Energy and Emission Monitoring Pipeline for AI Model Training

Trần Nhật Minh, Pia Höfers, Pascal Arnold

1 Problem Definition

1.1 Selected Problem

Modern machine learning model training is computationally intensive and consumes a significant amount of energy, particularly as model sizes, dataset volumes, and training durations continue to increase. Despite growing awareness of sustainability issues in the field of artificial intelligence, the energy consumption and resulting CO_2 emissions associated with model training remain largely opaque.

In practice, relevant information is fragmented across low-level hardware metrics, system monitoring tools, and cloud provider dashboards. These sources are often isolated, lack a unified data model, and do not provide an integrated view of energy consumption in relation to specific training runs, users, or models. As a result, it is difficult for practitioners and organizations to assess, compare, and ultimately reduce the environmental impact of their machine learning workloads.

This project addresses this problem by designing and implementing a streaming-based big data pipeline that continuously collects training-related system and context metrics and derives estimates of energy consumption and carbon emissions. By processing these metrics in near real time, the system aims to increase transparency and provide a foundation for monitoring and analyzing the sustainability impact of machine learning training processes.

1.2 Suitability of the Problem for Big Data Processing

The problem of monitoring energy consumption and emissions during machine learning training is highly suitable for big data technologies and stream processing architectures.

First, the data is characterized by high velocity, as system and training metrics are generated at short, regular intervals throughout the training process. Second, machine learning training jobs can run for extended periods of time, resulting in continuous, unbounded data streams rather than finite datasets. Third, the monitored information originates from multiple logical sources, including training metadata, system resource utilization, and contextual information such as users or execution environments.

In addition, the system must support scalability, as multiple training jobs may run concurrently across different machines or users. From an analytical perspective, meaningful insights require time-based aggregations, windowing operations, and

cumulative calculations, which are difficult to implement efficiently using traditional batch-oriented processing approaches.

Due to these characteristics, classical batch processing systems are insufficient. Instead, the problem necessitates a stream-first, event-driven architecture capable of ingesting, processing, and aggregating data in near real time. Technologies such as distributed message queues and structured stream processing frameworks are therefore a natural and appropriate choice.

1.3 Scope and Limitation

This project implements an end-to-end big data processing system for real-time monitoring of energy consumption and CO_2 emissions generated during machine learning model training. The primary objective is to design and deploy a stream-oriented data pipeline that is capable of ingesting high-frequency training metrics, processing them in near real time, persisting both raw and aggregated results in scalable storage systems, and making the data available for interactive visualization.
From a conceptual perspective, the project focuses on the operational monitoring layer of machine learning workloads. The system treats model training processes as external metric producers and does not aim to optimize training algorithms, influence hyperparameter selection, or improve model accuracy. Instead, it observes training executions and derives sustainability-related metrics based on emitted events.
The implemented solution covers the complete data lifecycle, including:

- metric generation during training,
- event-based ingestion via a messaging system,
- real-time stream processing and aggregation,
- persistent storage of raw and derived data,
- and interactive analytical consumption through a visualization dashboard.

The system is designed to be framework-agnostic with respect to machine learning libraries. Different training frameworks, such as scikit-learn, PyTorch, or TensorFlow, can be integrated as long as they emit metrics in the expected event format.
Several limitations remain. Energy consumption and emission values are estimated rather than directly measured, and therefore depend on the accuracy of the underlying assumptions and input metrics. Furthermore, the project uses simulated training workloads and does not integrate with real hardware-level power measurement interfaces. These limitations are accepted in favor of focusing on the design, implementation, and evaluation of a scalable big data streaming architecture.

2 Architecture and Design

2.1 Architectural Overview

The system is designed as a real-time, event-driven data pipeline following the principles of the Kappa Architecture. All data generated during machine learning

training is treated as an immutable stream of events that is processed through a single, unified streaming pipeline. No separate batch layer is maintained.

This architectural choice is well suited for continuous monitoring scenarios, where data is produced incrementally over extended periods of time. By relying exclusively on stream processing, the system avoids duplicated processing logic and reduces architectural complexity compared to Lambda-based designs.

At a high level, the architecture integrates metric generation, event-based ingestion, stream processing, persistent storage, and interactive visualization. Components are loosely coupled and communicate exclusively through well-defined interfaces, enabling scalability, fault tolerance, and independent evolution of individual layers.

Figure 1 illustrates the overall system architecture and data flow of the proposed streaming pipeline.

