**ĐẠI HỌC BÁCH KHOA HÀ NỘI**

TRƯỜNG CÔNG NGHỆ THÔNG TIN VÀ TRUYỀN THÔNG

--------------»«--------------

A red and yellow logo

AI-generated content may be incorrect.

BÁO CÁO PROJECT I

**Hệ thống thu thập ,lưu trữ , phân tích mật độ giao thông**

**thời gian thực triển khai trên Kubernetes**

**Chu Quang Hưng – 20225325**

[*Hung.cq225325@sis.hust.edu.vn*](mailto:Hung.cq225325@sis.hust.edu.vn)

Chuyên ngành Kỹ thuật máy tính

***Giảng viên hướng dẫn: Thầy Hoàng Việt Dũng***

***Bộ môn:*** *IT-3150 : Project I*

***Mã lớp:*** 755566

**Hà Nội, 11/2025**

# I. Tổng Quan Dự Án

## 1.1 Mục Tiêu

Xây dựng một hệ thống phân tích và trực quan hóa mật độ giao thông tại các nút giao thông ở TP.HCM trong thời gian thực.

Lưu trữ dữ liệu lịch sử về mật độ giao thông để phục vụ việc phân tích xu hướng và dự đoán .

Cung cấp một dashboard hiển thị bản đồ nhiệt (heatmap) và các biểu đồ về tình trạng giao thông.

Xây dựng một data pipeline sử dụng Kubernetes làm nền tảng.

## 1.2 Phạm Vi

**Thu thập dữ liệu:** Chỉ thu thập dữ liệu hình ảnh từ các camera tại các nút giao thông được cung cấp công khai qua cổng thông tin giaothong.hochiminhcity.gov.vn.

**Phân tích:** Áp dụng mô hình Học máy (Object Detection) để đếm số lượng các loại phương tiện cơ bản (xe máy, ô tô, xe buýt/tải).

**Tính toán mật độ:** Từ số lượng phương tiện, tính toán ra chỉ số mật độ (ví dụ: Thấp, Trung bình, Cao, hoặc %).

**Hiển thị:** Cung cấp 01 dashboard Grafana hiển thị trạng thái thời gian thực (real-time) và dữ liệu lịch sử theo thời gian

**Hạ tầng:** Toàn bộ các dịch vụ được triển khai và điều phối trên cụm Kubernetes (K8s).

# II. Kiến trúc & Luồng Dữ liệu Chi tiết trên K8s

A diagram of a computer network

AI-generated content may be incorrect.

## ****Thu thập và đếm phương tiện****

Giai đoạn đầu tiên của hệ thống tập trung vào việc thu thập dữ liệu và xác định số lượng phương tiện. Một microservice được triển khai đồng thời thực hiện hai nhiệm vụ chính: thu thập ảnh tĩnh từ cổng thông tin **giaothong.hochiminhcity.gov.vn** và sử dụng mô hình học máy **YOLOv8** để phát hiện, đếm phương tiện. Kết quả được tổng hợp thành một thông điệp JSON chứa dữ liệu đếm xe thô, sau đó gửi vào hệ thống hàng đợi

## ****Hàng đợi dữ liệu****

Dữ liệu thô sau khi được thu thập được chuyển vào hệ thống **Apache Kafka**, đóng vai trò là bộ đệm trung gian. Kafka giúp đảm bảo tính bền vững và chống chịu lỗi cho hệ thống, ngăn ngừa mất mát dữ liệu khi các dịch vụ xử lý phía hạ tầng gặp sự cố hoặc cần bảo trì.

## ****Xử lý dữ liệu****

Giai đoạn trung tâm của hệ thống là xử lý dữ liệu, được thực hiện bởi **Apache Spark Streaming**. Ứng dụng Spark, triển khai dưới dạng job trên Kubernetes thông qua **Spark Operator**, tiêu thụ dữ liệu từ Kafka. Tại đây, dữ liệu được làm giàu bằng cách thêm các ngữ cảnh quan trọng từ các API bên ngoài .

