RedHat
Emerging Technology Team

From Gradients to Governance

Making PyTorch Lineage-Aware

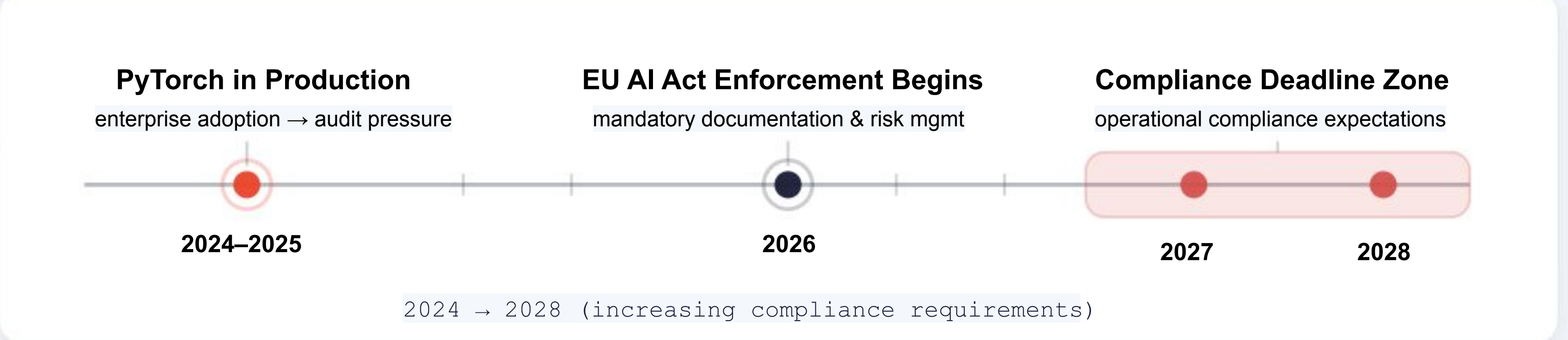
Why data sovereignty belongs in the runtime

Kateryna Romashko
Associate Software Engineer

Clodagh Walsh
Software Engineer

The Uncomfortable Reality

Enforcement and audit readiness timelines are converging with production ML adoption.



Why this matters for PyTorch deployments

Regulatory Imperative

GDPR, LGPD, and emerging AI acts demand complete auditability, consent tracking, and policy enforcement

Data is No Longer Static

Training data flows dynamically across cloud providers, geographic regions, and distributed systems in real-time

70% Prioritize Sovereignty

Enterprises now mandate data sovereignty considerations in all cloud architecture decisions and deployment blueprints

The Critical Gap

Data residency (keeping data in a region) ≠ Data sovereignty (governance following the data wherever it flows)

Takeaway: by 2027–2028, auditors will ask "what data trained this model, where did it flow, and was it allowed?"

When "Where" Matters More Than "How"

The problem isn't in the code — it's in the **origin of the data** .

Real-world lineage failures

Data Leak

ChatGPT incident

Accidental publication of private chats due to API bug. **Root cause:** lineage tracking failure — model couldn't distinguish private vs. public conversation context.

Invalid Output

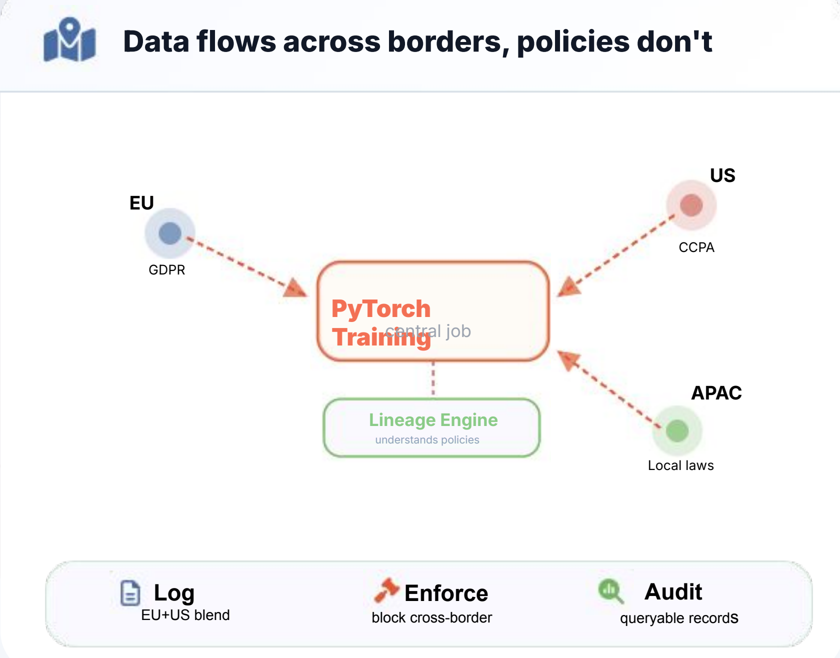
Hallucination risk

Models trained on web-scraped data (not verified government sources) start "hallucinating" fake laws and regulations. **Lineage gap:** no proof of authoritative source.

