STREAMKERNEL

Production-Grade Embedding Pipelines with MongoDB Atlas Vector Search

How to run ONNX embedding inference inside a JVM streaming pipeline, write 200,000+ vectorized documents to MongoDB in 10 minutes on a developer laptop, and achieve 28.3× throughput improvement — with zero record loss, no GPU, and no external inference service

By Steven Lopez · StreamKernel · April 2026

28.3×	200,928	100%	::97%
Pipeline Throughput Gain	**Docs / 10-min Window**	**Pipeline Integrity**	**Of Inference Ceiling**
11.9 r/s → 336.8 r/s	Embedded + written to MongoDB	Zero record loss, all runs	Pipeline overhead ≈3%

Audience	Engineers and architects building real-time AI enrichment pipelines
Date	April 2026
Status	Technical white paper — April 2026
Benchmark model	all-MiniLM-L6-v2 (FP32 ONNX), CPU-only, Java 21
Sink	MongoDB 7.x, MONGO_VECTOR profile, Atlas-ready schema

EXECUTIVE SUMMARY

The problem: vector databases are only as good as what feeds them

Vector search is only as useful as the pipeline that feeds it. MongoDB Atlas Vector Search is a powerful platform for semantic query — but it receives value only after embeddings are continuously generated, enriched, and written into collections. For many engineering teams, that upstream work is where the real complexity lives.

StreamKernel is a JVM-native streaming runtime that eliminates that upstream gap. It runs ONNX embedding inference in-process — no Python sidecar, no REST call to a model server, no serialization boundary between inference and persistence. Records flow from source through tokenization, batched ONNX inference, metadata enrichment, and MongoDB bulk write inside a single JVM process.

CORE THESIS

The hardest part of deploying vector search is not the database — it is the pipeline that fills it.

StreamKernel collapses embedding generation, vector enrichment, and MongoDB persistence into one JVM process. This paper benchmarks that claim across 14 runs and 28.3× of measured throughput improvement.

Question	Conventional approach	StreamKernel approach
How are embeddings generated?	Custom pipelines, model services, or customer-built jobs	**In-process ONNX inference inside the pipeline runtime**
Where does complexity live?	Before MongoDB: orchestration, infrastructure, security, DevOps	**Collapsed into one deployable runtime boundary**
How does MongoDB get populated?	Atlas waits until upstream work is solved, often months later	**Continuous write path from inference to Atlas, starting immediately**
What does the operator benefit from?	Multi-service stacks with multiple approval and failure boundaries	**One deployable, one log stream, one configuration boundary**

THE PROBLEM WITH CONVENTIONAL EMBEDDING PIPELINES

Embedding generation is where production timelines slip

A vector database becomes valuable only after embeddings are continuously generated and delivered into it. For most engineering teams, that means assembling a chain of ETL jobs, embedding services, storage layers, queueing systems, access controls, deployment approvals, observability stacks, and database writes.

This creates a predictable pattern: the database may be ready, but the vector generation path remains bespoke and fragile. The more custom that upstream path becomes, the slower the team moves from proof-of-concept to production.

Current pattern	Operational burden	Customer impact
ETL + model service + Atlas	Multiple deployables, multiple teams, multiple failure domains	Longer approval cycles and slower rollout
Remote embedding APIs	Network dependency, rate limits, data movement, external vendor risk	Harder for regulated or air-gapped workloads
Python inference sidecar	Separate runtime, dependency drift, serialization boundary	More DevOps overhead and more points of failure
Batch-only vector generation	Delayed freshness and limited real-time behavior	AI experiences feel stale or incomplete

THE ATTRIBUTION PROBLEM

When embedding pipelines are hard to deploy, teams describe the entire vector search initiative as difficult — even when the database layer is working perfectly. The friction lives upstream. StreamKernel is designed to remove it.

ARCHITECTURE

Architecture: inference inside the pipeline process

StreamKernel is a transport-agnostic, sink-agnostic streaming runtime that runs ONNX embedding inference inside the JVM process. The model loads once at startup through Deep Java Library (DJL) and ONNX Runtime, executes through a configurable predictor pool, and writes enriched records directly to MongoDB through the MONGO_VECTOR sink — all without leaving the process boundary.

