3.1 — Data Streaming at Scale

Data Processing at Scale

2026-03-27

Data Processing at Scale — Day 3

Course website

https://vbergeron.github.io/data-processing-at-scale/

This presentation

https://vbergeron.github.io/data-processing-at-scale/3.1-data-streaming-at-scale.pdf

Bounded vs unbounded

Bounded — a finite dataset

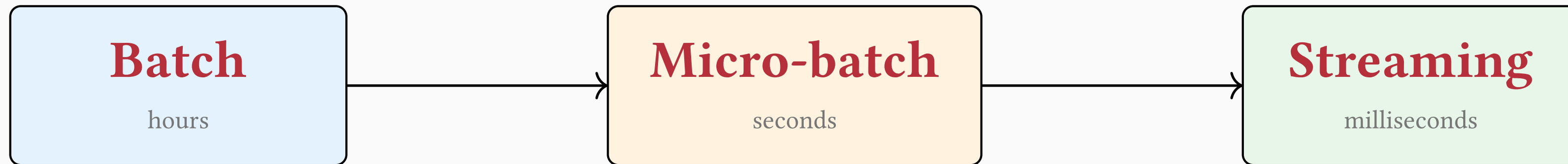
A file, a table snapshot, a CSV export.
You know where it ends before you start.
Processing terminates and produces a final result.

Unbounded — an infinite dataset

A live event feed, a sensor stream, a log tail.
There is no last record to wait for.
Results must be emitted continuously as data arrives.

The shift to streaming is a **semantic** choice — it changes when results are needed, not just how fast they are produced.

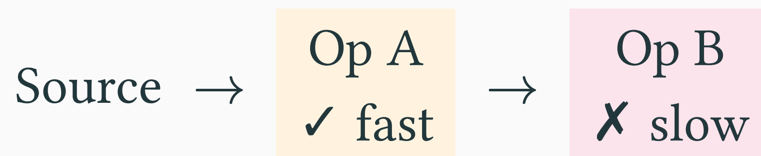
Batch ⟶ micro-batch ⟶ streaming



The trigger model sets a hard floor on achievable latency. No optimizer can reduce it below this value.

- **Batch**: scheduled jobs, MapReduce, Spark in batch mode
- **Micro-batch**: Spark Structured Streaming with timed triggers
- **Streaming**: Apache Flink, event-at-a-time with low-latency watermarks

Backpressure is how a slow consumer signals upstream producers to slow down.



Three strategies when a consumer can't keep up:

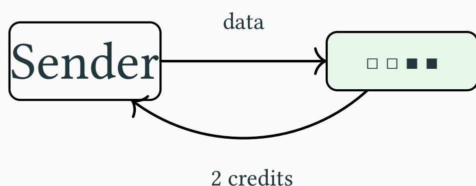
- **Drop**: discard excess records — data loss, at-most-once
- **Block**: pause the producer — safe, but stalls the whole pipeline
- **Buffer**: absorb bursts in a queue — useful for spikes, not sustained overload

Flink uses credit-based blocking — no separate signaling, pressure propagates through the network buffer layer.

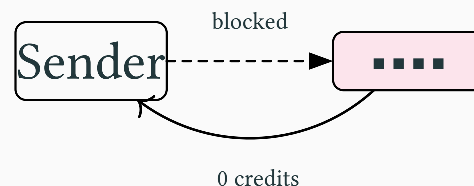
Credit-based flow control

Each downstream input channel has a fixed pool of buffers. The receiver grants one **credit** per free buffer slot. The sender can only transmit as many records as it holds credits.

Buffer has space — flowing



Buffer full — blocked



When the sender has no credits it blocks, its own input queue fills, and its upstream sender loses credits in turn — pressure propagates all the way back to the source with no dedicated signaling channel.

Push vs pull

Push — forward chaining

Source emits events as they occur.
Downstream operators react immediately.

Minimal latency.
Backpressure must be managed explicitly.

Flink, Kafka consumers, Storm

Pull — backward chaining

Downstream requests data when ready.
Upstream responds on demand.

Natural backpressure — consumer sets pace.
Higher latency — no data until asked.

Spark, SQL query engines

Monotonic computations

A computation is **monotonic** if adding more input can only extend the output — never retract or correct a previously produced result.

Monotonic	**Non-monotonic**
`COUNT(*)` — count only increases	`AVG()` — can decrease with new data
Set union — elements only added	Set membership deletion
`MAX`, `MIN` on append-only sources	Median — can shift in either direction

Monotonicity is the key to distribution: a monotonic computation converges to the correct result regardless of message arrival order.

The CALM Theorem

Consistency As Logical Monotonicity — Hellerstein & Alvaro, 2010.

A program has a consistent, coordination-free distributed implementation **if and only if** it is monotone.

Monotonic → no coordination needed	**Non-monotonic → coordination required**
Nodes may diverge temporarily, but will converge	Nodes can produce permanently wrong results without sync
`COUNT`, `UNION`, `MAX`	`DELETE`, `UPDATE`, `AVERAGE`

Design insight: encode non-monotonic intent as monotonic operations. A delete becomes an insert into a tombstone set. A correction becomes a versioned record.

Idempotency & determinism

Two properties that make distributed pipelines safe to retry.

Idempotent: applying the same operation multiple times produces the same result as applying it once.

"Write key K = V" is idempotent. "Increment counter by 1" is not.

Deterministic: given the same input, always produces the same output.

"Filter events where amount > 100" is deterministic. "Filter events from the last 5 seconds of wall-clock time" is not.

Both properties enable fault tolerance: a failed task can be retried or replayed from a checkpoint without corrupting downstream state.

Delivery guarantees

Guarantee	Meaning	Cost & use case
At-most-once	Events may be lost, never duplicated	No retries — metrics, monitoring, lossy telemetry
At-least-once	Events may be duplicated, never lost	Retries on failure — requires idempotent consumers
Exactly-once	Effect applied exactly once, end-to-end	Coordination overhead — checkpoints, transactions

"Exactly-once" does not mean an event is **processed** once. It means the **effect on downstream state** is applied once — achieved via idempotent sinks or two-phase commit, not by preventing retries.

The problem of time

Streaming data has two notions of time that are almost never the same.

Event time	**Processing time**
When the event **occurred** — embedded in the record	When the event **arrived** at the processor — wall clock
Correct for business logic ("all orders before midnight")	Simpler — no out-of-order records
Requires watermarks to know when a window is complete	No special mechanism needed
Reproducible on replay	Different result on replay

Network delays, retries, and mobile clients cause events to arrive **out of order**. A purchase made at 23:59 may arrive at the processor at 00:03. Processing time would assign it to the wrong day.

