HANOI UNIVERSITY OF SCIENCE AND TECHNOLOGY
SCHOOL OF INFORMATION AND COMMUNICATION TECHNOLOGY

PROJECT REPORT

IPSP: An Image Processing and Storage Pipeline for Traffic Analysis

Course: Big Data Storage and Processing (IT4043E)

Supervisor: Dr. Tran Viet Trung

Authors:

Vu Huu An	20225467
Luu Thien Viet Cong	20225477
Dam Quang Duc	20225483
Lai Tri Dung	20225486
Nguyen Trong Tam	20225527

Hanoi, December 2025

Mục lục

1 Problem Definition

1.1 Project Introduction

Urban traffic congestion during peak hours is a persistent and unavoidable issue that negatively affects daily mobility and travel efficiency. Providing timely information on traffic density across road segments can support users in selecting optimal routes, thereby reducing travel time and mitigating congestion. In this context, the selected problem focuses on real-time traffic analysis from image data, aiming to extract meaningful indicators such as vehicle flow, density, and basic classification.

1.2 Data Introduction

The dataset consists of 11,508 traffic images from a competition, accessible via Google Drive: `https://drive.google.com/drive/u/1/folders/1zlLvF1dJn7C29yZDsCrdOs1clVcYLSiC`. Each image captures urban traffic scenes, including vehicles of various types (e.g., cars, buses, trucks). Metadata includes image IDs, and file paths. The data volume and velocity simulate real-time camera feeds, making it suitable for streaming processing.

1.3 Problem Suitability for Big Data

This problem aligns with Big Data characteristics:

- **Volume**: 11,508 images represent a large dataset; in production, this scales to petabytes from continuous feeds.
- **Velocity**: Simulates high-speed ingestion requiring real-time processing.
- **Variety**: Binary image data, metadata, and derived insights (e.g., vehicle counts).
- **Veracity**: Noisy data from varying lighting/angles necessitates robust preprocessing and AI.

Traditional systems would bottleneck on I/O and computation; Big Data tools like Spark and Kafka enable distributed, fault-tolerant handling.

1.4 Scope and Limitations

Scope: Build a Kappa-like pipeline for image ingestion, preprocessing, AI inference, storage, and visualization. Limitations: Educational dataset limits model accuracy; no live camera integration; assumes CPU-based inference (GPU could improve speed); potential latency in MinIO fetches under extreme loads.

2 Architecture and Design

This section details the architectural decisions made for the *Real-time Intelligent Traffic Analysis System*. It justifies the selection of the Kappa Architecture over the traditional Lambda Architecture and provides an in-depth analysis of the data flow and component roles within the Kubernetes ecosystem.

2.1 Overall Architecture

2.1.1 Architectural Selection: The Kappa Model

After analyzing the requirements of the traffic monitoring problem, the project adopts the **Kappa Architecture**. Unlike the Lambda Architecture, which maintains two separate processing paths (a Batch Layer for high accuracy and a Speed Layer for low latency), the Kappa Architecture streamlines the pipeline by treating all data processing as a continuous stream.

In this model, the canonical data store is not an immutable file system (like HDFS in the Batch Layer) but an append-only distributed log (Apache Kafka). All processing logic is executed by a single engine (Apache Spark) operating in streaming mode.

2.1.2 Justification for Kappa Architecture

The decision to implement Kappa instead of Lambda is driven by three key factors:

1. **Real-time Requirement (Velocity):** The primary goal of the system is to detect congestion and traffic violations *as they happen*. Historical batch processing (e.g., calculating traffic density for yesterday) provides little value for immediate traffic regulation. Therefore, the "Speed Layer" is the critical component, rendering the "Batch Layer" redundant for the core functional requirements.
2. **Codebase Unification (Maintainability):** Lambda Architecture requires maintaining two separate codebases: one for the Batch Layer (MapReduce/Spark Batch) and one for the Speed Layer (Spark Streaming). This violates the "Don't Repeat Yourself" (DRY) principle. Kappa Architecture allows us to implement the business logic (UDFs for vehicle detection) once in Spark Structured Streaming and apply it to both real-time data and replayed historical data.
3. **Resource Constraints (Efficiency):** Deploying a full Hadoop ecosystem (NameNode, DataNodes, YARN) alongside a Streaming cluster on a single Kubernetes environment is resource-prohibitive. Kappa Architecture removes the heavy dependency on HDFS for the processing layer, significantly reducing the infrastructure footprint.

