Falcon: Live Reconfiguration for Stateful Stream Processing on the Edge

Pritish Mishra*, Nelson Bore†, Brian Ramprasad*, Myles Thiessen*, Moshe Gabel*, Alexandre da Silva Veith‡
Oana Balmau†, Eyal de Lara*

‡Nokia Bell Labs alexandre.da_silva_veith@nokia-bell-labs.com
†McGill University nelson.bore@mail.mcgill.ca, oana.balmau@mcgill.ca
*University of Toronto {pritish,brianr,mthiessen,mgabel,delara}@cs.toronto.edu

***Abstract*—Stream processing is an attractive paradigm for deploying applications in geo-distributed edge-cloud environments. However, the reverse economics of scale in edge networks and the movement of data sources between edges require the ability to dynamically reconfigure the deployment of stateful applications to adapt to workload variations and user mobility. Unfortunately, existing stream processing engines either do not support the reconfiguration of stateful operators or are ill-suited to edge-cloud environments since they stop application processing during reconfiguration or require costly duplication of application state.**

We propose Falcon, a new stream processing engine. At its core lies a live key migration approach to allow reconfiguration to occur with minimal disruption to processing, even across distant datacenters. Falcon supports the reconfiguration of stateful operators including different windowing approaches and source mobility across different edge regions. It scales gracefully with network latency, the number of datacenters, and the size and number of keys. Our evaluation in geo-distributed edge-cloud deployments shows that Falcon reduces the length of processing interruptions and their impact on latency by 2 to 4 orders of magnitude compared to the existing state-of-the-art frameworks such as Apache Flink, Trisk, and Meces.

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

I. Introduction

Emerging mobile and edge applications, such as traffic monitoring [1], autonomous driving [2], remote health monitoring [3]–[6], and augmented reality [7] generate substantial data from edge and mobile sensors requiring low latency of a few tens of milliseconds. To meet these demands, cloud providers have started to adopt a hierarchical approach that augments their traditional wide-area datacenters with additional (smaller) datacenters that get progresively close to the edge of the network [8], [9]. For instance, Amazon offers options like Local Zones (datacenters serving metropolitan areas), Wavelength (datacenters integrated with 5G networks), and Outposts (on-premises server racks) for deploying computation and data [10]. Similarly, Microsoft recently introduced Azure Stack Edge [11].

Stream processing is a common paradigm for data processing in which applications are structured as a dataflow graph with vertices representing operators and edges representing data streams along which data tuples propagate between operators [12]. Several works such as R-Storm [13], SpanEdge [14], EdgeWise [15], E2DF [16], and DART [17], have proposed stream processing frameworks for edge-cloud environments. These frameworks, however, assume a static placement of operators between edge and cloud datacenters.

The smaller size of edge datacenters, however, leads to higher storage and compute costs (compared to the cloud) that make it inefficient to run operators continuously on the edge. It is therefore desirable for edge stream processing frameworks to be able to react to workload changes by seamlessly reconfiguring the number of operator copies, or *instances*, and their placement. For example, the framework should spawn an additional instance of an operator on an edge datacenter only when the reduction in bandwidth costs outweighs the higher CPU costs of the edge datacenter. Recently, we proposed Shepherd [18], a stream processing framework that supports this by providing seamless reconfiguration with minimal downtime using a network of routers to transfer data tuples between operators. Unfortunately, Shepherd only supports *stateless* operators.

In practice, however, many operators are *stateful*. These operators store contextual information as state to be used for future computations. For instance, recommendation systems store and utilize the user's past activity record for personalized recommendations. Similarly, online model training stores updated model parameters after each training round [19]. Crucially, operations like aggregation and join, which are common in many applications, require processing data in small batches using *windows* that are stored as operator state [20].

Compared to stateless operator reconfiguration, disruption-free reconfiguration of stateful operators is a much harder problem that requires addressing four key challenges: (i) preserving state correctness, since tuples that can modify the state are being processed concurrently to the state migration; (ii) maintaining the semantics of in-order processing and ensuring that tuples are processed exactly once, which further requires fault-tolerance and avoiding duplicate tuple processing; (iii) migrating large open windows and maintaining windowing semantics such that windows are closed in exactly the same manner as they would have been without a reconfiguration; and (iv) addressing *source mobility*, where mobile data sources move from one edge to another requiring *keys* associated with the data source to be migrated to the new edge. Source mobility is not a concern for stateless operators as tuples are independent from each other and can be processed at any operator instance.

Our contributions: We present Falcon, a new stream processing framework for the hierarchical edge-cloud that enables seamless *stateful* operator reconfiguration and supports source mobility.

Falcon uses a novel *live key migration* protocol that combines four techniques: *Dual routing* [21], [22] creates a temporary duplicate data flow that routes tuples to both the source and destination instances. This technique allows Falcon to mask the latency of state transfer to the new operator by continuing to run the application on the original instance. *Marker-based synchronization* [23], [24] injects special *punctuation marker* tuples into the live data stream to demarcate the phases of the protocol. This avoids the need for lengthy message exchanges to coordinate between source and destination which would incur latency spikes due to network delays. Lastly, *emission filters* [18] allow an operator instance to synchronize state by processing buffered tuples without emitting output, thus avoiding the need to later de-duplicate emitted tuples. To tie these together, Falcon adopts the *late binding* approach to tuple routing introduced by our prior work, Shepherd [18], but adapts it to more challenging problems of *stateful* operator reconfiguration and data source mobility.

Falcon also includes a *source mobility protocol* that detects when a source has moved between edges and adapts to this movement by seamlessly migrating the processing and state belonging to this source. This protocol also handles data source "ping-pong" where a source moves back and forth between two edge nodes.

We evaluate Falcon on a geo-distributed edge-cloud network of AWS datacenters. During reconfiguration, Falcon achieves a disruption lasting 10–15 milliseconds, lower than the round-trip time to the root datacenter. In contrast, disruption in existing frameworks ranges between hundreds of milliseconds to several tens of seconds. Moreover, Falcon reduces peak latency during reconfiguration to just 40-45 milliseconds, a decrease of up to 4 orders of magnitude compared to competitors.

In summary, the paper makes the following contributions:

1) A study of the limitations of existing stream processing frameworks when deployed in a hierarchical edge-cloud environment, centered on state management during operator reconfiguration (Section II).
2) The design and implementation of Falcon, the first stream processing framework to support low-latency state migration during operator reconfiguration and source mobility for the hierarchical edge-cloud (Section III). Falcon's source code is available at https://github.com/delara/falcon.
3) An experimental evaluation on geo-distributed edge-cloud datacenters, showing that Falcon reduces processing disruption and peak latency during reconfiguration by 2–4 orders-of-magnitude compared to the state-of-the-art frameworks (Section IV).

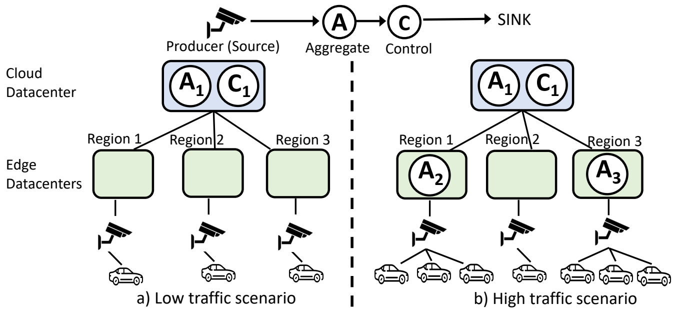


Fig. 1: State reconfiguration for a stream processing application deployed in an edge-cloud environment.

II. Background and Motivation

Stream processing frameworks use a dataflow execution model that represents an application as a directed acyclic graph (DAG) whose vertices represent *operators* and edges represent the *data streams* between operators. Operators can be *sources* that ingest data into the stream, *sinks* that collect the results, or *processing functions* that do transformations.

Users can specify the operators and how data flows between them using an abstraction called a *logical plan* [25]. During deployment, a *physical plan* specifies the number of instances for each logical operator and on which computing node to place each operator instance. A *reconfiguration plan* modifies the physical plan by changing the mapping of operator instances (and their states) across computing nodes, for example, scaling to a different number of replicas, or moving an instance to a different node. In this work, we focus on modifications to the physical plan. Falcon assumes that physical plans are valid implementations of the logical plan.

A. State in Stream Processing Frameworks

Stateful operators (e.g., windows, aggregates, reductions) usually maintain the computation variables (e.g., counters, windows, ML models) in the form of an internal state, while stateless operators (e.g., filter, map) do not have such state. Windows are common stateful operators, where the state stored by each key holds a collection of objects. Closing a window periodically releases this collection, aggregating the tuples. The size of the state can be large for long duration windows, multiple open windows, or large window elements.

Stream processing frameworks allow the splitting of the data stream into multiple sub-streams based on keys. These keys represent unique data attributes that can be specified by the developer. The operator state is partitioned and scoped to these keys. A fundamental assumption of the stateful stream processing model is that a given key can be mapped to a single operator instance. This operator instance is solely responsible for processing tuples belonging to this key and modifying the corresponding state of the key.

B. Reconfiguration of stateful operators

Figure 1 illustrates the benefit of frequent reconfiguration in an example real-time traffic monitoring application. It com-

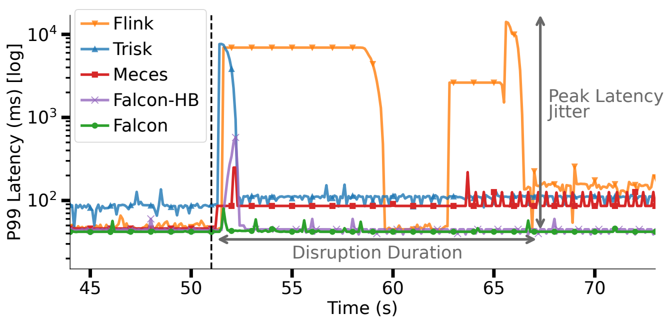


Fig. 2: P99 latency during reconfiguration for a two-tier (edge-cloud) infrastructure deployment. The dashed black line indicates the reconfiguration trigger.

prises an *aggregation operator* (A) which aggregates information from images generated by each motion-activated street camera and stores this information per camera, and a *control operator* (C) that analyzes the resulting events to generate decisions guiding the traffic. In Figure 1a, during low car traffic, running operators solely in the cloud datacenter proves most cost-effective. The lower cost of computing at the cloud datacenter compensates for the bandwidth cost of transmitting raw images. Once traffic in Regions 1 and 3 increases (Figure 1b), it becomes profitable to create additional instances of operator A on edge nodes where vehicle traffic is high as the bandwidth savings from processing camera data locally are higher than the processing costs of running on the pricier edge resources (compared to cloud).