Cultural Incompatibility

Local norms

Model output violates regional norms or regulations because training data lacked provenance markers for cultural/legal context.



The problem isn't in the code — it's in the provenance of data.

Lineage isn't optional. It's the only way to know where data comes from, whether it's allowed to mix, and if the output respects regulations.

The Missing Primitive

PyTorch tracks **how** models learn. It doesn't track **whether** they should.

PYTORCH TRACKS TODAY

- ✕ **Gradients**
- ✕ **Computation graph**
- ✕ **Training loss**
- ✕ **Model parameters**

These primitives optimize learning, but don't express governance constraints.

PYTORCH SHOULD TRACK

- ✓ **Data origin** (country, dataset ID)
- ✓ **Consent boundaries** (training-only)
- ✓ **Regulatory jurisdiction** (GDPR, HIPAA)
- ✓ **Purpose limitations** (non-commercial)

These primitives enable enforcement: prevent disallowed compute before it happens.

> **Lineage is not metadata.**
> **It's a runtime constraint.**
>
> — Runtime enforcement paradigm

Why Current Tools Fall Short

Two dominant approaches, one critical blind spot: runtime enforcement.

API-Based Instrumentation

MLflow · TFX · Kubeflow

- ✓ **Flexible** — users log what they want
- ✗ **Invasive** — requires code refactoring
- ✗ **Error-prone** — users forget to log artifacts
- ✗ **Inconsistent** — different tracking across pipeline stages
- ✗ **Vendor lock-in** — changing systems requires code changes

Filesystem Metadata Extensions

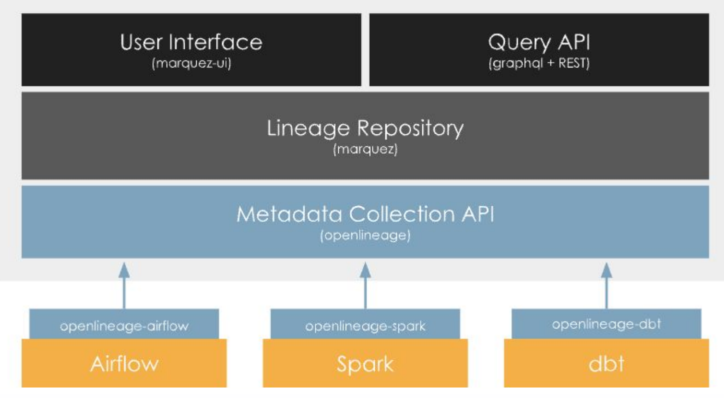
Hopsworks provenance framework

- ✓ **Automatic capture at filesystem level**
- ✓ **Consistent metadata replication**
- ✗ **Storage-layer dependent**
- ✗ **Doesn't capture computation lineage**
- ✗ **Can't enforce policy at runtime**

Existing tools solve observability. None solve runtime enforcement.

Both approaches observe after execution — they can't stop a policy violation before it happens.

OpenLineage: Lineage You Can Query After the Fact



What it is

- **Open standard** for collecting lineage metadata
- **Framework & language agnostic** event model
- **Marquez** is a popular reference implementation

What it gives you

- **Visibility** into jobs, datasets, and relationships
- **Standard schema** for audit trails & reporting
- **Foundation** for lineage-aware tooling across the pipeline

OpenLineage Integration: MLflow + Kubeflow

MLflow Tracking

```
# URI prefix activates the plugin
tracking_uri = "openlineage+http://mlflow:5000"

# Standard MLflow — no code changes
with mlflow.start_run():
  mlflow.log_param("lr", 0.01)
  mlflow.log_metric("accuracy", 0.95)
```

- `OpenLineageTrackingStore` wraps the real store (SQL, REST, file)
- `create_run()` → emits **START** event to Marquez
- Accumulates params & metrics, emits **COMPLETE** on end_run

Kubeflow Pipeline

```
@dsl.component(base_image=FKM_IMAGE)
def ds_model_training(dataset: Input[Dataset]):
  from openlineage_oai.adapters.kfp import
kfp_lineage
  with kfp_lineage("kfp-model_training",
    inputs=[dataset], outputs=[...],
    url="http://marquez"):
      # component work here
```

- `kfp_lineage` context manager inside each `@dsl.component`
- Emits **START** on enter, **COMPLETE / FAIL** on exit
- Parses `dsl.Input` / `Output` URIs into OL datasets

Marquez / OpenLineage

```
pipeline → kfp-model_training →
mlflow/run
→ datasets, params, metrics, model
artifacts
```

Events from MLflow

START on create_run, COMPLETE with accumulated params + metrics + model outputs

Events from Kubeflow

START/COMPLETE per step, input/output datasets with schema facets

Result: Unified OpenLineage standard → end-to-end data lineage across MLflow experiment tracking & KFP pipeline orchestration

github.com/rh-waterford-et/practice-mlops

Where Should Lineage Live in PyTorch?