Windows

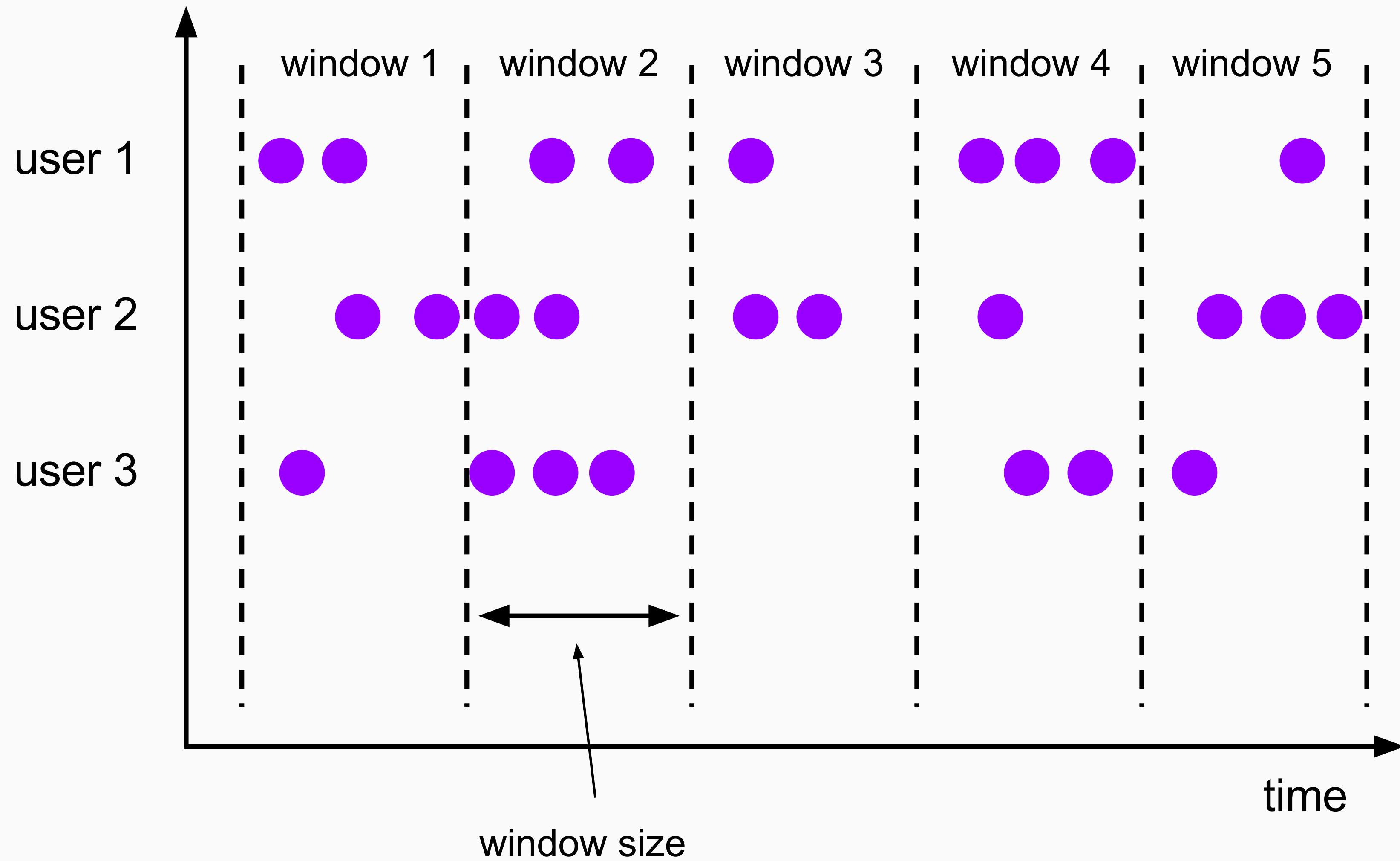
Aggregating an infinite stream requires bounding it into finite chunks

windows.

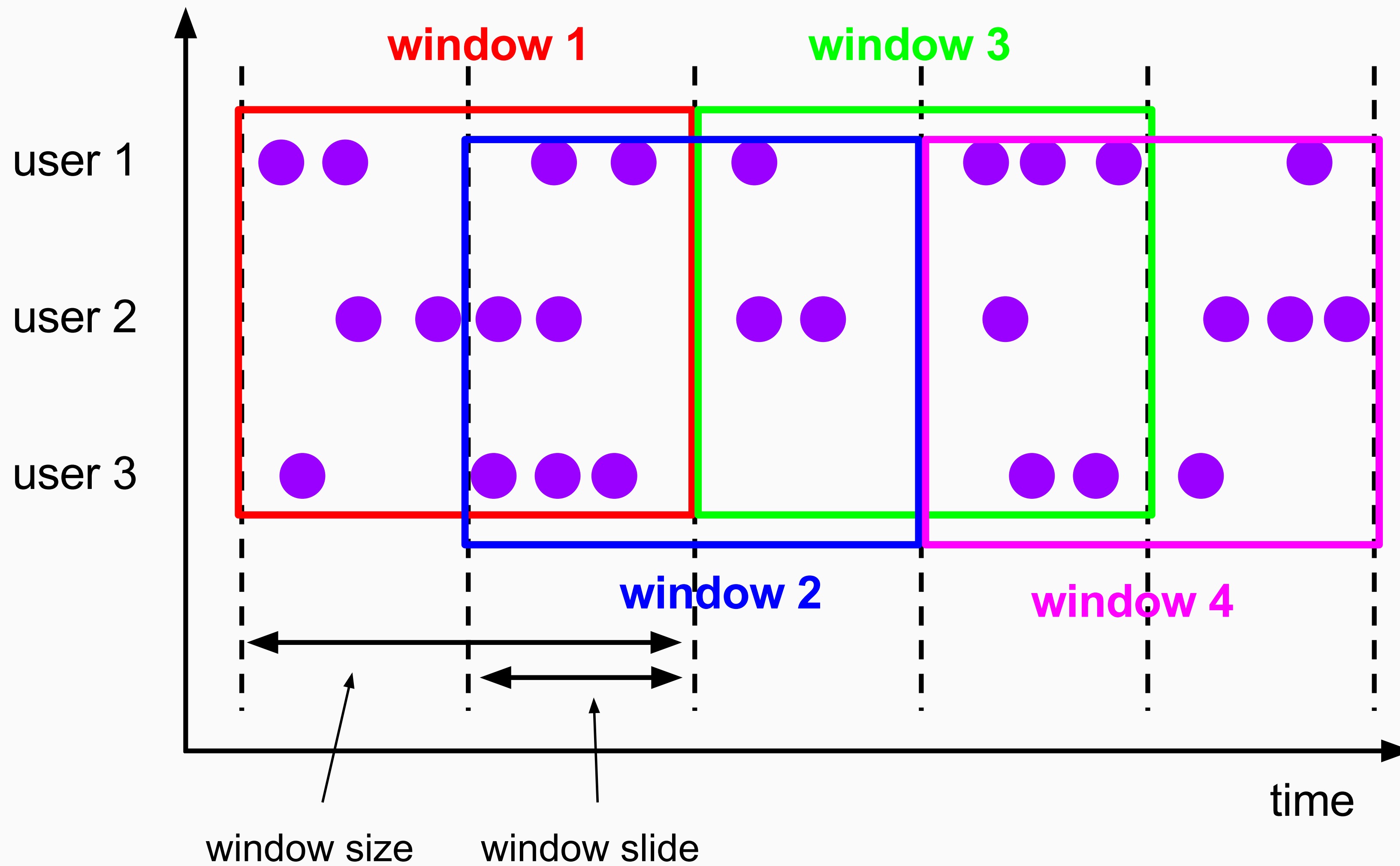
Without windows, a `COUNT(*)` on an unbounded stream never terminates. Windows let you ask: “how many events in the last minute?” instead of “how many events ever?”

