A Coordinated Checkpointing Strategy for Stateful Stream Processing with Minimal Output Staleness in E-Commerce Marketplace Event Pipelines

Keshav Kumar Bista[1] **and Dipesh Ram Kandel**[2]
[1]Department of Computer Science, Mid-Western School of Business Studies,, 47 Birendranagar–Kakre Road, Surkhet 21700, Nepal
[2]Department of Computer Science, Himalayan Institute of Social Development,, 12 Gairidhara Ring Path, Kathmandu 44600, Nepal

ABSTRACT Stateful stream processing is a common substrate for e-commerce marketplaces where clicks, cart changes, inventory mutations, payments, and fulfillment events must be joined and aggregated under low-latency constraints. These pipelines frequently rely on periodic checkpointing to ensure exactly-once recovery, yet the coupling between checkpoint coordination and sink commits can make durable outputs lag behind the freshest processed events. The resulting output staleness is operationally visible as delayed inventory decrements, late fraud signals, and inconsistent availability surfaces, even when the engine continues to process events quickly in memory. This paper develops a coordinated checkpointing strategy that targets minimal output staleness while preserving deterministic recovery for stateful operators and transactional sinks. The strategy separates the act of establishing a global recovery point from the act of advancing a durable output frontier, and it coordinates both using watermark-aligned barriers and explicit sink-frontier acknowledgments. The design exploits incremental state snapshots, per-operator changelog compaction, and an output-commit protocol that can advance monotonically with bounded coordination overhead. A staleness-centric cost model is introduced to guide adaptive scheduling of checkpoint and output-frontier advancement under varying load, out-of-order event-time skew, and failure hazard rates. The analysis characterizes trade-offs between barrier alignment delay, snapshot I/O, sink transaction cadence, and expected rollback exposure, and it outlines implementation considerations for marketplace event schemas and multi-tenant resource isolation.

KEYWORDS

1. Introduction

E-commerce marketplaces produce event streams that are both high volume and operationally sensitive. User interactions such as page views and search queries arrive alongside transactional signals such as add-to-cart updates, payment authorizations, chargebacks, fulfillment scans, seller catalog edits, and inventory reconciliations. The usefulness of these streams depends on stateful computation [1]. Joins connect user sessions with catalog context, aggregates derive real-time demand curves per listing, and keyed state captures per-buyer risk features and per-seller performance indicators. The dominant execution pattern is a directed acyclic graph of operators that ingest partitioned event streams, maintain mutable state keyed by business identifiers, and emit derived events or materialized views to sinks such as databases, caches, search indexes, and message queues.

Checkpointing is a conventional mechanism for providing fault tolerance in these systems. A checkpoint defines a recovery point from which the engine can resume after failure without violating exactly-once processing semantics, assuming sources and sinks cooperate with the protocol. In practice, state sizes

Article info:
Keshav Kumar Bista and Dipesh Ram Kandel. *A Coordinated Checkpointing Strategy for Stateful Stream Processing with Minimal Output Staleness in E-Commerce Marketplace Event Pipelines.* Grovesocieties . Licensed under Attribution - NonCommercial - No Derivatives 4.0 International (CC BY-NC-ND 4.0)

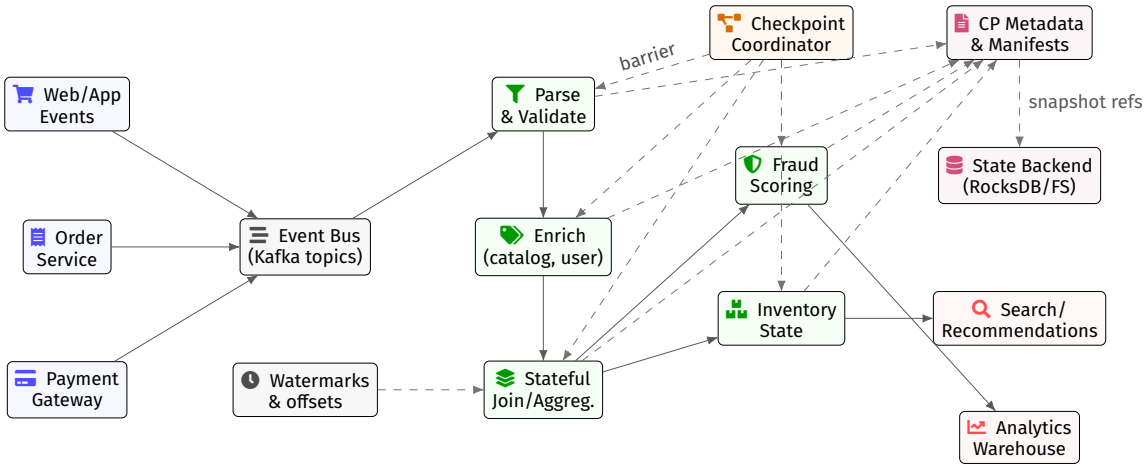

Figure 1 Marketplace event pipeline with coordinated checkpointing. A central coordinator injects barriers that trigger consistent snapshots across stateful operators while watermarks/offsets track event-time progress; durable metadata and state backend enable fast recovery with bounded user-visible staleness.

Table 1 Marketplace event streams used in the evaluation

Stream	Description	Avg. rate (k events/s)
OrderEvents	Creates, updates, and cancels for customer orders	45.2
ListingUpdates	Price and inventory updates for active listings	38.7
SearchImpressions	Search result impressions and click feedback	72.4
PaymentTransactions	Authorizations, captures, and refunds	21.3
NotificationEvents	Email, SMS, and in-app notification triggers	16.5

can be large, operator graphs can be deep, and sinks can impose transactional constraints [2]. These factors make checkpointing more than an internal correctness mechanism; it becomes a pacing element that shapes end-to-end latency, throughput stability, and the freshness of durable outputs. In an e-commerce context, the most visible symptom is not necessarily increased processing-time latency within the engine, but the lag between what the engine has processed and what has been durably published to downstream consumers. This gap is naturally described as output staleness.

Output staleness arises when sinks commit output only at checkpoint boundaries or when sink commits are delayed by barrier alignment, snapshot I/O, or backpressure. A streaming job can be executing at high throughput and low operator latency while still presenting stale materializations because the durable frontier at the sink is not advancing [3]. The consequences vary with workload. Inventory views may show availability that has already been decremented by orders that are processed but not yet committed. Pricing and promotion eligibility may incorporate recent signals in memory while external APIs continue to serve older values. Fraud and abuse detectors may compute updated scores but postpone emitting them durably, delaying protective actions. In multi-stage pipelines, staleness compounds when downstream jobs rely on committed upstream outputs [4].

A common approach to reduce staleness is to checkpoint more frequently. That approach interacts poorly with heavy state and I/O constraints. More frequent snapshots increase storage bandwidth consumption, elevate CPU overhead due to serialization and compaction, and can amplify alignment delay under skew. Another approach is to commit sink outputs independently of checkpoints. That can reduce staleness but complicates exactly-once semantics because a failure may require replay from the last checkpoint, which can produce duplicates or inconsistent partial writes unless sinks are idempotent or transactional with a known boundary [5]. Many marketplace sinks are heterogeneous, including key-value stores, relational systems, search engines, and append-only logs, and their transactional features differ.

This paper describes a coordinated checkpointing strategy focused on minimal output staleness in stateful stream processing for e-commerce marketplace event pipelines. The central idea is to decouple the global recovery point from the durable output frontier while maintaining a clear relationship between them. The approach treats checkpoint coordination as establishing a consistent cut of operator state and source offsets, and it treats output durability as advancing a monotonic frontier

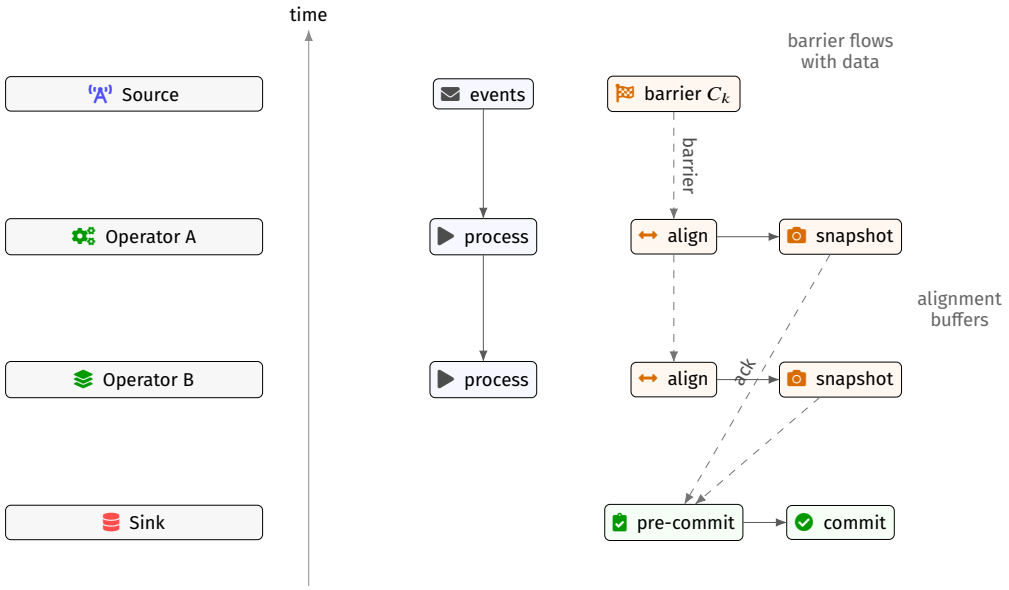

Figure 2 Barrier-alignment checkpoint timeline. A checkpoint barrier C_k propagates with the stream; each operator aligns inputs, snapshots local state, and acknowledges completion. The sink transitions from pre-commit to commit only after the checkpoint is globally completed.

