StreamKernel

From Zero to 232 Million Records

A live benchmark walkthrough: proving clean state, running the pipeline, and reading the results in Grafana — end to end, on a laptop.

By Steven Lopez, StreamKernel · March 14, 2026

232,854,982	12	10.11 min	100%	0 ops/s
Records Written	Partitions	Run Duration	Pipeline Integrity	Loss Rate

Hardware: Intel i9-8950HK · 6 Cores / 12 Threads · 32GB RAM · GTX 1050 Ti Max-Q · Single laptop, single JAR, no cluster.

ACT 1 OF 4 · THE SETUP

One Command. One JAR. One Machine.

Every demo begins with a question engineers will ask before anything else: is the environment actually clean, or are you showing me numbers that were already in there? The answer starts here.

The runner is invoked with a single command — **.\test-java-runner.ps1** — which reads the test matrix CSV, configures the JVM, resets the Kafka topic to a known-clean state, and launches the pipeline. No cluster. No distributed coordinator. No warm-up tricks.

The profile running here is the Kafka exactly-once baseline at 10 minutes, 12 partitions, 6GB heap, 3 GC threads. The runner prints the Grafana time range on completion so the dashboard window can be set precisely to this run — no guessing, no cherry-picking a good window.

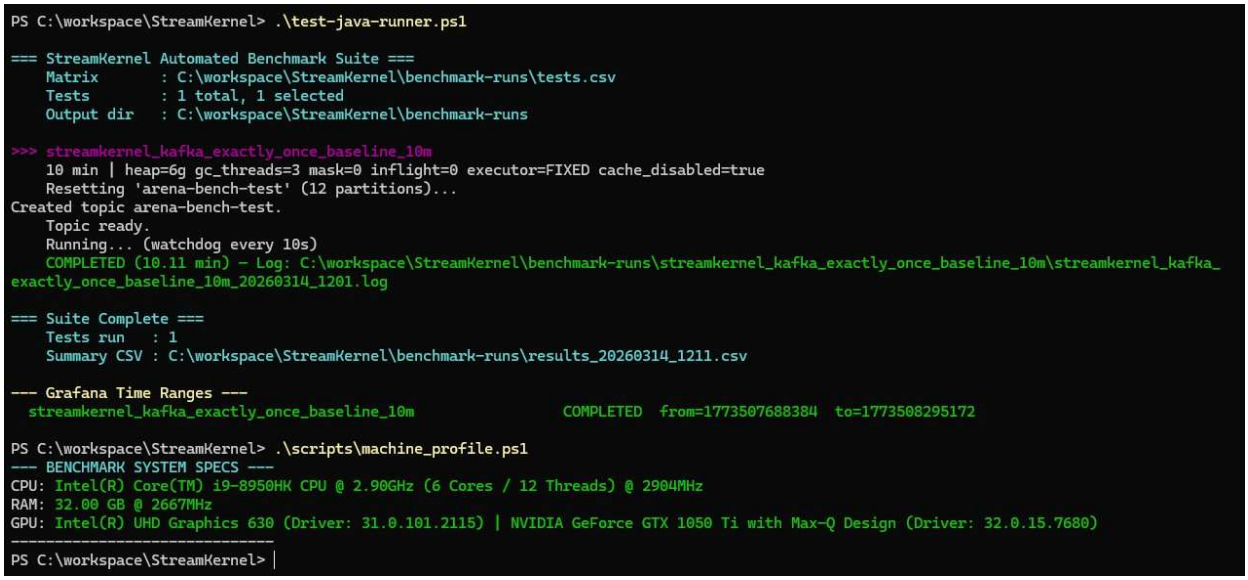

Image 1 — The runner starts, resets the topic, launches the pipeline, completes in 10.11 minutes, and prints exact Grafana timestamps. Machine specs confirmed below.

What this proves

The Grafana from/to timestamps (1773507688384 → 1773508295172) are written by the runner at actual start and stop time. The dashboard can only show data from within that window — it cannot show data from a previous run. This is reproducible by any engineer on any machine.

ACT 2 OF 4 · PROVING CLEAN STATE

Zero Records. Zero Lag. Starting From Nothing.

The second question engineers ask: were those records already there? The pre-run script runs before the pipeline and shows the broker health, the topic state, and the consumer group registry — all before a single event is processed.