Type	Definition
Tumbling	Fixed size, no overlap — each event belongs to exactly one window
Sliding	Fixed size, fixed slide interval — events can belong to multiple windows
Session	Gap-based — window closes after a period of inactivity per key

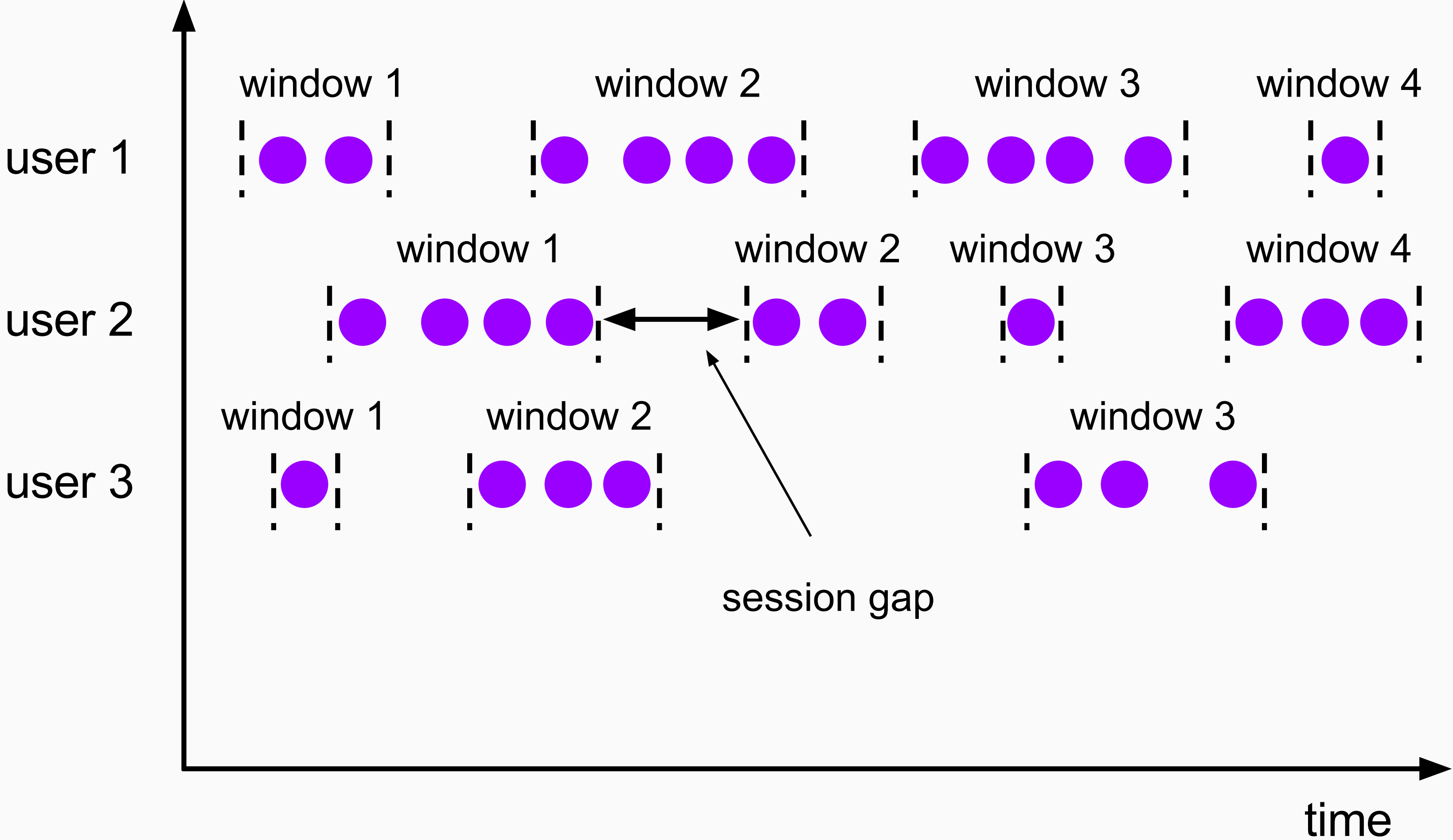
Tumbling windows



Sliding windows

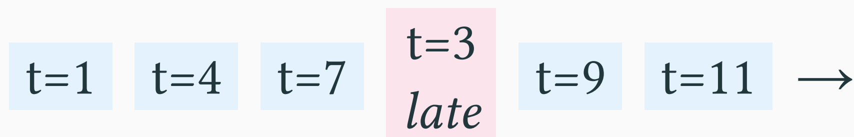


Session windows



Watermarks

On a finite dataset, you know when you've seen all the data. On a stream, you never do. A **watermark** is the system's best estimate: "all events with timestamp $t < W$ have now arrived."

