Falcon: Live Reconfiguration for Stateful Stream Processing on the Edge

Pritish Mishra
University of Toronto
pritish@cs.toronto.edu

Nelson Bore
McGill University
nelson.bore@mail.mcgill.ca

Brian Ramprasad
University of Toronto
brianr@cs.toronto.edu

Myles Thiessen
University of Toronto
mthiessen@cs.toronto.edu

Moshe Gabel
University of Toronto
mgabel@cs.toronto.edu

Alexandre da Silva Veith
Nokia Bell Labs
alexandre.da_silva_veith@nokia-bell-labs.com

Oana Balmau
McGill University
oana.balmau@cs.mcgill.ca

Eyal de Lara
University of Toronto
delara@cs.toronto.edu

ABSTRACT

Stream processing is an attractive paradigm for deploying applications in geo-distributed edge-cloud environments. The reverse economics of scale of edge networks, as well as the movement of data sources between edges, however, require the ability to dynamically reconfigure application deployment to adapt to workload variations and user mobility. Unfortunately, existing stream processing engines are ill-suited for edge-cloud environments: they either stop application processing while reconfiguration takes place or require an expensive 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, as well as 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.

PVLDB Reference Format:
Pritish Mishra, Nelson Bore, Brian Ramprasad, Myles Thiessen, Moshe Gabel, Alexandre da Silva Veith, Oana Balmau, and Eyal de Lara. Falcon: Live Reconfiguration for Stateful Stream Processing on the Edge. PVLDB, 16(1): XXX-XXX, 2023.
doi:XX.XX/XXX.XX
PVLDB Artifact Availability:
The source code, data and/or other artifacts could not **yet** be made available during the paper submission due to the patenting process. We commit to make them available after we apply for a provisional patent application.

Proceedings of the VLDB Endowment, Vol. 16, No. 1 ISSN 2150-8097.
doi:XX.XX/XXX.XX

1 INTRODUCTION

Emerging mobile and edge applications (e.g., traffic monitoring [30], autonomous driving [34], and augmented reality [46]) pose significant cost, bandwidth, and latency constraints. These new applications produce large amounts of data from edge sensors and have latency requirements in the range of a few tens of milliseconds, with the added challenge of data source mobility. Cloud providers are anticipating new applications' needs by moving compute closer to the data with a rollout of a *geo-distributed hierarchical infrastructure*, where progressively smaller datacenters are deployed closer to the edge of the network near base stations. This hierarchy can span multiple tiers on the path to the largest root datacenter. For instance, Amazon provides such services via AWS Wavelength, Outposts, and Local Zones [2], and Microsoft recently introduced Azure Stack Edge [6].

Stream processing systems are an ideal candidate for hierarchical datacenter deployments. These frameworks (e.g., Flink [4], Storm [5]) structure the application as a dataflow graph whose vertices represent operators and edges represent the data streams between operators [19] and have been successfully used in cloud environments for applications, such as data analytics, online maps, ads serving, video streaming, and fraud detection [18, 24]. This works well for the hierarchical setting, as processing data closer to where it is created has the potential for significant bandwidth reduction, order-of-magnitude lower latency, and better load balancing.

Stream processing in hierarchical datacenter deployments, however, has to address two new challenges: (1) the reverse economics of scale of edge networks (i.e., smaller size of the edge datacenters leads to higher per-unit costs) create a need for *frequent operator reconfiguration* across the edge-cloud hierarchy; (2) the limited coverage area of edge data centers (e.g., a city or neighborhood instead of a county) requires accounting for the *movement of data sources*.

Figure 1 illustrates how the reverse economics of scale of edge networks require frequent reconfiguration in an example real-time traffic monitoring application consisting of an *aggregation operator* (A) that aggregates information from images generated by motion-activated street cameras, and a *control operator* (C) that analyzes the resulting events to generate decisions directing the traffic. Figure 1a

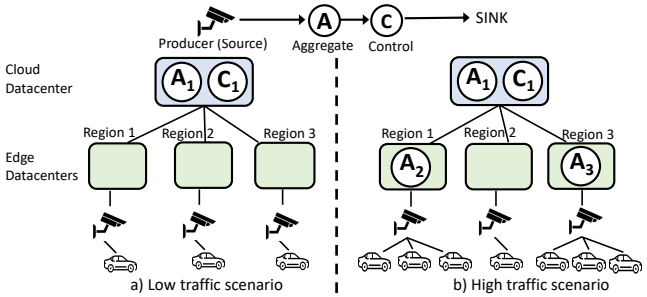


Figure 1: State reconfiguration for a stream processing application deployed in an edge-cloud environment.

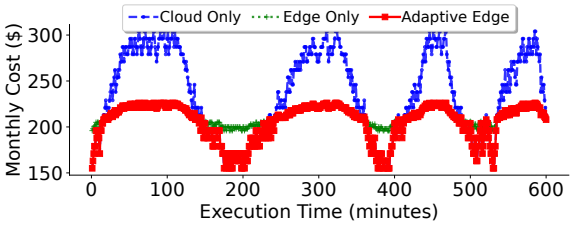


Figure 2: Cost of traffic monitoring application when deployed on cloud only (dashed), edge only (dotted), or using an adaptive strategy (solid) that reconfigures the application operators. Figure reproduced from [39].

shows that when car traffic is low, it is most beneficial to run operators only in the cloud datacenter. 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). As shown in Figure 2, deploying this traffic monitoring application by adaptively moving operators between cloud and edge resources results in a lower cost than running them solely on the cloud or the edge [39]. Thus, to optimize cost, stream processing applications running on edge networks require the ability to dynamically reconfigure application deployment with minimal interruptions to running client applications, for instance, caused by disruptions in tuple processing latency.

Unfortunately, the techniques used to reconfigure stream processing applications in the cloud (i.e. full-restart [4], partial pause [27, 42], on-demand state transfer [26], and hot backups [25]) are inappropriate for reconfiguring applications deployed on edge networks. These approaches use early binding designs where the socket connections between all upstream and downstream operators must be coordinated globally and re-established after reconfiguration. While these designs are appropriate for the cloud where latency is low, our evaluation shows that on hierarchical cloud-edge networks these techniques result in disruptions that last tens of seconds, and latency peaks that are 3-4 orders of magnitude higher than the steady state latency. Moreover, none of the cloud frameworks address source mobility between edges, as all the data is processed in a single datacenter.