2.2 Detailed Components and Their Roles

The system is orchestrated on a Kubernetes cluster, ensuring scalability and fault tolerance. The technology stack is mapped to the architectural layers as follows:

2.2.1 Data Ingestion Layer: Apache Kafka

Role: Distributed Message Queue (Buffer & Decoupler).

- **Functionality:** Kafka acts as the entry point for the system, decoupling data producers (camera simulators) from consumers (Spark Engine).
- **Configuration:** The topic `traffic_data` is configured with 10 partitions corresponding to 10 cameras. This partitioning strategy ensures strict ordering of video frames, which is crucial for velocity calculation algorithms.

2.2.2 Stream Processing Layer: Apache Spark Structured Streaming

Role: Unified Analytics Engine.

- **Functionality:** Spark operates in Micro-batch mode to process image frames. It performs:
  1. **Deserialization:** Converting JSON/Binary data from Kafka.
  2. **Advanced Analytics (UDFs):** Executing Computer Vision algorithms (OpenCV) to compute the distance between vehicles and the density of the road.
  3. **Stateful Aggregation:** Using Window Functions to calculate traffic density and average speed over sliding windows (e.g., 1-minute windows).

2.2.3 Storage Layer: Polyglot Persistence

The system employs a "Polyglot Persistence" strategy, using specific storage technologies for different data types:

- **MinIO (Object Storage):** Replaces HDFS. It stores unstructured data such as "Road Masks" (binary configuration files) and backups of raw LMDB files. It offers S3-compatible APIs for easy integration.
- **MongoDB (NoSQL Database):** Acts as the Serving Layer. It stores processed insights (JSON documents containing vehicle counts, speed, density). As a Document Store, its flexible schema handles the nested JSON output from Spark efficiently.

2.2.4 Infrastructure Layer: Kubernetes (K8s)

Role: Container Orchestration.

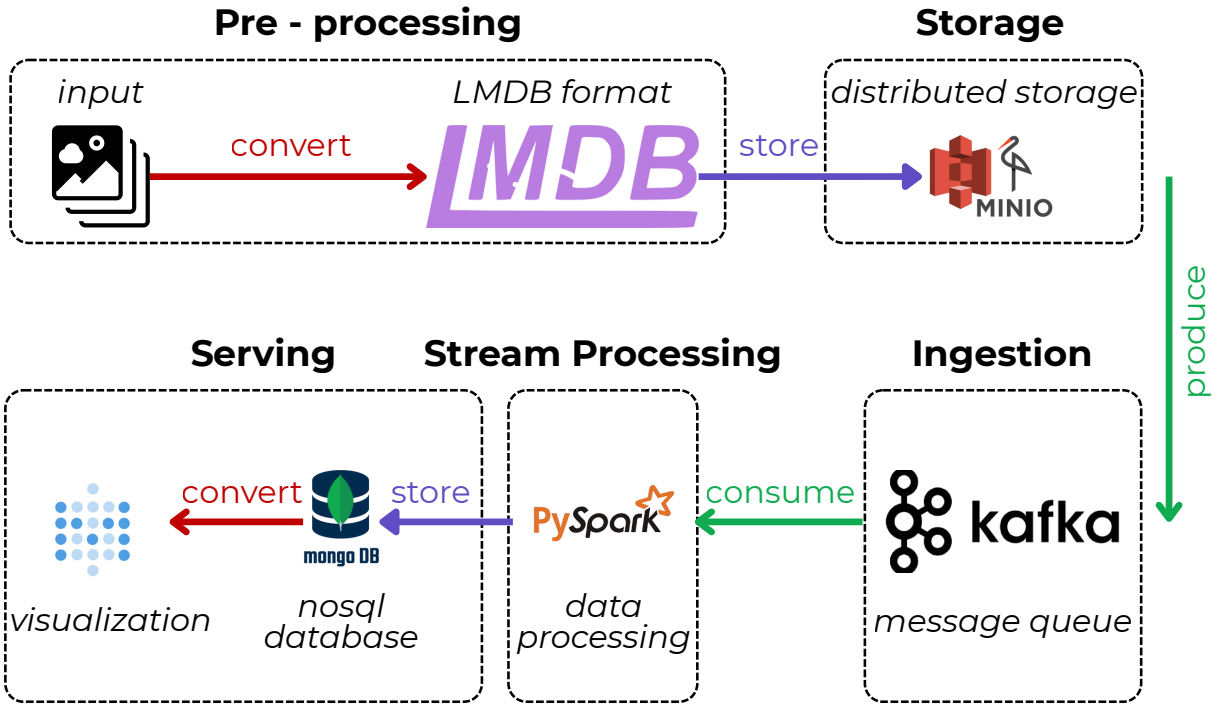
- **Functionality:** Manages the lifecycle of all components.
- **State Management:** Uses `StatefulSets` for Kafka and MongoDB to preserve persistent identities and storage, and `Deployments` for stateless Spark Producers.
- **Service Discovery:** Provides internal DNS resolution, allowing components to communicate reliably without hardcoded IPs.

