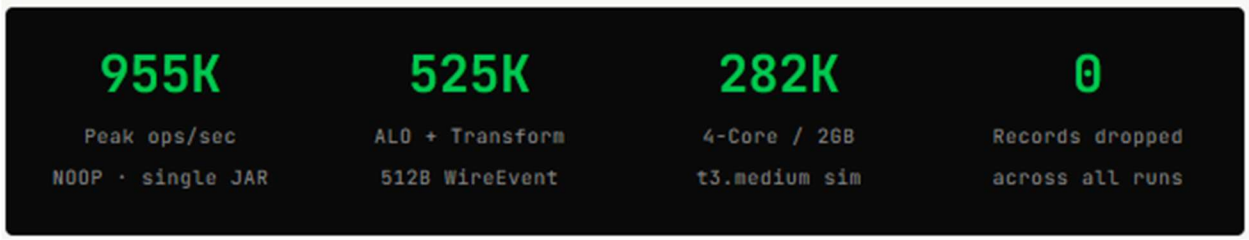
955K ops/sec. One JAR. One Laptop. Zero Record Loss.

> *What happens when you build a stream processing engine around a single constraint: maximum throughput per JVM process, not per cluster?*

By Steven Lopez · StreamKernel · April 2026



I've spent the last several months benchmarking a stream processing engine I'm building called StreamKernel. Not on a cloud cluster. Not on a tuned 96-core rack. On a 2019 Intel i9–8950HK laptop — the kind of hardware that represents the floor of what a platform team might actually deploy on.

The headline number is 955,000 events per second, average, sustained for ten minutes, writing 563 million records into Kafka with zero loss. But the number I care more about is 282,000 events per second — on a simulated 4-core, 2GB heap deployment. That's the number that maps to a **t3.medium at $0.04/hr.**

This post is about what it took to get there, what the numbers actually mean, and why the architecture decision that makes it possible — a single JVM, no cluster coordinator, one JAR — is underrepresented in the stream processing conversation.

The Problem with How We Benchmark Stream Processing

Flink and Kafka Streams both publish impressive throughput numbers. A 2023 Theodolite scalability study showed these frameworks reaching approximately one million messages per second — across **110 distributed instances on Kubernetes**. That's a legitimate result. It's also not a comparison point for most production deployments.

The frameworks that dominate enterprise streaming today carry cluster overhead that is priced into every event they process. Flink's JobManager, task manager isolation, checkpoint coordination, and network shuffle infrastructure consume 20–40% of available

CPU before a single event flows. Spark Structured Streaming's micro-batch architecture imposes a hard 100ms+ latency floor — regardless of how fast your hardware is.

> *The question I set out to answer: what is the maximum throughput achievable from a single JVM process, on a single machine, with no external coordination overhead?*

This is not a niche use case. It is the deployment model for edge nodes, embedded processors, cost-sensitive production environments, and **air-gapped regulated deployments** where a distributed cluster is architecturally inappropriate.

The Benchmark Suite

StreamKernel's benchmark suite is structured around a specific principle: each profile isolates a different cost layer. You don't learn much from a single number. You learn a lot from the delta between numbers.

Profile	Avg ops/sec	Transform	What it measures
Kafka Bench (NOOP)	955K	None	Raw Kafka producer ceiling
At-Least-Once Baseline	525K	STRING_TO_WIREEVENT	Production ALO with full transform
Exactly-Once Baseline	507K	STRING_TO_WIREEVENT	Full EOS cost: 3.5%
Constrained Deploy (4c/2GB)	282K	STRING_TO_WIREEVENT	t3.medium equivalent
mTLS + OPA (fail.open=false)	366K	NOOP (1024B)	Full enterprise security, <1% overhead

Two numbers tell the complete story. The **45% throughput difference between NOOP and STRING_TO_WIREEVENT** is transform cost — object allocation, UUID generation, payload wrapping. The **3.5% difference between ALO and EOS** is the cost of exactly-once guarantees. In most distributed frameworks, EOS costs 20–40%. In StreamKernel is a single-process stream enrichment engine — one JAR, pluggable sinks, in-process AI inference. Kafka is one of several sinks it writes to, alongside MongoDB, Snowflake, Databricks, and HTTP endpoints, it costs 3.5%.