Reconfiguration of stateful operators requires migrating the state stored by the keys (in this example, camera IDs are the keys) to a different operator instance at the new location and switching the data flow belonging to these keys to the new operator instance such that subsequent tuples (including the ones in-flight) are processed at the new location.

A key requirement in the reconfiguration process is maintaining *state correctness* [26]: the state stored by the keys after the reconfiguration must be equivalent to what it would have been had the reconfiguration never happened.

To achieve this, tuples must be processed in the same order as they arrive and no tuple is processed more than once. Reconfiguration must also ensure that windows are closed in exactly the same manner at the new operator instance.

C. Existing stateful reconfiguration solutions fail on the edge

State-of-the-art frameworks implementing reconfiguration on the cloud typically use one of three strategies: *full-restart* (Flink), *partial-pause* (Trisk, Meces), or *hot backups* (Falcon-HB). Figure 2 shows the behavior of frameworks using these approaches and our proposed solution, Falcon, during reconfiguration of the traffic monitoring application (Figure 1). In this experiment, processing of data from Region 1 is moved from A_1 in the cloud to a new A_2 on the edge node.

Apache Flink [27] uses a full-restart approach: it stops the entire application, migrates the affected operators and their state, and then resumes the application. This leads to a substantial stoppage in application processing, as shown by the spike in application latency after the reconfiguration is triggered. Since network latency causes the application to restart on the cloud earlier than the edge datacenter, we see a double spike in Flink's performance.

Trisk [28] implements a partial-pause approach [29], [30] where only the processing of keys affected by migration is paused. While a step forward, we observe it still requires stopping the parts of the application that are affected by the reconfiguration, leading to latency peaks in the order of a few seconds. Meces [31] uses *on-demand state transfer*, an optimization of the partial-pause approach that downloads the state on-demand and processes keys in the background. Meces reduces the latency spike to a few hundred milliseconds but does not improve the disruption duration as fetching of each key's state over high latency network link is quite expensive.

Rhino [32] uses the *hot backups* approach that maintains up-to-date replicas of operator state at other nodes, and switches processing to backups as needed. It still incurs stoppage for migrating the state of keys changed between the last replication and the trigger of reconfiguration. Moreover, maintaining hot backups for each operator is infeasible in large deployments on expensive resources. Since Rhino is not open-source, we implement its hot backup strategy over Falcon. This Falcon-HB version benefits from late-binding design (Section III-C), which explains the low disruption duration.

Conversely, the Falcon *live reconfiguration* approach detailed in the next section, performs very well. It reduces the peak latency by 1 order of magnitude compared to the next best approach, Meces, while also decreasing disruption duration to 10–15 milliseconds.

We also observe that cloud-based frameworks incur higher latency after the reconfiguration (e.g., at 67-second mark for Flink) since they use a single message broker on the cloud to which all sources connect. Post-reconfiguration, tuples generated at the edge must first be sent to the cloud broker, only to be then routed back to the edge. Since cloud-edge links have significantly higher latency than links inside the cloud, the result is higher application latency. In contrast, application latency remains unchanged for Falcon (and Falcon-HB) as it uses a broker network for tuple routing (Section III-C).

In summary, we find that existing techniques for stateful reconfiguration in cloud deployments fall short in edge-cloud environments. Disruption incurred by these techniques is not tolerable for edge applications requiring real-time processing. For instance, a disruption of a few seconds for the traffic monitoring application could lead to a delay in detecting an increase in traffic leading to more congestion due to a missed opportunity to redirect traffic. Similarly, a spike of hundreds of milliseconds could lead to a delay in detecting accidents or missing toll notifications. Moreover, given their poor reconfiguration performance, existing stream processing frameworks cannot efficiently support source mobility, a critical requirement for edge applications.

III. Falcon

Falcon is a stream processing framework that supports efficient reconfiguration in hierarchical edge-cloud environments. Given a reconfiguration plan that switches a running application from one physical plan to another, Falcon implements it while avoiding disruptions to the application processing. In addition, Falcon automatically detects and moves tuple processing to handle moving data sources.

Falcon's key design goals are minimizing processing interruptions during reconfiguration, avoiding message-based coordination between source and destination instances, supporting source mobility, and supporting a wide array of reconfiguration operations. Falcon maintains strict in-order and exactly-once processing guarantees. Our one key assumption is operator determinism, meaning replaying a tuple after restoring the operator state from a checkpoint yields the same state.

The rest of this section is organized as follows. We start with an overview of the system architecture (Section III-A). We next detail how Falcon handles key state and windowing operators (Section III-B), and how it routes tuples (Section III-C). We then introduce the Migrate primitive which supports many reconfiguration operations (Section III-D), and how the live key migration protocol implements this primitive (Section III-E). Section III-F describes how Falcon handles the movement of data sources. Finally, we discuss fault tolerance and implementation (Sections III-G and III-H).

A. System Architecture

Falcon is designed for hierarchical datacenter deployments organised like a tree, where the root node is a cloud datacenter and the leaf nodes are edge datacenters located in close proximity to data sources. Additional datacenter nodes can form the intermediate tiers between the root and the leaves.

Falcon's design relies on the root datacenter (the cloud) having a global view of deployed operator instances and their keyspaces. It hosts an instance of each operator in the application DAG, enabling processing of unhandled tuples from lower hierarchy levels. This root datacenter also manages reconfiguration by redeploying operator instances (replicas of the homologous operators in the cloud) following a reconfiguration plan or source mobility detection. Falcon installs a router on every datacenter in the hierarchy by following late-binding routing design [18]. The routers send incoming tuples to the operator instance on the current node if possible, or send it up to the parent datacenter (Details in Section III-C).

Falcon allows routing rules at both individual key and key-range granularity levels. In contrast to only the key-range granularity supported by other frameworks [28], [31], [33], Falcon's design allows handpicking individual keys for migration to better support source mobility without sacrificing scalability.

Falcon is composed of three subsystems: the *Job Manager*, the *Routers* and the *Workers*.

- The *Job Manager* running at the root of the hierarchy manages applications and monitors deployed operators. In particular, it manages reconfiguration by redeploying the operators following a reconfiguration plan received from the client (Sections III-B, III-D and III-E), and it manages source mobility (Section III-F).
- The *Routers* are deployed at each datacenter node in the edge-cloud hierarchy. Routers manage the flow of tuples across operators and datacenters. Data sources connect to the nearest datacenter and tuples generated by them first arrive at a common *datacenter queue*. The local router either sends tuples to one of the *operator queues* within the datacenter or forwards them to the datacenter queue of the parent datacenter (Section III-C).
- Multiple *Workers* can be deployed at each datacenter node in the edge-cloud hierarchy. Each Worker, managed by the Job Manager, can serve multiple operator instances (e.g., one per core). Each operator instance contains an operator process, a state manager, and an I/O port implemented as a ZeroMQ socket [34]. The operator process reads tuples from its queue, updates the state in memory, and writes output using the I/O port. To maintain state, each node has a replica of a geo-distributed key-value store shared by the Workers, such as SessionStore [35], [36] or Feather [37].

B. Keyed State and Windows

State in Falcon is partitioned by keys, a common approach for stateful operators in stream processing engines [38]. To preserve correct processing semantics, an instance of a stateful operator can only write to the keys it controls.

Falcon allows an application to seamlessly switch between physical plans during its runtime. One assumption in Falcon's design is that tuples flow only up the hierarchy and a valid physical plan must ensure that the operator instance responsible for processing a key is located at a node that is parent to all the producers generating tuples for this key. State in Falcon is managed as follows.

Falcon's Job Manager gives each operator in the logical plan its own keyspace and *mapping function*, which assigns a key to each incoming tuple (based on its contents, the originating source, a bucketing function, etc.). The logical plan specifies how keys are processed by each operator, allowing operations such as splitting the stream to multiple streams by a key ("count different types of objects") and combining them ("sum partial counts"). Each logical plan can then be translated to one of many valid physical plans that specify where the processing of each key is deployed.

Falcon also adds an implicit key $*$ to each keyspace, which serves as a *catch-all* for all tuples whose key was not assigned to an operator instance for processing. In Falcon, $*$ keys for each operator are processed at the root node (the cloud). An important benefit of the catch-all key is being able to process new keys which are unknown to the application (e.g., if a new source is added) at the root node.

Managing Windows. Falcon supports tumbling, sliding, and event-based windows. Tumbling and sliding windows store tuples as part of the operator state, with separate windows for each key. Once a window is closed, its tuples are handed

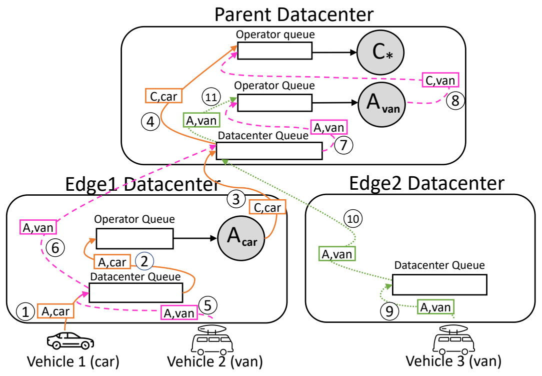


Fig. 3: Example of tuple routing in Falcon. Operator instances A_{car} and A_{van} process tuples of car and van keys, while operator instance C_* processes for all keys.

over to the operator for processing and the window state is cleared (incremental aggregation can be implemented similarly). Event-based windows are closed once either tuples or heartbeat watermarks are received from all sources whose timestamp exceeds the window closing time, or when a configurable timeout is exceeded, similarly to Flink [33]. During the key migration, all open windows for the key are seamlessly migrated as part of the operator state.

