Fugue: Online Elasticity for Distributed Stateful Stream Processing

Yuqiu Zhang*
qzhang@confluent.io
Confluent Inc.
Toronto, ON, Canada

Yunhao Mao
yunhao.mao@mail.utoronto.ca
University of Toronto
Toronto, ON, Canada

Hans-Arno Jacobsen
jacobsen@eecg.toronto.edu
University of Toronto
Toronto, ON, Canada

ABSTRACT

Stateful stream processing engines are critical for real-time analytics but lack efficient mechanisms for runtime elasticity. The dominant "stop-the-world" model, used by systems like Apache Flink, requires halting applications globally for a long time, while recent on-the-fly protocols introduce severe trade-offs: proactive approaches impose a continuous resource tax by constantly replicating state, and existing reactive solutions suffer from architectural complexity and external dependencies. This paper introduces Fugue, a novel, self-contained reactive protocol that provides seamless and resource-efficient elasticity. The core of Fugue is a two-phase design that combines a pre-emptive background state transfer with an atomic, lightweight barrier-based cutover. By moving the bulk of an operator's state off the critical path and unifying the final ownership transfer with the system's native exactly-once synchronization mechanism, Fugue guarantees correctness with minimal disruption and steady-state overhead. We implemented Fugue in Apache Flink and our evaluation on realistic benchmarks shows it reduces tail reconfiguration latency by up to 98.6% relative to native Flink while maintaining over 90% of peak throughput. Compared to reactive pull-based baselines, Fugue reduces end-to-end migration latency by up to 93.7%. Compared to proactive replication, it reaches comparable handover performance while avoiding continuous replication overhead. Together, these results demonstrate a strong combination of robustness, performance, and operational simplicity.

PVLDB Reference Format:
Yuqiu Zhang, Yunhao Mao, and Hans-Arno Jacobsen. Fugue: Online Elasticity for Distributed Stateful Stream Processing. PVLDB, 19(11): 3288 - 3301, 2026.
doi:10.14778/3836663.3836689

PVLDB Artifact Availability:
The source code, data, and/or other artifacts have been made available at https://github.com/MSRG/Fugue.

1 INTRODUCTION

The evolution of cloud computing has transformed data processing into a utility, where resources can be provisioned and released on demand [25, 48, 56]. This paradigm is a natural fit for modern real-time analytics, which are dominated by long-running, stateful stream processing applications for use cases ranging from financial fraud detection to IoT monitoring [11, 20, 26, 41, 45]. These applications, often expressed as dataflow graphs of operators like joins and aggregations [2, 9], must maintain distributed state while providing continuous, low-latency processing [1, 12, 15, 28, 34]. The dynamic nature of the cloud, with its fluctuating workloads and resource availability, creates a fundamental requirement for these systems to be *elastic*: the ability to adapt their parallelism at runtime to meet performance goals while minimizing operational cost [27, 32].

However, providing true, seamless elasticity for stateful stream processing remains a major challenge, as state must be consistently redistributed alongside the computational logic [14]. The dominant approach employed by mature systems like Apache Flink [10] and Spark Streaming [9, 54] is a disruptive "stop-the-world" model. To change parallelism, the entire query is paused, a global, durable snapshot of its state is taken, the state is repartitioned and shuffled across the network to a new set of workers, and the query is finally restarted with the new configuration. This process introduces a significant service blackout, which is prohibitively expensive for large-scale streaming applications.

Recognizing this limitation, recent research has explored on-the-fly reconfiguration protocols [13, 23, 29, 31, 46, 47, 53], which aim to minimize disruption during scaling events. These approaches can be broadly categorized into two main strategies. *Proactive approaches*, exemplified by Rhino [23], achieve fast state handovers by continuously replicating operator state to standby nodes. However, this design trades low state migration latency for a continuous and significant steady-state overhead in storage and network bandwidth. In contrast, *reactive approaches* incur no overhead during normal operation and only initiate state transfer when a scaling decision is made. While more resource-efficient, existing reactive solutions often introduce their own complexities. Systems like Meces [29] employ a pull-based "fetch-on-demand" model that, while prioritizing hot keys, introduces significant complexity due to its reliance on external systems like message queues and key-value stores. A clear gap remains for a simple but self-contained reactive protocol that can serve as a practical foundation for seamless elasticity.

In this paper, we propose Fugue, a system that provides online, fine-grained elasticity for stateful stream processing with minimal disruption. The core of Fugue is a two-phase mechanism that decouples the expensive state transfer from the atomic cutover of processing responsibility. First, upon a scaling decision, Fugue performs a **pre-emptive background copy**, where the old operator instance begins pushing the relevant state partition to the new

*Work done while at the University of Toronto.

Proceedings of the VLDB Endowment, Vol. 19, No. 11 ISSN 2150-8097.
doi:10.14778/3836663.3836689

instance in the background, using an iterative strategy to minimize the final set of state changes that accumulate during the copy process. Second, the final ownership transfer is orchestrated by a **lightweight alignment barrier** that propagates through the data stream using the system's native exactly-once synchronization mechanism. This barrier establishes a consistent point-in-time for the cutover across all distributed participants. Crucially, this design provides the strong ordering guarantees of a global synchronization signal without the prohibitive performance cost of a full checkpoint, as non-participating operators incur near-zero overhead. Fugue's push-based, self-contained architecture avoids the operational burden of external dependencies and the network inefficiency of excessive remote procedure calls (RPCs), offering a robust and practical solution to online state migration.

We have implemented Fugue as an extension to Apache Flink [5], a widely-used open-source stream processing framework. Our evaluation on a variety of benchmarks shows that Fugue can achieve state migration with minimal disruption and operational overhead. Our results demonstrate that Fugue reduces tail reconfiguration latency by up to 98.6% relative to Flink's native stop-the-world mechanism, while maintaining over 90% of its peak processing throughput during the migration. Against state-of-the-art protocols, Fugue's efficient, self-contained design reduces end-to-end migration latency by up to 93.7% compared to the latest reactive approaches, while achieving handover behavior comparable to proactive replication without paying its continuous steady-state overhead.

The main contributions of this paper are:

(1) We identify the challenges of providing seamless elasticity for stateful stream processing and analyze the limitations of existing approaches.
(2) We design Fugue, a novel two-phase protocol for online state migration that combines pre-emptive background copy with a lightweight, unified barrier for atomic ownership transfer, ensuring correctness with minimal service disruption.
(3) We implement Fugue on top of Apache Flink as a self-contained migration extension which requires no external state-transfer system and only targeted changes to the underlying runtime.
(4) We conduct extensive experimental evaluations against state-of-the-art proactive and reactive approaches, demonstrating Fugue's combination of low-overhead steady-state operation and fast, stable performance during reconfigurations.

The remainder of this paper is organized as follows. We begin with background and motivation in Section 2. We detail Fugue's core protocols in Section 3. We describe our system architecture and implementation in Section 4 and present a thorough evaluation in Section 5. Finally, we discuss related work in Section 6 and conclude in Section 7.

2 BACKGROUND AND MOTIVATION

In this section, we first establish the foundational concepts of modern stateful stream processing, then motivate the need for a new elasticity protocol by analyzing the limitations of existing approaches.

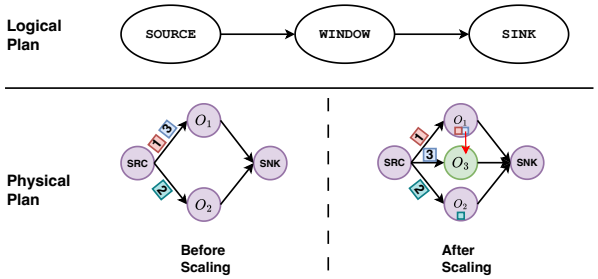


Figure 1: Stateful operator scaling in stream processing.

2.1 Stateful Stream Processing

Modern distributed Stream Processing Engines (SPEs), such as Apache Flink [5] and Spark Streaming [8], model computations as a Directed Acyclic Graph (DAG) of logical operators. These operators encapsulate application logic such as transformations (`map`), aggregations (`window`), or joins. To achieve parallelism, the SPE instantiates each logical operator into one or more parallel *operator instances*, which are the smallest units of computation typically executed as threads within worker processes distributed across a compute cluster. Data flows from *upstream* operator instances to *downstream* operator instances in the form of keyed records. A data record r is a tuple (k, v), where k is the key and v is the payload value.

A key challenge in this model is the management of *state*. Many essential operators are *stateful*. A stateful operator instance must maintain and consult internal state to process incoming records. For example, a `GROUP BY` aggregation operator maintains the current aggregate (e.g., a running count) for each key it is responsible for, and a `JOIN` operator must store tuples from one stream while waiting for matching tuples from another within a time window. This intermediate data, known as *operator state*, is partitioned across the parallel instances based on the key in the data records. An upstream operator instance uses a partitioning function (e.g., hashing) to route a record with key k to the specific downstream instance responsible for managing the state for k. This co-partitioning of data and state is fundamental to enabling scalable, distributed state management.

To guarantee correctness and state consistency in the presence of failures, modern SPEs rely on a distributed snapshot mechanism, often a variant of the Chandy-Lamport algorithm [10, 16]. Periodically, special control messages called *checkpoint barriers* are injected into the data streams. These barriers flow through the DAG alongside regular data records. When an operator instance receives a barrier from all its inputs, it triggers a snapshot of its local state to durable storage (e.g., HDFS [50] or S3 [3]). Once all operator instances have completed their snapshots for a given checkpoint, a globally consistent snapshot of the entire application's state is created and can be used for failure recovery to ensure exactly-once processing of data records.

2.2 State Management Challenges in Elasticity

The need for elasticity arises from the dynamic nature of both workloads and cloud infrastructure. A sudden spike in workload traffic

may require *scaling out* an operator (adding more parallel instances) to handle the increased load, while a period of low activity is an opportunity for *scaling in* (removing existing instances) to reduce operational costs. Scaling a stateless operator (e.g., a filter or map) is trivial: one can simply add or remove instances and update the routing tables of its upstream operators.