## ****Lưu trữ dữ liệu****

Dữ liệu sau xử lý được lưu trữ theo kiến trúc **hai luồng** nhằm phục vụ các mục đích khác nhau:

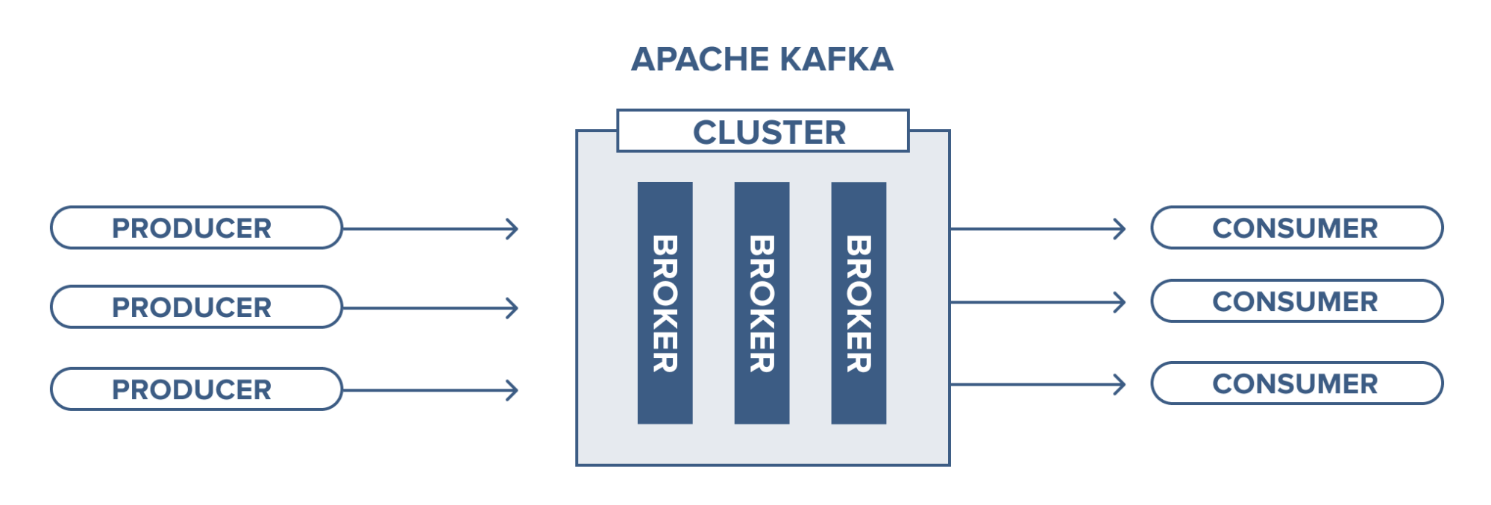
* **Lưu trữ Thời gian Thực:** Sử dụng TimescaleDB để lưu trữ dữ liệu đã xử lý, phục vụ các yêu cầu truy xuất nhanh cho dashboard.
* **Lưu trữ Lịch sử:** Dữ liệu đồng thời được ghi vào MinIO dưới dạng file Parquet, hình thành một Data Lake phục vụ lưu trữ dài hạn và các tác vụ phân tích, huấn luyện mô hình dự đoán trong tương lai.

## Trực quan hóa dữ liệu

Giai đoạn cuối cùng là trực quan hóa, được thực hiện bởi **Grafana**, triển khai dưới dạng Deployment và Service trên Kubernetes. Grafana kết nối trực tiếp với TimescaleDB để truy vấn dữ liệu , cung cấp cho người dùng cuối các dashboard tương tác, bao gồm biểu đồ và bản đồ nhiệt về tình trạng giao thông.

# III. Triển khai hệ thống

## 1. Triển Khai Kafka Trên Kubernetes



Kafka được triển khai trên Kubernetes bằng một StatefulSet một replica, sử dụng Kafka 3.8 với kiến trúc **KRaft** (broker kiêm controller). Việc triển khai này cho phép tạo một điểm thu thập dữ liệu thời gian thực cho toàn bộ pipeline phân tích giao thông.

### 1.1. Thành phần StatefulSet Kafka

Cấu hình StatefulSet định nghĩa một Kafka node duy nhất (replicas: 1) trong namespace hugedata. Container sử dụng image apache/kafka:3.8.0 và mở ba cổng:

* **9092** – INTERNAL listener dùng cho giao tiếp trong cluster.
* **9093** – CONTROLLER listener phục vụ KRaft controller.
* **9094** – EXTERNAL listener để truy cập khi cần port-forward.

