TRƯỜNG ĐH CÔNG NGHỆ KỸ THUẬT TP. HỒ CHÍ MINH

KHOA CÔNG NGHỆ THÔNG TIN

ĐỒ ÁN CUỐI KỲ

MÔN HỌC: XỬ LÝ DỮ LIỆU LỚN

XÂY DỰNG HỆ THỐNG DATA LAKEHOUSE STREAMING TRÊN AZURE BLOB STORAGE ĐỂ PHÂN TÍCH VÀ DỰ ĐOÁN ÙN TẮC GIAO THÔNG TẠI NEWYORK

Giảng viên: ThS. Lê Thị Minh Châu

Danh sách sinh viên thực hiện

Mã số SV	Họ và tên	Mức độ đóng góp (%)
23110280	Huỳnh Thanh Nhân	100
23110300	Trương Tấn Sang	100
23110327	Huỳnh Ngọc Thắng	100

TP. Hồ Chí Minh, tháng 07 năm 2026

BẢNG PHÂN CÔNG CÔNG VIỆC

STT	Mã SV	Họ và tên	Chi tiết công việc
1	23110327	Huỳnh Ngọc Thắng	Thiết lập Docker Compose cho hệ thống; cấu hình Azure Blob Storage và cấu trúc Bronze – Silver – Gold – Artifacts. (Mục 3.4 và 3.5) Xây dựng pipeline Bronze -> Silver để đọc dữ liệu historical, chuẩn hóa schema, lọc cột không cần thiết, xử lý dữ liệu lỗi. (Mục 4.1)
2	23110280	Huỳnh Thanh Nhân	Xây dựng pipeline Silver → Gold để tạo tập dữ liệu huấn luyện. Tính free_flow_speed, speed_ratio, nhãn dự đoán 15 phút tới và các đặc trưng thời gian, lag, trend. (Mục 4.2) Chia Train/Validation/Test theo thời gian, huấn luyện RandomForestClassifier bằng Spark MLlib, sau đó lưu model lên Azure Blob Storage. (Mục 4.3) Đánh giá bằng Macro F1-score và Confusion Matrix (Mục 4.4)
3	23110300	Trương Tấn Sang	Xây dựng Job A để thu thập dữ liệu realtime từ NYC DOT Feed. Cấu hình Kafka truy xuất dữ liệu 90 phút gần nhất. Xây dựng Job B để đọc Kafka, load model và dự đoán. Ghi kết quả vào Gold layer và xây dựng Power BI dashboard để trực quan hóa kết quả dự đoán. (Đã sửa lỗi trên máy là làm lại trên máy cá nhân, có chụp minh chứng)

LỜI CẢM ƠN

Lời đầu tiên, nhóm xin được gửi lời cảm ơn chân thành nhất đến ThS. Lê Thị Minh Châu. Qua những bài học, lời khuyên và sự góp ý của cô đã giúp tất cả các thành viên trong nhóm hiểu hơn về môn học Xử lý dữ liệu lớn cũng như trong quá trình hoàn thành báo cáo bài tập cuối kỳ.

Do thời gian đợt học lần này đã ngắn đi nên cùng với những kiến thức và bài học tiếp thu được, nhóm đã cố gắng thực hiện đề tài hết mình, cùng nhau thực hiện một cách tốt nhất với khả năng. Chính vì thế trong quá trình đôi khi sẽ có những sai sót không mong muốn, từ đó nhóm hi vọng sẽ nhận được sự góp ý từ cô để nhóm có thể sửa chữa và rút ra bài học hướng đến mục tiêu là hoàn thiện các đồ án sau này.

Một lần nữa nhóm xin chân thành cảm ơn cô. Kính chúc cô thật nhiều sức khỏe, niềm vui trong sự nghiệp trồng người. Nhóm chúng em xin cảm ơn cô.

TP. Hồ Chí Minh, ngày 02 tháng 07 năm 2026

Nhóm sinh viên thực hiện

Huỳnh Thanh Nhân

Trương Tấn Sang

Huỳnh Ngọc Thắng

MỤC LỤC

BẢNG PHÂN CÔNG CÔNG VIỆC .....
LỜI CẢM ƠN .....
MỤC LỤC .....
DANH MỤC HÌNH ẢNH .....
PHẦN MỞ ĐẦU 1
1. Lý do chọn đề tài 1
2. Mục tiêu của đồ án 1
2.1. Mục tiêu tổng quát 1
2.2. Mục tiêu cụ thể 1
3. Bài toán cần giải quyết 2
3.1. Phát biểu bài toán 2
3.2. Đối tượng dự đoán 2
3.3. Đầu vào của bài toán 2
3.4. Đầu ra của bài toán 2
4. Phạm vi thực hiện đồ án 3
4.1. Phạm vi dữ liệu 3
4.2. Phạm vi kỹ thuật 3
4.3. Phạm vi phân tích 3
5. Phương pháp thực hiện 3
6. Bố cục báo cáo 3
PHẦN NỘI DUNG 4
CHƯƠNG 1. CƠ SỞ LÝ THUYẾT VỀ BIG DATA 4
1.1. Tổng quan về Big Data 4
1.1.1. Khái niệm Big Data 4
1.1.2. Đặc trưng của Big Data 4
1.1.3. Vai trò của Big Data trong bài toán giao thông đô thị 5
1.2. Tổng quan về Data Lakehouse 6
1.2.1. Data Warehouse 6
1.2.2. Data Lake 6

1.2.3. Data Lakehouse 7
1.3. Kiến trúc Lakehouse 8
1.3.1. Lớp Ingestion - Thu thập dữ liệu 8
1.3.2. Lớp Storage - Lưu trữ dữ liệu 8
1.3.3. Lớp Metadata - Quản lý siêu dữ liệu 9
1.3.4. Lớp API/Processing - Truy cập và xử lý dữ liệu 9
1.3.5. Lớp Consumption - Khai thác dữ liệu 9
1.4. Kiến trúc Medallion trong Lakehouse 9
1.4.1. Bronze Layer - Dữ liệu thô 9
1.4.2. Silver Layer - Dữ liệu đã làm sạch 10
1.4.3. Gold Layer - Dữ liệu phục vụ phân tích, huấn luyện và trực quan hóa 10
1.5. Các công nghệ sử dụng trong đề tài 10
1.5.1. Docker 10
1.5.2. Apache Kafka 10
1.5.3. Apache Spark 11
1.5.4. Spark Mllib 11
1.5.5. Delta Lake và Apache Parquet 12
1.5.6. Azure Blob Storage 12
1.5.7. Power BI 13
CHƯƠNG 2. BÀI TOÁN, NGUỒN DỮ LIỆU VÀ KỊCH BẢN PHÂN TÍCH 14
2.1. Mô tả bài toán dự đoán ùn tắc giao thông 14
2.1.1. Bối cảnh giao thông tại thành phố New York 14
2.1.2. Mục tiêu dự đoán mức độ tắc nghẽn 14
2.1.3. Đơn vị dự đoán 14
2.1.4. Lý do chọn mốc dự đoán 15 phút 15
2.2. Cơ sở phân loại mức độ tắc nghẽn 15
2.2.1. Khái niệm free_flow_speed 15
2.2.2. Công thức speed_ratio 16
2.2.3. Quy tắc phân loại 3 mức tắc nghẽn 16
2.2.4. Ý nghĩa các lớp: Thông thoáng, Tắc nghẽn, Nghiêm trọng 18
2.3. Nguồn dữ liệu sử dụng 18
2.3.1. Dữ liệu lịch sử: NYC Open Data - DOT Traffic Speeds NBE 18
2.3.2. Dữ liệu gần thời gian thực: NYC DOT Real-Time Traffic Speed Feed 18