However, scaling a stateful operator is more complex and requires careful management of the operator's state. When an operator's parallelism changes, the mapping of keys to instances is altered. Consequently, the distributed state must be repartitioned and physically migrated across the network to the new responsible instances. For example, as shown in Figure 1, when scaling out a `window` operator from two to three instances, keys in partition P_3 that were previously handled by instance O_1 might now be reassigned to the new instance O_3. The state associated with P_3 must be migrated from O_1 to O_3 (the red arrow in Figure 1) to ensure that future records for those keys are processed correctly. This state migration process, ensuring data is moved efficiently and consistently without violating the application's exactly-once processing semantics, is the primary challenge for any elasticity mechanism.

2.3 Enabling Online Elasticity

The most common approach to stateful elasticity is a disruptive "stop-the-world" model, employed by industrial-strength SPEs like Apache Flink [5]. This approach leverages the existing checkpointing mechanism described in Section 2.1 to naturally ensure consistency. To change the parallelism of an operator, the entire running query is paused to take a global snapshot of all operator state. The system then re-partitions this state according to the new desired parallelism and shuffles it over the network to the new set of operator instances. Finally, the query is resumed with the new configuration only after all state redistribution is complete. While this method guarantees correctness, the entire pipeline is halted during the scaling process, which can lead to serious performance degradation [31].

To avoid a full blackout, recent research has proposed mechanisms that enable *online elasticity* (or, *on-the-fly reconfiguration*) [13, 23, 29, 31, 46, 47, 53], allowing the system to migrate state with minimal service downtime during scaling events. These protocols fall into two main families: *proactive* and *reactive*.

2.3.1 Proactive Approaches. The core idea of proactive approaches is to add extra mechanisms to non-scaling (steady-state) processing to smooth out the latency spikes during scaling events. For example, Megaphone [31] introduces a routing indirection layer that replaces a single stateful operator with a two-operator structure, with the first operator acting as a router. This allows the large state transfer during scaling events to be decomposed into many smaller, less disruptive migrations, but incurs additional overhead during normal operation, as every record must pass through the new routing layer. Rhino [23] takes a different approach by requiring the system to continuously and proactively maintain hot replicas of an operator's state on other nodes using a continuous replication protocol. When a scale-out is needed, the system simply "hands over" processing to a new instance on a node that already has a recent copy of the state, effectively addressing latency spikes but at the cost of significant steady-state overhead in storage and network bandwidth.

2.3.2 Reactive Approaches. Reactive approaches, in contrast, incur no overhead during normal operation and only initiate state transfer when a scaling decision is made. However, existing reactive solutions often introduce their own challenges. Partial pause-and-resume approaches like Chi [38] pause computation only for the affected subgraph of operators during scaling, reducing disruption but nevertheless causing a blackout for that portion, which can still be costly if it involves critical processing paths. Pull-based systems like Meces [29] require the new operator instance to fetch each piece of state on demand, introducing additional architectural complexity via external message queues and key-value stores.

These observations highlight a need for a simple, self-contained reactive protocol that provides seamless elasticity without the continuous resource tax of proactive approaches or complex runtime modifications and external systems required by existing reactive designs. This is the gap that Fugue is designed to fill.

3 FUGUE DESIGN

At the heart of the Fugue system is a novel protocol for the online migration of stateful operator partitions. This section details the protocol's design, which is guided by three primary goals. **Correctness** ensures that the migration preserves the application's exactly-once processing semantics without data loss or duplication. **Low Overhead** keeps the migration process from substantially affecting the end-to-end latency and throughput of the running query. **Generality** allows the protocol to serve as a single primitive for various elasticity scenarios, including scale-out, scale-in, and dynamic load rebalancing.

To achieve these goals, Fugue decouples the expensive bulk state transfer from the time-sensitive ownership transfer. This is realized through a two-phase process: a **pre-emptive background copy** phase (Section 3.1), where the bulk of the state is moved off the critical path, followed by an **atomic cutover** phase, which uses lightweight, in-band barriers to finalize the migration and transfer ownership with minimal processing disruption.

3.1 Phase 1: Pre-emptive Background Copy

The migration process begins when the central Migration Coordinator makes a scaling decision to migrate a partition P_k from an origin operator instance O_{old} to a target instance O_{new}. The objective is to transfer the vast majority of P_k's state without impacting the ongoing processing at O_{old}. A naive single-pass copy of the live state is inefficient: if the state is large or the update rate is high, the amount of changed state (the *delta*) that accumulates during the copy can itself be very large, leading to a long final pause. To address this, Fugue employs an iterative pre-copy strategy, inspired by live VM migration techniques [19]. The core idea is to perform multiple rounds of state copying in the background, where each round only transfers the delta since the last round. The system only needs to pause the migrating partition briefly at the end to transfer the small final delta. Specifically, the process is shown in Figure 2, which unfolds as follows:

(1) **Initialization**: Through direct, out-of-band RPCs (①), the coordinator notifies all relevant components of the impending migration: the source instance O_{old}, the target instance O_{new}, and all upstream operator instances that route data

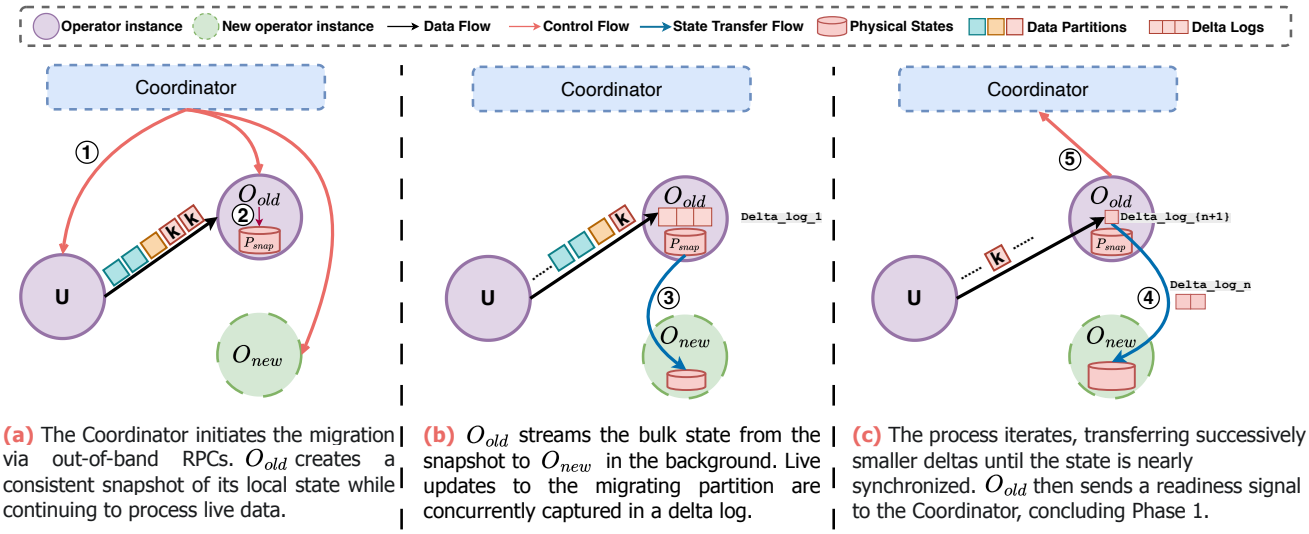


Figure 2: Illustration of the pre-emptive background copy phase in Fugue. A single upstream operator instance is assumed and only the migration-relevant operator instances are shown for the sake of brevity.

to this operator stage. Upon receiving the notification, O_{old} continues processing incoming records for partition P_k while preparing to send its state to O_{new}.

(2) **Initial Snapshot and Bulk Transfer**: To bootstrap the migration, O_{old} first requests a consistent and point-in-time snapshot of its state backend for partition P_k (②). This is a low-overhead operation in modern state backends like RocksDB [39], and does not require pausing the processing of live data. O_{old} then begins to stream the data from this immutable snapshot over the network to O_{new} (③). This bulk transfer is rate-limited and treated as a low-priority background task to avoid interfering with the operator's primary processing traffic. While the bulk transfer is in flight, O_{old} continues to process new records for P_k, and any resulting state modifications are recorded in a memory-resident delta log, `Delta_Log_1`.

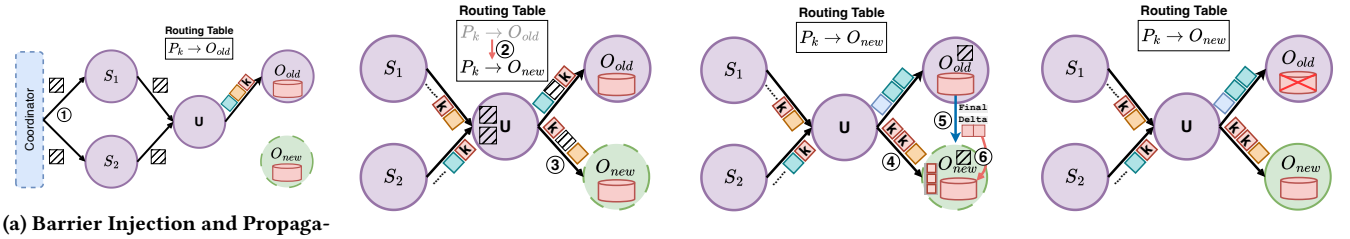
(3) **Iterative Delta Transfer**: Once the initial bulk transfer is complete, the migration enters a series of iterative "pre-copy" rounds. In Round 2, O_{old} transfers the first delta log, `Delta_Log_1`, to O_{new}. Because it is typically much smaller than the original bulk state, this transfer is much faster. During this time, any new state updates for P_k are recorded in a new delta log, `Delta_Log_2`. This process repeats with each round transferring the delta from the previous round (④). The size of the delta logs decreases as the amount of data to be sent shrinks with each round.

