CURRENT SAN FRANCISCO 2026

Kafka Streams Needs Real Timers

Hartmut Armbruster

Data streaming architect. Builds StoatFlow.

StoatFlow

One key, one time, one callback

A TIMER IS

1. a fundamental primitive of Flink's **DataStream API**
2. a callback, `onTimer()`, for **one key** at **one time**
3. **processing-time** timers: the machine's wall clock
4. **event-time** timers: fire when the watermark reaches them
5. timers are **deduplicated** and **checkpointed**, and can be **deleted**



APACHE FLINK: TEN YEARS OF TIMERS

- **2015** · 0.10: inside windows, for Flink's own operators
- **2017** · 1.2: for users, in `ProcessFunction`
- **2018** · 1.6: delete a timer; timers in RocksDB
- **2020** · 1.10: fired on the task thread
- **2025** · 2.1: named timers, in SQL
- **2026** · Confluent Cloud: SQL timers GA

WHAT TIMERS SOLVE

Flink builds its own operators on timers

Windows

`WindowOperator` arms a timer at each window's end, and one more to clean up its state. SQL window aggregations fire the same way.

Joins

`IntervalJoinOperator` drops each buffered row with a cleanup timer once the join's bounds have passed, in DataStream and in SQL.

CEP*

`CepOperator` buffers events by timestamp, arms a timer there, and replays them in order when it fires.

SQL functions

Process Table Functions (2.1) register named timers: registering a name again replaces the timer.

Your own

`KeyedProcessFunction`: state plus a timer. Next: a window, by hand.

In Flink, a window is state plus a timer, and all of these run on one `InternalTimerService` (read in the Flink `release-2.2` source).

* CEP: complex event processing, pattern matching across events (Flink CEP; `MATCH_RECOGNIZE` in SQL).

Event-time windowing with a timer, in Flink

```java
private ValueState<Long> sum;  // one sum per key

public void processElement(Tuple2<String, Long> in, Context ctx,
    Collector<Tuple2<String, Long>> out) throws Exception {
  if (sum.value() == null) {  // the window's first record: a timer at its end
    long end = ctx.timestamp() - Math.floorMod(ctx.timestamp(), MINUTE) + MINUTE;
    ctx.timerService().registerEventTimeTimer(end);
    sum.update(in.f1);
  } else {
    sum.update(sum.value() + in.f1);
  }
}

public void onTimer(long end, OnTimerContext ctx,
    Collector<Tuple2<String, Long>> out) throws Exception {
  out.collect(Tuple2.of(ctx.getCurrentKey(), sum.value()));  // the watermark passed the end
  sum.clear();
}
```

Adapted from Alibaba Cloud's **Best practices for timers in DataStream**, in Flink 2.x form: "The window API and a custom timer can produce equivalent results."

WHAT TIMERS SOLVE

Heartbeat timeout in Flink

```java
private ValueState<Long> lastTimer;  // (1) kept only to delete it

public void processElement(String device, Context ctx,
    Collector<String> out) throws Exception {
  Long previous = lastTimer.value();
  if (previous != null)
    ctx.timerService().deleteProcessingTimeTimer(previous);
  long timeout = ctx.timerService().currentProcessingTime() + 60_000;
  ctx.timerService().registerProcessingTimeTimer(timeout);
  lastTimer.update(timeout);
}

public void onTimer(long ts, OnTimerContext ctx, Collector<String> out) {
  out.collect("Heartbeat timeout: " + ctx.getCurrentKey());  // (2)
}
```

> (1) "To delete a timer, the function must record the registration timestamp in state. Flink provides no API to query whether a timer is currently registered."

THE RUNTIME DOES THIS FOR YOU

- at most one timer per key and timestamp
- `onTimer` never runs alongside `processElement`
- timers are checkpointed with the state
- overdue processing-time timers fire on restore

Adapted from Alibaba Cloud's **Heartbeat timeout alert**, in Flink 2.x form, with `open()` and the pass-through trimmed. (2) The key comes back with `getCurrentKey()`.

What about Kafka Streams?