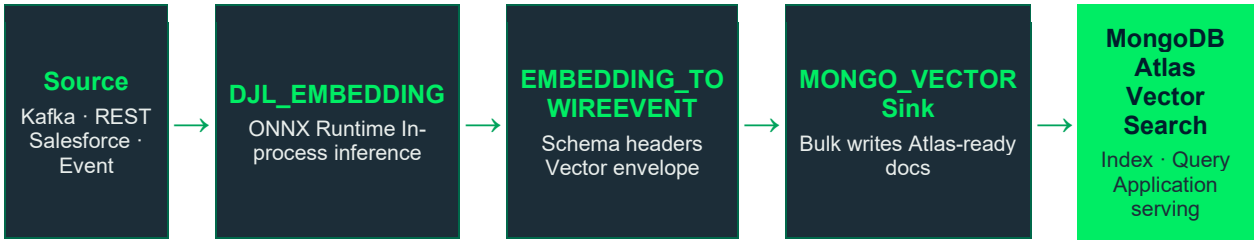

Figure 1. The full pipeline from source to MongoDB Atlas Vector Search runs inside a single JVM process. MongoDB Atlas (highlighted) receives ready-to-index vector documents and owns the query and serving layer.

The key architectural choice is the removal of the network hop between pipeline enrichment and persistence. In conventional patterns, records cross process boundaries before and after inference. In StreamKernel, source, embedding transform, metadata transform, and sink execute in one process — with Prometheus/Grafana metrics and enterprise security profiles available around the runtime.

Layer	StreamKernel role	Value delivered
Ingestion	Accepts streaming or synthetic/event sources through pluggable source SPI	Continuous vector supply into MongoDB; no batch jobs required
Inference	Runs ONNX embeddings in-process through DJL/ONNX Runtime	Eliminates the Python model server and its operational surface area entirely
Vector envelope	Adds schema metadata, dimensions, encoding, event IDs, and vector payload	Every MongoDB document carries typed, versioned embedding metadata
Persistence	Bulk writes through MONGO_VECTOR into MongoDB collections	MongoDB receives validated, bulk-written vector documents on every batch
Observability	Exposes pipeline EPS, integrity, JVM, queue, and sink metrics via Prometheus	Single log stream, single metrics endpoint, single operational boundary to monitor

Output Document Schema

Every record written by StreamKernel arrives in MongoDB with a consistent, Atlas-indexable schema. The vector_embedding field is named and typed correctly for a MongoDB Atlas Search vector index on first creation:

Field	Type	Value / Purpose
`_id / ticketId`	string	UUID v4 — business key as document identity, ready for Atlas filtering
`vector_embedding`	**Array[384]**	**Float32 L2-normalized MiniLM-L6-v2 vectors — Atlas vectorSearch path field**
`headers.content-type`	string	application/x-streamkernel+vec-f32
`headers.embedding.dims`	string	"384" — explicit dimension declaration for index validation
`headers.sk.schema`	string	wireevent/vector-f32/v1 — versioned schema identity
`updatedAt`	timestamp	Record enrichment time — supports time-range filtering in Atlas queries

TECHNICAL PROOF

Benchmark evidence: production-relevant throughput, developer-grade hardware

The benchmark series measured all-MiniLM-L6-v2 sentence embeddings running end-to-end through the full StreamKernel pipeline — from synthetic source through inference through MONGO_VECTOR sink — on a developer laptop with Java 21, ONNX Runtime 1.20.0, and MongoDB 7.x Community Edition. No GPU. No hardware changes. 14 runs.

11.9	336.8	28.3×	0
r/s Baseline	**r/s Optimized**	**Improvement**	**Record Loss**
Run 1 — starting point	Run 12 — same hardware	CPU-only, no GPU	Across all 14 runs

The most important benchmark result is not the raw throughput number. It is the ratio of end-to-end pipeline throughput to standalone ONNX inference throughput. At short sequence lengths, StreamKernel operated at approximately 97% of the pure inference ceiling. Pipeline overhead — source, queueing, metadata transform, MongoDB vector write, metrics, and JVM management — accounts for only ~3% of total processing time.

WHAT THIS MEANS IN PRACTICE

StreamKernel feeds MongoDB with correctly shaped, validated vector documents while keeping the primary bottleneck where it belongs: model compute. The MongoDB write overhead is not the constraint — at 97% of the standalone ONNX inference ceiling, the pipeline plumbing is essentially free.