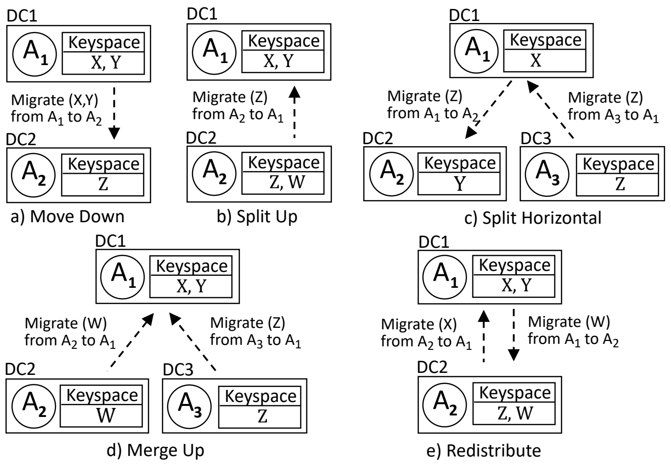
C. Tuple Routing

Falcon generalizes the *late binding* routing design proposed in our prior work on stateless stream processing in hierarchical networks [18], [39]. In late binding, the local router in each datacenter inspects each incoming tuple to see which operator emitted it, and hence which operator will consume it. If there is a local replica of the consuming operator, the tuple is processed locally; otherwise the router forwards it to the parent node, and the process continues. Late binding enables independent operator deployment without knowing the location of upstream or downstream operators.

Falcon proposes *keyed late binding*, which generalizes the existing late binding routing design described in to work with stateful operators. We extend this design by also considering the key of the tuple and the application ID when deciding whether a tuple can be processed locally or should be forwarded to the router of the parent datacenter. This allows us to support keyed state, manage multiple applications and handle source mobility which are crucial for modern stream processing applications [40].

This simple scheme can result in complex routing patterns, which we explore via the following example. Figure 3 demonstrates routing in a two-operator application (source $\rightarrow$ A $\rightarrow$ C) deployed in an edge-cloud environment. Tuples originate from vehicles and follow three possible paths that exemplify an application deployed in such an environment. Each such flow is illustrated in a different color:

- *Orange flow (solid):* ① Vehicle 1 connects to the datacenter queue in edge1 datacenter and emits tuples of type $\boxed{\text{A,car}}$ meant for operator A with key "car". ② The router in the edge1 datacenter routes $\boxed{\text{A,car}}$ tuples to the local operator A_{car}. ③ A_{car} emits tuples $\boxed{\text{C,car}}$ for operator C with key "car". Since no operator C exists locally, the router forwards them to the datacenter queue in the parent datacenter. ④ Router in the parent datacenter routes $\boxed{\text{C,car}}$ to the local operator C, which accepts all keys.
- *Pink flow (dashed):* ⑤ Vehicle 2 connects to the datacenter queue in edge1 datacenter and emits tuples $\boxed{\text{A,van}}$ meant for operator A with key "van". ⑥ Since no operator A processing key "van" exist locally, the router forwards them to the datacenter queue in parent datacenter. ⑦ $\boxed{\text{A,van}}$ tuples are routed to local operator A_{van}. ⑧ A_{van} emits tuples $\boxed{\text{C,van}}$, which are routed to local operator C which accepts all keys.
- *Green flow (dotted):* ⑨ $\boxed{\text{A,van}}$ tuples emitted by vehicle 3 are ⑩ routed to the parent datacenter since no local operator A exists and ⑪ are handled by A_{van} there.



	Move	**Split**	**Merge**	**Redistribute**
Up	MU	MU	MU + MU	MU + MD
Down	MD	MD	MD + MD	
Horizontal	MU followed by MD			

MU: Migrate Up MD: Migrate Down +: simultaneous

Fig. 4: Example reconfiguration plans that can be composed using Falcon's Migrate primitive.

D. The Migrate Primitive

Falcon offers a single simple primitive that supports a wide range of reconfiguration operations:

> **Migrate(K,S,D)**: migrate keyset (*K*) from source instance (*S*) to destination instance (*D*).

The Migrate primitive migrates keys belonging to keyset K from the operator instance currently processing it (source) to another operator instance (destination). If the destination operator instance does not already exist, it is created during the reconfiguration process. If the source instance is left with zero keys after reconfiguration, this instance will be deleted.

Using the Migrate primitive, Falcon supports a wide range of reconfigurations, which are crucial to source mobility: *Moving* an operator instance from one datacenter to another is implemented by migrating all the keys in the source to the destination (Figure 4a). *Splitting* a part of an instance, for example, due to user mobility or to improve performance, is implemented by migrating only the relevant keys (Figure 4b). Splitting a keyspace *horizontally* to a sibling edge is implemented as a migrate up to the parent datacenter followed by a migrate down to the child (Figure 4c). Falcon also allows *merging* of multiple instances into one instance by migrating keys from multiple sources into a single destination and supports key *redistribution* between two instances running on, say cloud and edge, with a combination of migrate up and migrate down. Such reconfiguration operations that require multiple migrations can be run in parallel if there are no keys common between the operations.

E. The Live Key Migration Protocol

The *live key migration* protocol implements the Migrate primitive (Section III-D). Intuitively, the idea is to continue the tuple processing on the source instance while the destination instance transfers the state. Once the transfer completes, these tuples are replayed at the destination to synchronize the state.

To achieve seamless live migration, Falcon uses a novel combination of three techniques. *Dual routing* [21], [22] creates a duplicate data flow that routes tuples to both the source and destination instances. *Marker-based synchronization* [23], [24] injects special *punctuation marker* tuples (reconfig and termination markers) into the datacenter queue to demarcate the phases of the protocol. This avoids the need for lengthy message exchange to coordinate between source and destination that would incur latency spikes due to network delays (Section IV-D). Lastly, *emission filters* [18], allow an operator instance to synchronize state by processing buffered tuples without emitting output, thus avoiding the need to later de-duplicate emitted tuples.

Protocol Steps. Figure 5 shows the steps of the protocol, which we next discuss in detail. Consider the underlying Migrate primitive for a *Split Up* operation (Figure 4b): Migrate K from A_S to A_D, where a set of keys K need to be migrated from source instance A_S executing in a child datacenter to a destination instance A_D executing in a parent datacenter.

Phase 0 (Before Reconfiguration): In the initial deployment, source instance A_S processes tuples belonging to keyset $S \cup K$. As shown in Figure 5a, tuples 1 and 2 belonging to keyset S and K respectively arriving at the datacenter queue of DC2 are forwarded to the operator queue of A_S. Similarly, tuple 3 belonging to keyset D is forwarded to the parent datacenter DC1 to be processed by destination instance A_D. Also, note that each operator instance has a keyspace and processes a tuple only if it belongs to a key that is present in its keyspace (we will see its importance in Phase 1).

Phase 1 (Create Dual Route): Once the reconfiguration is triggered, the Router begins *dual routing* where tuples belonging to keyset K are sent to both the source and destination instances. To achieve this, the system adds K to the routing rule of the destination instance. As shown in Figure 5b, routing of A_D is now modified from D to $D \cup K$, while the routing rule of A_S remains as $S \cup K$. Hence, tuples 5 and 7 of keyset K arriving after the creation of dual-route are routed to both A_S and A_D. Note that at this stage, these tuples are ignored at the destination instance since its keyspace is still D rather than $D \cup K$. This prevents the dual-routed tuples from being processed twice.

Phase 2 (Inject Reconfig Markers): Immediately after creating the dual route, Falcon injects two reconfig markers, R_S and R_D, to demarcate the start of the state transfer phase (Phase 3). To avoid pausing the operator processing queue, Falcon injects these markers into the datacenter queue of the child datacenter (DC2 in Figure 5b). R_S will then be routed to the source instance and R_D to the destination instance.

Phase 3 (State Transfer): Upon processing R_S, the source instance A_S creates an on-demand checkpoint by copying the current state of keyset K from memory to the local persistent storage. This on-demand checkpoint also copies all the K tuples that had arrived at A_S between the last periodic checkpoint and the reconfig marker, R_S to allow for tuple replay later at the destination.

Upon processing R_D, the destination instance A_D pauses output for tuples belonging to K, adds K to its keyspace and triggers restore, i.e., download of K's state from the source instance. During the restore, K tuples are still being processed in parallel due to dual-routing. For example, tuple 9 belonging to K that arrives during restore is processed by A_S and is buffered by A_D. Once restore is complete, A_D starts processing the buffered tuples including the ones downloaded from the source instance during restore. However, this processing is done solely to synchronise the state and to avoid processing tuples twice, we enable an emit filter on A_D, which prevents output when processing buffered tuples.

Phase 4 (Inject Termination Markers): Since the destination instance needs some time to clear the backlog of buffered tuples, we continue the dual routing even after restore. Once the backlog falls below a threshold (which depends on the tuple arrival rate), the router injects two termination markers, T_S and T_D to trigger the end of the reconfiguration process.

Phase 5 (Killing Dual Route): Immediately after injecting the termination markers, Falcon *kills* the dual route by deleting K from the routing rule of the source instance A_S. This means tuples belonging to K will now be routed only to the destination instance A_D. In our example, the routing of A_S is now modified from $S \cup K$ to S. Since the routing rule of A_D remains as $D \cup K$, tuples of K arriving after killing of dual-route will be routed only to A_D.

