times$ expensive compared to S3.⁶ As shown in prior work [36], [37], ElastiCache provides extremely high performance for inter-function communication but at a high monetary cost as compared to S3. For ElastiCache, we used a single-node Redis cache of the node type `cache.m6g.16xlarge` having 64 vCPUs with 25 Gb/s NIC, which is one of the peak performance configurations priced at \$4.7 per hour.

4) *Microbenchmarks:* We use a number of microbenchmarks, implemented in Golang 1.18, each of which evaluates one of the data transfer patterns commonly used in serverless computing (§III-B1), namely producer-consumer (1-1), scatter, gather, and broadcast. All these patterns comprise various numbers of instances of the producer and the consumer functions communicating one or more objects from the former

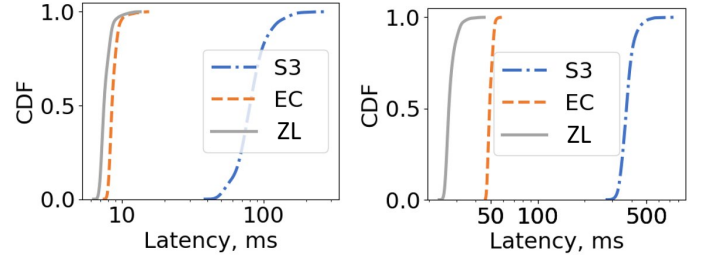
⁶We configure ElastiCache in the on-demand mode, which is $>4\times$ cheaper than its newly-added serverless mode [11].

to the latter. From here on, by saying a producer (consumer), we mean a producer (consumer) function instance.

5) *Real-World Workloads*: We use three data-intensive applications from the vSwarm benchmarking suite [8], which features representative workloads widely used in serverless computing, with their reference inputs. Each workload is comprised of multiple functions, deployed with Knative Serving [5], that call one another using the blocking interface, i.e., a caller function waits for the callee to respond. Each of the workloads uses one or more data transfer patterns to communicate across functions. We modify the workloads to support Zipline along with the S3-based and ElastiCache baselines using the same communication API: `invoke()`, `get()` and `put()` (§III-B1).

The studied workloads have different communication patterns. *Video Analytics (VID)* shows the 1-1 and scatter patterns, as it features a pipeline of video streaming, frame decoder, and object recognition functions; where the frame decoder function invokes the object recognition function once for several frames in a decoded fragment in the scatter communication pattern. *Stacking Ensemble Training (SET)* is a distributed ML training application that fits the serverless programming model well due to its speed, low memory footprint, and low computational complexity [24], [25], [58]. In this workload, the first function broadcasts the training dataset when invoking several training tasks in parallel, and the last function gathers and reconciles the trained ensemble model. Hence, this workload’s execution highly depends on the efficiency of the broadcast and gather communication patterns. Finally, the *MapReduce (MR)* workload implements the Aggregation Query from the representative AMPLab Big Data Benchmark [55]. The gather pattern’s execution is critical for the MapReduce workload, due to the data-intensive shuffling phase between the mapper and the reducer functions.

6) *Cost Model*: We estimate the cost of executing the applications we study from the application developer’s perspective, according to the AWS pricing models [11]–[13]. The price of a single function invocation, from the perspective of an application developer, comprises a small fixed fee for invoking a function, another fee proportional to the product of the processing time and maximum memory footprint of that invocation, and the cost of storage for transferring the data. For all studied functions, we assume the maximum function memory footprint of 512MB, and use the processing times as measured in §VI-B. Storage costs are billed on the GB/month (AWS S3 [12]) or GB/hour (on-demand AWS ElastiCache [11]) bases. In our cost model, we take the minimal possible price for storing transferred data, assuming that ephemeral storage de-allocates transferred data immediately after the last retrieval. Note, this is rarely the case in today’s production systems: e.g., AWS S3 lacks support for expiration time below one day as of 2024, effectively leaving expired objects cleaning to the application.



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

Figure 6: Transfer latency cumulative distribution functions (CDFs) for S3, ElastiCache (EC) and Zipline in the 1-1 workflow. Note the log. scale on the horizontal axis.

VI. EVALUATION

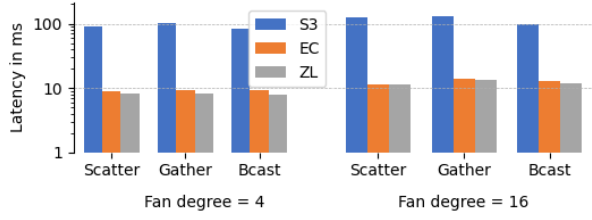
We compare Zipline to through-storage transfers based on Amazon S3 and ElastiCache (EC). We first study the performance of the evaluated communication mechanisms on microbenchmarks. We then assess the performance and cost of real-world applications running in a serverless cloud.