(4) **Convergence**: The iterative process continues until the size of the most recent delta log falls below a predefined threshold (e.g., a few megabytes). At this point, O_{new} has a nearly up-to-date copy of P_k's state, and the amount of state remaining to be transferred is minimal. O_{old} signals its readiness to the Coordinator (⑤), and the system is ready to proceed to the atomic cutover phase. The threshold is a tunable performance policy which one can derive from the business requirements of the application. A smaller threshold spends more time in pre-copy to reduce cutover latency, while a larger threshold shortens total migration time at the cost of more work in the bounded cutover interval.

The central assumption of the iterative pre-copy strategy is that it will converge, i.e., the size of the delta log generated will be smaller than the size of the delta log transferred in that round. This convergence is predicated on the condition that the available network bandwidth rate for the transfer (`R_net`) is greater than the rate at which the operator instance is modifying the state of the migrating partition (`R_dirty`). This condition is commonly met in real-world scenarios, as modern data center networks are usually equipped with high-speed links (e.g., 10Gbps+) [4, 49, 55] that can easily outpace the rate at which a single operator instance (typically single-threaded [6]) can dirty state, which is often limited by CPU and I/O constraints. In the rare case where `R_dirty` exceeds `R_net`, Fugue implements a safeguard: if the size of the delta log fails to decrease, the coordinator can simply abort the migration, in which case it sends a cancellation signal to O_{old} and O_{new}. O_{old} then ceases the migration process and discards its delta logs, while O_{new} deletes any partially received state. We will further discuss the performance impacts of `R_net` and `R_dirty` in Section 5.4.

3.2 Phase 2: Atomic Cutover with Lightweight Migration Barriers

Once the iterative pre-copy has converged, the next step is to atomically and consistently transfer ownership of the partition from O_{old} to O_{new}. To achieve this, Fugue employs a novel use of *lightweight migration barriers*, which are special control messages injected into the data streams. These barriers serve two main purposes: (1) they synchronize the state of the operator instances involved in the migration, and (2) they ensure that all in-flight records are fully



(a) Barrier Injection and Propagation.

(b) Atomic Routing Switch.

(c) Finalization and Activation.

(d) Post-Migration Steady State.

Figure 3: Illustration of the atomic cutover phase in Fugue. ▨ indicates the migration barrier. A single upstream operator instance is assumed and only the migration-relevant operator instances are shown for the sake of brevity.

processed by either the old instance or the new instance, but not both. This process reuses the semantics of checkpoint barriers in modern stream processing systems (Section 2.1) that are already designed to propagate through the dataflow graph and establish consistent cut points, but is lightweight in that it avoids the overhead of a full, durable checkpoint. The cutover process is shown in Figure 3 and unfolds as follows:

(1) **Barrier Injection**: Upon receiving the readiness signal from O_{old} in the previous phase, the Coordinator injects a lightweight migration barrier into all sources of the dataflow graph (①). It carries a migration plan as metadata, specifying the set of partitions to be migrated (e.g., `[`P_k`, `O_{old} `-> `O_{new}`]`). The barrier then propagates through the DAG in-band with the data records to establish a totally ordered boundary in the distributed dataflow.

(2) **At upstream operators of the migration**: When an upstream operator instance receives the barriers, it first aligns them across its inputs (i.e., waits until it has received the barrier on all its input streams). It then inspects the migration plan within the barrier. Since its downstream target for partition P_k is changing, it atomically updates its routing table to redirect any new records for P_k to O_{new} (②). This update is logically bound to the barrier: all records for P_k that arrive before the barrier are sent to O_{old}, and all records sent after the barrier are rerouted to O_{new}. The operator then forwards the barrier downstream (③).

(3) **At non-participating operators**: Any operator instance not involved in the migration simply aligns the barrier and forwards it. It performs no state snapshotting and no I/O. For these operators, the barrier is a near-zero-cost passthrough message that serves only to maintain the global stream order.

(4) **At the new instance** O_{new}: O_{new} has already received the bulk of P_k's state during the pre-copy phase. As O_{new} starts to receive *post-barrier* records for P_k, it buffers these records without processing them (④). When O_{new} has received the barrier from *all* its input streams, it knows that from this moment on, it is safe to apply the final delta and subsequently start consuming its input buffer. It now waits for one final message from O_{old}.

(5) **At the old instance** O_{old}: O_{old} continues processing pre-barrier records for P_k as normal until it receives the barrier on all its input streams. At this point, O_{old} knows that it has processed all records for P_k that were sent before the barrier and will receive no further pre-barrier records. It now captures the final state delta (`Delta_Log_Final`) that has accumulated since the last iterative copy round, sends it to O_{new} (⑤) and sends a completion acknowledgement to the Coordinator, then finally deletes its local state for P_k to fully relinquish its ownership.

(6) **Finalization at** O_{new}: Upon receiving `Delta_Log_Final` from O_{old}, O_{new} applies this final delta to the state it received during the background copy (⑥). Its state for P_k is now complete and consistent up to the point of the barrier. It now begins processing the post-barrier records it has buffered and any new incoming records for P_k, and sends a completion acknowledgement to the Coordinator. At this point, the migration is fully complete, and O_{new} is the sole owner of partition P_k.

3.3 Correctness Guarantees

The primary correctness requirement for any state migration protocol is to preserve the application's exactly-once processing semantics. In the context of a stateful operator, this means that for any given record, the state modifications must appear as if they occurred in a single and serializable history without data loss or duplication. The Fugue protocol provides this guarantee through the composition of its state migration and synchronization mechanisms. We argue for correctness by demonstrating two key properties: *record integrity* and *state consistency*.

Record Integrity. In Fugue, the routing table update at the upstream operator is atomic with respect to the migration barrier. For any given record r and its partition P_k, if r appears in the stream *before* the barrier, it is deterministically routed to O_{old} and processed against the state at O_{old}. If r appears in the stream *after* the barrier, it is deterministically routed to O_{new}. Because O_{new} only begins processing records for P_k after it has received and applied the complete state from O_{old}, r is guaranteed to be processed against the fully migrated and correct state. There is no time window during which a record could be lost or processed by both instances.

State Consistency. The protocol ensures that the state handed over to the new operator instance is a complete and consistent representation of the migrated partition P_k at the logical time of the cutover. To show this, let t_1 be the timestamp when O_{old} takes the

initial snapshot, t_2 be the timestamp corresponding to the propagation of the migration barrier. Further, let $S(P_k, t)$ denote the true logical state of partition P_k at time t, and let $S_{old}(P_k)$ and $S_{new}(P_k)$ be the physical state for partition P_k stored at instances O_{old} and O_{new}, respectively. The objective is thus to prove the following claim:

Proposition 3.1. *$S_{new}(P_k) = S(P_k, t_2)$ upon completion of the migration.*

Proof. First, at t_1, O_{old} takes a snapshot of its state, capturing $S_{old}(P_k)$ as a consistent representation of $S(P_k, t_1)$. This immutable snapshot data is then streamed in the background to O_{new}. As the source data is static and the network transfer is reliable, O_{new} is guaranteed to possess an identical copy of this snapshot. Therefore, prior to the commencement of Phase 2, the physical state at O_{new} for partition P_k is: $S_{new}(P_k) = S(P_k, t_1)$. Then, by protocol definition, O_{old} continues to process all records for partition P_k that arrive before it has aligned the barrier from all of its inputs. The barrier's in-band propagation ensures that any record $R(P_k, t)$ with $t \leq t_2$ will be fully processed by O_{old} before the barrier alignment is complete, and all state transformations induced by these records are reflected in its local physical state $S_{old}(P_k)$. The protocol requires O_{old} to capture these deltas and transfer them to O_{new} immediately *after* it has achieved barrier alignment. This timing ensures the final delta sent by O_{old} includes the effects of every single pre-barrier record that arrived after the initial snapshot was taken. Therefore, the final delta transfer, combined with the deltas from the iterative pre-copy rounds, constitutes $\Delta(t_1, t_2)$, which denotes the sequence of state transformations resulting from processing all records $R(P_k, t)$ where $t_1 < t \leq t_2$. Finally, since O_{new} receives the base state $S(P_k, t_1)$ during Phase 1 and applies the complete state delta $\Delta(t_1, t_2)$ during Phase 2, the resulting physical state at O_{new} is now $S_{new}(P_k) = S(P_k, t_1) + \Delta(t_1, t_2) = S(P_k, t_2)$, concluding our proof. At this point, O_{new} holds the correct state for partition P_k, and can then proceed to process all post-barrier records $R(P_k, t)$ for all $t > t_2$.

□

3.4 Discussion

In previous sections, we have detailed the design of the Fugue protocol exemplified by a scale-out migration scenario. Here, we discuss how Fugue, designed as a general primitive, can naturally handle other elasticity scenarios. We will also discuss fundamental trade-offs that drive specific design choices within Fugue.

Handling Scale-In and Load Rebalancing. The scale-in operation involves removing an operator instance and consolidating its state onto one or more remaining instances. When the Coordinator decides to remove an instance O_d, it first determines the new owners for each partition currently hosted by O_d. It then initiates a series of parallel migration operations, one for each partition, targeting the designated remaining instances (e.g., `[`P_i`, `O_d` -> `O_i`]`, `[`P_j`, `O_d` -> `O_j`]`, etc.). The background state transfer allows the remaining instances to remain fully productive on their existing workload, with the state ingestion consuming only background resources. The O_d instance is only terminated after the Coordinator has received acknowledgement that all of its partitions have been successfully migrated to their new owners. The protocol is also well-suited for reactively addressing workload skew, or "hotspot" partitions. The Coordinator may detect that an instance, O_1, is overloaded due to high traffic for a specific partition, P_{hot}. To balance load, it can simply trigger a migration of P_{hot} from O_1 to a less loaded instance O_2, with $O_{old} = O_1$ and $O_{new} = O_2$. The migration proceeds online with minimal disruption to the processing of all other partitions.

