BEAM SUMMIT 2026 · NEW YORK · 22 JUN

Real-Time AI Pipelines at Scale.

Embedding LLMs into Apache Beam for live, per-event inference.

Samir Sengupta · AI/ML Engineer, SygentAI

PYTHON APACHE BEAM RAG VLLM FAISS AWS BEDROCK

THE PRODUCTION GAP

The model is the easy part.

A capable fraud model is an afternoon of work. Running it on a live stream of transactions, deciding fraud or not in real time without breaking, is the hard part. Three pressures decide it.

Latency

A fraud verdict has to come back before the transaction clears, in a fraction of a second.

Cost

A heavy model on every transaction burns money. Spend it only where it earns its keep.

Reliability

A live stream never stops. If the model fails, fall back to rules and keep going.

The trap teams polish the model for months, then lose the launch to latency, cost, and uptime.

A FAIR QUESTION

Fraud scoring is decades old. Where does the AI fit?

Banks have scored fraud with classical machine learning, gradient boosted trees like XGBoost, for years, returning a verdict in milliseconds. That instant approve or decline is not the LLM's job.

The fast tier

A lightweight model makes the millisecond call inside the authorization window. Proven, fast, and cheap.

The reasoning tier (new)

The LLM works the harder cases, the review queue, and the explanations. It reasons over messy context and, with RAG, adapts to a new scam by updating data instead of retraining.

This talk is not about replacing fraud scoring. It is about how you run any heavy model, including a slow and expensive LLM, on a live stream at scale. Apache Beam is how.

In practice Mastercard shipped a RAG-based scam-detection system in 2024; LLMs sit in the review tier, not the millisecond gate.

The honest number the LLM takes a few seconds per call; the fast tier holds sub-100ms and the LLM adds reasoning where it counts.

APACHE BEAM

Write once. Run anywhere.

Apache Beam is a unified programming model for data pipelines. You express the logic once, and a runner executes it, locally or across a cloud cluster.

A pipeline is just the sequence of steps your data flows through. Because the model is unified, the same pipeline runs on your laptop during development and on a production cluster at scale, with no rewrite. You change where it runs, not what it does.

Unified
One API for many execution engines and for both batch and streaming.

Portable
Pick the runner without touching your logic. Avoid lock-in.

ML-aware
RunInference makes serving a model a first-class step in the pipeline.

Who runs it Spotify uses Beam on Dataflow for ML pipelines over millions of items; the same code also runs on Flink and Spark.

THE RUNNER

One pipeline, many engines.

The runner is the engine that executes your pipeline. You swap it without changing your code.

DIRECTRUNNER	DATAFLOWRUNNER / FLINK / SPARK
Runs locally on your machine	Distributed across a cluster
Optimized for correctness, not performance	Autoscaling workers, spills to disk
Holds data in memory, for development and testing	Production scale and reliability
Where you build and debug fast	Same pipeline, one config change

Develop on the DirectRunner, then point the same code at Dataflow when you are ready for production.

Why it helps the DirectRunner enforces Beam's rules locally, so a clean local run ports to Dataflow unchanged.

THE 4 PARTS OF BEAM · 1

PCollection.

Your data in motion: the stream of elements flowing between steps.

a continuous stream of transactions, never all held in memory

P is for parallel Beam assumes it is huge and spread across many workers.

Never fully in memory elements stream through, they are not loaded all at once.

In our pipeline a PCollection is the stream of transactions.

Mental model elements are encoded by coders so they can move between workers; you never hold the whole set.

THE 4 PARTS OF BEAM · 2

PTransform.

One step. It takes a PCollection in and produces a new PCollection out.

A single step filter, enrich, classify, or route the data.

Chained you connect steps with the pipe operator, top to bottom.

In our pipeline source, retrieve, classify, route, about five transforms.

Key idea a PTransform never mutates the input; it produces a brand new PCollection that flows onward.

THE 4 PARTS OF BEAM · 3

DoFn.

Your per-element logic: the code that runs on each item. The LLM call lives here.

process(element) runs once for every element, your core logic.

setup() runs once per worker, where you load models, the cost pattern.

In our pipeline retrieve, classify, and route are each a DoFn.

Remember a DoFn can emit zero, one, or many outputs per input; it is more general than a filter.

THE 4 PARTS OF BEAM · 4

ParDo.

Runs your DoFn on every element, fanning it into many parallel copies.

You write one DoFn ParDo is the transform that applies it.

Runs in parallel the runner spins up many copies, each on a slice of the data.

Autoscaled the runner grows and shrinks the copies; you do not write that.

The split happens at runtime you author one DoFn; the runner replicates it across workers, each handling part of the stream.

THE COST PATTERN

Load once.
Infer millions of times.

load once in setup, then the worker reuses it on every event

A DOFN · THE LOAD-ONCE PATTERN

```
class LlmClassify(beam.DoFn):
  def setup(self):    # once per worker
    self.m = load_model() # pay it once

  def process(self, tx): # per event
    yield self.m.classify(tx)
```

The rookie bug loading the model in process reloads it per event; setup runs once per worker, the biggest single throughput win.