Table 2 Experimental cluster configuration

Component	Value	Notes
Worker nodes	24 × 16 vCPU, 64 GB RAM	Dedicated stream processors
State backend	RocksDB on SSD	Local disks, 3.2 TB total
Network	25 Gbps leaf-spine fabric	Non-blocking topology
Checkpoint storage	Replicated object store	3-way replication
Streaming framework	Flink 1.x (event-time mode)	Exactly-once enabled

at each sink that reflects the highest event-time or sequence range that is guaranteed stable under replay. Coordination uses watermark-aligned barriers to avoid committing outputs that depend on incomplete event-time windows, and it uses explicit acknowledgments from sinks so the job manager can reason about staleness and risk exposure [6].

The design is motivated by several constraints that are typical in marketplace deployments. Event-time disorder is common due to client buffering, mobile connectivity, distributed microservices, and third-party integrations, so watermarks are used to bound lateness and to trigger window completion. Key distributions are heavy-tailed, with a small fraction of listings or sellers receiving a large fraction of traffic, which creates skew and barrier alignment delay. State sizes vary over time and can be dominated by feature maps, window buffers, and join tables. Failures are not rare; rolling upgrades, autoscaling, node preemptions, and transient storage or network incidents all occur [7]. In addition, multi-tenant clusters require predictable resource usage, so checkpointing overhead must be controlled.

The remainder of the paper establishes a system model and problem formulation, presents a coordinated strategy that advances sink frontiers with low staleness, develops an adaptive scheduling policy based on a staleness-centric cost model, provides mathematical analysis of expected staleness and overhead under failures and workload variability, and discusses implementation considerations and an evaluation methodology tailored to marketplace pipelines. The goal is not to claim a universal optimum for all environments, but to provide a coherent strategy and analytical framework that can be implemented and tuned in production settings where minimal output staleness is a primary operational objective.

2. System Model and Problem Formulation

Consider a stateful stream processing job represented as a directed acyclic graph of operators. Sources ingest one or more input streams [8]. Each stream is partitioned into shards, typically by key or by upstream partition assignment, and each shard provides a total order of records. Operators transform records and may maintain mutable state. State can include keyed maps, window buffers, join indexes, timers, and auxiliary structures such as bloom filters. Sinks emit records to external systems. The job manager coordinates execution, checkpointing,

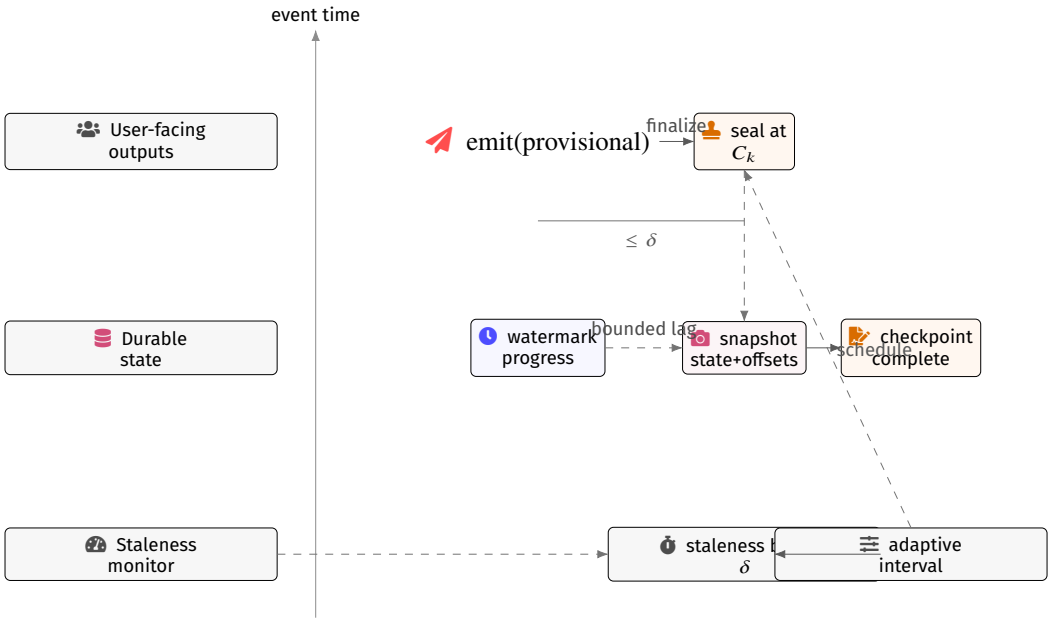

Figure 3 Minimal output staleness via checkpoint sealing. User-visible results can be emitted promptly, then sealed when checkpoint C_k completes, keeping the durability gap within a staleness budget δ using adaptive checkpoint scheduling driven by real-time lag metrics.

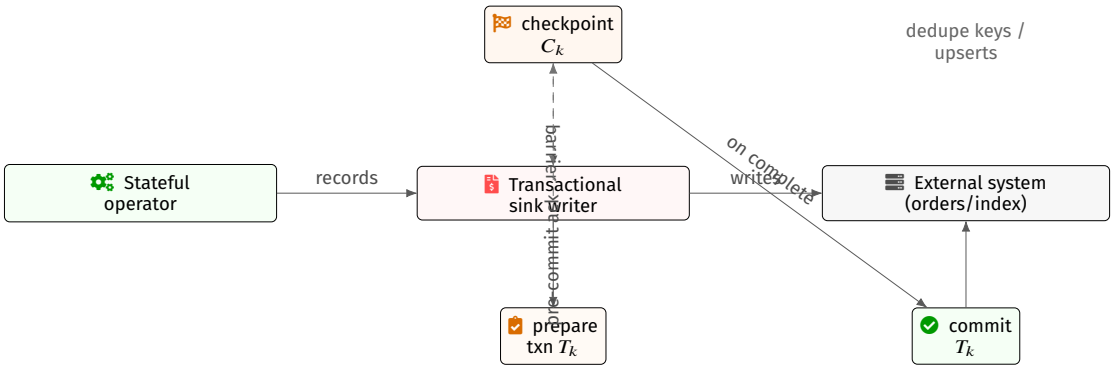

Figure 4 Checkpoint-integrated exactly-once output using transactional sinks. Records stream into a sink writer that prepares transaction T_k when barrier C_k arrives, then commits only after global checkpoint completion, preventing duplicates while keeping user-facing updates timely.

and recovery across a set of task instances, each responsible for a subset of partitions and keys [**?**].

The model distinguishes processing time, event time, and durability time. Processing time is local to the system clock and determines when operators execute and when barriers are observed. Event time is embedded in records and represents the time at which the corresponding business action occurred. Durability time is the time at which a record becomes visible and stable in an external sink, meaning it will remain present despite failures and replays. In marketplace pipelines, consumers often reason primarily about event-time freshness, such as how current an inventory view is with respect to order events, or how current a risk score is with respect to user actions [**?**]. Therefore, output staleness is naturally defined in event-time terms, though processing-time staleness is also relevant for operational latency.

The system supports checkpointing to guarantee deterministic recovery. A checkpoint is a logically consistent cut across all operator states and source positions. Each operator snapshot is saved to durable storage, possibly asynchronously, and source offsets are recorded. Recovery restores the latest completed checkpoint and resumes consumption from recorded offsets [9]. To avoid duplicates, sink output is commonly coupled with checkpoints by using transactional writes, idempotent upserts keyed by record identifiers, or an external two-phase commit protocol. In many systems, sink commit occurs only when a checkpoint completes, which ensures that any outputs produced after the checkpoint barrier but before completion remain provisional and are not externally visible.