Out-of-band RPCs vs. In-band Barriers. Fugue adopts a separation of control-plane and data-plane delivery for migration control messages. In Phase 1, Fugue employs out-of-band RPCs from the Coordinator to directly notify all relevant operator instances (the *preparation*). In Phase 2, it employs in-band barriers to orchestrate the atomic cutover (the *execution*). This is essential because the cutover requires a causally consistent synchronization point within the data stream itself, a guarantee that only an in-band mechanism can provide. This hybrid design stands in contrast to prior work that mostly uses in-band messages for both preparation and execution [23, 29, 38, 46]. While these pure data-plane approaches simplify the control flow integration with existing SPE runtimes, they can also lead to increased latency and bottlenecks: relevant operators are not notified of the migration until the control message has propagated through the entire preceding dataflow, which can take a long time in large and complex topologies. Fugue's decoupled strategy thus optimizes for both speed and correctness, combining the low-latency advantage of a direct control channel for preparation with the strong consistency guarantees of an in-band signal for execution.

Sequential vs. Parallel Phasing. Fugue follows a sequential ordering of its two phases, where the barrier-based cutover (Phase 2) is initiated only after the background copy (Phase 1) has converged. A seemingly faster alternative would be to parallelize these phases by injecting the barrier at the same time the background copy begins to reduce overall latency by overlapping state transfer and barrier propagation. However, this parallel approach can introduce a severe risk of system instability. In such a scenario, the barrier could propagate quickly and reroute data to the new instance long before the state transfer is complete. This would force the new instance to buffer a potentially enormous volume of incoming data, leading to memory exhaustion or cascading backpressure that could stall the entire query. By adopting a sequential model, Fugue makes a deliberate trade-off: it prioritizes predictable and bounded resource consumption and the preservation of low-latency service during migration over a marginal, best-case reduction in total migration time.

4 IMPLEMENTATION

To validate our protocol and demonstrate its practicality, we have implemented Fugue as a non-invasive extension to Apache Flink [5]. Our implementation respects Flink's core principles, such as its master-worker design and its distributed snapshotting mechanism, while augmenting them to support our online migration protocol. This section details the components of the Fugue architecture and discusses key implementation challenges.

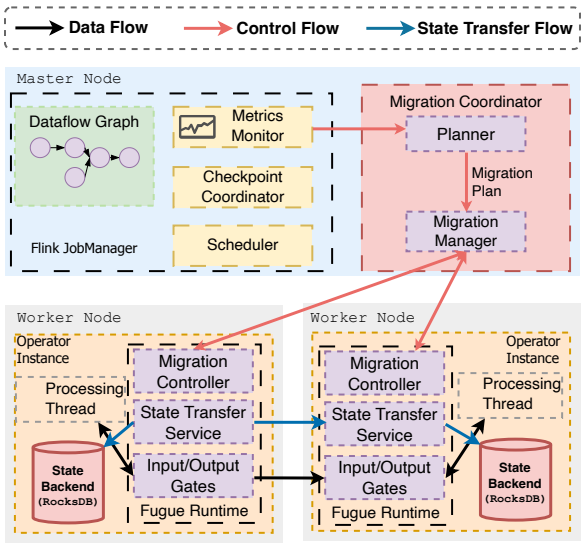


Figure 4: Architecture of the Fugue system.

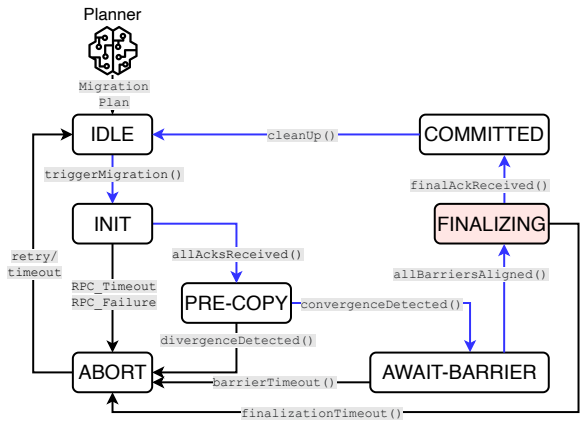


Figure 5: State machine transitions at the Migration Manager.

4.1 System Architecture

As illustrated in Figure 4, the Fugue architecture consists of two primary components: a central Migration Coordinator that extends Flink's `JobManager`, and a set of Fugue Runtimes that augment the operator instances on Flink's `TaskManagers`.

4.1.1 The Migration Coordinator. The Migration Coordinator is a service integrated into Flink's `JobManager`, serving as the master orchestrator for all elasticity operations. It is responsible for observing the state of the system, deciding when and what to migrate, and driving the migration protocol through its state machine. It comprises two main sub-components: the *Elasticity Planner* and the *Migration Manager*.

Elasticity Planner. This component is responsible for the *policy* of elasticity, deciding when to scale and what partitions to migrate. Since our work focuses on the elasticity mechanism, we implement a pluggable policy module that provides a threshold-based policy by default, and the design of advanced elasticity policies is an orthogonal direction worth exploring (Section 6). Regarding what to migrate, the planner employs a consistent hashing scheme to minimize data movement [33]. The output of the planner is a concrete migration plan (e.g., `[`P_k`,` O_{old} `->` O_{new}`]`) that is then passed to the Migration Manager.

Migration Manager. This component is the core stateful engine that executes the migration plan. It acts as the master of the Fugue protocol, overseeing the state transitions by maintaining a state machine of each migration, as shown in Figure 5. It communicates with the relevant `TaskManagers` using Flink's existing RPC framework to initiate Phase 1, and injects the migration barrier into the data stream by sending an RPC to the `StreamSource` operators of the DAG in Phase 2. The Migration Manager also handles failure scenarios by implementing retry mechanisms and can abort migrations if necessary. Finally, it tracks the progress of the barrier and the finalization acknowledgements to safely commit or abort the migration. Throughout migration, processing of the migrating partition P_k is paused only in `Finalizing`, where O_{new} buffers post-barrier records while waiting for the final delta from O_{old} and applies it before resuming processing. All other stages are non-blocking for P_k and the rest of the system.

4.1.2 The Fugue Runtime. The Fugue Runtime is a set of components co-located with each stateful operator instance on the `TaskManagers`. It augments the instance's normal data processing logic with the capabilities required to participate in the migration protocol.

Migration Controller. Each operator instance is equipped with a Migration Controller, which acts as its local agent for the migration protocol. It receives and handles the out-of-band RPCs from the Migration Manager. Upon receiving a `prepareMigrationSource` command, it initiates the background copy process. If it receives a `prepareMigrationTarget` command, it prepares to receive state. The controller is responsible for managing the state of the local instance and coordinating with the other runtime components.

State Transfer Service. This service, managed by the Migration Controller, is responsible for the physical movement of state data. When acting as a source, it interacts with the local State Backend to create a consistent snapshot and then serializes and sends this state over a dedicated TCP connection to the target instance. When acting as a target, it receives, deserializes, and writes the incoming state into the target's local State Backend.

Modified Input/Output Gates. To handle the in-band migration barriers, we have modified Flink's network buffer and serialization stack. The `StreamRecord` serializer is extended to recognize the `MigrationBarrier` type. When an upstream operator's output buffer encounters a barrier, it ensures the barrier is broadcast to all downstream channels. The operator's input gate is modified to intercept incoming barriers and manage the buffering of post-barrier records for partitions that are in the process of being migrated to the local task. This component is responsible for blocking the processing of these records until the Migration Controller signals that the final state delta has been received and applied.

4.2 Implementation Details and Challenges

State Management and Transfer. The backend *State Store* in our implementation is based on RocksDB [39], a high-performance, embedded key-value store optimized for fast storage. To initiate the background copy, the State Transfer Service on O_{old} uses RocksDB's native `Checkpoint` feature. This allows us to create a consistent, point-in-time snapshot of the required key-groups representing the partition without pausing the live operator. The primary challenge is to transfer a live RocksDB instance's state efficiently, where a naive approach of serializing every key-value pair would be prohibitively slow. Instead, we leverage RocksDB's file-based structure. Our State Transfer Service operates at the level of SST (Sorted String Table) files. The initial background copy involves creating a RocksDB checkpoint on the source TaskManager's local disk and then transferring the resulting SST files over the network to a temporary directory on the destination TaskManager. The target instance receives these SST files and uses RocksDB's `IngestExternalFile` API to load them directly into its own RocksDB instance. This is a highly optimized bulk-loading mechanism that bypasses the slower write path (through memtable and WAL), which significantly speeds up the state ingestion process. The iterative deltas, which are smaller, are transferred as key-value pairs and ingested through the normal write path.

Barrier Handling and Multiplexing. A critical goal is to ensure that the migration barrier does not cause head-of-line blocking for non-migrating partitions. For this purpose, our input gate extends Flink's native barrier handling logic: When a migration barrier arrives on an input channel, the gate stops processing data only from that channel. It continues to process data from other, unaligned channels. When the barrier is aligned, it triggers the migration logic. Crucially, once the migration logic is complete, the gate immediately unblocks the processing of post-barrier data for all non-migrating partitions. Only the data for the specific, newly-arriving partitions remains buffered until the final delta is applied. This ensures that the migration of one partition has a minimal performance impact on the processing of all other partitions handled by the same operator instance. We implemented our lightweight barrier by extending Flink's `CheckpointBarrier` class with a new subtype and a payload containing the migration plan, allowing us to reuse the entire message propagation and serialization mechanisms.