NO KIP, ONE COMMENT

No KIP proposes per-key timers, and no JIRA ticket asks for them. The closest is one comment:

> "In the end, it seems you don't really want to schedule a **punctuation** but a **timer**?"
>
> MATTHIAS J. SAX · KAFKA-13678 · 2022

The ticket was closed in 2026 as a duplicate of anchored punctuation (KIP-1146).

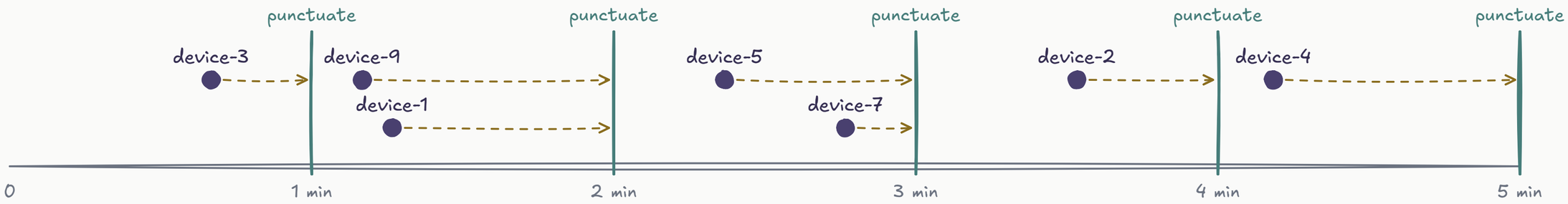
WHERE FLINK USES TIMERS, KAFKA STREAMS SCANS

- **Windows**: segmented stores. A window closes on a range scan and expires with its segment.
- **Joins**: an outer-join store, scanned from the front.
- **CEP**: not available.

Punctuators

```
context.schedule(Duration interval, PunctuationType type, Punctuator callback);
```

	STREAM_TIME	WALL_CLOCK_TIME
ticks on	the records the task processes	the system clock
first fire	on the first record	after one interval
a quiet task	never ticks	ticks
after a gap	one fire for the whole jump	one fire after the pause



One callback per processor, per task: no key.

Heartbeat timeout, the Kafka Streams way

```java
public void init(ProcessorContext<String, String> ctx) {
  super.init(ctx);
  lastSeen = ctx.getStateStore("last-seen");  // 1
  ctx.schedule(Duration.ofSeconds(10), WALL_CLOCK_TIME, this::sweep);
}

public void process(Record<String, String> r) {
  lastSeen.put(r.key(), r.timestamp());  // 2
}

private void sweep(long now) {  // 3
  try (KeyValueIterator<String, Long> it = lastSeen.all()) {
    it.forEachRemaining(e -> {
      if (e.value + TIMEOUT > now) return;
      context().forward(new Record<>(e.key, "timeout: " + e.key, now));
      lastSeen.delete(e.key);
    });
  }
}
```

1. a store and a key layout for **when**
2. an entry on every event
3. scan, decide, act, delete, on every tick
4. crash and restore edges: yours

The bill

CORRECTNESS

- a clock mix: record time stored, wall clock compared
- moving a deadline needs a second index, or a ghost fires
- fire, crash, re-fire under at-least-once

LIVENESS

- STREAM_TIME needs records: a quiet task never ticks
- the workaround in the wild: producers send control messages to every partition

COST

- `all()` per tick, per task, on the stream thread
- no record is processed while it runs

None of these is Kafka Streams' fault. It is what happens when every team writes the same mechanism by hand.

NEXT

A sketch for Kafka Streams

What would it take to ask for a timer, instead of emulating one?

SKETCH

Design goals and the API

Processor API only | additive | same task thread | exactly-once | stream time, no watermarks

```java
public interface TimerService<K> {
  void registerEventTimeTimer(K key, long timestamp);       // on stream time
  void registerProcessingTimeTimer(K key, long timestamp);  // on wall clock
  void deleteEventTimeTimer(K key, long timestamp);
  void deleteProcessingTimeTimer(K key, long timestamp);
  long currentStreamTimeMs();
  long currentSystemTimeMs();
}

// ProcessorContext / FixedKeyProcessorContext
<K> TimerService<K> timerService();

// Processor
default void onTimer(long timestamp, KIn key, TimerCallbackContext<KOut, VOut> ctx) {}
```

Explicit key: a Kafka Streams processor may change the key it forwards, and `init()` and punctuators have no current record.