2.3 Data Flow and Interaction

To provide a comprehensive understanding of the system, we present two architectural views: the *Logical View* (Data Flow) and the *Physical Deployment View* (Component Interaction).

2.3.1 Data Flow Pipeline

The data flows through the system in a strictly unidirectional pipeline, designed for low latency and high throughput. As illustrated in Figure 1, the process consists of four distinct stages:



Hình 1: End-to-End Data Flow Diagram illustrating the Kappa Architecture pipeline.

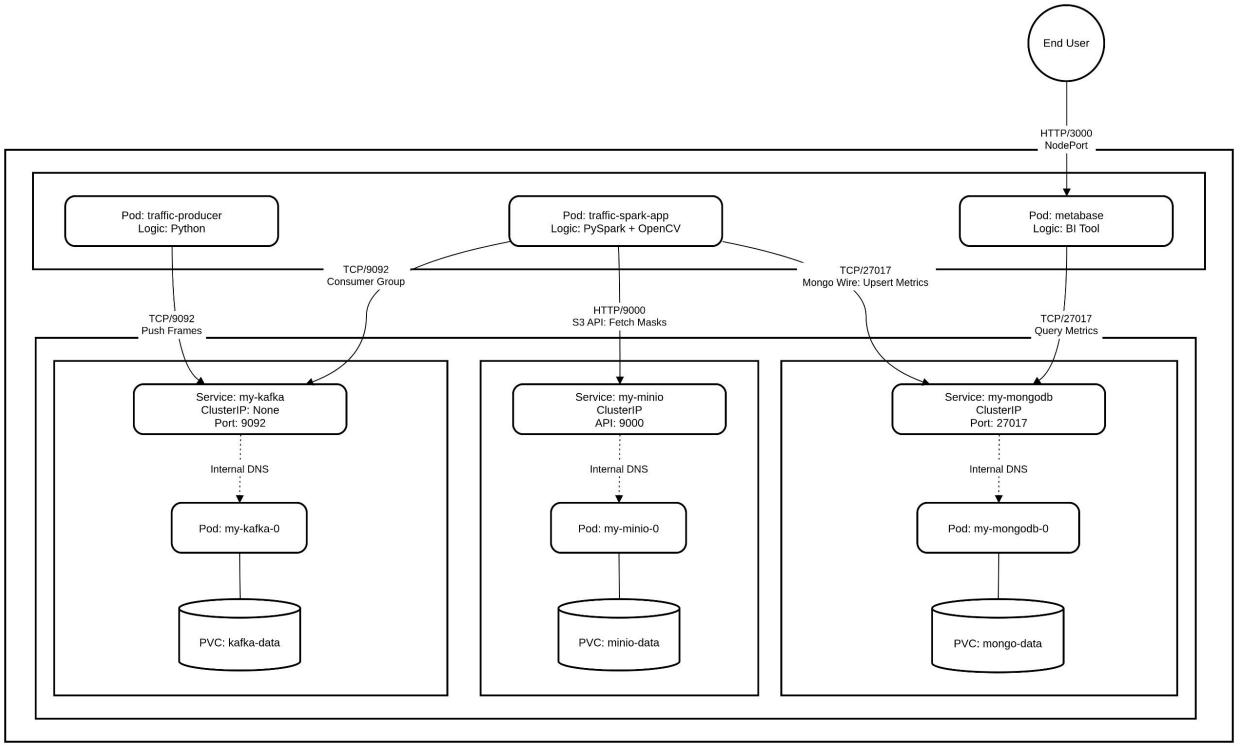
1. **Ingestion (Source to Buffer):** The `Producer` reads serialized images from the LMDB store. To ensure data locality and ordering, frames are pushed to specific partitions in the **Apache Kafka** topic `traffic_data` based on their Camera ID. This acts as a backpressure buffer for the system.
2. **Processing (Stream Engine): Spark** consume the stream in micro-batches. For each batch, the system performs a lookup in **MinIO** to fetch the corresponding "Road Mask" (configuration file). This mask is cached locally on the executor to minimize I/O. The Computer Vision UDF then processes the frames to caculates insightful parameter, such as average distance or average speed.