Recently, our prior work, Shepherd [39] introduced an alternative approach that supports dynamic reconfiguration of *stateless* operators on edge networks. Shepherd uses a network of software routers to transfer data tuples between operators and implements a late binding approach to routing that enables dynamic reconfiguration of *stateless* operator replicas with minimal application disruption. Shepherd, however, does not support stateful operators (such as the aggregation operator in the above vehicle monitoring example) and does not support data source mobility.

We present Falcon, a new stream processing system designed for *stateful* applications running on the hierarchical edge-cloud. Falcon builds on Shepherd's approach to tuple routing, but applies it to the more challenging problem of stateful operator reconfiguration, which requires maintaining *state correctness* [45]. To achieve this, the state needs to be migrated between operator instances, while tuples continue to 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. Falcon preserves the semantics of in-order and exactly-once processing, supports count and event-based window states, which are commonly used by applications today [40], and supports data source mobility.

Falcon uses a novel *live key migration* protocol to move the state between nodes. Falcon relies on three techniques that work together to achieve seamless live migration. *Dual routing* creates a 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* injects special 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*, allow an operator instance to synchronize state by processing buffered tuples without emitting output, thus avoiding the need to later de-duplicate emitted tuples.

We evaluate Falcon on a geo-distributed edge-cloud network of AWS datacenters and compare it against state-of-the-art frameworks for stateful streaming: Flink [21], Trisk [36] and Meces [26]. During reconfiguration, Falcon achieves a disruption lasting 10–15 milliseconds, which is lower than the round-trip time to the root datacenter. In contrast, disruption in state-of-the-art frameworks goes up to several tens of seconds. Moreover, Falcon reduces the peak latency compared to steady state by up to 4 orders of magnitude. Falcon's latency peak during reconfiguration is 40–45 milliseconds, compared to several tens of seconds for its 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.
(2) The design and implementation of Falcon, a new open-source stream processing framework designed for the hierarchical edge-cloud. Falcon is the first system to support low-latency state migration during operator reconfiguration and source mobility. We plan to open-source Falcon upon paper publication.
(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.

2 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.

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 migrated state can be large if the windowing duration is long, if there are multiple open windows, or if the window elements are large.

Users can specify the operators and how data flows between them using an abstraction called a *logical plan* [31]. During application deployment, users (or a placement component) also define a *physical plan*, which specifies the instances that must be spawned for each logical operator and on which computing node to place each operator instance. A *reconfiguration plan* introduces modifications to 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.

2.1 State in Stream Processing Frameworks

Stream processing frameworks allow the splitting of the data stream into multiple sub-streams based on keys that represent a unique data attribute 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.

Reconfiguration of stateful operators requires migrating the state stored by these keys to a different operator instance at the new location. The data flow belonging to these keys must be switched to the new operator instance and subsequent tuples (including the ones in-flight) need to be processed at the new location.

A key requirement in the reconfiguration process is maintaining *state correctness* [45]. 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.

2.2 Reconfiguration in the Edge

Figure 3 demonstrates the application performance when using current stream processing frameworks to reconfigure deployment of the traffic monitoring application (Figure 1) where processing of data from Region 1 is moved from A_1 in the cloud to a new A_2 on the edge node, for example as a result of a 5G MEC notification [11, 49].

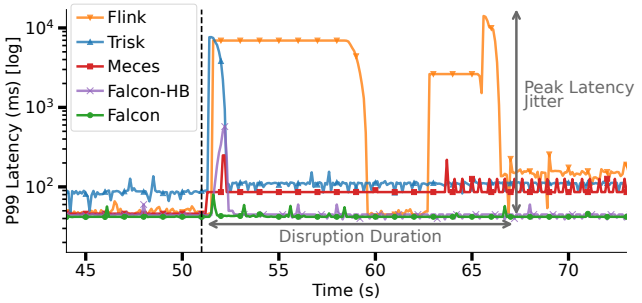


Figure 3: P99 latency during reconfiguration for a two-tier (edge-cloud) infrastructure deployment. The dashed black line indicates the reconfiguration trigger.

State-of-the-art systems use one of the following strategies when handling reconfiguration: *full-restart* (for Flink), *partial-pause* (for Trisk, Meces), or *hot backups* (for Falcon-HB). Figure 3 shows the behavior of systems implementing these approaches during reconfiguration, as well as the behavior of Falcon, our proposed solution.

Apache Flink [4] 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 [36] implements a partial-pause approach [27, 42] 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 [26] 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.

Rhino [25] is an example of the *hot backups* approach that maintains up-to-date replicas of the operator state at other nodes, and switches processing between backups as needed. Rhino still incurs stoppage while 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 since resources are expensive. Since Rhino is not open-source, we implement its hot backup strategy on top of Falcon. This Falcon-HB version benefits from the late-binding design (Section 3.3), which explains the low disruption duration.

Cloud-based systems like Flink, Trisk, and Meces incur higher latency after reconfiguration (at the 67 seconds mark) because they deploy a single message broker in the cloud to which all sources connect. Consequently, tuples generated at the edge are triangularly routed through the cloud to be processed back at the edge (Region 1). In cloud deployments, this solution is not detrimental, because all compute nodes are connected via low latency links. However, in an edge-cloud deployment, reconfiguration leads to an increase in latency which is proportional to the depth of the datacenter hierarchy. In contrast, Falcon uses a hierarchy of brokers which allows localized processing of tuples, and thus application latency remains unchanged for Falcon (and Falcon-HB).

Finally, Falcon reduces the peak latency by 1 order of magnitude compared to Meces. Moreover, the disruption duration is decreased to 10–15 milliseconds. Our analysis shows 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 for applications used in the Linear Road benchmark [20]. Given the poor reconfiguration performance, we conclude that existing stream processing frameworks cannot efficiently support source mobility, which is a crucial requirement for edge applications.

3 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. We do, however, make one important assumption: operators are deterministic, i.e., replay of a tuple after restoring the operator state from a checkpoint results in the same state.

3.1 System Overview

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.