Phase 6 (Terminate Reconfiguration): Upon processing the termination marker T_S, the source instance removes K from its keyspace. Hence, tuples belonging to K that arrive between Phases 4 and 5 (e.g., tuple 11) will only be processed by the destination instance and not by the source instance. On processing the termination marker T_D, the destination instance disables the emit filter and resumes emitting output

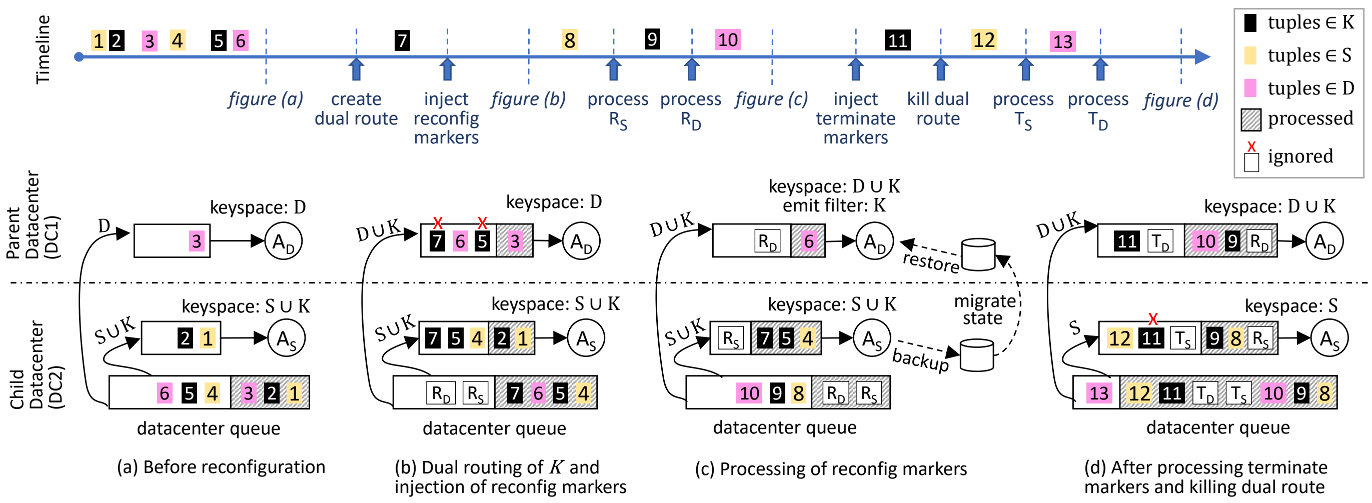


Fig. 5: Reconfiguration steps when migrating the key K from operator instance A_S in a child datacenter to instance A_D in a parent datacenter. Annotated timeline of arriving tuples indicates the reconfiguration steps and each figure presents a snapshot of the system at a particular point of the timeline. See Section III-E for more details.

for tuples belonging to K. Thus, tuples arriving after Phase 4 are processed solely by A_D and their results are emitted.

Before emitting the results of tuples arriving after the termination marker T_D (e.g., tuple 11), the destination instance A_D waits for an acknowledgment from the source A_S confirming that it has processed its own termination marker T_S. This prevents a corner case where A_D would emit output of new tuples (e.g. tuple 11) out of order, before a slower A_S has processed T_S and any preceding tuples (e.g., tuple 9).

Our protocol is generic: it supports all reconfiguration actions (Figure 4) and is exactly the same for both upward and downward directions. The dual routing design minimizes delays in the processing of tuples belonging to K. The sole interruption in processing occurs while awaiting acknowledgment from the source to the destination instance, approximately half the round trip network latency. This is necessary to maintain the strict guarantees of in-order tuple processing.

When selecting an operator instance to migrate keys, it's crucial to choose one with sufficient spare capacity for processing these keys. This ensures the instance can handle tuple replay and catch up to the source instance. If no suitable instance is available, spinning up a new one is straightforward since states are not shared and the protocol minimizes disruptions. Keys migrated to a destination operator that cannot handle the load would likely incur a large and increasing queue backlog, potentially overflowing it. However, this would happen with any system, even without live migration. Overloading an instance is a failure of the migration *policy* rather than the migration *mechanism*, the focus of our work.

State correctness. Falcon guarantees correctness by preserving: (1) in-order processing of tuples, and (2) exactly-once processing of tuples. Our migration protocol ensures these properties as follows:

- During reconfiguration (Phases 1–5), tuples are routed to both source and destination instances. These tuples are processed and their results are emitted at the source instance. The destination instance only processes these tuples and doesn't emit the results. By not emitting output tuples at the destination, tuples emitted by the migrated key during Phases 1–5 are only seen once by the downstream operators. This prevents duplicate tuple processing. In addition, our fault-tolerance mechanism prevents the dropping of any tuples. Thus, exactly-once processing is maintained.
- For in-order processing, Falcon ensures tuples arriving post-reconfiguration (Phase 6) are processed at the destination instance only after those that arrived during reconfiguration (being processed at the source instance). This is done in Phase 6 where the destination instance waits for the source instance to process and emit the terminate marker.
- In addition, Falcon injects the two sets of punctuation markers that indicate start and end of reconfiguration in a single queue at the downstream data center and then forwards these markers to the upstreams. This ensures that both source and destination instances have the same perception of tuples arriving before and after a marker.

In our experiments (Section IV-B), we evaluated the correctness for operator migration using a deterministic dataset and ran experiments with and without reconfiguration. We verified that the state was identical in both experiments.

F. The Source Mobility Protocol

The second core mechanism of Falcon is its source mobility protocol, accounting for the common scenario where data sources move across edge nodes. This protocol involves two steps: switching the network connection between edge routers, and reconfiguring the stream processing application.

Router switchover. Falcon registers all routers deployed on the edge nodes, including their IP addresses, using ETSI MEC's Edge Platform Application Enablement [41]. A data source

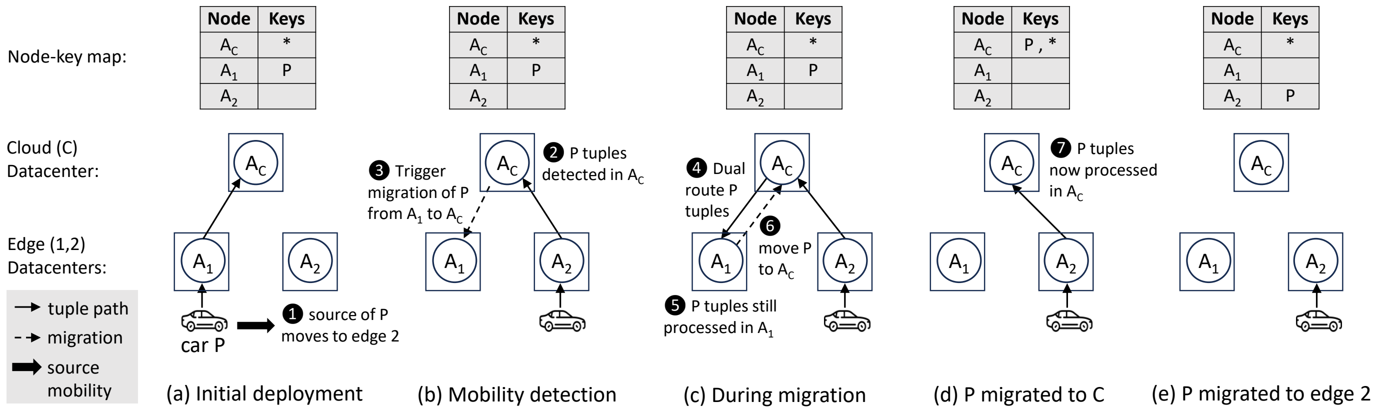


Fig. 6: Mobility of car emitting key P moving from edge 1 to edge 2. For simplicity, we only show one operator (A), replicated on cloud C, edge 1, and edge 2. Solid arrows indicate flow of P tuples. See Section III-F for more details.

(e.g., a car, or other mobile device) uses the device application interface to retrieve the Falcon router IP address to connect. When the data source moves from edge A to edge B, MEC sends a notification to the data source (using device application assisted user context transfer in the MEC standard [42]). The notification contains communication information, such as the IP address of the edge B router. The data source then closes its connection to the edge A router and opens a new connection to the edge B router for service continuity.

Application reconfiguration. Consider a mobile source that produces tuples with key P coming into an operator A, as shown in Figure 6a. These tuples are processed by the operator instance A_1 located on edge 1. Falcon maintains a global node-key map at the root node (the cloud) that stores all operator instances and their assigned keys (top of Figure 6). When the data source moves from edge 1 to edge 2 ❶, A_C detects that the source has moved since it has received P tuple which is already mapped to a different operator instance, A_1 ❷ and it triggers a migration of the key P from A_1 to A_C ❸. For scaling to an N-level topology, this global mapping on the cloud can be extended to a hierarchical mapping where parents are only aware of the key spaces of their direct children.

To achieve seamless reconfiguration, Falcon must continue processing of P tuples during migration. This is challenging since P tuples now arrive at edge 2, yet their processing is done at A_1 on edge 1. To address this, Falcon uses the dual routing mechanism (Section III-E, Phase 2) to forward P tuples arriving at C to A_1 ❹ while also collecting them at A_C. This is same as the usual live migration protocol, except that the tuples are routed down the hierarchy rather than up as usual. A_1 continues to process incoming P tuples, while A_C replays them ❺, while the state of P is migrated from A_1 to A_C ❻. Once the migration is completed, P tuples arriving at the cloud node are processed by A_C ❼. To avoid unnecessary migrations during continued movement, Falcon migrates the processing of P from A_C to A_2 (Figure 6e) only if the data source remains connected to edge 2 for a configurable minimum duration (default: 5 seconds). Note that this design can easily be extended if the edge nodes - A_1 and A_2 have a direct point-to-point connection.

Data source "ping-pong". One interesting corner case is when a data source moves back and forth between the same two edge nodes. Continuing our example, while Falcon is migrating the key P from A_C to A_1 in Figure 6c, the source could move back to edge 1. If the Job Manager detects this during the migration from A_1 to A_C, Falcon allows the migration to continue since A_C can process tuples from all edges (via the catch-all mechanism), and the source could continue moving between the edge nodes. On the other hand, if the Job Manager detects the move back to edge 1 during the migration from A_C to A_2, Falcon terminates the migration. The ping-pong scenario can lead to another challenging corner case for in-order processing when tuples generated by the source before disconnecting from edge 2 arrive at A_C *after* the first tuple produced by the source on reconnecting to edge 1. To ensure tuples are processed in order, Falcon buffers incoming P tuples when detecting a ping-pong. After a short configurable duration (by default 100 ms), tuples are re-ordered and processed based on their timestamps.

G. Fault Tolerance

For fault tolerance, Falcon relies on a set of standard assumptions: (1) operators are deterministic, (2) machine and network failures are not permanent, and (3) the underlying message broker provides exactly-once processing, is fault-tolerant, and supports tuple replay and acknowledgment. The broker at the transport layer handles network packet loss; other failures are handled by tuple replay. Note that failures during reconfiguration can incur stoppage in application processing.

During steady state, all operators in a single datacenter can be considered a single application with one broker. We use a combination of asynchronous checkpointing, deterministic tuple replay, and tuple acknowledgment – the same strategy used by Flink [33] and other frameworks [28], [31], [32], [43].

During reconfiguration, there are two failure points: message brokers and operator instances. Both these failures could occur in Phases 1 and 5 of the live key migration protocol (Section III-E). Phases 2 and 4 depend only on the broker and Phases 3 and 6 can only be affected by operator instance failure. To handle message broker failures during reconfiguration, Falcon waits for recovery and retries failed operations (via dual routing, and punctuation marker injection). If an operator instance fails during reconfiguration, Falcon uses replay: since coordination is based on punctuation markers and the broker is fault tolerant, instances that failed after marker processing are restarted with tuples and markers re-delivered.