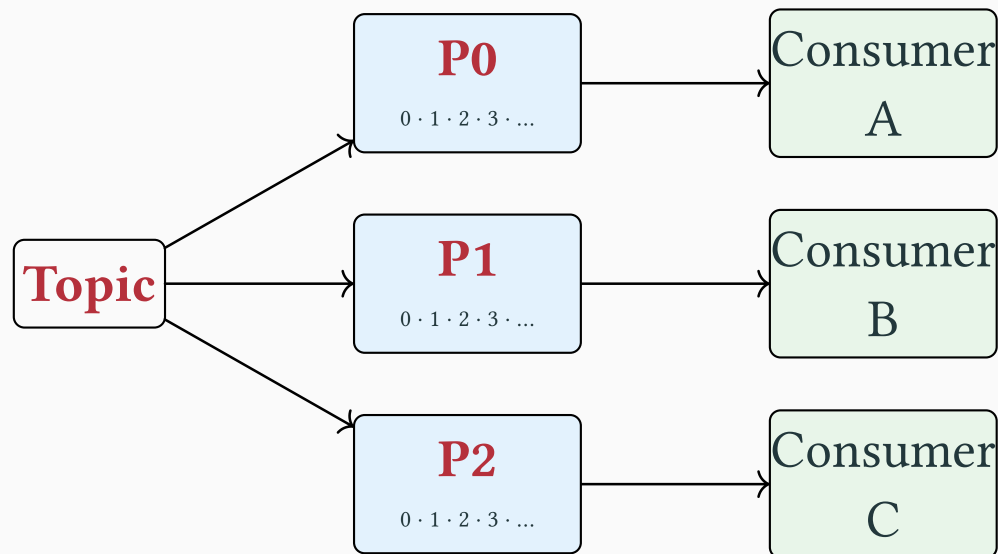


The watermark advances as: $W = \max(\text{observed event time}) - \text{tolerated lag}$.

When the watermark passes the end of a window, the window is **closed** and its result emitted. The tolerated lag is a tunable trade-off:

- Larger lag ⟶ more late events captured, higher result latency, more state held
- Smaller lag ⟶ lower latency, more events dropped as late

Kafka Recap



Each partition is an ordered, immutable log. Ordering is guaranteed **within a partition**, not across partitions. One consumer per group reads each partition — this is the unit of parallelism scaling.

The offset contract

The **offset** is a monotonically increasing integer that identifies each record within a partition.

- Producers append records; the broker assigns the next offset
- Consumers track their position by committing the last-read offset
- On restart, a consumer resumes from its committed offset

Delivery guarantees come from **when** the offset is committed:

- Commit **before** processing $\longrightarrow$ at-most-once (may lose the record if the consumer crashes)
- Commit **after** processing $\longrightarrow$ at-least-once (may reprocess if committed offset is lost)
- Commit **atomically with output** $\longrightarrow$ effectively exactly-once (requires idempotent producers + transactional API)

What Kafka does not do

Kafka guarantees **durability** and **ordering within a partition**. It does not:

- Transform events
- Join two streams
- Aggregate over time windows
- Track event-time watermarks
- Maintain keyed state across events

These are the responsibilities of a **stream processing framework** that sits downstream of Kafka. The rest of this session is about what those frameworks add.

Spark Structured Streaming

The Structured Streaming model

Structured Streaming treats the input as an **infinite append-only DataFrame**. Every new batch of records extends the table; the query runs continuously over it.

```
val events = spark.readStream
  .format("json")
  .schema(schema)
  .load("data/input/")

val counts = events
  .groupBy($"userId", window($"ts", "1 minute"))
  .count()

counts.writeStream
  .outputMode("append")
  .format("parquet")
  .option("checkpointLocation", "checkpoints/")
  .start()
```

The Structured Streaming model

The DataFrame API is unchanged — the same optimizer, same physical plans, same code generation.

Trigger modes

How often Spark wakes up to process new data:

Trigger	Behavior	Min latency
`ProcessingTime("1 minute")`	Micro-batch: collect all records arrived in the interval, then process	seconds
`Once`	Run exactly one micro-batch then stop — for scheduled jobs	batch
`AvailableNow`	Process all available data in one or more micro-batches, then stop	batch
`Continuous("1 second")`	Experimental low-latency mode; processes records as they arrive	ms

Continuous mode offers lower latency but supports fewer operations and is not production-ready for complex queries.

Output modes

What Spark writes to the sink on each trigger:

Mode	When to use
Append	Only new rows since last trigger — for queries with no updates (e.g., stateless filters, windowed aggregations after watermark)
Update	Only rows that changed since last trigger — for aggregations that produce updates
Complete	Rewrite the entire result table on every trigger — only for aggregations without a watermark

Not all modes are valid for all query types. Spark validates the combination at plan time.

Watermarks in Spark

Watermarks tell Structured Streaming how late data can arrive before a window is closed.

```
events
  .withWatermark("ts", "10 seconds")   // tolerate up to 10s of late arrival
  .groupBy(window($"ts", "1 minute"))
  .count()
```

- Spark tracks the maximum event timestamp seen so far
- The watermark advances to `max(ts) - threshold`
- Windows whose end time is below the watermark are finalized and emitted in Append mode
- Records arriving after their window's watermark are **dropped**

Watermarks enable exactly-once semantics in Append mode by bounding state size.

Where Spark Streaming fits

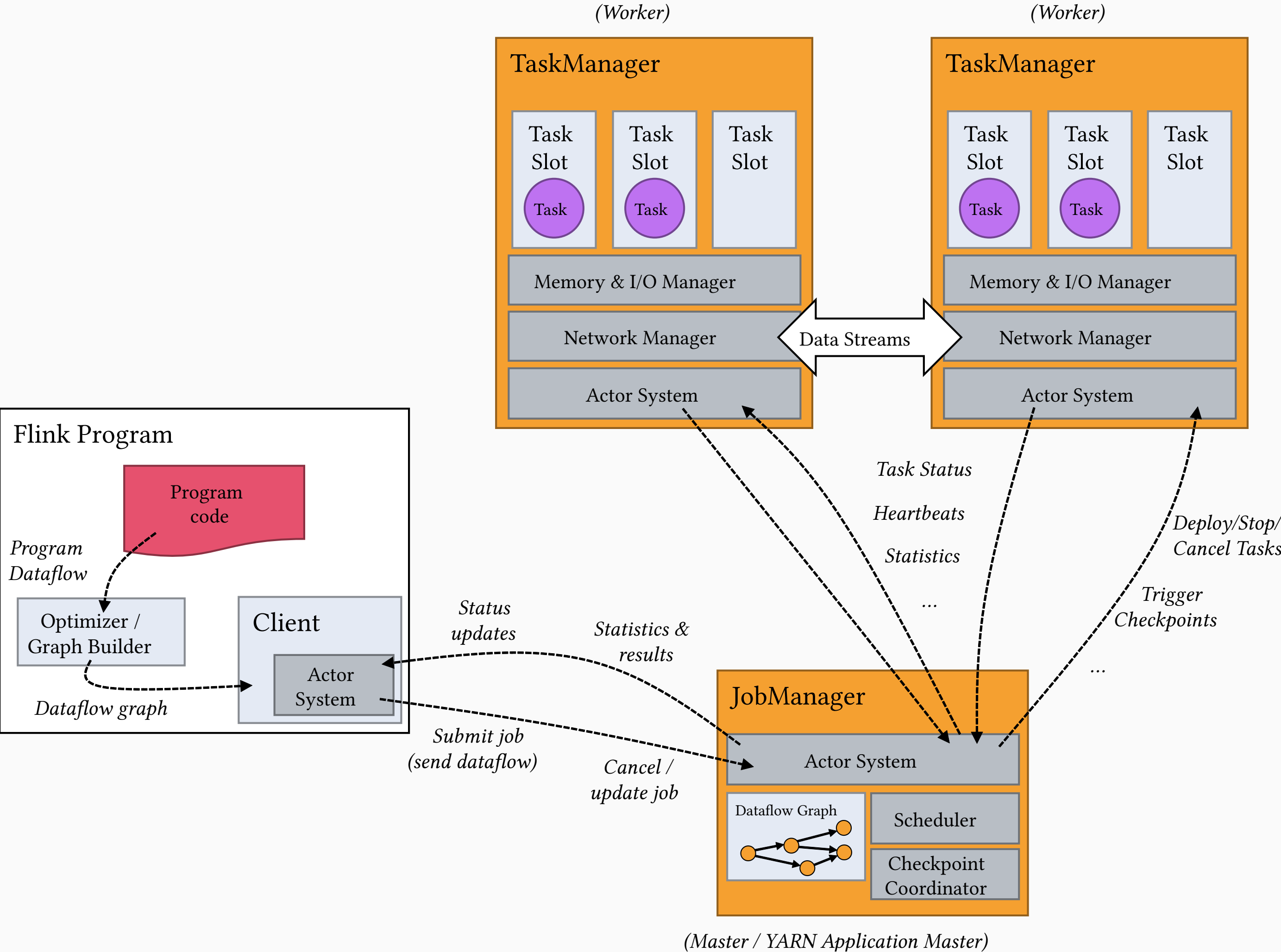
Good fit	**Poor fit**
Team already uses Spark for batch	Sub-second latency requirements
SQL-first pipelines with DataFrame API	Complex event time logic (arbitrary out-of-order)
Micro-batch latency (seconds) is acceptable	Fine-grained stateful processing per key
Unified batch + streaming codebase	Dynamic parallelism adjustment at runtime