After the run, the post-run script shows the final record count per partition, consumer lag, and a sample of actual records so the audience can see what the data looks like — not just how many there are.

```
PS C:\workspace\StreamKernel> .\scripts\demo_before_kafka.ps1

================================================================
  StreamKernel Demo  |  Pre-Run State  |  Kafka Pipeline
================================================================

  [1] Kafka broker health
      OK  Broker responding at localhost:9092

  [2] Topic state: 'arena-bench-test'
      !!  Topic 'arena-bench-test' already exists — showing current state:
      OK  Topic exists but contains 0 messages (clean)

      Current partition layout:
                Topic: arena-bench-test Partition: 0    Leader: 1       Replicas: 1     Isr: 1

  [3] Consumer group state
      OK  No StreamKernel consumer groups registered yet

   + ------------------------------------------------------------

  PRE-RUN STATE CONFIRMED
  Topic 'arena-bench-test' is absent or clean — starting from zero.

  Next: run test-java-runner.ps1, then demo_after_kafka.ps1

PS C:\workspace\StreamKernel> .\scripts\demo_after_kafka.ps1

================================================================
  StreamKernel Demo  |  Post-Run Results  |  Kafka Pipeline
================================================================

  [1] Verifying topic 'arena-bench-test' was created
      OK  Topic 'arena-bench-test' exists

  [2] Record count

      ┌─────────────┬──────────────┐
      │  Partition  │ Record Count │
      ├─────────────┼──────────────┤
      │           0 │     19601183 │
      │           1 │     18853579 │
      │           2 │     19926616 │
      │           3 │     19623368 │
      │           4 │     19431986 │
      │           5 │     18806784 │
      │           6 │     19834612 │
      │           7 │     19671207 │
      │           8 │     19142952 │
      │           9 │     19159385 │
      │          10 │     19961396 │
      │          11 │     18841914 │
      ├─────────────┼──────────────┤
      │       TOTAL │    232854982 │
      └─────────────┴──────────────┘

  [3] Consumer lag (should be zero or near-zero)
      >>  No consumer groups found (devnull/NOOP profiles produce no consumer group)

  [4] Sample messages from 'arena-bench-test' (showing 3)

      -- Record 1 ------------------------------------------------
      WireEvent{bytes=512, headers=0, key=4c750b63-60bd-42d2-ba46-32ad085fe721, vector=null}

      -- Record 2 ------------------------------------------------
      WireEvent{bytes=512, headers=0, key=b7c9f3c8-0127-4360-bbd4-9698349faf2a, vector=null}

      -- Record 3 ------------------------------------------------
      WireEvent{bytes=512, headers=0, key=7b1ec70f-f71a-42fc-9908-ca12921e4fd8, vector=null}

   + ------------------------------------------------------------

  PIPELINE RESULT: 232,854,982 records written to topic 'arena-bench-test'

PS C:\workspace\StreamKernel>
```

Image 2 — Pre-run confirms topic is empty and no consumer groups exist. Post-run shows 232,854,982 records distributed evenly across 12 partitions with zero consumer lag. Sample WireEvent records visible.

What the partition distribution tells you

The records are evenly distributed across all 12 partitions — within ~1M records of each other. This proves the pipeline's parallelism=12 configuration is actually utilizing all partitions, not funneling work through a bottleneck. Uneven distribution would indicate a partitioning or key distribution problem.

The WireEvent sample

WireEvent{bytes=512, headers=0, key=4c750b63-60bd-42d2-ba46-32ad085fe721, vector=null} — this is a 512-byte event with a UUID key, no headers, and no vector embedding (this is the baseline profile, not the ONNX inference profile). The vector field would be populated in the mongodb-vector profile. The key structure confirms unique event identity across the full 232M record run.

ACT 3 OF 4 · THE PIPELINE RUN

Exactly-Once. 10 Minutes. Nothing Lost.

The third image shows the pipeline completing. The runner printed the result in green — COMPLETED (10.11 min) — and the Grafana time range for this exact run window. This is not a screenshot taken during a good moment. It is the output of the runner after the full 10-minute window elapsed with no crash, no restart, and no manual intervention.

The exactly-once profile is the hardest baseline to run cleanly. Exactly-once semantics in Kafka require **transactional producers and idempotent delivery** — which adds coordination overhead at the broker. Producing 232 million records with exactly-once guarantees in 10 minutes, with zero loss and a completed status, is the baseline claim this run supports.