H. Implementation

Falcon is implemented in Java (approx. 50K LOC). Application operators are implemented as Java applications running inside Docker containers. Apache ActiveMQ Artemis [44] serves as the message broker for our routing system, with Falcon's routing rules implemented as Artemis filters within message queues. The dual-routing technique is implemented by adding diverts within the queues for routing tuples to two locations simultaneously, using custom filter expressions [45] to identify tuples belonging to specific keys. To minimize latency in intra-datacenter communication, operators within the same datacenter utilize ZeroMQ sockets [34], with brokers solely employed for inter-datacenter communication. For state storage, we utilize SessionStore [35], [36], an open-sourced geo-distributed key-value store built on top of Cassandra [46].

IV. Experimental Evaluation

In this section, we set out to answer the following questions:

1) How does the reconfiguration performance of our live key migration approach compare to the full-restart, partial-pause and hot backup approaches? (Section IV-B)
2) What is the impact of source mobility? (Section IV-C)
3) How do network latency and topology size affect Falcon's reconfiguration performance? (Section IV-D)
4) How do the application characteristics affect Falcon's reconfiguration performance? (Section IV-E)

We define three metrics of reconfiguration performance. **Disruption duration** measures how long processing is disrupted due to a reconfiguration event. We detect disruption when the end-to-end tuple processing latency is greater or equal to the mean latency during steady state, plus five times the standard deviation. **Peak latency jitter** is the impact of the interruption on application performance, defined as the difference between peak and mean end-to-end tuple processing latency. Lastly, **reconfiguration duration** is defined as the time between the start of the reconfiguration event and the emission of output tuples at the destination instance. Depending on the reconfiguration mechanism, this duration could include migration of application state and in-flight followed by a replay of these tuples at the destination instance.

A. Experimental Setup

We evaluate Falcon on an emulated hierarchical edge-cloud deployment made of two AWS datacenters: one in North California acting as the root (i.e., cloud) and one in Montreal acting as the child node (edge) near the data sources. The round-trip latency between the edge datacenter and the cloud datacenter is measured to be 80 milliseconds for all experiments except the latency experiment in Figure 10. We use m5.2xlarge EC2 instances running on a 3.1 GHz Intel Xeon Platinum 8175M with 8 threads and 32 GB RAM. The average intra-datacenter bandwidth was 2.5 Gbps, while inter-datacenter bandwidth was 1 Gbps.

Baselines. Shepherd is the only existing solution that supports reconfiguration on the edge. However, since it does not support stateful reconfiguration and there are no other edge frameworks that do this, we instead compare Falcon to existing cloud-based frameworks that support stateful reconfiguration.

We compare Falcon to baselines representing full-restart (Flink [47]), partial-pause (Trisk [48]), on-demand state transfer (Meces [49]) and hot backups (Falcon-HB). Falcon-HB's hot backups mechanism is inspired by Rhino [32] since its source code is not available and Rhino does not natively support hierarchical edge-cloud deployment. Our Falcon-HB version uses the late-binding routing design, avoids global coordination and state alignment during reconfiguration, and can consume tuples from a co-located message broker instead of downloading them from the cloud broker. The checkpointing interval in Falcon-HB is set to 500 milliseconds to minimize reconfiguration disruption. Finally, we evaluate against a Falcon version that allows out-of-order tuple processing (Falcon-OOP), to illustrate the added cost of in-order processing.

Applications. As there is no standardized benchmark for edge-based stream processing applications at the time of writing, we develop the traffic monitoring application TM (utilized as a recurring example in this paper). Additionally, we modify four workloads from Nexmark [50], [51] and two workloads from Linear road [52], [53] to suit an edge-cloud hierarchy. These selected workloads are typically used in evaluating stream processing engines [28], [31], [32], [43]. Figure 7 shows the logical plans. These applications represent the most popular kinds of state: key-value state (TM), count-based (NQ6), and event-based tumbling window (NQ7), and, count-based (LR-AN) and event-based sliding window (NQ8, LR-TN) [40].

(1) Traffic Monitoring (TM). This stateful application monitors the vehicles on a street to detect the ones violating the speed limit, and allows fine-grained control of experimental parameters. Vehicles generate tuples containing their current speed, and the stateful MOV AVG operator computes a running average speed for each vehicle. MOV AVG uses a key-value state where the key is the vehicle ID and the value is its traffic statistics (current average speed and number of observations). The next stateless TRIG operator triggers an alert if the average speed of a vehicle exceeds the speed limit. The lightweight nature of this application ensures that any impact of reconfiguration on application performance is clearly visible. The data production rate is 1500 tuples per second, with 10 keys and 32 bytes of state per key.

(2) Nexmark Benchmark (NQ5-8). Query 5 (NQ5) uses an event-based sliding window (WIN COUNT) of size 1-minute (and 1-second slide) to count the number of bids per item from

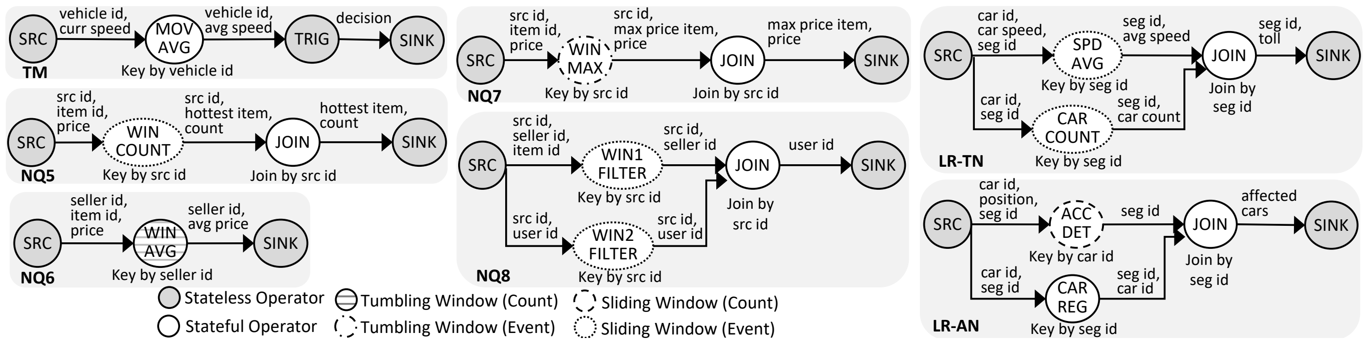


Fig. 7: Logical plan of TM, Nexmark Benchmark (NQ5-8), and Linear Road Benchmark (LR-TN, LR-AN) applications.

a stream of bids generated by a data source and generates the hottest item with maximum bids along with the bid count. NQ7 uses a similar logic to calculate the maximum priced item by instead using an event-based tumbling window in the WIN MAX (Windowing Max) operator. The JOIN operator aggregates the data for the entire stream.

Query 6 (NQ6) calculates the average selling price of the last 10 items sold by a seller using a count-based tumbling window (WIN AVG) from a stream of auction bids. Query 8 (NQ8) uses two filters in the event-based sliding windows (WIN1, WIN2 FILTER) to respectively find the users who joined the system in the last hour and who submitted a bid in that period. The JOIN operator determines users common in the two filtered results. This query uses long-running windows (1-hour size, 1-second slide) emitting results every second.

For all queries, we configure the Nexmark Data Generator to use a skewed data distribution with a ratio of hot to cold items of 100. Each data source is placed at the edge node producing 1500 tuples/second. There are 10 keys and the state size per key is 1.3KB, 1.6KB, 1KB and 82 KB for NQ5, NQ6, NQ7 and NQ8 respectively.

(3) Linear Road Benchmark (LR-TN & LR-AN). The Toll Notification (LR-TN) query computes tolls for each segment of an expressway. The SPD AVG (Speed Average) operator uses an event-based sliding window (1-min size and 1-second slide) to report the latest average speed of all cars on a segment. Likewise, the CAR COUNT operator calculates the number of cars on the segment. Finally, the JOIN operator uses both these values to calculate the toll for a segment (See [52] for the specific formula).

Accident Notification (LR-AN) query determines which cars are affected when an accident occurs on a segment. ACC DET (Accident Detect) operator uses a count-based sliding window (size=10, slide=1) to report an accident if the last 10 positions of a car are same. CAR REG (Car Register) operator maintains a mapping of each segment with the cars currently on it and returns all cars on the segment. If an accident is reported on a segment, JOIN operator returns the list of all cars on the segment.

The dataset contains 101 segments and 124,000 cars with the state size per key (where key is segment ID) of 2.8-3.1 KB for LR-TN and 50-70 KB for LR-AN. Reconfiguration

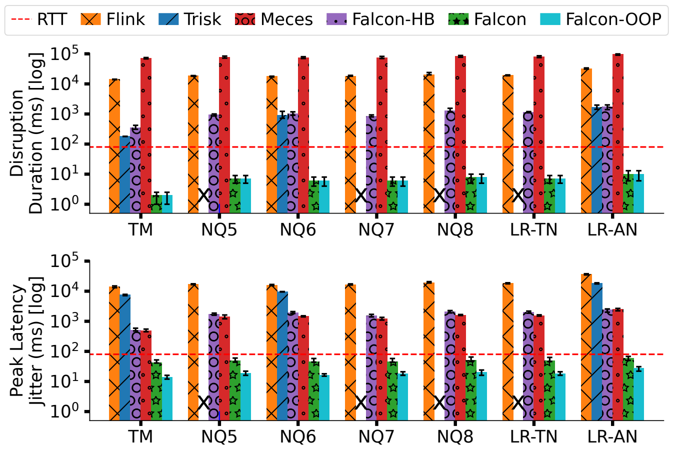


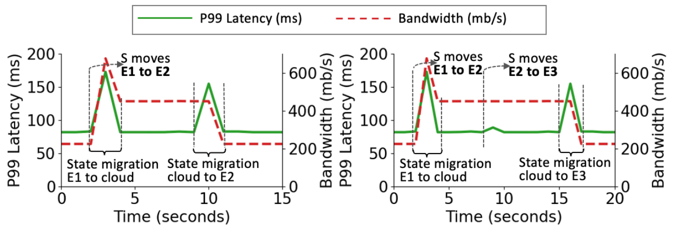
Fig. 8: Reconfiguration stoppage. Dashed red line indicates round-trip inter-datacenter latency. Trisk does not support reconfiguration for event-based windows (marked X).

benefits both queries: when the number of cars on a segment increases, the processing of the stateful operator can be moved to the edge node to reduce the amount of data transferred to the cloud. Conversely, moving the processing back to the cloud during low traffic avoids expensive edge resources.