Structured Streaming is a strong choice when you need **streaming semantics with a batch mindset**. When you need true event-at-a-time processing with full state control, Flink is the right tool.

Apache Flink

Apache Flink

Architecture



DataStream API

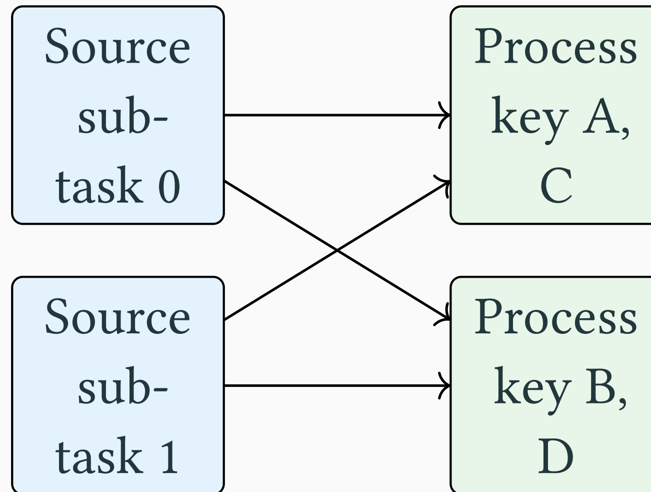
A Flink job is a **graph of operators** connected by data streams.

```
val env = StreamExecutionEnvironment.getExecutionEnvironment

val events: DataStream[Event] = env
  .fromSource(fileSource, WatermarkStrategy.noWatermarks(), "events")

events
  .filter(_.amount > 0)
  .keyBy(_.userId)
  .process(new CountPerUser())
  .print()

env.execute("user-count")
```

Nothing runs until `env.execute()` — Flink builds a job graph first, then submits it to the JobManager.

keyBy — partitioning a stream

`keyBy` hashes each record to a logical key and routes it to the sub-task responsible for that key.



All records with the same key land on the same sub-task — state for that key is always local, never remote. `keyBy` is the Flink equivalent of a shuffle, but records continue flowing without materializing to disk.

Watermark strategy

Before windowing, Flink needs to know **which field carries the event timestamp** and **how much out-of-order lag to tolerate**.

```
val strategy = WatermarkStrategy
  .forBoundedOutOfOrderness[Event](Duration.ofSeconds(5))
  .withTimestampAssigner((event, _) => event.ts)

val timestamped = events.assignTimestampsAndWatermarks(strategy)
```

- `forBoundedOutOfOrderness(d)` — advances the watermark to `max(seen_ts) - d`; records older than the watermark are considered late
- `withTimestampAssigner` — extracts the event-time field from each record
- A larger `d` ⟶ more correct results (fewer late records dropped), but higher latency

Windowed aggregation

With timestamps assigned, attach a window and an aggregation function.

```
timestamped
  .keyBy(_.userId)
  .window(TumblingEventTimeWindows.of(Time.minutes(1)))
  .aggregate(new SumAggregator())
```

- `.keyBy` — routes each record to the sub-task that owns its key (state stays local)
- `.window(...)` — choose the assigner: `TumblingEventTimeWindows`, `SlidingEventTimeWindows`, or `EventTimeSessionWindows`
- `.aggregate` / `.process` — called once per closed window; result flows downstream

Allowed lateness

The watermark closes a window — but late records can still trickle in. `.allowedLateness` keeps the window accumulator alive for a grace period and **re-emits** an updated result for each late arrival.

```
timestamped
  .keyBy(_.userId)
  .window(TumblingEventTimeWindows.of(Time.minutes(1)))
  .allowedLateness(Time.seconds(30))
  .sideOutputLateData(lateTag)
  .aggregate(new SumAggregator())
```

Two independent knobs:

Watermark lag	**Allowed lateness**
How long before the window closes	How long after closing to accept corrections
Controls first-result latency	Controls correction window
Records inside lag → included	Records inside lateness → trigger re-emit

Allowed lateness

After allowed lateness expires, any further late record goes to the **side output** — never silently dropped.

Side outputs

A side output is a secondary typed stream branching off any operator. The main path is unaffected.

```
val lateTag = OutputTag[Trade]("late-trades")

val result = trades
  .keyBy(_.symbol)
  .window(TumblingEventTimeWindows.of(Time.minutes(1)))
  .allowedLateness(Time.seconds(30))
  .sideOutputLateData(lateTag)
  .aggregate(new NotionalAggregator())

// Main output — on-time aggregated results
result.addSink(mainSink)

// Side output — truly late records, routed separately
result.getSideOutput(lateTag).addSink(lateAuditSink)
```

Side outputs

Side outputs are not limited to late data. Any `ProcessFunction` or `KeyedProcessFunction` can call `ctx.output(tag, value)` to split a stream by any condition — invalid records, high-value alerts, debug traces — without forking the pipeline.