Option A:
PyTorch as OpenLineage Producer

Dataset

Option B:
PyTorch with Internal Lineage Engine

Tensor

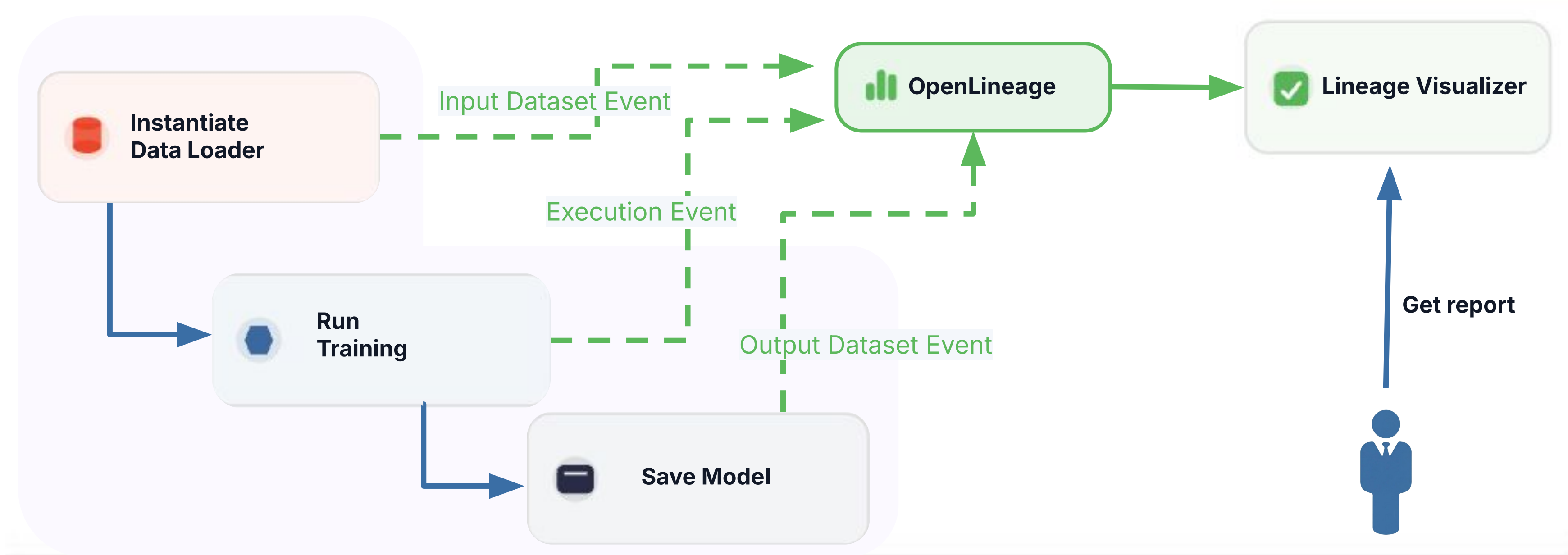
Lineage Engine

Option C:
?

