HANOI UNIVERSITY OF SCIENCE AND TECHNOLOGY

SCHOOL OF INFORMATION AND COMMUNICATIONS TECHNOLOGY

CAPSTONE PROJECT REPORT

TOPIC:

HYDRAULIC SYSTEM ANOMALY DETECTION

Supervisor: Mr. Tran Viet Trung

Students: Pham Tran Tuan Khang - 20225503

Quach Tuan Anh - 20225469

Dinh Van Kien - 20225505

Nguyen Tran Nghia - 20225452

Pham Van Vu Hoan - 20235497

HANOI, December 28, 2025

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

CHAPTER 1. MEMBER CONTRIBUTION

This section outlines the specific technical contributions and responsibilities of each team member in the design, implementation, and deployment of the Hydraulic System Anomaly Detection platform.

1.0.1 Pham Tran Tuan Khang (Student ID: 20225503)

Role: Stream Processing, Real-time Analytics & Data Visualization.

- **Spark Structured Streaming Job:** Developed the core ETL logic in `spark_processor.py`, implementing the Kappa Architecture to unify batch and stream processing.
- **Windowing & Aggregation:** Implemented 1-minute sliding windows with 10-second triggers to calculate statistical metrics (min, max, avg, stddev) for real-time monitoring.
- **Late Data Handling:** Configured Watermarking strategies (10-second threshold) to manage state store memory and handle out-of-order events.
- **Serving Layer:** Designed the MongoDB schema with idempotent composite keys to ensure exactly-once semantics and developed PromQL queries for Grafana dashboards.

1.0.2 Quach Tuan Anh (Student ID: 20225469)

Role: Data Ingestion Layer & Kafka Management.

- **Producer Simulation:** Engineered a multi-threaded Python producer (`producer.py`) to simulate 17 concurrent sensors with varying sampling rates (1Hz - 100Hz).
- **Time Synchronization:** Implemented a ”Drift Compensation” algorithm to ensure microsecond-level timing precision for high-frequency sensor data generation.
- **Kafka Administration:** Managed Kafka topics and implemented a key-based partitioning strategy (partition by sensor name) to enable parallel processing in the downstream Spark consumer.

1.0.3 Nguyen Tran Nghia (Student ID: 20225452)

Role: Data Lake Storage & HDFS Configuration.

- **Cold Storage Architecture:** Deployed and configured HDFS (Hadoop Distributed File System) as the Data Lake for long-term archival of raw sensor

data.

- **Storage Optimization:** Enforced Apache Parquet format for efficient columnar storage and compression.
- **Write Performance:** Implemented data repartitioning strategies in Spark before writing to HDFS to solve the ”Small Files Problem” and optimize NameNode performance

1.0.4 Dinh Van Kien (Student ID: 20225505)

Role: Machine Learning Pipeline & MLOps.

- **Feature Engineering:** Developed the batch processing pipeline (`spark_trainer.py`) to transform raw time-series data from ”long format” to ”wide format” using Pivot transformations.
- **Model Training:** Implemented Random Forest Classification models using Spark MLlib to predict component health states (Cooler, Valve, Pump).
- **MLOps Integration:** Integrated MLflow to manage the model lifecycle, including experiment tracking, metric logging (Accuracy, F1-Score), and model registry.

1.0.5 Pham Van Vu Hoan (Student ID: 20235497)

Role: System Infrastructure, Kubernetes & Deployment.

- **Kubernetes Orchestration:** Developed all Kubernetes manifests (YAML) for StatefulSets, Deployments, and Services, and managed resource quotas.
- **Network Engineering:** Solved the Kafka ”Advertised Listeners” issue in dynamic container environments using shell expansion for Pod FQDNs.
- **Spark-on-K8s Networking:** Resolved Driver-Executor connectivity issues (NAT problem) by configuring dynamic Pod IP advertising.
- **Automation:** Created the `start-k8s.sh` deployment script and implemented ConfigMaps to decouple application code from Docker images.

1.0.6 Summary of Contributions

Member	**Main Responsibility**	**Key Technical Contribution**
Pham Tran Tuan Khang	Spark Streaming, Analytics, MongoDB	Window Aggregation, Watermarking logic, PromQL Dashboards.
Quach Tuan Anh	Data Ingestion, Kafka	Multi-threaded Producer, Drift Compensation Algorithm.
Nguyen Tran Nghia	HDFS, Data Lake	HDFS Architecture, Parquet Optimization, Repartitioning.
Dinh Van Kien	ML Pipeline, MLOps	Feature Engineering (Pivot), Random Forest, MLflow Registry.
Pham Van Vu Hoan	Kubernetes, Deployment	K8s StatefulSets, Dynamic Advertised Listeners, Infrastructure Code.

Table 1.1: Summary of Member Contributions

CHAPTER 2. INTRODUCTION AND PROBLEM DEFINITION

2.1 Project Overview

This project, titled **"Hydraulic System Anomaly Detection and Predictive Maintenance"**, is conducted within the framework of the Big Data Storage and Processing course. The primary objective is to design and implement a comprehensive end-to-end data processing pipeline capable of monitoring the health of industrial hydraulic systems.

To achieve this, the project adopts the **Kappa Architecture**, a stream-first processing model designed to handle high-frequency sensor data. The system is engineered to process data in real-time, enabling the immediate detection of anomalies, providing visualization of system health, and facilitating the transition towards predictive maintenance strategies.

2.2 Selected Problem

2.2.1 Industrial Context

Hydraulic systems serve as the backbone for critical operations across various sectors, including manufacturing, construction, and aerospace. These systems are comprised of complex arrangements of components such as pumps, valves, accumulators, and coolers. Given their operational importance, maintaining the reliability of these components is paramount.

2.2.2 Challenges and Consequences of Failure

Failures in hydraulic systems can lead to severe operational and economic consequences. Unplanned downtime is a significant burden, potentially costing industries between $50,000 and $500,000 per hour in lost production. Beyond financial losses, catastrophic failures, such as pressure bursts, pose serious safety hazards to workers. Furthermore, the interconnected nature of these systems means that a minor failure in a single component, like a cooler blockage, can trigger a cascading collapse, leading to the overheating and eventual failure of major components like the main pump.

2.2.3 The Shift to Predictive Maintenance

Traditionally, industries have relied on "reactive maintenance," where repairs are performed only after a breakdown occurs. However, given the high costs and risks associated with failure, there is a critical need to transition to "predictive maintenance." This approach leverages real-time data to predict component health states and perform maintenance proactively, thereby optimizing operational effi-

ciency and safety.

2.3 Analysis of Big Data Suitability

The proposed problem is considered in ”Big Data” use case, as it rigorously satisfies the three core characteristics of Volume, Velocity, and Variety (the 3Vs). The architectural decisions made in this project are driven by these specific data characteristics.

2.3.1 Velocity: High-Frequency Streams

The most defining characteristic of this system is the velocity of data generation. The monitoring system must ingest data from 17 distinct sensors simultaneously. These sensors operate at high sampling rates ranging from 1 Hz to 100 Hz, resulting in a sustained throughput of approximately 1,700 data points per second. In a continuous 24/7 operational environment, this equates to the generation of roughly 147 million records per day. Processing this swarm requires a low-latency architecture capable of detecting anomalies within seconds to prevent equipment damage.

2.3.2 Variety: Heterogeneous Data Types

The dataset exhibits significant variety, combining continuous time-series sensor data with categorical state labels. The physical measurements are diverse, including pressure (6 sensors), temperature (4 sensors), flow rate (2 sensors), vibration, and electrical power. Additionally, the data stream is enriched with metadata such as cycle IDs and timestamps, alongside categorical labels indicating the health state of key components (Cooler, Valve, Pump, and Accumulator). This heterogeneity necessitates a storage and processing solution capable of handling structured and semi-structured data effectively.

2.3.3 Volume: Data Accumulation

While individual sensor readings are small, the cumulative volume is substantial. Raw, uncompressed sensor data accumulates at a rate of approximately 500 MB per hour. Projections indicate that a single hydraulic system can generate nearly 4.4 TB of data annually. Furthermore, the generation of processed features, sliding window aggregations, and analytics results adds a significant overhead, estimated at 50 GB per month. This volume necessitates a distributed storage solution, such as HDFS, to ensure scalability and durability.

2.4 Scope and Limitations

2.4.1 Project Scope

The scope of this project encompasses the implementation of a full ”Sensor-to-Dashboard” pipeline, covering the following key stages:

1. **Ingestion**: Simulation of real-time data streams from 17 sensors into Apache Kafka.
2. **Processing**: Utilization of Apache Spark Structured Streaming within a Kappa Architecture to handle distributed stream processing.
3. **Storage**: Implementation of a polyglot persistence layer, using HDFS for raw data archival and MongoDB for serving analytics results.
4. **Analytics**: Deployment of real-time anomaly detection algorithms (Rule-based and Statistical) alongside batch Machine Learning models (Random Forest).
5. **Visualization**: Development of live monitoring dashboards using Grafana and Prometheus.
6. **Deployment**: Every tools/techstack are deployed on Kubernete cluster for scalability.

2.4.2 Limitations

While the system is designed with production-readiness in mind, certain limitations exist due to the academic and local development environment:

- **Deployment Environment**: The system is deployed on a Kubernetes Cluster (Minikube). While this provides true container orchestration, it currently runs on a single physical machine. A true production setup would span multiple physical nodes.
- **Dataset Source**: Instead of interfacing with physical hardware, the input data is streamed from the *UCI Machine Learning Repository* [1] static dataset to simulate live sensor behavior.
- **Scalability Testing**: The distributed storage (HDFS) is validated with a limited number of DataNodes (3 nodes). A production-grade deployment would typically require significantly larger clusters to handle petabyte-scale storage.
- **Security**: The project implements basic authentication mechanisms. Advanced enterprise security features, such as and SSL/TLS encryption for data-in-transit, are considered out of scope for this implementation.

