Shepherd: Seamless Stream Processing on the Edge

Brian Ramprasad*, Pritish Mishra*, Myles Thiessen*, Hongkai Chen*, Alexandre da Silva Veith*, Moshe Gabel*
Oana Balmau†, Abelard Chow‡, Eyal de Lara*

‡Huawei abelard.chow@huawei.com
†McGill University oana.balmau@cs.mcgill.ca
*University of Toronto {brianr,pritish,mthiessen,chk,aveith,mgabel,delara}@cs.toronto.edu

***Abstract*—Next generation applications such as augmented/virtual reality, autonomous driving, and Industry 4.0, have tight latency constraints and produce large amounts of data. To address the real-time nature and high bandwidth usage of new applications, edge computing provides an extension to the cloud infrastructure through a hierarchy of datacenters located between the edge devices and the cloud.**

Outside of the cloud and closer to the edge, the network becomes more dynamic requiring stream processing frameworks to adapt more frequently. Cloud based frameworks adapt very slowly because they employ a stop-the-world approach and it can take several minutes to reconfigure jobs resulting in downtime.

In this paper, we propose Shepherd, a new stream processing framework for edge computing. Shepherd minimizes downtime during application reconfiguration, with almost no impact on data processing latency. Our experiments show that, compared to Apache Storm, Shepherd reduces application downtime from several minutes to a few tens of milliseconds.

***Index Terms*—stream processing, reconfiguration, late binding, hierarchical edge computing, seamless**

I. Introduction

Next generation applications such as autonomous driving, augmented/virtual reality, smart technologies, interactive games, and Industry 4.0 produce massive scales of data that must be analyzed in a timely fashion [1]. Stream processing frameworks are often used to address this need [2].

Stream processing applications are often structured as a dataflow graph. Vertices can be sources that generate streams of data tuples, or operators that execute a function over incoming data streams. Sinks, are a special type of vertices that consume the processed data but are terminal and represent the end of the flow. Traditionally, all application components are placed in the cloud to take advantage of powerful datacenters. Unfortunately, this approach is not compatible with next generation applications, since sending data over wide-area links to the cloud results in high bandwidth usage and high application latency.

Edge computing expands cloud computing with a hierarchy of computational resources located along the path between the edge and the cloud [1], [3]. In this paper, we argue that efficient use of edge computing for stream processing requires support for seamless reconfiguration and deployment of application operators without disrupting application execution. In particular, throughput should be stable, and latency should not spike during the reconfiguration. Seamless reconfiguration is required to enable efficient resource sharing between applications running on edge datacenters. The smaller size of an edge datacenter, leads to higher costs for storage and computation relative to the cloud (e.g., AWS Wavelength [4] is 40% more expensive than EC2). It is therefore impractical (and may not be even possible due to limited resources) to run all applications continuously on the edge.

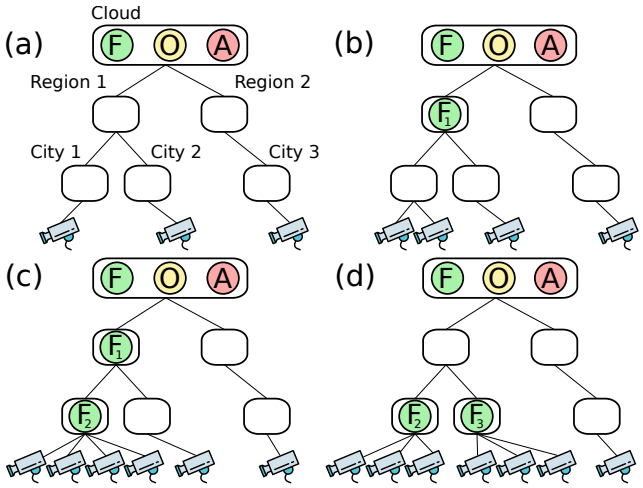

Fig. 1. Seamless Reconfiguration: A video analytics service is dynamically reconfigured to adapt to the change in workload at the edge of the network. This makes more efficient use of the network by reducing the number of video frames transmitted over the WAN.

Figure 1 illustrates the benefits of dynamic reconfiguration for an application running on a hierarchy of edge datacenters with three levels: cloud, region, and city. The application provides analytics about traffic on city roads by performing object detection on video frames produced by a network of motion-activated street cameras. The application consists of a sequence of three operators: [F], a frame filter operator that removes frames that do not significantly change compared to the previous frame; [O], an object detector operator; and finally, [A], an aggregation operator that computes statistics.

Initially, all operators are deployed on the cloud as the number of active cameras is small, and the network cost of transmitting the raw images to the cloud is low (Figure 1a). As more cameras become active in cities [1,2] located in Region 1, network traffic grows and it becomes more cost-effective to filter frames closer to the source by deploying operator [F_1]

978-1-6654-8611-8/22/$31.00 ©2022 IEEE
DOI 10.1109/SEC54971.2022.00011

in Region 1 (Figure 1b). The original [F] operator continues to run in the cloud, where it processes traffic from Region 2. Similarly, operators [O] and [A] are not replicated and continue to run only on the cloud. Operator [O] is CPU intensive and benefits from the cheaper cloud cycles. Operator [A] aggregates data across regions and has to run on the cloud, where it has a global view. As traffic continues to grow in City 1, the application adapts by creating a new replica of the operator [F_2] in this city (Figure 1c). This process is repeated with the creation of [F_3] as more cameras become active in City 2. Finally, the [F_1] operator in Region 1 is removed as it no longer necessary (Figure 1d).

State-of-the-art frameworks, such as Apache Flink [5], Apache Spark [6], and Apache Storm [7] do not support seamless application reconfiguration. These frameworks were designed to run on the cloud, where application reconfiguration is rare. It is therefore not surprising that reconfiguration in these frameworks is implemented as a high-latency, system-wide *stop-the-world* event that involves expensive coordination, often in the form of a global barrier. Global coordination is required because these frameworks use *early-binding* routing, where upstream and downstream operators have direct socket connections. Our experiments show that even for a small deployment of a few nodes, the stoppage time (the interval where the application stops processing data) is measured in tens of seconds and grows with the size of the deployment. Such an approach is incompatible with edge applications, which often have real-time requirements and where reconfiguration is a common occurrence.

This paper introduces Shepherd, a stream processing framework for edge networks. Shepherd enables seamless application reconfiguration with minimal stoppage time. Shepherd's architecture uses a network of software routers to transfer data tuples between operators. Shepherd implements a *late-binding* approach to routing, where an operator does not need to know the location of the next operator that will consume the tuples that it produces. Instead, tuples are *shepherded* to their destination. This approach allows for flexible reconfiguration with minimal stoppage time and without requiring global coordination. As an optimization, Shepherd uses direct network connections (bypassing the router) to transfer messages between operators deployed in the same datacenter; however, this is done without compromising reconfigurability.

We evaluated Shepherd on a hierarchical edge network composed of several Amazon datacenters geographically distributed across North America. Shepherd reduces stoppage time by up to 97.5% compared to Apache Storm. Moreover, in contrast to Apache Storm, Shepherd's stoppage time does not increase with the number of datacenters in the network, network latency, or application size.

In summary, this paper makes the following contributions:

- A stream processing framework for the edge that leverages late binding routing to enable seamless application reconfiguration. We plan to open source the Shepherd implementation;
- A mechanism that mitigates operator slow start;
- An evaluation on an emulated edge network using real-world wide area links.

This paper is structured as follows: Section II provides background on existing stream processing systems focusing on their reconfiguration mechanisms; Sections III, IV and V describe the design and implementation of Shepherd; Section VI presents our experimental evaluation; Sections VII and VIII discuss related work and present our conclusions.

II. Motivation and Background

We first discuss how cloud-based stream processing frameworks (e.g., Apache Storm, Apache Flink) handle reconfiguration. We then discuss previous edge-based stream processing efforts and their limitations.

A. Cloud Frameworks

Stream processing frameworks use a dataflow programming model where an application is represented as a direct acyclic graph (DAG). A DAG consists of vertices that represent operators and edges that represent data streams between operators. Operators can be data sources, sinks, or user-defined functions, such as filtering, aggregation, convolution. Data tuples are produced by data sources and consumed by operators until the data reaches a sink. In edge computing, data sources are often sensors, while data sinks can be placed on cloud datacenters as well as actuators at the network edge.

An application submitted by a user is called a *logical plan*. The stream processing framework transforms the logical plan into a *physical plan* that contains the number of physical instances of each logical operation, and their mapping to computing devices. Reconfiguration changes the physical plan by modifying the number of operator replicas or their mapping to devices. Reconfiguration is typically triggered by changes in the application's workload and is intended to improve performance metrics such as cost, bandwidth, latency and may reduce CO_2 emissions [8].

Reconfigurations can cause significant disruptions to the applications' throughput and latency. To demonstrate this phenomenon, we perform an experiment using a state-of-the-art stream processing framework (Apache Storm) with an application that performs Twitter sentiment analysis. The application consists of six operators [P, F, S, T, C, E] and was deployed on an emulated 2-level edge network consisting of a cloud datacenter and five edge datacenters. Attached to each edge is a group of Twitter users, that collectively produce 400 messages per second. Each emulated datacenter runs on a separate virtual machine (VM), collocated in a single Amazon datacenter, with no extra network latency added.