SKETCH

Heartbeat timeout, rewritten

```
private void sweep(long now) { /* scan all(), decide, forward, delete */ }
```

```
public void process(Record<String, String> r) {
  Long prev = deadline.get(r.key());
  if (prev != null)  // move the deadline
    context.timerService().deleteProcessingTimeTimer(r.key(), prev);
  long next = context.currentSystemTimeMs() + TIMEOUT;
  context.timerService().registerProcessingTimeTimer(r.key(), next);
  deadline.put(r.key(), next);
}

public void onTimer(long ts, String device, TimerCallbackContext<String, String> ctx) {
  ctx.forward(new Record<>(device, "Heartbeat timeout: " + device, ts));
  deadline.delete(device);
}
```

No scan, no index, no delete loop. One store left, to move the deadline: the same as Flink.

SKETCH

Regular and named timers, and headers

> **A named timer** is one timer per key and name:
> register the name again and it moves to a new time.

	REGULAR	NAMED
Register	`registerEventTimeTimer(key, ts)`	`registerEventTimeTimer(key, "abandon", ts)`
Again	same key and time: still one timer	replaces it: a new time, or the other clock
Delete	`deleteEventTimeTimer(key, ts)`	`deleteTimer(key, name)` · `deleteAllNamedTimers(key)`
Look up	by time range via IQv2	`namedTimer(key, name)`: when, or `null`

Timer headers

```
context.timerService().registerProcessingTimeTimer(key, "retry", at, headers);  // your headers, upserted
Headers h = ctx.headers();                          // in onTimer: the arming record's headers, plus yours
ctx.forward(new Record<>(key, value, ts, h));  // pass them on: the trace id survives the wait
```

SKETCH

TopologyTestDriver

```java
var driver = new TopologyTestDriver(builder.build());
var in  = driver.createInputTopic("heartbeats", new StringSerializer(), new StringSerializer());
var out = driver.createOutputTopic("alerts", new StringDeserializer(), new StringDeserializer());

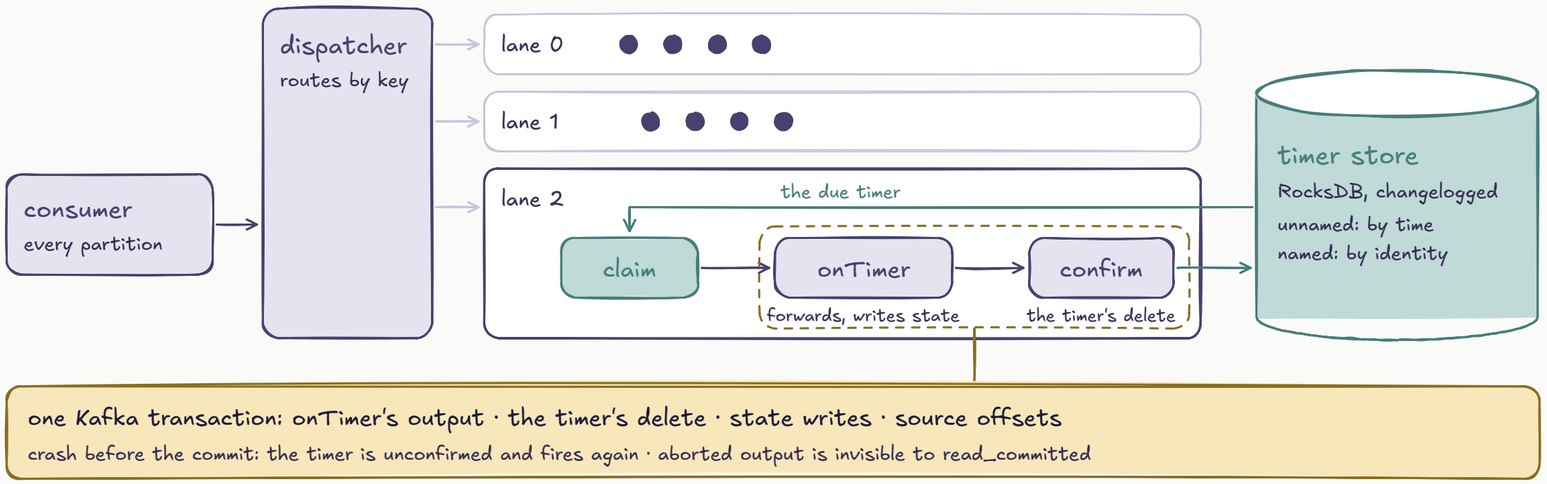
in.pipeInput("device-7", "ok");
driver.advanceWallClockTime(Duration.ofSeconds(30));
in.pipeInput("device-7", "ok");                          // moves the deadline
driver.advanceWallClockTime(Duration.ofSeconds(61));
assertEquals("Heartbeat timeout: device-7", out.readValue());
```