2.4. Mô tả cấu trúc dữ liệu 19
2.4.1. Các cột dữ liệu ban đầu 19
2.4.2. Các cột được sử dụng trong đề tài 19
2.4.3. Các cột bị loại bỏ 19
2.4.4. Đặc điểm cập nhật của dữ liệu 20
2.4.5. Vấn đề độ trễ, khoảng trống thời gian và dữ liệu cảm biến lỗi 20
2.5. Phạm vi dữ liệu huấn luyện 20
2.5.1. Lý do chọn dữ liệu từ 01/2024 đến 2026 20
2.5.2. Lý do chỉ sử dụng các link_id đang hoạt động 21
CHƯƠNG 3. THIẾT KẾ, THIẾT LẬP VÀ TRIỂN KHAI KIẾN TRÚC DATA LAKEHOUSE 22
3.1. Yêu cầu thiết kế hệ thống 22
3.1.1. Yêu cầu thu thập dữ liệu 22
3.1.2. Yêu cầu lưu trữ dữ liệu lớn 23
3.1.3. Yêu cầu xử lý dữ liệu bằng Apache Spark 23
3.1.4. Yêu cầu huấn luyện mô hình dự đoán 24
3.1.5. Yêu cầu dự đoán gần thời gian thực 24
3.1.6. Yêu cầu trực quan hóa kết quả 24
3.2. Tổng quan kiến trúc hệ thống 24
3.2.1. Luồng dữ liệu lịch sử phục vụ huấn luyện 25
3.2.2. Luồng dữ liệu real-time phục vụ dự đoán 25
3.2.3. Luồng dữ liệu Gold phục vụ Power BI 26
3.3. Ánh xạ 5 lớp Lakehouse vào đề tài 26
3.3.1. Ingestion: NYC Open Data API, NYC DOT Feed và Apache Kafka 26
3.3.2. Storage: Azure Blob Storage, Delta Lake và Parquet 27
3.3.3. Metadata: schema, partition, lookup table và model artifacts 27
3.3.4. Processing/API: Apache Spark, Spark MLlib, Job A và Job B 27
3.3.5. Consumption: Power BI Dashboard 27
3.4. Cấu trúc thư mục trên Azure Blob Storage 27
3.5. Thiết lập môi trường triển khai 30
3.5.1. Cài đặt Docker Desktop 30
3.5.2. Tạo cấu trúc thư mục project 32
3.5.3. Tạo Azure Blob Storage 33

3.5.4. Cấu hình file .env 36
3.5.5. Thiết lập file docker-compose.yml.....37
3.5.6. Khởi động Kafka, Spark Master và Spark Worker.....39
3.5.7. Kiểm tra Spark UI 40
3.5.8. Khởi động Jupyter Notebook trong cùng Docker network 41
CHƯƠNG 4: XỬ LÝ DỮ LIỆU, HUẤN LUYỆN MÔ HÌNH VÀ ĐÁNH GIÁ.....44
4.1. Quy trình thu thập, lưu Bronze và xử lý dữ liệu từ Bronze sang Silver.....44
4.1.1. Thu thập dữ liệu lịch sử từ NYC Open Data.....44
4.1.2. Thu thập dữ liệu real-time từ NYC DOT Feed 47
4.1.3. Chuẩn hóa schema dữ liệu, kiểu dữ liệu, múi giờ, lọc dữ liệu cảm biến lỗi, lọc các link_id đang hoạt động và trích xuất đặc trưng thời gian 50
4.2. Xử lý từ Silver sang Gold 53
4.2.1. Tính free_flow_speed theo P85 của từng link_id 55
4.2.2. Tính speed_ratio và xác định current_congestion.....57
4.2.3. Tạo đặc trưng trễ 15, 30, 45 và 60 phút, nhãn future_congestion_15min 57
4.2.4. Tạo đặc trưng xu hướng tắc nghẽn và xu hướng tốc độ.....59
4.2.5. Danh sách đặc trưng đầu vào và biến mục tiêu đầu ra.....59
4.3. Xử lý và huấn luyện mô hình RandomForestClassifier bằng Spark MLlib 63
4.3.1. Chia Train, Validation và Test theo thời gian 66
4.3.2. Xử lý mất cân bằng lớp bằng weightCol.....67
4.3.3. Xây dựng Pipeline trong Spark Mllib 68
4.3.4. Lưu mô hình lên Azure Blob Storage.....69
4.4. Đánh giá mô hình dự đoán.....71
4.4.1. Phương pháp tính Macro F1 71
4.4.2. Kết quả đánh giá trên Validation set.....72
4.4.3. Kết quả đánh giá trên Test set 72
4.4.4. Phân tích Confusion Matrix trên Test set 73
4.4.5. Phân tích mức độ quan trọng của đặc trưng.....75
4.4.6. Phân tích tương quan giữa đặc trưng và nhãn.....77
4.5. Triển khai luồng dự đoán gần thời gian thực.....78
4.5.1. Cấu trúc Kafka topic nyc-traffic-snapshots.....78
4.5.2. Cơ chế hoạt động của Job A – Collector 79
4.5.3. Cơ chế warm-up dữ liệu Kafka 84

4.5.4. Cơ chế hoạt động của Job B – Predictor 85
4.5.5. Cơ chế tìm lag features từ dữ liệu Kafka 88
4.5.6. Cơ chế bỏ qua các link_id không đủ dữ liệu 89
4.5.7. Ghi kết quả dự đoán vào Gold Layer 90
4.6. Kết quả triển khai và kiểm thử hệ thống.....91
4.6.1. Kiểm tra kết quả dự đoán trong Gold Layer 91
4.6.2. Kiểm tra dashboard Power BI 92
PHẦN KẾT LUẬN 99
1. Kết quả đạt được 99
1.1. Ưu điểm 99
1.2. Nhược điểm 99
2. Hướng phát triển 100
TÀI LIỆU THAM KHẢO .....

DANH MỤC HÌNH ẢNH

Hình 1.1: Các đặc trưng của Bigdata.....5
Hình 1.2: Minh họa Data Lakehouse.....7
Hình 1.3: Kiến trúc 5 lớp trong Data Lakehouse 8
Hình 1.4: Kiến trúc Medallion trong Lakehouse.....9
Hình 1.5: Docker 10
Hình 1.6: Apache Kafka 11
Hình 1.7: Apache Spark.....11
Hình 1.8: Spark Mllib.....12
Hình 1.9: Delta Lake và Apache Parquet 12
Hình 1.10: Azure Blob Storage 13
Hình 1.11: PowerBI.....13
Hình 3.1: Trang web NYC Open Data – DOT Traffic Speeds NBE.....22
Hình 3.2: Trang NYC DOT Real-Time Traffic Speed Feed.....23
Hình 3.3: Tổng quan kiến trúc hệ thống.....25
Hình 3.4: Cấu trúc thư mục lưu trữ Azure Blob Storage.....30
Hình 3.5: Giao diện Docker Desktop 31
Hình 3.6: Biến môi trường 31
Hình 3.7: Kiểm tra Docker version 32
Hình 3.8: Cấu trúc thư mục project.....33
Hình 3.9: Tạo Azure Blob Storage 34
Hình 3.10: Tạo container.....35
Hình 3.11: Kiểm tra thông tin đăng nhập 36
Hình 3.12: Cấu hình file .env 37
Hình 3.13: Dockerfile tùy chỉnh cho spark-master và spark-slave 37
Hình 3.14: Thiết lập file docker-compose.yml.....38
Hình 3.15: Khởi động Kafka, Spark Master và Spark Worker.....39
Hình 3.16: Giao diện docker sau khi khởi động các container 40
Hình 3.17: Kiểm tra phiên bản python của master và slave.....40
Hình 3.18: Giao diện Spark UI.....40
Hình 3.19: Khởi động Jupyter Notebook trong cùng Docker network 42
Hình 3.20: Truy cập Jupyter Notebook thông qua trình duyệt web 43
Hình 4.1: Luồng xử lý lưu dữ liệu vào tầng Bronze 44