Component X

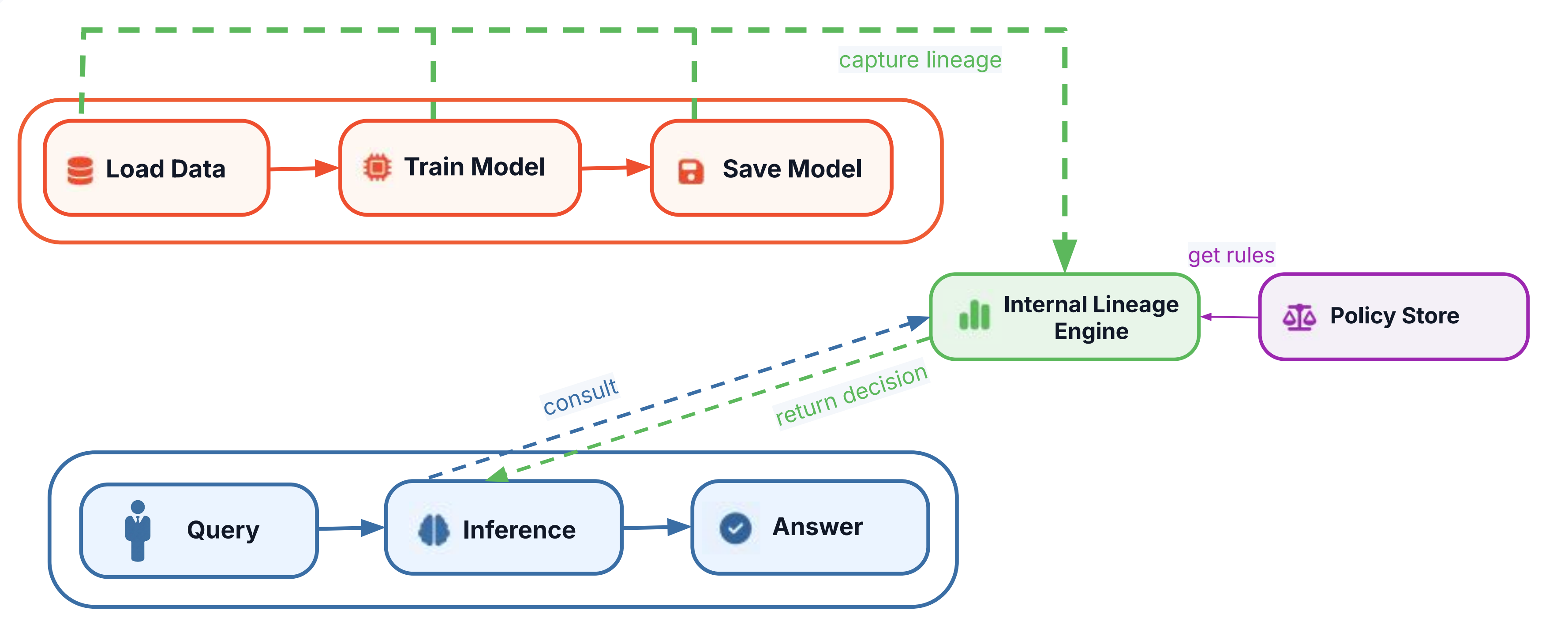
Component Y

PyTorch as OpenLineage Producer



Once PyTorch can emit these events the next step is enforcement — but this requires lineage inside the runtime.

Native Model



Tensors = Potential Lineage Carrier

- Lineage Metadata Field
- Origin
- Jurisdiction
- Consent Terms
- Pointer to Parent

Autograd != Solution

- Principle of single responsibility
- Gradient graph very different from lineage graph
- Autograd engine disabled during inference

Could either of these legacy components be reused?
PyTorch
TorchArrow
DataPipes
- TorchArrow does data preprocessing
- DataPipes produce a data graph

Let's Build This Together

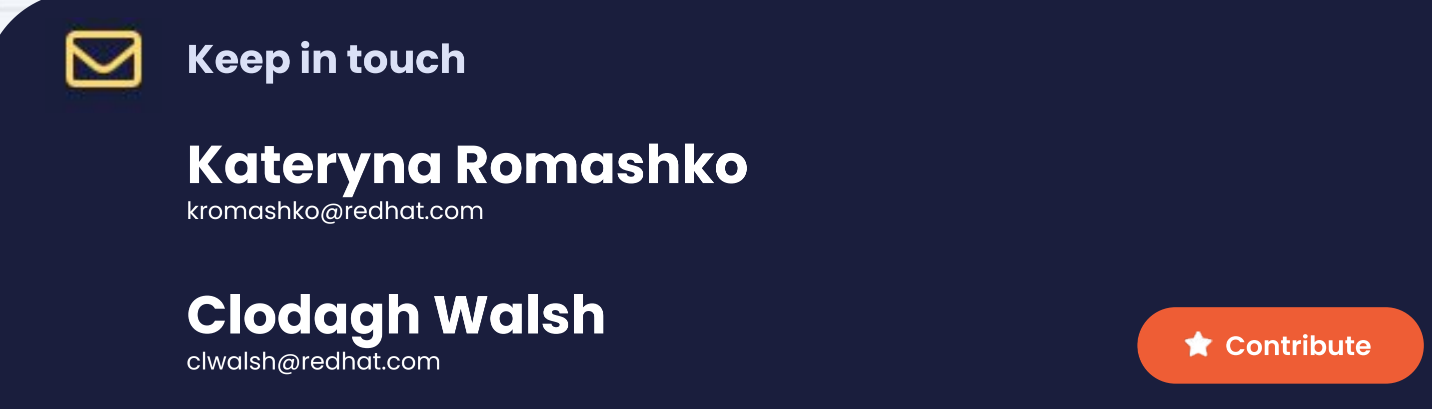
Open Questions

- ❏ Distributed environment
- ❏ Performance overhead
- ❏ Attestation

Let's continue the discussion at https://discuss.pytorch.org/t/supporting-data-lineage/224759

Thank You!

To everyone who joined the conversation — your insights shape the future of lineage in PyTorch.



✦ Let's make lineage a runtime primitive — together ✦

Red Hat Emerging Technology · Open Source · Built with the PyTorch community