Figure 2 shows the physical plan at the start of the experiment (a), with all operators [P,F,S, T, C, E] deployed on the cloud. Operators [P, F] are deployed on Edge 1 which has the only active data source. In a second stage (b), the data source attached to Edge 2 becomes active and the application is reconfigured by deploying additional replicas of operators [P, F] on Edge 2. This process is repeated 3 more times until

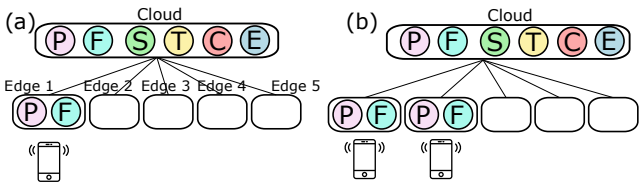

Fig. 2. Operators at the Edge: Multiple changes to the physical plan are needed in the Twitter Sentiment Analysis application to maintain quality of service. Replicas of [P,F] are deployed closer to the users location, to reduce bandwidth.

all 5 sources are active and replicas of the [P, F] operators are running on all edges.

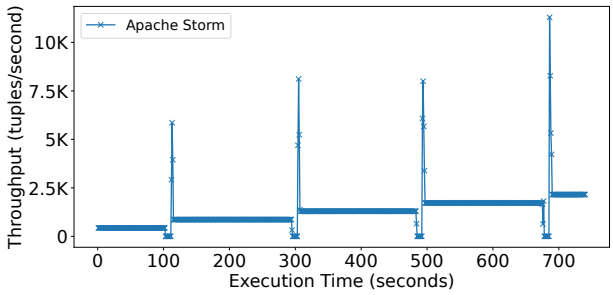

Fig. 3. Job Disruption: During a reconfiguration to add the [P,F] replica set to each edge, the Apache Storm job incurs a stop to apply the changes and this results in disruptions to the throughput.

Figure 3 shows the application throughput in Apache Storm. The throughput increases as more data sources become active; however, each reconfiguration results in significant throughput fluctuations due to stoppage time, as the application stops processing messages with each triggered reconfiguration.

Reconfiguration is a high-latency operation because cloud stream processing frameworks implement a *stop-the-world* approach via a global barrier. During reconfiguration, stream processing frameworks empty all queues, wait for all operators to reach a stable state, make all changes, and finally restart.

Such global coordination is required because these frameworks exploit *early-binding* routing, where upstream and downstream operators have direct socket connections; during a reconfiguration, all socket connections are recreated. Note that long reconfiguration times are acceptable when this event is rare, such as a typical cloud environment. In contrast, reconfiguration is frequent in edge computing applications, which often have real-time requirements and cannot tolerate long stoppage times.

B. Static-Partitioned Edge Frameworks

Applying existing stream processing frameworks to edge computing is difficult because their architectures do not incorporate features to handle the high variation in latency and bandwidth. To achieve good throughput, users need to configure multiple heartbeat values and set the size of low-level network parameters, such as the TCP window size. Moreover, typical flow control in cloud-based stream processing relies on the next operator explicitly requesting the next items, one at a time. This type of flow control is severely obstructed by high-latency reconfigurations.

Previous efforts at enabling stream processing on edge networks have focused on adapting existing frameworks to operate over high latency links [9]–[11]. These approaches statically partition an application into sub-applications, where each sub-application comprises a set of operators and is placed in a different datacenter. The communication between sub-applications happens using a queueing system, such as Apache ActiveMQ, RabbitMQ, or Apache Kafka to transfer tuples from one datacenter to another.

In general, the static-partitioned approach complicates the design and deployment of stream processing applications, as it requires application developers to manually build multiple different sub-applications. In addition, this approach makes reconfiguration slower by requiring coordination between the sub-applications, each of which has to execute its own stop-the-world event.

III. Shepherd

The Shepherd framework transparently deploys and reconfigures latency-sensitive stream processing applications to meet tight SLA guarantees by significantly reducing disruptions caused by job downtime. For ease of use, Shepherd provides a familiar DAG-based dataflow programming model.

A. Overview

Shepherd takes an alternative approach to existing stream processing system design and decouples the transport layer from the processing layer. Existing systems often tightly integrate the communication logic and the runtime logic, which makes it difficult to modify either one without causing a disruption. Shepherd's transport layer is malleable so that quick changes can be made without impacting the processing of tuples.

Figure 4 shows an example application deployment in Shepherd, consisting of three operators [F, O, A]. Our design is based on a hierarchical edge network where nodes communicate through interconnected routers that form a router-network that provides a pathway from the edge to the cloud. The computing hierarchy is organized as a tree, with a traditional wide-area cloud datacenter at its root and an arbitrary number of additional layers of datacenters.

Shepherd uses a master worker architecture where the master runs at the root of the tree and the workers can be deployed anywhere in the hierarchy. Tuples are produced by users on the edge, and are forwarded by the router-network towards the cloud. The framework deploys a minimum of 1 instance of each operator on the cloud datacenter so that the tuples that are propagated to the cloud can always be processed. A tuple is consumed by the first compatible operator along this path. If none are deployed along the path, a tuple is consumed by the cloud replica. For instance, in Figure 4a, a tuple that needs to be processed by operator [F] is sent to the Cloud datacenter.

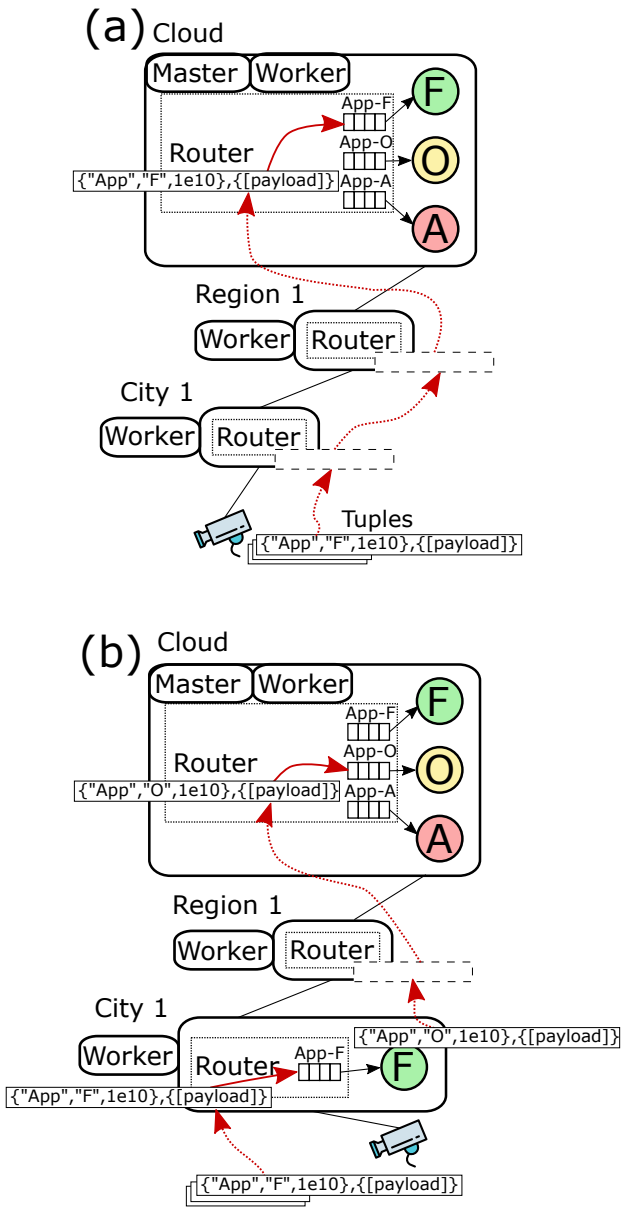

Fig. 4. Late Binding Approach to Stream Processing: Shepherd operators are deployed as a tree along a geo-distributed network hierarchy. The framework uses a *late-binding* design that enables operators to be seamlessly added and removed at runtime *without global knowledge* of existing upstream or downstream operators.

In contrast, in Figure 4b, this same tuple is consumed earlier, as an operator replica for [F] is placed in City 1.

Assumptions and limitations. Shepherd provides at-most-one tuple delivery semantics (i.e., a tuple is delivered only once or not at all). If an operator or router fails, it is restarted and any tuples in flight are dropped. Shepherd, however, provides resiliency to intermittent network failures which are common on the edge. Adding fault tolerance support to Shepherd is the subject of our future work. Moreover, Shepherd currently only supports the reconfiguration of stateless operators. This is not a significant limitation for a wide range of edge computing applications as the operators that benefit the most from running close to the edge (e.g., parsing, filtering, threshold) are typically stateless. Relaxing this limitation, however, is the subject of our future work.