B. Reconfiguration Performance

We evaluate the impact of handling one reconfiguration. Initially, all operators are deployed on the cloud node. After 60 seconds, each experiment triggers a reconfiguration spawning a new instance of the stateful operator (MOV AVG for TM; WIN COUNT, WIN AVG, WIN MAX, WIN FILTERs for NQ5-8; SPD AVG, CAR COUNT, CAR REG for LR queries) on the edge node and migrating processing of 50% of the keys to this new instance. Note that state migration in Meces and Trisk was designed for dynamic scaling rather than for improving data processing locality. Unlike Falcon, they do not allow the user to choose which subset of the keys to migrate. Hence, we limit experiments to scenarios with uniformly distributed tuples across data sources, favoring baseline frameworks.

Figure 8 shows the disruption duration and peak latency jitter (99^{th} percentile), averaged over 5 runs. Falcon achieves 1–4 orders of magnitude reductions in both disruption duration



a) Source S moves from edge E1 to E2. b) Source S moves from E1, to E2, to E3.

Fig. 9: Falcon reconfiguration performance when the source moves between 2 edges (a) and 3 edges (b). We show up to 3 edges here for clarity of different phases. Falcon shows a similar performance even for 32 edges where a source moves from edge 1 to edge 32.

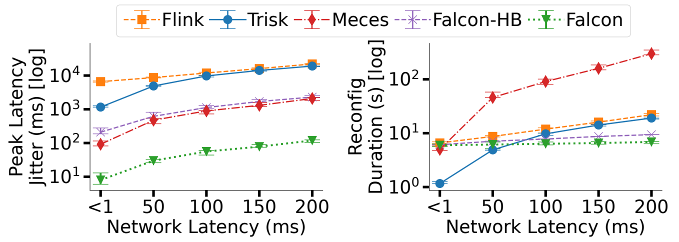


Fig. 10: Effect of round-trip latency on peak latency jitter (left) and reconfiguration duration (right).

and peak latency jitter across the board. The peak latency jitter of Flink and Trisk is in the range of tens of seconds, while Falcon-HB and Meces bring it down to hundreds of milliseconds. However, note that the hot backup approach has the disadvantage of linearly increasing bandwidth with the number of edges and state size. In contrast, Falcon achieves the lowest jitter of ∼45 milliseconds. Falcon's live key migration mechanism continues tuple processing in parallel to migration. Disruption incurred by Falcon is lower than even the round-trip network latency (80 ms, on average) because coordination is done through markers. The only message exchange between the source and destination incurs a single one-way message delay to guarantee in-order processing (Sec. III-E, phase 6), which is often overlapped by processing at the destination. By omitting this message, Falcon-OOP achieves a further improvement (by 14-20 ms) in peak latency jitter with no change in disruption duration, showing that the cost of guaranteeing in-order processing for Falcon is low.

A breakdown of Falcon's performance across its six phases is as follows: Phases 1, 2, 4, and 5 each take 2–3 ms, while Phase 6 takes 14–16 ms for post-processing. Phase 3 takes 600 ms to 1.5 s; however, its execution is overlapped by the dual-routing mechanism, ensuring it does not impact the overall processing latency of the application.

C. Support for Data Source Mobility

To evaluate the mobility of data sources, we set up a two-tier edge-cloud deployment comprised of one root (i.e., cloud) and three child data centers (edges). We deploy the LR-AN application using a single car as the data source and simulate the mobility using the MEC Sandbox [41] as a high-velocity vehicle that connects to a new edge node every few seconds. The sandbox also sends MEC-based notifications when the source moves from one edge node to another. When the car moves from one edge to another, this triggers a reconfiguration of Accident Detect (ACC DET) operator to migrate the processing of the key corresponding to the car accordingly. We do not include Flink, Trisk and Meces in this experiment, as they do not support the mobility of sources.

Figure 9 depicts the 99th percentile latency (solid green line) and total bandwidth utilization (dashed red line) over time, as the data source (i.e., the car) shifts from (a) edge 1 to edge 2 and (b) from edge 1 to edge 3, via edge 2. Latency is measured at one-second intervals. Throughout the majority of the source movement, the tail latency stays close to the 80ms round trip time, except for three peaks. First, the peaks at the 3-second mark (in both Figures 9a and 9b) are caused by the state migration from edge 1 to cloud, causing an approximate 80ms latency increase. Mobility detection latency is negligible, typically a few milliseconds. Second, the peaks at the 12-second mark (Figure 9a) and the 18-second mark (Figure 9b) are the cost of migrating state from the cloud to the edge, where the data source is now located. Recall that Falcon avoids too frequent migrations for fast-moving sources by waiting a configurable time (5 seconds here) before moving the processing from cloud to edge (Section III-F). The third, smaller, peak that can be seen at 9 seconds in Figure 9b is the cost of the data source re-establishing a connection with edge 2, which results in a slight delay in tuple emission.

Bandwidth usage momentarily peaks during state migration (at 3 seconds) due to dual routing: when moving processing from edge 1 to the cloud, tuples arriving from edge 2 to the cloud are temporarily forwarded to edge 1 (Sec. III-F). Processing tuples in the cloud during the transition also incurs higher bandwidth usage temporarily: once the migration from the cloud to the edge is complete, the bandwidth usage drops to its normal value.

In all cases, Falcon achieves a disruption duration of 10ms and peak latency jitter of 50–80ms, which is the same or less than the round-trip link latency of 80ms. Both are orders of magnitude better than what Flink, Trisk, Meces, and Falcon-HB would achieve (based on results from Section IV-B).

We observe similar reconfiguration performance in the ping-pong scenario where the source moves back and forth between two edges. We omit the results due to space constraints.

D. Impact of Network and Topology Size

We evaluate the impact of network latency on reconfiguration performance by using Linux Traffic Control [54] to add latency between parent and child nodes. In the TM application, 50% of keys at the MOV AVG operator are migrated to a new instance on the child node during reconfiguration. As round-trip network latency increases, peak latency jitter rises for all frameworks including Falcon (Figure 10, left). Flink and Trisk encounter delays in state migration due to increased

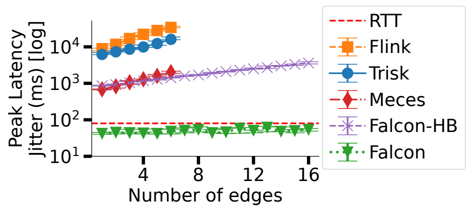


Fig. 11: Impact of the number of edges on reconfiguration. Reconfiguration triggered on Flink, Trisk and Meces fails for 7 or more edges due to timeouts.

bandwidth-delay product, while Falcon-HB incurs this delay in migrating tuples since the last checkpoint. Meces's on-demand fetch approach is a poor match for high-latency edge links, as the increased round-trip per request starts to accumulate [55]. Falcon's peak latency jitter is around half the network round trip time, the time taken by the source to acknowledge reconfiguration completion to the destination (Section III-E, Phase 6)

The reconfiguration duration of Flink, Trisk and Meces increases with an increase in latency while it remains nearly constant for Falcon and Falcon-HB (Figure 10, right). The dominant factor in reconfiguration duration for Flink, Trisk and Meces is the time taken to transfer the backlog of in-flight tuples, which increases with latency because of the increased pause in tuple processing. For Meces, the duration is caused by the delay in fetching the state on demand. In Falcon (and Falcon-HB), the dominant factor is the nearly constant time to start the new operator instance.

We next evaluate how the number of edges existing in a deployment prior to the reconfiguration can affect its performance. Initially, we have a deployment of N edges where an instance of MOV AVG in the TM application is deployed on the cloud and $N-1$ edge nodes. During reconfiguration, we create an instance of this operator on the N^{th} edge and migrate the processing of a subset of the keys to this instance.

Figure 11 shows that Falcon readily scales to many edges: its reconfiguration performance is unaffected with increasing edges. Conversely, as edges are added, Flink, Trisk, and Meces experience exponential increase in peak latency jitter due to their early-binding design, where the socket connections between all upstream and downstream operators are coordinated globally and re-established after reconfiguration. Falcon-HB shows a linear increase as adding edges amplifies the number of tuples requiring replay. Falcon avoids the latency increase since there is no direct connection between upstream and downstream operators and there is no effect of tuple replay on disruption. Since this design only involves the source and destination instances, peak latency jitter stays constant regardless of the physical plan's operator count or edge additions.

E. Impact of Application Characteristics

We next explore the effects of state size per key, number of keys, and window size on reconfiguration performance.

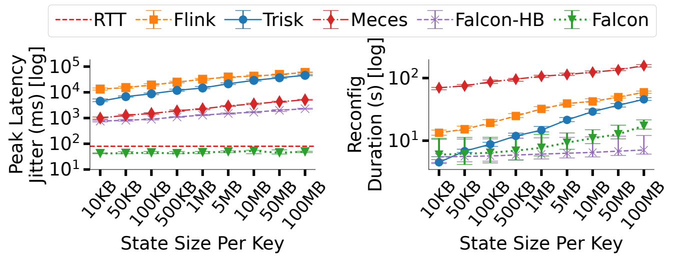


Fig. 12: Impact of state size on peak latency jitter (left) and reconfiguration duration (right).

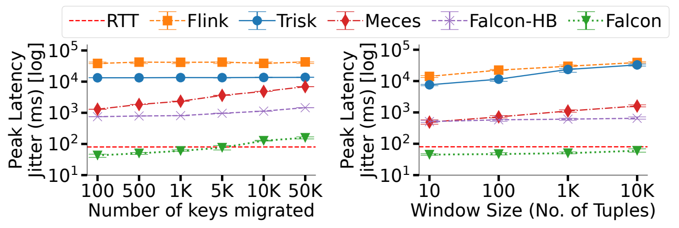


Fig. 13: Impact of the number of keys (left) and window size (right) on reconfiguration performance.

In Figure 12 (left), we vary the state size per key in the TM application from 10KB to 100MB. The increase in state size per key leads to an increase in peak latency jitter for all frameworks except Falcon. For Flink, Trisk and Meces, the disruption due to the state download over the network increases with the state size. Falcon-HB achieves a lower increase by avoiding this download. State download has no impact on Falcon as it occurs in parallel to tuple processing.