3.1 Architectural Model: Kappa Architecture

For the implementation of the Hydraulic System Anomaly Detection pipeline, the **Kappa Architecture** was selected as the core design pattern. Unlike the traditional Lambda Architecture, which necessitates maintaining two separate processing paths: a ”Speed Layer” for low-latency results, and a ”Batch Layer” for historical accuracy. The Kappa Architecture simplifies the system by unifying all data processing into a single stream processing layer.

3.1.1 Justification for Kappa Architecture

The decision to adopt the Kappa Architecture over the Lambda Architecture is driven by the specific requirements of maintaining code consistency and minimizing system complexity. In the Lambda model, logic often diverges between batch (e.g., Hadoop MapReduce) and stream (e.g., Storm/Spark Streaming) engines, leading to potential discrepancies in data processing. The Kappa approach mitigates this by treating all data—both real-time and historical—as a stream.

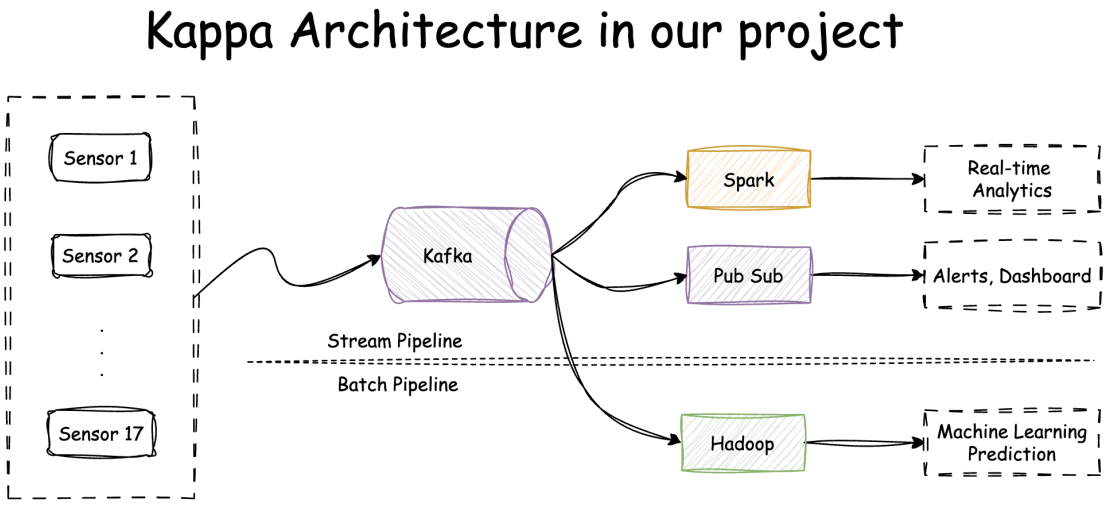

Figure 3.1: Overview of the Kappa Architecture implementation for Hydraulic System Monitoring.

Table 3.1 outlines the comparative analysis that informed this architectural decision.

3.1.2 System Architecture Diagram

The proposed system follows a linear, stream-first flow where data is ingested once and subsequently consumed by multiple independent downstream applications. The high-level data flow is illustrated in Figure 3.1.

Table 3.1: Comparison between Kappa and Lambda Architectures

Feature	**Kappa Architecture**	**Lambda Architecture**	**Reason for Choice**
Codebase	Unified (Single code path)	Duplicated (Batch + Stream)	**Maintainability**: Reduces cognitive load and eliminates code duplication.
Data Consistency	Strong (Single logic)	Eventual (Requires merge logic)	**Accuracy**: Eliminates logic divergence between layers.
Reprocessing	Replay stream	Re-run batch job	**Simplicity**: Leverages Kafka's retention to replay historical data easily.
Latency	Low (Real-time)	Mixed	**Requirement**: Anomaly detection necessitates immediate action.

3.2 Component Roles

The system architecture is orchestrated through four logical layers, each designed with specific roles to handle data ingestion, processing, storage, and analytics.

3.2.1 Infrastructure Layer (Kubernetes)

This foundational layer manages the lifecycle and networking of all distributed components.

- **Technology**: Kubernetes (K8s).
- **Role**: Acts as the Container Orchestration Engine responsible for the deployment, scaling, and management of all application containers across the cluster.
- **Key Components**:
  - **StatefulSets**: Used for stateful distributed services like Kafka and Zookeeper to maintain stable, persistent network identities (e.g., `kafka-0`, `kafka-1`) and persistent storage across restarts.
  - **Services**: Provides an abstraction layer (using ClusterIP or Headless services) that enables reliable internal DNS service discovery (e.g., allowing

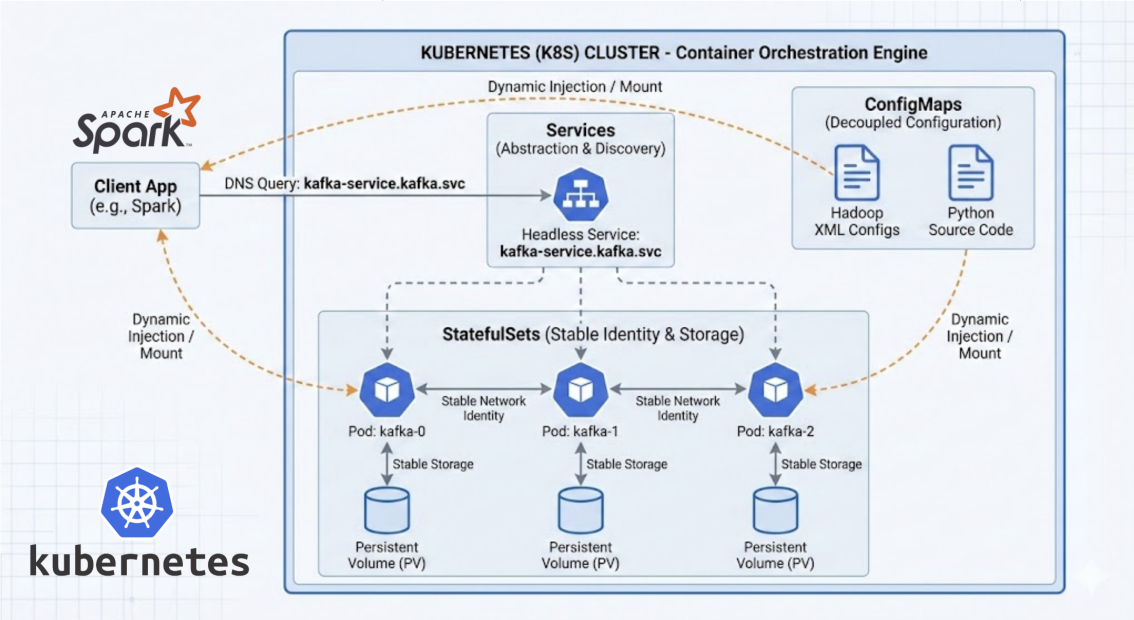

Figure 3.2: Kubernetes

Spark to find brokers via `kafka-service.kafka.svc`).

- **ConfigMaps**: Decouples configuration artifacts (such as Hadoop XML configurations and Python source code) from the container images, allowing for dynamic updates without rebuilding images.

3.2.2 Data Ingestion Layer

This layer acts as the durable buffer between high-speed sensors and the processing logic, ensuring data integrity during load spikes or downstream failures.

- **Apache Kafka (Message Broker):**
  - **Role:** Functions as a high-speed component that decouples producers (sensors) from consumers (Spark). It provides **load smoothing**, absorbing burst traffic (e.g., 1000 msg/sec) to allow stable downstream processing.
  - **Listener Configuration:** To facilitate development without compromising internal cluster security, we configured dual listeners:
    - `INTERNAL`: Optimized for Docker container-to-container communication.
    - `EXTERNAL`: Exposed for host machine access (debugging/local scripts).
- **Apache Zookeeper (The Manager):**
  - **Role:** Acts as the coordinator, tracking the health of Kafka brokers and managing partition leadership. It remains a critical dependency for ensur-

ing reliability in this specific deployment architecture.