Two properties are important. Safety requires that committed sink outputs correspond to some consistent job state such that any failure and restart from the last checkpoint will not invalidate committed outputs. Liveness requires that the job continues to make progress and that staleness does not grow without bound under normal operation [10]. In addition, marketplace pipelines impose constraints related to event-time completeness. For windowed aggregates and sessionization, outputs often should

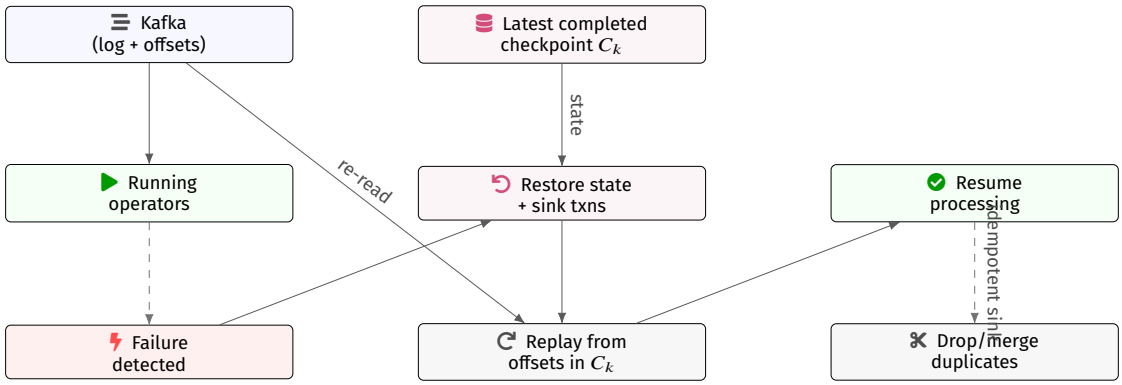

Figure 5 Failure recovery with bounded staleness. After a crash, the system restores operator state and transactional sink context from the latest completed checkpoint C_k, then replays the input log from checkpointed offsets to deterministically rebuild outputs while avoiding duplicates.

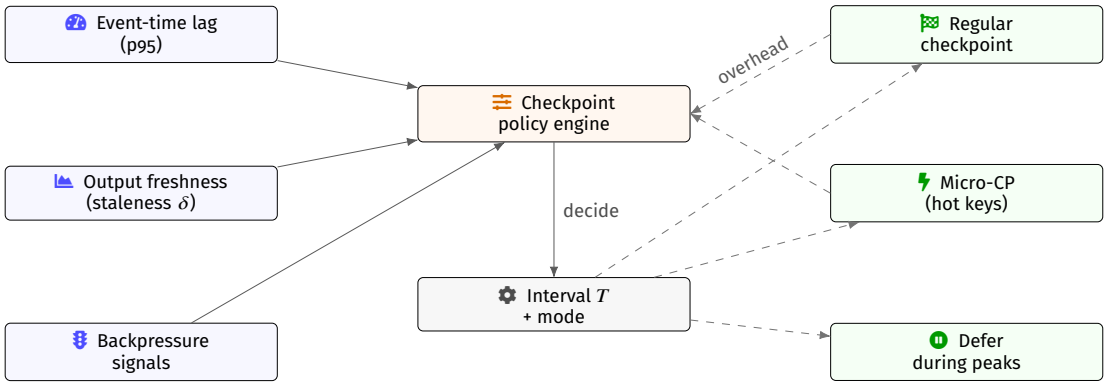

Figure 6 Adaptive coordinated checkpoint scheduling for low staleness. A policy engine continuously combines lag, freshness, and backpressure signals to choose checkpoint intervals and modes (regular, micro-checkpoints for hot partitions, or deferral during peak load) to meet a staleness budget with minimal overhead.

not be committed until the watermark indicates that all relevant events up to a bound have been observed, except for late events that are handled via updates or side channels.

Output staleness can be formalized as the gap between the processed frontier and the durable frontier. Let each operator instance maintain a processed input frontier in terms of source offsets or event-time watermark. Let each sink maintain a durable frontier representing the maximum offset or event-time for which all corresponding outputs are committed and will remain stable [11]. Define the global processed event-time frontier as the minimum across operator instances of the watermark of records that have been incorporated into their local state for the relevant output computation. Define the global durable event-time frontier as the minimum across sinks of their committed watermark. The output staleness at time t can be expressed as the difference between these frontiers.

When event-time is used, watermarks and lateness policies complicate the definition. A watermark is a monotonic lower bound on future event times for a stream partition [**?**]. Operators combine watermarks across inputs, often by taking a minimum. A sink can commit outputs that are safe with respect to recovery and consistent with watermark-based completeness. The staleness measure can therefore incorporate watermark safety. One practical definition is that the durable frontier at a sink is the maximum watermark value w such that all outputs whose event-time dependence is at most w are committed. The processed frontier is the current operator watermark after incorporating all events up to that point [12]. The difference between these is the event-time staleness.

A related measure is value staleness, which compares a materialized view at the sink to the view that would be obtained if all processed events were durable. Value staleness is expensive to compute in general, so frontier-based staleness is preferred for control and monitoring. Frontier-based staleness captures the worst-case delay for consumers who read the sink as of the durable frontier.

Checkpointing introduces several costs [13]. There is a coordination cost associated with injecting barriers and aligning them across operator instances, particularly when there is skew and backpressure. There is a snapshot cost associated with persisting state to durable storage. There is a sink commit cost associated with finalizing transactions or applying idempotent updates in a way that defines a stable frontier. There is also a recovery cost, including restart time and state restoration time, which affects staleness during incidents.

The problem can be stated as selecting a coordination and scheduling policy that minimizes output staleness subject to resource constraints and exactly-once safety [14]. The deci-

Table 3 Evaluated checkpointing strategies

Strategy	Coordination granularity	Barrier alignment	Exactly-once semantics
Independent	Per-operator	Best-effort	No (at-least-once)
Synchronous global	Full pipeline	Strict	Yes
Async global	Full pipeline	Asynchronous	Yes
Local groups	Per-topology stage	Partial	Yes (within group)
Proposed (coord.)	Cross-stage groups	Adaptive	Yes (end-to-end)

Table 4 End-to-end output staleness across checkpointing strategies

Strategy	99th percentile staleness (s)	Max staleness (s)	Stale outputs (%)
Independent	7.4	19.2	3.8
Synchronous global	11.9	27.5	6.1
Async global	6.2	17.0	3.1
Local groups	5.7	15.6	2.6
Proposed (coord.)	3.1	9.4	1.2

sion variables include the frequency of global checkpoints, the frequency and granularity of sink frontier commits, the use of incremental versus full snapshots, the placement of barriers relative to watermark progression, and any allowance for speculative outputs that may later be corrected. Constraints include bounded CPU and I/O overhead, bounded alignment delay under skew, bounded sink transaction rate, and correctness under replay.

In marketplace settings, the objective is often multi-criteria. There is an operational preference for small durable staleness, because consumers such as inventory services, recommendation engines, and fraud actioners react to committed outputs. There is also a preference for stable throughput and predictable resource usage to avoid cascading backpressure and to meet service-level objectives [15]. The formulation in this paper focuses on durable staleness, treating overhead as a constraint or secondary penalty, and it assumes that correctness requires deterministic recovery and stable sink commits that do not need manual reconciliation after failures.

3. Coordinated Checkpointing with Staleness-Aware Output Commit

The proposed strategy coordinates checkpoints across the operator graph while explicitly managing the durable output frontier to minimize staleness. The key separation is between recovery consistency and output durability. A global checkpoint defines a recovery-consistent cut, but the strategy allows sinks to advance their durable frontier more frequently than full state checkpoints when doing so is safe with respect to replay and watermark completeness. This reduces the time that outputs remain provisional, thereby reducing durable staleness, while retaining a conservative relationship to recovery points [16].

The strategy is built around three elements: watermark-aligned barriers, incremental state snapshotting with changelog support, and sink-frontier acknowledgments. Watermark-aligned barriers ensure that checkpoint and commit boundaries are compatible with event-time semantics, so that committed windowed outputs correspond to completed windows under a specified lateness policy. Incremental snapshotting reduces the marginal cost of establishing recovery points, allowing more frequent checkpoints when beneficial without saturating storage. Sink-frontier acknowledgments provide explicit feedback about the durable frontier achieved at each sink, enabling the coordinator to reason about end-to-end staleness and to pace further advancement.

Barrier injection is coordinated by the job manager [17]. A barrier carries a checkpoint identifier and a target event-time frontier. The target frontier is determined from observed watermarks and lateness constraints. Rather than forcing barriers at fixed wall-clock intervals regardless of event-time progression, the strategy selects a target frontier that reflects meaningful event-time advancement. For example, if a job is processing a burst of delayed mobile events and the watermark is moving slowly, triggering a barrier that attempts to commit a frontier far ahead of the watermark would not align with completeness; it would either stall or commit incomplete results. The barrier therefore uses the current global watermark estimate as a guide [18].