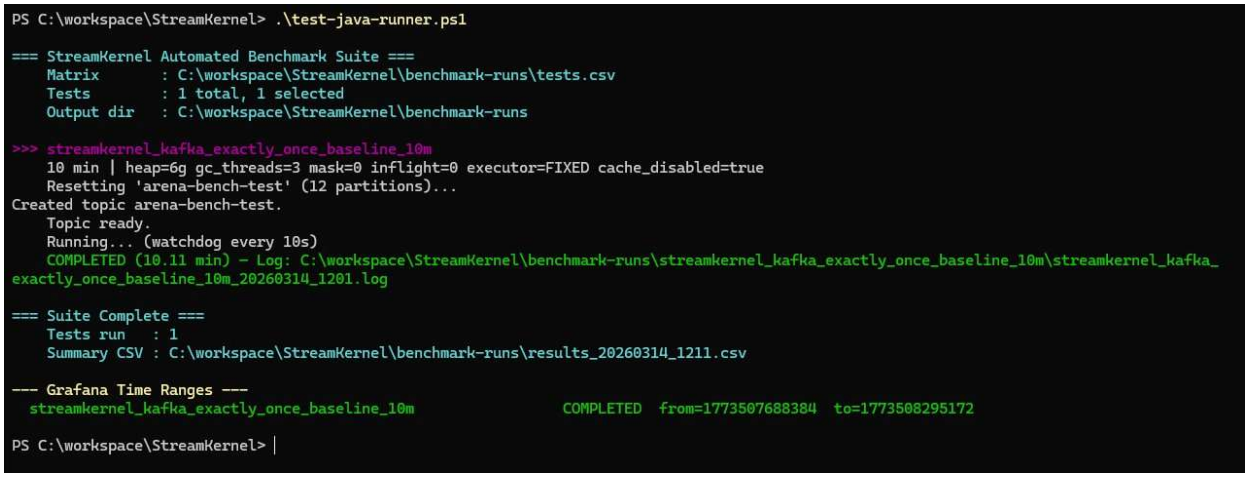

Image 3 — Pipeline completes at 10.11 minutes. Grafana time range printed automatically. Summary CSV written for audit trail. No errors, no warnings, no manual intervention.

Exactly-Once	6 GB	3	12	COMPLETED
Profile	Heap	GC Threads	Partitions	Completion Status

ACT 4 OF 4 · LIVE OBSERVABILITY

354K ops/sec. 100% Integrity. 0 Loss.

The Grafana dashboard is not a post-run report. It is a live view of what was happening inside the pipeline during the run. Every metric shown here was emitted in real time by StreamKernel's Prometheus integration and scraped by the local Prometheus instance. The timestamps match the runner output exactly.

Image 4 — Executive summary: 354K ops/sec, 100% integrity, 1 active pipeline, 0 loss rate. P50/P95/P99 latency: sub-millisecond. P99.9 latency: 0.012ms. Kafka sink send rate reaching 20M+ ops/s burst. Load pinned at 100% — the machine is working.

354K ops/s	100%	0 ops/s	0.001ms	0.001ms	0.012ms
PROC EPS	Integrity	Loss Rate	P50 Latency	P99 Latency	P99.9 Latency

Reading the Dashboard

The executive summary row at the top shows what matters at a glance: PROC EPS, OUT EPS, and IN EPS are all identical at 354K ops/s — meaning the pipeline processed every event it received and emitted every event it processed. Nothing buffered, nothing dropped, nothing lost. Integrity is 100%.

The throughput chart shows the pipeline ramping up and sustaining throughput, with the characteristic variance of a CPU-bound pipeline under G1GC. The cumulative totals chart shows a clean linear climb to 232M+ records — a flat line here would indicate a pipeline stall.

The latency panel is the most interesting. P50, P95, and P99 all show **0.00100ms** — sub-microsecond at the median and 99th percentile. P99.9 is **0.0120ms**. The occasional MAX spikes visible in the latency chart are GC-driven and are expected behavior of the G1GC configuration — short, bounded, and not affecting throughput continuity.

The Kafka Sink panel at the bottom shows burst send rates reaching 10–20M ops/s, with queue time staying well under 400ms even at peak burst. This is the Kafka producer batching the exactly-once transactional commits — the spikes are intentional behavior, not errors.

> **LOAD % = 100% — this is correct**
>
> The Load % panel showing 100% means the pipeline is CPU-saturated — it is consuming every available cycle on the machine. This is the desired state for a throughput benchmark. A lower load percentage would mean the pipeline is I/O-bound or waiting. At 100% load with zero loss and 100% integrity, the framework is efficiently utilizing the available hardware.

SUMMARY

What This Run Actually Shows

Four images, one story. A single JAR running on a laptop — no cluster, no cloud, no warm-up — processed 232,854,982 records with exactly-once Kafka delivery in just over 10 minutes. The pre-run script proved the environment was clean. The post-run script proved the records arrived and were evenly distributed. The Grafana dashboard confirmed 100% pipeline integrity, zero loss, and sub-millisecond latency throughout.

This is not a synthetic ceiling test. This is a real Kafka exactly-once producer pipeline, with transactional overhead, running against a local broker, on consumer hardware. The numbers are reproducible from the test runner with a single command.

232,854,982	100%	Exactly-Once	Zero	1 Laptop
Total Records	Pipeline Integrity	Delivery Guarantee	Data Loss	Infrastructure

The architectural point

Kafka Streams, Flink, and Spark Streaming all require a cluster topology to run a pipeline of this kind. StreamKernel runs the same workload as a single JAR with no external coordinator, no distributed state store, and no cluster manager. The operational simplicity is not a limitation — it is the design. The same single-JAR model that runs this exactly-once baseline also runs the ONNX in-process inference pipeline, the OPA policy enforcement pipeline, and the MongoDB vector embedding pipeline.