Concurrent Migration Management. The Migration Manager in the `JobManager` must be able to orchestrate multiple potentially overlapping migrations safely. To manage this complexity, the Migration Manager maintains a map of all ongoing migrations, each with its own state machine instance (Section 4.1.1). A global lock ensures that the generation of migration plans is serialized. For instance, we do not attempt to schedule two different migrations for the same partition simultaneously. However, the execution of migrations for different, non-overlapping partitions can proceed in parallel. Our unified barrier mechanism handles this naturally: a single migration barrier can carry a list of independent migration plans. When an operator receives this barrier, it simply iterates through the list and performs all actions relevant to it, allowing for the atomic, concurrent cutover of many partitions at the same time.

Fault Tolerance and Recovery. Fugue achieves fault tolerance and recovery by integrating the migration protocol with Flink's native distributed snapshotting mechanism. When no migration is in progress, Fugue's components are passive and the system relies on Flink's standard checkpointing and recovery mechanism. If a failure occurs while a migration is in flight, Fugue follows a conservative rule: the in-flight migration is atomically aborted, partially received target state is discarded, and the job recovers from the latest durable checkpoint whose topology has already committed. The target is not allowed to become the authoritative owner after an old-owner crash unless the final delta has been received, applied, and committed. This policy may redo migration work after a crash, but it avoids recovering into a partially migrated intermediate state. Specifically, the protocol distinguishes between two main failure types:

1) Transient Failures. These include lost RPCs or temporary unresponsiveness of a TaskManager during the out-of-band communication of Phase 1. As detailed in Section 4.1.1, the Migration Manager handles these with a configurable retry-with-timeout policy. If all retries fail, the migration is considered to have failed pre-emptively. The Coordinator issues `cancelMigration` commands to roll back any preparations and aborts the operation. The query continues to run in its stable, pre-migration configuration.

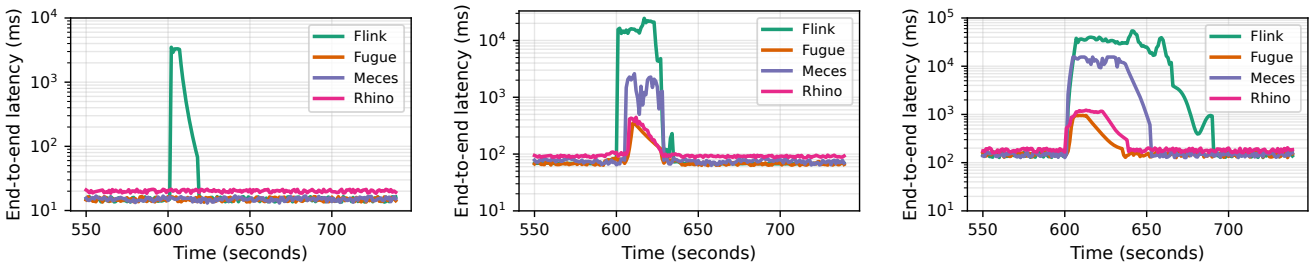
2) Task or Machine Failures. TaskManager crashes or task exceptions are detected by Flink's native failure detection mechanism, which triggers Flink's standard global recovery procedure. The `JobManager`, including the Migration Coordinator, is first notified of the global restart. The Coordinator then immediately transitions every in-flight migration to the `ABORT` state. Finally, the Flink runtime redeploys the query and instructs all operator instances to restore their state from the latest committed, globally consistent checkpoint. Because a restored checkpoint contains only a committed ownership configuration, the system recovers to a globally consistent state rather than to an ambiguous partial handover.

5 EVALUATION

5.1 Experimental Setup

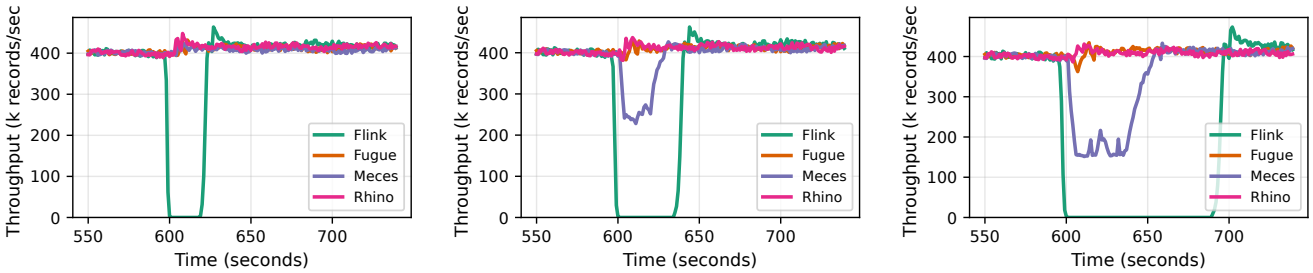
Testbed Setup. Our testbed cluster is composed of five physical nodes, each equipped with two 24-core AMD EPYC 9254 CPUs (2.90GHz) and 768 GB of RAM. The nodes run Ubuntu 20.04.5 LTS with Linux kernel v5.4 and are interconnected via a 10 Gbps Ethernet backbone. We deploy Apache Flink version 1.18 as the underlying stream processing engine, with RocksDB as the state backend. Apache Kafka 3.8.1 is used as the message broker for input and output streams. Each node is configured to host 5 task slots, resulting in a maximum parallelism of 25 per operator. Following related work and best practices [23, 29], we set the total number of key-groups (partitions of state) to 128 and Flink's checkpointing interval to 60 seconds, unless otherwise specified.

Workloads. We evaluate Fugue primarily using the NEXMark benchmarking suite [52], an online auction system widely adopted by stream processing research that simulates a variety of event streams, including auctions, bids, and users. Specifically, we focus on three representative queries from NEXMark. **Query 1** is a currency-conversion query that involves only the stateless map operator and is used to measure the baseline resource overhead of different systems without the complexity of state. **Query 5** is a sliding window count of bids per auction that involves a small



(a) Query 1: Stateless Map (b) Query 5: Sliding Window Count (c) Query 8: Tumbling Window Join

Figure 6: End-to-end latency during scaling events across three representative queries from NEXMark.



(a) Query 1: Stateless Map (b) Query 5: Sliding Window Count (c) Query 8: Tumbling Window Join

Figure 7: Throughput during scaling events across three representative queries from NEXMark.

amount of state. **Query 8** is a tumbling window join that finds new users who have registered and created auctions over a time window, which involves larger state and complex stateful operations. For sensitivity analysis, we also implemented a microbenchmark of a simple three-stage query (`Source` → `Keyed Window` → `Sink`) in which we can easily tune various parameters and study their impact on performance. The workloads are generated using the Kafka producer with a default rate of 400k events/sec to controllably saturate our testbed.

Metrics. Unless otherwise specified, latency is measured using in-band markers from Kafka production to final-sink emission. The open-loop Kafka producer inserts one lightweight marker after every 1,000 data records. Each marker carries the producer timestamp, follows the same Kafka topic, source reader, operator queues, and sink path as data records, and is excluded from state updates. Recovery time is counted only after both p99 latency and Kafka consumer lag return to near their pre-migration baselines, so broker-side buffering cannot hide a migration backlog.

Comparison Baselines. We compare Fugue against three carefully chosen baselines representing the spectrum of existing approaches:

Native Flink is Apache Flink's built-in scaling mechanism that relies on checkpointing and restarting at the new parallelism. It serves as a representative of industry-standard production approaches.

Rhino (Proactive) [23] represents Rhino's core proactive replication protocol. As the original implementation is not publicly available, we implemented a faithful representation of this protocol on top of Flink. Each stateful operator is paired with a background replication task that periodically (default every 60 seconds) ships RocksDB's write-ahead log (WAL) entries to a hot standby replica on another `TaskManager`. During rescaling, we perform the handover by greedily prioritizing the standby replica.

Meces (Reactive Pull-based) [29] is adapted from the open-source Meces implementation with an off-cluster Redis deployment to avoid stealing compute from the job. When a new instance encounters a key for which it has no state, it issues a synchronous RPC to fetch it from the old instance via Redis.

5.2 End-to-end Performance

To study the end-to-end performance of Fugue, we conduct a scale-out experiment for the NEXMark workloads. The queries are initially running with 8 parallel instances of the key operators (`map` in Q1, `window` in Q5, `join` in Q8). After a 10-minute warm-up period, we trigger a scale-out to 12 instances. We evaluate the end-to-end latency of each query during the scaling event from record creation at the source to its emission from the final sink, measured using in-band latency markers. For throughput, we measure the total number of output records emitted by the sink operator per second.

Figure 6 and Figure 7 present the time-series results, with scaling triggered around the 600-second mark. The end-to-end latency traces, shown on a logarithmic scale in Figure 6, reveal the differences in user-facing disruption caused by each elasticity mechanism. The stateless Q1 workload confirms that the online protocols add little disruption when there is no state to move, while Native Flink still pays a restart blackout. As state grows in Q5 and Q8, the differences come from how each system prepares the target.

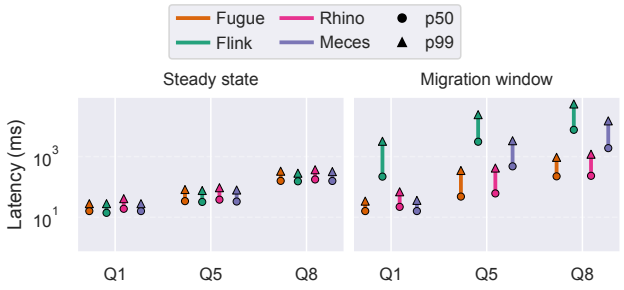


Figure 8: Latency quantiles during steady-state and migration windows. Each vertical range connects p50 (circle) to p99 (triangle) end-to-end marker latency for the steady-state and migration windows. Both panels use a log scale.

Figure 9: Fugue cutover breakdown. Bars decompose total cutover into Barrier Align, Transfer, Apply, and Drain components. Labels report total time and Final Delta size.