Deployment on the edge-cloud. Falcon's design assumes that the root datacenter (the cloud) has a global view of the deployed operator instances, along with their keyspaces. In addition, the root datacenter has an instance of each operator in the application DAG and is thus capable of processing tuples that were not processed at lower levels of the hierarchy. The root datacenter also manages reconfiguration by redeploying operator instances (replicas of the homologous operators in the cloud) following a reconfiguration plan or detection of source mobility. In the rest of the datacenter hierarchy (i.e., intermediate nodes, and leaves), Falcon installs late-binding routers. The routers send incoming tuples to the operator instance on the current node if possible, or send it up to the parent datacenter.

Keyed state overview. 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 [21, 26, 36], Falcon's design allows handpicking individual keys for migration to better support source mobility without sacrificing scalability.

System design. 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 3.2, 3.4 and 3.5), and it manages source mobility (Section 3.6).
- 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 3.3).
- Several *Workers* can be deployed at each datacenter node in the edge-cloud hierarchy. Each Worker can serve multiple operator instances (e.g., one per core), assigned by the Job Manager. Each operator instance maintains an operator process, a state manager, and an I/O port implemented as a ZeroMQ socket [16]. The operator process reads tuples from its operator queue, updates the state in memory, and writes output using the I/O port. To maintain state, each datacenter node has a replica of a distributed key-value store, which Workers share.

3.2 Keyed State and Windows

State in Falcon is partitioned by keys, a common approach for stateful operators in stream processing engines [43]. To preserve correct processing semantics, an instance of a stateful operator can only write to the keys it controls, and must not mix state from different keys. For example, an operator A maintaining different counts for keys X and Y may not emit the sum of these counts, since Falcon or the client may choose to move the processing of Y keys to a different machine. Instead, the counts emitted by A must be processed by a subsequent operator B to compute the sum.

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 which operator instance processes which keys, 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 are handled by storing tuples that fall in the window as part of the operator state, with separate windows for separate keys. Once a window is closed, its tuples are handed 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 [21]. During the key migration, all open windows for the key are seamlessly migrated as part of the operator state.

3.3 Tuple Routing

Falcon generalizes the late-binding routing design proposed in prior work on stateless stream processing [39]. In existing designs, the routers inspect each arriving tuple. If there is a matching operator in the datacenter, tuples are processed locally; otherwise, the router forwards the tuple to the parent node. Late-binding allows for independent deployment of an operator without knowing the location of its upstream or downstream operators.

We generalize the design above to handle keyed state and windows both crucial for modern stream processing applications [40]. Seamless state migration is also beneficial for environments where source mobility is common. Falcon extends late-binding tuple routing as follows.

The Falcon Routers make decisions based on three tuple attributes: application ID, operator type and key. Figure 4 demonstrates routing in a two-operator application (source → A → 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 A,car meant for operator A with key "car". ② The router in the edge1 datacenter routes A,car tuples to the local operator A_{car}. ③ A_{car} emits tuples 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 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 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 A,van tuples are routed to local operator A_{van}. ⑧ A_{van} emits tuples C,van, which are routed to local operator C which accepts all keys.
- *Green flow (dotted):* ⑨ 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.

3.4 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).

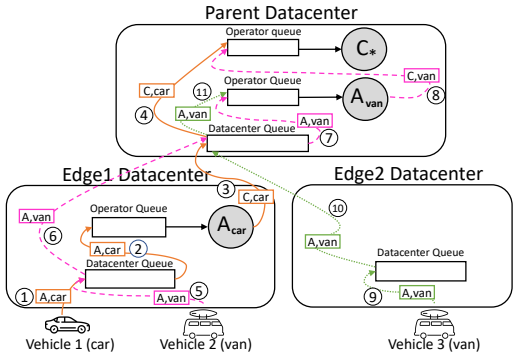


Figure 4: Example of tuple routing in Falcon. Operator instances A_{car} and A_{van} process tuples of car and van keys, while operator instance C_* processes for all keys.

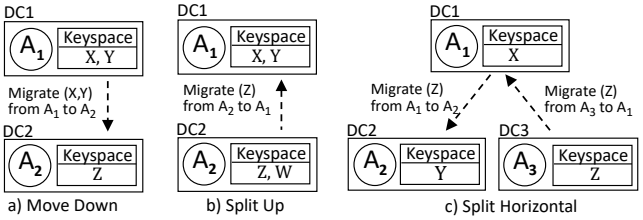


Figure 5: Example reconfiguration plans that can be composed using Falcon's Migrate primitive.

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 5a). *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 5b). 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 5c). 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.

3.5 The Live Key Migration Protocol

The *live key migration* protocol is one of the two core mechanisms of Falcon, together with the source mobility protocol. Live key migration implements the Migrate primitive (Section 3.4). 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 relies on three techniques that work together. *Dual routing* creates a duplicate data

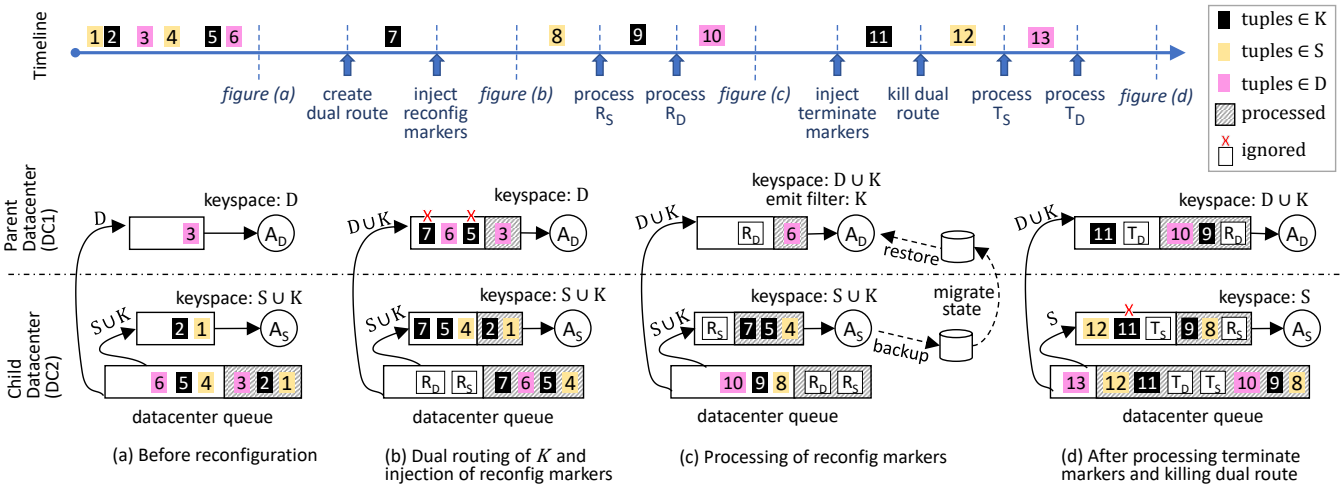


Figure 6: Reconfiguration steps when migrating the key K from operator instance A_S in a child datacenter to instance A_D in a parent datacenter. Annotated timeline of arriving tuples indicates the reconfiguration steps and each figure presents a snapshot of the system at a particular point of the timeline. See Section 3.5 for more details.