A. Microbenchmarks

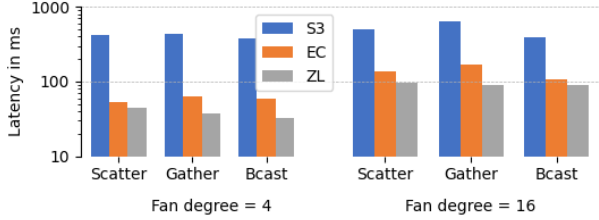
Next, we quantify the latency and effective bandwidth of Zipline in common communication scenarios (§V-4): 1-1, gather, scatter, and broadcast. The 1-1, or producer-consumer, pattern is typical of chained function invocations accomplished via the `invoke()` API call. Gather, or reduce, is essential for applications with functions whose input is the output of several other functions and that use the `put()/get()` API. Scatter, or map, is important when functions have a large fan-out of calls to other functions, passing the objects via the `invoke()` and the `put()/get()` APIs. Broadcast is used by functions that distribute the same data among many consumers, accomplished via a single `put()` call followed by multiple `get()` calls with the *same* S3 key or Zipline reference.

1) *Producer-Consumer Communication*: We focus on the 1-1 (producer-consumer) pattern to study the latency characteristics of the communication methods. Latency is a key metric for interactive, user-facing cloud services, with both median and tail latency considered critical.

Figure 6 plots the median and tail (99th percentile) latency for S3, ElastiCache and Zipline-based transfers for 10KB (small) and 10MB (large) objects. For small objects, transfers through ElastiCache in-memory cache offer much lower latency than transfers through S3, a cold storage service. The median (tail) latency with ElastiCache is 89% (92%) lower than that with S3. Zipline offers a further improvement compared to ElastiCache, with median (tail) latency 12% (10%) lower than ElastiCache. Zipline has better latency than transfers through S3 and ElastiCache because Zipline avoids writing and reading the object on intermediate nodes. For large objects, the median (tail) latency of the ElastiCache-based transfers is 87% (90%) lower than the S3-based ones. Zipline shows median and tail transfer latencies 45% and 34% shorter than those with ElastiCache. Larger object sizes incur higher write and read latencies while transferring the objects through third-party services, which



(a) 10KB object transfers



(b) 10MB object transfers

Figure 7: Transfer latency of the scatter, gather, and broadcast communication patterns with the fan degrees of 4 and 16. Note that both subfigures use a logarithmic scale on the vertical axis, but the scales differ across subfigures.

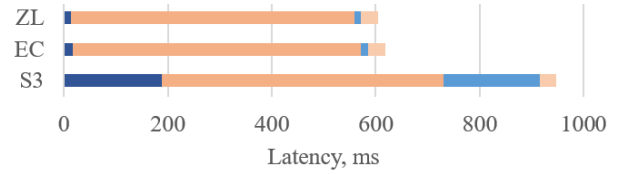
explains the performance advantages of Zipline over both S3 and ElastiCache.

2) *Collective Communication*: We evaluate the speed of the collective communication patterns, namely the gather, scatter, and broadcast, by comparing their latency and effective bandwidth, calculated as the size of the transferred objects divided by the end-to-end transfer time. We study fan-in (gather) and fan-out (scatter, broadcast) degrees of 4 and 16, and consider 10KB (small) and 10MB (large) transfer sizes.

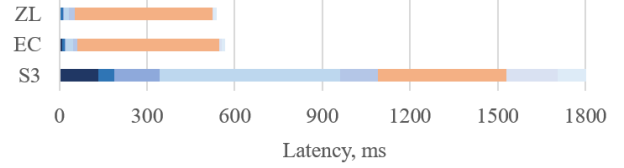
Figure 7a shows the results for S3, ElastiCache, and Zipline transfers of 10KB. For the small transfers, ElastiCache consistently outperforms S3, delivering a latency $9.2\text{--}11.0\times$ lower at the fan degree of 4 and $7.8\text{--}10.8\times$ lower at the fan degree of 16. This result corroborates prior work [37] that also noted that transfers via in-memory storage such as ElastiCache significantly improve performance over transfers via S3. Zipline consistently improves performance over transfers via S3. Zipline consistently matches or outperforms ElastiCache, with a latency up to $1.16\times$ lower than ElastiCache.