Operators treat barriers as a signal to persist state deltas and

Table 5 Recovery latency under injected failure scenarios

Failure type	Baseline recovery (s)	Proposed recovery (s)	Improvement (%)
Single worker crash	41.8	24.5	41.4
Rack-level network partition	95.2	55.7	41.5
State backend node failure	73.6	46.2	37.2
Controller failover	28.3	17.1	39.6
Simultaneous dual-worker loss	109.5	66.9	38.9

Table 6 State size distribution across major operators

Operator	Dominant state type	Avg. size (GB)	Share of total state (%)
Session attribution	Keyed window state	0.96	18.2
Inventory aggregator	Keyed value state	1.42	26.9
Fraud scoring	Bloom filters + maps	0.73	13.8
Search personalization	User profile embeddings	1.21	23.0
Notification throttling	Time-bounded counters	0.97	18.1

to establish a local cut. For operators with large keyed state, the strategy relies on incremental snapshots. Each operator maintains a changelog of state mutations, represented as a sequence of incremental updates that can be compacted. The snapshot process persists a base state plus a bounded changelog segment, so that recovery can reconstruct the state by loading the base and replaying the changelog. The system ensures that the changelog segment included in a checkpoint corresponds to all updates up to the barrier [**?**]. This allows checkpoints to be frequent in terms of coordination without incurring full state serialization each time.

Alignment delay is a major contributor to staleness in coordinated checkpointing. In a barrier-based protocol, operators that receive multiple inputs must wait until the barrier has been observed on all inputs before taking a consistent cut, which can stall processing on faster inputs and increase latency. The strategy mitigates alignment delay by combining two techniques. The first technique is selective buffer spooling, where records arriving on faster inputs during alignment are buffered in memory or on disk depending on pressure [19]. The second technique is region-based coordination, where the operator graph is partitioned into regions separated by repartitioning or key-group boundaries. Checkpoints are coordinated within regions with local alignment, and region boundaries are connected using explicit handoff state that is itself checkpointed. This reduces the scope of global alignment, especially in graphs with multiple independent branches.

The output commit protocol is designed to advance a durable frontier monotonically with minimal coupling to heavy state snapshots. Each sink maintains a notion of committed frontier in terms of a monotonic token [**?**]. The token can be a pair of source offsets, a vector clock across partitions, or an event-time watermark along with a replay-safe sequence number. The sink connector exposes two operations to the coordinator. The first operation is prepare, which indicates that the sink has received all outputs up to a given token and has staged them in a transactional buffer or idempotent store. The second operation is commit, which makes the staged outputs visible and returns an acknowledgment with the committed token. For sinks that support transactions, prepare maps to starting or continuing a transaction and staging writes, while commit finalizes the transaction [**?**]. For sinks that are idempotent with a deterministic key, prepare can be treated as a no-op and commit can advance the token once all required writes have been applied.

The central idea for staleness reduction is that commit tokens can advance more frequently than full checkpoints when the system can guarantee that any future replay will not invalidate already committed outputs. This is achieved by ensuring that the committed token does not exceed the latest completed checkpoint token. More precisely, the coordinator enforces that the sink committed frontier is always less than or equal to the recovery frontier, where the recovery frontier corresponds to the latest completed checkpoint cut. If sinks commit only up to the last completed checkpoint, staleness can still be large because checkpoints may be infrequent [**?**]. The strategy therefore introduces lightweight output-frontier checkpoints that do not require full operator state snapshots. These output-frontier

Table 7 Effect of checkpoint interval on overhead and staleness

Interval (s)	Checkpoint CPU cost (% of cores)	Mean latency overhead (%)	99p staleness (s)
10	18.7	12.4	2.1
20	12.9	8.3	2.6
40	8.4	5.1	3.1
80	5.1	3.4	4.2
160	3.6	2.7	6.0

Table 8 Ablation study of coordinated checkpointing components

Variant	Description	Normalized throughput	99p staleness (s)
Full design	Grouped + adaptive barriers + prioritization	1.00	3.1
No group coordination	Independent per-operator checkpoints	0.84	5.8
No barrier tuning	Fixed alignment strategy	0.91	4.4
No prioritization	FIFO checkpoint scheduling	0.95	3.8
Baseline system	Framework defaults	0.79	6.3

checkpoints capture only the mapping from input frontiers to sink commit tokens and the minimal metadata needed to ensure replay safety. They are cheaper because they avoid serializing large operator state; instead, they rely on the existing operator changelog mechanism to reconstruct state between full checkpoints.

In this scheme, there are two classes of recovery points. A full checkpoint persists operator state bases and required changelog segments [20]. An output-frontier checkpoint persists the current changelog offsets and the sink commit tokens, asserting that operator state can be reconstructed from the last full checkpoint plus the persisted changelog range. This requires that the changelog itself is durable and replayable. Many modern state backends already maintain such durability, either by logging mutations to a distributed log or by writing incremental diffs to object storage. The output-frontier checkpoint thus acts as a consistent cut over changelog positions and sink tokens, enabling the system to replay mutations and outputs deterministically to the same sink frontier after a failure. Since output-frontier checkpoints are smaller, they can be executed frequently to reduce durable staleness [21].

Watermark completeness is preserved by restricting output-frontier advancement to watermark-safe boundaries. For windowed operations, the job defines a lateness bound and a policy for late events, such as producing updates or routing late events to a correction stream. The coordinator chooses a commit watermark w_c such that w_c does not exceed the minimum watermark across relevant inputs minus the lateness bound. Operators only include outputs in the commit range that are stable under the chosen lateness policy. In the common case where late events result in updates to previously emitted aggregates, the sink must support upserts keyed by window and key, which is often feasible in marketplace materializations [**?**]. In the case where late events are handled by corrections, the durable frontier represents the boundary after which no further corrections will be produced for that range, which is a different but still monotonic notion of stability.

The protocol maintains a clear causal chain. Records are processed and update operator state. Operator state changes are logged in a durable changelog. Outputs are emitted to sinks with an associated token that identifies the input frontier that produced them [22]. Output-frontier checkpoints record the changelog offset and the highest token for which outputs have been durably committed. In the event of failure, the job restores from the most recent output-frontier checkpoint if available, loading the last full state base and replaying the changelog up to the recorded offset, then resuming output emission from the recorded sink token. Because the sink token is at most the token recorded in the checkpoint, the sink will not observe duplicate committed outputs, and any provisional outputs beyond the token that were previously staged but not committed can be safely discarded or overwritten.

Table 9 Sensitivity to marketplace load spikes

Load multiplier	Baseline 99p staleness (s)	Proposed 99p staleness (s)	Throughput change (%)
1×	6.3	3.1	+0.0
1.5×	9.8	4.2	-2.7
2×	15.4	6.7	-5.3
3×	28.1	11.3	-9.4
4×	42.7	18.9	-15.8

Table 10 Failure injection schedule during long-running experiment

ID	Time in run (min)	Failure type	Affected components
F1	15	Single worker crash	Session attribution task 07
F2	42	Rack network partition	Workers 08–11
F3	67	State backend node restart	Storage shard B
F4	103	Controller leader failover	Job manager
F5	128	Dual simultaneous worker loss	Fraud scoring tasks 03, 05

The marketplace setting motivates attention to multi-sink consistency. A pipeline may write derived views to a low-latency cache and to a durable analytical store [23]. The staleness experienced by consumers often depends on the slowest sink. The coordinator therefore tracks per-sink committed frontiers and can choose to advance them independently, while also optionally maintaining a global committed frontier for sinks that require cross-sink consistency. For example, if a personalization service reads from a cache but occasionally backfills from a database, inconsistent frontiers can cause transient anomalies. The strategy supports both modes by allowing the coordinator to commit sinks independently when isolation is acceptable, or to commit them under a shared barrier when consistency is required.

By separating full checkpoints from output-frontier checkpoints, using watermark-aligned commit boundaries, and explicitly acknowledging sink frontiers, the strategy aims to minimize the time between processing an event and making its derived effect durable [24]. This reduction in durable staleness is achieved without assuming that all sinks support heavyweight transactions at high frequency, because the protocol can rely on idempotent writes and frontier tokens, and it can adjust the frequency of full state snapshots to match storage constraints.

4. Adaptive Coordination and Scheduling for Minimal Staleness

A fixed checkpoint interval is rarely optimal in a marketplace pipeline because workload, skew, and watermark behavior vary over time. A job may experience diurnal traffic patterns, promotions that create bursts, and upstream incidents that increase event-time disorder. The proposed strategy therefore includes an adaptive scheduler that selects when to perform full checkpoints and when to perform output-frontier checkpoints, with the explicit goal of keeping durable staleness small while respecting resource budgets.