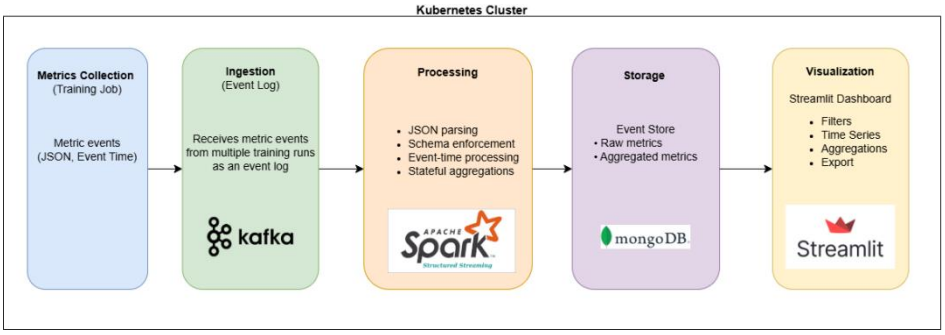

Figure 1 : Overall architecture

2.2 System Architecture Layers

2.2.1 Metrics Collection

The Metrics Collection Layer is responsible for observing machine learning training runs and emitting structured metric events. It operates inside the training process and captures training progress and system-related signals without influencing training logic or optimization decisions.

Each emitted event represents a point-in-time observation and includes training metrics (e.g., epoch, loss, accuracy), estimated energy consumption, derived CO_2 emissions, and contextual metadata such as run identifiers, model information, execution environment, and geographic region. Events are timestamped at creation time and follow event-time semantics.

Metric events are serialized as JSON and published asynchronously to a messaging system. In addition, events are written to a local file as a fallback mechanism, enabling offline inspection or replay if required.

2.2.2 Ingestion

The ingestion layer is implemented using Apache Kafka and serves as the central event transport mechanism of the system. All metric events produced by training jobs are published to a dedicated Kafka topic.

Kafka decouples metric producers from downstream consumers and provides durable, ordered, and fault-tolerant event storage. This buffering capability allows the system to absorb fluctuations in event rates and supports scalable, parallel consumption by stream processing jobs. Kafka also enables event replay, which is essential for recovery and future reprocessing scenarios.

2.2.3 Processing

The processing layer is implemented using Apache Spark Structured Streaming and continuously consumes events from Kafka.
Incoming messages are parsed into a predefined schema and processed based on event-time semantics. The streaming job performs both event-level transformations and stateful aggregations, such as time-based summaries and cumulative metrics per training run. These operations enable near real-time analysis of energy consumption and emissions over time.
Checkpointing is used to persist processing state and Kafka offsets, ensuring fault-tolerant execution and consistent recovery in case of failures. The processing results are written incrementally to persistent storage for downstream consumption.

2.2.4 Storage

The storage layer persists both raw and aggregated metric data and provides efficient access for analytical queries. MongoDB is used as the primary operational data store, storing processed events in a document-oriented format.

This storage design supports flexible schemas and low-latency read access, making it suitable for interactive dashboards and exploratory analysis. In addition, the architecture conceptually allows integration with distributed file systems for long-term archival and batch-oriented analysis, although this is not part of the core implementation.

2.2.5 Visualization

The visualization layer provides a read-only interface for monitoring energy consumption and CO_2 emissions generated during training runs. It is implemented using **Streamlit** and retrieves data directly from the operational storage layer.
The dashboard enables interactive exploration through filters, time-series visualizations, and aggregated views at run and user level. By decoupling visualization from ingestion and processing, analytical workloads do not interfere with the streaming pipeline.

2.3 Architectural design considerations

The system relies on event-time processing to ensure temporal correctness, even in the presence of delayed or out-of-order events. Stateful processing and checkpointing contribute to robustness and fault tolerance.

Scalability is achieved through the independent scaling of architectural layers. Kafka supports parallel ingestion, Spark enables distributed stream processing, and the storage layer can accommodate growing data volumes.

The selected technologies align with the course objectives and demonstrate the practical application of distributed messaging systems, stream processing frameworks, NoSQL storage, and containerized deployment environments.

3 Implementation Details

3.1 Source Code with full documentation

The complete source code of the system is available in a publicly accessible Git repository. The repository contains all components of the pipeline, including the training job, stream processing logic, infrastructure definitions, and the visualization dashboard.

The codebase is structured by functional responsibility, separating metric generation, stream processing, storage interaction, and visualization. Each component includes inline documentation and descriptive naming conventions to ensure readability and maintainability. A comprehensive README file documents the overall system architecture, setup instructions, configuration parameters, and execution steps, enabling reproducibility and ease of deployment.

Git Repository : https://github.com/piiaa01/energy-emission-pipeline

3.2 Environment-specific configuration files

All environment-dependent parameters are externalized using environment variables and Kubernetes ConfigMaps. This approach avoids hard-coded values and enables flexible deployment across different execution environments.

Configuration parameters include:

- Kafka bootstrap servers and topic names
- MongoDB connection URI, database, and collection names
- Spark checkpoint locations
- Runtime environment identifiers (e.g., local vs. cluster)

Default values are provided to support local testing, while Kubernetes-specific values are injected at deployment time. This design supports reproducibility and aligns with cloud-native configuration best practices.

3.3 Deployment strategy

All pipeline components are deployed within a Kubernetes cluster, providing a production-like execution environment with scalable and fault-tolerant services.

