Lean Roadmap: NATS JetStream Benchmark (Arroyo vs RisingWave)

Frozen plan and step-by-step build order • January 10, 2026

Scope

You are benchmarking a windowed aggregation pipeline using NATS JetStream for transport, optionally Arroyo or RisingWave for processing, and TiDB as the sink. The goal is not to build a production platform; it is to produce comparable end-to-end numbers (throughput, lag, p95/p99 latency) with minimal code.

Hard guardrails (keep it small)

- JetStream only (including bench.prod); no core NATS.
- Single Gradle Kotlin project. No frameworks (no Spring/DI, no metrics stacks).
- One file per runnable main. Print one metrics line per second (plus one startup line).
- Implement in order: compose -> streams -> Case A -> TiDB table -> Case B -> engines (C/D/E/F). Stop when a step fails.

Frozen configuration

- Partitions p in {0..15} (16 subjects).
- Workload: processing-time 10s tumbling window GROUP BY key with aggregates count and sum(value).
- Payload: 512 bytes fixed.
- Keys: num_keys = 20,000; key = "k" + (seq % num_keys).
- DB writer: batch_size_rows=500, flush_interval_ms=50, autocommit=false.
- Run timing: warmup 20s, measure 120s.

Cases (A-F) overview

Case	Flow	What it tells you
A	`K -> events.<p> -> K`	Transport sanity for raw events.* (JetStream + client).
B	`Mock -> results.mock.<p> -> K writer -> TiDB`	Sink baseline: JetStream + writer + TiDB with same output schema.
C	`K -> events.<p> -> Engine -> results.<engine>.<p> -> K`	Engine benchmark without DB (observe results at NATS).
D	`K -> events.<p> -> Engine -> results.<engine>.<p> -> K writer -> TiDB`	End-to-end with Kotlin sink path.
E	`K -> events.<p> -> Arroyo -> results.arroyo.<p> -> Benthos -> TiDB`	End-to-end with Benthos sink path (Arroyo only).

F	`K -> events.<p> -> RisingWave -> TiDB`	End-to-end with RisingWave direct sink (no results subject).

Streams, subjects, durables

Streams

Stream	Subjects	Notes
EVENTS	`events.*`	16 partitions: events.0 ... events.15
RESULTS	`results.*`	results.<variant>.<p>, where variant in {mock, arroyo, rw}
CTRL	`bench.*`	bench.prod heartbeat messages

Durable consumer names (must match)

Case	Durable	Subscribes to
A	`dur.A.events`	`events.*`
B	`dur.B.results_mock`	`results.mock.*`
C	`dur.C.events.<engine>`	events.* (engine input)
C	`dur.C.results.<engine>`	results.<engine>.* (Kotlin observer)
D	`dur.D.events.<engine>`	events.* (engine input)
D	`dur.D.results.<engine>`	results.<engine>.* (Kotlin writer)
E	`dur.E.events.arroyo`	events.* (engine input)
E	`dur.E.results.arroyo`	results.arroyo.* (Benthos input)
F	`dur.F.events.rw`	events.* (engine input)

Schemas and metrics contracts

Event schema (events.)

- run_id
- event_id = run_id + "-" + seq
- seq (monotonic integer, producer-side)
- producer_ts_ms
- key = "k" + (seq % num_keys)
- value (long)
- payload_bytes (512B fixed)

Partitioning

p = hash(key) % 16; publish to events.<p>.

Result schema (results..)

- run_id
- key
- window_end_ms (end boundary of 10s tumbling window, epoch-aligned)
- count
- sum_value
- seq = max(input seq in that key/window) (progress marker; not contiguous)
- producer_ts_ms = max(input producer_ts_ms in that key/window)

Mock results (Case B)

Mock producer publishes the same result schema to results.mock.<p>. Use producer_ts_ms = window_end_ms and seq as a simple global monotonic counter (start at 1, increment by 1 per published result row).

bench.prod heartbeat (CTRL)

Published every 1s by the active producer (events producer for A/C/D/E/F, mock producer for B).

- run_id
- prod_max_seq
- prod_rate (msgs/s)
- producer_ts_ms_now

Observers keep the latest bench.prod per run_id.

TiDB table

bench.window_results columns:

- variant (mock/arroyo/rw)
- run_id, key, window_end_ms
- count, sum_value, seq, producer_ts_ms
- db_insert_ts (default now on insert/commit)

Primary key: (variant, run_id, key, window_end_ms) for idempotent inserts.

Latency and throughput definitions

- Primary latency: db_insert_ts - window_end_ms (end-to-end after window closes).
- Secondary latency: db_insert_ts - producer_ts_ms (sensitive to key sparsity; keep as supporting signal).
- Lag: prod_max_seq - obs_max_seq.
- A/C obs_max_seq: max seq seen by Kotlin consumer (A from events seq, C from result seq field).
- B/D/E/F obs_max_seq: MAX(seq) from TiDB filtered by run_id and variant.
- Rates: Producer rate is events msgs/s; C observer rate is result rows/s; DB rate is committed rows/s.