Các biến môi trường quan trọng được khai báo trực tiếp trong manifest:

* KAFKA\_NODE\_ID=1 xác định ID của node.
* KAFKA\_PROCESS\_ROLES=broker,controller cho phép node vừa làm broker vừa làm controller theo chuẩn Kafka 3.x.
* KAFKA\_LISTENERS cấu hình ba listener nội bộ container: INTERNAL, EXTERNAL và CONTROLLER.
* KAFKA\_ADVERTISED\_LISTENERS định nghĩa endpoint mà client sẽ truy cập:
  + INTERNAL qua DNS Kubernetes:  
    kafka.hugedata.svc.cluster.local:9092
  + EXTERNAL qua 127.0.0.1:9094 để phục vụ debug hoặc port-forward.
* KAFKA\_INTER\_BROKER\_LISTENER\_NAME=INTERNAL đảm bảo trao đổi nội bộ giữa broker sử dụng listener INTERNAL.
* KAFKA\_CONTROLLER\_QUORUM\_VOTERS=1@kafka-0.kafka.hugedata.svc.cluster.local:9093 khai báo địa chỉ controller duy nhất.
* CLUSTER\_ID đặt mã định danh cụm Kafka theo chuẩn KRaft.

Những tham số này cho phép Kafka chạy độc lập mà không cần Zookeeper.

### 1.2. Dịch vụ Truy cập Kafka

Một Service tên **kafka** được tạo để cung cấp endpoint tĩnh cho các ứng dụng nội bộ. Service ánh xạ hai port:

* Port 9092 dành cho INTERNAL listener.
* Port 9094 dành cho EXTERNAL listener.

Service sử dụng selector app: kafka và chạy trong cùng namespace hugedata, giúp các ứng dụng trong cluster kết nối Kafka qua DNS: kafka.hugedata.svc.cluster.local:9092

### 1.3. Tích hợp Producer

Producer được triển khai dưới dạng một Deployment độc lập trong namespace hugedata, sử dụng image kafka-producer:dev. Container nhận biến môi trường KAFKA\_BOOTSTRAP\_SERVERS, trỏ đến Service nội bộ của Kafka:

kafka.hugedata.svc.cluster.local:9092

Ứng dụng producer được phát triển bằng Python và thực hiện toàn bộ quy trình thu thập – xử lý – gửi dữ liệu theo các bước sau:

#### 1.3.1. Khởi tạo Kafka Producer

Ứng dụng đọc biến môi trường KAFKA\_BOOTSTRAP\_SERVERS, tách thành danh sách server, sau đó tạo KafkaProducer với cấu hình:

* Gửi dữ liệu dưới dạng JSON (value\_serializer).
* acks='all' để đảm bảo broker xác nhận ghi dữ liệu.
* linger\_ms=100 nhằm gom nhiều message nhỏ để giảm overhead.

Producer mặc định gửi dữ liệu vào Kafka topic **traffic**.

#### 1.3.2. Tải danh sách camera vào bộ nhớ

Danh sách camera được đọc từ tệp cameras.json và lưu sẵn vào bộ nhớ để tránh phải load lại mỗi vòng chạy. Mỗi mục camera bao gồm:

* cam\_id
* display\_name
* Tọa độ GPS được trích xuất từ chuỗi JSON trong trường Location.

Ứng dụng sử dụng thư viện **orjson** để tăng tốc độ tải và parse cấu trúc.

#### 1.3.3. Tải ảnh từ camera

Ứng dụng sử dụng thư viện requests với cơ chế retry, timeout và HTTPAdapter để lấy hình ảnh JPEG từ API:

https://giaothong.hochiminhcity.gov.vn/render/ImageHandler.ashx?id=<cam\_id>

Ảnh được resize về kích thước 320×180 nhằm giảm tài nguyên tính toán khi xử lý YOLO.

#### 1.3.4. Phân tích phương tiện bằng YOLO

Mô hình YOLOv8n được tải trực tiếp trong container.  
Quy trình phân tích:

* Chạy dự đoán trên CPU với mức confidence 0.1.
* Trích chỉ số lớp từ results.boxes.cls.
* Chuyển đổi chỉ số lớp về tên phương tiện theo model.names.
* Giữ lại các phương tiện thuộc các nhóm: car, motorcycle, bus, truck.
* Tổng hợp số lượng bằng collections.Counter.

#### 1.3.5. Đóng gói và gửi message vào Kafka

Mỗi khung hình được xử lý tạo ra một JSON message dạng:

{

"camera": <tên camera>,

"camera\_id": <id>,

"latitude": <lat>,

"longitude": <lon>,

"timestamp": <UTC ISO time>,

"counts": { "car": x, "motorcycle": y, ... }

}

Message được gửi vào topic **traffic** thông qua KafkaProducer.

Sau mỗi batch camera, producer gọi producer.flush() để đảm bảo toàn bộ dữ liệu được đẩy vào broker.