Component	Deployment Model
Training Job	Kubernetes Job
Kafka	Deployment with Service
Spark Streaming Job	Kubernetes Job
MongoDB	Deployment with persistent storage
Streamlit Dashboard	Deployment with Service

Kubernetes orchestrates all core services of the data pipeline, including Apache Kafka, Apache Spark Structured Streaming, MongoDB, and the Streamlit dashboard. Kafka and MongoDB are deployed with persistent volumes to ensure data durability across pod restarts. Spark is executed in cluster mode, with the driver running inside the Kubernetes cluster and dynamically managing executor pods. The Streamlit dashboard is deployed as a stateless service and accesses MongoDB in read-only mode.

The pipeline is deployed within a dedicated Kubernetes namespace to isolate resources and simplify lifecycle management. Infrastructure components are deployed first, followed by the Spark streaming job and the training job. Kubernetes handles pod scheduling, service discovery, and automatic restarts, enabling resilient and modular operation of the end-to-end pipeline.

3.4 Monitoring setup

Monitoring is implemented at multiple levels of the system.

Application-Level Monitoring

- Structured logging from training jobs, Spark streaming jobs, and the dashboard
- Explicit logging of configuration values at startup
- Visibility into Kafka ingestion via Spark input metrics

Spark Streaming Monitoring

- Spark Web UI for inspecting input rates, batch latency, task execution, and resource utilization
- Checkpointing to ensure fault tolerance and recovery of streaming state

Infrastructure-Level Monitoring

- Kubernetes-native monitoring via pod status, logs, and restart behavior
- Manual inspection using kubectl logs and kubectl rollout status

No external monitoring stack (e.g., Prometheus or Grafana) is integrated, as the focus of this project lies on pipeline design and stream processing rather than operational observability. However, the architecture is designed to support such extensions without changes to the core pipeline.

The system incorporates multiple fault-tolerance mechanisms:

- Kafka provides durable and replayable event storage
- Spark Structured Streaming ensures consistent processing through checkpointing
- MongoDB persistence protects processed data from pod failures
- Kubernetes automatically restarts failed components

Together, these mechanisms provide robust end-to-end processing suitable for continuous metric ingestion and analysis.

4 Lessons Learned

4.1 Lesson 1: Designing Robust Data Ingestion for Streaming Pipelines

Problem Description
The system ingests metric events generated by multiple training jobs and publishes them to a shared messaging infrastructure. Challenges arose from handling heterogeneous event producers, ensuring consistent event formats, and maintaining reliable ingestion without tight coupling between components. Any instability at this stage directly affected downstream processing.

Approaches Tried
Approach 1: Direct, synchronous metric emission from the training process.
Approach 2: Asynchronous event emission via Kafka with buffering.

Final Solution
An asynchronous ingestion approach using Kafka was adopted. Training jobs emit JSON-serialized metric events to a dedicated Kafka topic, decoupling metric production from downstream processing.

Key Takeaways
Asynchronous ingestion improves robustness and scalability.

4.2 Lesson 2: Managing Stateful Stream Processing with Spark

Problem Description
Processing training metrics required cumulative aggregations over time.

Approaches Tried
Stateless processing vs. stateful aggregations with checkpointing.

Final Solution
Spark Structured Streaming with checkpointing enabled.

Key Takeaways
Stateful processing is essential for meaningful streaming analytics.

4.3 Lesson 3 : Applying Event-Time Semantics in Stream Processing

Problem Description
Out-of-order event arrival affected temporal correctness.

Approaches Tried
Processing-time vs. event-time semantics.

Final Solution
Event-time processing with watermarks.

Key Takeaways
Event-time semantics ensure correct aggregations.

4.4 Lesson 4: Choosing Flexible Storage for Streaming Outputs

Problem Description
Need to store raw and aggregated data with low latency.

Approaches Tried
Rigid schemas vs. flexible document storage.

Final Solution
MongoDB as operational store.

Key Takeaways
Flexible schemas support evolving pipelines.

4.5 Lesson 5: Monitoring and Debugging Distributed Streaming Systems

Problem Description
Limited visibility across distributed components.

Approaches Tried
Logs only vs. combined Spark and Kubernetes monitoring.

Final Solution
Use Spark Web UI and Kubernetes-native tools.

Key Takeaways
Multi-layer monitoring is essential.

4.6 Lesson 6: Scaling Considerations in Streaming Architectures

Problem Description
Need to support multiple concurrent training jobs.

Approaches Tried
Fixed resources vs. scalable design.

Final Solution
Architecture supports horizontal scaling.

Key Takeaways
Designing for scalability is critical.

4.7 Lesson 7: Ensuring Reliability through Fault Tolerance Mechanisms

Problem Description
Handling failures without data loss.

Approaches Tried
Best-effort vs. built-in fault tolerance.

Final Solution
Kafka durability, Spark checkpointing, MongoDB persistence.

Key Takeaways
Fault tolerance must be addressed end-to-end.