B. Architecture

Shepherd has two major subsystems: the Cloud Master and the Worker, shown in Figure 5. The Master is deployed only at the root of the tree (the cloud) and the Workers can be deployed anywhere (including the cloud). The Shepherd Cloud Master has three major components: (1) *The Job scheduler* which is responsible for accepting user codes and translating it into physical plans, (2) the *Reconfiguration Engine* which is responsible for triggering re-configurations by generating new physical plans, and (3) the *Metrics Monitor* that receives profiling information from across the cluster. This profiling information is used to drive the decisions made by the Reconfiguration Engine.

The Worker contains 2 primary components: (1) the *Job Manager* and the (2) *Router*. The Job Manager receives instructions from the Master to add and remove new operator instances. Jobs deployed on the Shepherd framework have the full application running in the cloud and can process tuples that are not handled by operators running on a lower layer of the edge hierarchy. The Router provides reliable transport over WANs and for this reason it is preferred over simple ssh socket connections which have no built-in resiliency. Once the Job Manager receives an instruction to update the physical plan, it unpacks the payload which contains the user code and allocates a slot on one of its machines in that datacenter. A slot is a unit of compute which is a CPU core and the Job Manager is aware of how many slots it can allocate to ensure the slots are not over-provisioned. The operator code (a Task) is then started and makes a connection to the router that will persist for the life of the operator. In the case where the instruction from the Shepherd Cloud Master is to remove

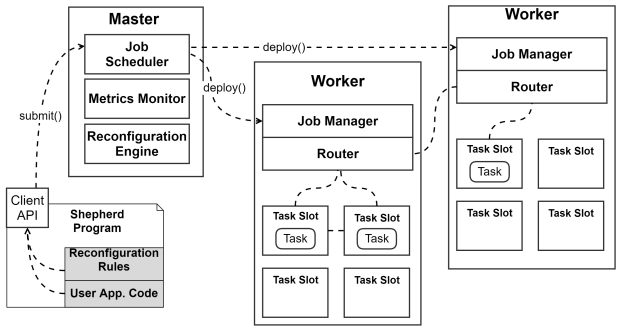

Fig. 5. The Shepherd Architecture: Users submit their application and reconfiguration rules to the Master which then deploys the program as a set of tasks running on the Worker nodes. Tasks can communicate using direct links or between datacenters via the router-network.

a replica, a signal is sent to terminate the operator and clean up the queues if there is no longer any replicas of that type in the datacenter.

C. The Data Model API

Shepherd provides familiar API features of state-of-the-art stream processing frameworks through a new Java API. This API is part of the Client API and it is the mechanism for users to define and submit a logical plan as a DAG to the Job Scheduler. The DAG is composed of a list of source and destination operators, where the user provides a pair of source and destination operators individually, as shown in Listing 1.

```
ArrayList<Pair> dag = new ArrayList<>();
dag.add(new Pair<>("OpF","OpO"));
dag.add(new Pair<>("OpO","OpA"));
dag.add(new Pair<>("OpA","NONE"));
submit(dag);
```

Listing 1. Submission of the logical plan in Shepherd.

The user must also provide user-defined functions written in Java for each one of the operators, as depicted in Listing 2. The operators extend the class *ShepherdOperator*, and the business logic that needs to be applied to incoming tuples is invoked when the *processTuple* function is called.

```
public class OpF extends ShepherdOperator{
    public OpF(){
        \\Constructor
    }
    public void processTuple(Tuple tuple){
       \\User-defined functions
    }
}
```

Listing 2. Operator code in Shepherd.

IV. Seamless Reconfiguration

Seamless reconfiguration in Shepherd is made possible by a new late-binding routing design, an easy to use reconfiguration API and a novel warm up system that collectively reduces disruptions caused by global job reconfiguration events.

A. Late Binding Routing System

Late-binding routing allows the connections between operators to be adaptive and dynamic, where operator replicas can be added or removed without stopping the application flow to (re)create the links between source and destination operators. To make a stream processing system adaptive, producers and consumers need to come and go as needed with the expectation that removing or adding operators will not break the stream processing job. Late binding makes it possible to independently deploy operators without having global knowledge of where producers and consumers are physically located.

To late bind a replica, routing information is needed from incoming tuples. In Shepherd, a tuple carries a header and a body. The header contains base attributes such as the type of tuple (`type_id`), the `topology_id`, and the `timestamp` of when the tuple was created in the data source. The body comprises the payload, which is the application data (e.g., a sensor reading or image frame as a byte array).

Using the `topology_id` and the `type_id`, Shepherd finds an online operator that can consume the data. The Router searches for an operator by reading the `topology_id` and `type_id` and determining if there is a local operator installed in its datacenter that can consume the tuple. Otherwise, the tuple is pushed up to its parent. The decision to route local or up to the parent is the only decision made by the Router whenever any tuple arrives at a given datacenter.

For instance, looking again at Figure 4a, a pipeline application named App contains operators [F, O, A] and is initially deployed in the Cloud. Before the reconfiguration occurs, the [F] tuples are sent to the cloud. After the reconfiguration as depicted in Figure 4b, the [F] tuples get consumed at the City level and the resulting [O] tuples get routed up to the Cloud where they are consumed.

B. The Router: A Modifiable Transport Layer

Routing within the datacenter. The key to the success of the late binding system is that Shepherd performs all the steps required to bring an operator online, and then a final action is triggered to redirect tuples to the new operator replica. To do this, we created a modifiable transport layer that could be manipulated independently of the data processing layer where the tuple processing takes place.

Figure 5 illustrates the deployment process. The Job Manager launches the operators and from this point on, its only function is to decommission the operator if needed. The operators are autonomous, consuming data and publishing metrics directly to a separate data stream that sends metrics data to the cloud. The key point of this design is that the Router can setup complex queue structures *around* the live stream and only introduce this new structure to the tuple flow when all operator code has been downloaded and the operators have acknowledgement back to the Job Manager that they are ready to receive tuples from the router. Current state-of-the-art frameworks deactivate the topology from the moment the reconfiguration is triggered and therefore they incur downtime that consists of downloading the user code, standing up the operator instance and finally activating the network layer to let tuples flow between operators.

Routing between datacenters. Network latency can differ significantly between the cloud datacenter and the edges, depending on the distance in between. Stream processing frameworks that operate within datacenters do not have to deal with this problem because LAN latency is sub millisecond. However, WAN latency can be dozens of milliseconds and this has an impact on the ability to process tuples at high throughput and the network latency can impact the time it takes to stand up new nodes in the network because large amounts of code must be delivered to the target datacenter. To mitigate this problem, Shepherd can change the number of TCP connections at run-time when sending data over the Internet to other datacenters. This allows more packets to be

in flight at once and can be customized per link depending on the amount of latency and available bandwidth on the link.

C. Reducing Operator Cold Start

A common issue with all VM-based programming languages, such as Java, is that they suffer from a cold start because the code is lazily optimized based on how often a particular branch of code is invoked. During the cold start phase the CPU utilization spikes because the Just-In-Time (JIT) compiler is trying to optimize the code [12]. This leaves fewer resources to make progress on the tuple stream and results in longer processing time and fewer tuples processed. This causes the throughput to drop until the JIT compiler has finished optimizing the code. This disruption presents a serious problem for stream processing systems that cannot accept sudden variations and increases to tuple latency because a new cold replica was allowed to process tuples. Shepherd uses a novel warm-up technique to overcome this problem that uses a sample of the live stream of tuples to warm up the operator code *before* the operator starts to participate in contributing to the processing of tuples for the job.

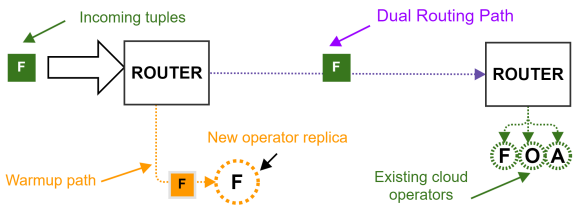

Fig. 6. Cold Operators: A sample of the tuple stream is used to warm up the cold operator to reduce the impact of cold start on the live data stream. The dual route ensures that during the warm-up phase, the live stream continues to flow between datacenters.

Figure 6 illustrates the warming process for a new [F] operator replica that is added to the edge datacenter. The router is reconfigured to provide a sample of the data stream to the operator for a configurable period of time. Once this time elapses, it is assumed that the code inside the operator is now 'hot'. A reset tuple is injected into the data stream and the flow is released behind the reset tuple. When the reset tuple arrives inside the operator, the reset() method is called and the operator begins processing live tuples. The inject and switch operations have minimal impact on the tuple latency and finish in under 20 ms. This feature supports chained warm-ups as part of a multiple replica reconfiguration in the same datacenter.

Stream consistency is maintained during the transition into and out of the warm-up phase by using network queues that preserve ordering. Stream correctness is maintained by using the brief pause periods (20ms) to modify the tuple flow path. This ensures tuples are only transferred to the intended destination once the tuple flow path is un-paused, thus avoiding the chance of duplication.