#### 1.3.6. Cơ chế xử lý theo batch

Ứng dụng chạy vòng lặp vô hạn. Trong mỗi vòng:

* Camera được chia theo batch (20 camera/lượt).
* Mỗi batch sử dụng ThreadPoolExecutor với tối đa 4 worker để tải và xử lý song song.
* Sau mỗi batch, chương trình nghỉ 0.5 giây trước khi chuyển sang batch tiếp theo.
* Thời gian toàn bộ vòng xử lý được in ra để theo dõi hiệu suất.

## 2. Triển khai Spark Streaming trong Kubernetes

Hệ thống phân tích dữ liệu giao thông thời gian thực được mở rộng bằng việc triển khai Spark Structured Streaming trên Kubernetes thông qua Spark Operator. Ứng dụng Spark chạy dưới dạng một SparkApplication ở chế độ cluster, trong đó Spark Operator tự động tạo ra driver pod và executor pod dựa trên container image spark-application:dev. Cả hai pod đều được cấu hình chung các biến môi trường như KAFKA\_BOOTSTRAP\_SERVERS và KAFKA\_TOPIC, cho phép kết

nối trực tiếp đến Kafka nội bộ thông qua dịch vụ kafka.hugedata.svc.cluster.local:9092

A computer screen shot of a computer

AI-generated content may be incorrect.

### 2.1. Cấu hình Spark và môi trường Hadoop

Ứng dụng khởi tạo SparkSession và cấu hình hệ thống Hadoop nhằm cho phép driver và executor truy cập MinIO bằng giao thức S3A. Các tham số như fs.s3a.access.key, fs.s3a.secret.key, fs.s3a.endpoint và fs.s3a.path.style.access được thiết lập phù hợp với môi trường MinIO, đồng thời vô hiệu hóa SSL để đảm bảo kết nối nội bộ. Cách cấu hình này cho phép Spark ghi dữ liệu theo dạng đối tượng trên MinIO giống như Amazon S3, tạo điều kiện thuận lợi cho việc lưu trữ dữ liệu dài hạn.

### 2.2. Định nghĩa schema dữ liệu và luồng vào Kafka

Dữ liệu được định nghĩa dưới dạng cấu trúc JSON phù hợp với định dạng mà Kafka Producer tạo ra. Schema bao gồm các trường camera, camera\_id, toạ độ GPS, timestamp và một Map mô tả số lượng phương tiện theo từng loại. Khi đọc dữ liệu từ Kafka, Spark Structured Streaming sử dụng nguồn “kafka” với các tuỳ chọn startingOffsets = earliest để tải toàn bộ lịch sử phục vụ kiểm thử và failOnDataLoss = false nhằm đảm bảo job không bị dừng đột ngột nếu Kafka xoá các phân đoạn log cũ. Dữ liệu nhị phân từ Kafka được chuyển sang chuỗi JSON và ánh xạ vào DataFrame dựa trên schema đã định nghĩa.

### 2.3. Xử lý luồng và ghi dữ liệu ra MinIO

Luồng dữ liệu sau khi được parse được chuyển sang sink đầu ra dạng Parquet lưu trữ trong MinIO. Ứng dụng sử dụng outputMode “append” để ghi liên tục, kết hợp checkpoint nhằm hỗ trợ quản lý trạng thái và khôi phục nếu job bị khởi động lại. Đường dẫn lưu trữ và checkpoint được đặt trong bucket traffic-data, tạo thành một pipeline ổn định và có khả năng mở rộng. Khi driver không thể khởi tạo kết nối S3A do MinIO chưa sẵn sàng, ứng dụng tự động chuyển sang chế độ ghi tệp vào filesystem nội bộ trong container để duy trì hoạt động, đảm bảo dữ liệu không bị mất trong quá trình thử nghiệm.

### 2.4. Triển khai Spark Streaming lên Kubernetes bằng Spark Operator

Việc triển khai ứng dụng Spark Structured Streaming được thực hiện thông qua Spark Operator, một thành phần tự động hoá toàn bộ vòng đời của ứng dụng Spark trên Kubernetes. Cấu hình SparkApplication định nghĩa mode hoạt động là cluster, sử dụng image spark-application:dev chứa mã nguồn main.py cùng toàn bộ thư viện phụ trợ như spark-sql-kafka và hadoop-aws. Spark Operator tiếp nhận manifest này, khởi tạo driver pod và executor pod dưới namespace hugedata, đồng thời giám sát liên tục vòng đời của ứng dụng để tự động restart nếu có lỗi.