flow that routes tuples to both the source and destination instances. *Marker-based synchronization* injects special 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 4.4). Lastly, *emission filters*, 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 6 shows the steps of the protocol, which we next discuss in detail. Consider a *Split Up* operation (Figure 5b) composed using our migrate primitive - 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 6a, 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 6b, 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 markers into the datacenter queue of the child datacenter (DC2 in Figure 6b). 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 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 5) and is exactly the same for both upward and downward directions. The dual routing design ensures that there are minimal delays in the processing of tuples belonging to K. The only break in processing is waiting for the acknowledgment from the source to the destination instance which is approximately half the round trip network latency. This is necessary to maintain the strict guarantees of in-order tuple processing.

When choosing an operator instance for the migrated key in the destination datacenter, it is important to select an instance with sufficient spare capacity for processing of the keys being migrated. This also ensures that it can complete tuple replay and catch up to the source instance. If no existing instance has sufficient spare capacity, it is straightforward to spin up a new operator instance for the migrated keys since states are not shared and the protocol ensures starting a new instance does not incur processing disruptions. (Note that this work focuses on the mechanism for live migration. Mechanisms for data center capacity management or for selecting which keys to migrate are beyond the scope of this 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 after reconfiguration (Phase 6) are processed at the destination instance only after tuples that arrived during reconfiguration (being processed at the source instance). This is achieved in Phase 6 by making the destination instance wait until the terminate marker has been processed and emitted by the source instance.
- In addition, Falcon injects the two sets of 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 4.2), 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.

3.6 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: 1) switching the network connection between edge routers, 2) reconfiguring the stream processing application.

Router switchover. Falcon registers all routers deployed on the edge nodes, including their IP addresses, using the Edge Platform Application Enablement in the ETSI MEC [7]. A data source (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 [11]). 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 7a. 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 7). 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 3.5, 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 7e) 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.

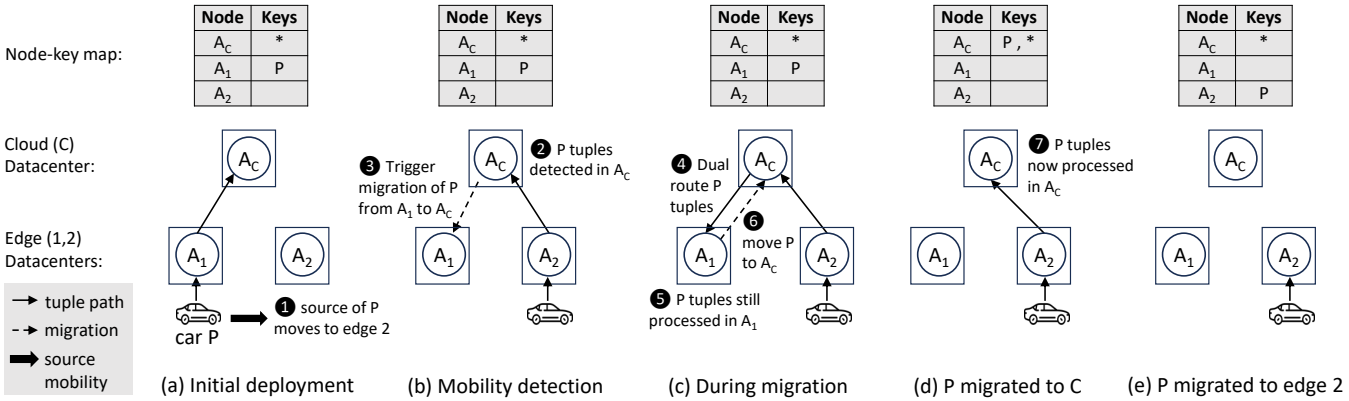


Figure 7: Mobility of car emitting key P moving from edge 1 to edge 2. For simplicity, we only show one operator (A), replicated on cloud C, edge 1, and edge 2. Solid arrows indicate flow of P tuples. See Section 3.6 for more details.

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 7c, 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.

3.7 Fault Tolerance

For fault tolerance, Falcon relies on a set of standard assumptions: (1) operators are deterministic, (2) failures are not permanent, and (3) the underlying message broker provides exactly-once processing, is fault tolerant, and supports tuple replay and acknowledgement. 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 [21] and other frameworks [25, 26, 29, 36].

During reconfiguration, there are two points of failure: message brokers and operator instances. Both these failures could occur in Phases 1 and 5 of the live key migration protocol (Section 3.5). Phases 2 and 4 depend only on the broker and Phases 3 and 6 can only be affected by failure of the operator instances. To handle message broker failures during reconfiguration, Falcon waits for recovery and retries failed operations (via dual routing, and marker injection). If an operator instance fails during reconfiguration, Falcon relies on replay: since coordination is based on markers and the broker is fault tolerant, instances that failed after marker processing are restarted with tuples and markers re-delivered.

3.8 Implementation

We intend to open-source Falcon upon paper publication. Falcon is implemented in Java (approx. 50K LOC). Application operators are implemented as Java applications running inside Docker containers. We use Apache ActiveMQ Artemis [3] as the message broker to implement our routing system, and implement Falcon's routing rules as Artemis filters within the message queues. The dual-routing technique is implemented by adding diverts within the queues for routing tuples to two locations simultaneously. These diverts use custom filter expressions [1] to detect if the tuples belong to a specific key. To reduce latency for situations where complex routing is not needed, operators inside the same datacenter communicate via ZeroMQ sockets [16] and brokers are used only for communication across datacenters. To store operator state, we use our prior open-sourced work, SessionStore [13, 37] as a geo-distributed persistent key-value store. The implementation of SessionStore uses Cassandra, a popular distributed key-value store [32].