Build roadmap (stop at each gate)

Each step has a deliverable and a verification check. Do not move forward until it passes.

Step 1 - Bring up infrastructure

Do

- Start docker compose with NATS (JetStream enabled, memory storage) and TiDB.
- Ensure you can reach NATS and TiDB from host.

Verify

- `docker compose up -d`
- `docker compose logs -n 50 nats`
- `docker compose logs -n 50 tidb`

Step 2 - Create JetStream streams

Do

- Create streams: EVENTS (events.*), RESULTS (results.*), CTRL (bench.*).
- Keep storage=memory for all streams.

Verify

- `nats stream add EVENTS --subjects "events.*" --storage memory --retention limits --discard old --max-age -1 --no-confirm`
- `nats stream add RESULTS --subjects "results.*" --storage memory --retention limits --discard old --max-age -1 --no-confirm`
- `nats stream add CTRL --subjects "bench.*" --storage memory --retention limits --discard old --max-age -1 --no-confirm`
- `nats stream ls`

Step 3 - Kotlin connectivity smoke test

Do

- Implement a tiny JetStream publish and subscribe test (one message) to bench.prod.
- Confirm the client can create a durable consumer.

Verify

- Run publisher once; run subscriber once; confirm it receives the message.

Step 4 - Case A (events transport baseline)

Do

- Run ProducerMain (events.* + bench.prod).
- Run ConsumerMain (dur.A.events on events.*).
- Verify metrics print once per second.

Verify

- Expect: obs_rate roughly matches prod_rate; lag stays bounded; p95/p99 stable.

Step 5 - TiDB schema

Do

- Create database/schema and bench.window_results table with PK (variant, run_id, key, window_end_ms).

Verify

- Run a manual INSERT and SELECT COUNT(*) to confirm connectivity.

Step 6 - Case B (sink baseline)

Do

- Run MockResultsProducerMain (results.mock.* + bench.prod).
- Run WriterMain (dur.B.results_mock) to write to TiDB with batching.
- Run DB validator query loop (can be inside WriterMain) to print DB rows/s, lag, p95/p99.

Verify

- Expect: rows/s stable; lag bounded; table fills with variant=mock.

Step 7 - Engine bring-up (one at a time)

Do

- Deploy Arroyo OR RisingWave locally (docker is fine).
- Configure: input events.* (dur.C.events.<engine>), output results.<engine>.<p> (Case C only).

Verify

- Do not proceed until engine consumes events and produces results.

Step 8 - Case C (engine benchmark, no DB)

Do

- Run ProducerMain.
- Run engine producing results.<engine>.<p>.
- Run Kotlin ObserverMain consuming results.<engine>.* (dur.C.results.<engine>).

Verify

- Expect: observer sees result rows; lag shows a sawtooth pattern around 10s windows; no unbounded growth.

Step 9 - Case D (engine + Kotlin writer + TiDB)

Do

- Reuse engine from Step 8.
- Run WriterMain consuming results.<engine>.* (dur.D.results.<engine>) and writing variant=<engine>.

Verify

- Expect: DB rows/s stable; lag bounded.

Step 10 - Case E (Arroyo + Benthos + TiDB)

Do

- Run Arroyo producing results.arroyo.<p>.
- Run Benthos consuming results.arroyo.* (dur.E.results.arroyo) and writing to TiDB as variant=arroyo.

Verify

- Expect: stable DB rows/s and lag.

Step 11 - Case F (RisingWave direct sink)

Do

- Configure RisingWave sink to TiDB (variant=rw).
- No results.rw.* subject required for F.

Verify

- Expect: rows appear in TiDB with variant=rw.

Step 12 - Benchmark run matrix

Do

- Run: A, B, C-arroyo, C-rw, D-arroyo, D-rw, E, F.
- For each: warmup 20s, measure 120s, save logs to a file.
- Capture docker stats for containers during measure window.

Verify

- Deliverable: one log file per run + a short summary table (peak rate, p95, p99, lag trend).

Metrics log line contract (copy/paste)

Print exactly one line per second per component during warmup and measure.

Producer

```
PROD rate=50000/s max_seq=1200000 run_id=...
```

Observer (Case A and Case C)

```
OBS rate=2000/s max_seq=1189000 lag=11000 p95=35ms p99=80ms run_id=...
```

DB validator (Case B/D/E/F)

```
DB rows=2000/s max_seq=1189000 lag=11000 p95=120ms p99=250ms run_id=...
```

Notes

- In A, rate is events/sec. In C, rate is result rows/sec.
- Lag in C uses output seq (max input seq per key/window), so it is a progress signal, not a result-count signal.
- If lag grows across multiple windows without dropping, the pipeline is falling behind.