The scheduler observes metrics that can be collected with low overhead [25]. These include the current global watermark, per-operator lag in terms of unprocessed input offsets, buffer occupancy during alignment, snapshot throughput to durable storage, changelog growth rate, sink commit latency, and sink backpressure. In addition, the scheduler maintains an estimate of failure hazard, which can be derived from historical restart rates, planned maintenance windows, and infrastructure signals. The scheduling decision is made in terms of advancing a frontier, not merely triggering by wall-clock time. The scheduler aims to advance the durable frontier at a rate comparable to the processed frontier, such that the difference remains bounded.

A central control variable is a target staleness bound $S_{\max}$ expressed in event time. The scheduler attempts to ensure that the durable frontier lags the processed frontier by no more than $S_{\max}$ under normal conditions. When the job is healthy and re-

sources are available, it can schedule output-frontier checkpoints frequently enough to keep the bound small [26]. When resources are constrained, it may allow staleness to increase temporarily, but it does so with explicit accounting, and it prioritizes commits that reduce staleness most effectively.

The adaptive policy recognizes that the marginal cost of an output-frontier checkpoint is typically lower than the marginal cost of a full checkpoint, but it is not negligible. Output-frontier checkpoints still require barrier coordination to establish a consistent cut over changelog offsets and source positions, and they may still induce alignment delay. In addition, if output-frontier checkpoints are too frequent, sink commit overhead may dominate, especially for transactional sinks that incur per-transaction setup and commit costs. The scheduler therefore chooses a cadence that balances these costs [27].

One way to formalize this balance is to treat staleness as an accumulated penalty over time, while treating checkpointing overhead as a resource consumption penalty. The scheduler can then choose actions that minimize a weighted objective. The objective can be implemented as a receding horizon controller that evaluates a small set of candidate actions, such as performing an output-frontier checkpoint now, deferring it until the watermark advances by a certain amount, or performing a full checkpoint to reduce changelog length and future recovery cost. Because the job must remain stable under backpressure, the scheduler also monitors the risk of alignment amplification. If buffers are near capacity or skew is severe, initiating a barrier may increase latency disproportionately; in that case, delaying the barrier until the skew subsides may reduce overall staleness because it prevents a longer stall [28].

The scheduler also differentiates between operator regions. In marketplace pipelines, some regions may be dominated by stateless transformations and can coordinate quickly, while others may have heavy keyed state and require more time to snapshot. The scheduler can prioritize output-frontier advancement through lighter regions to deliver partial freshness improvements for certain sinks, while scheduling full checkpoints for heavy regions less frequently. This requires that the system can reason about which sinks depend on which regions, and it benefits from a region-based coordination mechanism that avoids forcing the entire job into a synchronized pause.

Event-time disorder influences frontier-based staleness because the watermark can stall [29]. If the watermark stalls, then advancing the durable frontier may not reduce event-time staleness in the strict watermark sense, because the processed frontier is also stalled. However, marketplace consumers often care about freshness for the majority of events, not just the worst-case out-of-order tail. The scheduler can therefore use a percentile-based notion of progress in addition to the watermark. For example, it can track the distribution of observed event-time delays and commit based on a high percentile boundary, while still providing correctness via late-event updates. This does not violate exactly-once semantics; it changes which outputs are considered stable at a given frontier by allowing updates [30]. The scheduler can increase commit frequency during periods where most events are in-order, and it can decrease frequency during periods of heavy disorder to avoid committing and then immediately updating the same keys frequently, which would increase sink write amplification.

Sink behavior is a major driver of the scheduling decision. Some sinks can absorb high-frequency commits with low overhead, such as append-only logs with idempotent producer sequences. Others, such as relational systems with strict transaction semantics, may have a minimum efficient transaction size. The scheduler therefore uses sink feedback to choose commit batching [31]. In particular, it can batch commits based on accumulated output volume, on elapsed time, or on frontier advancement. When sink commit latency increases, the scheduler can reduce commit frequency to avoid piling up concurrent transactions, but it must then account for increased staleness. This trade-off is explicit rather than incidental.

The scheduling policy also considers recovery exposure. If full checkpoints are infrequent, the changelog grows, and recovery requires replaying more mutations [32]. During a failure, durable staleness increases because the job is down, and once it restarts, it may take longer to catch up. In an e-commerce setting, recovery time affects operational visibility and can lead to temporary inconsistencies across services. The scheduler therefore ensures that full checkpoints occur often enough to bound changelog replay time, particularly during high-risk periods such as deployments. It can also trigger a full checkpoint opportunistically when the system is underutilized, thereby reducing future recovery cost without increasing staleness during busy periods.

The overall adaptive strategy is thus not a single fixed parameter but a control loop [33]. It measures processed frontier, durable frontier, alignment delay, snapshot throughput, sink commit behavior, and changelog growth. It chooses when to perform output-frontier checkpoints to keep durable staleness small, and it chooses when to perform full checkpoints to keep recovery cost bounded. The next section provides a mathematical model that makes these trade-offs explicit and that can guide parameterization for different marketplace workloads.

5. Mathematical Modeling and Guarantees

A mathematical model is useful for understanding how checkpoint frequency, commit cadence, and failures interact to determine expected output staleness. The model in this section is intentionally abstracted from any specific engine implementation, but it aims to capture the dominant mechanisms: frontier advancement, coordination overhead, snapshot cost, sink commit latency, and recovery under failures [34].

Let the processed frontier in event time at wall-clock time t be denoted $W_p(t)$. Let the durable frontier at the sink be $W_d(t)$. Define the instantaneous event-time staleness as

$$S(t) \quad = \quad W_p(t) \; - \; W_d(t). \tag{1}$$

In steady operation without failures, $W_p(t)$ typically increases over time as the job processes newer events and as watermarks advance [35]. $W_d(t)$ increases when the sink commits outputs corresponding to a frontier advancement. The staleness depends on how often and how far $W_d(t)$ is advanced relative to $W_p(t)$.

Model output-frontier commits as occurring at times $\{t_k\}$, with committed event-time increments $\Delta_k = W_d(t_k^+) - W_d(t_k^-)$.

Between commits, assume $W_d(t)$ is constant. For analytic tractability, approximate $W_p(t)$ as increasing at an average rate v in event-time units per wall-clock unit when the job is not stalled. The value of v depends on input rates and event-time mapping, but in many workloads, v is near one when event times are close to wall-clock, and it can be smaller during watermark stalls [36].

Let the effective coordination overhead per output-frontier checkpoint be c_o, measured as an average pause or slowdown time due to barrier alignment and metadata persistence. Let the sink commit time be c_s, measured as the time during which outputs are staged and then committed. In systems where processing continues concurrently with sink staging, c_s may not fully translate into a pause, but it can contribute to backpressure. Let the full checkpoint overhead be c_f, including incremental snapshot persistence and any compaction work, and suppose full checkpoints occur every T_f time units, while output-frontier checkpoints occur every T_o time units with $T_o \le T_f$ in typical configurations.

A basic steady-state approximation treats the durable frontier as advancing in event-time increments proportional to vT_o, but reduced by coordination overhead [37]. If coordination induces a pause of c_o per commit, then the net processing time between commits is approximately $T_o - c_o$, and the processed frontier increases by $v(T_o - c_o)$. If the commit advances the durable frontier to match the processed frontier at the barrier, then the staleness just after commit is small, and just before commit it grows approximately linearly with slope v over the interval. Under these approximations, the time-average staleness over one commit interval is

$$\mathcal{S} \approx \frac{1}{T_o}\int_{\varkappa}^{T_o} v\tau\, d\tau = \frac{vT_o}{}, \tag{2}$$

ignoring the effect of pauses and assuming the commit fully catches up. If the commit can advance only up to a watermark-safe frontier that lags the processed frontier by a bounded lateness allowance L, then the post-commit staleness may be at least L, yielding [38]

$$\mathcal{S} \approx L + \frac{vT_o}{}. \tag{3}$$

This highlights that even with frequent commits, late-event policy affects the achievable minimum staleness in the strict watermark sense.

Now incorporate failures. Model failures as a Poisson process with hazard rate λ per unit wall-clock time. Upon failure, the job is unavailable for a downtime duration D, which includes detection, restart, and state restoration. After restart, the job resumes from the latest recovery point [39]. In the proposed strategy, a recovery point may be an output-frontier checkpoint that references a full checkpoint base plus changelog offsets. Let R denote the recovery replay time required to restore state from the base and apply the changelog up to the recovery point, and let C denote the catch-up time needed to reprocess any inputs beyond the recovery point that were processed before failure but not included in it. If output-frontier checkpoints are frequent, C is small; if they are infrequent, C is larger.