4 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 4.2)
(2) What is the impact of source mobility? (Section 4.3)
(3) How do network latency and topology size affect Falcon's reconfiguration performance? (Section 4.4)
(4) How do the application characteristics affect Falcon's reconfiguration performance? (Section 4.5)

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

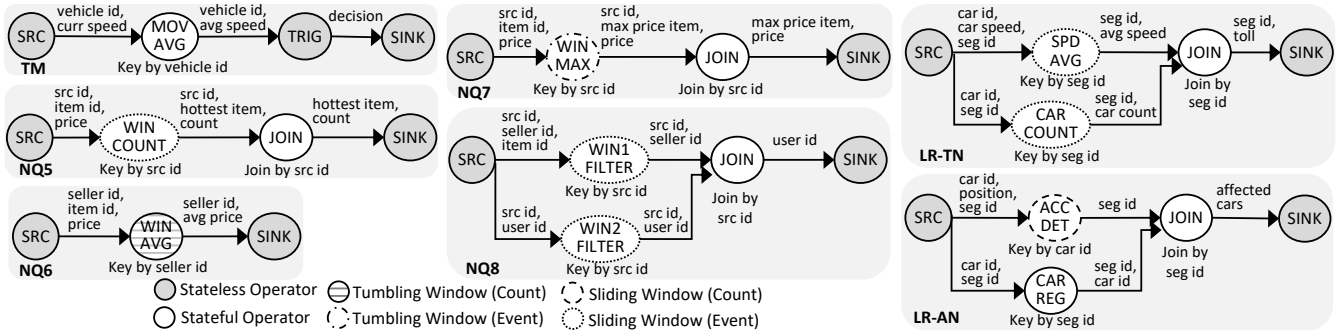


Figure 8: Logical plan of TM, Nexmark Benchmark (NQ5-8), and Linear Road Benchmark (LR-TN, LR-AN) applications.

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.

4.1 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 11. 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. We compare Falcon to baselines representing full-restart (Flink [8]), partial-pause (Trisk [15]), on-demand state transfer (Meces [10]) and hot backups (Falcon-HB). Falcon-HB's hot backups mechanism is inspired by Rhino [25] 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. Note that we do not include Shepherd in the evaluation, as it does not support stateful operators – the focus of Falcon's techniques. We use the Apache ActiveMQ Artemis message broker for all the frameworks.

Applications. We evaluate applications representing 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].

Figure 8 shows the logical plans. Since, at the time of this writing, there is no standard benchmark for edge-based stream processing applications, we create the traffic monitoring application TM (used as a running example throughout the paper), along with adapting four workloads from Nexmark [12, 44] and two workloads from Linear road [17, 20] for an edge-cloud hierarchy. Both Nexmark and Linear road are popular benchmarks and the workloads we select are typically used in the evaluation of stream processing engines [25, 26, 29, 36].

(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 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) uses a count-based tumbling window (WIN AVG) to calculate the average selling price of the last 10 items sold by a seller from a stream of auction bids. Finally, Query 8 (NQ8) uses two filters in the event-based sliding windows (WIN1, WIN2 FILTER) to respectively find the users that joined the system in the last hour and the ones that submitted a bid in that period. The users common in the two filtered results are determined by the JOIN operator. This query uses long-running windows of size 1 hour with 1-second slide that emits 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 calculates the toll 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. Similarly, CAR COUNT operator calculates the number of cars on the segment.

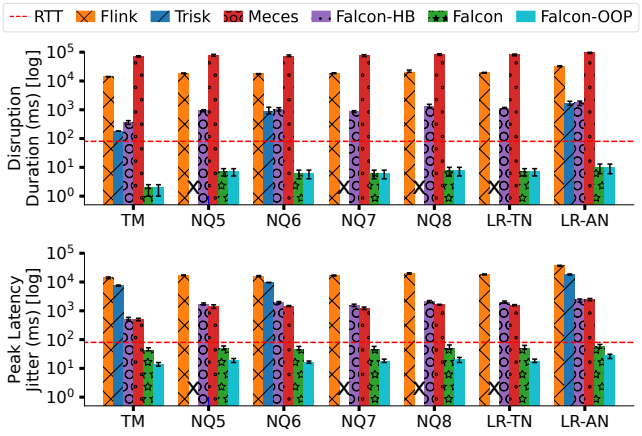


Figure 9: Reconfiguration stoppage. Dashed red line indicates round-trip inter-datacenter latency. Trisk does not support reconfiguration for event-based windows (marked X).

Finally, JOIN operator uses both these values to calculate the toll for a segment (See [20] for the specific formula).

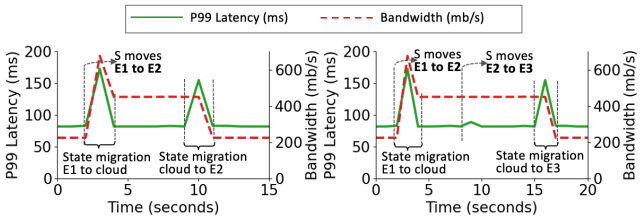
Accident Notification (LR-AN) query determines which cars are affected when an accident occurs on a segment. ACC DET (Accident Detect) operator uses a count-based sliding window (size=10, slide=1) to report an accident if the last 10 positions of a car are same. CAR REG (Car Register) operator maintains 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 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.

4.2 Reconfiguration Performance

We evaluate the impact of handling one reconfiguration. Initially, all operators are deployed on the cloud node. After 60 seconds, for each experiment, we trigger a reconfiguration that creates a new instance of the stateful operator of the application (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 migrates 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. We therefore limit our experiments to a scenario where tuples are uniformly distributed across data sources – allowing an advantage for the baseline systems.

Figure 9 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 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



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

Figure 10: 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.

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 (Section 4.5, cost of hot backups). 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. 3.5, 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.

4.3 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 and simulate the mobility using the MEC Sandbox [7] 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 this key accordingly. We do not include Flink, Trisk and Meces in this experiment, as they do not support the mobility of sources.