### 2.5. Cấu hình SparkApplication và cơ chế tạo Driver–Executor

SparkApplication xác định driver và executor sử dụng cùng biến môi trường KAFKA\_BOOTSTRAP\_SERVERS và KAFKA\_TOPIC, giúp mỗi thành phần kết nối trực tiếp vào Kafka Service nội bộ. Driver được cấp 1 vCPU và 1 GB RAM, trong khi executor cũng được cấp tài nguyên tương tự với số lượng phiên bản là một, phù hợp với môi trường thử nghiệm. Ứng dụng tham chiếu tệp mainApplicationFile qua đường dẫn local:///app/main.py, cho phép Spark chạy trực tiếp mã nguồn có trong image. Trường spark.jars chỉ định bộ thư viện được nhúng vào container, bảo đảm driver và executor có đầy đủ tích hợp Kafka và S3A khi xử lý luồng.

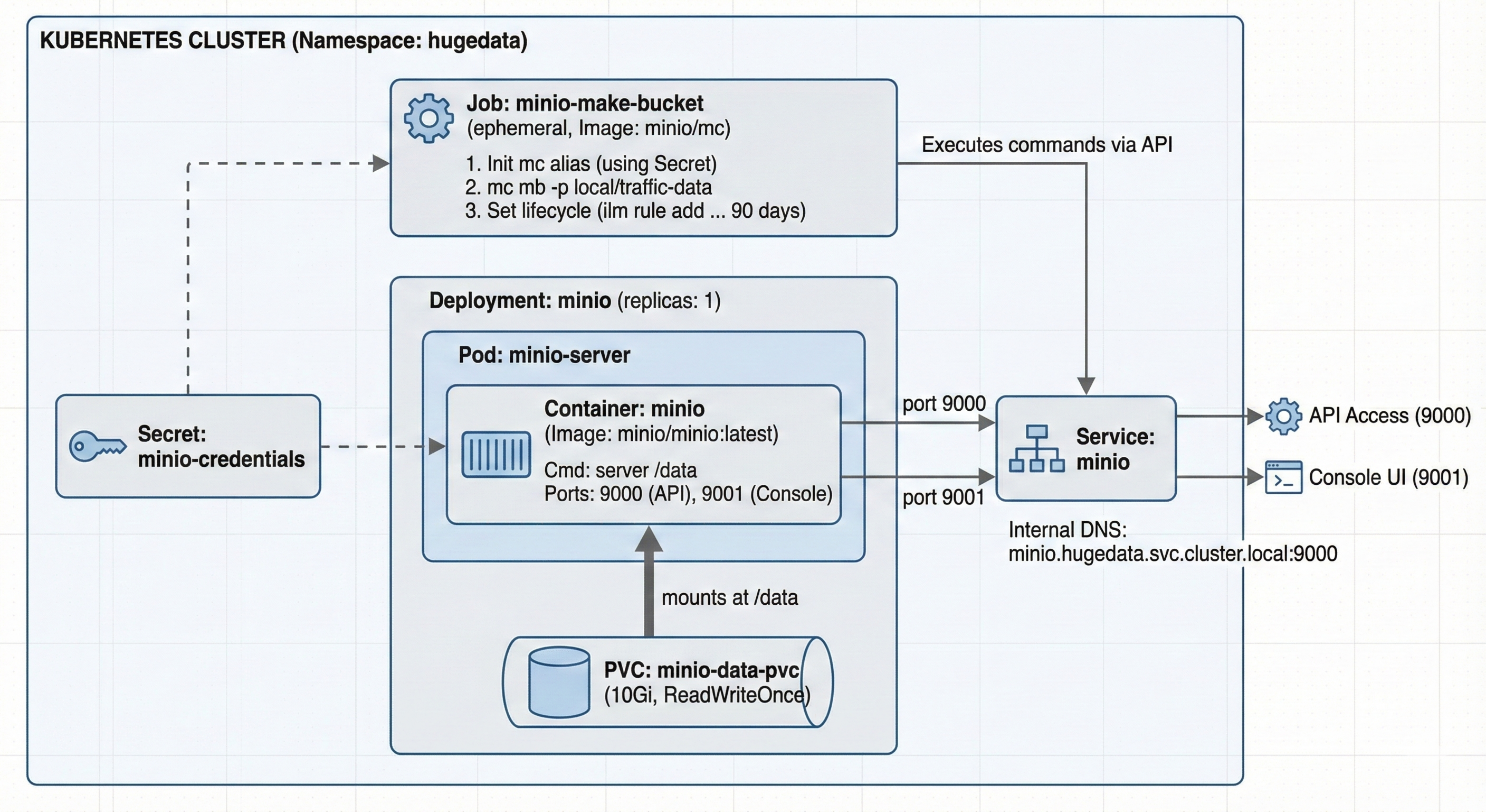
### 2.6. Thiết lập RBAC cho Spark Operator và Spark Application