Hình 4.2: Luồng xử lý từ Bronze sang Silver 44
Hình 4.3: Hàm thu thập dữ liệu lịch sử từ NYC Open Data 45
Hình 4.4: Hàm ghi dữ liệu lịch sử xuống Bronze Layer và kiểm tra 46
Hình 4.5: Kiểm tra dữ liệu Bronze trên Azure Blob Storage 47
Hình 4.6: Kiểm tra định dạng của dữ liệu Bronze.....47
Hình 4.7: Gọi real-time feed và kiểm tra format của NYC DOT.....48
Hình 4.8: Lọc ra active_link_ids và upload lên Azure Blob Storage.....48
Hình 4.9: Vận hành real-time 49
Hình 4.10: Kiểm tra dữ liệu Bronze realtime trên Azure Blob Storage 50
Hình 4.11: Hàm các bước xử lý Bronze sang Silver 52
Hình 4.12: Kiểm tra định dạng dữ liệu sau khi thực hiện xử lý Bronze sang Silver 52
Hình 4.13: Kiểm tra dữ liệu Silver trên Azure Blob Storage và định dạng của dữ liệu53
Hình 4.14: Minh họa luồng xử lý từ Silver sang Gold.....53
Hình 4.15: Thiết lập SparkSession, các biến môi trường trong 04_silver_to_gold.ipynb 54
Hình 4.16: Khai báo các đường dẫn để làm việc với Azure Blob Storage.....54
Hình 4.17: Đọc Silver và kiểm tra tổng quan.....55
Hình 4.18: Tách riêng các tháng train để tính toán 55
Hình 4.19: Tính free_flow_speed bằng P85 trên tập Train 56
Hình 4.20: Ghi file free_flow_speed_lookup lên Azure Blob Storage 56
Hình 4.21: Join thêm cột vào lookup.....57
Hình 4.22: Tính tỷ lệ tốc độ và phân loại trạng thái tắc nghẽn hiện tại thành ba lớp. ..57
Hình 4.23: Tạo các đặc trưng trễ 15, 30, 45, 60 phút và nhãn dự đoán sau 15 phút.....58
Hình 4.24: Lọc lại DataFrame df_gold.....59
Hình 4.25: Loại bỏ các dòng bị thiếu dữ liệu quan trọng.....59
Hình 4.26: Tạo đặc trưng xu hướng tắc nghẽn và xu hướng tốc độ.....59
Hình 4.27: Đặc trưng đầu vào và biến mục tiêu đầu ra.....60
Hình 4.28: Output tổng số cột 60
Hình 4.29: Tiến hành ghi vào gold 62
Hình 4.30: Load lại từ tầng gold kiểm tra 62
Hình 4.31: Kiểm tra gold trên Azure Blob Storage.....63
Hình 4.32: Minh họa luồng hoạt động trong xử lý và huấn luyện 63
Hình 4.33: File 05_train_model.ipynb 64

Hình 4.34: Thiết lập đường dẫn và các biến môi trường làm việc 64
Hình 4.35: Đọc dữ liệu từ Gold 65
Hình 4.36: Chuẩn bị dữ liệu cho huấn luyện 66
Hình 4.37: Ghi checkpoint lên Azure Blob Storage 66
Hình 4.38: Checkpoint đã ghi trên Azure Blob Storage 66
Hình 4.39: Chia dữ liệu Train, Validation và Test theo trục thời gian 67
Hình 4.40: Tính trọng số lớp theo inverse frequency 68
Hình 4.41: Pipeline trong Spark Mllib 69
Hình 4.42: Huấn luyện mô hình 69
Hình 4.43: Lưu model 70
Hình 4.44: Kiểm tra model đã ghi trên Azure Blob Storage 71
Hình 4.45: Hàm tính Macro F1, Precision và Recall cho từng lớp 72
Hình 4.46: Kết quả đánh giá mô hình trên Validation set 72
Hình 4.47: Kết quả đánh giá mô hình trên Test set 73
Hình 4.48: Confusion Matrix trên Test set 75
Hình 4.49: Mức độ quan trọng của các đặc trưng trong mô hình RandomForest 76
Hình 4.50: Heatmap tương quan giữa đặc trưng và nhãn 77
Hình 4.51: Topic được đặt tên nyc-traffic-snapshots 79
Hình 4.52: Message được cấu trúc dưới dạng JSON 79
Hình 4.53: Job A 80
Hình 4.54: Job A - trường hợp xử lý 82
Hình 4.55: Bronze realtime trên Azure Blob Storage 82
Hình 4.56 : Job A đã khởi động thành công và ghi dữ liệu vào tầng bronze 83
Hình 4.57: Cơ chế warm-up dữ liệu Kafka 85
Hình 4.58: Cơ chế hoạt động của Job B 86
Hình 4.59: Chạy job B 87
Hình 4.60: Job B tính các đặc trưng 87
Hình 4.61 : Job B – Predictor khi pipeline realtime đã bắt đầu dự đoán thành công 88
Hình 4.62: Tìm lag features từ dữ liệu Kafka 89
Hình 4.63: Lọc lag từ dữ liệu Kafka 90
Hình 4.64: Ghi kết quả dự đoán vào Gold Layer 91
Hình 4.65 : Kiểm tra kết quả trên Gold layer 91
Hình 4.66: Kiểm tra kết quả dự đoán trên Azure Blob Storage 92

Hình 4.67: Code kết nối với tầng gold 93
Hình 4.68: Bảng preview query của df_gold 93
Hình 4.69: Code lấy vị trí của đường 94
Hình 4.70: Bảng preview query của df_coords 94
Hình 4.71: Lấy tốc độ và tên của đường 95
Hình 4.72: Bảng preview query của df_lookup 95
Hình 4.73: Code lấy data realtime 96
Hình 4.74: df_realtime, df_dashboard và df realtime_dashboard biểu thị thông tin hoàn chỉnh cho dự đoán 15 phút tới 97
Hình 4.75: Bảng dự đoán tắc nghẽn trong 15 phút tiếp theo.....98
Hình 4.76: Bản đồ độ tắc nghẽn hiện tại và trong 15 phút tiếp theo.....98
Hình 4.77: Bảng các Link_name tắc nghẽn theo giờ (Theo cơ chế lọc) và Biểu đồ phân bố tắc nghẽn theo quận 98

PHẦN MỞ ĐẦU

1. Lý do chọn đề tài

Trong bối cảnh đô thị hóa ngày càng phát triển, giao thông tại các thành phố lớn luôn là một trong những lĩnh vực phát sinh khối lượng dữ liệu lớn, liên tục và có giá trị phân tích cao. Thành phố New York là một đô thị có mật độ giao thông dày đặc, hệ thống đường bộ phức tạp và nhu cầu di chuyển lớn. Vì vậy, việc thu thập, lưu trữ, xử lý và phân tích dữ liệu giao thông có ý nghĩa quan trọng trong việc theo dõi tình trạng của hệ thống giao thông, đồng thời hỗ trợ dự đoán sớm các khu vực có nguy cơ ùn tắc.

Trong thực tế, dữ liệu giao thông thường được sinh ra liên tục từ nhiều nguồn như cảm biến tốc độ, hệ thống giám sát giao thông và các API công khai. Những dữ liệu này có đặc điểm là khối lượng lớn, cập nhật thường xuyên, có yếu tố thời gian rõ rệt và cần được xử lý bằng các công nghệ phù hợp. Nếu chỉ sử dụng các phương pháp lưu trữ và xử lý truyền thống, hệ thống sẽ gặp nhiều hạn chế trong việc mở rộng, quản lý dữ liệu thô, xử lý dữ liệu gần thời gian thực và phục vụ đồng thời cho phân tích, huấn luyện mô hình cũng như trực quan hóa.

Từ yêu cầu trên, nhóm lựa chọn đề tài *"Xây dựng hệ thống Data Lakehouse Streaming trên Azure Blob Storage để phân tích và dự đoán ùn tắc giao thông tại thành phố New York"*. Đề tài phù hợp với định hướng vì gắn với dữ liệu giao thông có khối lượng lớn, được cập nhật liên tục và cần được tổ chức, xử lý bằng một kiến trúc dữ liệu hiện đại. Việc sử dụng Data Lakehouse giúp kết hợp khả năng lưu trữ linh hoạt của Data Lake với khả năng quản lý và khai thác dữ liệu của Data Warehouse, từ đó đáp ứng tốt nhu cầu phân tích dữ liệu lịch sử và dự đoán trên dữ liệu gần thời gian thực.

2. Mục tiêu của đồ án

2.1. Mục tiêu tổng quát

Xây dựng hệ thống Data Lakehouse Streaming phục vụ phân tích và dự đoán mức độ ùn tắc giao thông tại NYC.

