NIMBUSPAY

Transforming card Issuance

Integrating Machine Learning models with Java

London, United Kindom

Tuesday 2nd July 2026

Agenda

- The Artificial Intelligence landscape — where Machine Learning sits within AI
- Building & exporting a model — Python, JupyterLab, ONNX
- Transformers
- External inference
- Embedded inference
- CPU vs GPU
- Deployment considerations

What is Artificial Intelligence?

Building computer systems that can do things we normally think of as needing human intelligence — seeing, reasoning, making decisions, and using language.

- **Machine Learning** — systems that learn patterns from data, not hard-coded rules
  - **Deep Learning** — multi-layer neural networks, a powerful subset of ML
- **Natural Language Processing** — understanding and generating human language
- **Computer Vision** — interpreting images and video
- **Robotics**, **planning** & **expert systems** — perception, action and logic-based reasoning

Focus on Machine Learning…

ML builds a mathematical model from example data so it can make predictions on new, unseen inputs. Three classic paradigms:

- **Supervised learning** — trained on labelled examples (classification, regression)
- **Unsupervised learning** — finds structure in unlabeled data (clustering, reduction)
- **Reinforcement learning** — learns through reward signals from interaction

Our goal — run a Python-trained model's inference integrated with a Java application

Real world examples

- **Fraud Detection & Prevention** (is a Transaction Fraud or Not Fraud)
  - Transaction Value; Merchant Type; Time; Location…
- **Credit Risk Management** (Credit Scoring – calculating risk of default)
  - Missed Payments; Indebtedness; Income; Outgoings; Location…

Building a Model in JupyterLab with Python

JupyterLab *gives data scientists an interactive notebook to explore, train, and evaluate models iteratively. A typical workflow:*

- Load & explore data — pandas, NumPy and matplotlib for cleaning and inspection
- Split data — into training, validation and test sets
- Train — scikit-learn for classical ML; PyTorch or TensorFlow for neural networks
- Evaluate — accuracy, precision/recall, F1 or loss on held-out data
- Iterate & export — tune until acceptable, then save the trained artifact

Compiling the Model with ONNX

ONNX (Open Neural Network Exchange) is an open, framework-neutral format that decouples the model from the framework it was trained in.

- Train in **PyTorch**, **scikit-learn** or **TensorFlow** — export once to a single .onnx file
- The file captures the computation graph, operators and learned weights
- Runs anywhere an ONNX-compatible runtime exists — including the JVM (***Portability***)
- Enables optimization passes — operator fusion and quantization — before deployment
- Becomes the contract between the Python and Java worlds, feeding both deployments

The Role of Transformers

Transformers are the architecture behind most modern language — and increasingly vision — models. Their core innovation is self-attention.

- Attention weighs how much each input element matters to every other
- Captures context and long-range relationships across a sequence
- Processes sequences in parallel — efficient to train at scale
- Foundation for LLMs, embeddings, translation and summarisation
- Exportable to ONNX like any model — but large and compute-heavy

Two ways to deploy

With the ONNX artifact in hand, two fundamentally different topologies — the choice shapes latency, scaling, and operational complexity.