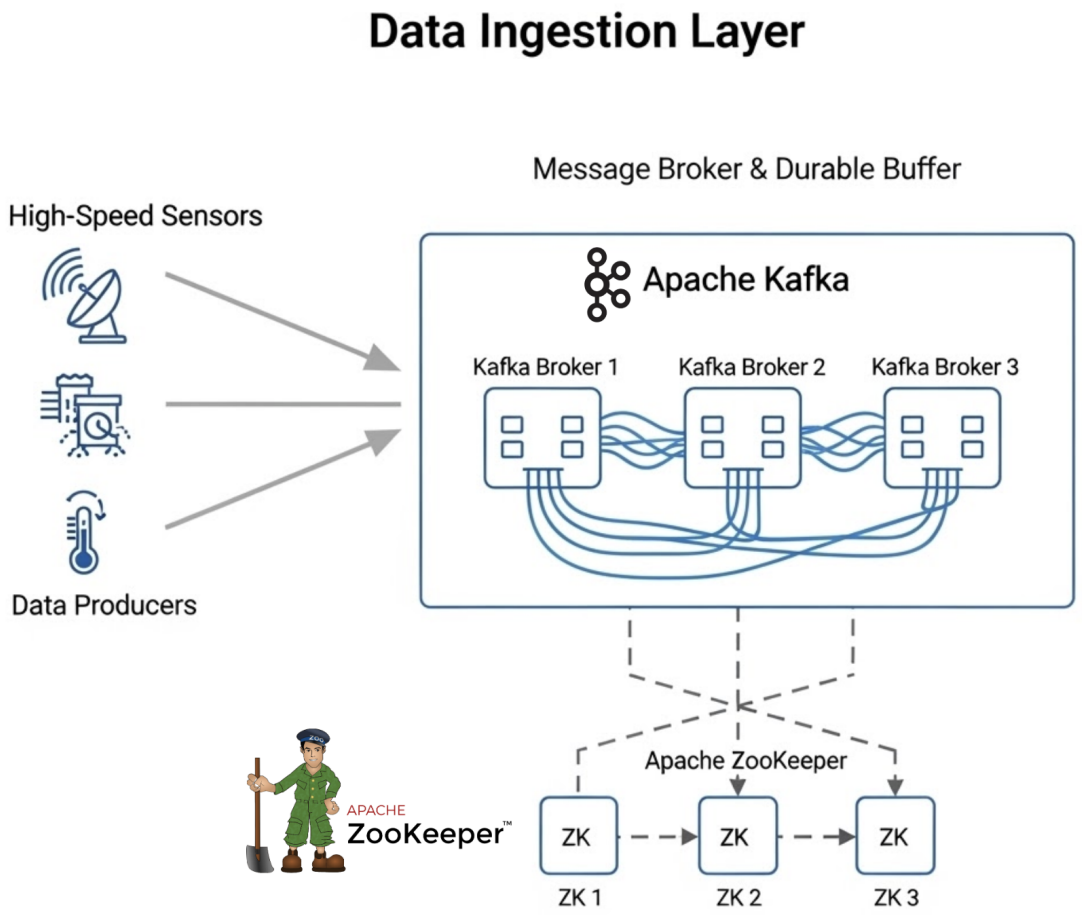

Figure 3.3: Data Ingestion Layer

3.2.3 Stream Processing Layer (The Brain)

Apache Spark Structured Streaming performs the heavy lifting, treating the never-ending stream of sensor data as an unbounded table.

- **Unified Engine:** We utilize the same code base for both real-time streaming and historical batch processing, significantly reducing maintenance overhead.
- **Key Responsibilities (ETL):**
  - **Parsing:** Converts raw JSON text into structured numerical types (Float/-Double).
  - **Windowing & Aggregation:** Groups data into time buckets (e.g., 1 minute) to calculate metrics like *average, max,* and *min* pressure.
  - **Anomaly Detection:** Applies logic rules (e.g., threshold violations) to flag dangerous states in real-time.

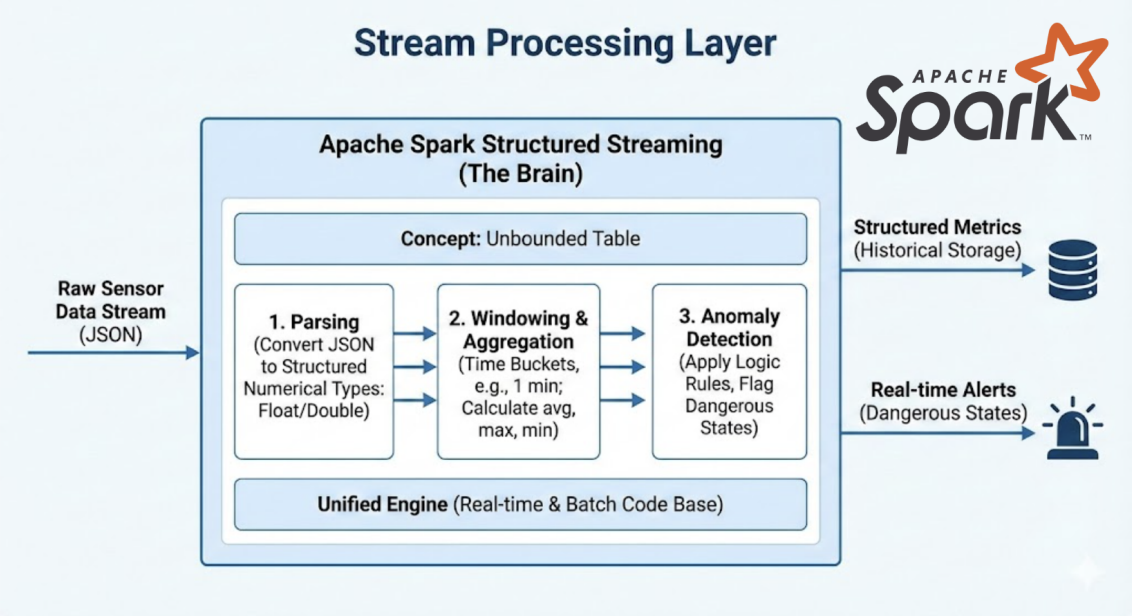

Figure 3.4: Stream Processing Layer

3.2.4 Storage Layer (Memory Systems)

We separating storage based on access patterns (Hot vs. Cold data).

- **HDFS (The "Long-Term Memory" / Data Lake):**
  - **Purpose:** Designed for *Write-Once, Read-Many* operations. It stores massive amounts of raw sensor data efficiently for future Model Training.
  - **Format:** Data is stored in **Apache Parquet**. This columnar format provides high compression and enables 100x faster reads for specific columns compared to text files.
- **MongoDB (The "Short-Term Memory" / Serving Layer):**
  - **Purpose:** Acts as the Hot Storage or Serving Layer. Unlike HDFS, MongoDB is optimized for low-latency lookups.
  - **Role:** Stores processed results to power the Dashboard, allowing instant retrieval of specific records (e.g., "Pressure at 10:05:01") in milliseconds.

3.2.5 Operations & MLOps Layer

This layer ensures system health and manages the lifecycle of machine learning models.

- **Prometheus (The Collector):** A time-series database that scrapes operational metrics (CPU usage, message rates) every 5 seconds to monitor system.
- **Grafana (The Dashboard):** A pure visualization tool that queries Prometheus

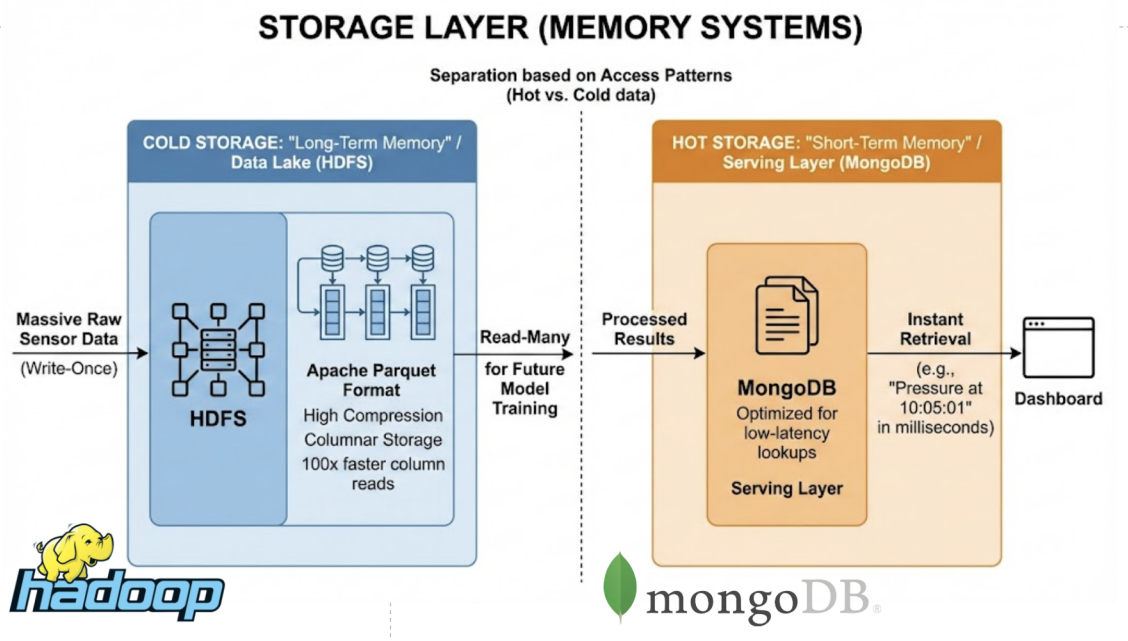

Figure 3.5: Storage layer

to display health charts and status indicators, allowing operators to assess system status at a glance.

- **MLflow (The "Lab Notebook"):**
  - **Tracking:** Records every experiment detail (code version, hyperparameters, accuracy metrics) to ensure reproducibility.
  - **Model Registry:** Manages model versioning, serving as the central repository for the "Best Model" used in production, eliminating manual file management.

3.3 Data Flow

The data flows through the system via two primary, complementary pipelines.

3.3.1 Real-Time Anomaly Detection Pipeline

The real-time pipeline is designed to detect and alert on anomalies within seconds of their occurrence.

1. **Generation**: Python-based producers simulate sensor behavior, pushing JSON events to Kafka at frequencies up to 100Hz.
2. **Ingestion**: Spark Structured Streaming subscribes to the Kafka topics and parses the incoming JSON payloads.
3. **Processing**: Spark applies window functions and evaluates data against predefined thresholds (e.g., `Pressure > 180 bar`).