D. The Reconfiguration API

Shepherd provides an API that allow users to write custom reconfiguration triggers that define scheduling policies that are used to update the physical plan of a running job. The Reconfiguration API is part of the Client API and the custom rules defined by the user can be sent to the Master by invoking the submit() operation which uploaded the trigger logic along with the logical plan. The Reconfiguration Engine inside the Master uses these custom rules to monitor and reconfigure the job by updating the physical plan as defined by the rules and implements the changes by calling the Schedulers *deploy()* operation. For example, in Figure 4 a trigger was activated in the cloud in response to increased traffic at the edge and as result, a replica of type [F] was deployed. The trigger was fired based on the metrics data (e.g., operator replica metrics – latency, throughput, arrival rate, departure rate, and queue size – or computing/network metrics – memory, CPU, and bandwidth) that is transmitted back to the cloud from all the child datacenters. The reconfiguration instructions come strictly from the Shepherd framework instance running in the cloud. One or more triggers can be defined as a package to deploy multiple changes in parallel in the case where many datacenters need to be reconfigured at once. Triggers are defined by the user and are submitted to the framework along with the user code as a package. Each user defined trigger can contain any type of rule which is called in a feedback loop to check whether the current state of the edge hierarchy requires a reconfiguration. Listing 3 shows an example of programming code that extends class *ReconfigurationTrigger* to construct a user-defined reconfiguration trigger (*ruleTriggered*) and a custom scheduling policy (*getExecutionPlans*). The *ReconfigurationStats* and *DataCenterManager* objects contain the collected performance metrics from all datacenters.

Shepherd reconfiguration actions are minimally disruptive, Shepherd only contacts datacenters involved in the reconfiguration. Shepherd performs the reconfiguration by computing the difference between the previous physical plan and the new physical plan to reduce the number of impacted datacenters.

```
public class UserTrigger extends
ReconfigurationTrigger{
 public UserTrigger(String triggerID,
 String topologyID) {
    //Constructor
 }
 @Override
 public boolean ruleTriggered(
 ReconfigurationStats reconfigurationStats){
    //User-defined trigger
 }

 @Override
 public ArrayList<ExecutionPlan>
 getExecutionPlans(DataCenterManager
 dataCenterManager){
    //Scheduling policy
 }
}
```

Listing 3. Example of a custom reconfiguration trigger and scheduling policy.

V. Implementation

We implemented a prototype of Shepherd as a Java application that makes use of Apache ActiveMQ Artemis [13] which is an off the shelf message broker. This broker provides Shepherd with the building blocks necessary to construct our modifiable transport layer. Artemis was chosen over other widely available messaging systems such as RabbitMQ [14] and Kafka [15] because Artemis is optimized for WAN based communication. The processing layer is implemented as Java processes running inside Docker containers.

A. Transport Layer Implementation

The Shepherd Router relies on the Artemis broker to establish an interconnected network that spans the hierarchy from cloud to edge. Message brokers typically implement a variety of queuing structures that can be used to create complex routing paths and mechanisms to adjust the configuration at run-time. Shepherd uses the broker network to transport both user data and management messages.

Separate communication paths are established for user data, restricting management channels for Shepherd only. This ensures that tuples from different domains and data from multiple running jobs are not mixed. User data is transported inside Shepherd tuples, which are then wrapped in the broker messaging protocol to make our tuples compatible with the broker's network stack. This wrapper contains its own header that is used to route messages in the broker network.

The information in the ShepherdTuple header can be extracted and exposed to the broker network message header to route tuples based on user-defined `(key, value)` pairs. This allows the Shepherd Router to consider tuple metadata appended by the operator when making routing decisions.

Rationale for Artemis broker. Artemis is a good choice for creating an edge computing network layer because it has built in support for resilient communication (timeout and automatic retry mechanisms) when working with unstable and high latency links. Most importantly, it supports duplicate checking, which helps ensure correctness in Shepherd. Artemis also supports very large messages (GB) which helps Shepherd to deploy large amounts of user code.

In addition, Artemis provides bridges to interconnect brokers, which provides Shepherd with the ability to setup multiple connections to overcome WAN latency. Security in the transport layer is also provided out of the box (encryption, credentialed access, etc) and any incoming tuples that do not belong to a topology are routed to a dead-letter-address in the cloud for further review. Tuples are only routed to operator instances if the data source (IoT device) can authenticate to the broker and has the unique `topology_id`. Essentially, Artemis functions as proxy for data to reach Shepherd.

B. Intra-Datacenter Communication Optimization

In our initial design of the framework, we used the broker for all communication between operators. This approach has two drawbacks. First, it creates a lot of load on the router. Second, the extra communication adds to the overall end-to-end latency of the application. Since edge computing applications are already subject to fixed high network latency, we optimized our design to reduce the communication overhead when operators in the *same* datacenter need to communicate. ZeroMQ [16], a lightweight message passing library, is used to enable direct point-to-point communication between operator containers collocated on the same datacenter; however, this is done without compromising reconfigurability.

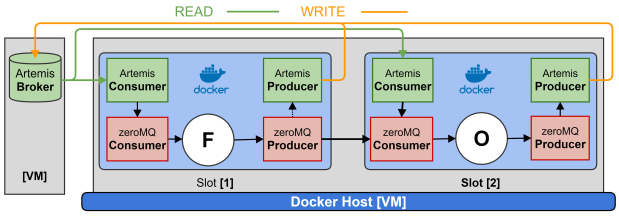

Fig. 7. Optimizing Operator Communication: To improve performance between co-located operators, we designed a Hybrid ZeroMQ and ActiveMQ Artemis approach. Direct connections are used between operators yet they retain the flexibility to be bound late to the router via a dual connectivity pattern.

Operators are implemented as Java applications running inside Docker containers. As shown in Figure 7, operators are simultaneously connected to Artemis and the other operators via ZeroMQ. This dual connectivity pattern allows Shepherd to retain its flexibility to late bind operators because adding the operator to the Artemis broker does not interrupt the stream processing job or the flow of tuples moving through the broker. Since operators can be added or removed at anytime, an important point to note is that when we late bind an operator to an existing deployment, an extra step is needed if the *predecessor* operator is currently writing its output to the broker. Looking again at Figure 7 the [F] operator would have been sending its output to the broker if the [O] operator was not installed. When the [O] operator is added at a later date, it goes through 2 steps to become an active participant in the topology. First, the new replica gets its warm up tuples *directly* from the [F] operator via special sockets that provide a copy of [O] tuples that it is producing. Second, after the warm-up phase, the [O] operator automatically detaches from the warm-up socket, attaches itself to the live output socket of [F], sends a message to the [F] operator via the management channel to stop sending tuples to the broker and instead send its output to the new [O] replica. The new [O] replica then becomes responsible for writing out to the broker. This switch takes about 10ms to occur.

VI. Performance Evaluation

In this section, we conduct an experimental comparison of Shepherd with Apache Storm on emulated edge environments using open source benchmarks and applications. First, we provide a costing analysis that measures the financial benefits of deploying applications with Shepherd. We next empirically evaluate reconfiguration in low network latency scenarios.

Then we measure the effect of WAN latency on reconfiguration events. Next, we examine the reconfiguration performance of existing datacenters as opposed to adding new ones. Also, we evaluate the performance of Shepherds new warm up feature which mitigates the drawbacks caused by Java cold start. Lastly, we compare the performance ActiveMQ vs ZeroMQ to demonstrate the benefits of our hybrid approach for operator communication.

A. Experimental Setup

We conducted our evaluation using an emulated distributed edge network consisting of virtual machines running on 4 Amazon datacenters (N. California, Oregon, Ohio, and Virginia), which we organized as a 2-level hierarchy with N. California as the root and the other 3 datacenters configured as edge nodes as depicted in Figure 8. The figure also shows the measured average round trip times (RTTs) between datacenters.

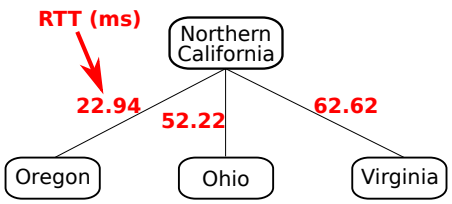

Fig. 8. Experimental Topology: The 2-tier network infrastructure with Oregon, Virginia, Ohio as edges, and the cloud is N. California. The network latencies are the average round trip times (RTTs).

All VMs used in the experiments are of the type m5.2xlarge and have 32 GB of RAM with 8 vCPU/Threads from a 3.1 GHz Intel Xeon® Platinum 8175M processor. All clocks are synchronized using Amazons NTP time service, which is the best practice for time keeping on the AWS infrastructure.

B. Benchmark Applications

To the best of our knowledge, there is no current standard benchmark for stream processing in edge networks. Instead, we use in our evaluation three applications that we believe are representative of functionality that is likely to leverage edge streaming: data filtering, natural language processing, and object recognition. The applications have different workload profiles, code sizes and latency requirements. The code sizes for each application are listed in Table I.