2.2. Mục tiêu cụ thể

Để đạt được mục tiêu tổng quát, đồ án tập trung vào các mục tiêu cụ thể sau: thu thập dữ liệu giao thông từ NYC DOT, tổ chức lưu trữ dữ liệu trên AZURE BLOB STORAGE theo kiến trúc Data Lakehouse, xử lý dữ liệu bằng Apache Spark, xây dựng tập dữ liệu đặc trưng phục vụ mô hình dự đoán, huấn luyện mô hình

RandomForestClassifier bằng Spark MLlib, triển khai luồng dự đoán gần thời gian thực bằng Apache Kafka và trực quan hóa kết quả bằng Power BI.

3. Bài toán cần giải quyết

3.1. Phát biểu bài toán

Bài toán chính của đồ án là dự đoán mức độ tắc nghẽn giao thông trong 15 phút tiếp theo cho từng đoạn đường tại thành phố New York dựa trên dữ liệu tốc độ giao thông lịch sử và dữ liệu gần thời gian thực.

Cụ thể với mỗi đoạn đường tại một thời điểm xác định, cần xác định xem sau 15 phút đoạn đường đó sẽ có trạng thái gì. Đối với bài toán này, việc dự đoán trước trong khoảng thời gian ngắn có ý nghĩa thực tế hơn so với dự đoán quá xa, vì tình trạng giao thông đô thị thường thay đổi nhanh theo thời điểm, khu vực và điều kiện vận hành.

3.2. Đối tượng dự đoán

Đối tượng dự đoán của hệ thống là từng đoạn đường trong mạng lưới giao thông NYC, được định danh bằng trường link_id. Mỗi link_id đại diện cho một đoạn đường có hướng di chuyển cụ thể.

3.3. Đầu vào của bài toán

Đầu vào của bài toán là dữ liệu tốc độ giao thông theo từng đoạn đường từ NYC DOT. Từ dữ liệu này, hệ thống xây dựng các đặc trưng thời gian, đặc trưng tốc độ, đặc trưng tắc nghẽn hiện tại và đặc trưng trễ để phục vụ mô hình dự đoán.

3.4. Đầu ra của bài toán

Đầu ra của bài toán là mức độ tắc nghẽn giao thông dự đoán sau 15 phút cho từng đoạn đường. Kết quả dự đoán được phân thành ba lớp. Lớp 0 biểu diễn trạng thái Thông thoáng, tức là đoạn đường có tốc độ di chuyển tương đối tốt so với tốc độ tự do. Lớp 1 biểu diễn trạng thái Tắc nghẽn, tức là tốc độ di chuyển đã giảm đáng kể và dòng phương tiện bị hạn chế. Lớp 2 biểu diễn trạng thái Nghiêm trọng, tức là đoạn đường có mức độ ùn tắc cao, tốc độ thấp và có nguy cơ rơi vào trạng thái di chuyển chậm hoặc gián đoạn.

Ngoài nhãn dự đoán, hệ thống còn có thể sinh ra độ tin cậy của dự đoán thông qua xác suất đầu ra của mô hình. Kết quả này được ghi xuống tầng Gold của Data Lakehouse và được sử dụng để xây dựng dashboard trực quan hóa trên Power BI.

4. Phạm vi thực hiện đồ án

4.1. Phạm vi dữ liệu

Đồ án sử dụng dữ liệu lịch sử từ NYC Open Data - DOT Traffic Speeds NBE để xây dựng tập dữ liệu huấn luyện, đồng thời sử dụng dữ liệu gần thời gian thực từ NYC DOT Real-Time Traffic Speed Feed để phục vụ quá trình dự đoán.

Dữ liệu lịch sử được sử dụng trong phạm vi từ tháng 01/2024 đến tháng 06/2026.

4.2. Phạm vi kỹ thuật

Đồ án sử dụng Docker để triển khai môi trường, Apache Kafka để tiếp nhận dữ liệu gần thời gian thực, Apache Spark để xử lý dữ liệu phân tán, Spark MLlib để huấn luyện mô hình, Azure Blob Storage kết hợp Delta Lake/Parquet để lưu trữ dữ liệu theo kiến trúc Lakehouse (3 tầng) và Power BI để trực quan hóa kết quả.

4.3. Phạm vi phân tích

Phạm vi phân tích của đồ án bao gồm phân tích dữ liệu lịch sử, xây dựng đặc trưng, huấn luyện mô hình, dự đoán gần thời gian thực và trực quan hóa kết quả. Với dữ liệu lịch sử, đồ án phân tích tình trạng giao thông theo đoạn đường, quận, thời gian trong ngày và xu hướng thay đổi trong 60 phút gần nhất. Với mô hình dự đoán, đồ án tập trung vào bài toán phân loại mức độ tắc nghẽn trong 15 phút tiếp theo.

5. Phương pháp thực hiện

Đồ án được thực hiện bao gồm thu thập dữ liệu, lưu trữ theo kiến trúc Lakehouse, xử lý bằng Spark, huấn luyện mô hình bằng Spark MLlib, triển khai dự đoán gần thời gian thực qua Kafka và trực quan hóa bằng Power BI. Quy trình triển khai chi tiết được trình bày trong Chương 3 và Chương 4.

6. Bố cục báo cáo

Báo cáo được tổ chức thành: phần mở đầu, phần nội dung và phần kết luận.

Phần nội dung gồm bốn chương:

Chương 1 trình bày cơ sở lý thuyết về Big Data và Data Lakehouse

Chương 2 trình bày bài toán, nguồn dữ liệu và kịch bản phân tích

Chương 3 trình bày thiết kế, thiết lập và triển khai kiến trúc Data Lakehouse

Chương 4 trình bày quy trình xử lý dữ liệu, xây dựng đặc trưng, huấn luyện mô hình, dự đoán gần thời gian thực, kiểm thử hệ thống, demo và đánh giá kết quả.

PHẦN NỘI DUNG

CHƯƠNG 1. CƠ SỞ LÝ THUYẾT VỀ BIG DATA

1.1. Tổng quan về Big Data

Big Data, hay dữ liệu lớn, là nền tảng quan trọng trong các hệ thống dữ liệu hiện đại. Sự phát triển của Internet, điện toán đám mây, cảm biến và các hệ thống thu thập dữ liệu tự động làm cho dữ liệu ngày càng lớn về quy mô, nhanh về tốc độ phát sinh và đa dạng về định dạng. Vì vậy, các phương pháp lưu trữ và xử lý truyền thống thường không còn đáp ứng tốt, đặc biệt đối với những bài toán cần xử lý dữ liệu liên tục và phân tích gần thời gian thực.

1.1.1. Khái niệm Big Data

Big Data là thuật ngữ dùng để chỉ các tập dữ liệu có khối lượng lớn, tốc độ phát sinh nhanh, đa dạng về nguồn và định dạng, đồng thời cần các công nghệ xử lý phù hợp để lưu trữ, quản lý, phân tích và khai thác giá trị. Big Data không chỉ được hiểu là dữ liệu có dung lượng lớn, mà còn là dữ liệu vượt quá khả năng xử lý hiệu quả của các công cụ truyền thống.

1.1.2. Đặc trưng của Big Data

Big Data thường được mô tả thông qua mô hình 5V, gồm Volume, Velocity, Variety, Veracity và Value.