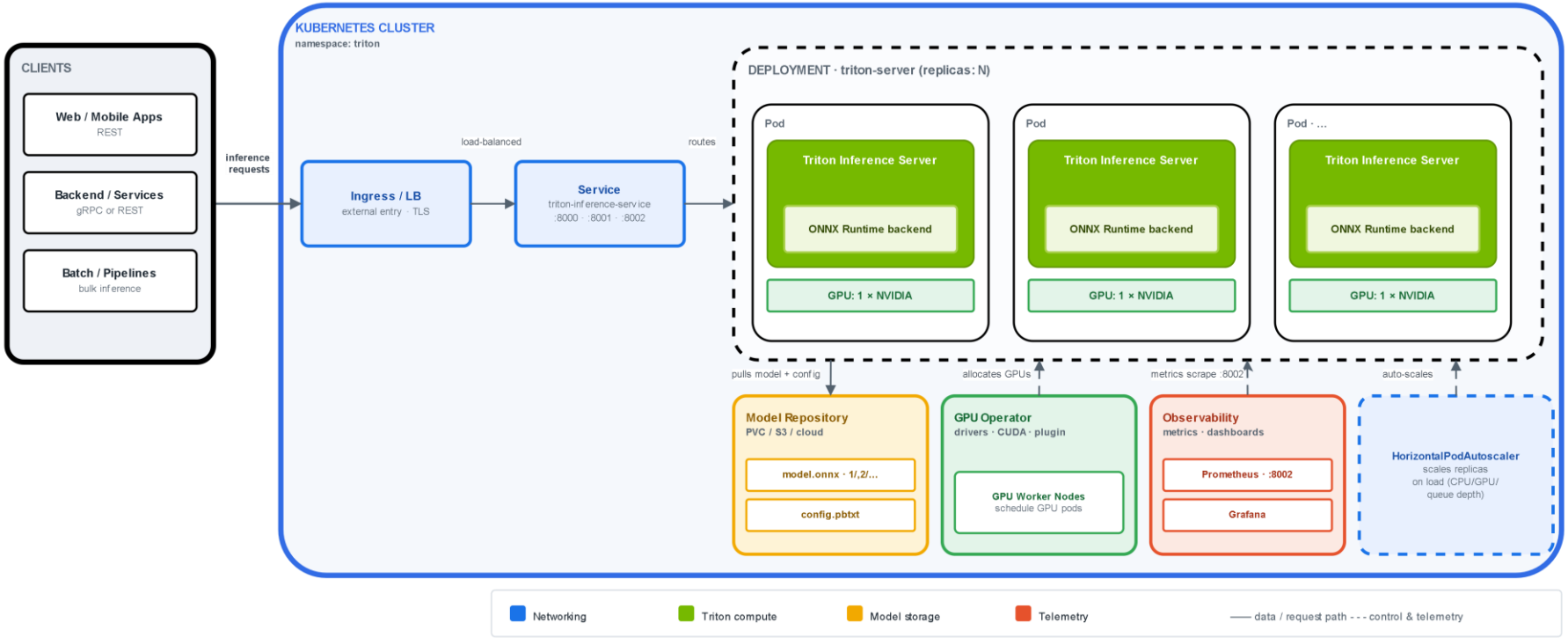
- **External Inference**
  - Model runs in Triton Inference Server as a separate service.
  - The Java application calls it over the network.
- **Embedded Inference**
  - Model runs inside the Java process via ONNX Runtime for Java.
  - No network hop — inference happens in-process.

External Inference with Triton

NVIDIA Triton Inference Server hosts models behind a standard network API; the Java app becomes a thin client.

- Java app calls Triton over **HTTP/REST** or **gRPC**
- **Triton** handles batching, model versioning and concurrent requests
- Serves many models and frameworks from a single server
- Scales independently — add GPU capacity without touching application code
- Best for large, GPU-bound or shared models updated without redeployments
- **Trade-offs:** adds a network hop and latency, plus a separate service to deploy, secure and monitor.