These trends persist for larger 10MB transfers as well, shown in Figure 7b). ElastiCache continues to outperform transfers through S3, with the transfer latency up to $7.7\times$ lower. Meanwhile, Zipline improves on ElastiCache by delivering $1.2\text{--}1.9\times$ lower latency.

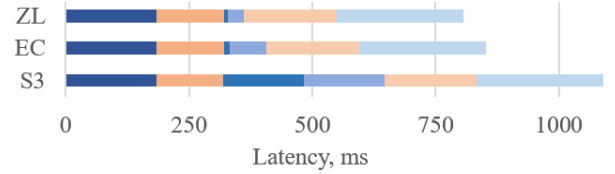
Lastly, Zipline achieves higher effective bandwidth than S3 and ElastiCache. For 10MB transfers with a fan degree of 32, Zipline achieves 16.4Gb/s (82% of the NIC peak bandwidth of 20Gb/s). In contrast, ElastiCache-based transfers deliver 14.0Gb/s (70% of the peak bandwidth) while S3-based transfers deliver 5.5Gb/s (28% of the peak bandwidth).



(a) Video Analytics



(b) Stacking Ensemble Training



(c) MapReduce

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

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

B. Real-World Workloads

Next, we study three data-intensive applications (§V-5), presenting their end-to-end latency along with a detailed breakdown of the sources of latency (Figure 8) and estimating the associated cost (Table II) of executing an invocation for each of the studied applications.

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

Cost. A single invocation processed in a Zipline-enabled system lowers the cost by $3\times$ and $56\times$, compared to S3 and ElastiCache-based configurations, respectively.

2) *Stacking Ensemble Training (SET)*: **Performance.** SET spends 76% and 14% of execution time in communication in the S3-based and ElastiCache-based configuration, respectively. The largest fraction of data communication is the *gather trained models* latency component, accounting for 34% and 4% of the overall execution time in the S3-based and ElastiCache-based configurations, respectively. Using Zipline decreases the gather fraction to 3% of the end-to-end latency, driving the communication fraction down to 12%. Thus, Zipline delivers a $3.4\times$ speedup over the S3 baseline and $1.05\times$ vs. ElastiCache. **Cost.** Zipline is cheaper by $2\times$ and $17\times$, when compared to the S3 and ElastiCache based alternatives, respectively.

3) *MapReduce (MR)*: **Performance.** The workload shows 70% and 62% of execution time spent in communication for the S3 and ElastiCache 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 Zipline. The rest, i.e., 30% of time, are subject to Zipline optimization. Zipline delivers $1.26\times$ overall speedup over the S3 baseline and $1.05\times$ over ElastiCache. Zipline’s speedup is due to a significant decrease in data shuffling, namely mapper-put and the reducer-get phases, which are reduced by $23.4\times$ and $4.8\times$, respectively, compared to the S3 baseline, and by 30% and 55%, respectively, compared to ElastiCache.

Cost. Compared to the two previously-discussed workloads, Zipline reduces the cost of executing MapReduce by $5\times$ and $772\times$ vs. S3- and ElastiCache-based alternatives, respectively. This large cost reduction associated with Zipline is attributable to the large amount of ephemeral data transferred during the shuffle phase of MapReduce, making through-storage/cache transfers particularly expensive.

C. Summary

Zipline enables efficient transfer of ephemeral data across functions without adding cost or complexity to the application logic, delivering high performance, compatibility with existing semantics and API, and native autoscaling. By design, Zipline is much faster than transferring data via conventional storage services, such as AWS S3. Zipline avoids unnecessary writing and reading to the durable tier of storage services, which incurs high latency overheads and carries a monetary cost. Compared to an in-memory cache, Zipline offers similar or better performance *without* the staggering cost or complexity overheads associated with using an additional service. Indeed, our results show that the cost overheads of an in-memory cache are prohibitive, exceeding compute costs by one to two orders of magnitude (Table II). Meanwhile, using an additional service for the caching tier burdens the developer with additional design complexity in the application logic and may require manual reconfiguration (or further application complexity) to accommodate changes in load or data volume. In contrast, Zipline avoids the need for an additional service

and its bandwidth naturally scales with the number of producer and consumer instances.