Figure 10 illustrates the 99^{th} percentile latency (solid green line) and total bandwidth utilization (dashed red line) over time, as the data source transitions from (a) edge 1 to edge 2 and (b) from edge 1 to edge 3, via edge 2. The latency is calculated 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 seconds mark (in both Figures 10a and 10b) are caused by the state migration from edge 1 to cloud, which results in a latency increase of approximately 80ms. The mobility detection latency is negligible, in the order of a few milliseconds. Second, the peaks at the 12 seconds mark (Figure 10a) and the 18 seconds mark (Figure 10b) 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

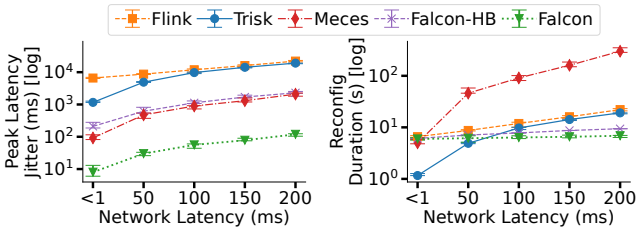


Figure 11: Effect of round-trip latency on peak latency jitter (left) and reconfiguration duration (right).

moving the processing from cloud to edge (Section 3.6). The third, smaller, peak that can be seen at 9 seconds in Figure 10b 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. 3.6). 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 4.2).

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

4.4 Impact of Network and Topology Size

We evaluate the impact of network latency on reconfiguration performance by using Linux Traffic Control [9] to add latency between the parent and the child nodes. The reconfiguration migrates the processing of 50% of the keys at the MOV AVG operator of the TM application to a new instance of the operator on the child node. We observe that the peak latency jitter increases as round-trip network latency grows for all frameworks except Falcon (Figure 11, left). Flink and Trisk need more time to migrate state due to increased bandwidth-delay product. Falcon-HB faces a similar problem when migrating tuples arrived 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 [23].

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 11, 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

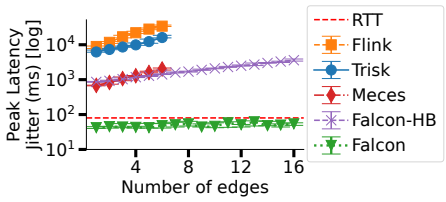


Figure 12: Impact of the number of edges on reconfiguration. Reconfiguration triggered on Flink, Trisk and Meces fails for 7 or more edges due to timeouts.

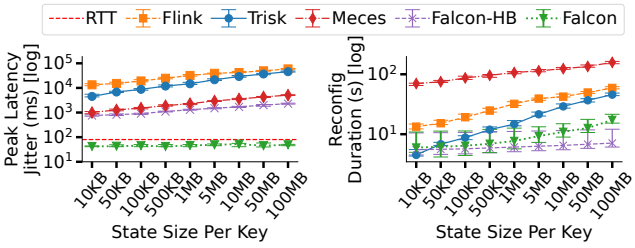


Figure 13: Impact of state size on peak latency jitter (left) and reconfiguration duration (right).

operator on the N^{th} edge and migrate the processing of a subset of the keys to this instance.

Figure 12 shows that Falcon readily scales to many edges: its reconfiguration performance is unaffected as we increase the number of edges. Conversely, adding edges leads to an exponential increase in the peak latency jitter for Flink, Trisk and Meces, and a linear increase for Falcon-HB. Flink, Trisk and Meces use an early-binding design where the socket connections between all upstream and downstream operators must be coordinated globally and re-established after reconfiguration. Increasing edges also increases the number of tuples that need to be replayed in Falcon-HB. 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. In fact, reconfiguration in Falcon involves only the source and the destination instances, irrespective of the number of operator instances in the physical plan. Since there is no dependency on the number of pre-existing operators, peak latency jitter is constant even if the number of edges increases.

4.5 Impact of Application Characteristics

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

In Figure 13 (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 13 (right) shows a similar pattern. Here, the increase in state size per key leads to an increase in the reconfiguration duration for Flink, Trisk and Meces. This is because 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,

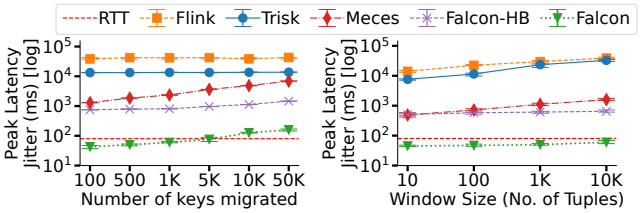


Figure 14: Impact of the number of keys (left) and window size (right) on reconfiguration performance.

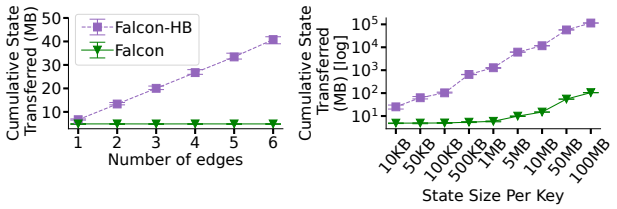


Figure 15: Impact of the number of edges (left, linear scale) and state size (right, log scale) on the total size of transferred data. The hot backup approach used by Falcon-HB incurs substantial overhead.

Falcon has to download the application state and fewer tuples during reconfiguration. Falcon-HB is able to achieve a low increase rate by avoiding the state download and only needs to download tuples that arrived since the last checkpoint.

In Figure 14 (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 14 (right), we evaluate the impact of window size on reconfiguration performance, by varying the size of the count-based window in the NQ6 application. We observe that an increase in window size leads to an increase in the peak latency jitter for all the frameworks except Falcon-HB. A larger window implies a larger state, as we do not use incremental averaging. Even for a window of 10,000 tuples, the peak latency jitter incurred by Falcon is lower than even the network latency.

Cost of hot backups. Network transfers incur costs, especially in edge deployments. To evaluate the network overhead incurred by hot backups, we calculate the total amount of data transferred between the cloud and edge datacenters for the duration of experiments conducted in Section 4.2. Falcon-HB's checkpointing interval is the same as those experiments, 500 milliseconds, as this yields the best reconfiguration performance.

The size of state transferred by Falcon-HB increases linearly with the number of edges (Figure 15, left) and the state size per key (Figure 15, 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.