External Inference with Triton



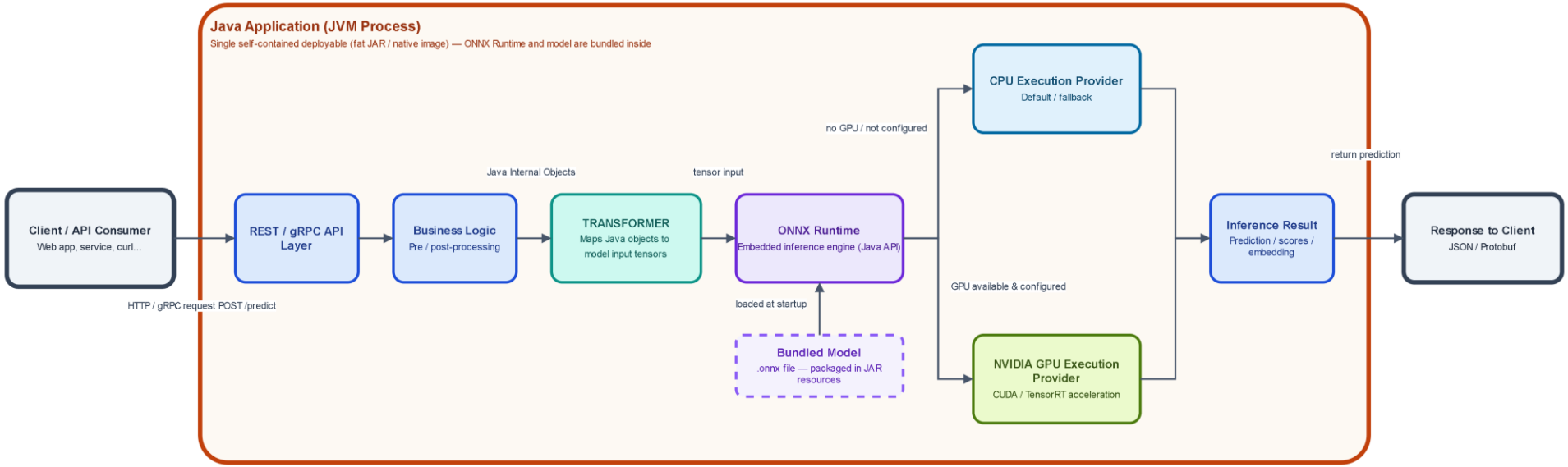
NIMBUSPAY
Transforming card Issuance

Embedded ONNX Runtime for Java

ONNX Runtime for Java loads the .onnx file directly inside the JVM and runs inference in-process — no server, no network call.

- Add the **ONNX Runtime** Java dependency via **Maven** or **Gradle**
- Load the model into an **OrtSession**; feed tensors; read predictions
- Inference shares the process with business logic — lowest possible latency
- Optional GPU acceleration via execution providers such as **CUDA**
- Best for low latency, modest models or a single self-contained artifact
- **Trade-offs:** the model ships with the app, so updates mean redeploying; heavy models compete for memory.

Embedded ONNX Runtime for Java



NIMBUSPAY
Transforming card Issuance

Inference on CPU vs GPU (Embedded Runtime)

Where the model computes its predictions strongly affects latency, throughput and cost. Both Triton and embedded runtimes can target either.

- CPU is default unless a dedicated accelerator is available — power/thermal budgets rule out discrete GPUs
- Batch size = 1 in most cases, so GPU parallelism advantage shrinks
- Memory, not compute, is the real constraint
- Quantization (INT8/INT4) and pruning matter more than hardware choice
- Lightweight runtimes drive latency more than raw FLOPS

Inference on CPU vs GPU (External Inference)

Where the model computes its predictions strongly affects latency, throughput and cost. Both Triton and embedded runtimes can target either.

- GPU wins once batching is possible — dynamic batching drives most throughput gains
- CPU remains cost-effective for small models or low-QPS services
- MIG/partitioning right-sizes GPU allocation per model vs. over-provisioning
- Preprocessing/network overhead often dominates latency more than CPU vs. GPU choice
- For low-latency, batch=1 scoring, optimized CPU inference can outperform GPU due to transfer overhead

Choosing the Right Approach

Embedded optimizes for power and latency at batch=1, while external/server optimizes for throughput and cost at scale.

- Batch size — embedded is typically batch=1; external can batch for throughput
- Latency profile — embedded prioritizes deterministic low latency; external can trade latency for volume
- Cost model — embedded is fixed per-device hardware; external is pay-for-compute, optimizable via right-sizing
- Connectivity — embedded must often run offline/on-device; external assumes network availability
- Model footprint — embedded forces quantization/pruning; external can run larger, unoptimized models if GPU-backed
- Change Frequency / Mutability – if the model changes frequently:
    - New Features = New Transformer and New Model (significant impact) – Redeployment Required
    - New Model (**consistent** features) – recommend External Inference (easier to publish change)

Bonus: NVIDIA Multi-Instance GPU (MIG)

- Objective: Drive GPU infrastructure as highly utilized as possible (TCO)
- Average GPU Utilization globally ~30%
- NVIDIA Multi-Instance GPU (MIG)
  - Partitioning a Datacenter-grade GPU into multiple isolated logical GPUs (e.g. A100, H100)
  - Alternative to Time-Slicing
- Combine with other strategies (e.g. Microbatching)
- Use-Case:
  - Inference serving – **Yes**
  - Model training – **No** (whole GPU recommended)

Thank you