Meces avoids a full blackout, but its fetch-on-demand path creates a long tail because records alternate between local state hits and remote state fetches. On Q5, Fugue's migration-window p99 is 347ms, which is 98.6% lower than Native Flink and 89.5% lower than Meces. On Q8, Fugue's migration-window p99 is 943ms, 98.3% lower than Native Flink and 93.7% lower than Meces. Rhino and Fugue keep the handover shorter because the target is already warm when ownership changes. The key insight here is that the causes of latency tails differ across systems. Native Flink's tail is driven by restart recovery, while Meces's tail comes from repeated post-cutover remote state fetches. Fugue bounds the tail by reducing the handover to the final state delta, so the remaining disruption is shorter and decoupled from the full state migration cost.

The aggregate system throughput shown in Figure 7 demonstrates similar trends. Notably, for the large-state join in Q8, Meces has the largest drop because the increased state size and the two-input nature of the join operator lead to most records triggering slow, synchronous fetches and frequent cache misses. In contrast, Rhino and Fugue are more robust, with Fugue showing only a well-contained dip of 10% for approximately 16s compared with 104s for Native Flink and 54s for Meces. This result illustrates the efficiency of Fugue's push-based, pre-emptive state transfer, which ensures the new operator instances are "warmed up" and ready to process at nearly full capacity the moment they become active, a critical advantage over pull-based designs under heavy state contention.

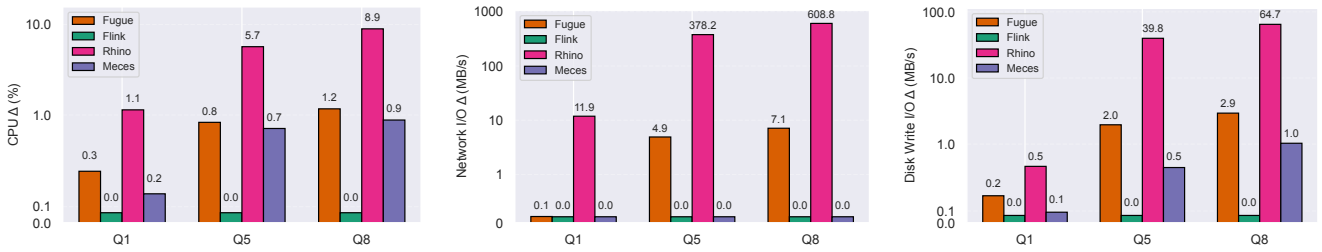
Figure 8 makes the latency metric explicit by showing the p50 to p99 latency quantiles during steady-state and migration windows. Fugue's median latency changes little during migration, while its tail grows predictably as more state must be finalized. Rhino shows a similar warm-target handover pattern, but it pays for that behavior continuously through replication. Meces and Native Flink separate sharply in the tail: Meces pays for remote state fetches after cutover, while Native Flink pays for restart and backlog recovery. The quantile summary therefore supports the time-series claim that Fugue does not eliminate all migration latency, but keeps the tail bounded and short-lived.

To study Fugue's cutover latency at a granular level, Figure 9 decomposes cutover latency into barrier alignment, final-delta transfer, state application at the target, and draining records buffered after the barrier. Only the migrating partition is blocked during this interval. Non-migrating partitions continue processing after barrier alignment. The breakdown shows that stateful workloads spend most cutover time in final-delta transfer and state application rather than in barrier coordination. Q8 takes 6.03s, or 2.6× Q5's duration of 2.31s, because the join leaves a 780MB final delta versus 210MB for Q5. Transfer and apply account for 69.7% of Q8's cutover time and 57.6% of Q5's, so the added cost remains bounded data-movement work without global coordination cost.

5.3 Overhead Analysis

A critical aspect of any elasticity mechanism is the resource overhead it introduces, both during steady-state operation and during an active migration. This section quantifies these costs to demonstrate Fugue's resource efficiency.

Steady-State Overhead. The core trade-off between proactive and reactive approaches is performance during migration versus cost during normal operation. To measure this, we run the NEXMark workloads under the same stable input rates for 30 minutes without triggering any scaling operations. We monitor the average resource utilization across all worker nodes. The results are presented in Figure 10, which compares the average relative overhead of CPU utilization, aggregate network I/O, and aggregate disk write I/O for each system relative to native Flink. These results show that Meces remains near the baseline across metrics, as its pull-based strategy avoids work until a migration is triggered. Similarly, Fugue does little work off the migration critical path. However, Fugue can incur a modest overhead in some scenarios because Fugue keeps a few readiness mechanisms active so that it can quickly respond to migration requests when they arise. For example, each operator exchanges heartbeats with the migration coordinator and exports extra metrics such as per-partition dirty-byte counters, which may slightly increase CPU and network usage. In contrast, Rhino's continuous replication strategy results in a significant overhead across all metrics, with CPU usage increasing by up to 8.9% due to the constant work of serializing, sending, receiving, and deserializing WAL entries, and network and disk I/O surging by over 600MB/s and 60 MB/s respectively because state changes are continuously replicated. Moreover, Figure 11 gives the measured Q8 memory trace over time. The line shows median `TaskManager` memory, and the band shows the interquartile range. Fugue's memory overhead is concentrated in the migration interval: it rises from a 32.4GB pre-migration average to a 36.9GB peak, then returns to 33.4GB after cutover. Rhino shows the opposite profile: standby replicas and



(a) CPU Overhead (b) Network Overhead (c) Disk I/O Overhead

Figure 10: Steady-state overhead for different approaches across three representative queries from the NEXMark benchmark suite. The numbers presented are the average overhead relative to the non-scaling Flink baseline over a 30-minute period. The y-axes use symmetric-log scaling to keep small reactive overheads visible alongside Rhino's larger continuous-replication cost.

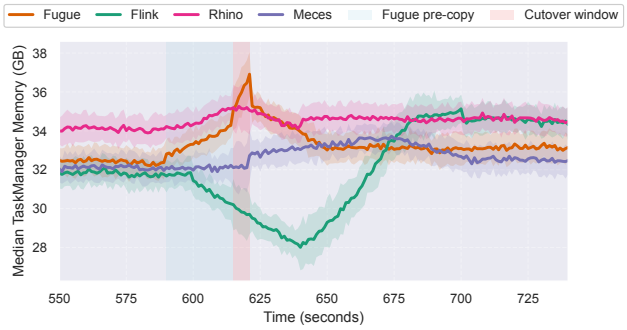


Figure 11: Per-`TaskManager` memory consumption over time for the Q8 scale-out experiment. The shaded regions mark Fugue's pre-copy and cutover windows.

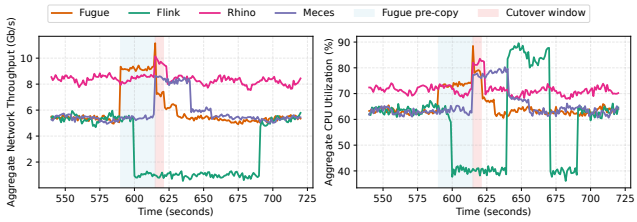


Figure 12: CPU and network overhead during migration.

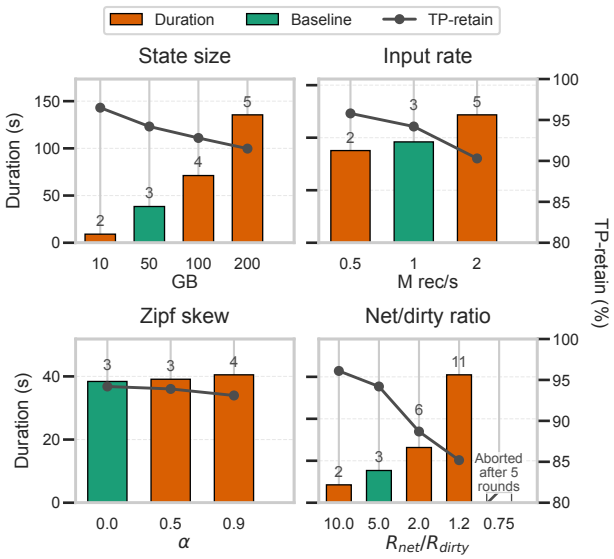


Figure 13: Sensitivity analysis of Fugue's migration performance. Each panel varies one parameter from the baseline configuration. Bars show reconfiguration duration, the line shows throughput retention, and numbers above bars show pre-copy rounds.

replication buffers keep memory elevated even before any migration is requested. Meces stays close to native Flink before migration but develops a post-cutover cache and remote-fetch pressure tail. These experiments highlight the continuous "resource tax" of proactive models. Fugue, as a reactive protocol, imposes only a modest steady-state resource footprint, making it a more economically and operationally efficient choice for deployments where migrations are infrequent but must be seamless when they occur.

Migration-Phase Overhead. While Fugue is efficient at rest, its reactive nature means it must expend resources to perform the migration when triggered. To analyze this, we measure migration-phase CPU and network resource traces during the scale-out of the NEXMark Q8 query. The results are shown in Figure 12. Fugue's preparatory pre-copy phase begins before the visible ownership transfer and adds a temporary CPU/network load without blocking the workload. This is followed by a brief cutover window where the final delta transfer and barrier alignment cause a short peak in network and CPU usage before resources promptly return to their stable baseline. In contrast, native Flink idles during restart and spikes during recovery, Rhino's baseline network and CPU usage are consistently elevated before the migration even begins due to the continuous overhead of state replication, and Meces pays its resource cost after the cutover where a prolonged fetch period causes a significant resource spike as the cold instances pull state on demand. Fugue therefore moves bulk transfer off the critical path and confines disruptive work to the cutover.

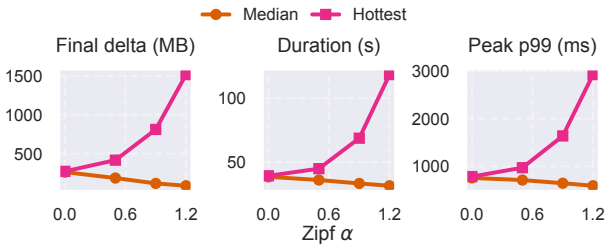


Figure 14: Skew sensitivity when migrating a median-load partition versus the hottest partition.