Kubernetes yêu cầu cấu hình quyền truy cập rõ ràng để Spark Operator và ứng dụng Spark có thể tạo pod, service hoặc configmap trong quá trình hoạt động. Role spark-operator-role được thiết kế với quyền toàn diện trên các tài nguyên lõi như pods, services, secrets và sparkapplications, bảo đảm Spark Operator có khả năng triển khai, theo dõi và xoá ứng dụng khi cần. RoleBinding gán quyền này cho service account spark-operator-1-controller, là thành phần điều khiển chính của Spark Operator. Bên cạnh đó, driver và executor cần có quyền để tự tạo pod và PVC, do đó một role riêng tên spark-app-role được khai báo và gán cho service account default thông qua RoleBinding.

## 3. Triển khai hệ thống lưu trữ MinIO trên Kubernetes

Hệ thống sử dụng MinIO làm kho lưu trữ đối tượng (Object Storage) tương thích với giao thức

S3 của Amazon. Việc triển khai được thực hiện trong namespace hugedata thông qua các tài nguyên Kubernetes tiêu chuẩn bao gồm Deployment, Service, PVC và Job.



### 3.1. Kiến trúc Deployment và Quản lý tài nguyên

MinIO được triển khai dưới dạng một **Deployment** đơn lẻ (replicas: 1) để đảm bảo tính nhất quán của dữ liệu trong mô hình Standalone.

**Container Image:** Sử dụng minio/minio:latest với tham số khởi động server /data và kích hoạt giao diện quản trị Console tại cổng :9001.

**Cổng kết nối (Ports):** Container mở hai cổng:

* + 9000 (API): Dành cho các kết nối S3 từ Spark Application và các dịch vụ khác.
  + 9001 (Console): Dành cho giao diện quản trị web UI.

### 3.2. Tạo kho lưu trữ dữ liệu

Để đảm bảo dữ liệu không bị mất khi Pod MinIO bị khởi động lại hoặc di chuyển sang node khác, hệ thống sử dụng **PersistentVolumeClaim (PVC)** tên minio-data-pvc.

**Dung lượng:** Yêu cầu cấp phát 10Gi lưu trữ.

**Access Mode:** Thiết lập ReadWriteOnce, nghĩa là volume chỉ có thể được mount bởi một node tại một thời điểm, phù hợp với mô hình triển khai đơn lẻ của MinIO.

**Mounting:** PVC này được mount vào đường dẫn /data bên trong container, nơi MinIO thực hiện ghi các object.

### 3.3. Dịch vụ mạng (Service Exposure)

Tài nguyên **Service** minio được định nghĩa để cung cấp một địa chỉ IP và DNS ổn định trong nội bộ cluster.

Các ứng dụng khác trong namespace hugedata (như Spark Driver/Executor) kết nối tới MinIO thông qua DNS: minio.hugedata.svc.cluster.local:9000.Service mapping cả hai cổng 9000 (API) và 9001 (Console) từ container ra mạng nội bộ cluster.

### 3.4. Khởi tạo Bucket

Thay vì tạo bucket thủ công, hệ thống sử dụng một **Kubernetes Job** (minio-make-bucket) chạy image minio/mc (MinIO Client) để tự động hóa quy trình khởi tạo ban đầu. Job thực hiện các tác vụ sau khi MinIO đã sẵn sàng:

**Kết nối:** Sử dụng thông tin từ Secret minio-credentials để thiết lập alias local tới MinIO Service.

**Tạo Bucket:** Lệnh mc mb -p local/traffic-data đảm bảo bucket traffic-data được tạo ra (tham số -p giúp bỏ qua lỗi nếu bucket đã tồn tại).

**Thiết lập vòng đời (Lifecycle Management):** Áp dụng quy tắc ilm rule add để tự động xóa các đối tượng dữ liệu cũ sau **90 ngày**. Điều này giúp kiểm soát dung lượng lưu trữ, ngăn chặn việc PVC bị đầy theo thời gian mà không cần can thiệp thủ công.