VII. RELATED WORK

Prior works [37], [47], [57], [60], [65] consider several ephemeral storage service designs, aiming to provide high-performance transfers at a reasonable cost. However, we show in §II-C that the cost of even the slowest tier (e.g., AWS S3 as in several works [37], [47], [57]) can dominate the overall cost of executing a data-intensive application in serverless clouds. Other prior works [26], [35], [51], [60], [63], [64], [70] consider extending serverless with a distributed shared memory (DSM) tier and pass references over the DSM around instead of data objects. In contrast to these proposals, the data objects transmitted via Zipline are immutable, avoiding the complexity of supporting data consistency models. YuanRong [23] is a Huawei production system that can pass data by references, but the paper lacks implementation details.

Other works explore support for *connection-based* direct communication for serverless applications [53], [67], [71], [72], which can help to port microservice and monolith applications to serverless but is generally not used in serverless-native applications [27], [60]. We argue that such optimizations undermine the core principle of serverless, namely the cloud provider’s transparent management of cloud infrastructure, particularly autoscaling. With connection-based approaches, scaling the number of instances of a function up or down is not transparent to its producers and consumers, requiring a reconfiguration of the workflow topology to accommodate. In contrast, Zipline is fully compatible with the object-centric `get()`/`put()` API widely adopted by applications and seamlessly works with the existing cloud autoscaling infrastructure.

Like in the Zipline design, researchers have proposed separating the control and data planes to avoid centralized bottlenecks and deliver high performance. For example, Crab [39] and Prism [32] follow a similar separation to reduce the load on L4 and L7 load balancers, respectively. Dataflower [43] and FUYAO [44] adopt asynchronous transfers to decouple them from the control plane.

Zipline ships a function invocation’s data along the compute for processing, which is complementary to the approaches that ship compute to data or data to compute. Shredder [75] suggests running compute operations directly at the storage tier. Other works [20], [73], [74] investigate the balance between moving data vs. moving compute, suggesting hybrid schemes to combine both.

Zipline enables high-performance transfers without making assumptions on function instances co-location and data locality, which makes it fundamentally different to the following prior works. SAND [10] accelerates data communication proposing a hierarchical messaging bus to facilitate transfers between co-located function instances. FaaSFlow [42], Sledge [46], and Wukong [22] focus on leveraging locality to accelerate the execution of serverless multi-function applications. Night-core [33] suggests exchanging messages over OS pipes for co-located functions. Despite the potential efficiency gains,

today’s commercial systems, e.g., AWS Lambda, tend to avoid serverless function co-location as such placements may lead to hotspots [9], [19], instead relying on statistical multiplexing across a wide server fleet.

VIII. CONCLUSION

The cost and performance of data-intensive serverless applications heavily depends on the efficiency of inter-function data transfers. The state-of-the-art data transfer methods for serverless clouds fall short of serverless applications’ demands. In response, we introduce Zipline, a high-speed API-preserving direct function-to-function communication method that integrates seamlessly with the existing autoscaling infrastructure. Zipline leverages control/data separation and references to provide low latency and high bandwidth for all typical serverless communication patterns. A Zipline prototype reduces the overall cost of executing the end-to-end application invocation by $2\text{--}5\times$ over the through-storage transfers and accelerates real-world serverless applications by $1.3\text{--}3.4\times$. Compared to through-cache transfers, Zipline slashes the cost by $17\text{--}772\times$ while yielding speedups of $2\text{--}5\%$.

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] Istio. Available at <https://istio.io>.
- [4] Knative. Available at <https://knative.dev>.
- [5] Knative Serving. Available at <https://knative.dev/docs/serving>.
- [6] Kubernetes: Production-Grade Container Orchestration. Available at <https://kubernetes.io>.
- [7] vHive Benchmarking Methodology. Available at <https://github.com/eas-e-lab/vhive/blob/main/docs/benchmarking/methodology.md>.
- [8] vSwarm - Serverless Benchmarking Suite. Available at <https://github.com/vhive-serverless/vSwarm/tree/main/benchmarks>.
- [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] Amazon. Invoke API. Available at https://docs.aws.amazon.com/lambda/latest/dg/API_Invoke.html.
- [19] Bharathan Balaji, Christopher Kakovitch, and Balakrishnan Narayanaswamy. FirePlace: Placing Firecracker Virtual Machines with Hindsight Imitation. *Proceedings of the 34th Workshop on Machine Learning for Systems at NeurIPS 2020*, 3, 2021.
- [20] Ankit Bhardwaj, Chinmay Kulkarni, and Ryan Stutsman. Adaptive Placement for In-memory Storage Functions. In *Proceedings of the 2020 USENIX Annual Technical Conference (ATC)*, pages 127–141, 2020.
- [21] 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.
- [22] 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.
- [23] 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.
- [24] Federico Divina, Aude Gilson, Francisco Gómez-Vela, Miguel García Torres, and José F Torres. Stacking Ensemble Learning for Short-term Electricity Consumption Forecasting. *Energies*, 11(4):949, 2018.
- [25] Jie Dou, Ali P Yunus, Dieu Tien Bui, Abdelaziz Merghadi, Meheub Sahana, Zhongfan Zhu, Chi-Wen Chen, Zheng Han, and Binh Thai Pham. Improved Landslide Assessment Using Support Vector Machine with Bagging, Boosting, and Stacking Ensemble Machine Learning Framework in a Mountainous Watershed, Japan. *Landslides*, 17(3):641–658, 2020.
- [26] 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.
- [27] 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.
- [28] 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.
- [29] 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.
- [30] 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.
- [31] Google. Google Cloud Functions. Available at <https://cloud.google.com/functions>.
- [32] 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.
- [33] 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.
- [34] 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.
- [35] 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.
- [36] 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.
- [37] 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.