5.4 Sensitivity Analysis

To understand the robustness of Fugue and the performance impact of key system parameters, we conduct a sensitivity analysis using our controlled three-stage microbenchmark. In these experiments, we trigger a scale-out from 20 to 25 instances and measure three key metrics: *Reconfiguration Duration* (the total time from migration trigger to latency stabilization), *Throughput Retention* (the minimum throughput during the cutover as a percentage of the steady-state throughput), and the *Number of Pre-copy Rounds* required for convergence. We vary four parameters independently: state size, workload intensity (input rate), data skewness (Zipf parameter α), and the `R_net` / `R_dirty` ratio. The results of our analysis are summarized in Figure 13. The baseline configuration involves a 50 GB state, a 1M records/sec input rate, uniform data distribution (α=0.0), and `R_net` / `R_dirty`= 5.0×. Our findings demonstrate the robustness of the Fugue protocol across a wide range of conditions. First, as we increase the state size, the reconfiguration duration scales sub-linearly because larger state extends background pre-copy, while the cutover remains short. Second, higher input rates raise the dirty rate and require more pre-copy rounds, but throughput retention remains high because the added work is still mostly outside the critical path. Third, skew affects migration only when the migrated partition is itself hot. When skew increases but the migrated partition is cold or median-load, the aggregate input distribution changes without substantially increasing the dirty rate of the state being copied. In contrast, migrating the hottest partition leaves a much larger final delta, requires more pre-copy rounds, and produces a larger p99 spike. At $\alpha = 1.2$, moving the hottest partition leaves a 1516MB final delta, compared with 88MB for a median-load partition, and increases reconfiguration time from 31.4s to 118.6s. Peak p99 latency rises from 594ms to 2924ms, while throughput retention drops from 96.1% to 80.7%. Figure 14 therefore explains why the original aggregate skew row was insensitive: skew matters when the migration target is the hot partition, not when the partition being moved is cold or median-load.

The most insightful results come from directly manipulating the `R_net` / `R_dirty` ratio by throttling the background transfer bandwidth. As the available network bandwidth shrinks relative to the state dirty rate, the number of pre-copy rounds required for convergence increases significantly, from 2 rounds at a 10.0x ratio to 11 rounds at a 1.2x ratio. This directly translates to a longer Reconfiguration Duration. Crucially, even in the most constrained network scenario, the protocol still converges and maintains a high throughput retention of 85.2%. Under the extreme scenario where the available bandwidth is less than the dirty rate, the protocol's built-in non-convergence safeguard triggers to abort the migration after a handful of rounds, preserving correctness and availability by falling back to the last checkpoint.

6 RELATED WORK

Elastic Data Stores and Live Migration. Dynamic repartitioning is well studied in distributed storage systems [18, 21, 44, 51], including Dynamo [22], Cassandra [35], and BigTable [17]. These systems move passive data under replica protocols, often with eventual consistency. Stream processors instead move state that is coupled to active in-flight computation and must be synchronized with the input stream at an exact logical boundary. Fugue is also inspired by live VM and container migration [19, 36, 37, 40, 42, 43], which use iterative warmup copies followed by a short final-copy phase. Unlike VM migration, Fugue exploits structured operator state and uses in-band barriers to make the migrated state consistent with the distributed dataflow, not only with one process.

Elasticity in Stream Processing. Enabling elastic stream processing has been an active research area over the past decade, from stop-the-world scaling [7, 9, 10, 54] to online proactive or reactive protocols [14, 24, 29–31, 38, 46, 47, 53]. Proactive systems such as Megaphone [31] and Rhino [23] keep standby state warm but pay continuous replication overhead. Reactive systems migrate only when needed: SEEP [14] uses asynchronous control signals, Chi [38] introduces barrier-based coordination, Meces [29] fetches state on demand through external services, and DRRS [46] uses a multi-signal protocol to decouple migration triggers from routing confirmation. Fugue combines the best of both worlds with a simple, self-contained reactive protocol that minimizes disruption without continuous overhead or complex external dependencies.

7 CONCLUSIONS

In this paper, we present Fugue, a self-contained state migration framework for enabling seamless online elasticity in stateful stream processing systems. The core contribution of Fugue is a robust, two-phase reactive protocol. By first performing a pre-emptive, iterative background copy of the migrating state, Fugue decouples the expensive data transfer from the critical path of data processing. The final state ownership transfer is then orchestrated by a lightweight, in-band alignment barrier that leverages the underlying SPE's native synchronization mechanism to guarantee correctness with minimal disruption. Our implementation within Apache Flink and extensive evaluation show that Fugue provides a superior balance of operational efficiency, performance, and robustness under varying system conditions.

ACKNOWLEDGMENTS

We thank the anonymous VLDB reviewers for their valuable feedback. We thank Jeyhun Karimov for his insightful discussions at the early stages of this work. This research was undertaken, in part, thanks to funding from the Canada Research Chairs Program, NSERC, and ORF.

REFERENCES

[1] Tyler Akidau, Alex Balikov, Kaya Bekiroğlu, Slava Chernyak, Josh Haberman, Reuven Lax, Sam McVeety, Daniel Mills, Paul Nordstrom, and Sam Whittle. 2013. Millwheel: Fault-Tolerant Stream Processing at Internet Scale. *Proceedings of the VLDB Endowment* 6, 11 (2013), 1033–1044.

[2] Tyler Akidau, Robert Bradshaw, Craig Chambers, Slava Chernyak, Rafael J Fernández-Moctezuma, Reuven Lax, Sam McVeety, Daniel Mills, Frances Perry, Eric Schmidt, et al. 2015. The Dataflow Model: A Practical Approach to Balancing Correctness, Latency, and Cost in Massive-Scale, Unbounded, Out-of-Order Data Processing. *Proceedings of the VLDB Endowment* 8, 12 (2015), 1792–1803.

[3] Amazon Web Services. 2025. Amazon S3. https://aws.amazon.com/s3/. Official service page. Accessed July 18, 2026.

[4] Alexey Andreyev, Xu Wang, and Alex Eckert. 2019. Reinventing Our Data Center Network with F16, Minipack. https://engineering.fb.com/2019/03/14/data-center-engineering/f16-minipack/. Engineering at Meta blog. Accessed July 18, 2026.

[5] Apache Software Foundation. 2025. Apache Flink. https://flink.apache.org/. Official project website. Accessed July 18, 2026.

[6] Apache Software Foundation. 2025. Parallel Execution (Flink DataStream API). https://nightlies.apache.org/flink/flink-docs-master/docs/dev/datastream/execution/parallel/. Flink documentation (nightlies). Accessed July 18, 2026.

[7] Apache Software Foundation / Apache Kafka Project. 2025. Kafka Streams - Apache Kafka Documentation. https://kafka.apache.org/documentation/streams/. Official project documentation. Accessed July 18, 2026.

[8] Apache Spark Project. 2025. Spark Streaming Programming Guide. https://spark.apache.org/docs/latest/streaming-programming-guide.html. Latest version documentation. Accessed July 18, 2026.

[9] Michael Armbrust, Tathagata Das, Joseph Torres, Burak Yavuz, Shixiong Zhu, Reynold Xin, Ali Ghodsi, Ion Stoica, and Matei Zaharia. 2018. Structured Streaming: A Declarative API for Real-Time Applications in Apache Spark. In *Proceedings of the 2018 International Conference on Management of Data*. 601–613.

[10] Paris Carbone, Stephan Ewen, Gyula Fóra, Seif Haridi, Stefan Richter, and Kostas Tzoumas. 2017. State Management in Apache Flink: Consistent Stateful Distributed Stream Processing. *Proceedings of the VLDB Endowment* 10, 12 (2017), 1718–1729.

[11] Paris Carbone, Marios Fragkoulis, Vasiliki Kalavri, and Asterios Katsifodimos. 2020. Beyond Analytics: The Evolution of Stream Processing Systems. In *Proceedings of the 2020 ACM SIGMOD international conference on Management of data*. 2651–2658.

[12] Paris Carbone, Asterios Katsifodimos, Stephan Ewen, Volker Markl, Seif Haridi, and Kostas Tzoumas. 2015. Apache Flink: Stream and Batch Processing in a Single Engine. *The Bulletin of the Technical Committee on Data Engineering* 38, 4 (2015), 28–38.

[13] Valeria Cardellini, Francesco Lo Presti, Matteo Nardelli, and Gabriele Russo Russo. 2022. Runtime Adaptation of Data Stream Processing Systems: The State of the Art. *Comput. Surveys* 54, 11s (2022), 1–36.

[14] 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 2013 ACM SIGMOD international conference on Management of data*. 725–736.

[15] Badrish Chandramouli, Jonathan Goldstein, Mike Barnett, Robert DeLine, Danyel Fisher, John C Platt, James F Terwilliger, and John Wernsing. 2014. Trill: A High-Performance Incremental Query Processor for Diverse Analytics. *Proceedings of the VLDB Endowment* 8, 4 (2014), 401–412.

[16] K Mani Chandy and Leslie Lamport. 1985. Distributed Snapshots: Determining Global States of Distributed Systems. *ACM Transactions on Computer Systems (TOCS)* 3, 1 (1985), 63–75.

[17] Fay Chang, Jeffrey Dean, Sanjay Ghemawat, Wilson C Hsieh, Deborah A Wallach, Mike Burrows, Tushar Chandra, Andrew Fikes, and Robert E Gruber. 2008. Bigtable: A Distributed Storage System for Structured Data. *ACM Transactions on Computer Systems (TOCS)* 26, 2 (2008), 1–26.