5 RELATED WORK

Reconfiguration using full-restart in the cloud. Flink [21], Spark [14], Storm [5], and Stella [48] use a full-restart reconfiguration approach. The full restart interrupts 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 [22, 28, 35, 36, 42] reduces the system disruption time in redistributing operator workloads by pausing only the processing of the affected operators. For instance, Megaphone [29] splits the state into small parts and moves the state in increments. Meces [26] fetches the state for a key only when it encounters a tuple belonging to this key. In contrast, Rhino [25], Chronostream [47], and Gloss [38] 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 limits them to inefficient triangular routing for edge-cloud deployment and becomes a bottleneck during rescaling.

Stream processing for edge deployments. DART [33] uses a peer-to-peer overlay network to distribute operators on to edge datacenters. SpanEdge [41] provides a user-friendly programming environment for operator placement. Both systems take a full-restart approach for operator scaling and reconfiguration. Shepherd [39] supports reconfiguration for *stateless* operators with low disruption. The system avoids global coordination by using late binding routing. Built upon the late binding idea, our system addresses the more challenging scenario of reconfiguration in *stateful* applications, as well as providing support for source mobility.

6 CONCLUSION

We presented Falcon, a stream processing framework for live reconfiguration of stateful operators in a hierarchical edge-cloud deployment. This means two things: (1) support for moving the state between operator replicas; and (2) support for source mobility. To achieve reconfiguration and source mobility support, Falcon uses live key migration, marker-based synchronization, and emission filters. Falcon reduces reconfiguration latency from tens of seconds to a few milliseconds and supports a wide range of reconfiguration operations across various kinds of application states deployed on geo-distributed edge-cloud infrastructure.

REFERENCES

[1] 2023. ActiveMQ Artemis Filter Expressions. https://activemq.apache.org/components/artemis/documentation/latest/filter-expressions

[2] 2023. Amazon ECS clusters in Local Zones, Wavelength Zones, and AWS Outposts. https://docs.aws.amazon.com/AmazonECS/latest/developerguide/cluster-regions-zones.html

[3] 2023. Apache ActiveMQ Artemis. https://activemq.apache.org/components/artemis/

[4] 2023. Apache Flink. http://flink.apache.org/

[5] 2023. Apache Storm. https://storm.apache.org/

[6] 2023. Azure Stack Edge release notes. https://learn.microsoft.com/en-us/azure/databox-online/azure-stack-edge-gpu-2202-release-notes

[7] 2023. ETSI MEC Sandbox. https://try-mec.etsi.org/

[8] 2023. Flink Github. https://github.com/apache/flink

[9] 2023. Linux Traffic Control. https://access.redhat.com/documentation/en-us/red_hat_enterprise_linux/8/html/configuring_and_managing_networking/linux-traffic-control_configuring-and-managing-networking

[10] 2023. Meces Github. https://github.com/ATC2022No63/Meces

[11] 2023. Multi-access Edge Computing (MEC); Application mobility service API. https://www.etsi.org/deliver/etsi_gs/MEC/001_099/021/02.02.01_60/gs_mec021v020201p.pdf

[12] 2023. Nexmark Github. https://github.com/nexmark/nexmark

[13] 2023. Pathstore Github. https://github.com/PathStore/pathstore-all

[14] 2023. Spark Streaming. https://spark.apache.org/streaming/

[15] 2023. Trisk Github. https://github.com/sane-lab/Trisk

[16] 2023. ZeroMQ. https://zeromq.org/

[17] 2024. Linear Road Webpage. https://www.cs.brandeis.edu/~linearroad/

[18] Tyler Akidau, Alex Balikov, Kaya Bekiroglu, Slava Chernyak, Josh Haberman, Reuven Lax, Sam McVeety, Daniel Mills, Paul Nordstrom, and Sam Whittle. 2013. MillWheel: Fault-Tolerant Stream Processing at Internet Scale. In *Proceedings of the International Conference on Very Large Data Bases (VLDB)*.

[19] Tyler Akidau, Robert Bradshaw, Craig Chambers, Slava Chernyak, Rafael J. Fernández-Moctezuma, Reuven Lax, Sam McVeety, Daniel Mills, Frances Perry, Eric Schmidt, and Sam Whittle. 2015. 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)*.

[20] Arvind Arasu, Mitch Cherniack, Eduardo Galvez, David Maier, Anurag S Maskey, Esther Ryvkina, Michael Stonebraker, and Richard Tibbetts. 2004. Linear road: a stream data management benchmark. In *Proceedings of the Thirtieth international conference on Very large data bases-Volume 30*. 480–491.

[21] Paris Carbone, Stephan Ewen, Gyula Fóra, Seif Haridi, Stefan Richter, and Kostas Tzoumas. 2017. State Management in Apache Flink: Consistent Stateful Distributed Stream Processing. In *Proceedings of the International Conference on Very Large Data Bases (VLDB)*.

[22] Raul Castro Fernandez, Matteo Migliavacca, Evangelia Kalyvianaki, and Peter Pietzuch. 2013. Integrating Scale Out and Fault Tolerance in Stream Processing Using Operator State Management. In *Proceedings of the SIGMOD international conference on Management of Data*.

[23] Jun Lin Chen, Daniyal Liaqat, Moshe Gabel, and Eyal de Lara. 2022. Starlight: Fast Container Provisioning on the Edge and over the WAN. In *Proceedings of the USENIX Symposium on Networked Systems Design and Implementation (NSDI)*.

[24] Tathagata Das, Yuan Zhong, Ion Stoica, and Scott Shenker. 2014. Adaptive Stream Processing Using Dynamic Batch Sizing. In *Proceedings of the ACM Symposium on Cloud Computing (SoCC)*.

[25] Bonaventura Del Monte, Steffen Zeuch, Tilmann Rabl, and Volker Markl. 2020. Rhino: Efficient Management of Very Large Distributed State for Stream Processing Engines. In *Proceedings of the SIGMOD International Conference on Management of Data*.

[26] Rong Gu, Han Yin, Weichang Zhong, Chunfeng Yuan, and Yihua Huang. 2022. Meces: Latency-efficient Rescaling via Prioritized State Migration for Stateful Distributed Stream Processing Systems. In *Proceedings of the USENIX Annual Technical Conference (USENIX ATC)*.