TABLE I
THE SIZE OF THE APPLICATION'S EXECUTABLE

Application	Size in MB
Object Detection - Lite	305
Object Detection - Full	1002
Twitter Sentiment Analysis	54
ETL	54

Object Detection [17], [18]. This application uses video cameras and OpenCV [19] to track vehicles for smart traffic monitoring [20], [21]. The data set used in these experiments is a camera feed of city road traffic obtained from Urban Tracker [22]. The Object Detection application pipeline consists of a chain of 3 operators: frame filter [F] removes frames that are not significantly different from the previous frame; object detection [O] uses a pre-trained YoloV3 [23] model to detect objects in the image frame; car counter [A] aggregates statistics. The [O] operator is CPU intensive and can process just a few frames per second. The Obj Detection app has two versions. The Lite version does not embed the OpenCV library into the application executable, which was done to reduce the payload size of the user executable. Instead, the OpenCV library is pre-loaded directly into Storm's classpath in all datacenters. The Obj Detection Full version embeds the OpenCV library *with* the executable. The purpose of creating 2 versions of this application was to measure the impact of the executable size on the overall reconfiguration time.

Twitter Sentiment Analysis [24]. This application applies NLP to text-based tweets to analyze the polarity of tweets by counting positive and negative words and computing the difference. The Sentiment Analysis application represents a class of applications that produce small tuples at a very high frequency. The tweets are JSON dictionaries, each tweet corresponding to a tuple. The application pipeline consists of a chain of 6 operators: parsing [P], English filtering [F], lowercase and symbol removal [S], remove stop words [T], word counter [C], and positive or negative word score [E]. The data set that we used contains real tweets collected from the global public Twitter API. The regional nature of news broadcasting makes Twitter an interesting application because the geographic context of a Tweet adds to its meaning. For example, recent work has shown that deploying Twitter on the edge can enable location-based top-k popular topic queries and return the result with lower latency as compared to offloading the processing to the cloud [25].

Extraction, Transform and Load (ETL) [26]. This pre-processing application consumes data from smart building environmental sensors to be used in analytics applications that are used to optimize building maintenance and energy usage. The dataset for this experiment is from Sense Your City [27]. The sources are sensors that emit JSON tuples. Processing this information earlier at the edge can reduce the time to find anomalous readings in the data stream. The following pipeline processes the tuples: two filters, range [R] and Bloom [B] filter outliers, interpolation [I] predicts the values that fall within a range and lastly, annotation [A] adds additional meta-data into the observed fields of the tuple. This application has smaller payloads as compared to the Twitter application (integer data vs text data) but has similar data production patterns.

C. Benefits of Dynamic Reconfiguration

We illustrate the benefits of reconfiguration using the object detection application. In this scenario, the application consumes video frames (i.e., standard definition – dimensions 884x498 pixels and average size of 130 KB) from five motion-activated cameras spread across multiple streets in a large

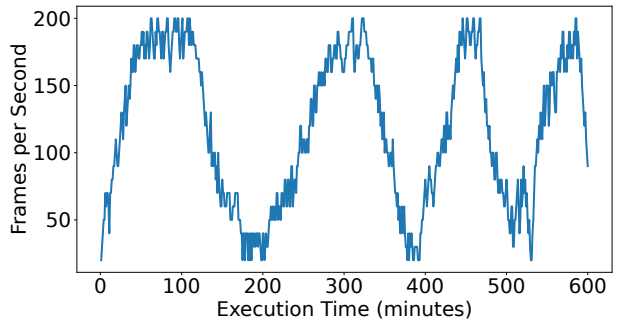

Fig. 9. Application Workload: The workload pattern of 5 different cameras deployed in a large North American city shows large variation across the day.

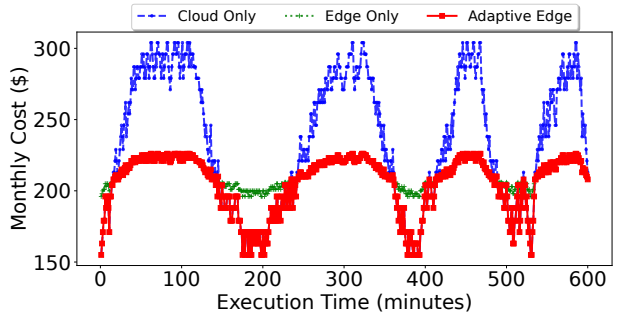

Fig. 10. Monthly Cost: The adaptive edge strategy leverages both the cloud and edge, which results in the lowest monthly average cost as compared to cloud or edge only.

North American city. Each camera emits video frames at different rate as some streets are constantly busy, some are idle, and some are busy on specific hours. Figure 9 demonstrates the skewness of the combined production rate, where the number of active cameras varies over time.

The service provider pays for data transfers and the utilization of computing resources. The communication cost is based on AWS PrivateLink [28] that charges users by bandwidth usage, and the computation cost is priced according to AWS Wavelength [4] for edge datacenters and AWS EC2 for cloud datacenters. Deploying VMs on AWS Wavelength increases the cost by approximately 40%.

We consider three deployment options: Cloud Only, Edge Only, and Adaptive Edge, which dynamically adapts the placement of operators based on the workload. The application has a significant data reduction after the object detection operator [O], which decreases transferred data by over 80%.

Figure 10 shows that deploying the application on Cloud Only, the traditional method, results in the highest cost given the high communication cost of sending data over AWS PrivateLinks when there is high demand. In contrast, Edge Only reduces cost overall, but is more expensive than Cloud Only when the frame rate is low as the AWS Wavelength cost dominate. Adaptive Edge achieves the lowest cost by leveraging the edge on high demand times and the cloud on low demand periods. This adaptive approach reduces the costs by over 13% and 3% when compared to Cloud Only and Edge Only, respectively. The savings are more significant when there is a high fluctuation in the production rate, numerous cameras involved, or dataframes with higher resolution (e.g., HD, 4K, and 8K).

D. Shepherds background reconfiguration time

Shepherd reconfigures in the background before making changes to the running job. The time to reconfigure a job does not cause downtime. Figure 11 is an analysis of the timing of each background step when adding new replicas to a datacenter for the Twitter application and the Object Detection Full Size. These two applications represent small code and larger code bases respectively to show the difference.

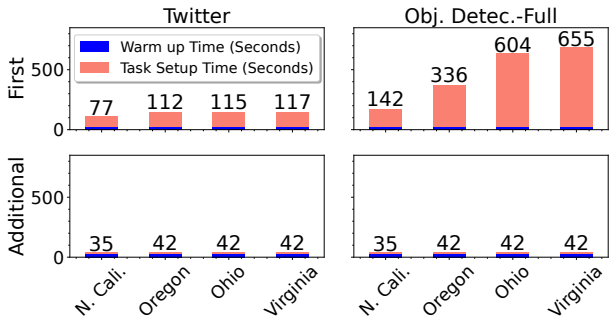

Fig. 11. Reconfiguration Time: Shepherd reconfigures in the *background* because the warm-up time and task setup time do not cause downtime when reconfiguring. The 30-second warm-up time shown here is configurable. The results for ETL (not shown) are similar to Twitter.

Task setup time is the time it takes for the operator code to be downloaded into the new datacenter and for the operator to come online. The results show that Shepherds task setup time within the same datacenter (N. California) takes 107 seconds when installing the first operator and additional reconfigurations take a maximum of 42 seconds to complete in any datacenter. The warm-up time is constant at 30 seconds and it is set by the user but it is a configurable value to allow for longer warm ups if necessary. The framework only incurs a minor disruption of 20ms to update the physical plan once the task setup and warm-up steps have been completed. This transition time is less than 20ms because Shepherd performs reconfigurations in the *background* before applying the changes to the job.

E. Low Latency Case

We evaluated the effect that reconfiguration has on the performance of the Twitter Sentiment Analysis application when running on top of Shepherd and Storm. We run this application on an emulated 2-level edge network consisting of a cloud datacenter and 5 edge datacenters. Attached to each edge is a separate data source that produces 600 messages per second when active, and 0 otherwise. Each emulated

TABLE II
STORM: DOWNTIME, RECOVERY TIME, AND NUMBER OF SLA VIOLATIONS (LATENCY > 100 MS) FOR RECONFIGURATIONS INSIDE A LAN

	2nd	3rd	4th	5th
Downtime (seconds)	9.1	8.6	8.2	8.3
Recovery Time (seconds)	3	3	3	3
Number of SLA Violations	11404	15929	18074	22778

datacenter runs on a separate virtual machine (VM). Since reconfiguration time is affected by high network latency, we consider an idealized best-case scenario where all VMs are collocated in a single Amazon datacenter (N. California).

The reconfigurations were triggered in Storm using the re-balance command. We created a custom Storm scheduler by extending the $IScheduler$ class that can react to the throughput increases and create new physical plans. Likewise, in Shepherd, triggers were created to react to the increase in traffic load and to deploy a new physical plan.