Figure 12 (right) shows a similar pattern. The increase in state size per key correlates with increased reconfiguration duration for Flink, Trisk and Meces. This occurs as the download duration of the application state and the transfer duration of the in-flight tuples increase with the increase in state size. In contrast, Falcon has to download the application state and fewer tuples during reconfiguration. Falcon-HB achieves a lower increase rate by avoiding the state download and only downloading tuples that arrived since the last checkpoint.

In Figure 13 (left), we vary the number of keys migrated during the reconfiguration of TM, using the default per-key state size of 32 bytes. As expected, peak latency jitter for Flink, Trisk and Falcon-HB remains nearly constant due to the modest increase in the migrated state size. On-demand state-transfer approach of Meces scales poorly because increasing the number of keys results in increased stalling and queuing overhead. Falcon's peak latency jitter is still orders of magnitude lower compared to the other frameworks.

Finally, in Figure 13 (right), we evaluate the impact of window size on reconfiguration performance, by varying the count-based window size in the NQ6 application. Larger windows mean larger states (ignoring incremental averaging). Hence, increasing the window size leads to an increase in the peak latency jitter for all the frameworks except Falcon-HB. Even for a window of 10,000 tuples, Falcon's peak latency

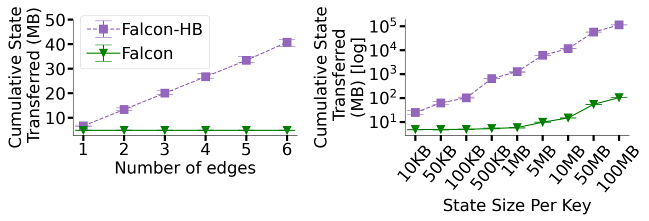


Fig. 14: Impact of the number of edges (left, linear scale) and state size (right, log scale) on the transferred data.

jitter is lower than even the network latency.

Cost of hot backups. Network transfers incur costs, especially in edge deployments. To evaluate the overhead of hot backups, we measured the total data transferred between the cloud and edge datacenters during experiments conducted in Section IV-B. Falcon-HB's checkpointing interval is set as 500 ms, as this yields the best reconfiguration performance. The size of state transferred by Falcon-HB increases linearly with the number of edges (Figure 14, left) and the state size per key (Figure 14, right). The former is due to the need to replicate state to all edges for a possible future reconfiguration, while the latter is because of the increase in the amount of state replicated for every checkpoint. This suggests hot backup is ill-suited for reconfiguration in hierarchical edge networks, which are seldom limited to a handful of nodes. In contrast, Falcon transfers the state only once during the reconfiguration, so its transfer size remains constant regardless of the number of edges, and increases slowly as the state size per key grows.

V. Related Work

Reconfiguration using full-restart in the cloud. Flink [33], Spark [56], Storm [57], and Stella [58] use a full-restart reconfiguration approach interrupting the application to perform an on-demand state checkpoint, move the checkpoint to a new host, and restore the checkpoint. This reconfiguration approach delivers state correctness but falls short of meeting stringent requirements of latency peaks and system disruptions for applications that need frequent runtime modifications. In contrast, Falcon targets edge-cloud deployments with mobile sources, where frequent and efficient reconfiguration is needed.

Reconfiguration using partial-pause in the cloud. Partial-pause reconfiguration [24], [28], [30], [59], [60] reduces the system disruption time in redistributing operators by pausing only the processing of the affected operators. For instance, Megaphone [43] splits the state into small parts and moves the state in increments. Meces [31] fetches the state for a key only when it encounters a tuple belonging to this key. In contrast, Rhino [32], Chronostream [21], and Gloss [22] implement replicated dataflows. The application tuples and operators (or checkpoints) are replicated to several hosts where an operator could be assigned. This solution offers robustness but is expensive, as additional resources are required for the replicated state. Moreover, the early-binding approach used in these systems leads to inefficient triangular routing for edge-cloud deployment, becoming a bottleneck during rescaling.

Stream processing for edge deployments. DART [17] uses a peer-to-peer overlay network to distribute operators on to edge datacenters. SpanEdge [14] provides a user-friendly programming environment for operator placement. Both systems take a full-restart approach for operator scaling and reconfiguration. Our prior work, Shepherd [18] supports reconfiguration for *stateless* operators with low disruption, leveraging late binding routing to avoid global coordination. StreamBucket [61] is another solution designed for hierarchical edge-cloud deployment that focuses on improving the scalability of dual-routing networks. However, both Shepherd and StreamBucket lack support for stateful application reconfiguration. Our system addresses the more challenging scenario of reconfiguration in *stateful* applications and also supports for source mobility.

Orchestration for edge deployments. Oakestra [62] was recently proposed with the goal of improving the control plane in hierarchical orchestration for edge-cloud infrastructure. Similarly, Starlight [55] accelerates the provisioning of containers on edge devices. These solutions complement our work, and Falcon can leverage them to streamline operator deployment.

Stateful Operator Migration vs. Live VM Migration. Live VM migration bears some similarities to stateful operator migration. The mutation of page tables and file modifications resemble how tuple processing affects state during migration, and techniques like checkpointing are used in both domains. However, unlike operator migration, the entire VM must be transferred to the new host [63]. This results in longer downtimes, prompting service providers to focus on adapting specific workloads [64]. Conversely, stateful operator migration works at a finer granularity as splitting the state into keys and migrating these keys presents a more nuanced challenge.

VI. Conclusion

We presented Falcon, a stream processing framework for live reconfiguration of stateful operators in a hierarchical edge-cloud deployment. Falcon's live key migration approach allows seamlessly moving operator state between replicas; its source mobility protocol supports data sources that move between different edge nodes of the network. Falcon reduces reconfiguration latency from tens of seconds to a few milliseconds on geo-distributed edge-cloud infrastructure and supports a wide variety of application states and reconfiguration operations. Future work will focus on developing adaptive placement strategies to predict reconfiguration pro-actively and improving the scalability of our routing mechanism.

Acknowledgements

This work was supported in part by funding from the Innovation for Defence Excellence and Security (IDEaS) program from the Department of National Defence (DND), Huawei Technologies Canada Co., Ltd and Natural Sciences and Engineering Research Council of Canada (NSERC), Collaborative Research and Development Grants, 543885-2019.

References

[1] S. Hua, M. Kapoor, and D. C. Anastasiu, "Vehicle Tracking and Speed Estimation from Traffic Videos," in *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition Workshops*, 2018.

[2] I. Lujic, V. D. Maio, K. Pollhammer, I. Bodrozic, J. Lasic, and I. Brandic, "Increasing Traffic Safety with Real-Time Edge Analytics and 5G," in *Proceedings of the International Workshop on Edge Systems, Analytics and Networking*, 2021.

[3] A. Tiwari, S. Liaqat, D. Liaqat, M. Gabel, E. de Lara, and T. H. Falk, "Remote copd severity and exacerbation detection using heart rate and activity data measured from a wearable device," in *2021 43rd Annual International Conference of the IEEE Engineering in Medicine & Biology Society (EMBC)*, 2021.

[4] D. Liaqat, S. Liaqat, J. L. Chen, T. Sedaghat, M. Gabel, F. Rudzicz, and E. de Lara, "Coughwatch: Real-world cough detection using smartwatches," in *ICASSP 2021 - 2021 IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP)*, 2021.

[5] T. Sedaghat, S. Liaqat, D. Liaqat, R. Wu, A. Gershon, T. Son, T. H. Falk, M. Gabel, A. Mariakakis, and E. de Lara, "Unobtrusive monitoring of copd patients using speech collected from smartwatches in the wild," in *2022 IEEE International Conference on Pervasive Computing and Communications Workshops and other Affiliated Events (PerCom Workshops)*, 2022.

[6] I. Ray, D. Liaqat, M. Gabel, and E. de Lara, "Skin tone, confidence, and data quality of heart rate sensing in wearos smartwatches," in *2021 IEEE International Conference on Pervasive Computing and Communications Workshops and other Affiliated Events (PerCom Workshops)*, 2021.

[7] J. Wang, Z. Feng, Z. Chen, S. George, M. Bala, P. Pillai, S.-W. Yang, and M. Satyanarayanan, "Bandwidth-efficient Live Video Analytics for Drones via Edge Computing," in *Proceedings of the IEEE/ACM Symposium on Edge Computing (SEC)*, 2018.

[8] T. Meuser, L. Lovén, M. Bhuyan, S. G. Patil, S. Dustdar, A. Aral, S. Bayhan, C. Becker, E. de Lara, A. Y. Ding *et al.*, "Revisiting edge ai: Opportunities and challenges," *IEEE Internet Computing*, vol. 28, no. 4, pp. 49–59, 2024.

[9] S. Hossein Mortazavi, M. Salehe, M. Gabel, and E. de Lara, "Data management systems for the hierarchical edge," *GetMobile: Mobile Comp. and Comm.*, vol. 27, no. 2, p. 11–17, Aug. 2023.

[10] "Amazon ECS clusters in Local Zones, Wavelength Zones, and AWS Outposts," 2023. [Online]. Available: https://docs.aws.amazon.com/AmazonECS/latest/developerguide/cluster-regions-zones.html

[11] "Azure Stack Edge release notes," 2023. [Online]. Available: https://learn.microsoft.com/en-us/azure/databox-online/azure-stack-edge-gpu-2202-release-notes

[12] T. Akidau, R. Bradshaw, C. Chambers, S. Chernyak, R. J. Fernández-Moctezuma, R. Lax, S. McVeety, D. Mills, F. Perry, E. Schmidt, and S. Whittle, "The Dataflow Model: A Practical Approach to Balancing Correctness, Latency, and Cost in Massive-Scale, Unbounded, Out-of-Order Data Processing," in *Proceedings of the International Conference on Very Large Data Bases (VLDB)*, 2015.

[13] 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

[14] 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 *Proceedings of the IEEE/ACM Symposium on Edge Computing (SEC)*, 2016.

[15] X. Fu, T. Ghaffar, J. C. Davis, and D. Lee, "Edgewise: a Better Stream Processing Engine for the Edge," in *Proceedings of the USENIX Annual Technical Conference (USENIX ATC)*, 2019.

[16] 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.