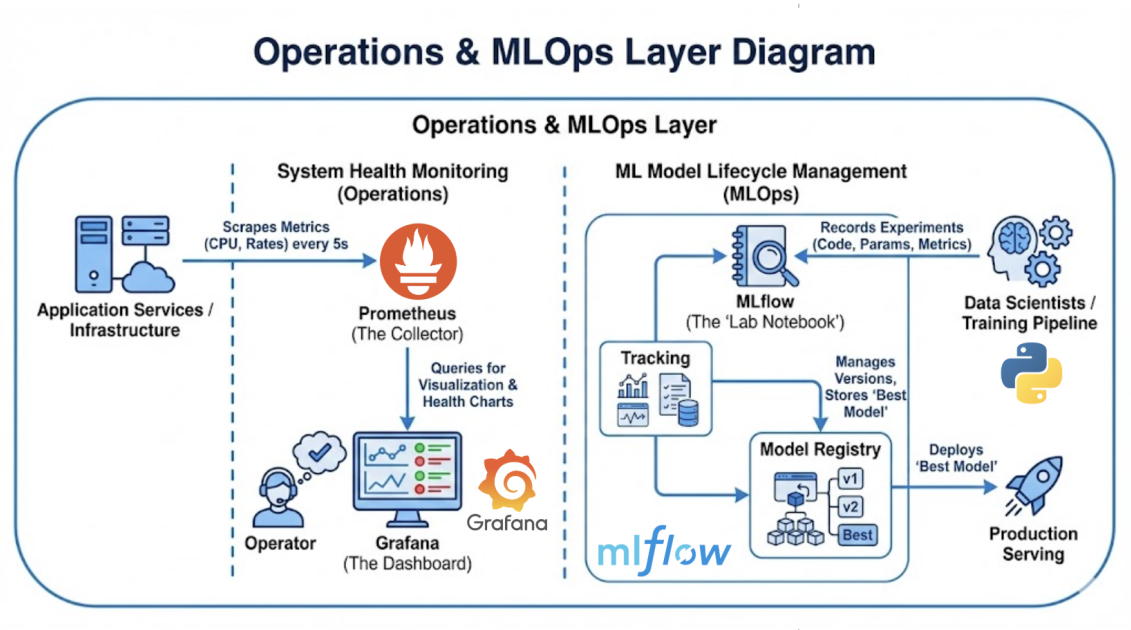

Figure 3.6: Operation of MLOPs Layers

4. **Branching**: The stream splits into two paths:
   - *Path A (Archival)*: Raw and processed data is appended to HDFS Parquet files for long-term storage.
   - *Path B (Alerting)*: Aggregated anomaly flags are written back to a secondary Kafka topic (`hydraulic-analytics`).
5. **Consumption**: Specialized consumers read the analytics topic and push metrics to the Prometheus Gateway.
6. **Visualization**: Grafana queries Prometheus to update dashboards with a latency of less than 3 seconds.

3.3.2 Batch Machine Learning Pipeline

Complementing the real-time flow, the batch pipeline focuses on predictive modeling using historical data.

1. **Trigger**: The pipeline is initiated via a scheduled cron job or manual trigger.
2. **Loading**: Spark loads historical `labels` and `raw sensor data` from HDFS.
3. **Feature Engineering**: Data is pivoted and transformed to create a "Cycle-based" feature vector, where each row represents a complete hydraulic cycle.
4. **Training**: A Random Forest Classification model is trained to predict the health state (stable/unstable) of system components.
5. **Registry**: The trained model, along with its performance metrics (Accuracy,

F1-Score), is logged to MLflow for version control and deployment management.

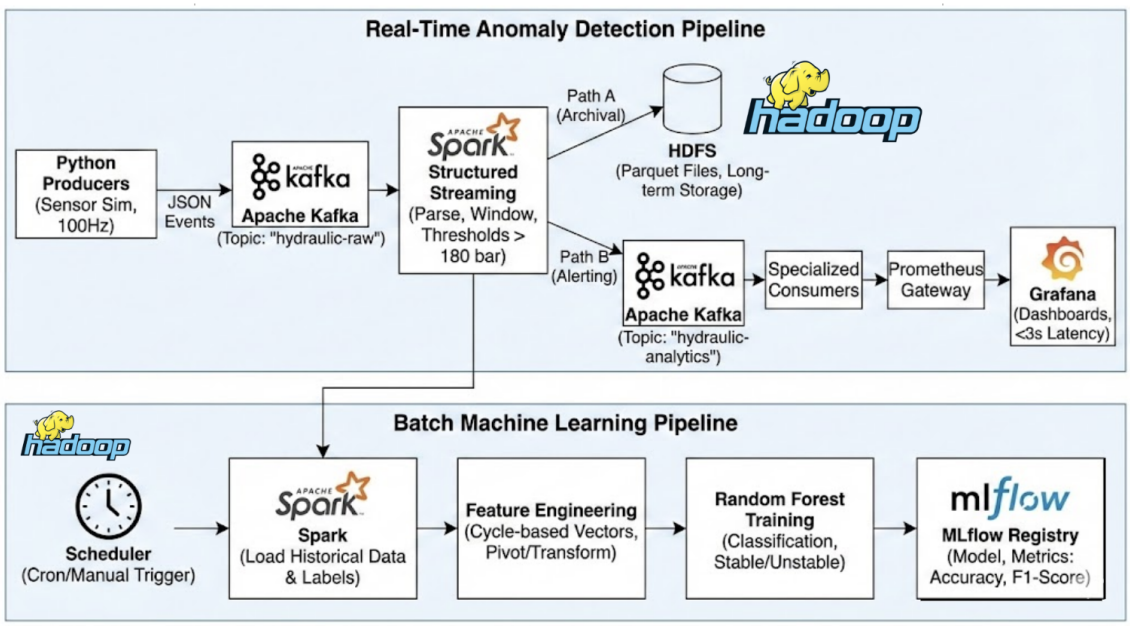

Figure 3.7: Dataflow fore real-time and batch processing

4.1 Implementation Details

This chapter provides a comprehensive technical implementation guide, documenting the exact code, configurations, and deployment strategies used to build the Hydraulic System Anomaly Detection platform.

4.1.1 Project Structure Overview

The complete codebase is organized into a modular structure separating data, source code, Spark jobs, Kubernetes configurations, and deployment scripts.

```
-Hydraulic-System-Anomaly-Detection/
|
|-- data/                           # Raw UCI Dataset (17 sensor files)
|   |-- PS1.txt                     # Pressure Sensor 1 (2205 cycles x 6000 samples)
|   |-- PS2.txt, PS3.txt, ...       # Pressure Sensors 2-6
|   |-- TS1.txt, TS2.txt, ...       # Temperature Sensors 1-4
|   |-- FS1.txt, FS2.txt            # Flow Sensors 1-2
|   |-- EPS1.txt                    # Motor Power Sensor
|   |-- VS1.txt                     # Vibration Sensor
|   |-- CE.txt, CP.txt, SE.txt      # Virtual Sensors
|   `-- profile.txt                 # Labels (Cooler/Valve/Pump/Accumulator health)
|
|-- src/                            # Python Source Code
|   |-- producer.py                 # (353 lines) Multi-threaded Kafka producer
|   |-- producer_labels.py          # Label ingestion script
|   |-- consumer_analytics.py       # (341 lines) Prometheus + MongoDB consumer
|   `-- consumer_raw.py             # Raw data consumer (deprecated)
|
|-- spark-apps/                     # Spark Applications
|   |-- spark_processor.py          # (131 lines) Streaming ETL job
|   `-- spark_trainer.py            # (147 lines) ML training pipeline
|
|-- k8s/                            # Kubernetes Manifests
|   |-- hdfs-setup/
|   |   |-- config.yaml             # Hadoop ConfigMap
```

```
|   |   |-- namenode-setup/
|   |   |   |-- deploy.yaml          # NameNode StatefulSet
|   |   |   |-- pv.yaml              # PersistentVolume
|   |   |   `-- pvc.yaml             # PersistentVolumeClaim
|   |   |-- datanode-setup/
|   |   |   `-- node0-setup/
|   |   |       `-- deploy.yaml      # DataNode Deployment
|   |   `-- spark-setup/
|   |       |-- spark-client-image.Dockerfile
|   |       |-- spark-submit.yaml    # Spark client pod
|   |       `-- spark-config.yaml
|   |
|   |-- kafka-setup/
|   |   |-- deploy.yaml              # Kafka StatefulSet + Services
|   |   |-- config.yaml              # Kafka ConfigMap
|   |   `-- pv.yaml, pvc.yaml, sc.yaml
|   |
|   |-- mongodb-setup/
|   |   |-- deploy.yaml              # MongoDB StatefulSet
|   |   `-- credential.yaml          # Secrets
|   |
|   `-- hydraulic-setup/
|       |-- app.yaml                 # Producer + Consumer Deployments
|       |-- analytics-consumer.yaml
|       |-- monitoring.yaml          # Prometheus + Grafana + Pushgateway
|       `-- deploy.sh                # Deployment automation script
|
|-- scripts/                         # Utility Scripts
|   |-- create_kafka_topics.py       # Topic initialization
|   |-- quick_test.sh                # Health check script
|   `-- reset_data.sh                # Clean environment
|
|-- config/                          # Configuration Files
|   `-- prometheus.yml               # Prometheus scrape config
|
|-- Dockerfile                       # Root Dockerfile for Python services
|-- start-k8s.sh                     # Master deployment script
|-- requirements.txt                 # Python dependencies
|-- docker-compose.khang.yml         # Docker Compose (legacy)
`-- README.md                        # Project overview
```

Listing 4.1: Project Directory Structure

a, Key File Descriptions

File	Purpose
`src/producer.py`	Simulates 17 sensors with multi-threading, precise timing control
`src/consumer_analytics.py`	Consumes Kafka analytics topic, pushes to Prometheus, writes MongoDB
`spark-apps/spark_processor.py`	Spark Structured Streaming: Reads Kafka → Watermarking → Window Aggregation → HDFS + Kafka
`spark-apps/spark_trainer.py`	ML pipeline: Feature engineering (pivot), RandomForest training, MLflow tracking
`k8s/kafka-setup/deploy.yaml`	Kafka StatefulSet with dynamic `ADVERTISED_LISTENERS` using shell expansion
`k8s/hydraulic-setup/app.yaml`	Deployments for producer and consumers with initContainers for data download
`start-k8s.sh`	Automated build → load → deploy workflow for Kind/Minikube

4.1.2 Data Ingestion Layer - Producer Implementation

a, Producer Architecture Overview

The data ingestion layer simulates 17 physical sensors by reading historical data from the UCI Machine Learning Repository dataset and streaming it to Kafka at accurate sampling rates (1Hz to 100Hz).

Design Challenge: How do we simulate 17 independent sensors running at different frequencies while maintaining cycle synchronization?