During downtime and catch-up, the durable frontier may stop advancing or advance slowly. Output staleness grows not only because new events continue to occur externally, but also because the job cannot commit derived effects [**?**]. A simplified approximation treats staleness growth during downtime as proportional to elapsed wall-clock time mapped into event time at an external rate v_e, representing how event time progresses in the environment. In marketplace systems, v_e is often close to one in real time, though event time can be delayed by ingestion. Under this approximation, each failure contributes an additional staleness area roughly proportional to $v_e(D + R + C)$ times the magnitude of lag accumulated. For a renewal process view, the long-run average staleness can be decomposed into the steady-state staleness between commits and the incident-induced staleness due to failures.

A more explicit formulation treats the expected staleness as the expected value of $S(t)$ at a random time [**?**]. Under stationarity and using renewal reward arguments, one can approximate

$$\mathbb{E}[S] \approx L + \frac{vT_o}{} + \lambda\,\mathbb{E}[A_f], \tag{4}$$

where $\mathbb{E}[A_f]$ is the expected additional contribution per failure to the time-averaged staleness. The term $\lambda\mathbb{E}[A_f]$ reflects that more frequent failures increase average staleness.

To relate $\mathbb{E}[A_f]$ to checkpoint cadence, observe that the rollback exposure is bounded by the time between recovery points. If output-frontier checkpoints occur every T_o, then the maximum rollback in wall-clock time is approximately T_o and the expected rollback given a uniformly distributed failure time within an interval is about $T_o/$ when coordination delays are ignored. Translating rollback to event-time frontier rollback gives approximately $vT_o/$. This rollback implies that some outputs that were processed but not durably committed will be recomputed. If sink writes are idempotent, recomputation does not create incorrect duplicates, but it delays durable advancement [40]. The additional staleness due to rollback and catch-up can be approximated by

$$\mathbb{E}[A_f] \approx v_e(D + R) + \alpha\, v\frac{T_o}{}, \tag{5}$$

where α captures how quickly the system catches up relative to real time, incorporating spare capacity and backlog processing. When the job has headroom, α can be less than one because catch-up is faster than normal processing; when the job is saturated, α can exceed one. The key observation is that shorter T_o reduces rollback exposure and therefore reduces failure-induced staleness.

However, shorter T_o increases overhead [**?**]. Let the resource cost rate due to output-frontier checkpoints be approximately c_o/T_o and due to full checkpoints be c_f/T_f, with additional sink transaction overhead terms. If the system has a resource budget B for checkpoint-related overhead, one can impose a constraint

$$\frac{c_o}{T_o} + \frac{c_f}{T_f} \le B, \tag{6}$$

where B is measured in a normalized fraction of available compute or I/O capacity. The scheduler then chooses T_o and

T_f to minimize expected staleness subject to this constraint and sink limitations such as maximum commit rate.

Alternatively, one can formulate a weighted objective that penalizes staleness and overhead simultaneously: [**?**]

$$J(T_o, T_f) \quad = \quad \mathbb{E}[S] \; + \; \beta \left(\frac{c_o}{T_o} \; + \; \frac{c_f}{T_f} \right), \tag{7}$$

where β converts overhead into staleness-equivalent units based on operational priorities. Substituting the approximations yields an expression with competing terms in T_o. A representative form is

$$J(T_o, T_f) \quad \approx \quad L \; + \; \frac{vT_o}{} \; + \; \lambda \left(v_e(D+R) \; + \; \alpha v \frac{T_o}{} \right) \; + \; \beta \frac{c_o}{T_o} \; + \; \beta \frac{c_f}{T_f} \tag{8}$$

Holding T_f fixed, the dependence on T_o includes a linear term in T_o and an inverse term in T_o. Differentiating with respect to T_o and setting to zero yields an approximate optimum: [41]

$$\frac{\partial J}{\partial T_o} \quad = \quad \frac{v}{} (\; + \; \lambda\alpha) \; - \; \beta \frac{c_o}{T_o} \quad = \quad \varkappa, \tag{9}$$

$$T_o^{\star} \quad = \quad \sqrt{\frac{\beta c_o}{v(\; + \lambda\alpha)}}. \tag{10}$$

This expression has an intuitive interpretation. When the processed frontier advances quickly in event time, or when the failure hazard rate is high, the optimal commit interval decreases. When the coordination overhead per commit is high, the optimal interval increases. The parameter β captures how willing the system is to spend resources to reduce staleness. The form resembles classical checkpoint interval results but is shaped by the explicit inclusion of staleness as a primary objective and by the separation between output-frontier checkpoints and full checkpoints [42].

The model can be refined to account for sink constraints. Suppose the sink can sustain at most $r_{\max}$ commits per unit time, then $T_o \geq /r_{\max}$. If the sink commit cost depends on batch size, then c_s and the effective staleness may change with T_o. In a transactional sink, smaller commits may reduce staleness but increase overhead due to per-transaction fixed costs, effectively increasing c_o. In an idempotent upsert sink, frequent commits may be cheaper but can increase write amplification when late events trigger updates. These effects can be modeled by letting c_o be a function of T_o and by letting L depend on the lateness handling policy [**?**].

Correctness guarantees follow from the relationship between recovery points and sink commits. The system ensures that any sink commit token is recorded in a durable output-frontier checkpoint that references a full checkpoint base and a changelog offset. On recovery, the system restores state to that base, replays the changelog to the recorded offset, and resumes output emission from the recorded token. Because the sink committed frontier does not exceed the recorded token, the sink never observes a committed output that cannot be reproduced by replay from the recovery point. This establishes exactly-once durability with respect to the committed frontier [43]. If the sink supports transactional staging, the prepare and commit operations ensure that partial writes beyond the recorded token are not visible. If the sink uses idempotent writes, deterministic keys ensure that re-emitted outputs overwrite or duplicate harmlessly without changing the final value.

In addition, watermark-aligned commits ensure that committed outputs correspond to a consistent event-time boundary under the defined lateness policy. This provides a form of semantic completeness guarantee: for event-time windows, committed results up to the committed watermark are stable under the system's late-event handling rules. The specific meaning of stability depends on whether late events produce updates, but the committed frontier remains monotonic, which supports downstream reasoning about freshness [44].

6. Implementation Considerations and Evaluation Methodology in Marketplace Pipelines

Implementing the strategy in a marketplace environment requires attention to state representation, changelog durability, sink connector behavior, and operational isolation. Marketplace event schemas are heterogeneous. Some streams are append-only logs of immutable facts, such as order creation events. Others represent change-data-capture from databases, such as listing edits, inventory adjustments, and fulfillment status updates. User interaction streams can be noisy and out-of-order, and they often include deduplication identifiers [**?**]. A practical implementation must handle these differences while preserving deterministic recovery and minimizing output staleness.

State backends must support incremental snapshots and durable changelogs. A common approach is to store keyed state in an embedded key-value store and to log mutations to a distributed log or to object storage. The changelog must be ordered and partitioned in a way that aligns with operator parallelism, and it must be replayable on recovery. Compaction is required to prevent unbounded growth [45]. The full checkpoint then includes a compacted base plus a pointer to the changelog segment that follows it. Output-frontier checkpoints include the changelog offsets per operator instance and the sink commit tokens. In a multi-tenant cluster, changelog I/O must be throttled to avoid interfering with primary processing, which suggests the use of prioritized I/O scheduling and backpressure-aware compaction.

Barrier coordination must be integrated with the engine's existing flow control mechanisms. If barriers are injected too aggressively, they can interact with backpressure and cause alignment stalls [**?**]. Region-based coordination can reduce this risk by limiting barrier scope. Implementing regions requires identifying boundaries where state handoff can be checkpointed independently, such as repartitioning operators or shuffle edges. The engine must ensure that region boundaries propagate consistent tokens so that output-frontier checkpoints can be assembled without a full global stop-the-world alignment. This can be achieved by associating each data record with a token representing its originating frontier and ensuring that tokens are monotonic along each edge.

Sink connectors require careful design because sink hetero-

geneity is common in marketplace systems [46]. Some sinks are logs used for downstream streaming jobs. Others are key-value stores used for serving. Others are analytical stores used for reporting and experimentation. The commit protocol should expose a uniform interface to the coordinator while allowing specialized behavior per sink. For a log sink with exactly-once producer semantics, the commit token can be the log offset, and committing can be as cheap as flushing producer buffers and recording sequence numbers [47]. For a key-value store with idempotent upserts, the commit token can be a frontier recorded in an auxiliary metadata table. For a relational sink with transactions, the connector must manage transaction lifecycles and ensure that transaction commit aligns with tokens. In all cases, the connector must be able to report the committed frontier and to stage writes beyond it without making them visible.