[17] P. Liu, D. Da Silva, and L. Hu, "DART: A Scalable and Adaptive Edge Stream Processing Engine," in *Proceedings of the USENIX Annual Technical Conference (USENIX ATC)*, 2021.

[18] B. Ramprasad, P. Mishra, M. Thiessen, H. Chen, A. da Silva Veith, M. Gabel, O. Balmau, A. Chow, and E. de Lara, "Shepherd: Seamless Stream Processing on the Edge," in *Proceedings of the IEEE/ACM Symposium on Edge Computing (SEC)*, 2022.

[19] P. Carbone, M. Fragkoulis, V. Kalavri, and A. Katsifodimos, "Beyond analytics: The evolution of stream processing systems," in *Proceedings of the 2020 ACM SIGMOD international conference on Management of data*, 2020, pp. 2651–2658.

[20] J. Verwiebe, P. M. Grulich, J. Traub, and V. Markl, "Survey of window types for aggregation in stream processing systems," *The VLDB Journal*, vol. 32, no. 5, pp. 985–1011, 2023.

[21] Y. Wu and K.-L. Tan, "ChronoStream: Elastic Stateful Stream Computation in the Cloud," in *Proceedings of the International Conference on Data Engineering (ICDE)*, 2015.

[22] S. Rajadurai, J. Bosboom, W.-F. Wong, and S. Amarasinghe, "Gloss: Seamless Live Reconfiguration and Reoptimization of Stream Programs," in *Proceedings of ASPLOS*, 2018.

[23] P. A. Tucker, D. Maier, T. Sheard, and L. Fegaras, "Exploiting punctuation semantics in continuous data streams," *IEEE Transactions on Knowledge and Data Engineering*, vol. 15, no. 3, pp. 555–568, 2003.

[24] L. Mai, K. Zeng, R. Potharaju, L. Xu, S. Suh, S. Venkataraman, P. Costa, T. Kim, S. Muthukrishnan, V. Kuppa *et al.*, "Chi: A Scalable and Programmable Control Plane for Distributed Stream Processing Systems," in *Proceedings of the International Conference on Very Large Data Bases (VLDB)*, 2018.

[25] S. Kulkarni, N. Bhagat, M. Fu, V. Kedigehalli, C. Kellogg, S. Mittal, J. M. Patel, K. Ramasamy, and S. Taneja, "Twitter Heron: Stream Processing at Scale," in *Proceedings of the ACM SIGMOD International Conference on Management of Data*, 2015.

[26] G. Wang, L. Chen, A. Dikshit, J. Gustafson, B. Chen, M. J. Sax, J. Roesler, S. Blee-Goldman, B. Cadonna, A. Mehta *et al.*, "Consistency and Completeness: Rethinking Distributed Stream Processing in Apache Kafka," in *Proceedings of the ACM SIGMOD International Conference on Management of Data*, 2021.

[27] "Apache Flink," 2023. [Online]. Available: http://flink.apache.org/

[28] Y. Mao, Y. Huang, R. Tian, X. Wang, and R. T. Ma, "Trisk: Task-Centric Data Stream Reconfiguration," in *Proceedings of the ACM Symposium on Cloud Computing (SoCC)*, 2021.

[29] V. Gulisano, R. Jimenez-Peris, M. Patino-Martinez, C. Soriente, and P. Valduriez, "Streamcloud: An Elastic and Scalable Data Streaming System," *IEEE Transactions on Parallel and Distributed Systems*, vol. 23, no. 12, 2012.

[30] M. A. Shah, J. M. Hellerstein, S. Chandrasekaran, and M. J. Franklin, "Flux: An Adaptive Partitioning Operator for Continuous Query Systems," in *Proceedings of the International Conference on Data Engineering (ICDE)*, 2003.

[31] R. Gu, H. Yin, W. Zhong, C. Yuan, and Y. Huang, "Meces: Latency-efficient Rescaling via Prioritized State Migration for Stateful Distributed Stream Processing Systems," in *Proceedings of the USENIX Annual Technical Conference (USENIX ATC)*, 2022.

[32] B. Del Monte, S. Zeuch, T. Rabl, and V. Markl, "Rhino: Efficient Management of Very Large Distributed State for Stream Processing Engines," in *Proceedings of the SIGMOD International Conference on Management of Data*, 2020.

[33] P. Carbone, S. Ewen, G. Fóra, S. Haridi, S. Richter, and K. Tzoumas, "State Management in Apache Flink: Consistent Stateful Distributed Stream Processing," in *Proceedings of the International Conference on Very Large Data Bases (VLDB)*, 2017.

[34] "ZeroMQ," 2023. [Online]. Available: https://zeromq.org/

[35] S. H. Mortazavi, M. Salehe, B. Balasubramanian, E. de Lara, and S. PuzhavakathNarayanan, "Sessionstore: A Session-aware Datastore for the Edge," in *Proceedings of the IEEE International Conference on Fog and Edge Computing (ICFEC)*, 2020.

[36] "Pathstore Github," 2023. [Online]. Available: https://github.com/PathStore/pathstore-all

[37] S. H. Mortazavi, M. Salehe, M. Gabel, and E. d. Lara, "Feather: Hierarchical Querying for the Edge," in *Proceedings of the IEEE/ACM Symposium on Edge Computing (SEC)*, 2020.

[38] Q.-C. To, J. Soto, and V. Markl, "A Survey of State Management in Big Data Processing Systems," in *Proceedings of the International Conference on Very Large Data Bases (VLDB)*, 2018.

[39] A. Tiwari, B. Ramprasad, S. H. Mortazavi, M. Gabel, and E. d. Lara, "Reconfigurable Streaming for the Mobile Edge," in *Proceedings of the*

International Workshop on Mobile Computing Systems and Applications (HotMobile), 2019.

[40] H. Röger and R. Mayer, “A Comprehensive Survey on Parallelization and Elasticity in Stream Processing,” *ACM Computing Surveys (CSUR)*, vol. 52, no. 2, 2019.

[41] “ETSI MEC Sandbox,” 2023. [Online]. Available: https://try-mec.etsi.org/

[42] “Multi-access Edge Computing (MEC); Application mobility service API,” 2023. [Online]. Available: https://www.etsi.org/deliver/etsi_gs/MEC/001_099/021/02.02.01_60/gs_mec021v020201p.pdf

[43] M. Hoffmann, A. Lattuada, F. McSherry, V. Kalavri, J. Liagouris, and T. Roscoe, “Megaphone: Latency-conscious State Migration for Distributed Streaming Dataflows,” in *Proceedings of the International Conference on Very Large Data Bases (VLDB)*, 2019.

[44] “Apache ActiveMQ Artemis,” 2023. [Online]. Available: https://activemq.apache.org/components/artemis/

[45] “ActiveMQ Artemis Filter Expressions,” 2023. [Online]. Available: https://activemq.apache.org/components/artemis/documentation/latest/filter-expressions

[46] A. Lakshman and P. Malik, “Cassandra: a Decentralized Structured Storage System,” *ACM SIGOPS Operating Systems Review*, vol. 44, no. 2, 2010.

[47] “Flink Github,” 2023. [Online]. Available: https://github.com/apache/flink

[48] “Trisk Github,” 2023. [Online]. Available: https://github.com/sane-lab/Trisk

[49] “Meces Github,” 2023. [Online]. Available: https://github.com/ATC2022No63/Meces

[50] P. Tucker, K. Tufte, V. Papadimos, and D. Maier, “Nexmark–a Benchmark for Queries Over Data Streams,” Technical Report. Technical report, OGI School of Science & Engineering, Tech. Rep., 2008.

[51] “Nexmark Github,” 2023. [Online]. Available: https://github.com/nexmark/nexmark

[52] A. Arasu, M. Cherniack, E. Galvez, D. Maier, A. S. Maskey, E. Ryvkina, M. Stonebraker, and R. Tibbetts, “Linear road: a stream data management benchmark,” in *Proceedings of the Thirtieth international conference on Very large data bases-Volume 30*, 2004, pp. 480–491.

[53] “Linear Road Webpage,” 2024. [Online]. Available: https://www.cs.brandeis.edu/~linearroad/

[54] “Linux Traffic Control,” 2023. [Online]. Available: https://access.redhat.com/documentation/en-us/red_hat_enterprise_linux/8/html/configuring_and_managing_networking/linux-traffic-control_configuring-and-managing-networking

[55] J. L. Chen, D. Liaqat, M. Gabel, and E. de Lara, “Starlight: Fast Container Provisioning on the Edge and over the WAN,” in *Proceedings of the USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, 2022.

[56] “Spark Streaming,” 2023. [Online]. Available: https://spark.apache.org/streaming/

[57] “Apache Storm,” 2023. [Online]. Available: https://storm.apache.org/

[58] L. Xu, B. Peng, and I. Gupta, “Stela: Enabling Stream Processing Systems to Scale-in and Scale-out On-demand,” in *Proceedings of the IEEE International Conference on Cloud Engineering (IC2E)*, 2016.

[59] T. Heinze, Z. Jerzak, G. Hackenbroich, and C. Fetzer, “Latency-aware Elastic Scaling for Distributed Data Stream Processing Systems,” in *Proceedings of the ACM International Conference on Distributed Event-Based Systems*, 2014.

[60] R. Castro Fernandez, M. Migliavacca, E. Kalyvianaki, and P. Pietzuch, “Integrating Scale Out and Fault Tolerance in Stream Processing Using Operator State Management,” in *Proceedings of the SIGMOD international conference on Management of Data*, 2013.

[61] B. Ramprasad, P. Mishra, M. L. Peixoto, and E. de Lara, “StreamBucket: In-Network Adaptation for Late-binding Stream Processing Systems,” in *Proceedings of the IEEE International Conference on Cloud Networking (CloudNet)*, 2024.

[62] G. Bartolomeo, M. Yosofie, S. Bäurle, O. Haluszczynski, N. Mohan, and J. Ott, “Oakestra: A lightweight hierarchical orchestration framework for edge computing,” in *2023 USENIX Annual Technical Conference (USENIX ATC 23)*, 2023, pp. 215–231.

[63] E. Volnes, T. Plagemann, and V. Goebel, “To Migrate or Not to Migrate: An Analysis of Operator Migration in Distributed Stream Processing,” *IEEE Communications Surveys & Tutorials*, 2023.

[64] T. Le, “A survey of live virtual machine migration techniques,” *Computer Science Review*, vol. 38, p. 100304, 2020.