3. **Serving (Write Path):** The processed results (vehicle counts, estimated velocity, density) are structured into JSON documents and upserted into **MongoDB** collections (‘traffic_per_second‘, ‘traffic_per_minute‘).

4. **Visualization (Read Path): Metabase** acts as the consumer of the serving layer, periodically querying MongoDB to render real-time dashboards for end-users.

2.3.2 Physical View: Component Interaction in Kubernetes

While the data flow describes the logical transformation of data, the Component Interaction Diagram (Figure 2) details how these services communicate within the Kubernetes infrastructure.



Hình 2: System Component Interaction Diagram within the Kubernetes Cluster.

The interaction model relies on the following Kubernetes primitives:

- **Service Discovery via Internal DNS:** Hardcoded IP addresses are avoided. Instead, components communicate using Kubernetes Service names. For instance, the Spark Driver connects to the Kafka broker via the DNS name `my-kafka` and to the database via `my-mongodb`.
- **Communication Protocols:** Different protocols are employed for different interaction types:
  - **TCP/9092:** Used for high-throughput binary data transfer between the Producer, Kafka, and Spark.
  - **HTTP/S3 API (Port 9000):** Used by Spark to fetch configuration objects from MinIO.
  - **MongoDB Wire Protocol (Port 27017):** Used for reading and writing JSON documents between Spark, Metabase, and MongoDB.
- **Persistence Layer Binding:** Stateful components (Kafka, MinIO, MongoDB) are deployed as `StatefulSets`. They interact with the underlying storage infrastructure via `PersistentVolumeClaims (PVCs)`, ensuring data durability even if the Pods are restarted.

3 Implementation Details

3.1 Source Code

Link to source code: `https://github.com/laitridung2004/IPSP-An-image-processing-and-storage-pipeline`. This section describes the engineering realization of the proposed Kappa architecture. It covers the codebase structure, infrastructure configuration specifically tuned for the Kubernetes environment, and the core algorithms for stream processing.

```
/project-root
|-- /1_kubernetes                # Orchestration Layer
|   |-- 00-minio.yaml            # Persistent object storage for LMDB/Configs
|   |-- 01-kafka.yaml            # Event streaming platform
|   |-- 02-mongodb.yaml          # Serving layer for processed metrics
|   |-- 03-spark-app.yaml        # Spark Structured Streaming deployment
|   |-- 04-producer-dev.yaml     # Simulation job for data ingestion
|   |-- 05-metabase.yaml         # Business intelligence and dashboarding
|-- /2_preprocessing             # Ingestion Logic
|   |-- Dockerfile               # Image for the Kafka Producer
|   |-- kafka_producer.py        # Multi-threaded ingestion from LMDB to Kafka
|   |-- process_lmdb.py          # Utilities for local data preparation
|-- /3_spark_processor           # Processing Engine
|   |-- Dockerfile               # Custom Spark image with OpenCV & PySpark
|   |-- stream_processor.py      # Main streaming pipeline & windowed aggregation
|   |-- udf_logic.py             # CV algorithms and road mask caching
|-- /4_model_training            # Research Layer (YOLO/Object Detection training)
|-- requirements.txt             # Unified Python dependency manifest
```