[27] Vincenzo Gulisano, Ricardo Jimenez-Peris, Marta Patino-Martinez, Claudio Soriente, and Patrick Valduriez. 2012. Streamcloud: An Elastic and Scalable Data Streaming System. *IEEE Transactions on Parallel and Distributed Systems* 23, 12 (2012).

[28] Thomas Heinze, Zbigniew Jerzak, Gregor Hackenbroich, and Christof Fetzer. 2014. Latency-aware Elastic Scaling for Distributed Data Stream Processing Systems. In *Proceedings of the ACM International Conference on Distributed Event-Based Systems*.

[29] Moritz Hoffmann, Andrea Lattuada, Frank McSherry, Vasiliki Kalavri, John Liagouris, and Timothy Roscoe. 2019. Megaphone: Latency-conscious State Migration for Distributed Streaming Dataflows. In *Proceedings of the International Conference on Very Large Data Bases (VLDB)*.

[30] Shuai Hua, Manika Kapoor, and David C Anastasiu. 2018. Vehicle Tracking and Speed Estimation from Traffic Videos. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition Workshops*.

[31] Sanjeev Kulkarni, Nikunj Bhagat, Maosong Fu, Vikas Kedigehalli, Christopher Kellogg, Sailesh Mittal, Jignesh M Patel, Karthik Ramasamy, and Siddarth Taneja. 2015. Twitter Heron: Stream Processing at Scale. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*.

[32] Avinash Lakshman and Prashant Malik. 2010. Cassandra: a Decentralized Structured Storage System. *ACM SIGOPS Operating Systems Review* 44, 2 (2010).

[33] Pinchao Liu, Dilma Da Silva, and Liting Hu. 2021. DART: A Scalable and Adaptive Edge Stream Processing Engine. In *Proceedings of the USENIX Annual Technical Conference (USENIX ATC)*.

[34] Ivan Lujic, Vincenzo De Maio, Klaus Pollhammer, Ivan Bodrozic, Josip Lasic, and Ivona Brandic. 2021. Increasing Traffic Safety with Real-Time Edge Analytics and 5G. In *Proceedings of the International Workshop on Edge Systems, Analytics and Networking*.

[35] Luo Mai, Kai Zeng, Rahul Potharaju, Le Xu, Steve Suh, Shivaram Venkataraman, Paolo Costa, Terry Kim, Saravanan Muthukrishnan, Vamsi Kuppa, et al. 2018. Chi: A Scalable and Programmable Control Plane for Distributed Stream Processing Systems. In *Proceedings of the International Conference on Very Large Data Bases (VLDB)*.

[36] Yancan Mao, Yuan Huang, Runxin Tian, Xin Wang, and Richard TB Ma. 2021. Trisk: Task-Centric Data Stream Reconfiguration. In *Proceedings of the ACM Symposium on Cloud Computing (SoCC)*.

[37] Seyed Hossein Mortazavi, Mohammad Salehe, Bharath Balasubramanian, Eyal de Lara, and Shankaranarayanan PuzhavakathNarayanan. 2020. Sessionstore: A Session-aware Datastore for the Edge. In *Proceedings of the IEEE International Conference on Fog and Edge Computing (ICFEC)*.

[38] Sumanaruban Rajadurai, Jeffrey Bosboom, Weng-Fai Wong, and Saman Amarasinghe. 2018. Gloss: Seamless Live Reconfiguration and Reoptimization of Stream Programs. In *Proceedings of ASPLOS*.

[39] Brian Ramprasad, Pritish Mishra, Myles Thiessen, Hongkai Chen, Alexandre da Silva Veith, Moshe Gabel, Oana Balmau, Abelard Chow, and Eyal de Lara. 2022. Shepherd: Seamless Stream Processing on the Edge. In *Proceedings of the IEEE/ACM Symposium on Edge Computing (SEC)*.

[40] Henriette Röger and Ruben Mayer. 2019. A Comprehensive Survey on Parallelization and Elasticity in Stream Processing. *ACM Computing Surveys (CSUR)* 52, 2 (2019).

[41] Hooman Peiro Sajjad, Ken Danniswara, Ahmad Al-Shishtawy, and Vladimir Vlassov. 2016. Spanedge: Towards Unifying Stream Processing Over Central and Near-the-edge Data Centers. In *Proceedings of the IEEE/ACM Symposium on Edge Computing (SEC)*.

[42] Mehul A Shah, Joseph M Hellerstein, Sirish Chandrasekaran, and Michael J Franklin. 2003. Flux: An Adaptive Partitioning Operator for Continuous Query Systems. In *Proceedings of the International Conference on Data Engineering (ICDE)*.

[43] Quoc-Cuong To, Juan Soto, and Volker Markl. 2018. A Survey of State Management in Big Data Processing Systems. In *Proceedings of the International Conference on Very Large Data Bases (VLDB)*.

[44] Pete Tucker, Kristin Tufte, Vassilis Papadimos, and David Maier. 2008. *Nexmark–a Benchmark for Queries Over Data Streams.* Technical Report. Technical Report. Technical report, OGI School of Science & Engineering.

[45] Guozhang Wang, Lei Chen, Ayusman Dikshit, Jason Gustafson, Boyang Chen, Matthias J Sax, John Roesler, Sophie Blee-Goldman, Bruno Cadonna, Apurva Mehta, et al. 2021. Consistency and Completeness: Rethinking Distributed Stream Processing in Apache Kafka. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*.

[46] Junjue Wang, Ziqiang Feng, Zhuo Chen, Shilpa George, Mihir Bala, Padmanabhan Pillai, Shao-Wen Yang, and Mahadev Satyanarayanan. 2018. Bandwidth-efficient Live Video Analytics for Drones via Edge Computing. In *Proceedings of the IEEE/ACM Symposium on Edge Computing (SEC)*.

[47] Yingjun Wu and Kian-Lee Tan. 2015. ChronoStream: Elastic Stateful Stream Computation in the Cloud. In *Proceedings of the International Conference on Data Engineering (ICDE)*.

[48] Le Xu, Boyang Peng, and Indranil Gupta. 2016. Stela: Enabling Stream Processing Systems to Scale-in and Scale-out On-demand. In *Proceedings of the IEEE International Conference on Cloud Engineering (IC2E)*.

[49] Yifan Yu. 2016. Mobile Edge Computing Towards 5G: Vision, Recent Progress, and Open Challenges. *China Communications* 13, Supplement2 (2016).