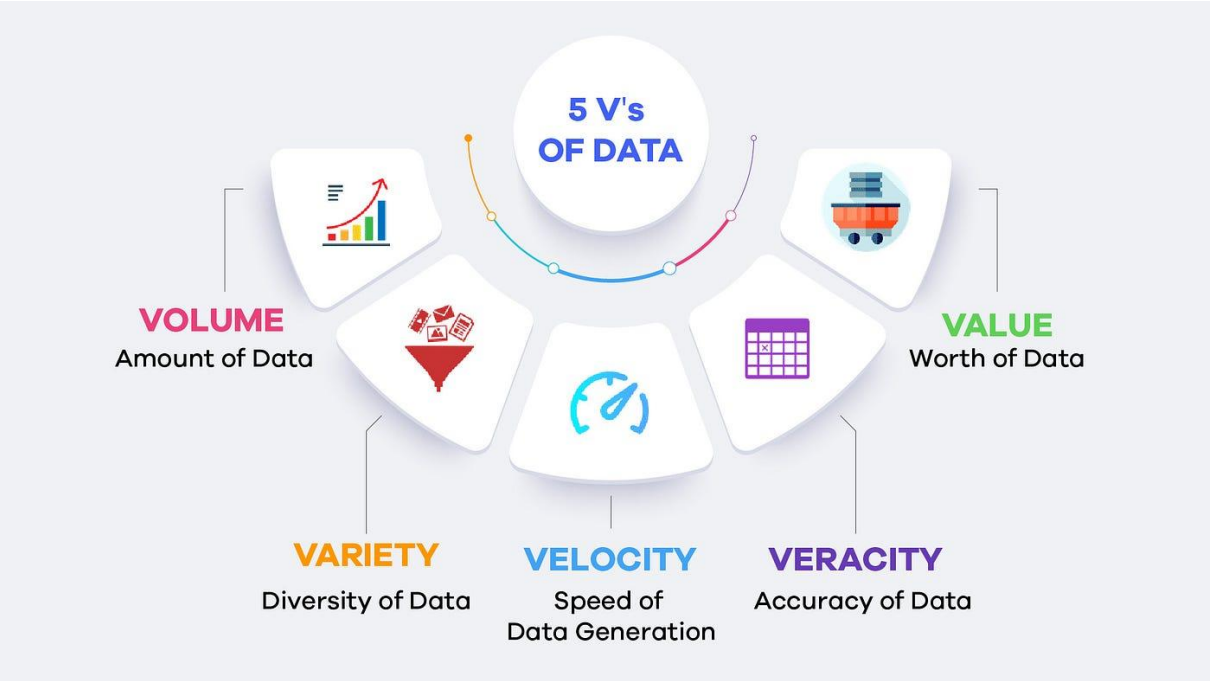
Volume - Khối lượng dữ liệu. Trong bài toán giao thông, dữ liệu tăng theo số lượng đoạn đường, tần suất cập nhật và thời gian thu thập. Khi dữ liệu lịch sử được tích lũy trong nhiều tháng, hệ thống cần một nền tảng lưu trữ có khả năng mở rộng như Azure Blob Storage.

Velocity - Tốc độ phát sinh và xử lý dữ liệu. Với dữ liệu giao thông, các bản ghi mới có thể được cập nhật liên tục hoặc gần thời gian thực. Điều này đặt ra yêu cầu hệ thống không chỉ lưu trữ được dữ liệu lịch sử, mà còn phải tiếp nhận và xử lý dữ liệu mới trong quá trình vận hành.

Variety - Sự đa dạng của dữ liệu. Dữ liệu trong đề tài đến từ dữ liệu lịch sử, dữ liệu real-time, thông tin đoạn đường, quận, thời gian ghi nhận và các đặc trưng được tạo ra trong quá trình xử lý. Vì vậy, hệ thống cần chuẩn hóa schema và tổ chức dữ liệu theo các tầng Bronze, Silver và Gold.

Veracity - Độ tin cậy và chất lượng dữ liệu. Dữ liệu giao thông có thể chứa bản ghi lỗi, tốc độ bất thường, dữ liệu thiếu hoặc sai lệch thời gian. Do đó, các bước làm sạch, lọc lỗi và chuẩn hóa dữ liệu là cần thiết trước khi đưa dữ liệu vào mô hình dự đoán.

Value - Giá trị khai thác từ dữ liệu. Trong đề tài, giá trị của Big Data nằm ở khả năng phân tích tình trạng giao thông, nhận diện xu hướng ùn tắc và dự đoán mức độ ùn tắc trong tương lai gần. Kết quả dự đoán được lưu ở tầng Gold và trực quan hóa bằng Power BI.



Hình 1.1: Các đặc trưng của Bigdata

1.1.3. Vai trò của Big Data trong bài toán giao thông đô thị

Giao thông đô thị là một lĩnh vực có nhiều đặc điểm phù hợp với Big Data. Dữ liệu giao thông thường được phát sinh liên tục, có tính thời gian rõ rệt và phản ánh trạng thái vận hành của hệ thống đường bộ. Khi được thu thập và xử lý hiệu quả, dữ liệu này có thể hỗ trợ phân tích mật độ giao thông, phát hiện các khu vực thường xuyên ùn tắc, đánh giá xu hướng di chuyển theo thời gian và dự đoán tình trạng giao thông.

Trong các thành phố lớn như New York, tình trạng giao thông có thể thay đổi nhanh theo giờ cao điểm, thời tiết, sự kiện, tai nạn hoặc các điều kiện vận hành khác. Vì vậy, việc dựa vào dữ liệu lịch sử kết hợp dữ liệu gần thời gian thực giúp hệ thống có khả năng nhận biết diễn biến giao thông tốt hơn. Các mô hình học máy có thể khai thác mối

quan hệ giữa tốc độ hiện tại, xu hướng tốc độ trong quá khứ gần, thời điểm trong ngày và vị trí đoạn đường để đưa ra dự đoán.

1.2. Tổng quan về Data Lakehouse

Data Lakehouse là một kiến trúc dữ liệu hiện đại được hình thành từ nhu cầu kết hợp ưu điểm của Data Warehouse và Data Lake.

1.2.1. Data Warehouse

Data Warehouse là hệ thống kho dữ liệu được thiết kế nhằm lưu trữ dữ liệu đã qua xử lý, có cấu trúc rõ ràng và phục vụ chủ yếu cho mục đích báo cáo, phân tích và ra quyết định. Dữ liệu trong Data Warehouse thường được chuẩn hóa, tích hợp từ nhiều nguồn và tổ chức theo mô hình phù hợp cho truy vấn phân tích. Ưu điểm của Data Warehouse là khả năng quản lý dữ liệu tốt, hỗ trợ truy vấn hiệu quả và đảm bảo tính nhất quán trong quá trình phân tích.

Tuy nhiên, Data Warehouse cũng có một số hạn chế khi áp dụng cho các bài toán dữ liệu lớn hiện đại. Do thường yêu cầu dữ liệu có cấu trúc rõ ràng trước khi nạp vào hệ thống, Data Warehouse kém linh hoạt hơn khi cần lưu trữ dữ liệu thô, dữ liệu bán cấu trúc hoặc dữ liệu phát sinh liên tục. Ngoài ra, chi phí lưu trữ và mở rộng của Data Warehouse có thể cao khi dữ liệu tăng nhanh về khối lượng.

Trong bài toán giao thông, nếu chỉ sử dụng Data Warehouse truyền thống, hệ thống sẽ gặp hạn chế trong việc lưu trữ dữ liệu thô, xử lý linh hoạt và phục vụ đồng thời cho nhiều mục đích như phân tích, huấn luyện mô hình và dự đoán gần thời gian thực.

