

# Streaming & Pipeline Processing

## Event Time, Windows, State, and Reliable Delivery

Diogo Ribeiro

January 14, 2026

# Learning Objectives

By the end, participants can:

- Explain event time vs processing time and watermarks.
- Choose windowing strategies and triggers for different problems.
- Design stateful operators and manage late/duplicate data.
- Reason about delivery guarantees and end-to-end exactly-once.
- Join streams and handle backpressure safely.
- Test, observe, and operate streaming pipelines in production.

# Agenda (5 min)

- ➊ Why Streaming? (10 min)
- ➋ Time Model: Event vs Processing Time & Watermarks (15 min)
- ➌ Windows, Triggers, Lateness (20 min)
- ➍ Stateful Processing & Delivery Guarantees (20 min)
- ➎ Joins, Backpressure, Fault Tolerance (15 min)
- ➏ Testing, Observability, Cost (10 min)
- ➐ Exercises & Wrap-up (10 min)

# Motivation (10 min)

- Many domains are **event-driven**: telemetry, payments, logs, sensors.
- Streaming yields **low latency** insights and reactive products.
- Unified batch+stream architectures reduce duplication and staleness.
- Cost/ops trade-offs: continuous compute vs scheduled batch.

# Event Time vs Processing Time (8 min)

- **Event Time:** when an event actually happened.
- **Processing Time:** when the system observed it.
- Networks, buffers, and clocks cause reordering and delay.

# Watermarks (7 min)

- A **watermark** asserts that most events older than  $T$  have arrived.
- Drives window completeness and trigger firing.
- Heuristics: fixed lag (e.g., `eventTime - 2 min`), source-derived, or adaptive.

## Timestamp Assignment (5 min)

```
source("clicks")  
  .assignTimestamp(event.ts)      // choose event time field  
  .watermark(event.ts - 120s)    // simple fixed-lag watermark  
  .map(...)  
  .sink("analytics")
```

# Windowing Strategies (8 min)

- **Tumbling**: fixed, non-overlapping ( $[0, 1)$ ,  $[1, 2)$ ).
- **Hopping/Sliding**: fixed, overlapping (size, slide).
- **Session**: data-driven gaps, per key.
- **Global**: unbounded with triggers (e.g., count-based).



## Triggers & Accumulation (7 min)

- **Triggers:** when to emit (on watermark, processing time, count, custom).
- **Accumulation:** discard, accumulate, or accumulate+retract.
- Re-emit updates as late data refines results.

- **Allowed Lateness:** accept events arriving after watermark by  $\Delta$ .
- Side outputs for too-late events (dead-letter, audit, or reprocess).
- Deduplication keys mitigate duplicates on retries.

## Windowed Aggregation (Pseudocode) (6 min)

```
source("purchases")  
  .assignTimestamp(event.ts)  
  .watermark(event.ts - 2m)  
  .keyBy(userId)  
  .window(tumbling, size=5m)  
  .aggregate(sum(amount))  
  .trigger(onWatermark)  
  .allowedLateness(1m)  
  .sideOutput("late")  
  .sink("user_spend_5m")
```

# Stateful Processing (8 min)

- Operator state: per-key or global.
- Timers: register callbacks for event/processing time.
- Snapshots/checkpoints persist state for recovery and exactly-once.

## Per-Key State Example (6 min)

```
// Rolling unique URLs per user with TTL
source("clicks")
  .assignTimestamp(ts).watermark(ts - 1m)
  .keyBy(userId)
  .process(state.set("urls", TTL=24h)) { e =>
    state.urls.add(e.url)
    emit(size=state.urls.count(), userId=e.userId)
  }
  .sink("unique_urls")
```

- **At-most-once**: may drop data, no duplicates.
- **At-least-once**: no data loss, duplicates possible.
- **Exactly-once** (effectively-once): no loss, no duplicates (with idempotence/transactions).

# Exactly-Once Pattern (*6 min*)

- Input: partition+offset or sequence IDs.
- Processing: transactional state snapshots (barriers/checkpoints).
- Output: idempotent/transactional sink with commit on checkpoint.

## Stream–Stream Joins (*8 min*)

- Requires windows + time alignment + per-key state.
- **Inner/Left/Right/Outer** joins over temporal intervals.
- Reordering late events: update/retract results as needed.



## Join Sketch (Pseudocode) (6 min)

```
source("orders").assignTimestamp(ts)
  .join(source("payments").assignTimestamp(ts))
  .onKey(orderId)
  .within(window=15m)           // temporal join bound
  .trigger(onWatermark)
  .emit(orderId, status)
  .sink("order_payment_join")
```

# Backpressure & Flow Control (7 min)

- When downstream is slower than upstream, buffers fill.
- Strategies: bounded queues, rate limiting, adaptive batching.
- Scale-out hotspots by key partitioning and rebalancing.