Metric	Baseline (Run 1)	Optimized (Run 12)	Why it matters
End-to-end throughput	11.9 r/s	**336.8 r/s**	End-to-end measurement: source to MongoDB write, not just inference
Documents / 10 min	6,720	**200,928**	Practical production window measurement
Pipeline integrity	100%	**100%**	Throughput improved without correctness trade-off
Record loss	0	**0**	No dropped records across the full benchmark path

Metric	Baseline (Run 1)	Optimized (Run 12)	Why it matters
Heap used (steady state)	~430 MiB	**~258 MiB**	Efficiency improved alongside throughput
Pipeline vs. inference ceiling	—	**~97%**	Pipeline plumbing consumes ~3% of total time; model compute owns the rest

WHO SHOULD USE THIS

StreamKernel is most useful when

StreamKernel is most useful when the blocking problem is not the vector database — it is the pipeline that feeds it. The following scenarios represent the strongest fit:

Scenario	Without StreamKernel	With StreamKernel
Custom embedding models	Each team builds and maintains a custom embedding generation stack	Reusable runtime for ONNX model execution and Atlas-ready vector delivery
Internal approvals / security review	Each service requires separate approval, CVE tracking, and runbook	Reduces deployables and narrows the operational boundary to one runtime
Latency-sensitive AI	REST calls to embedding APIs introduce per-record network latency	Runs inference and MongoDB writes in one process path, no network boundary
Data governance / residency	Records leave the network boundary for embedding, creating audit gaps	Supports fully local, private, or cloud-contained inference without external calls
Observability	Root cause analysis requires correlating logs across 3+ services	Prometheus/Grafana-first operational view: EPS, sink writes, integrity, JVM

THE ARCHITECTURE BET

The conventional embedding stack has at least three moving parts: a data source, a model server, and a database. StreamKernel collapses those into one JVM process.

Fewer moving parts means fewer failure modes, fewer deployment approvals, fewer log streams to correlate, and fewer SLOs to define. That operational simplicity is not a feature — it is the architecture.

TECHNICAL ALIGNMENT WITH MONGODB ATLAS

Why MongoDB Atlas is the right sink for this architecture

StreamKernel's MONGO_VECTOR sink is not a generic database writer. It is designed specifically to produce Atlas-indexable documents: explicit field naming, correct vector payload encoding, schema version headers, and bulk write optimization. The pipeline and the database are designed to hand off cleanly — StreamKernel ends where Atlas begins.

Atlas capability	How StreamKernel prepares records for it
Increase Atlas Vector Search adoption	Shortens the path from customer data to searchable vectors in Atlas
Improve customer time-to-value	Reduces the number of custom services required before Atlas is useful
Support proprietary model customers	Customers bring their own ONNX-compatible embedding models without rebuilding ingestion
Strengthen enterprise AI narrative	Connects ingestion, inference, and vector persistence into a credible reference architecture
Reduce field friction for CS and SA teams	Provides a concrete, reproducible pattern for upstream vector generation

The division of responsibility is clean: StreamKernel owns data motion, inference, and vector materialization. MongoDB Atlas owns indexing, querying, hybrid search, filtering, and application serving. Neither system tries to replace the other.

REFERENCE ARCHITECTURE

MongoDB Atlas-ready vector pipeline

A production-oriented StreamKernel + MongoDB architecture can be packaged as a reference implementation for teams that want to operationalize vector search without introducing a dedicated model-serving layer.

Stage	Reference component	Production consideration
Data source	Kafka, REST, Salesforce, Pulsar, or synthetic/test source	Use customer's existing ingestion path; StreamKernel is transport-agnostic
Embedding transform	DJL_EMBEDDING with ONNX Runtime	Tune pool size and intra-op threads within CPU core budget
Vector envelope	EMBEDDING_TO_WIREEVENT	Attaches dims, type, encoding, schema version, and event identity to each record
MongoDB sink	MONGO_VECTOR	Bulk writes to Atlas-ready collection; configurable business key as _id
Search layer	MongoDB Atlas Vector Search	Create vector index on vector_embedding field; query via Atlas-native $vectorSearch
Observability	Prometheus + Grafana	EPS, queue depth, integrity, JVM heap, MongoDB sink write rate — all pre-wired
Security	OPA/mTLS capable profiles	Validated at 366K ops/sec with mTLS + OPA; PERMIT_ALL for benchmark mode only

CLEAN HANDOFF POINT

StreamKernel's job ends when the vector document is durably written to MongoDB. Atlas's job begins there: indexing, querying, hybrid search, filtering, and serving results to applications.

The boundary is deliberate. StreamKernel does not attempt to replicate Atlas capabilities. It produces the documents Atlas needs, in exactly the shape it expects.