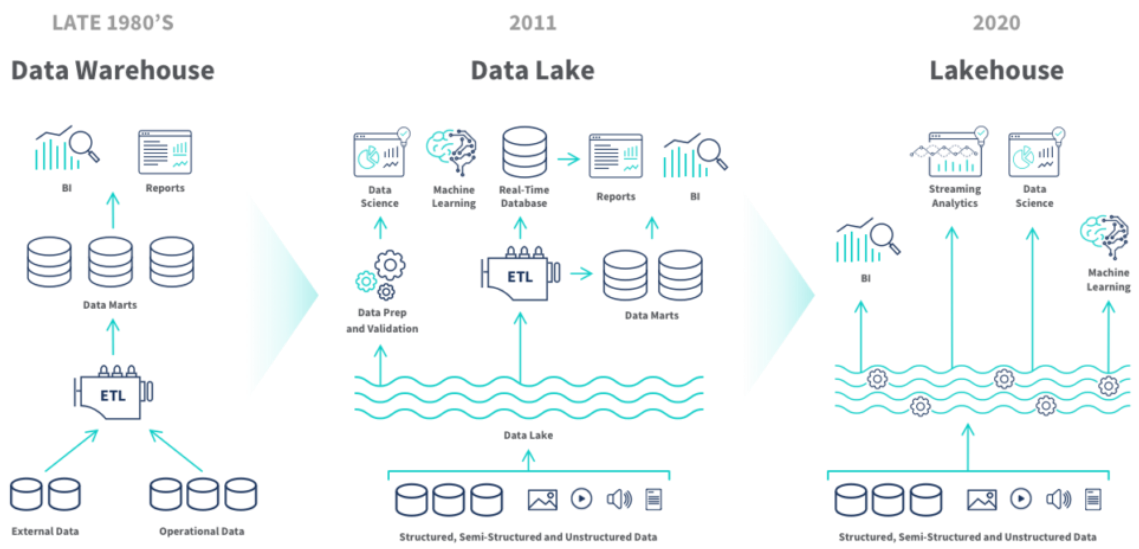
1.2.2. Data Lake

Data Lake là kiến trúc lưu trữ cho phép lưu dữ liệu ở dạng thô với nhiều định dạng khác nhau. Khác với Data Warehouse, Data Lake không yêu cầu dữ liệu phải được xử lý và chuẩn hóa hoàn toàn trước khi lưu trữ. Nhờ đó, Data Lake có khả năng lưu trữ linh hoạt, phù hợp với các hệ thống có dữ liệu lớn, dữ liệu đa dạng và dữ liệu phát sinh liên tục.

Ưu điểm lớn của Data Lake là khả năng mở rộng và chi phí lưu trữ hợp lý, đặc biệt khi kết hợp với các nền tảng lưu trữ đám mây như Azure Data Lake Storage, Azure Blob Storage, AWS S3 hoặc Google Cloud Storage. Data Lake cho phép lưu trữ dữ liệu thô, dữ liệu đã làm sạch và dữ liệu phục vụ phân tích trong cùng một hệ thống lưu trữ.

Tuy nhiên, Data Lake nếu không được quản lý tốt có thể dẫn đến tình trạng dữ liệu thiếu tổ chức, khó truy vết và khó đảm bảo chất lượng. Khi dữ liệu được lưu trữ quá tự do mà không có cơ chế quản lý metadata, schema, phiên bản và chất lượng dữ liệu, hệ thống có thể trở thành "data swamp", tức là kho dữ liệu khó sử dụng và khó kiểm soát. Do đó, các hệ thống hiện đại thường cần một kiến trúc kết hợp được sự linh hoạt của Data Lake với khả năng quản lý của Data Warehouse.

1.2.3. Data Lakehouse



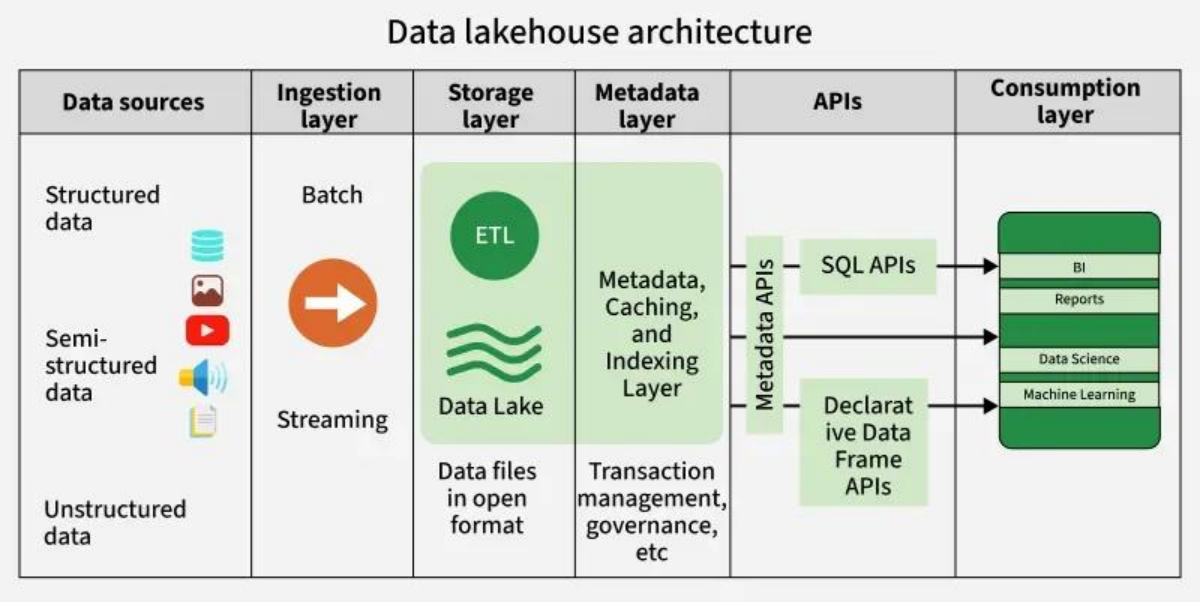
Hình 1.2: Minh họa Data Lakehouse

Data Lakehouse là kiến trúc kết hợp khả năng lưu trữ linh hoạt của Data Lake với khả năng quản lý và khai thác dữ liệu của Data Warehouse. Kiến trúc này thường sử dụng các định dạng dữ liệu mở như Parquet, kết hợp với các cơ chế quản lý bảng như Delta Lake để hỗ trợ kiểm soát schema, quản lý metadata,…

Trong kiến trúc Lakehouse, dữ liệu có thể được lưu trữ trên object storage như Azure Data Lake Storage hoặc Azure Blob Storage, sau đó được tổ chức và quản lý bằng các định dạng như Delta Lake, Apache Iceberg hoặc Apache Hudi. Các công cụ xử lý như Apache Spark đóng vai trò xử lý dữ liệu phân tán, xây dựng pipeline dữ liệu, thực hiện truy vấn, tạo đặc trưng và huấn luyện mô hình. Ở lớp khai thác, các công cụ BI như Power BI có thể sử dụng dữ liệu đã xử lý để xây dựng báo cáo và dashboard.

1.3. Kiến trúc Lakehouse

Kiến trúc Data Lakehouse có thể được tổ chức thành năm lớp chính gồm: Ingestion, Storage, Metadata, API/Processing và Consumption. Mỗi lớp đảm nhiệm một vai trò riêng trong toàn bộ vòng đời dữ liệu, từ lúc dữ liệu được thu thập, lưu trữ, quản lý, xử lý cho đến khi được khai thác phục vụ phân tích và ra quyết định. Đối với đề tài này, năm lớp Lakehouse giúp tổ chức hệ thống dự đoán ùn tắc giao thông tại New York theo một quy trình rõ ràng, có khả năng mở rộng và phù hợp với dữ liệu lớn cũng như dữ liệu gần thời gian thực.



Hình 1.3: Kiến trúc 5 lớp trong Data Lakehouse

1.3.1. Lớp Ingestion - Thu thập dữ liệu

Lớp Ingestion là lớp đầu tiên trong kiến trúc Lakehouse, chịu trách nhiệm tiếp nhận dữ liệu từ các nguồn khác nhau vào hệ thống. Dữ liệu có thể được thu thập theo dạng batch từ các nguồn lịch sử hoặc theo dạng streaming từ các nguồn cập nhật liên tục. Mục tiêu của lớp này là bảo đảm dữ liệu từ nguồn được đưa vào hệ thống một cách ổn định, đúng định dạng cần thiết và có thể tiếp tục được xử lý ở các lớp phía sau.

1.3.2. Lớp Storage - Lưu trữ dữ liệu

Lớp Storage chịu trách nhiệm lưu trữ dữ liệu của toàn hệ thống. Trong kiến trúc Lakehouse, lớp lưu trữ thường được xây dựng trên nền tảng lưu trữ đám mây hoặc hệ thống lưu trữ phân tán. Dữ liệu có thể được lưu ở nhiều định dạng khác nhau, nhưng với các bài toán phân tích dữ liệu lớn, các định dạng dạng cột như Parquet thường được ưu tiên vì hỗ trợ truy vấn hiệu quả và tiết kiệm dung lượng lưu trữ.