No new test API: `advanceWallClockTime` fires processing-time timers, as it fires wall-clock punctuators today, and piped records move stream time for event-time timers.

UNDER THE HOOD

StoatFlow* Timers

* A stream processor with the Kafka Streams DSL and Processor API, on a single JVM.



EXACTLY ONCE, BY DEFAULT

- a fire is **claim,** `onTimer`**, confirm**, on the key's lane
- the confirm commits with the output: **one transaction**
- crash before the commit: the timer **fires again**

Event time is the watermark. The minimum over live partitions, minus the out-of-orderness bound; a quiet partition drops out after the idleness timeout (5 min). Not Kafka Streams' per-task maximum.

Bounded timer cache. At most 100,000 timers per domain in memory, the rest in RocksDB, loaded in time order. Event time waits at the first timer not yet loaded, so none fires late.

Configure, inspect, measure

```yaml
stoatflow:
  processing-guarantee: EXACTLY_ONCE  # the default
  timer-backend:
    type: HYBRID                # or PERSISTENT, HEAP
    max-cached-timers: 100000   # per clock, in memory
    cache-horizon-ms: 60000     # HYBRID: load 60 s ahead
    changelog-enabled: true     # restore from Kafka
```

BACKEND	TIMERS LIVE	RESTART
HYBRID (default)	RocksDB; the next 60 s in memory	restored
PERSISTENT	RocksDB; the next second in memory	restored
HEAP	memory only: tests	lost

INSPECT

- `namedTimer(key, name)`: is one pending, and when
- `GET /watermarks`: is event time held by a timer
- `GET /topology/compiled`: which processors use timers
- timer IQ (IQv2): planned for 1.1.0

METRICS

- `stoatflow.timer.registered|fired|deleted.total`
- `stoatflow.timer.cache.loaded`
- `stoatflow.watermark.timer.held`

Event-time windowing with a timer

A one-minute sum per key, from state and one event-time timer. `windowedBy(...).aggregate(...)` does this for you.

```
1  public void process(Record<String, Long> r) {
2    long end = r.timestamp() - Math.floorMod(r.timestamp(), MINUTE) + MINUTE;
3    if (end <= context.timerService().currentWatermark()) return;  // 1 late: its window has fired
4    String window = r.key() + "@" + end;  // 2 state keyed by window
5    Long sum = sums.get(window);
6    sums.put(window, (sum == null ? 0 : sum) + r.value());
7    context.timerService().registerEventTimeTimer(r.key(), end);  // 3 one timer per (key, end)
8  }
9
10 public void onTimer(long end, String key, TimerContext<String, Long> ctx) {
11   String window = key + "@" + end;
12   ctx.forward(new Record<>(key, sums.get(window), end));
13   sums.delete(window);
14 }
```

2 Alibaba's version keeps one sum per key: a record for the next window that arrives before the watermark passes this window's end is added to this window.