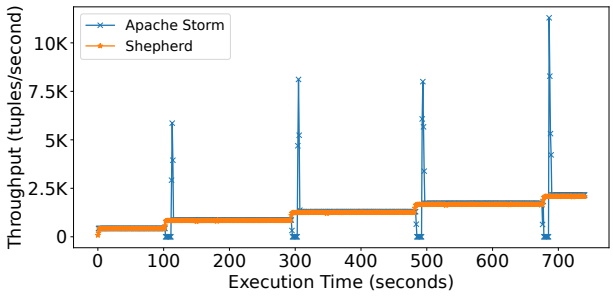

Fig. 12. Effect on Throughput: Shepherd incurs no throughput drops when reconfiguring to add new replicas [P, F] to each edge in response to increased Twitter traffic. Storm incurs downtime of several seconds, with throughput spikes after each reconfiguration.

Results Figure 12 is a comparison between Storm and Shepherd depicting 4 reconfiguration events at intervals of 200 seconds. The physical plan at the start of the experiment has operators [S, T, C, E] deployed on the cloud, and operators [P, F] deployed on edge 1 which has the only active data source. After 200 seconds, the Twitter traffic at edge 2 increases and the application is reconfigured by deploying additional replicas of operators [P, F] on edge 2 (see Figure 2b). This process is repeated 3 more times in response to the increased traffic at each edge and replicas of the [P, F] operators are running on all edges. At each reconfiguration event, Storm *stops* processing tuples for several seconds and the throughput goes to zero. Throughput is the number of tuples that have arrived at the sink (measured per second). In contrast to this downtime, Shepherd continues to process the tuples and the reconfiguration event has little to no impact on the throughput.

Table II shows detailed results for Storm. *Downtime* is defined as the amount of time that has elapsed since the framework has stopped processing tuples; *SLA violations* measure the number of tuples during the experiment that arrived at the sink with an end-to-end latency of *greater* than 100 ms; and *recovery time* reflects the time period in which the framework is working through the backlog of the tuples. Shepherd does not incur enough disruption (>100 ms), and therefore all values for these metrics are zero and not shown in the table for brevity. In contrast, every reconfiguration call in Storm causes a backlog that grows while the framework is in the process of building a new physical plan, uploading operator code to the target worker machines and standing up the operators. This is what causes the throughput spikes and the late tuples measured as SLA violations in Storm. The more edges there are, the larger the number SLA violations occur and they will continue to grow because the Downtime will likely grow as more worker nodes take longer to synchronize when they are added to the Storm cluster.

F. The Effect of WAN Latency

We next asses the effects of WAN latency on application reconfiguration for both Apache Storm and Shepherd. Apache Storm's Zookeeper and Nimbus run in an isolated VM in N. California. In each edge datacenter, a data source produces 400 (Twitter and ETL) or 2 (Object Detection) messages per second when active, and 0 otherwise. When a data source becomes active, a reconfiguration is called to add [P, F] for the Twitter Sentiment Analysis application, [F, O] for the Object Detection, and [P, R] for the ETL to the same datacenter as the data source.

At the beginning of each the experiment all operator instances are in the cloud and there are no sources. We then add 3 sources, 1 at a time and with 200 seconds in-between to let the system stabilize. The addition of a source triggers a reconfiguration each time. We measure then measure the downtime, recovery time and number of SLA violations caused by each of these reconfiguration events.

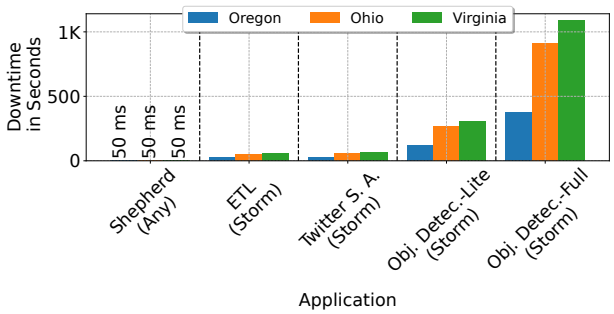

Fig. 13. Downtime: Shepherd downtime is always <50ms when reconfiguring over the WAN, while for Storm the impact of user executable code size and network latency causes downtime of up to many minutes.

WAN Downtime Results Figure 13 shows the downtime measured in seconds of each framework when reconfiguring the job by adding replicas to new edge datacenters. The results show that neither the network latency nor the application sizes affect Shepherd's downtime (less than 50 ms and a 99%

improvement over Apache Storm). This is because the bulk of the reconfiguration process in Shepherd happens in the background. In contrast, for Apache Storm, network latency and the application size have a large impact on downtime. The more interesting result is seen with the OBJ-Lite and OBJ-Full applications, where the differences in downtime are significant. The Twitter application and ETL application have the same size of executable and therefore incur the same downtime.

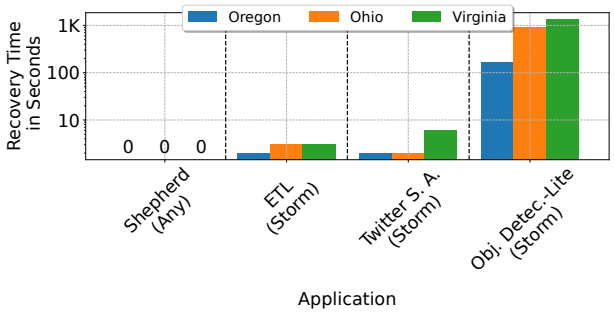

Fig. 14. Recovery Time: Shepherd needs no time to recover after a reconfiguration over the WAN because it incurs <50ms of downtime. Storm spends several minutes in recovery, trying to clear the backlog of tuples.

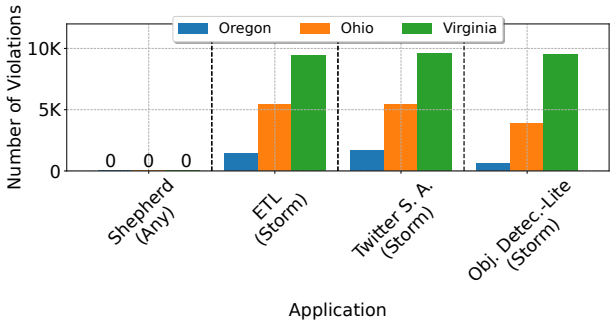

Fig. 15. SLA Violations: Shepherd has no SLA violations (defined as latency $>$ 1 second) after reconfiguring on the WAN, while Storm experiences large numbers of SLA violations (late tuples) because of its long recovery time.

WAN Recovery Time and SLA Results. Figure 14 captures the recovery time in seconds when adding new replicas to each of the edge datacenters. Recovery time is the duration that tuples are being delivered late according to a specified SLA threshold, in this case 1 second. The results show that Shepherd does not incur a recovery time penalty. Figure 15 captures the number of late tuples during the recovery time and the results demonstrate that for Shepherd all tuples are delivered under the SLA threshold of less than 1 second. In contrast, the recovery time for Apache Storm is significant in a geographically distributed environment. The recovery time is a consequence of the downtime, as many tuples pile up during the reconfiguration. Depending on the application, this recovery time is longer if the processing time is large and the operators cannot clear the backlog fast enough to make progress on the tuples that arrived after the reconfiguration. For example, the Object Detection application has a machine learning operator that requires at least 250 ms of compute time to detect objects. The 2, 13, and 16 minute recovery time for the OBJ-lite application likely is not an acceptable amount of downtime under any practical scenario when reconfiguring to any of the three edge datacenters. The amount of late tuples is excessive and grows at similar rate to the LAN experiment, and is further amplified by the WAN latency. These violations can be a deal breaker as many applications cannot afford to deliver late tuples.

G. Moving One Operator Closer to the Edge

This experiment evaluates the impact of a reconfiguration of a job in an *existing* datacenter rather than the addition of replicas to new datacenters which was evaluated in the previous section. The purpose of the experiment is to highlight the fact that even when Storm has all the user code already pre-loaded in a given worker node, adding one more operator still incurs a bit of downtime and some late tuples. The experiment starts with having the Parser operator running on the Oregon datacenter (edge), and the rest of the operators placed to the N. California datacenter. Then, at mark 110 seconds, a reconfiguration pushes down the Range Filter to the edge.

TABLE III
WHEN RECONFIGURING AN EXISTING DATACENTER, SHEPHERD DELIVERS ALL OF ITS TUPLES ON TIME, WHEREAS STORM STILL HAS LATE TUPLES EVEN WHEN THE RECONFIGURATION DOWNTIME IS VERY SHORT

	Shepherd	**Apache Storm**
Recovery Time (seconds)	0	2.6
Downtime (seconds)	.05	2
Number of SLA Violations	0	106

The impact of the reconfiguration between Storm vs Shepherd is depicted in Table III where we measure the recovery time, downtime and number of SLA violations for this simple reconfiguration. Shepherd outperforms Apache Storm by reducing the downtime by 97.50%. This also reduces the recovery time and the number of SLA violations, showing that Shepherd can meet tight SLA latency requirements that Storm cannot.

H. Operator Warm-up

This set of experiments demonstrates the value of having the warm-up feature in Shepherd. The experiment scenario begins with the Twitter Sentiment Analysis application entirely deployed on the N. California cloud datacenter, and at the 120s mark, the [F] operator is migrated to the Ohio datacenter. Figure 16 captures the disruptions to the throughput when there is a reconfiguration event and contrasts the performance when the warm-up feature is turned on.