A DoFn's setup() runs once when a worker starts. Its process() runs for every event.

Load the model in setup() and you pay the expensive cost once, then reuse it for millions of events. Load it in process() and you reload a giant model on every transaction, and the pipeline crawls. This one distinction separates a toy from production. And you rarely write this by hand: RunInference applies the same load-once pattern for you, so for the model step you write a ModelHandler, not a DoFn.

ARCHITECTURE

Four steps.

Our pipeline screens credit card transactions for fraud in real time.

01
Source
Transactions arrive. Prod: Kafka or Pub/Sub.

→

02
Retrieve
Find similar known fraud patterns. RAG.

→

03
Classify
LLM verdict, risk score, and reason.

→

04
Route
Fraud to alerts. Cleared to analytics.

Four transforms, about five lines of pipeline code. The interesting work is in steps two and three.

```
In our code the whole pipeline is five chained transforms; swap the source
for ReadFromKafka and it is production-shaped.
```

STEP 2 · UNDER THE HOOD

RAG: embeddings and vector search.

RAG means fetching the relevant fraud patterns at decision time, not making the model memorize them.

a transaction flows through the steps and comes out a verdict

RETRIEVE

```
# text to a vector that captures meaning
vec = embedder.embed(tx)
# nearest known patterns in the vector DB
hits = index.search(vec, k=2)
# attach as context for the model
prompt = build_prompt(tx, hits)
```

Embedding text turned into numbers, so similar meanings sit close together.

Vector database a store you search by meaning. FAISS in the demo, Pinecone in production.

Retrieval fetch the closest patterns at decision time.

Update the data, not the model. No retraining.

The payoff a new scam appears, you add it to the vector database, and behavior changes instantly. No retraining.

BEAM-NATIVE · RUNINFERENCE

RunInference.

Beam's built-in step to serve a model in a pipeline, instead of calling it by hand in a DoFn.

Load once Beam calls your load_model once per worker and reuses it.

Batching and metrics grouped inference and instrumentation, for free.

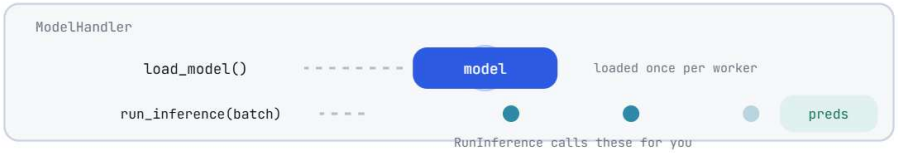
Swappable sklearn, PyTorch, Hugging Face, or a remote LLM, same step.

Why it matters you stop writing the setup/process lifecycle for the model; RunInference handles it.

BEAM-NATIVE · MODELHANDLER

ModelHandler.

The small adapter you write: how to load the model, and how to run it on a batch.



load_model() returns your model; RunInference calls it once per worker.

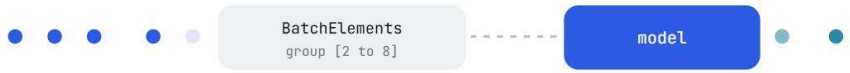
run_inference(batch) runs the model on a batch and returns predictions.

That is all you write RunInference does the rest of the plumbing.

The pairing RunInference is the transform; ModelHandler is the adapter you give it. Together they replace a hand-written DoFn for the model step.

BEAM-NATIVE · BATCHING

Batched inference.



one call for the whole batch · GPU stays busy

RunInference groups elements into batches before they hit the model, using batch_elements_kwargs to set the bounds.

Hardware that runs models is far more efficient on a batch than on one item at a time, so batching is the main throughput and cost lever. It is a different idea from routing: batching is about running several together efficiently, routing is about not sending the easy cases to the expensive model at all. Good systems use both.

2 to 8

batch size bounds in the demo, tunable for the GPU

Batching

Run many events together. Efficiency.

Routing

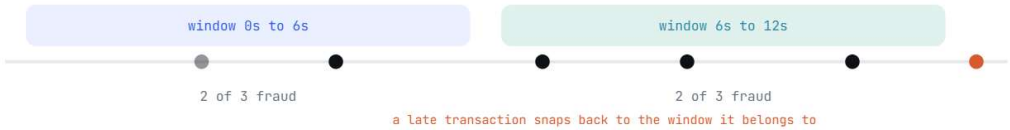
Skip the model for the obvious. Savings.

The lever bigger batches keep the GPU busy; tune the min and max batch size for your hardware.

BEAM-NATIVE · WINDOWING

Windowing by event time.

Group transactions into fixed time buckets, by when they actually happened.



Event time Beam buckets by the transaction's own timestamp using watermarks and triggers, so late or out-of-order data still lands in the correct window.

BEAM-NATIVE · METRICS

Metrics.

Counters and distributions, built into the pipeline, for monitoring its health.

Counters transactions processed, frauds detected, fallbacks.

Distributions inference latency and risk score spread.

Queried after the run exactly what you wire into a dashboard.

`Not the same as windowing` metrics measure the pipeline's health; windowing groups the data to compute a business number like the fraud rate.