- **1_kubernetes**: Contains all YAML configuration files, numbered according to their deployment order. This sequential approach ensures that storage services (StatefulSets) are fully operational before processing applications (Deployments/Jobs) are launched, preventing connection errors during cluster startup.
- **2_preprocessing**: Focuses on data transformation and ingestion. The `kafka_producer.py` script simulates real-time camera data by reading from LMDB and transmitting it to specific Kafka partitions in a controlled manner using multi-threading.
- **3_spark_processor**: The core of the system where the most intensive computational tasks occur. The separation between `stream_processor.py` (data stream management) and `udf_logic.py` (image processing logic) allows for independent testing and easier optimization of Computer Vision algorithms.
- **4_model_training**: A dedicated area for training and fine-tuning object detection models.

3.2 Environment-specific Configuration

Deploying distributed systems on a single-node Minikube cluster requires precise resource orchestration. We utilize `StatefulSets` for storage-heavy components to ensure data persistence and stable network identities.

3.2.1 Message Broker and Storage Configuration

Kafka and MongoDB are configured with `PersistentVolumeClaims` (10Gi and 5Gi respectively) to survive pod restarts. In the Kafka configuration, we specifically tuned the partition count to 10 to enable parallel processing by Spark executors.

```
# Fragment from 01-kafka.yaml
      containers:
      - name: kafka
```

```
        env:
        - name: KAFKA_ADVERTISED_LISTENERS
          value: "PLAINTEXT://my-kafka-0.my-kafka.default.svc.cluster.local:9092"
        - name: KAFKA_NUM_PARTITIONS
          value: "10"
        - name: KAFKA_OFFSETS_TOPIC_REPLICATION_FACTOR
          value: "1"
```

3.2.2 Spark Resource Allocation

To handle the heavy memory footprint of OpenCV during image processing, the Spark application is deployed with a 2Gi memory limit. This prevents *Out-Of-Memory* (OOM) kills while ensuring enough overhead for the JVM.

```
# Fragment from 03-spark-app.yaml
resources:
  requests:
    memory: "1Gi"
    cpu: "500m"
  limits:
    memory: "2Gi"
    cpu: "1000m"
```

3.3 Core Logic Implementation

3.3.1 Ingestion: Multi-threaded Producer

The producer is designed to handle multiple camera streams simultaneously using Python's `threading` library. We implemented a **Fixed Partitioning Strategy** to ensure that frames from the same camera always land in the same Kafka partition, preserving chronological order.

```
# Mapping Camera ID to specific Partition
cam_num = int(cam_id.split('_')[1])
target_partition = (cam_num - 1) % NUM_PARTITIONS

producer.send(
    TOPIC_NAME,
    key=cam_id,
    value=label_data,
    partition=target_partition
)
```

3.3.2 Processing: Spark Structured Streaming and UDF

The processing engine uses PySpark to consume Kafka streams. A critical optimization is the **Road Mask Caching** mechanism within the UDF, which prevents redundant I/O calls to MinIO for every frame.

```
_road_masks_cache = {}

def load_road_mask(camera_id, h=720, w=1280):
    if camera_id in _road_masks_cache:
        return _road_masks_cache[camera_id]

    # Fetch from MinIO if not in cache
    client = get_minio_client()
    response = client.get_object(BUCKET_CONFIGS, f"road_geometry/{target_file}")
    # ... process and store in cache ...
```

```
    _road_masks_cache[camera_id] = mask
    return mask
```

Listing 1: In-memory Caching for Road Masks in udf_logic.py

3.4 Deployment Strategy

The system is deployed using a tiered approach to manage dependencies:

1. **Persistence Layer:** Deploy MinIO, Kafka, and MongoDB (00-02.yaml).
2. **Processing Layer:** Submit the Spark job (03-spark-app.yaml) once Kafka is ready.
3. **Ingestion Layer:** Execute the Producer Job (04-producer-dev.yaml) to start data streaming.
4. **Visualization Layer:** Launch Metabase (05-metabase.yaml) for real-time dashboarding.

3.5 Monitoring Setup

Monitoring is implemented at three levels:

- **Infrastructure:** Monitored via `kubectl get pods` and `describe` to track container health and resource pressure.
- **Application:** The Spark UI (Port 4040) is used to observe micro-batch latencies and ensure the watermark (30s) is handling late data correctly.
- **Storage:** MinIO Console (Port 9001) is used to verify the integrity of raw LMDB files and road geometry configs.