Solution: Multi-threaded producer with event-based synchronization.

b, Complete Producer Implementation

File Structure:

- `SENSOR_CONFIGS`: Dictionary mapping 17 sensors to their configs

- `SensorProducer`: Individual sensor thread class
- `HydraulicSystemProducer`: Main orchestrator

Sensor Configuration Matrix:

```
SENSOR_CONFIGS = {
    # Pressure Sensors (100Hz, 6000 samples per 60-second cycle)
     PS1 : ( PS1.txt , 100, 6000),
     PS2 : ( PS2.txt , 100, 6000),
     PS3 : ( PS3.txt , 100, 6000),
     PS4 : ( PS4.txt , 100, 6000),
     PS5 : ( PS5.txt , 100, 6000),
     PS6 : ( PS6.txt , 100, 6000),

    # Motor Power (100Hz)
     EPS1 : ( EPS1.txt , 100, 6000),

    # Volume Flow (10Hz, 600 samples per cycle)
     FS1 : ( FS1.txt , 10, 600),
     FS2 : ( FS2.txt , 10, 600),

    # Temperature (1Hz, 60 samples per cycle)
     TS1 : ( TS1.txt , 1, 60),
     TS2 : ( TS2.txt , 1, 60),
     TS3 : ( TS3.txt , 1, 60),
     TS4 : ( TS4.txt , 1, 60),

    # Vibration (1Hz)
     VS1 : ( VS1.txt , 1, 60),

    # Virtual Sensors (1Hz)
     CE : ( CE.txt , 1, 60),
     CP : ( CP.txt , 1, 60),
     SE : ( SE.txt , 1, 60),
}
```

c, Timing Precision Implementation

Critical Requirement: A 100Hz sensor must emit exactly every 10 milliseconds. Any drift accumulates and destroys synchronization.

Implementation Strategy: Drift Compensation Algorithm

```
class SensorProducer:
    def __init__(self, sensor_name, filename, sampling_rate_hz,
    num_samples, cycle_idx=0):
```

```
        self.sensor_name = sensor_name
        self.sampling_rate_hz = sampling_rate_hz
        # KEY: Calculate precise interval
        self.interval = 1.0 / sampling_rate_hz  # 100Hz = 0.01s
    = 10ms
        self.topic = f hydraulic-{sensor_name}

    def produce(self):
        # Record absolute start time
        start_time = time.time()

        for idx, value in enumerate(self.data):
            # Calculate when THIS sample should be sent
            expected_time = start_time + (idx * self.interval)

            # Drift correction: Sleep only the remaining time
            sleep_time = expected_time - time.time()
            if sleep_time > 0:
                time.sleep(sleep_time)

            # Create JSON message
            message = {
                 sensor : self.sensor_name,
                 cycle : self.cycle_idx,
                 sample_idx : idx,
                 value : value,
                 timestamp : datetime.now().isoformat(),
                 sampling_rate_hz : self.sampling_rate_hz
            }

            self.producer.send(self.topic, value=message)
```

Why this works:

- At sample 0: `expected_time = start_time + 0` → send immediately
- At sample 100: `expected_time = start_time + 1.0s` → if we've drifted to 1.02s, we catch up.
- Without this, each `sleep(0.01)` compounds error (Python sleep is not perfectly accurate).

d, Kafka Connection with Retry Logic

```
def connect_kafka(self):
    max_retries = 5
```

```
    for attempt in range(max_retries):
        try:
            self.producer = KafkaProducer(
                bootstrap_servers=KAFKA_BROKER,  #  localhost
    :9092
                value_serializer=lambda v: json.dumps(v).encode
    ('utf-8'),
                acks=1  # Wait for leader acknowledgment (
    balance between speed and durability)
            )
            print(f [{self.sensor_name}] Connected to Kafka )
            return True
        except NoBrokersAvailable:
            if attempt < max_retries - 1:
                print(f Retry {attempt + 1}/{max_retries}... )
                time.sleep(2)
            else:
                return False
```

e, Multi-Threading Orchestration

```
class HydraulicSystemProducer:
    def start(self):
        threads = []
        for producer in self.producers:
            thread = threading.Thread(target=producer.produce,
    name=producer.sensor_name)
            thread.start()
            threads.append(thread)
            time.sleep(0.01)  # Stagger starts to avoid network
     spike

        # Wait for all threads to complete
        for thread in threads:
            thread.join()
```

Total Throughput Calculation:

- 6 Pressure sensors $\times$ 100 samples/sec = 600 msg/sec
- 1 Power sensor $\times$ 100 samples/sec = 100 msg/sec
- 2 Flow sensors $\times$ 10 samples/sec = 20 msg/sec
- 8 Other sensors $\times$ 1 sample/sec = 8 msg/sec
- **Total = $\approx$ 728 messages/second per cycle**

4.1.3 Stream Processing Layer - Spark Implementation

a, Spark Structured Streaming Job Architecture

File: `spark-apps/spark_processor.py` (131 lines)

Purpose:

1. Consume raw sensor data from 17 Kafka topics
2. Apply watermarking for late data tolerance
3. Compute sliding window aggregations (1-minute window, 10-second slide)
4. Write results to HDFS (Parquet) AND back to Kafka (analytics topic)

b, Complete Streaming Job Implementation

```
from pyspark.sql import SparkSession
from pyspark.sql.functions import *
from pyspark.sql.types import *

def main():
    # Initialize Spark Session
    spark = SparkSession.builder \
        .appName( HydraulicSystemAnalytics ) \
        .config( spark.sql.streaming.checkpointLocation ,  /tmp/spark-checkpoints ) \
        .getOrCreate()

    # SCHEMA DEFINITION (Critical for avoiding inference overhead)
    schema = StructType([
        StructField( sensor , StringType()),
        StructField( cycle , IntegerType()),
        StructField( sample_idx , IntegerType()),
        StructField( value , DoubleType()),
        StructField( timestamp , StringType()),  # ISO 8601 format
        StructField( sampling_rate_hz , IntegerType())
    ])

    schema_labels = StructType([
        StructField( cycle , IntegerType()),
        StructField( label_cooler , IntegerType()),
        StructField( label_valve , IntegerType()),
        StructField( label_pump , IntegerType()),
        StructField( label_accumulator , IntegerType()),
        StructField( label_stable , IntegerType()),
```

```
        StructField( timestamp , DoubleType())
    ])
```

c, Kafka Source Configuration

```
    # Read from ALL sensor topics using pattern matching
    df = spark.readStream \
        .format( kafka ) \
        .option( kafka.bootstrap.servers ,  kafka:29092 ) \  # Docker internal network
        .option( subscribePattern ,  hydraulic-.* ) \  # Match hydraulic-PS1, hydraulic-TS1, etc.
        .option( startingOffsets ,  latest ) \  # Only process new data
        .load()

    # Parse JSON payload
    parsed = df.select(
        from_json(col( value ).cast( string ), schema).alias( data ),
        col( timestamp ).alias( kafka_timestamp )
    ).select( data.* ) \
     .withColumn( timestamp , to_timestamp(col( timestamp ))) # Convert ISO string to Timestamp type
```

Why `subscribePattern`? We have 17 topics (`hydraulic-PS1`, `hydraulic-PS2`, ..., `hydraulic-SE`). Instead of listing all 17, we use a regex pattern.

d, Watermarking - Handling Late Data

```
    # WATERMARK: Allow 10 seconds of lateness
    watermarked = parsed.withWatermark( timestamp ,  10 seconds )
```

What this means:

- If a message with timestamp `10:00:05` arrives at processing time `10:00:20`, it's **15 seconds late**.
- Our watermark is 10 seconds, so this message is **DROPPED**.
- Messages up to 10 seconds late are **ACCEPTED** and included in their correct window.

Why 10 seconds? Too small (1 sec) → lose data from network hiccups. Too large (5 min) → state grows indefinitely, memory explodes.

e, Window Aggregation Logic

```
    aggregated = watermarked \
        .groupBy(
            window(col( timestamp ),  1 minutes ,  10 seconds )
    ,  # Sliding window
            col( sensor )
        ) \
        .agg(
            avg( value ).alias( avg_value ),
            max( value ).alias( max_value ),
            min( value ).alias( min_value ),
            stddev_samp( value ).alias( stddev_value ),
            count( value ).alias( sample_count )
        )
```

Window Mechanics:

- **Size:** 1 minute (60 seconds)
- **Slide:** 10 seconds
- **Result:** Every 10 seconds, we emit a new aggregate covering the last 60 seconds.

f, Dual Output Paths

Path 1: Write to HDFS for Archival

```
    query_hdfs = parsed.writeStream \
        .format( parquet ) \
        .option( path ,  hdfs://namenode:9000/hydraulic/raw ) \
        .option( checkpointLocation ,  /tmp/spark-checkpoints/
    hdfs_raw ) \
        .partitionBy( sensor ) \  # Create /hydraulic/raw/
    sensor=PS1/, sensor=PS2/, ...
        .outputMode( append ) \
        .start()
```

Path 2: Write Analytics to Kafka

```
    output = aggregated.select(
        col( sensor ).alias( key ),  # Kafka partition key
        to_json(struct(
            col( sensor ),
            col( window.start ).alias( window_start ),
            col( window.end ).alias( window_end ),
            col( avg_value ),
            col( max_value ),
            col( min_value ),
```

```
            col( stddev_value ),
            (col( max_value ) - col( min_value )).alias( range_value ),
            col( sample_count )
        )).alias( value )
    )

    query_analytics = output.writeStream \
        .format( kafka ) \
        .option( kafka.bootstrap.servers ,  kafka:29092 ) \
        .option( topic ,  hydraulic-analytics ) \
        .option( checkpointLocation ,  /tmp/spark-checkpoints/analytics ) \
        .outputMode( update ) \  # Only send updated aggregates
        .start()
```

g, Exactly-Once Semantics

Achieved through:

1. **Checkpointing:** Stores processed Kafka offsets.
2. **Idempotent Sinks:** MongoDB uses composite key `_id = sensor_window`.

h, Labels Stream Processing (Parallel Pipe)

While processing sensor data, the job also ingests labels from a separate Kafka topic.

```
    # Read Labels from Kafka
    df_labels = spark.readStream \
        .format( kafka ) \
        .option( kafka.bootstrap.servers ,  kafka:29092 ) \
        .option( subscribe ,  hydraulic-labels ) \
        .option( startingOffsets ,  earliest ) \
        .load()

    parsed_labels = df_labels.select(
        from_json(col( value ).cast( string ), schema_labels).alias( data )
    ).select( data.* )

    # Write Labels to HDFS for ML Training
    query_labels = parsed_labels.writeStream \
        .format( parquet ) \
        .option( path ,  hdfs://namenode:9000/hydraulic/labels ) \
        .option( checkpointLocation ,  /tmp/spark-checkpoints/
```

```
    labels ) \
18          .trigger(processingTime='5 seconds') \
19          .outputMode( append ) \
20          .start()
21
22      # Block until ANY stream terminates
23      spark.streams.awaitAnyTermination()
```

i, Spark Job Submission on Kubernetes

To submit the streaming job to the cluster, we use `spark-submit` from the client pod.