3 Register on every record: duplicates collapse to one timer per key and timestamp.

1 Without it, a late record re-opens its window: the timer is already due, fires on the next pass, and emits a one-record window.

WHAT IT UNLOCKS

Built on top

Debounce

```
// process(): keep the latest value, and restart the quiet period
latest.put(r.key(), r.value());
context.timerService().registerProcessingTimeTimer(r.key(), "quiet", now + 5_000);
// onTimer(): emit the latest, once
ctx.forward(new Record<>(key, latest.get(key), ts));
```

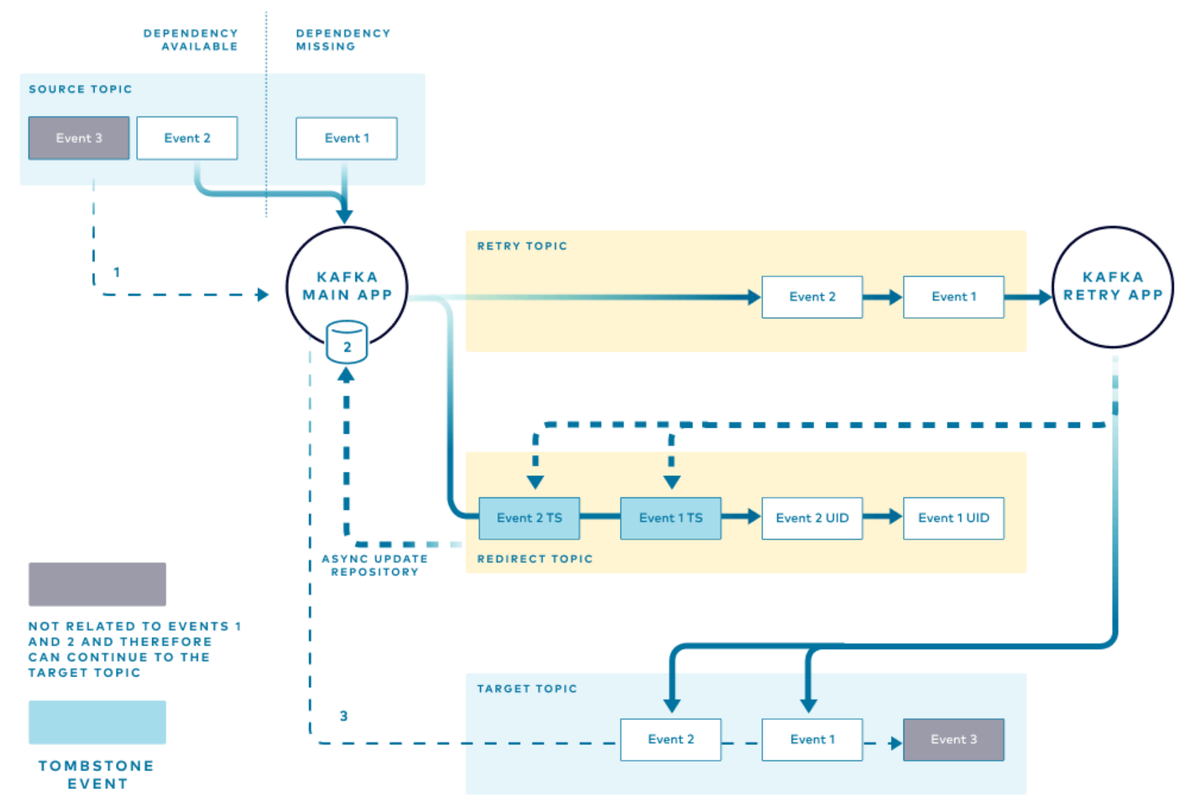
`within` for CEP (event-pattern matching)

A pattern must complete in time, or it times out: a payment within 10 min of checkout.

```
// process(): a partial match starts
context.timerService().registerEventTimeTimer(k, "within", earliestOpenDeadline(k));
// onTimer: expire matches past their deadline, emit them, re-arm for the next
```

WHAT IT UNLOCKS

Delayed retry, the Kafka way



A clean, well-known pattern:

- a **retry topic** and a second **retry app**
- a **redirect topic** and an **in-memory store**, to keep each item's order
- a tombstone clears an item once its retry lands

NOT IN IT

- a backoff that grows
- a budget of attempts
- a way to give up

Figure: Gerardo Villeda, **Error Handling Patterns for Apache Kafka Applications**, https://www.confluent.io/blog/error-handling-patterns-in-kafka/ Confluent blog, 2021, pattern 4. Its retries run "periodically until the conditions are met".