1.3.3. Lớp Metadata - Quản lý siêu dữ liệu

Lớp Metadata có nhiệm vụ quản lý các thông tin mô tả về dữ liệu, chẳng hạn như schema, kiểu dữ liệu, thông tin phân vùng, đường dẫn lưu trữ, bảng tra cứu và các artifacts của mô hình. Đây là lớp quan trọng giúp hệ thống hiểu dữ liệu đang được lưu ở đâu, có cấu trúc như thế nào và được sử dụng cho mục đích gì.

1.3.4. Lớp API/Processing - Truy cập và xử lý dữ liệu

Lớp API/Processing cung cấp khả năng truy cập, xử lý và biến đổi dữ liệu trong Lakehouse. Đây là lớp kết nối giữa dữ liệu được lưu trữ và các nhu cầu khai thác dữ liệu ở tầng trên. Trong các hệ thống Big Data, lớp này thường sử dụng các công cụ xử lý phân tán để xử lý dữ liệu lớn một cách hiệu quả.

1.3.5. Lớp Consumption - Khai thác dữ liệu

Lớp Consumption là lớp cuối cùng trong kiến trúc Lakehouse, nơi dữ liệu đã được xử lý được sử dụng để phục vụ phân tích, trực quan hóa, báo cáo hoặc ra quyết định. Đây là lớp thể hiện giá trị thực tế của toàn bộ hệ thống dữ liệu.

1.4. Kiến trúc Medallion trong Lakehouse

Kiến trúc Medallion là mô hình tổ chức dữ liệu phổ biến trong Data Lakehouse, gồm ba tầng chính: Bronze, Silver và Gold. Dữ liệu được xử lý theo hướng tăng dần về chất lượng: từ dữ liệu thô ban đầu, đến dữ liệu đã làm sạch, sau đó là dữ liệu đã được chuẩn hóa và làm giàu để phục vụ phân tích, huấn luyện mô hình và trực quan hóa.

Hình 1.4: Kiến trúc Medallion trong Lakehouse

1.4.1. Bronze Layer - Dữ liệu thô

Bronze Layer là tầng lưu trữ dữ liệu thô được thu thập trực tiếp từ nguồn. Dữ liệu ở tầng này thường được giữ gần nguyên bản nhằm bảo toàn thông tin gốc, phục vụ kiểm tra, đối chiếu và tái xử lý khi cần.

1.4.2. Silver Layer - Dữ liệu đã làm sạch

Silver Layer là tầng lưu trữ dữ liệu đã được làm sạch và chuẩn hóa từ tầng Bronze. Ở tầng này, hệ thống loại bỏ các bản ghi không hợp lệ, chuẩn hóa kiểu dữ liệu, chuẩn hóa thời gian và giữ lại những cột cần thiết cho quá trình phân tích tiếp theo.

1.4.3. Gold Layer - Dữ liệu phục vụ phân tích, huấn luyện và trực quan hóa

Gold Layer là tầng dữ liệu có chất lượng cao nhất trong kiến trúc Medallion. Dữ liệu ở tầng này đã được làm giàu và tổ chức theo mục đích sử dụng cụ thể như phân tích, huấn luyện mô hình, dự đoán hoặc xây dựng dashboard.

1.5. Các công nghệ sử dụng trong đề tài

Để xây dựng hệ thống Data Lakehouse Streaming phục vụ phân tích và dự đoán ùn tắc giao thông tại thành phố New York, đề tài sử dụng một số công nghệ chính gồm Docker, Apache Kafka, Apache Spark, Spark MLlib, Delta Lake, Apache Parquet, Azure Blob Storage và Power BI.

1.5.1. Docker

Docker là nền tảng dùng để đóng gói ứng dụng và môi trường chạy dưới dạng container. Mỗi container chứa các thành phần cần thiết để một dịch vụ có thể hoạt động độc lập, giúp giảm sự khác biệt giữa môi trường phát triển và môi trường triển khai. Trong đề tài, Docker được sử dụng để triển khai các thành phần như Kafka, Spark Master, Spark Worker, Job A và Job B. Nhờ đó, hệ thống dễ cài đặt, dễ kiểm thử và hạn chế lỗi khác biệt môi trường giữa các máy.

Hình 1.5: Docker

1.5.2. Apache Kafka

Apache Kafka là nền tảng truyền tải dữ liệu phân tán, thường được sử dụng trong các hệ thống streaming hoặc near real-time. Kafka hoạt động theo cơ chế topic, trong đó

dữ liệu được producer gửi vào topic và consumer đọc dữ liệu từ topic để xử lý. Kafka phù hợp với các hệ thống cần tiếp nhận dữ liệu liên tục, có khả năng mở rộng và tách biệt giữa nguồn sinh dữ liệu với các thành phần xử lý phía sau. Trong đề tài, Kafka đóng vai trò trung gian tiếp nhận dữ liệu giao thông gần thời gian thực từ NYC DOT. Job A thu thập dữ liệu tốc độ mới nhất và đẩy vào Kafka topic, trong khi Job B đọc dữ liệu từ Kafka để tạo đặc trưng trễ và thực hiện dự đoán. Kafka giúp tách rời quá trình thu thập dữ liệu và quá trình dự đoán, từ đó làm cho hệ thống linh hoạt và ổn định hơn.

Hình 1.6: Apache Kafka

1.5.3. Apache Spark

Apache Spark là công cụ xử lý dữ liệu phân tán, phù hợp với các bài toán dữ liệu lớn. Spark hỗ trợ xử lý dữ liệu theo lô, xử lý dữ liệu có cấu trúc và xử lý trên cụm máy tính. Trong đề tài, Spark được sử dụng để xử lý dữ liệu từ tầng Bronze sang Silver, từ Silver sang Gold, chuẩn hóa dữ liệu, lọc dữ liệu lỗi, tạo đặc trưng và chuẩn bị dữ liệu cho mô hình học máy.

Hình 1.7: Apache Spark

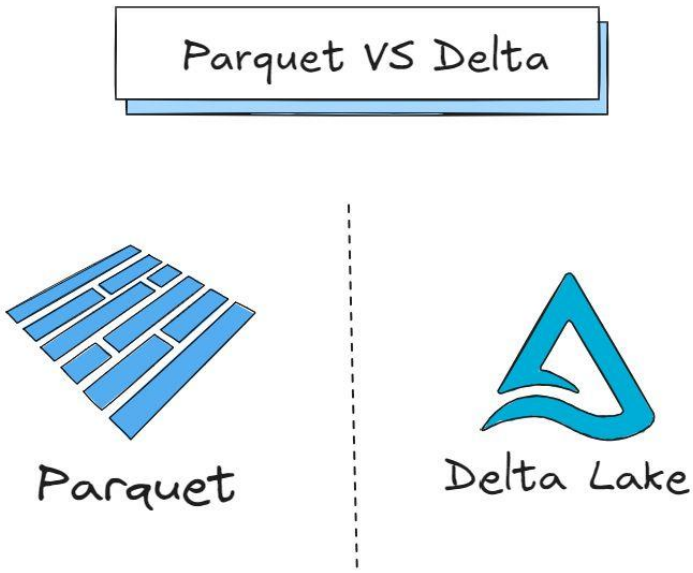
1.5.4. Spark Mllib

Spark MLlib là thư viện học máy của Apache Spark, hỗ trợ xây dựng các pipeline học máy trên dữ liệu lớn. Trong đề tài, Spark MLlib được sử dụng để huấn luyện mô hình RandomForestClassifier.

Hình 1.8: Spark Mllib

1.5.5. Delta Lake và Apache Parquet

Apache Parquet là định dạng lưu trữ dữ liệu dạng cột, giúp tối ưu hiệu năng đọc dữ liệu và tiết kiệm dung lượng lưu trữ. Delta Lake là lớp quản lý dữ liệu trên Data Lake, bổ sung các khả năng như quản lý schema, giao dịch dữ liệu và lưu vết thay đổi.



Hình 1.9: Delta Lake và Apache Parquet

1.5.6. Azure Blob Storage

Azure Blob Storage là dịch vụ lưu trữ object trên nền tảng điện toán đám mây Microsoft Azure, phù hợp để lưu trữ lượng lớn dữ liệu phi cấu trúc và bán cấu trúc. Trong đề tài, Azure Blob Storage được sử dụng làm nơi lưu trữ trung tâm của hệ thống Lakehouse. Dữ liệu lịch sử, dữ liệu real-time, dữ liệu đã làm sạch, dữ liệu đặc trưng, kết quả dự đoán và các artifacts của mô hình đều được tổ chức trên Azure Blob Storage.