# Fault Tolerance (*8 min*)

- Checkpoints + replay from durable logs.
- Exactly-once sinks: two-phase commit or idempotent upserts.
- Stateless vs stateful recovery times and trade-offs.

# Testing Strategies (6 min)

- Deterministic test harnesses: feed records with timestamps.
- Golden files for windowed outputs under reordering/latency.
- Property-based tests: idempotence, associativity of combiners.

## Observability (4 min)

- Metrics: input/output rates, watermark lag, busy time, backlogs.
- Logs and event samples for debugging late/duplicate paths.
- Traces across source → processor → sink.

## Cost & Capacity (4 min)

- Cardinality drives state size; state drives memory and checkpoint cost.
- Window width and lateness increase storage and compute.
- Right-size partitions and autoscaling policies.

## Exercise 1: Window Choices (6 min)

**Scenario:** Rolling user activity dashboard.

**Task:** Pick window type (tumbling/hopping/session) and trigger policy.

**Deliverable:** One-slide justification (latency vs accuracy vs cost).

## Exercise 2: Late Data Plan (6 min)

**Scenario:** Mobile telemetry with flaky connectivity.

**Task:** Choose watermark, allowed lateness, and side-output policy.

**Deliverable:** Policy doc: how to reconcile late updates in downstream stores.



## Exercise 3: Exactly-Once Sketch (*6 min*)

**Scenario:** Deduct credits once per order.

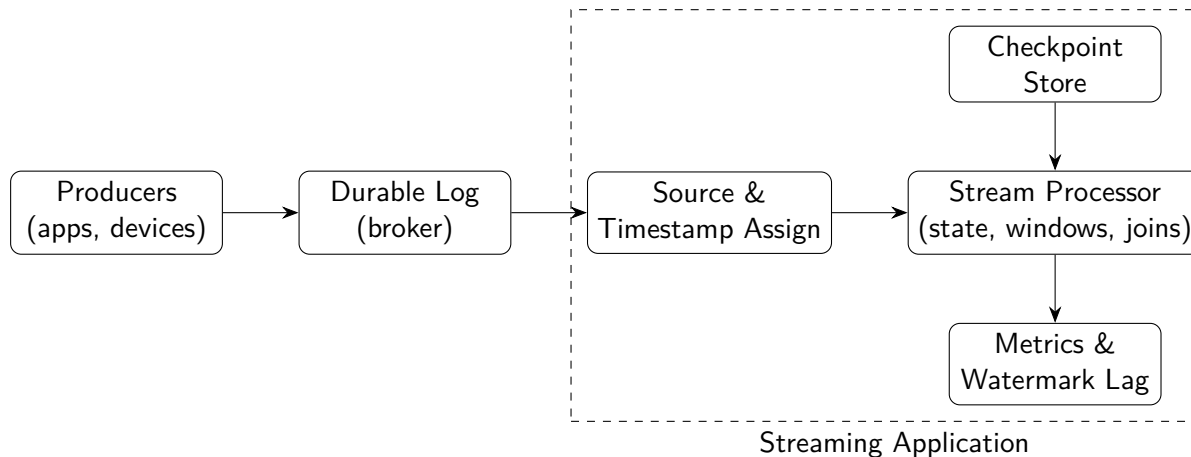
**Task:** Outline idempotent keys, checkpoint cadence, and sink commit flow.

**Deliverable:** Sequence diagram with failure/retry paths.

# Reusable Patterns (8 min)

- **Dead Letter Stream:** quarantine malformed or too-late events.
- **Outbox/Inbox:** transactional handoff between services.
- **Upsert Sink:** idempotent writes keyed by natural/business IDs.
- **Rekey/Compaction:** change partitioning without data loss.
- **Side Outputs:** route anomalies without blocking main flow.

# Reference Architecture (Conceptual) (6 min)



## Quick Quiz (6 min)

- ① Why prefer event time over processing time for aggregation?
- ② What role do watermarks play?
- ③ Compare tumbling, hopping, and session windows.
- ④ What is end-to-end exactly-once and how is it achieved?

## Key Takeaways (3 min)

- Treat time as data; design with watermarks and late events in mind.
- Windows + triggers define latency vs accuracy trade-offs.
- State and checkpoints enable correctness under failures.
- Backpressure and idempotence are operational essentials.

## Appendix: Glossary

Term	Meaning
Event Time	Timestamp of occurrence
Processing Time	Timestamp of observation
Watermark	Lower bound on unseen event time
Window	Temporal slice for grouping
Trigger	Rule for emitting results
Allowed Lateness	Grace period for late events
State	Persistent operator memory
Checkpoint	Consistent snapshot for recovery
Backpressure	Slower downstream throttles upstream

## Appendix: Timing Plan (90–120 min)

Segment	Minutes
Why Streaming	10
Time & Watermarks	15
Windows & Triggers	20
State & Guarantees	20
Joins & Backpressure	15
Testing/Observability/Cost	10
Exercises (3)	18
Quiz/Wrap	7
Buffer / Q&A	5–15

Thank you!