Flink SQL

The DataStream API and Flink SQL compile to the same execution graph.

```sql
CREATE TABLE events (
  user_id STRING,
  amount  DOUBLE,
  ts      TIMESTAMP(3),
  WATERMARK FOR ts AS ts - INTERVAL '5' SECOND
) WITH ('connector' = 'filesystem', 'path' = 'data/input/', 'format' = 'json');

SELECT user_id, COUNT(*) AS cnt
FROM events
GROUP BY user_id, TUMBLE(ts, INTERVAL '1' MINUTE);
```

DataStream API vs Flink SQL

DataStream API	**Flink SQL**
Full control over operators and state	Declarative — less code, same optimizer
Custom `ProcessFunction`, side outputs	Limited to what SQL can express
Required for complex CEP and graph algorithms	Preferred for joins, aggregations, projections

Flink State & Fault Tolerance

Keyed state vs operator state

State in Flink is always scoped to an operator instance.

Type	Scope	Use case
Keyed state	One cell per key, per sub-task. Requires `keyBy`.	Per-user counts, running totals, session tracking
Operator state	One cell per sub-task. No key required.	Kafka partition offsets, broadcast config

Keyed state is the common case. Each key's state lives on the sub-task that owns that key — never a remote lookup.

State backends

Flink stores in-flight state in a pluggable **state backend**:

HashMapStateBackend	**EmbeddedRocksDBStateBackend**
JVM heap — state as Java objects	Local disk + block cache (RocksDB)
Fast random access	5–10× slower reads; incremental checkpoints
Limited by heap; GC pressure at scale	State can exceed available RAM
Default for development	Default for production at scale

`MapState` with RocksDB stores each map entry as a separate RocksDB key — only the accessed entry is deserialised. With `HashMapStateBackend` the entire map is deserialised on every access.

ValueState

A single typed cell per key. The simplest and most common state primitive.

```
class CountPerUser extends KeyedProcessFunction[String, Event, String] {
  lazy val count: ValueState[Long] = getRuntimeContext
    .getState(new ValueStateDescriptor("count", classOf[Long]))

  override def processElement(e: Event, ctx: Context, out: Collector[String]):
Unit = {
    val c = Option(count.value()).getOrElse(0L) + 1
    count.update(c)
    out.collect(s"${e.userId}: $c")
  }
}
```

- `.value()` — reads current value; returns `null` if uninitialised
- `.update(v)` — writes a new value
- `.clear()` — deletes the cell (frees memory; TTL can also do this automatically)

ListState

An ordered list of values per key. Flink serialises the list efficiently — no need to read-modify-write the full list for append-only patterns.

```
lazy val buffer: ListState[Event] =
  getRuntimeContext.getListState(
    new ListStateDescriptor("buffer", classOf[Event])
  )

// append without reading the full list
buffer.add(event)

// read all buffered events for this key
buffer.get().asScala.foreach(process)

// replace the entire list
buffer.update(newList.asJava)
```

Use case: buffer events within a custom window, then emit when a timer fires.

MapState

A key→value map per Flink key. Avoids deserialising the full structure when you only need one entry.

```
lazy val positions: MapState[String, Long] =
  getRuntimeContext.getMapState(
    new MapStateDescriptor("positions", classOf[String], classOf[Long])
  )

// per-symbol net quantity inside a per-user keyed operator
val qty = Option(positions.get(symbol)).getOrElse(0L)
positions.put(symbol, qty + delta)
```

- `.get(k)` — point lookup; returns `null` if absent
- `.put(k, v)`, `.remove(k)`, `.contains(k)`, `.entries()` — standard map operations
- With RocksDB backend, each entry is stored as a separate key — only the accessed entry is deserialised

MapState

Use case: per-user portfolio positions (`userId` ⟶ `Map(symbol → netQty)`).

State TTL

Without expiry, state for inactive keys accumulates indefinitely. `StateTtlConfig` evicts entries automatically.

```
val ttl = StateTtlConfig
  .newBuilder(Time.hours(24))
  .setUpdateType(StateTtlConfig.UpdateType.OnCreateAndWrite)
  .setStateVisibility(StateTtlConfig.StateVisibility.NeverReturnExpired)
  .cleanupInBackground()
  .build()

val descriptor = new ValueStateDescriptor("count", classOf[Long])
descriptor.enableTimeToLive(ttl)
```

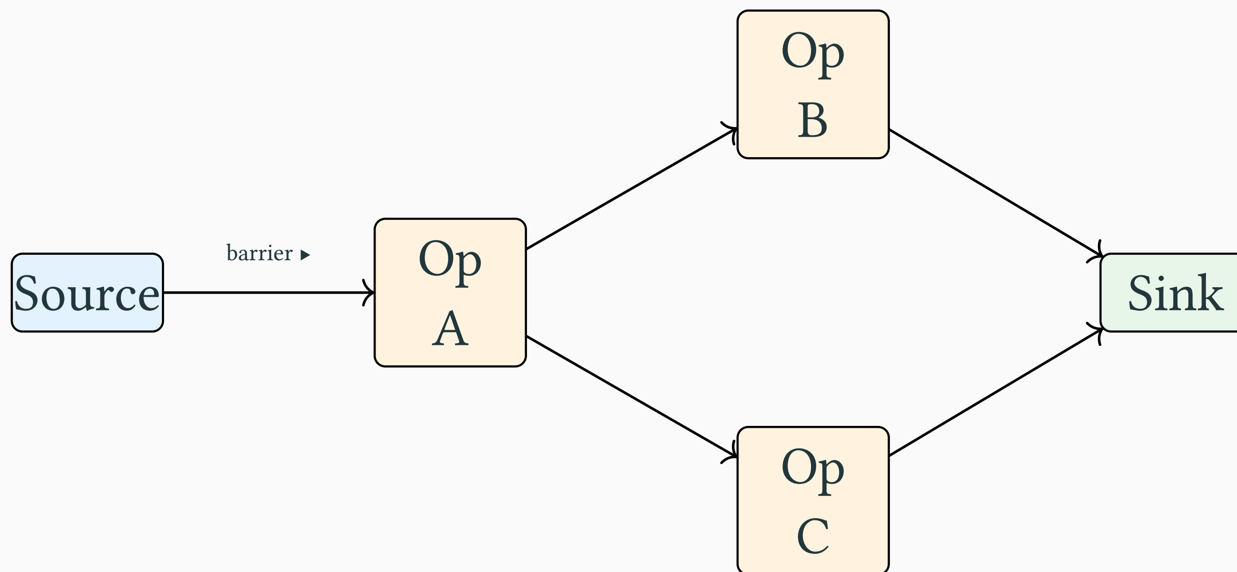
- `OnCreateAndWrite` — TTL resets on each `.update()` call; idle keys expire
- `NeverReturnExpired` — expired cells read as `null`, not stale data
- `cleanupInBackground()` — RocksDB compaction removes expired entries without a separate sweep

State TTL

TTL is the primary defence against unbounded state growth in long-running jobs.