Delayed retry: backoff and a budget in the headers

```java
public void process(Record<String, String> r) {
  if (send.test(r.value())) return;
  boolean waiting = pending.get(r.key()) != null;
  pending.put(r.key(), r.value());  // the latest wins
  if (!waiting) arm(context.timerService(), r.key(), 1);
}
public void onTimer(long ts, String key, TimerContext<String, String> ctx) {
  int attempt = attemptOf(ctx.headers());  // (2) 1, 2, 3 ...
  String req = pending.get(key);
  if (send.test(req)) { pending.delete(key); return; }
  if (attempt < MAX_ATTEMPTS) arm(ctx.timerService(), key, attempt + 1);
  else { ctx.forward(key, req); pending.delete(key); }  // dead letter
}
static void arm(TimerService<String> t, String k, int n) {
  t.registerProcessingTimeTimer(k, "retry",  // (1)
      t.currentProcessingTime() + (BASE_MS << n),  // (3) 2 s, 4 s, 8 s ...
      new RecordHeaders().add("x-attempt", bytes(n)));  // (2)
}
```

(1) **Named**: one retry per key; registering again moves it.

(2) **Headers**: the attempt count rides along, and so do the arming record's headers, so the dead letter keeps its trace id.

(3) **No blocking**: the partition keeps flowing during the backoff.

WHAT IT UNLOCKS

Delayed retry in Flink: the state a timer can’t carry

```java
private ValueState<String> pending;   // the request
private ValueState<Integer> attempt;  // 1 StoatFlow: in the timer's headers
private ValueState<Long> retryAt;     // 2 StoatFlow: the timer's name
public void onTimer(long ts, OnTimerContext ctx, Collector<String> out) throws Exception {
  if (!Objects.equals(retryAt.value(), ts)) return;  // 3 a stale fire: no name replaced it
  if (send.test(pending.value())) { clear(); return; }
  int n = attempt.value();
  if (n < MAX_ATTEMPTS) { arm(ctx.timerService(), n + 1); return; }
  ctx.output(DLQ, pending.value()); clear();  // 4 a side output, no record headers
}
private void arm(TimerService timers, int n) throws Exception {
  Long old = retryAt.value();
  if (old != null) timers.deleteProcessingTimeTimer(old);  // replace by hand
  long at = timers.currentProcessingTime() + (BASE_MS << n);
  timers.registerProcessingTimeTimer(at);
  retryAt.update(at); attempt.update(n);
}
```

Two extra states per key 1 2, a stale-fire check 3, the trace id through the serdes 4. Compiled against Flink 2.2.1.

CLOSE

Questions for Kafka Streams

- Would you like to see a KIP?
- Is the Processor API the right home for an explicit key and a timer store?
- Stream time is the task's maximum, so an out-of-order record can arrive after its key's timer has fired. A grace period for timers, or watermarks?
- Named timers and headers: in the first version, or later?
- A budget of fires per pass, so a restart doesn't become a storm?

Technical questions? Find me afterwards.

TWO TOOLS

Punctuator or timer?

Punctuator

Every minute, for the task

- works directly on state stores
- not tied to one key
- prefix and range scans

Best for: heartbeats out, flushes, samples, a sweep over all keys



Timer

At its time, for one key

- knows its key
- fires at a time the data sets
- one-shot: arm it, move it, delete it

Best for: timeouts, retries, debounce

Different ~~treats~~ tools.

Why not have both?

Records tell you what happened.
Only a timer can tell you what didn’t.

Kafka Streams already has the task thread, the changelogged stores and the transactions. What’s missing is two method calls. They run today, on the Processor API you already write.