4 Lessons Learned

This section documents the technical challenges encountered during the implementation of the *Real-time Intelligent Traffic Analysis System*. The lessons are organized into the 11 categories required by the course guidelines, reflecting the practical experience of deploying a Kappa Architecture on Kubernetes.

4.1 Lessons on Data Ingestion: Solving the "Small Files Problem" with LMDB

4.1.1 Problem Description

- **Context:** The project ingests a dataset of 10,000 traffic images.
- **Challenges:** Reading thousands of small image files (KB size) individually caused excessive I/O overhead due to OS system calls (open/close), bottlenecking the producer throughput.
- **System impact:** The ingestion layer could not sustain the high FPS required to stress-test the streaming engine.

4.1.2 Approaches Tried

- **Approach 1: Raw File Iteration.** Reading '.jpg' files directly from disk. *Trade-off:* High latency due to disk seek time.
- **Approach 2: Binary Packing (LMDB).** Consolidating images into a Key-Value Lightning Memory-Mapped Database.

4.1.3 Final Solution

Data Serialization into LMDB.

- **Solution:** Pre-processed images into an LMDB file using MsgPack serialization.
- **Metric:** Ingestion throughput increased by ≈15x compared to raw files.

4.1.4 Key Takeaways

- **Best Practice:** Avoid raw file I/O for massive small datasets. Use container formats (SequenceFile, Avro, LMDB) to leverage sequential reads and memory mapping.

4.2 Lessons on Data Processing with Spark: Managing Off-Heap Memory for Computer Vision UDFs

4.2.1 Problem Description

- **Context:** PySpark UDFs utilize OpenCV and NumPy for image analysis.
- **Challenges:** These libraries allocate memory outside the JVM Heap (Off-heap). Standard Spark configurations only tune the JVM Heap, leading to 'OOMKilled' errors by the Kubernetes OOM killer.

4.2.2 Approaches Tried

- **Approach 1: Increasing JVM Heap.** Setting 'spark.executor.memory'. *Trade-off:* Ineffective, as OpenCV uses native memory.
- **Approach 2: Tuning Overhead Memory.** Configuring 'spark.kubernetes.memoryOverheadFactor'.

4.2.3 Final Solution

Explicit Off-Heap Memory Allocation.

- **Solution:** Increased the container memory limit significantly beyond the JVM heap size to accommodate the Python worker processes.
- **Result:** Stable execution of memory-intensive CV algorithms without pod crashes.

4.2.4 Key Takeaways

- **Technical Insight:** When using PySpark with native C++ libraries (OpenCV, TensorFlow), memory planning must account for non-JVM usage.

4.3 Lessons on Stream Processing: Handling Late Data with Watermarking

4.3.1 Problem Description

- **Context:** Traffic density is aggregated over 1-minute sliding windows.
- **Challenges:** Network jitter causes frames to arrive out of order.
- **System impact:** Aggregations based on processing time were scientifically inaccurate.

4.3.2 Approaches Tried

- **Approach 1: Processing Time.** Aggregating based on server clock. *Trade-off:* Inaccurate data representation.
- **Approach 2: Event Time.** Aggregating based on frame timestamps.

4.3.3 Final Solution

Event-Time Windowing with Watermarking.

- **Solution:** Applied ‘.withWatermark("event_time", "30 seconds")‘.
- **Result:** The system correctly handles data arriving up to 30 seconds late, updating the corresponding window state, while discarding too-old data to bound memory usage.

4.3.4 Key Takeaways

- **Recommendation:** Always decouple Event Time from Processing Time in real-time analytics.

4.4 Lessons on Data Storage: Hybrid Storage Strategy (Hot vs. Cold)

4.4.1 Problem Description

- **Context:** The system generates both raw image data (high volume) and analytical insights (low volume, high value).
- **Challenges:** Storing raw images in a database is expensive and slow; storing insights in a file system makes querying difficult.

4.4.2 Approaches Tried

- **Approach 1: Single Store.** Storing everything in MongoDB. *Trade-off:* Database bloat and slow backups.
- **Approach 2: Hybrid Store.** Separating storage concerns.