In marketplace pipelines, a common requirement is that some outputs be visible quickly even if they may later be corrected, while other outputs must be strictly consistent. For example, near-real-time demand signals used for ranking may tolerate rapid updates and occasional corrections, while financial reconciliation views require strict correctness [**?**]. The strategy can accommodate this by using different sinks or different tables within a sink. A low-latency sink can receive frequent idempotent updates and can advance its committed frontier quickly, while a strict sink can commit less frequently with heavier transactional guarantees. The coordinator tracks staleness per sink so operators can reason about which outputs are fresh and which are conservative.

The interaction with late events is central to marketplace workloads. Late events occur for legitimate reasons such as client retries and upstream batching [48]. If the pipeline computes windowed aggregates, it must decide whether to incorporate late events as updates. Updates increase sink write volume but can reduce the need to delay commits waiting for late events. In practice, many marketplace materializations already accept upsert patterns, such as maintaining inventory counts per listing. Therefore, a reasonable approach is to commit outputs according to a watermark that reflects a bounded lateness allowance, while allowing late events to trigger deterministic updates that are also idempotent. The durable frontier then represents the boundary after which the probability of further updates is small under the lateness bound, and it provides downstream consumers with an interpretable freshness marker [49].

Operational concerns include multi-tenancy and isolation. Checkpoint and commit activity should not cause noisy-neighbor effects that degrade unrelated jobs. The scheduler should account for cluster-level constraints, such as shared object storage bandwidth and shared sink capacity. This suggests implementing per-job budgets for checkpoint I/O and sink transaction rates, and adapting within those budgets. It also suggests coordination with cluster schedulers that can allocate burst capacity during low traffic periods, enabling opportunistic full checkpoints that reduce recovery risk [50].

An evaluation methodology for the strategy should measure durable staleness directly rather than relying solely on processing latency. A practical measurement is to attach frontier tokens to outputs and to record, at sinks, the latest committed token along with the wall-clock time of commit. By correlating this with source watermarks and input event times, one can compute event-time staleness trajectories. For marketplace workloads, it is also valuable to measure downstream impact proxies, such as the delay in inventory decrements appearing in serving caches, the delay in fraud score updates reaching action systems, and the delay in seller performance metrics appearing in dashboards. These proxies should be analyzed cautiously because they can be influenced by downstream caching and read patterns, but they provide practical context [**?**].

The evaluation should also measure overhead and stability. Relevant signals include CPU utilization during barrier alignment, memory usage due to buffering, object storage bandwidth consumption, changelog growth rates, compaction CPU cost, and sink write amplification. Since the strategy relies on frequent output-frontier checkpoints, it is important to quantify how much additional coordination overhead is introduced and whether it is offset by reduced snapshot overhead compared to frequent full checkpoints. Failure scenarios should be included, such as induced task failures, node restarts, and rolling upgrades, to measure recovery time and post-recovery catch-up behavior. In marketplace pipelines, planned deployments are frequent, so a key operational metric is the staleness experienced during and after a deployment, not only during steady state [51].

A realistic workload model should include skew and heterogeneous event types. For example, a small set of high-traffic listings and sellers can dominate state access patterns. A pipeline might join clickstream events with catalog snapshots, producing high write rates to per-listing aggregates, while order events produce lower frequency but higher importance updates to inventory state. The evaluation should include both balanced and skewed key distributions, varying out-of-order delay distributions, and varying sink characteristics. The goal is to understand how the strategy behaves across plausible marketplace conditions, rather than under a single synthetic benchmark [52].

In implementation, correctness validation is essential because the protocol introduces additional moving parts compared to a naive checkpoint-everything-and-commit-at-end approach. Determinism tests can replay recorded input streams and verify that sink outputs are identical across runs despite different failure injection points. Frontier monotonicity tests can verify that committed frontiers never regress and that sinks never expose outputs beyond the recorded committed token. Late-event tests can verify that updates converge to correct values under the defined lateness policy. These tests should be integrated into continuous validation pipelines because marketplace event schemas evolve and connector behavior can change [53].

7. Conclusion

This paper has described a coordinated checkpointing strategy for stateful stream processing in e-commerce marketplace event pipelines with a focus on minimizing durable output staleness. The strategy separates the establishment of recovery-consistent cuts from the advancement of durable sink frontiers, enabling frequent, lightweight output-frontier checkpoints that reduce the time between processing an event and making its derived effect

stable in external sinks. Watermark-aligned barriers ensure that event-time semantics and lateness policies are respected when advancing commit boundaries, and sink-frontier acknowledgments allow the coordinator to observe and control end-to-end freshness explicitly.

A staleness-centric analytical model was introduced to clarify trade-offs between commit cadence, coordination overhead, snapshot cost, sink behavior, and failure hazard rates. The model suggests that an optimal output-frontier commit interval balances a linear staleness growth term against an inverse overhead term, with adjustments for failure-induced rollback exposure and catch-up dynamics [54]. While the model is simplified, it provides a basis for adaptive scheduling policies that respond to workload changes, skew, watermark behavior, and sink constraints.

Implementation considerations were discussed in the context of marketplace pipelines, emphasizing incremental state snapshots, durable changelog management, heterogeneous sink connectors, and operational isolation in multi-tenant environments. The evaluation methodology emphasizes direct measurement of durable frontier staleness, overhead signals, and behavior under failure scenarios, which are particularly relevant for operationally sensitive marketplace outputs such as inventory and risk signals.

The strategy does not remove fundamental limits imposed by late events, sink capabilities, and resource budgets. Instead, it provides a way to make the trade-offs explicit and controllable, enabling pipeline operators to tune for minimal output staleness while preserving deterministic recovery and predictable overhead [**?**].

Acknowledgments

We would like to thank the reviewers of this research for their helpful comments and suggestions.

References

[1] C. Jiang, J. wook Ahn, and N. Desai, "Environment transfer for distributed systems.," 1 2021.

[2] D. Pfleger and U. Schmid, "Sirocco - on knowledge and communication complexity in distributed systems.," 10 2018.

[3] C. Avasalcai, C. Tsigkanos, and S. Dustdar, "Resource management for latency-sensitive iot applications with satisfiability," *IEEE Transactions on Services Computing*, vol. 15, pp. 2982–2993, 9 2022.

[4] G. Hu, X. Yao, and Y. Yu, "Collie: Federated query processing via reinforcement learning," in *2025 IEEE International Symposium on Parallel and Distributed Processing with Applications (ISPA)*, pp. 1284–1291, IEEE, 10 2025.

[5] E. Feng, D. Feng, D. Du, Y. Xia, and H. Chen, "snpu: Trusted execution environments on integrated npus," in *2024 ACM/IEEE 51st Annual International Symposium on Computer Architecture (ISCA)*, pp. 708–723, IEEE, 6 2024.

[6] A. A. Hussein, A. AL-Zebari, N. Omar, K. J. Merceedi, A. J. Ahmed, N. O. M. Salim, S. S. Hasan, S. F. Kak, I. M. Ibrahim, H. M. Yasin, and A. A. Salih, "State of art survey for fault tolerance feasibility in distributed systems," *Asian Journal of Research in Computer Science*, pp. 19–34, 9 2021.

[7] R. Malik, S. Kim, X. Jin, C. Ramachandran, J. Han, I. Gupta, and K. Nahrstedt, "Mlr-index: An index structure for fast and scalable similarity search in high dimensions," in *International Conference on Scientific and Statistical Database Management*, pp. 167–184, Springer, 2009.

[8] S. Shankar, "The canonical controller for distributed systems," 1 2020.

[9] A. B. Chernyshev, V. Antonov, and Z. Mayransaev, "Unified sectoral criterion for the stability of distributed systems," in *Proceedings of X All-Russian Scientific Conference "System Synthesis and Applied Synergetics"*, pp. 351–355, Southern Federal University, 10 2021.

[10] K. Thonglek, C. Lee, H. Abe, A. Endo, T. Hirofuchi, R. Takano, K. Taniguchi, and S. Date, "Benchmarks for job scheduling in ultra-distributed systems," in *Proceedings of the 1st International Workshop on Middleware for the Computing Continuum*, pp. 24–29, ACM, 12 2023.

[11] R. R. Velarde, "Secure distributed systems over satoshi blockchains," 3 2021.

[12] N. V. Patel, "Selection bias in two-sided e-commerce marketplaces: A framework for propensity score matching implementation," *Journal of Business Intelligence Systems and Computational Social Science Applications*, vol. 11, no. 5, pp. 1–20, 2021.

[13] Z. Dong, C. Tang, J. Wang, Z. Wang, H. Chen, and B. Zang, "Optimistic transaction processing in deterministic database," *Journal of Computer Science and Technology*, vol. 35, pp. 382–394, 3 2020.

[14] Z. Zhao, M. Wu, H. Chen, and B. Zang, "Characterization and reclamation of frozen garbage in managed faas workloads," in *Proceedings of the Nineteenth European Conference on Computer Systems*, pp. 281–297, ACM, 4 2024.