The results show that when the warm-up feature is turned off there is a large drop in the throughput even at the lower tuple

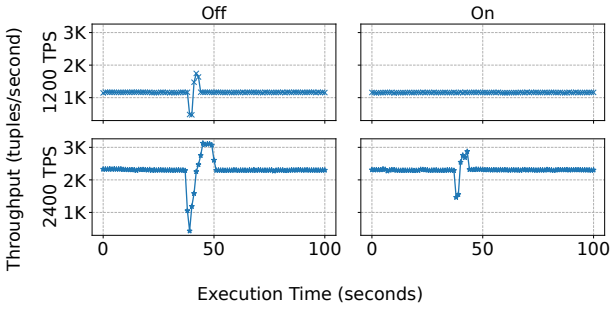

Fig. 16. Warm up Benefits: Without Shepherds warm up feature turned on, the throughput drops are significant. The warm-up feature helps Shepherd meet tight SLA guarantees by reducing disruptions that increase end-to-end tuple latency.

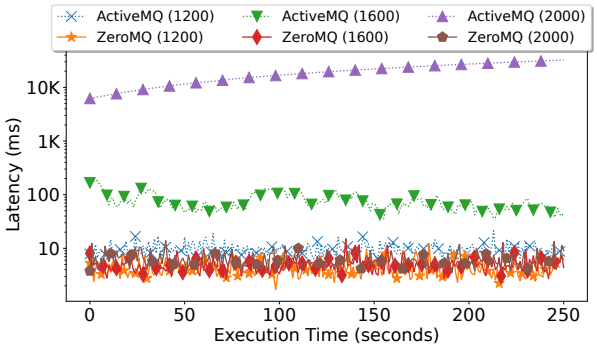

Fig. 17. Tuple Latency Comparison: When comparing the performance between ZeroMQ and ActiveMQ, we can see that using direct connections between operators via ZeroMQ reduces latency because the broker has to route fewer tuples between operators and therefore tuples spends less time sitting in queues waiting to be processed.

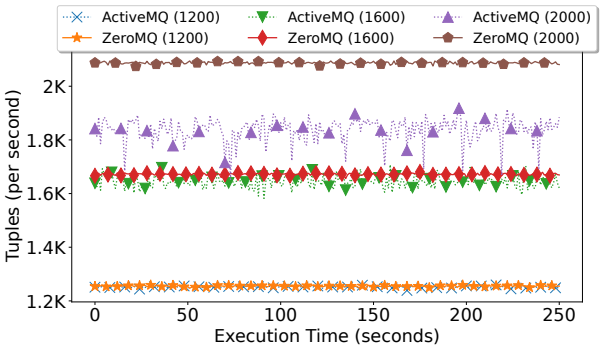

Fig. 18. Tuple Throughput Comparison: By using ZeroMQ as compared to ActiveMQ, the same system resources yield a higher tuple throughput when using ZeroMQ because buffers between operators are more efficiently managed as opposed to the multiple buffer transfers that are needed when sending the tuples through the brokers.

rate of 1200 per second. We chose the rates of 1200 and 2400 because the CPU of the machine reached 80% usage at the 2400 rate. The warm-up feature becomes even more valuable if we double the rate to 2400 because when the feature is turned off there is a throughput drop of 73% and it takes a longer time to return to normal processing levels. The warm-up feature allows new operator replicas to warm-up before they can accept live tuples, and therefore avoiding the cold-start problem of JVMs.

I. Improving Communication within the Datacenter: ActiveMQ vs ZeroMQ

We evaluate the performance gains of using ZeroMQ as compared to ActiveMQ for intra datacenter communication. The ETL application is used in this experiment with three different ingestion rates (1200,1600 and 2000 tuples per second). Figure 17 and Figure 18 show the benefits in terms of lower latency and higher throughput when replacing ActiveMQ with ZeroMQ for communication between operators in the same datacenter. The benefits increase at higher throughput levels as ActiveMQ begins to destabilize when the throughput hits 2000 tuples per second. Whereas, the two methods achieve similar latency and throughput when the system is processing a small number of tuples (1200). Even at the mid-range, the results show that the latency of ActiveMQ is over 93% higher than ZeroMQ when the production rate is 1600 tuples per second. As previously discussed in the Implementation section and shown in Figure 7, our use of ZeroMQ does not affect the reconfigurability of the operators, as they remain connected to the Shepherd Router and can benefit from the features of our modifiable transport layer.

VII. Related Work

Apache Flink [5], Apache Spark [6], and Apache Storm [7] were designed to run on a single monolithic datacenter where reconfigurations happen quite rarely. For this reason, classical stream processing frameworks conduct reconfigurations by using the stop-of-the-world approach [29] where the application is paused, the new physical plan is deployed, and then the application is resumed [30]–[33]. Unfortunately, this approach incurs high downtime since it requires synchronization of barriers to maintain correctness. The stoppage times become longer when such frameworks are deployed on an edge computing infrastructure as they require several rounds of communication to synchronize each step (e.g., all cluster nodes enter in reconfiguration, the operator replicas are reassigned, and the operators' connections are recreated). This multi-round communication synchronization pattern leads to stalls which could be minutes in duration. Prior works, such as R-Storm [34], SpanEdge [35], EdgeWise [36], Trisk [37], E2DF [38], and DART [2], are optimized to run on geographically distributed infrastructure. Some of these solutions also introduce methods that reduce the amount of transferred data to make the reconfiguration faster. However, reconfiguration in these approaches remains a multi-second global operation that requires barriers to synchronize the state of all cluster nodes.

Parallel track [39] arises as an attractive solution to execute seamless reconfiguration. This method creates a new operator

instance that runs concurrently until it synchronizes the old and the new operator replicas. However, the parallel track approach has not been deployed on the WAN due to the challenges with communication channel management and synchronization of multiple physical locations. For example, ChronoStream [40] and Gloss [41] use the parallel-track approach on a single datacenter. The former addresses the reconfiguration without accounting for the slow-start initialization of an operator replica, and it requires that downstream operators control duplicated messages by filtering them out. The latter looks at the reconfiguration as a recompilation process by rebuilding the whole DAG, running it concurrently for some time, and controlling the discarding of duplicate messages – making the reconfiguration more costly as it includes new operations. In contrast, Shepherd's design leverages a parallel track-inspired solution for an edge computing resource-limited shared infrastructure that overcomes the slow-start initialization of a new operator replica without creating the cost of duplicating the whole application DAG.

Finally, this work builds on our workshop paper [42], which first identified the challenges associated with stop-the-world reconfiguration and proposed the use of late-binding routing as an alternative. This earlier work did not provide a full design or implementation, and as a result did not address real-world issues, such as high latency links, operator warm-up, and router overhead. In addition, the workshop paper only presents results from a single experiment based on a simple 2-operator micro-benchmark that moves tuples around but did not do any processing.

VIII. Conclusion

In this paper, we introduced Shepherd, a novel stream processing framework for edge computing. Shepherd offers efficient dynamic placement of operators along a hierarchy of datacenters located between the edge devices and the cloud. Through its novel late binding, hierarchical routing, and warm-up techniques, Shepherd decreases downtime due to reconfiguration from a few minutes to milliseconds. Shepherd enables a wide-range of edge computing applications that rely on stateless stream processing. In future work, we plan to extend Shepherd with policies that automatically adjust the parallelism degree of operators, as well as offering support for stateful stream processing and fault tolerance.

References

[1] M. Satyanarayanan, "The emergence of edge computing," *Computer*, vol. 50, no. 1, pp. 30–39, 2017.

[2] P. Liu, D. D. Silva, and L. Hu, "DART: A scalable and adaptive edge stream processing engine," in *2021 USENIX Annual Technical Conference (USENIX ATC 21)*. USENIX Association, Jul. 2021, pp. 239–252. [Online]. Available: https://www.usenix.org/conference/atc21/presentation/liu

[3] B. Varghese, E. De Lara, A. Y. Ding, C.-H. Hong, F. Bonomi, S. Dustdar, P. Harvey, P. Hewkin, W. Shi, M. Thiele *et al.*, "Revisiting the arguments for edge computing research," *IEEE Internet Computing*, vol. 25, no. 5, pp. 36–42, 2021.

[4] "Wavelength," 2022. [Online]. Available: https://aws.amazon.com/wavelength/

[5] "Apache Flink," 2022. [Online]. Available: http://flink.apache.org/

[6] "Apache Spark," 2022. [Online]. Available: https://spark.apache.org/

[7] "Apache Storm," 2022. [Online]. Available: https://storm.apache.org/

[8] B. Ramprasad, A. da Silva Veith, M. Gabel, and E. de Lara, "Sustainable computing on the edge: A system dynamics perspective," in *Proceedings of the 22nd International Workshop on Mobile Computing Systems and Applications*, ser. HotMobile '21. New York, NY, USA: Association for Computing Machinery, 2021, p. 64–70. [Online]. Available: https://doi.org/10.1145/3446382.3448607

[9] Y. Fu and C. Soman, *Real-Time Data Infrastructure at Uber*. New York, NY, USA: Association for Computing Machinery, 2021, p. 2503–2516. [Online]. Available: https://doi.org/10.1145/3448016.3457552

[10] D. Battulga, D. Miorandi, and C. Tedeschi, "Fogguru: a fog computing platform based on apache flink," in *2020 23rd Conference on Innovation in Clouds, Internet and Networks and Workshops (ICIN)*, 2020, pp. 156–158.