Fault tolerance

Flink periodically takes a **consistent snapshot** of all operator state — a **checkpoint**.



The JobManager injects a **checkpoint barrier** into each source stream. Barriers flow with data; when an operator has seen a barrier on **every** input it snapshots its state and forwards the barrier. On failure, Flink resets all operators to the last completed checkpoint and replays from there.

Fault tolerance

Parallel inputs require barrier alignment — the operator buffers records from faster inputs until the barrier arrives on all channels, ensuring a globally consistent cut.

Checkpoint	**Savepoint**
Triggered automatically by Flink	Triggered manually: `flink savepoint <jobId>`
For failure recovery	For deliberate lifecycle events
Deleted when superseded	Retained indefinitely — you manage them
Opaque binary format	Portable — survives operator reordering with stable UIDs

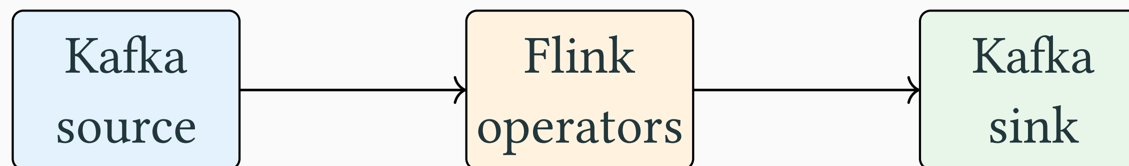
Zero-downtime upgrade with savepoints

1. Assign stable `uids` to all operators: `.uid("window-agg")`
2. Trigger a savepoint: `flink savepoint <jobId>`
3. Cancel the job
4. Deploy the new version: `flink run --fromSavepoint <path>`

State is restored key-by-key; operators that no longer exist are ignored; new operators start with empty state.

Exactly-once semantics

Checkpointing gives exactly-once for **internal state**. Making the **output** exactly-once requires sink cooperation.



- Source offsets are saved in the checkpoint — replay starts from the right position
- The Kafka sink uses **two-phase commit**: output is written in a transaction and committed only when the checkpoint succeeds
- Result: every record affects the sink **exactly once**, even across failures

Advanced Operators

Where to place logic

Flink operator class names are built from **adjectives** that compose independently.

Adjective	What it adds
(none)	Stateless transformation — `MapFunction`, `FlatMapFunction`, `FilterFunction`
`Rich`	`open()` / `close()` lifecycle — one-time initialisation (DB pool, ML model, config)
`Keyed`	Access to per-key state (`ValueState`, `ListState`, `MapState`) and event-time timers
`Co`	Two input streams joined at the operator level — both sides share the same state
`Broadcast`	One input is replicated to every parallel sub-task — used for shared rules or config

The adjectives stack: `KeyedBroadcastProcessFunction` = **Keyed** (per-key state) + **Broadcast** (shared config) + `ProcessFunction` (timers, side outputs).

Where to place logic

There is no fixed list of combinations

Where to place logic

`Rich` applies to any function, `Co` and `Broadcast` each add an input. Read the class name and you know exactly which capabilities are in scope.

Timers in KeyedProcessFunction

```
class InactivityAlert extends KeyedProcessFunction[String, Trade, Alert] {
  lazy val lastSeen: ValueState[Long] =
    getRuntimeContext.getState(new ValueStateDescriptor("lastSeen",
classOf[Long]))

  override def processElement(t: Trade, ctx: Context, out: Collector[Alert]):
Unit =
    lastSeen.update(ctx.timestamp())
    ctx.timerService().registerEventTimeTimer(ctx.timestamp() +
5.minutes.toMillis)

  override def onTimer(ts: Long, ctx: OnTimerContext, out: Collector[Alert]):
Unit =
    if ts == lastSeen.value() + 5.minutes.toMillis then
      out.collect(Alert(ctx.getCurrentKey, ts))
}
```

Timers in KeyedProcessFunction

- `registerEventTimeTimer(t)` — fires when the watermark advances past `t`
- `registerProcessingTimeTimer(t)` — fires at wall-clock time; not reproducible on replay
- Timers are checkpointed and survive failures

Serialization

Flink serializes records in two places: **between operators** (network shuffle) and **into checkpoints** (state). Getting it wrong costs performance or breaks savepoint compatibility.

Flink resolves a `TypeInformation` for every stream at compile time. Four tiers, in preference order:

Tier	When	Trade-off
Flink native	Primitives, strings, arrays, tuples	Fastest — zero-copy, no heap allocation
POJO / case class	All fields public or getter+setter, no-arg constructor	Efficient field-level serialization; schema evolution supported
Avro / Protobuf	Explicit annotation or registration	Schema evolution + cross-language interop
Kryo fallback	Type not recognised	Works, but slow — disables optimisations and breaks savepoints

Serialization

- **Diagnose Kryo at startup** — Flink logs a warning for every Kryo fallback. Treat these as errors in production.
- **Schema evolution** — adding a field to a POJO or Avro type is safe; removing or renaming breaks savepoint deserialisation. Kryo has no schema at all.

Async I/O

Enriching a stream with an external call (HTTP, DB) synchronously stalls the sub-task for every record. `AsyncDataStream` pipelines multiple requests concurrently.

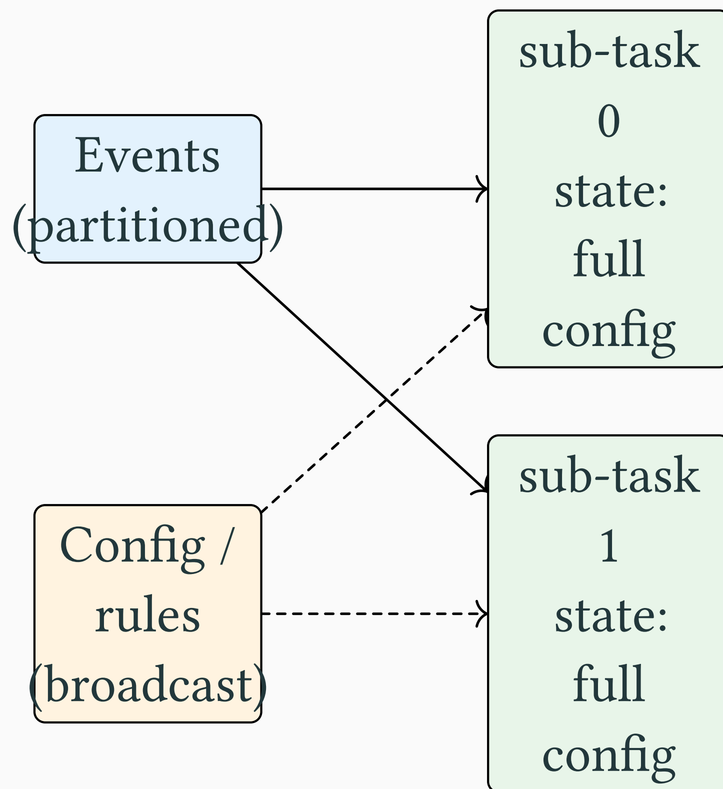
```
class AsyncPriceEnricher extends RichAsyncFunction[Trade, EnrichedTrade] {
  override def asyncInvoke(trade: Trade, result: ResultFuture[EnrichedTrade]):
Unit =
    priceService.getAsync(trade.symbol).thenAccept { price =>
      result.complete(List(trade.enrich(price)).asJava)
    }
}

AsyncDataStream.unorderedWait(
  trades,
  new AsyncPriceEnricher(),
  1000, TimeUnit.MILLISECONDS, // per-record timeout
  100                          // max concurrent requests
)
```