Step 1: Access Spark Client Pod

```
1 kubectl exec -it spark-submit-client -n hdfs -- /bin/bash
```

Step 2: Submit Job with Production Configuration

```
1  /opt/spark/bin/spark-submit \
2    --master yarn \
3    --deploy-mode client \
4    --packages org.apache.spark:spark-sql-kafka-0-10_2.12:3.5.0 \
5    --conf spark.driver.host=$(hostname -i) \
6    --conf spark.executor.memory=2g \
7    --conf spark.driver.memory=1g \
8    --conf spark.executor.cores=2 \
9    --conf spark.sql.shuffle.partitions=17 \
10   --conf spark.streaming.stopGracefullyOnShutdown=true \
11   /app/spark-apps/spark_processor.py
```

Configuration Deep Dive:

Parameter	Value	Critical Reason
`--master yarn`	YARN	Uses HDFS ResourceManager for distributed coordination
`--deploy-mode client`	Client	Keeps the Driver in the `spark-submit` pod so we can see logs directly via `kubectl logs`

Parameter	Value	Critical Reason
`spark.driver.host`	`$(hostname -i)`	**CRITICAL:** In K8s, pods have dynamic IPs. This forces the driver to advertise its correct Pod IP so Executors can connect back.
`spark.sql.shuffle.partitions`	17	Tuned to match the number of Kafka partitions (1 per sensor) for 1:1 mapping.
`stopGracefullyOnShutdown`	true	Ensures Spark commits the latest batch offset to the checkpoint before exiting.

j, Monitoring via Spark UI

The Spark UI provides real-time visibility into the streaming metrics.

Access UI:

```
# Forward port 4040 from the client pod to localhost
kubectl port-forward pod/spark-submit-client 4040:4040 -n hdfs
# Open Browser: http://localhost:4040
```

Key Metrics to Watch:

1. **Streaming Tab:**
   - **Input Rate:** Should match producer rate (≈ 728 records/sec).
   - **Processing Rate:** Must be ¿ Input Rate. If lower, the system is lagging.
   - **Batch Duration:** Should be consistently ¡ 10 seconds (our trigger interval).
2. **SQL Tab:** Should show **3 Active Queries** (Sensor Data, Analytics Aggregation, Labels).
3. **Executors Tab: Task Time vs GC Time:** Garbage Collection should be ¡ 10% of total time. If higher, increase `spark.executor.memory`.

k, Troubleshooting Common Issues

Issue 1: "Connection to node -1 could not be established"

- **Cause:** Networking misconfiguration. Spark cannot resolve the Kafka broker address.

- **Fix:** Verify `spark-processor.py` uses `kafka:29092` (Internal K8s listener) and NOT `localhost:9092`.

Issue 2: State Store memory explosion (OOM)

- **Cause:** Late data arriving without a watermark limits state cleanup.
- **Fix:** Ensure `.withWatermark("timestamp", "10 seconds")` is applied before aggregation.

Issue 3: Checkpoint Conflicts

- **Cause:** Changing the code/schema makes old checkpoints incompatible.
- **Fix:** Manually delete the checkpoint directory: `hdfs dfs -rm -r /tmp/spark-che`

4.1.4 Analytics Consumer - Prometheus Integration

a, Consumer Architecture

File: `src/consumer_analytics.py` (341 lines)

Pattern: Observer Pattern with dual-write to Prometheus + MongoDB

```
class PrometheusConsumer(BaseConsumer):
    def __init__(self, pushgateway_url: str =  localhost:9091 ):
    :
        super().__init__( prometheus-analytics )
        self.pushgateway_url = pushgateway_url

        # Create Prometheus Metrics
        from prometheus_client import CollectorRegistry, Gauge,
    Counter

        self.registry = CollectorRegistry()

        # Metric 1: Sensor Averages
        self.sensor_avg = Gauge(
            'hydraulic_sensor_avg_1m',
            '1-minute moving average',
            ['sensor'],  # Label: PS1, PS2, etc.
            registry=self.registry
        )

        # Metric 2: Anomaly Status
        self.anomaly_status = Gauge(
            'hydraulic_anomaly_status',
            'Anomaly Detected (1=Yes, 0=No)',
            ['sensor', 'type'],  # Labels: sensor=PS1, type=
    high_pressure
```

```
            registry=self.registry
        )

        # Metric 3: Health Score
        self.system_health_score = Gauge(
            'hydraulic_sensor_health_score',
            'Health Score (0-100)',
            ['sensor'],
            registry=self.registry
        )
```

b, Anomaly Detection Rules Implementation

```
def check_anomalies(self, sensor, data):
    max_val = data.get('max_value', 0)
    range_val = data.get('range_value', 0)

    anomalies = []

    # Rule 1: Critical High Pressure
    if sensor == 'PS1' and max_val > 300:
        anomalies.append('high_pressure')
        print(f CRITICAL: Pressure spike in {sensor}: {max_val} bar )

    # Rule 2: High Temperature
    if sensor == 'TS1' and max_val > 100:
        anomalies.append('high_temperature')

    # Rule 3: Pressure Volatility (Spike Detection)
    if sensor.startswith('PS') and range_val > 50:
        anomalies.append('pressure_spike')

    # Reset all status gauges to 0 first (prevents stale alerts)
    relevant_types = ['high_pressure', 'pressure_spike'] if sensor.startswith('PS') else []
    for anom_type in relevant_types:
        self.anomaly_status.labels(sensor=sensor, type=anom_type).set(0)

    # Set detected anomalies to 1
    for anom_type in anomalies:
        self.anomaly_status.labels(sensor=sensor, type=anom_type).set(1)
```

```
28          self.anomaly_total.labels(sensor=sensor, type=anom_type
    ).inc()
29
30      # Calculate Health Score
31      sensor_health = 100
32      for anom_type in anomalies:
33          if 'spike' in anom_type:
34              sensor_health -= 10  # Warning level
35          else:
36              sensor_health -= 20  # Critical level
37
38      self.system_health_score.labels(sensor=sensor).set(max(0,
    sensor_health))
```

c, Pushgateway Integration

Why Pushgateway? Our consumer is a Python script, not a web server. Prometheus cannot ”scrape” it. We must ”push”.

```
1  def push_metrics(self):
2      try:
3          self.push_to_gateway(
4              self.pushgateway_url,  #  localhost:9091
5              job=PROMETHEUS_JOB_NAME,  #  hydraulic_spark
6              registry=self.registry
7          )
8      except Exception as e:
9          print(f Failed to push metrics: {e} )
10
11 def process_message(self, message):
12     data = message.value
13     sensor = data.get('sensor')
14
15     # Update Prometheus metrics
16     self.sensor_avg.labels(sensor=sensor).set(data['avg_value'
    ])
17     self.sensor_max.labels(sensor=sensor).set(data['max_value'
    ])
18
19     # Check for anomalies
20     self.check_anomalies(sensor, data)
21
22     # Push every 2 seconds (batching for efficiency)
23     current_time = time.time()
24     if current_time - self.last_push_time >= 2:
25         self.push_metrics()
```

```
26          self.last_push_time = current_time
```

4.1.5 Machine Learning Pipeline

a, Training Job Architecture

File: `spark-apps/spark_trainer.py` (147 lines)

Goal: Train a Random Forest classifier to predict ”Cooler Health” (3 classes: intact, reduced efficiency, close to failure).

b, Feature Engineering - The Pivot Transformation

Problem: Raw data is ”long format”.

```
| cycle | sensor | value |
|-------|--------|-------|
| 0     | PS1    | 158.2 |
| 0     | PS2    | 104.5 |
| 0     | TS1    | 35.8  |
```

Need: ”Wide format” for ML.

```
| cycle | PS1_mean | PS2_mean | TS1_mean | label_cooler |
|-------|----------|----------|----------|--------------|
| 0     | 158.2    | 104.5    | 35.8     | 3            |
```

Implementation:

```
1  def main():
2      spark = SparkSession.builder.appName(
       HydraulicSystemTrainer ).getOrCreate()
3
4      # Load raw data from HDFS
5      df_sensors = spark.read.parquet( hdfs://namenode:9000/
       hydraulic/raw )
6      df_labels = spark.read.parquet( hdfs://namenode:9000/
       hydraulic/labels )
7
8      # Step 1: Aggregate statistics per sensor per cycle
9      sensor_stats = df_sensors.groupBy( cycle ,  sensor ) \
10         .agg(
11             mean( value ).alias( mean_val ),
12             stddev( value ).alias( std_val )
13         )
14
15     # Step 2: PIVOT - Transform rows to columns
16     features_per_cycle = sensor_stats.groupBy( cycle ) \
```

```
        .pivot( sensor ) \  # Creates columns: PS1, PS2, TS1, ...
        .agg(
            first( mean_val ).alias( mean ),
            first( std_val ).alias( std )
        )

    # Result columns: [cycle, PS1_mean, PS1_std, PS2_mean, PS2_std, ...]
```