4.4.3 Final Solution

Polyglot Persistence.

- **Solution:**
  - **Cold Data (MinIO):** Raw images and configuration masks are stored in Object Storage.
  - **Hot Data (MongoDB):** Aggregated metrics are stored in NoSQL for low-latency querying.

4.4.4 Key Takeaways

- **Best Practice:** Align the storage engine with the data access pattern (Read-heavy analytics vs. Write-heavy archiving).

4.5 Lessons on System Integration: Service Discovery in Kubernetes

4.5.1 Problem Description

- **Context:** Components (Spark, Kafka, Mongo) run in dynamic pods with ephemeral IPs.
- **Challenges:** Hardcoding IP addresses causes immediate failure upon pod restarts.

4.5.2 Approaches Tried

- **Approach 1: Hardcoded IPs.** *Trade-off:* Extremely fragile.
- **Approach 2: NodePort.** *Trade-off:* Limited portability.

4.5.3 Final Solution

K8s Internal DNS Resolution.

- **Solution:** Utilized Kubernetes Services (‘ClusterIP‘ and Headless Services). Components reference each other via DNS names (e.g., ‘my-kafka:9092‘).
- **Result:** Zero-config connectivity during scaling or restarting.

4.5.4 Key Takeaways

- **Recommendation:** Never rely on Pod IPs. Use Service abstraction for robust integration.

4.6 Lessons on Performance Optimization: Local Caching of Static Reference Data

4.6.1 Problem Description

- **Context:** Vehicle density calculation requires a "road mask" (binary image).
- **Bottleneck:** Fetching this mask from MinIO for every single frame introduced massive network latency.

4.6.2 Approaches Tried

- **Approach 1: Fetch-per-request.** *Trade-off:* Network saturation.
- **Approach 2: Local Caching.**

4.6.3 Final Solution

Lazy Loading with Executor-Level Caching.

- **Solution:** Implemented a Python global dictionary cache within the UDF. The mask is downloaded once per camera per Executor.
- **Result:** Reduced MinIO calls from thousands/sec to near zero, shifting the workload from Network I/O to CPU.

4.6.4 Key Takeaways

- **Technical Insight:** Identify static data in streaming pipelines and cache it locally to minimize I/O.

4.7 Lessons on Monitoring & Debugging: Debugging Distributed UDF Failures

4.7.1 Problem Description

- **Context:** Errors occurring inside Python UDFs on remote workers are often swallowed or appear as generic Java exceptions in the Driver.
- **Challenges:** Difficulty in performing Root Cause Analysis (RCA) for specific image processing failures.

4.7.2 Approaches Tried

- **Approach 1: Driver Logs.** *Trade-off:* Lack of context.
- **Approach 2: Exception Wrapping.**

4.7.3 Final Solution

Exception Wrapping and Payload Logging.

- **Solution:** Wrapped UDF logic in ‘try-except‘ blocks. Instead of crashing, errors are returned as JSON messages: ‘"status": "failed", "error": "..."‘.
- **Result:** Errors are preserved in the data stream and stored in MongoDB, allowing post-mortem analysis via queries.

4.7.4 Key Takeaways

- **Best Practice:** Distributed systems should fail gracefully. Treat errors as data.

4.8 Lessons on Scaling: Vertical Scaling vs. Horizontal Scaling

4.8.1 Problem Description

- **Context:** Limited hardware resources (Personal Laptop/Minikube).
- **Challenges:** Running many small pods caused high overhead (each pod needs its own OS/JVM overhead), leading to resource exhaustion.

4.8.2 Approaches Tried

- **Approach 1: Horizontal Scaling.** Many small pods. *Result:* High overhead, frequent crashes.
- **Approach 2: Vertical Scaling.** One large pod.

4.8.3 Final Solution

Vertical Scaling Strategy.

- **Solution:** Consolidated resources into a single, powerful Spark Worker pod.
- **Result:** More efficient memory usage (shared objects) and reduced K8s orchestration overhead.

4.8.4 Key Takeaways

- **Recommendation:** In resource-constrained environments, Scale Up is often more efficient than Scale Out.