[15] S. Dahdal, F. Poltronieri, A. Gilli, M. Tortonesi, R. Fronteddu, R. Galliera, and N. Suri, "Roamml: Distributed machine learning at the tactical edge," in *MILCOM 2023 - 2023 IEEE Military Communications Conference (MILCOM)*, pp. 33–38, IEEE, 10 2023.

[16] J. Barreiro-Gomez, *Distributed System Partitioning and DMPC*, pp. 179–192. Springer International Publishing, 8 2018.

[17] M. Han, R. Chen, W. Shen, H. Zhang, J. Yang, and H. Chen, "Real-time, work-conserving gpu scheduling for concurrent dnn inference," *ACM Transactions on Computer Systems*, vol. 44, pp. 1–42, 11 2025.

[18] P. Han, H. Li, G. Xue, and C. Zhang, "Distributed system anomaly detection using deep learning-based log analysis," *Computational Intelligence*, vol. 39, pp. 433–455, 4 2023.

[19] V. Vijaykumar, R. Chandrasekar, and T. Srinivasan, "An obstacle avoidance strategy to ant colony optimization algorithm for classification in event logs," in *2006 IEEE Conference on Cybernetics and Intelligent Systems*, pp. 1–6, IEEE, 2006.

[20] D. Schneider and B. Uekermann, "Efficient partition-of-unity radial-basis-function interpolation for coupled problems," *SIAM Journal on Scientific Computing*, vol. 47, pp. B558–B582, 4 2025.

[21] R. K. Mishra and S. R. Raman, *Introduction to PySpark SQL*, pp. 1–22. Apress, 3 2019.

[22] D. Sukhoplyuev and A. Nazarov, "Methods for analyse performance in distributed systems," *E3S Web of Conferences*, vol. 419, pp. 1029–01029, 8 2023.

[23] N. V. Patel, "Adaptive experimentation in marketplaces: Balancing exploration and revenue optimization," *Frontiers in Health Informatics*, vol. 11, pp. 760–786, 2022.

[24] C. M. Zurita and F. Assis, "Hybrid control strategies for greenhouse climate regulation: Pid, fuzzy, and neuro-fuzzy comparative implementation in temperate-dry crop systems," in *2025 XV Symposium on Computing Systems Engineering (SBESC)*, pp. 1–6, IEEE, 11 2025.

[25] S. He, C. Fu, G. Feng, J. Lv, and F. Deng, "Singular value manipulating: An effective drl-based adversarial attack on deep convolutional neural network," *Neural Processing Letters*, vol. 55, pp. 12459–12480, 10 2023.

[26] M. Dahlmanns, C. Sander, R. Decker, and K. Wehrle, "Secrets revealed in container images: An internet-wide study on occurrence and impact," in *Proceedings of the ACM Asia Conference on Computer and Communications Security*, pp. 797–811, ACM, 7 2023.

[27] J. F. Herculano, L. O. S. de Andrade, W. D. P. Pereira, and A. S. de Sá, "E-theta: An efficient and adaptive tdma approach for wireless body area networks," in *2024 XIV Brazilian Symposium on Computing Systems Engineering (SBESC)*, pp. 1–6, IEEE, 11 2024.

[28] V. N. S. Kiran, "Chaos engineering for building resilient distributed systems," *International Journal of Science and Research (IJSR)*, vol. 9, pp. 1678–1689, 3 2020.

[29] N. S. Chatharajupalli, R. Rotta, R. Karnapke, and J. Nolte, "Probing considered harmful: Leveraging rssi for link quality prediction," in *2025 IEEE 50th Conference on Local Computer Networks (LCN)*, pp. 1–9, IEEE, 10 2025.

[30] R. Chandrasekar and T. Srinivasan, "An improved probabilistic ant based clustering for distributed databases," in *Proceedings of the 20th International Joint Conference on Artificial Intelligence, IJCAI*, pp. 2701–2706, 2007.

[31] A. Girault, G. Gössler, R. Guerraoui, J. Hamza, and D.-A. Seredinschi, *FORTE - Monotonic Prefix Consistency in Distributed Systems*, pp. 41–57. Germany: Springer International Publishing, 5 2018.

[32] Z. J. Hamad and S. R. M. Zeebaree, "Recourses utilization in a distributed system: A review," 1 2021.

[33] A. Saleh, "Privacy-protection in cooperative distributed systems," 8 2018.

[34] H. Benítez-Pérez, J. L. Ortega-Arjona, P. E. Méndez-Monroy, E. Rubio-Acosta, and O. A. Esquivel-Flores, *Distributed Systems Modelling*, pp. 41–82. Germany: Springer International Publishing, 8 2018.

[35] J. Tang, X. Liang, and M. Huang, "Process migration and implementation in distributed system based on genetic algorithm," *Distributed Processing System*, vol. 3, 7 2022.

[36] V. Kale, *Distributed Systems*, pp. 159–182. CRC Press, 10 2018.

[37] N. V. Patel, "Applying synthetic control methods to address causal identification challenges in the ride-hailing industry," *Journal of Data Science, Predictive Analytics, and Big Data Applications*, vol. 8, no. 7, pp. 27–49, 2023.

[38] L. Su, X. Wang, and L. Wang, "A resilience analysis method for distributed system based on complex network," in *2021 IEEE International Conference on Unmanned Systems (ICUS)*, pp. 238–243, IEEE, 10 2021.

[39] A. Luntovskyy and D. Gütter, *Definition: Highly-Distributed Systems*, pp. 3–23. Springer International Publishing, 2 2022.

[40] M. Inoue and A. C. Badallo, *Parallel and Distributed Systems*. 2 2019.

[41] R. Chandrasekar, R. Suresh, and S. Ponnambalam, "Evaluating an obstacle avoidance strategy to ant colony optimization algorithm for classification in event logs," in *2006 International Conference on Advanced Computing and Communications*, pp. 628–629, IEEE, 2006.

[42] M. Huisman and C. Seceleanu, *ISoLA (1) - Verification and Validation of Concurrent and Distributed Systems (Track Summary)*, pp. 421–425. Germany: Springer International Publishing, 10 2020.

[43] N. Nischal, "Automated evaluation for distributed system assignments," 1 2023.

[44] V. Podile, N. Kulshrestha, S. Goswami, L. Durga, B. Rachanasree, T. P. Reddy, and P. S. Sarojini, "1the future of business management with the power of distributed systems and computing," 3 2024.

[45] S. Priyadarshini and S. Rodda, "Frequent subgraph mining by giraph distributed system," *International Journal of Engineering and Advanced Technology*, vol. 9, pp. 1267–1275, 6 2020.

[46] T. Srinivasan, R. Chandrasekar, V. Vijaykumar, V. Mahadevan, A. Meyyappan, and A. Manikandan, "Localized tree change multicast protocol for mobile ad hoc networks," in *2006 International Conference on Wireless and Mobile Communications (ICWMC'06)*, pp. 44–44, IEEE, 2006.

[47] M. Zakarya, L. Gillam, K. Salah, O. F. Rana, S. Tirunagari, and R. BUYYA, "Colocateme: Aggregation-based, energy, performance and cost aware vm placement and consolidation in heterogeneous iaas clouds," 6 2021.

[48] A. Miller, *Blockchain for Distributed Systems Security - Permissioned and Permissionless Blockchains*. Wiley, 3 2019.

[49] L. Gour and A. A. Waoo, "Fault tolerating mechanism in distributed computing environment," *International Journal of Engineering Applied Sciences and Technology*, vol. 5, pp. 610–615, 8 2020.

[50] S. Gorai, *Decentralization without Blockchains*, pp. 24–32. Productivity Press, 8 2024.

[51] M. Goudarzi, Q. Deng, and R. Buyya, *Resource management in edge and fog computing using FogBus2 framework*, pp. 17–52. The Institution of Engineering and Technology, 5 2024.

[52] R. Malik, C. Ramachandran, I. Gupta, and K. Nahrstedt, "Samera: a scalable and memory-efficient feature extraction algorithm for short 3d video segments.," in *IMMERSCOM*, p. 18, 2009.

[53] V. Anumolu, "Building secured microservices and distributed systems," *INTERNATIONAL JOURNAL OF INFORMATION TECHNOLOGY AND MANAGEMENT INFORMATION SYSTEMS*, vol. 16, pp. 717–739, 3 2025.

[54] I. B. Fink, L. Ferlemann, M. Dahlmanns, C. Thimm, and K. Wehrle, "Emulating and evaluating transport layer protocols for resilient communication in smart grids," in *NOMS 2025-2025 IEEE Network Operations and Management Symposium*, pp. 1–10, IEEE, 5 2025.