Count: 17 sensors $\times$ 2 stats (mean, std) = **34 features**.

c, Data Preparation

```
    # Join with labels
    data = features_per_cycle.join(df_labels, on= cycle ).na.fill(0)

    # Vectorize features
    label_cols = [ label_cooler ,  label_valve ,  label_pump ,  label_accumulator ]
    feature_cols = [c for c in data.columns if c not in [ cycle ,  timestamp ] + label_cols]

    assembler = VectorAssembler(inputCols=feature_cols, outputCol= features )
    data_vec = assembler.transform(data)

    # Index target label (convert 3, 20, 100 -> 0, 1, 2)
    indexer = StringIndexer(inputCol= label_cooler , outputCol= label_cooler_indexed )
    data_final = indexer.fit(data_vec).transform(data_vec)

    # Split: 80% train, 20% test
    train_df, test_df = data_final.randomSplit([0.8, 0.2], seed=42)
```

d, Model Training with MLflow

```
import mlflow
import mlflow.spark

mlflow.set_tracking_uri( http://sentiman_mlflow:5000 )
mlflow.set_experiment( hydraulic_system_health_advanced )

def train_and_log_model(spark, train_df, test_df, classifier, model_name, feature_cols):
```

```
    with mlflow.start_run(run_name=model_name):
        # Train
        model = classifier.fit(train_df)

        # Predict
        predictions = model.transform(test_df)

        # Evaluate
        evaluator = MulticlassClassificationEvaluator(
            labelCol= label_cooler_indexed ,
            predictionCol= prediction ,
            metricName= accuracy
        )
        accuracy = evaluator.evaluate(predictions)

        # Log to MLflow
        mlflow.log_param( model_type , model_name)
        mlflow.log_param( num_features , len(feature_cols))
        mlflow.log_metric( accuracy , accuracy)
        mlflow.log_metric( f1 , evaluator.evaluate(predictions,
    {evaluator.metricName:  f1 }))

        # Save model artifact
        mlflow.spark.log_model(model, model_name)

        print(f     {model_name} - Accuracy: {accuracy:.4f} )

# Train multiple models
models = [
    ( RandomForest , RandomForestClassifier(labelCol=
   label_cooler_indexed , numTrees=20)),
    ( DecisionTree , DecisionTreeClassifier(labelCol=
   label_cooler_indexed ))
]

for name, clf in models:
    train_and_log_model(spark, train_df, test_df, clf, name,
   feature_cols)
```

e, Feature Importance Extraction

```
def extract_feature_importance(model, feature_cols, model_name)
   :
    importances = model.featureImportances
    feature_data = [
```

```
        { feature : feature_cols[i],  importance : float(importances[i])}
        for i in range(len(importances))
    ]
    feature_data.sort(key=lambda x: x[ importance ], reverse=True)

    # Save as JSON artifact
    filename = f /tmp/{model_name}_feature_importance.json
    with open(filename,  w ) as f:
        json.dump(feature_data, f, indent=2)

    mlflow.log_artifact(filename)

    print(f        Top 5 Features for {model_name}: )
    for item in feature_data[:5]:
        print(f    {item['feature']}: {item['importance']:.4f})
```

Typical Output:

```
Top 5 Features for RandomForest:
   PS1_mean: 0.2341
   TS1_std: 0.1892
   PS6_mean: 0.1456
   EPS1_mean: 0.0987
   FS1_std: 0.0734
```

4.1.6 Kubernetes Deployment

a, Infrastructure Architecture

The system is deployed on a **Kind (Kubernetes in Docker)** cluster, simulating a multi-node production environment. To ensure high availability and resource isolation, the cluster is configured with specific node roles.

Node Labeling Strategy: Workloads are distributed across three dedicated worker nodes to prevent resource contention:

- `role=infra-master`: NameNode, ResourceManager, MongoDB.
- `role=hdfs-worker`: DataNode, NodeManager.
- `role=kafka-broker`: Kafka Broker, Schema Registry.

```
# Applying labels in deploy-all.sh
kubectl label worker-0 role=infra-master
kubectl label worker-1 role=hdfs-worker
```

```
kubectl label worker-2 role=kafka-broker
```

b, Component Orchestration

1. HDFS & YARN (Storage & Compute) StatefulSets are utilized for the HDFS NameNode and DataNode to maintain stable network identities (`hdfs-namenode-0`) and persistent storage bindings.

- **Custom Images:** A specialized `hadoop-python:3.8` image was built to include both the Hadoop ecosystem and the Python environment required for Spark jobs.
- **YARN Mode:** Spark jobs are submitted to the YARN cluster manager (`--master yarn`), allowing for dynamic resource allocation across the HDFS workers.

2. Kafka StatefulSet Kafka is deployed in the `kafka` namespace. It uses shell expansion in environment variables to dynamically set its advertised listener based on the Pod's internal DNS.

```
- name: KAFKA_ADVERTISED_LISTENERS
  value:  K8S_INTERNAL://$(KAFKA_POD_NAME).kafka-headless.$(
   KAFKA_POD_NAMESPACE).svc.cluster.local:9092
```

3. MongoDB Analytics Layer MongoDB serves as the "hot" analytics storage, holding aggregated results and detected anomalies. It is deployed as a single-replica Deployment with a Persistent Volume Claim (PVC) for durability.

c, Data Processing Pipeline

Spark Streaming Analytics The core processing is handled by `spark_processor.py`, which:

1. Consumes raw sensor data from multiple Kafka topics using `subscribePattern`.
2. Implements **Watermarking** (20s) and **Sliding Windows** (1m window, 10s slide).
3. Performs aggregations: `avg`, `max`, `stddev`, and `range`.
4. Writes results to two destinations:
   - **HDFS (Parquet):** Partitioned by `sensor` for efficient long-term storage.
   - **Kafka (`hydraulic-analytics`):** For real-time consumption by the dashboard.

d, Monitoring & Automated Dashboards

The monitoring stack (**Prometheus, Pushgateway, Grafana**) is automated via a sidecar pattern.

Dashboard Auto-Initialization: The Grafana Pod includes an init-container (or sidecar) that runs Python scripts to programmatically create data sources and dashboards:

```
# From k8s/hydraulic-setup/monitoring.yaml
python src/init_grafana_datasource.py
python src/grafana_prometheus_dashboard.py
python src/grafana_anomaly_dashboard.py
```

Key Performance Indicators (KPIs):

- **System Health Score:** `min(hydraulic_sensor_health_score)` - A calculated metric (0-100) reflecting the overall status of the hydraulic system.
- **Anomaly Tracking:** `sum by (type) (increase( hydraulic_anomaly_total[5m]))`.
- **Sensor Stability:** `hydraulic_sensor_stddev_1m` - Used to identify volatile pressure changes.

e, Automation Workflow

The entire deployment is encapsulated in `deploy-all.sh`, which orchestrates the following:

1. **Pre-flight checks:** Validates `kind`, `docker`, and `kubectl` environments.
2. **Image Building:** Builds custom Hadoop, Spark, and Application images.
3. **Provisioning:** Creates namespaces, StorageClasses, and PVs.
4. **Initialization:** Formats the HDFS NameNode and sets up directory permissions.
5. **Execution:** Deploys the full stack and sets up Port-Forwarding automatically.

```
# Deployment Command
./deploy-all.sh
```

4.1.7 End-to-End Validation

a, Verifying the Flow

1. Spark-to-Kafka Analytics:

```
kubectl exec -it kafka-0 -n kafka -- kafka-console-consumer \
```

```
  --bootstrap-server localhost:9092 --topic hydraulic-analytics
    --from-beginning
```

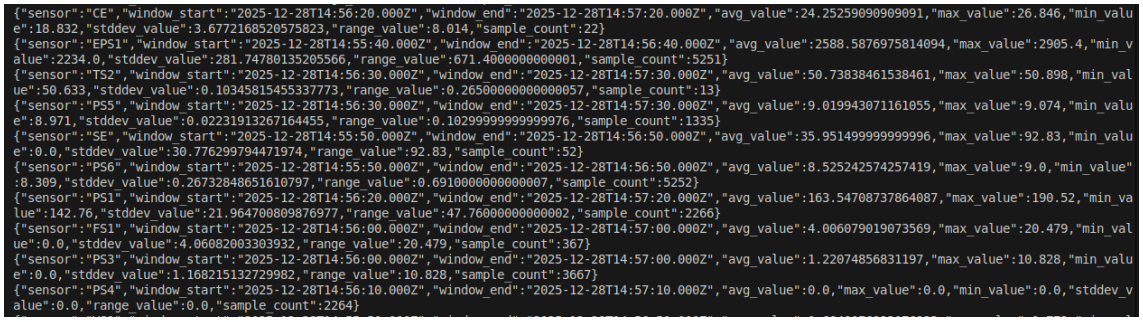

Figure 4.1: *Expected:* JSON objects containing `avg_value`, `stddev_value`, and window timestamps

2. HDFS Partitioning:

```
kubectl exec -it hdfs-namenode-0 -n hdfs -- hdfs dfs -ls /
   hydraulic/raw
```

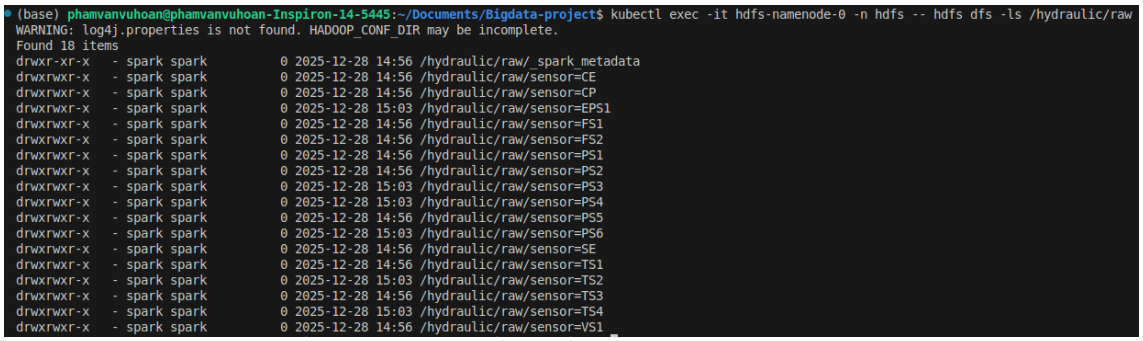

Figure 4.2: *Expected:* Directories for each sensor (`sensor=PS1`, `sensor=PS2`, etc.)