[18] Kaiji Chen, Yongluan Zhou, and Yu Cao. 2015. Online Data Partitioning in Distributed Database Systems.. In *EDBT*. 1–12.

[19] Christopher Clark, Keir Fraser, Steven Hand, Jacob Gorm Hansen, Eric Jul, Christian Limpach, Ian Pratt, and Andrew Warfield. 2005. Live Migration of Virtual Machines. In *Proceedings of the 2nd conference on Symposium on Networked Systems Design & Implementation-Volume 2*. 273–286.

[20] Gianpaolo Cugola and Alessandro Margara. 2012. Processing Flows of Information: From Data Stream to Complex Event Processing. *ACM Computing Surveys (CSUR)* 44, 3 (2012), 1–62.

[21] Carlo Curino, Evan Jones, Yang Zhang, and Sam Madden. 2010. Schism: A Workload-Driven Approach to Database Replication and Partitioning. *Proceedings of the VLDB Endowment* 3, 1-2 (2010), 48–57.

[22] Giuseppe DeCandia, Deniz Hastorun, Madan Jampani, Gunavardhan Kakulapati, Avinash Lakshman, Alex Pilchin, Swaminathan Sivasubramanian, Peter Vosshall, and Werner Vogels. 2007. Dynamo: Amazon's Highly Available Key-Value Store. *ACM SIGOPS operating systems review* 41, 6 (2007), 205–220.

[23] 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 2020 ACM SIGMOD International Conference on Management of Data*. 2471–2486.

[24] Raul Castro Fernandez, Matteo Migliavacca, Evangelia Kalyvianaki, and Peter Pietzuch. 2014. Making State Explicit for Imperative Big Data Processing. In *2014 USENIX annual technical conference (USENIX ATC 14)*. 49–60.

[25] Armando Fox, Rean Griffith, Anthony Joseph, Randy Katz, Andrew Konwinski, Gunho Lee, David Patterson, Ariel Rabkin, Ion Stoica, et al. 2009. Above the Clouds: A Berkeley View of Cloud Computing. *Dept. Electrical Eng. and Comput. Sciences, University of California, Berkeley, Rep. UCB/EECS* 28, 13 (2009), 2009.

[26] Xinwei Fu, Talha Ghaffar, James C Davis, and Dongyoon Lee. 2019. {EdgeWise}: A Better Stream Processing Engine for the Edge. In *2019 USENIX Annual Technical Conference (USENIX ATC 19)*. 929–946.

[27] Buğra Gedik, Scott Schneider, Martin Hirzel, and Kun-Lung Wu. 2013. Elastic Scaling for Data Stream Processing. *IEEE Transactions on Parallel and Distributed Systems* 25, 6 (2013), 1447–1463.

[28] Can Gencer, Marko Topolnik, Viliam Ďurina, Emin Demirci, Ensar B Kahveci, Ali Gürbüz, Ondřej Lukáš, József Bartók, Grzegorz Gierlach, František Hartman, Ufuk Yılmaz, Mehmet Doğan, Mohamed Mandouh, Marios Fragkoulis, and Asterios Katsifodimos. 2021. Hazelcast Jet: Low-Latency Stream Processing at the 99.99th Percentile. *Proceedings of the VLDB Endowment* 14, 12 (2021), 3110–3121.

[29] 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 *2022 USENIX Annual Technical Conference (USENIX ATC 22)*. 539–556.

[30] Thomas Heinze, Zbigniew Jerzak, Gregor Hackenbroich, and Christof Fetzer. 2014. Latency-Aware Elastic Scaling for Distributed Data Stream Processing Systems. In *Proceedings of the 8th ACM International Conference on Distributed Event-Based Systems*. 13–22.

[31] Moritz Hoffmann, Andrea Lattuada, Frank McSherry, Vasiliki Kalavri, John Liagouris, and Timothy Roscoe. 2019. Megaphone: Latency-Conscious State Migration for Distributed Streaming Dataflows. *Proceedings of the VLDB Endowment* 12, 9 (2019), 1002–1015.

[32] Vasiliki Kalavri, John Liagouris, Moritz Hoffmann, Desislava Dimitrova, Matthew Forshaw, and Timothy Roscoe. 2018. Three Steps Is All You Need: Fast, Accurate, Automatic Scaling Decisions for Distributed Streaming Dataflows. In *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18)*. 783–798.

[33] David Karger, Eric Lehman, Tom Leighton, Rina Panigrahy, Matthew Levine, and Daniel Lewin. 1997. Consistent Hashing and Random Trees: Distributed Caching Protocols for Relieving Hot Spots on the World Wide Web. In *Proceedings of the twenty-ninth annual ACM symposium on Theory of computing*. 654–663.

[34] 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 2015 ACM SIGMOD international conference on Management of data*. 239–250.

[35] Avinash Lakshman and Prashant Malik. 2010. Cassandra: A Decentralized Structured Storage System. *ACM SIGOPS operating systems review* 44, 2 (2010), 35–40.

[36] Haikun Liu, Hai Jin, Xiaofei Liao, Liting Hu, and Chen Yu. 2009. Live Migration of Virtual Machine Based on Full System Trace and Replay. In *Proceedings of the 18th ACM international symposium on High performance distributed computing*. 101–110.

[37] Haikun Liu, Cheng-Zhong Xu, Hai Jin, Jiayu Gong, and Xiaofei Liao. 2011. Performance and Energy Modeling for Live Migration of Virtual Machines. In *Proceedings of the 20th international symposium on High performance distributed computing*. 171–182.

[38] 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. *Proceedings of the VLDB Endowment* 11, 10 (2018), 1303–1316.

[39] Meta Platforms, Inc. (Facebook). 2025. RocksDB: A Persistent Key-Value Store. https://rocksdb.org/. Official project website. Accessed July 18, 2026.

[40] Andrey Mirkin, Alexey Kuznetsov, and Kir Kolyshkin. 2008. Containers Checkpointing and Live Migration. In *Proceedings of the Linux Symposium*, Vol. 2. 85–90.

[41] Derek G Murray, Frank McSherry, Rebecca Isaacs, Michael Isard, Paul Barham, and Martín Abadi. 2013. Naiad: A Timely Dataflow System. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*. 439–455.

[42] Michael Nelson, Beng-Hong Lim, Greg Hutchins, et al. 2005. Fast Transparent Migration for Virtual Machines.. In *USENIX Annual technical conference, general track*. 391–394.

[43] OpenVZ. 2025. CRIU — Checkpoint/Restore in Userspace. https://criu.org/Main_Page. Accessed July 18, 2026.

[44] M Tamer Özsu, Patrick Valduriez, et al. 1999. *Principles of Distributed Database Systems*. Vol. 2. Springer.

[45] Heejin Park, Shuang Zhai, Long Lu, and Felix Xiaozhu Lin. 2019. {StreamBox-TZ}: Secure Stream Analytics at the Edge with {TrustZone}. In *2019 USENIX*

Annual Technical Conference (USENIX ATC 19). 537–554.

[46] Yunfan Qing and Wenli Zheng. 2025. Towards Fine-Grained Scalability for Stateful Stream Processing Systems. In *2025 IEEE 41st International Conference on Data Engineering (ICDE)*. IEEE Computer Society, 3835–3848.

[47] Sumanaruban Rajadurai, Jeffrey Bosboom, Weng-Fai Wong, and Saman Amarasinghe. 2018. Gloss: Seamless Live Reconfiguration and Reoptimization of Stream Programs. In *Proceedings of the Twenty-Third International Conference on Architectural Support for Programming Languages and Operating Systems*. 98–112.

[48] Sriram Rao, Raghu Ramakrishnan, Adam Silberstein, Mike Ovsiannikov, and Damian Reeves. 2012. Sailfish: A Framework for Large Scale Data Processing. In *Proceedings of the Third ACM Symposium on Cloud Computing*. 1–14.

[49] Igor Sfiligoi. 2020. Demonstrating 100 Gbps in and Out of the Public Clouds. In *Practice and Experience in Advanced Research Computing 2020: Catch the Wave*. Association for Computing Machinery, 495–499.

[50] Konstantin Shvachko, Hairong Kuang, Sanjay Radia, and Robert Chansler. 2010. The Hadoop Distributed File System. In *2010 IEEE 26th symposium on mass storage systems and technologies (MSST)*. Ieee, 1–10.

[51] Alexander Thomson, Thaddeus Diamond, Shu-Chun Weng, Kun Ren, Philip Shao, and Daniel J Abadi. 2012. Calvin: Fast Distributed Transactions for Partitioned Database Systems. In *Proceedings of the 2012 ACM SIGMOD international conference on management of data*. 1–12.

[52] Pete Tucker, Kristin Tufte, Vassilis Papadimos, and David Maier. 2008. *NEXMark: A Benchmark for Queries Over Data Streams*. Technical Report. OGI School of Science and Engineering at OHSU.

[53] Yingjun Wu and Kian-Lee Tan. 2015. ChronoStream: Elastic Stateful Stream Computation in the Cloud. In *2015 IEEE 31st International Conference on Data Engineering*. IEEE, 723–734.

[54] Matei Zaharia, Tathagata Das, Haoyuan Li, Timothy Hunter, Scott Shenker, and Ion Stoica. 2013. Discretized Streams: Fault-Tolerant Streaming Computation at Scale. In *Proceedings of the twenty-fourth ACM symposium on operating systems principles*. 423–438.

[55] Zayo Group. 2025. The Zayo Bandwidth Report: Insights & Trends Driving the Bandwidth Boom. https://www.zayo.com/info/the-zayo-bandwidth-report/. Official report page. Accessed July 18, 2026.

[56] Yuqiu Zhang, Tongkun Zhang, Gengrui Zhang, and Hans-Arno Jacobsen. 2023. Lifting the Fog of Uncertainties: Dynamic Resource Orchestration for the Containerized Cloud. In *Proceedings of the 2023 ACM Symposium on Cloud Computing*. 48–64.