BEAM-NATIVE · TAGGED OUTPUTS

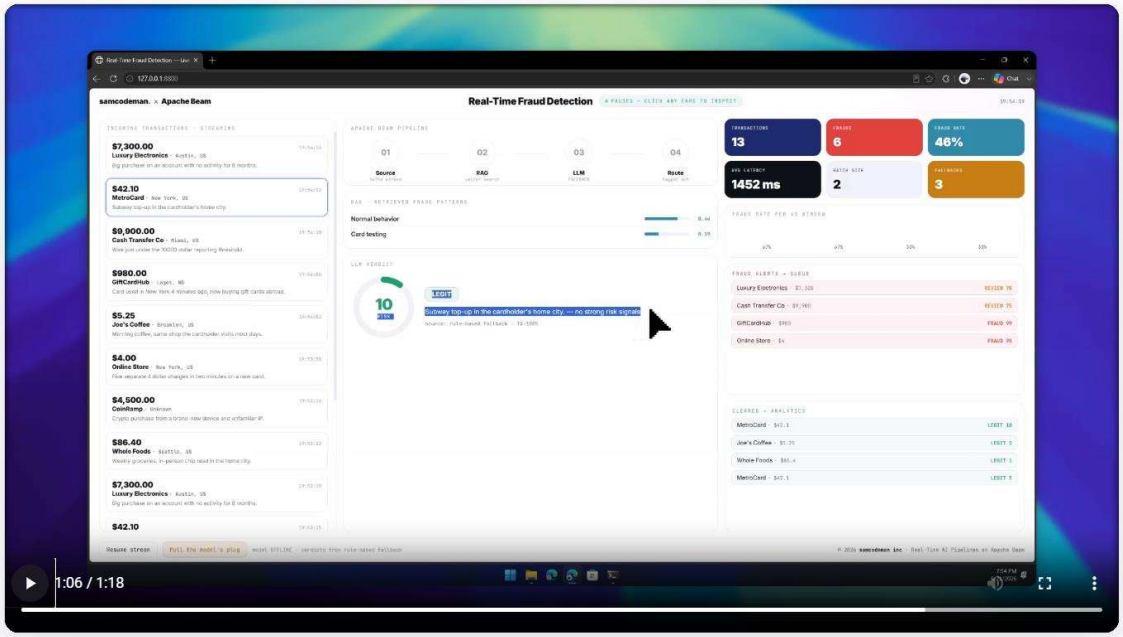
Tagged outputs.

One step, multiple labeled exits. Real pipelines fan out, they are not a straight line.

In our pipeline the route step sends fraud to an alert queue and cleared transactions to analytics, from the same DoFn.

LIVE DEMO

Classified in real time.



Watch for batched model calls, a fraud rate per time window, live metrics, and the rule-based fallback when the model is unplugged.

LATENCY, HONESTLY

The LLM is the reasoning layer, not the millisecond gate.

A fast, lightweight model makes the instant approve or decline. The LLM handles the harder cases, the review queue, and the explanations.

An LLM does not answer in milliseconds, and pretending otherwise invites a fair challenge. You make it viable with three levers, the same ones that control cost.

Batch

Run many transactions together so the GPU stays saturated.

Quantize

Lower precision means faster and cheaper, with little quality loss.

Route

Cheap rules clear the obvious, only the unsure reach the LLM.

FROM LAPTOP TO PRODUCTION

The same code that scales.

The pipeline does not change when you scale it. Only the components behind each step grow.

IN THIS DEMO	IN PRODUCTION
Timed generator source	Kafka or Pub/Sub
FAISS or numpy index	Pinecone or FAISS at scale
Local OpenAI-compatible endpoint	vLLM, or AWS Bedrock and SageMaker
DirectRunner on a laptop	DataflowRunner with Kubernetes

Same four steps. The laptop proves the pattern, the cloud runs it big.

What changes only the plug-ins: Kafka, Pinecone, vLLM or Bedrock, and the Dataflow runner. The four transforms stay identical.

TAKEAWAYS

What to remember.

01 **Beam unifies batch and streaming** one codebase, laptop to cloud.

02 **Load the model once per worker** the key cost pattern.

03 **RAG gives live context** update data, not weights.

04 **Use the Beam-native path** RunInference, batching, windowing, metrics.

05 **Design for graceful degradation** a stream must never hard-fail.

One line the model was never the hard part; the pipeline around it is, and that is what Apache Beam is for.

GRATITUDE

Thank you, Apache Beam.

None of this exists without the Apache Beam community: the unified model, the runners, and RunInference, all built and maintained in the open.

Thank you to the contributors who made serving a model a first-class step in a pipeline, to the Beam Summit organizers and volunteers for the room and the time, and to everyone here who keeps this project alive. It is a privilege to build on your work.

github.com/SamirSengupta/Apache-Beam-Summit-New-York-2026

scan for the code

LET'S CONNECT

Let's stay in touch.

I build SygentAI: production-grade agentic AI and LLM systems. If any of this resonated, I would love to trade notes. Scan to find me.

Samir Sengupta · AI/ML Engineer, SygentAI

www.samcodeman.com

github.com/SamirSengupta

linkedin.com/in/samirsengupta

samcodeman.com