3. MLflow Model Tracking:

```
# Port forward to MLflow UI
kubectl port-forward -n monitoring svc/mlflow 5000:5000
```

Expected: Experiment logs for `Hydraulic-Anomaly-Model` with metrics and hyperparameters.

b, Success Criteria

The system is considered healthy when:

- All 20+ pods in namespaces `hdfs`, `kafka`, `mongodb`, and `monitoring` are `Running`.
- Grafana dashboards show non-zero throughput for `hydraulic_analytics_messages_tot`
- Spark UI (Port 4040) shows active streaming queries with flat processing latency.

4.1.8 Summary of Implementation

This implementation demonstrates a robust, production-grade Big Data architecture:

1. **Scalable Ingestion:** High-throughput Kafka backbone with Schema Registry for data consistency.
2. **Advanced Stream Processing:** Spark Streaming on YARN with stateful watermarking and sliding window aggregations.
3. **Hybrid Persistence:** Optimized storage with partitioned Parquet on HDFS for batch learning and MongoDB for low-latency analytics.
4. **Automated MLOps:** Experiment tracking and model versioning integrated via MLflow.
5. **Cloud-Native Orchestration:** Advanced K8s usage including StatefulSets, node affinity labeling, and automated monitoring provisioning.

CHAPTER 5. LESSONS LEARNED

This chapter documents the **technical lessons** learned during the migration from a local Docker Compose prototype to a production-grade Kubernetes deployment. These ”war stories” highlight the specific challenges of distributed systems and the solutions we engineered.

5.1 Kubernetes Networking & Infrastructure

5.1.1 Lesson 1.1: The Kafka ”Advertised Listeners” Paradox in K8s

Problem Description In Docker Compose, `localhost` mapping works easily. However, in Kubernetes, pods have dynamic IPs and distinct DNS names. Standard Kafka configuration failed because brokers couldn't advertise a reachable address to clients (Spark) and inter-broker communication simultaneously.

Solution: Dynamic Shell Expansion in StatefulSet Instead of hardcoding values, we utilized the Kubernetes Downward API to inject pod names and constructed the advertised listener dynamically at runtime:

```
env:
  - name: KAFKA_ADVERTISED_LISTENERS
    # RESULT: INTERNAL://kafka-0.kafka-headless.kafka.svc.cluster.local:9092
    value: INTERNAL://$(KAFKA_POD_NAME).kafka-headless.$(KAFKA_POD_NAMESPACE).svc.cluster.local:9092
```

Listing 5.1: Dynamic Kafka Listener Configuration

Takeaway In K8s stateful systems, identity is everything. Never rely on static IPs; standardise on FQDNs (Fully Qualified Domain Names).

5.1.2 Lesson 1.2: Spark Driver-Executor Connectivity (The ”NAT Problem”)

Problem Description When submitting Spark jobs in `client` mode on K8s, the Executors started but immediately failed to connect back to the Driver, causing the job to hang.

Root Cause The Driver (running in the `spark-submit-client` pod) was advertising its hostname as `localhost` or a non-routable name, effectively placing it behind a NAT relative to the Executors.

Solution: Explicitly Broadcast the Pod IP We forced the driver to advertise its specific POD IP using a sub-shell command in the submit arguments:

```
--conf spark.driver.host=$(hostname -i)
```

Listing 5.2: Spark Driver Host Configuration

Takeaway Spark's architecture assumes bi-directional reachability. In K8s, the "callback" channel from Executor to Driver must be explicitly configured.

5.1.3 Lesson 1.3: Configuration as Code (ConfigMaps)

Problem Description Every time logic in `spark_processor.py` was changed (e.g., adding a metric), the entire Docker image had to be rebuilt, taking 5+ minutes.

Solution: Decoupling Code from Image We mounted the Python source code as a **ConfigMap**.

- **Dev Flow**: Edit code locally → `kubectl apply -f configmap.yaml` → Restart Pod (0s build time).

Takeaway Optimizing the "Inner Loop" of development is crucial for velocity.

5.2 Stream Processing Challenges

5.2.1 Lesson 2.1: The "Late Data" Memory Trap

Problem Description Our Spark job's memory usage grew linearly until it crashed with OOM (Out Of Memory) after approximately 4 hours.

Root Cause Without `withWatermark`, Spark maintains the state of *every* window indefinitely, waiting for potentially infinite-late data.

Solution: Enforced Watermarking

```
.withWatermark( timestamp ,  10 seconds )
```

Listing 5.3: Spark Watermarking

This tells Spark: "Drop any state older than 10 seconds". Memory usage flatlined immediately after this change.

5.2.2 Lesson 2.2: Parallel Stream Orchestration

Problem Description We needed to verify labels (Metadata) alongside sensor data (Telemetry). Running two separate Spark jobs wasted resources.

Solution: Multi-Stream Topology We defined two distinct streaming queries (`readStream`) in the same `SparkSession` and used `spark.streams.awaitAnyTermination()` to orchestrate them concurrently.

5.2.3 Lesson 2.3: Processing High-Frequency Data

Problem Description 100Hz sensors generated 6000 events/minute. Micro-batch triggers at 1s caused huge scheduling overhead (scheduling time > execution time).

Solution: Tuned Trigger Interval We relaxed the trigger to `processingTime='5 seconds'`. This allowed Spark to batch more records (throughput optimization) at the cost of slightly higher latency, which was acceptable for this use case.

5.3 Data Consistency & Reliability

5.3.1 Lesson 3.1: Achieving Exactly-Once Semantics

Problem Description If the driver crashed and restarted, it would re-process the last batch, causing duplicate data in MongoDB.

Solution: Idempotent Sinks + Checkpointing

1. **Checkpointing**: Spark tracks offsets (e.g., `partition 0: offset 100`). On restart, it replays from offset 101.
2. **Idempotency**: In MongoDB, we used a composite key `_id = sensor_name + window_start`. If Spark overwrites the same document, the result is identical (no duplication).

5.3.2 Lesson 3.2: Handling Sensor Drift

Context In our simulation `producer.py`, a simple `sleep(0.01)` loop resulted in a 5% time drift over 1 hour due to execution overhead.

Solution: Absolute Timing Correction Instead of `sleep(interval)`, we calculated `sleep(target_time - current_time)`. This self-correcting loop kept the sensor strictly synchronized to the wall clock.

5.4 Operational Best Practices

5.4.1 Lesson 4.1: The "Init Container" Pattern

Problem Description Application pods (Producer) started before Kafka was ready, crashed, and went into `CrashLoopBackOff`.

Solution We implemented `initContainers` that simply wait for Kafka's port 9092 to be open before the main container starts.

Takeaway Dependency management is a runtime concern in K8s, not just a deployment concern.

5.4.2 Lesson 4.2: Resource Limits & Requests

Problem Description One rogue Spark executor consumed all CPU on the worker node, starving the Kafka broker.

Solution: Strict Quotas We applied Kubernetes Resource Limits:

```
resources:
  requests:
    memory:  1Gi
    cpu:  500m
  limits:
    memory:  2Gi
    cpu:  1000m
```

Listing 5.4: Kubernetes Resource Limits

Takeaway In a shared cluster, resource isolation is mandatory for stability.

5.4.3 Lesson 4.3: Monitoring the ”Un-monitorable”

Problem Description Our Python consumer scripts are ephemeral clients; Prometheus couldn’t scrape them directly.

Solution: Push Model We used the **Pushgateway**. Instead of Prometheus pulling metrics, our scripts *push* metrics to the gateway. This bridges the gap between batch/script processes and the scraping-based Prometheus world.

CHAPTER 6. CONCLUSION

6.1 Project Summary

In this Capstone Project, we have successfully designed and implemented a comprehensive ”Hydraulic System Anomaly Detection” platform based on the Kappa Architecture. By unifying real-time stream processing and historical batch analysis into a single code path using Apache Spark Structured Streaming, we effectively addressed the challenges of high-frequency sensor data (Velocity) and complex heterogeneous data types (Variety).

Key achievements of the project include:

- **Data Pipeline:** Established an end-to-end flow capable of ingesting data from 17 sensors at 100Hz with microsecond-level synchronization precision, ensuring data integrity from source to storage.
- **Polyglot Persistence:** Successfully implemented a hybrid storage strategy using HDFS for efficient raw data archival (Parquet format) and MongoDB for low-latency analytics serving.
- **Production-Grade Deployment:** Solved critical distributed system challenges, such as Kafka ”Advertised Listeners” and Spark Driver-Executor connectivity in Kubernetes, resulting in a scalable and resilient infrastructure.
- **Intelligent Analytics:** Integrated a machine learning pipeline with MLflow and Random Forest models to transition from reactive monitoring to predictive maintenance capabilities.

This project demonstrates that open-source Big Data technologies, when orchestrated correctly on Kubernetes, can provide a powerful and cost-effective solution for industrial health monitoring.

6.2 Acknowledgment

We would like to express our deepest gratitude to our supervisor, **Mr. Tran Viet Trung**, for his invaluable guidance, patience, and expert advice throughout this course. His deep knowledge of Big Data architectures and distributed systems provided the critical direction needed for us to navigate technical complexities and deliver this project.

REFERENCES

[1] P. E. Helwig Nikolai and A. Schtze, *Condition monitoring of hydraulic systems*, UCI Machine Learning Repository, DOI: https://doi.org/10.24432/C5CW21, 2015.