[11] E. G. Renart, D. Balouek-Thomert, and M. Parashar, "An edge-based framework for enabling data-driven pipelines for iot systems," in *2019 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*, 2019, pp. 885–894.

[12] A. Khrabrov, M. Pirvu, V. Sundaresan, and E. de Lara, "JITServer: Disaggregated caching JIT compiler for the JVM in the cloud," in *2022 USENIX Annual Technical Conference (USENIX ATC 22)*. Carlsbad, CA: USENIX Association, Jul. 2022. [Online]. Available: https://www.usenix.org/conference/atc22/presentation/khrabrov

[13] B. Snyder, D. Bosanac, and R. Davies, "Introduction to apache activemq," *Active MQ in action*, pp. 6–16, 2017.

[14] "RabbitMQ," 2022. [Online]. Available: https://www.rabbitmq.com/

[15] "Apache Kafka," 2022. [Online]. Available: https://kafka.apache.org/

[16] "ZeroMQ," 2022. [Online]. Available: https://zeromq.org/

[17] X. Zeng, B. Fang, H. Shen, and M. Zhang, *Distream: Scaling Live Video Analytics with Workload-Adaptive Distributed Edge Intelligence*. New York, NY, USA: Association for Computing Machinery, 2020, p. 409–421. [Online]. Available: https://doi.org/10.1145/3384419.3430721

[18] A. Anjum, T. Abdullah, M. F. Tariq, Y. Baltaci, and N. Antonopoulos, "Video stream analysis in clouds: An object detection and classification framework for high performance video analytics," *IEEE Transactions on Cloud Computing*, vol. 7, no. 4, pp. 1152–1167, 2019.

[19] "OpenCV," 2022. [Online]. Available: http://opencv.org/

[20] Q. Zhang, H. Sun, X. Wu, and H. Zhong, "Edge video analytics for public safety: A review," *Proceedings of the IEEE*, vol. 107, no. 8, pp. 1675–1696, 2019.

[21] G. Ananthanarayanan, P. Bahl, P. Bodík, K. Chintalapudi, M. Philipose, L. Ravindranath, and S. Sinha, "Real-time video analytics: The killer app for edge computing," *Computer*, vol. 50, no. 10, pp. 58–67, 2017.

[22] J.-P. Jodoin, G.-A. Bilodeau, and N. Saunier, "Urban tracker: Multiple object tracking in urban mixed traffic," in *IEEE Winter Conference on Applications of Computer Vision*, 2014, pp. 885–892.

[23] J. Redmon and A. Farhadi, "Yolov3: An incremental improvement," 2018, cite arxiv:1804.02767Comment: Tech Report. [Online]. Available: http://arxiv.org/abs/1804.02767

[24] M. V. Bordin, D. Griebler, G. Mencagli, C. F. R. Geyer, and L. G. L. Fernandes, "Dspbench: A suite of benchmark applications for distributed data stream processing systems," *IEEE Access*, vol. 8, pp. 222 900–222 917, 2020.

[25] A. Jonathan, A. Chandra, and J. Weissman, "Locality-aware load sharing in mobile cloud computing," in *Proceedings of The10th International Conference on Utility and Cloud Computing*, ser. UCC '17. New York, NY, USA: Association for Computing Machinery, 2017, p. 141–150. [Online]. Available: https://doi.org/10.1145/3147213.3147228

[26] A. Shukla, S. Chaturvedi, and Y. Simmhan, "Riotbench: An iot benchmark for distributed stream processing systems," *Concurrency and Computation: Practice and Experience*, vol. 29, no. 21, p. e4257, 2017, e4257 cpe.4257. [Online]. Available: https://onlinelibrary.wiley.com/doi/abs/10.1002/cpe.4257

[27] "Urban environmental monitoring project," 2022. [Online]. Available: http://map.datacanvas.org

[28] "AWS PrivateLink," 2022. [Online]. Available: https://aws.amazon.com/privatelink

[29] V. Cardellini, F. Lo Presti, M. Nardelli, and G. Russo Russo, "Optimal operator deployment and replication for elastic distributed data stream processing," *Concurrency and Computation: Practice and Experience*, vol. 30, no. 9, p. e4334, 2018, e4334 cpe.4334. [Online]. Available: https://onlinelibrary.wiley.com/doi/abs/10.1002/cpe.4334

[30] F. Kalim, L. Xu, S. Bathey, R. Meherwal, and I. Gupta, "Henge: Intent-driven multi-tenant stream processing," in *Proceedings of the ACM Symposium on Cloud Computing*, 2018, pp. 249–262.

[31] S. Yi, Z. Hao, Q. Zhang, Q. Zhang, W. Shi, and Q. Li, "Lavea: Latency-aware video analytics on edge computing platform," in *Proceedings of the Second ACM/IEEE Symposium on Edge Computing*, 2017, pp. 1–13.

[32] L. Xu, B. Peng, and I. Gupta, "Stela: Enabling stream processing systems to scale-in and scale-out on-demand," in *2016 IEEE International Conference on Cloud Engineering (IC2E)*. IEEE, 2016, pp. 22–31.

[33] L. Xu, S. Venkataraman, I. Gupta, L. Mai, and R. Potharaju, "Move fast and meet deadlines: Fine-grained real-time stream processing with cameo," in *18th USENIX Symposium on Networked Systems Design and Implementation (NSDI 21)*, 2021, pp. 389–405.

[34] B. Peng, M. Hosseini, Z. Hong, R. Farivar, and R. Campbell, "R-storm: Resource-aware scheduling in storm," in *Proceedings of the 16th Annual Middleware Conference*, ser. Middleware '15. New York, NY, USA: Association for Computing Machinery, 2015, p. 149–161. [Online]. Available: https://doi.org/10.1145/2814576.2814808

[35] H. P. Sajjad, K. Danniswara, A. Al-Shishtawy, and V. Vlassov, "Spanedge: Towards unifying stream processing over central and near-the-edge data centers," in *2016 IEEE/ACM Symposium on Edge Computing (SEC)*, 2016, pp. 168–178.

[36] X. Fu, T. Ghaffar, J. C. Davis, and D. Lee, "Edgewise: A better stream processing engine for the edge," in *Proceedings of the 2019 USENIX Conference on Usenix Annual Technical Conference*, ser. USENIX ATC '19. USA: USENIX Association, 2019, p. 929–945.

[37] Y. Mao, Y. Huang, R. Tian, X. Wang, and R. T. B. Ma, *Trisk: Task-Centric Data Stream Reconfiguration*. New York, NY, USA: Association for Computing Machinery, 2021, p. 214–228. [Online]. Available: https://doi.org/10.1145/3472883.3487010

[38] M. Nardelli, G. Russo Russo, V. Cardellini, and F. Lo Presti, "A multi-level elasticity framework for distributed data stream processing," in *Euro-Par 2018: Parallel Processing Workshops*, G. Mencagli, D. B. Heras, V. Cardellini, E. Casalicchio, E. Jeannot, F. Wolf, A. Salis, C. Schifanella, R. R. Manumachu, L. Ricci, M. Beccuti, L. Antonelli, J. D. Garcia Sanchez, and S. L. Scott, Eds. Cham: Springer International Publishing, 2019, pp. 53–64.

[39] T. Heinze, L. Aniello, L. Querzoni, and Z. Jerzak, "Cloud-based data stream processing," in *Proceedings of the 8th ACM International Conference on Distributed Event-Based Systems*, ser. DEBS '14. New York, NY, USA: Association for Computing Machinery, 2014, p. 238–245. [Online]. Available: https://doi.org/10.1145/2611286.2611309

[40] Y. Wu and K.-L. Tan, "Chronostream: Elastic stateful stream computation in the cloud," in *2015 IEEE 31st International Conference on Data Engineering*, 2015, pp. 723–734.

[41] S. Rajadurai, J. Bosboom, W.-F. Wong, and S. Amarasinghe, "Gloss: Seamless live reconfiguration and reoptimization of stream programs," *SIGPLAN Not.*, vol. 53, no. 2, p. 98–112, mar 2018. [Online]. Available: https://doi.org/10.1145/3296957.3173170

[42] A. Tiwari, B. Ramprasad, S. H. Mortazavi, M. Gabel, and E. d. Lara, "Reconfigurable streaming for the mobile edge," in *Proceedings of the 20th International Workshop on Mobile Computing Systems and Applications*, ser. HotMobile '19. New York, NY, USA: Association for Computing Machinery, 2019, p. 153–158. [Online]. Available: https://doi.org/10.1145/3301293.3302355