Async I/O

- `unorderedWait` — results emitted as soon as ready; maximum throughput
- `orderedWait` — results emitted in input order; higher latency

Broadcast state

Broadcast solves a specific problem: you have a stream of **events** (high-volume, partitioned) and a stream of **rules or config** (low-volume, changes rarely) that every sub-task needs a full copy of.



Broadcast state

```
val configDescriptor = new MapStateDescriptor(
  "account-config", classOf[AccountId], classOf[AccountConfig]
)
val broadcastConfig = configStream.broadcast(configDescriptor)

eventStream
  .connect(broadcastConfig)
  .process(new AccountMatchProcessor())
```

- `processBroadcastElement` — updates `BroadcastState`; called on config changes
- `processElement` — reads `BroadcastState`; called for every event

Keyed broadcast state

`KeyedBroadcastProcessFunction` adds **keyed state** on top of broadcast — each sub-task has both a full copy of the config and per-key state for the events it owns.

```
eventStream
  .keyBy(_.accountId)          // partition events by account
  .connect(broadcastConfig)
  .process(new NotificationProcessor())
```

Inside `NotificationProcessor`:

- `processBroadcastElement` — updates shared `BroadcastState` (config for all accounts)
- `processElement` — reads `BroadcastState` **and** keyed `ValueState` / `MapState` (state specific to this account)

This is the pattern used in chainwatch: `CommandRequest` messages are broadcast to all sub-tasks; `BlockEvent` records are keyed by `accountId`, so per-account state stays local.

Note: in `processBroadcastElement`, keyed state is **read-only**. Write only from the keyed side.

Unit-test a Flink operator in plain JUnit/munit with no cluster. The test harness wraps the operator, controls watermarks, and captures outputs.

```
// KeyedCoProcessFunction (two keyed inputs)
val harness = new KeyedTwoInputStreamOperatorTestHarness(
  new KeyedCoProcessOperator(new NotificationProcessor()),
  (e: BlockEvent.Matched) => e.account.id,
  (c: CommandRequest)     => c.account.id,
  TypeInformation.of(classOf[Account.ID])
)
harness.open()
harness.processElement1(StreamRecord(blockEvent))
harness.processElement2(StreamRecord(commandRequest))
harness.processWatermark(new Watermark(ts))   // advance event time, fire
timers
```

Testing operators — the harness

```scala
val out  = harness.extractOutputValues().asScala
val late = harness.getSideOutput(lateTag).asScala
```

For `BroadcastProcessFunction`, use `TwoInputStreamOperatorTestHarness` wrapping `CoBroadcastWithNonKeyedOperator` — same push/extract pattern, no key selector needed.

Testing operators — MiniCluster

For integration tests that need a full topology (source → operators → sink) or to verify checkpoint-restore behaviour, use `MiniCluster` — runs inside the JVM, no Docker.

```
//> using dep org.apache.flink:flink-test-utils:2.2.0

val env = StreamExecutionEnvironment.createLocalEnvironment(parallelism = 2)
env.enableCheckpointing(500)

val source = env.fromCollection(testEvents)
val result = CollectSink.values  // thread-safe static collector

source
  .keyBy(_.userId)
  .process(new CountPerUser())
  .addSink(new CollectSink())
```

Testing operators — MiniCluster

```
env.execute()
assertEquals(result.toSet, expected)
```

Use the harness for **operator logic** (fast, deterministic, no I/O). Use MiniCluster for **topology correctness** — parallelism effects, serialisation round-trips, checkpoint round-trips. Keep MiniCluster tests in a separate module; they are 10–100× slower.

Wrap-Up

Key vocabulary

Term	**Definition**
Unbounded dataset	A data source with no defined end — events arrive continuously
Micro-batch	Collect records over a time interval, then process as a mini-batch (Spark model)
Push / pull	Push: upstream emits when data arrives. Pull: downstream requests when ready.
Monotonic	A computation whose output only grows as input grows — can be distributed without coordination
CALM theorem	A program is consistent without coordination if and only if it is monotone
Idempotent	Applying the same operation multiple times produces the same result as once
Event time	When an event **occurred**, embedded in the record — correct basis for aggregations

Key vocabulary

Processing time	When an event **arrived** at the processor — wall clock, simpler but not reproducible
Watermark	A signal that all events with timestamp $t < W$ have arrived; advances event-time progress
keyBy	Routes records with the same key to the same sub-task — enables co-located keyed state
Keyed state	State scoped to a key within an operator sub-task; always local, never remote
Allowed lateness	Grace period after a window closes during which late records still update the result
OutputTag	A typed label for a side output stream — used to route late or exceptional records
State TTL	Automatic expiry of idle state entries; prevents unbounded state growth

Key vocabulary

Checkpoint barrier	A marker injected into the data stream to trigger consistent state snapshots
Savepoint	A user-triggered, portable checkpoint used for job upgrades and migrations
Exactly-once	The effect on downstream state is applied once — not a guarantee against retries
Backpressure	Upstream slows when downstream cannot keep up — propagated through network buffers
Broadcast state	A copy of a stream sent to every sub-task — used for shared config or rules
AsyncDataStream	Pipelines multiple async external calls concurrently to avoid blocking sub-tasks
Test harness	An in-JVM wrapper around a Flink operator for unit testing without a cluster

One sentence to remember

A streaming system is correct when it produces the same answer regardless of **when** events arrive — and the tools for that are event time, watermarks, and state backed by durable checkpoints.

One sentence to remember

Lab

https://vbergeron.github.io/data-processing-at-scale/lab-3.1-data-streaming-at-scale.pdf