- [38] Knative. Knative offerings. Available at <https://knative.dev/docs/install/knative-offerings>.
- [39] 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.
- [40] 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.
- [41] 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.
- [42] 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.
- [43] 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.
- [44] 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.
- [45] Qingyuan Liu, Dong Du, Yubin Xia, Ping Zhang, and Haibo Chen. The Gap Between Serverless Research and Real-World Systems. In *Proceedings of the 2023 ACM Symposium on Cloud Computing (SoCC)*, page 475–485, 2023.
- [46] 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.
- [47] Ashraf Mahgoub, Karthick Shankar, Subrata Mitra, Ana Klimovic, Somali Chatterji, 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.
- [48] Microsoft. What are Durable Functions? Available at <https://aws.amazon.com/step-functions>.
- [49] Microsoft. Azure functions, 2019. Available at <https://azure.microsoft.com/en-gb/services/functions>.
- [50] Mikhail Shilkov. Making Sense of Azure Durable Functions. Available at <https://mikhail.io/2018/12/making-sense-of-azure-durable-functions>.
- [51] 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.
- [52] 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.
- [53] 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.
- [54] OpenTelemetry. Traces. Available at <https://opentelemetry.io/docs/concepts/signals/traces>.
- [55] 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.
- [56] 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.
- [57] 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.
- [58] Smitha Rajagopal, Poornima Panduranga Kundapur, and Katiganere Siddaramappa Hareesha. A Stacking Ensemble for Network Intrusion Detection Using Heterogeneous Datasets. *Security and Communication Networks*, 2020, 2020.
- [59] Red Hat. Knative Tutorial: Scaling. Available at <https://redhat-developer-demos.github.io/knative-tutorial/knative-tutorial/serving/scaling.html>.
- [60] Francisco Romero, Gohar Irfan Chaudhry, Iñigo Goiri, Pragna Gopa, Paul Batum, Neeraja J. Yadwadkar, Rodrigo Fonseca, Christos Kozyrakis, and Ricardo Bianchini. FaaS: A Transparent Auto-Scaling Cache for Serverless Applications. In *Proceedings of the 2021 ACM Symposium on Cloud Computing (SOCC)*, pages 122–137, 2021.
- [61] 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.
- [62] Mohammad Shahradd, 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.
- [63] 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.
- [64] 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.
- [65] 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.
- [66] Oren Teich and Pali Bhat. Cloud Run, a Managed Knative Service, is GA. Available at <https://cloud.google.com/blog/products/serverless/knative-based-cloud-run-services-are-ga>.
- [67] 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.
- [68] Dmitrii Ustiugov, Theodor Amariucui, and Boris Grot. Analyzing tail latency in serverless clouds with stellar. In *Proceedings of the 2021 IEEE International Symposium on Workload Characterization (IISWC)*, 2021.
- [69] 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.
- [70] 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.
- [71] 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.
- [72] 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.
- [73] Jie You, Jingfeng Wu, Xin Jin, and Mosharaf Chowdhury. Ship Compute or Ship Data? Why Not Both? In *Proceedings of the 18th Symposium on Networked Systems Design and Implementation (NSDI)*, pages 633–651, 2021.
- [74] Minchen Yu, Tingjia Cao, Wei Wang, and Ruichuan Chen. Following the data, not the function: Rethinking function orchestration in serverless computing. In *Proceedings of the 20th USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, 2023.
- [75] Tian Zhang, Dong Xie, Feifei Li, and Ryan Stutsman. Narrowing the Gap Between Serverless and its State with Storage Functions. In *Proceedings*

of the 2019 ACM Symposium on Cloud Computing (SOCC), pages 1–12, 2019.