RECOMMENDED USE CASES

Where StreamKernel is most compelling

Use case	Fit	Reason
Real-time support ticket enrichment	**High**	Short text, frequent updates, strong MongoDB document model fit
Internal enterprise semantic search	**High**	Custom data + custom governance + Atlas Vector Search = natural pairing
RAG ingestion pipelines	**High**	Embeddings must be generated before retrieval quality exists; real-time freshness matters
Regulated or private AI workloads	**High**	Local inference avoids external API dependency; supports air-gapped and data-resident deployments
Long-form document embedding	**Medium**	Supported, but sequence length dominates CPU throughput; INT8 quantization recommended
GPU-scale embedding farms	**Selective**	StreamKernel can orchestrate and feed Atlas, but GPU model servers may own raw inference

WHAT COMES NEXT

Planned work and reproducibility

The benchmark results in this paper are reproducible on any comparable hardware. The following assets are planned to make that reproduction straightforward and to extend the benchmark to additional configurations:

Next asset	Purpose
Atlas demo runbook	Show before/after collection state, vector document schema, Atlas vector index creation, and $vectorSearch queries against StreamKernel-produced vectors
Reference repository	Clean MongoDB-focused configuration with repeatable commands, Docker Compose stack, and documented property tuning guide
Grafana dashboard bundle	Pre-built panels for vector throughput, Mongo sink write rate, integrity, queue depth, and JVM behavior — importable and shareable
Customer architecture diagram	Communicate where StreamKernel sits relative to Atlas, blob/object storage, and end-user applications
INT8 benchmark follow-up	Show additional CPU efficiency gains where quantized models are acceptable; sweep data suggests 1.3×–1.6× improvement at production sequence lengths

CONCLUSION

The in-process architecture delivers on its promise

The benchmark series in this paper documents what is possible when ONNX inference runs inside the pipeline rather than beside it. A developer laptop. Java 21. No GPU. 14 runs. 28.3× throughput improvement from a starting point of 11.9 records per second to 336.8 records per second — with 100% pipeline integrity across every run and zero record loss.

The architectural lesson is not specific to this benchmark. Any team building a real-time embedding pipeline — for MongoDB Atlas or any other vector database — faces the same upstream friction: model servers to deploy, serialization boundaries to manage, network latency to absorb. Collapsing that into a single JVM process is not a micro-optimization. It is the architecture.

RESULTS SUMMARY

28.3× throughput improvement — 11.9 r/s → 336.8 r/s

200,928 embedded documents written to MongoDB in a single 10-minute run

100% pipeline integrity and zero record loss across all 14 benchmark runs

~97% of standalone ONNX inference ceiling — pipeline overhead is ~3%

CPU-only. Developer laptop. Java 21. No GPU. No external inference service.

One process. One log stream. One operational boundary.

APPENDIX: BENCHMARK CONTEXT

Test Environment

Parameter	Value
Hardware	12-logical-core CPU, Windows 11 Pro, High Performance power plan, PCIe ASPM off
Java	OpenJDK 21 (Eclipse Temurin), ZGC, 4 GiB heap
ONNX Runtime	1.20.0, CPU execution provider, ALL_OPT optimization level
DJL	0.32.0, OnnxRuntime engine, native singleton
Model	all-MiniLM-L6-v2 FP32 ONNX export, 384-dimensional output, L2-normalized
MongoDB	7.x Community Edition, Docker, localhost:27017, MONGO_VECTOR sink, bulk writes
Pipeline	parallelism=16, pool=1, intra-op threads=6, flush=25ms, payload=96 chars (SUPPORT profile)

Key Technical Findings

Finding	Implication
Input sequence length dominates throughput	Short text workloads (support tickets, queries, log lines) achieve the highest throughput
CPU thread budget must be respected	pool × intra-op-threads should stay within available core count; violations regress throughput severely
Pipeline overhead is bounded and small	StreamKernel reaches ~97% of standalone ONNX throughput; database write overhead is ~3% of total time
Correctness is invariant	All 14 benchmark runs reported zero drops and 100% pipeline integrity, including misconfigured runs
Atlas vector search remains downstream	StreamKernel produces correctly shaped vector documents; MongoDB Atlas indexes and queries them via $vectorSearch

Benchmark results reflect the test environment described above. Results on other hardware will vary. This paper is not a MongoDB-endorsed performance claim. All benchmark runs were conducted by StreamKernel Engineering.