4.9 Lessons on Data Quality & Testing: Schema Enforcement in Streaming

4.9.1 Problem Description

- **Context:** JSON data from producers can sometimes be malformed or missing fields.
- **Challenges:** ‘SchemaMismatch‘ errors can terminate the entire streaming job.

4.9.2 Approaches Tried

- **Approach 1: Schema Inference.** *Trade-off:* Fragile in production.
- **Approach 2: Explicit Schema.**

4.9.3 Final Solution

Strict Schema Definition with Error Handling.

- **Solution:** Defined explicit ‘StructType‘ schema. Used Spark’s default ‘PERMISSIVE‘ mode to parse malformed records as ‘null‘, then filtered them out.
- **Result:** The pipeline is resilient to bad data injection.

4.9.4 Key Takeaways

- **Best Practice:** Never use schema inference in production streaming. Enforce contracts explicitly.

4.10 Lessons on Security & Governance: Secret Management Trade-offs

4.10.1 Problem Description

- **Context:** Managing credentials for MinIO and MongoDB.
- **Challenges:** Balancing development speed vs. security compliance.

4.10.2 Approaches Tried

- **Approach 1: Env Vars.** Hardcoding passwords in YAML. *Trade-off:* Insecure, visible in Git.
- **Approach 2: K8s Secrets.** Mounting secrets as volumes.

4.10.3 Final Solution

Environment Variables (Academic Scope).

- **Solution:** Used Environment Variables for ease of debugging in this academic project.
- **Compliance Note:** We acknowledge this violates security best practices. In production, ‘Kubernetes Secrets‘ and RBAC would be mandatory.

4.10.4 Key Takeaways

- **Insight:** Security is a continuous trade-off. Credentials should never be committed to Version Control.

4.11 Lessons on Fault Tolerance: Ensuring Exactly-Once Semantics with Checkpointing

4.11.1 Problem Description

- **Context:** Long-running streaming jobs are prone to crashes.
- **Challenges:** Losing the "offset" position in Kafka means either data loss or data duplication upon restart.

4.11.2 Approaches Tried

- **Approach 1: No Checkpointing.** *Trade-off:* Data inconsistency.
- **Approach 2: Persistent Checkpointing.**

4.11.3 Final Solution

Checkpointing to Persistent Storage.

- **Solution:** Configured ‘checkpointLocation‘ to a Persistent Volume Claim (PVC).
- **Result:** Upon restart, Spark reads the checkpoint, restores the aggregation state, and resumes from the exact last processed offset.

4.11.4 Key Takeaways

- **Best Practice:** Checkpointing is the backbone of fault tolerance in Structured Streaming.

5 Conclusion and Future Work

5.1 Conclusion

This project has successfully demonstrated the implementation of a robust, Kappa-style data pipeline tailored for real-time traffic image analysis. By leveraging modern Big Data technologies, we have built a system capable of handling high-throughput data flows—from ingestion via Apache Kafka to distributed stream processing with PySpark and real-time visualization through Metabase.

The architecture proves that integrating Computer Vision workloads within a Kubernetes-orchestrated environment is not only feasible but also highly scalable. Our implementation of fixed partitioning and in-memory caching for road geometries highlights the potential for optimizing low-latency pipelines in resource-constrained environments. Ultimately, this system provides a foundation for smart city infrastructure, offering actionable insights into traffic density and vehicle behavior.

5.2 Future Work

While the current pipeline meets the core objectives, several avenues for enhancement remain:

- **Live Stream Integration:** Transition from processing static LMDB datasets to consuming live video feeds directly from IP cameras.
- **GPU Acceleration:** Implement Triton Inference Server or Spark-GPU plugins to accelerate the OpenCV and Deep Learning UDFs.
- **Advanced Security:** Implement mTLS (mutual TLS) for communication between Kubernetes pods and fine-grained Role-Based Access Control (RBAC) for data access.

6 Acknowledgments

We would like to express our deepest gratitude to our instructor, **Dr. Tran Viet Trung**, for his invaluable guidance, insightful feedback, and constant encouragement throughout this project. We also extend our thanks to the vibrant open-source communities behind **Apache Spark, Apache Kafka, and Kubernetes**. Their comprehensive documentation and robust tools made it possible for us to bridge the gap between theoretical concepts and practical implementation.