Hartmut Armbruster • **linkedin.com/in/hartmut-co-uk** • **stoatflow.io**

StoatFlow: a commercial stream processor with the Kafka Streams DSL and Processor API, on a single JVM.

BACKUP

Scheduled sources: output with no incoming data

> Report every device, every minute, **even one that went silent**.
> An event-time timer can’t: with no data, the watermark stands still.

```
builder.stream("heartbeats", Consumed.with(Serdes.String(), Serdes.String()))
    .groupByKey(Grouped.with(Serdes.String(), Serdes.String()))
    .count(Materialized.as("beats"));

builder.<String, Long>scheduled(Named.as("every-minute"), Duration.ofMinutes(1),
        PunctuationType.WALL_CLOCK_TIME, Serdes.String(), ctx -> {
      TimestampedKeyValueStore<String, Long> beats = ctx.getStateStore("beats");
      try (KeyValueIterator<String, ValueAndTimestamp<Long>> it = beats.all()) {
        it.forEachRemaining(kv -> ctx.forward(kv.key, kv.value.value()));
      }
    })
    .process(SinceLastReport::new, "last-reported")  // per device: total minus last report
    .to("beats-per-minute", Produced.with(Serdes.String(), Serdes.Long()));
```

Every minute, every device, even when nothing arrives: **a silent device reports 0**. The emitter observes; the lane acts. A **scheduled source** (`builder.scheduled`) runs on the wall clock or a cron expression, and its records do not move the watermark.

BACKUP

Timer, punctuator or scheduled source

USE	WHEN	RUNS	STATE
Timer	you know **which key** and **when** at ingest	the key's lane, serialised with `process()`	read/write
Punctuator	periodic work on the processing path: heartbeat, flush, sample; Kafka Streams code ported as-is	the punctuation lane; the commit waits for it	read/write
Scheduled source	a view across keys, a calendar fan-out, output with no input	an emitter on its own thread; the work on the lanes	emitter read-only, downstream read/write

From the scheduled-sources post on stoatflow.io.

BACKUP

What bit us

1. **The delete must commit with the output.** On many threads that takes a handshake: claim, `onTimer`, confirm, with the confirm in the output's transaction.
2. **Break ties on the full key bytes.** A 32-bit hash tiebreak silently dropped colliding timers: even odds of a collision with about 77,000 keys due at the same instant on one lane.
3. **Re-arming or deleting a timer while it fires.** Now a re-registration meeting a queued fire fires once, with the latest value, and a delete cancels it.
4. **The storm after downtime.** Every overdue timer was loaded into memory at once. Now at most N per domain are, and event time waits at the first one not yet loaded.

1 is the price of running on many threads. 2 to 4 are fixed.

BACKUP

The bounded timer cache, measured

- At most N timers per domain in memory, in the heaps and in flight together (default 100,000).
- Loaded in timestamp order behind one cursor per domain.
- Event time is held at the first stored timer not yet loaded. Processing time cannot be held, so a burst larger than N drains as fast as the lanes fire it.

Catch-up of 40,000 timers due a day ahead, measured locally (one broker, 2 partitions, 4 lanes):

N	TIME	RATE
2,000	3.1 s	13,114 timers/s
10,000	1.5 s	27,400 timers/s
uncapped	1.1 s	35,662 timers/s

Memory stays at N.

BACKUP

Where a named timer lives

- An **identity** row per (processor, key, name) holds the domain and the time. It is the changelogged truth.
- An **index** per domain, ordered by time, is derived locally, so a restore carries no orphans.
- The changelog key holds no time and no domain, so compaction does the replace, and moving a deadline, even across domains, costs one record.

Flink’s Process Table Functions pay the same second index: a `user-named-timers` timer service plus an `internal-named-timers-map` from name to timestamp.

BACKUP

Isn’t this just TTL?

TTL (KAFKA STREAMS’ SWEEP PROPOSAL)	TIMER
forgets a key	acts for a key
no output	`forward`, retry, alert
a sweep is the right tool	a scan is the wrong tool

They are complementary, and you would want both.