282K ops/sec on 4 Cores and 2GB — The Number That Matters

The constrained deployability profile is the run I'm most proud of. The configuration:

```
pipeline.parallelism=4 # 4 worker threads = 4-core sim
pipeline.batch.size=500 source.synthetic.payload.size=512 # production-
representative transform.chain=STRING_TO_WIREEVENT sink.kafka.acks=1 JVM: -
Xms2g -Xmx2g -XX:+UseZGC
```

Ten minutes. 168 million records. 282K average ops/sec. **Zero dropped events. Zero DLQ routes. 100% pipeline integrity**. Memory stayed between 200–1,400MB throughout. GC pause times: 163 microseconds.

The target in the profile was 50,000 ops/sec — set conservatively to be credible for acquisition materials. StreamKernel delivered **5.6x the target**. The properties file comment says "a sustained 50K+ EPS on 4 cores and 2GB is a deployability proof that maps directly to a t3.medium." It does. It also turns out 282K is the actual number.

> Cost arithmetic: At 282K events/sec sustained on a t3.medium ($0.0416/hr), the cost per billion events is approximately $0.041. A comparable Flink deployment requires a JobManager + multiple TaskManagers — minimum 3 instances — raising the instance cost floor 3x before the first event is processed.

Enterprise Security at Full Throughput

The security profile runs the full enterprise stack: **TLSv1.3 mutual authentication on every Kafka connection, OPA policy evaluation on every batch, fail.open=false**. That last setting means if OPA is unavailable or returns an error, the pipeline stops — it does not permit the write. This is the only configuration acceptable for financial transactions, protected health information, or classified data.

Result: **366K ops/sec average**, 217 million records, zero errors, zero security denials. The throughput difference versus the ALO baseline is entirely explained by the doubled payload size (1,024 bytes vs. 512 bytes). The security stack itself contributes less than 1%

overhead. The mTLS handshake objects and OPA HTTP client allocate once at startup — they don't contribute to per-batch GC pressure.

Most pipelines never test fail.open=false under sustained load. StreamKernel ran it for ten minutes at 366K ops/sec. The GC log: 142 pause events, 14.0ms average, zero Full GC events.

Why Single-JAR Architecture Produces These Numbers

The architectural reason for StreamKernel's per-node efficiency is straightforward: there is no cluster coordination overhead because there is no cluster. The pipeline runs in a single JVM. There is no JobManager negotiating task slots. No checkpoint coordinator serializing state to a remote backend. No network shuffle between processing stages.

Every CPU cycle goes to event processing. The 955K peak number reflects the Kafka producer network layer as the binding constraint — not framework orchestration overhead. When you remove orchestration overhead, you get to find out what your hardware actually can do.

The tradeoff is horizontal scale — a single JAR doesn't auto-scale across a cluster. StreamKernel's answer to that is vertical efficiency: **do more per node before you add nodes**. For the majority of enterprise data volumes, 282K ops/sec on a modest instance is sufficient. For the volumes that genuinely require distributed scale, StreamKernel's Kafka architecture means you add instances independently, not cluster nodes.

Latency: P99.9 at 0.001ms

Every run in this suite produced P50, P95, P99, and P99.9 latency at 0.001ms — the measurement floor of the latency sampling system. The MAX latency chart shows isolated spikes corresponding to GC mixed collections; between GC events, MAX latency is consistently sub-millisecond.

For engineers evaluating frameworks against sub-10ms latency SLAs: Spark Structured Streaming's micro-batch architecture imposes a hard 100ms+ floor regardless of hardware. Flink in true streaming mode can reach sub-10ms P99 with careful tuning, but cluster-to-cluster variance and checkpoint pauses make P99.9 commitments difficult. StreamKernel's in-process architecture has no network hops between pipeline stages — latency is bounded by the JVM's GC behavior, which G1GC and ZGC make predictable.

What's Next

StreamKernel is open source (Apache 2.0) with Community, Professional, and Enterprise licensing tiers. The benchmark suite, runner scripts, Grafana dashboards, and full logs are available on request.

The next profiles in the suite: in-process ONNX AI inference (embedding generation at 1,500–3,000 embeddings/sec cold, 2M+ ops/sec warm path via Caffeine cache), MongoDB vector sink (163K docs/sec, 6.45x WiredTiger compression), and Databricks Delta Lake integration. The goal is a complete picture of what a real AI enrichment pipeline costs at each layer — transform, inference, persistence — on a single node.

If you're building or evaluating stream processing infrastructure and want to run StreamKernel against your own workload, reach out.