Hình 1.10: Azure Blob Storage

1.5.7. Power BI

Power BI là công cụ phân tích và trực quan hóa dữ liệu của Microsoft. Công cụ này cho phép kết nối đến dữ liệu, xây dựng báo cáo, dashboard và các biểu đồ trực quan phục vụ phân tích.

Hình 1.11: PowerBI

CHƯƠNG 2. BÀI TOÁN, NGUỒN DỮ LIỆU VÀ KỊCH BẢN PHÂN TÍCH

2.1. Mô tả bài toán dự đoán ùn tắc giao thông

2.1.1. Bối cảnh giao thông tại thành phố New York

Thành phố New York là một trong những đô thị có mật độ giao thông cao, mạng lưới đường bộ phức tạp và nhu cầu di chuyển lớn. Trong môi trường đô thị như vậy, tình trạng giao thông có thể thay đổi nhanh theo thời điểm trong ngày, khu vực, hướng di chuyển và điều kiện vận hành thực tế.

Dữ liệu giao thông tại NYC được công bố thông qua các nguồn của NYC DOT, trong đó dữ liệu tốc độ giao thông được cung cấp dưới dạng các bản ghi theo từng đoạn đường. NYC Open Data mô tả mỗi dòng trong bộ dữ liệu DOT Traffic Speeds là tốc độ trung bình của phương tiện di chuyển giữa hai điểm đầu cuối, với các trường như speed, travel_time, data_as_of, link_id, borough và link_name. Ngoài ra, NYC DOT cũng cung cấp các nguồn dữ liệu, dashboard và dữ liệu mở phục vụ khai thác thông tin giao thông của thành phố.

Từ đặc điểm trên, bài toán dự đoán ùn tắc giao thông tại NYC có thể được xem là một bài toán phù hợp với định hướng xử lý dữ liệu lớn. Dữ liệu có yếu tố thời gian rõ rệt, được cập nhật liên tục, có nhiều đoạn đường khác nhau và cần được xử lý qua nhiều bước trước khi có thể sử dụng cho mô hình dự đoán.

2.1.2. Mục tiêu dự đoán mức độ tắc nghẽn

Mục tiêu của bài toán là dự đoán mức độ tắc nghẽn giao thông trong 15 phút tiếp theo cho từng đoạn đường tại NYC. Thay vì chỉ mô tả trạng thái giao thông hiện tại, hệ thống hướng đến việc dự đoán trạng thái trong tương lai gần, từ đó giúp người dùng quan sát được xu hướng giao thông sắp xảy ra trên từng đoạn đường.

Về bản chất, đây là bài toán phân loại đa lớp. Với mỗi bản ghi giao thông tại một thời điểm nhất định, hệ thống sử dụng thông tin tốc độ hiện tại, đặc trưng thời gian, vị trí đoạn đường và các đặc trưng trong quá khứ gần để dự đoán trạng thái giao thông sau 15 phút. Kết quả dự đoán được chia thành ba mức: Thông thoáng, Tắc nghẽn và Nghiêm trọng.

2.1.3. Đơn vị dự đoán

Đơn vị dự đoán trong đồ án là từng đoạn đường, được định danh bằng trường link_id. Mỗi link_id đại diện cho một đoạn đường có hướng di chuyển cụ thể. Do đó,

một tuyến đường hai chiều có thể được biểu diễn bằng hai link_id khác nhau, tương ứng với hai hướng di chuyển riêng biệt. Cách định danh này giúp hệ thống phân tích giao thông ở mức chi tiết hơn so với việc chỉ xét theo tên đường hoặc theo quận.

2.1.4. Lý do chọn mốc dự đoán 15 phút

Mốc dự đoán 15 phút được lựa chọn vì phù hợp với đặc điểm của bài toán dự đoán giao thông ngắn hạn. Trong nghiên cứu về short-term traffic forecasting, dự đoán ngắn hạn thường được hiểu là dự đoán trạng thái giao thông từ vài giây đến vài giờ trong tương lai, dựa trên thông tin giao thông hiện tại và quá khứ gần [1]. Một số nghiên cứu về dự đoán giao thông cũng sử dụng các khoảng dự đoán như 15, 30 và 60 phút, cho thấy mốc 15 phút là một khoảng thời gian thường gặp.

Đối với đồ án này, mốc 15 phút còn phù hợp với đặc điểm cập nhật của dữ liệu lịch sử. Theo nội dung nhóm tổng hợp, dữ liệu historical từ NYC Open Data khi ổn định thường có khoảng cách gần 5 phút giữa các bản ghi; do đó có thể sử dụng lead(..., 3) theo từng link_id để tạo nhãn dự đoán sau khoảng 15 phút. Cách xây dựng nhãn này giúp bài toán vừa phản ánh đúng bản chất dự đoán ngắn hạn, vừa tận dụng được cấu trúc thời gian sẵn có của dữ liệu.

Ngoài ra, mốc 15 phút có ý nghĩa thực tế trong bối cảnh giao thông đô thị. Nếu dự đoán quá gần, kết quả có thể chưa đủ giá trị để hỗ trợ ra quyết định. Ngược lại, nếu dự đoán quá xa, độ bất định của giao thông sẽ tăng lên do ảnh hưởng của nhiều yếu tố khó kiểm soát như sự kiện, tai nạn, thời tiết hoặc thay đổi dòng phương tiện. Vì vậy, mốc 15 phút được xem là lựa chọn cân bằng giữa tính ứng dụng và khả năng khai thác dữ liệu hiện có.

2.2. Cơ sở phân loại mức độ tắc nghẽn

2.2.1. Khái niệm free_flow_speed

free_flow_speed được hiểu là tốc độ tham chiếu của một đoạn đường trong điều kiện giao thông tương đối thông thoáng. Trong đồ án này, free_flow_speed không được gán cố định cho toàn bộ hệ thống, mà được tính riêng cho từng link_id. Cách làm này là cần thiết vì các đoạn đường tại NYC có đặc điểm khác nhau về loại đường, vị trí, giới

[1] Vlahogianni, E. I., Karlaftis, M. G., & Golias, J. C. (2014). *Short-term traffic forecasting: Where we are and where we're going*. Transportation Research Part C: Emerging Technologies, 43, 3-19.

hạn tốc độ và điều kiện vận hành; do đó việc dùng một ngưỡng tốc độ tuyệt đối chung cho mọi đoạn đường sẽ không phản ánh công bằng mức độ tắc nghẽn.

Nhóm sử dụng phân vị 85% (P_{85}) của tốc độ theo từng link_id trên tập Train để xác định free_flow_speed. Việc dùng phân vị 85% giúp chọn một mức tốc độ đại diện cho điều kiện vận hành tốt của đoạn đường, đồng thời tránh bị chi phối bởi một số giá trị cực đoan. free_flow_speed được tính bằng P85 của speed theo từng link_id, chỉ trên phần dữ liệu train, sau đó áp dụng nhất quán cho validation, test và luồng dự đoán.

Cách tính này cũng phù hợp với định hướng sử dụng tốc độ tham chiếu theo từng đoạn đường thay vì dùng tốc độ tuyệt đối. Trong các tài liệu giao thông, tốc độ và mức phục vụ của đường thường được xem xét trong mối quan hệ với điều kiện vận hành của từng loại đường hoặc từng đoạn đường. Tài liệu Highway Design Manual của NYSDOT có mục riêng về Speed Studies, Highway Capacity and Level of Service, đồng thời đề cập đến việc phân tích năng lực và mức phục vụ giao thông theo phương pháp HCM trong các bối cảnh thiết kế và đánh giá đường bộ.

2.2.2. Công thức speed_ratio

Sau khi có free_flow_speed, hệ thống tính:

$$speed_ratio = SPEED / free_flow_speed$$

Trong đó,

SPEED là tốc độ tại thời điểm ghi nhận.

speed_ratio cho biết tốc độ hiện tại đạt bao nhiêu so với tốc độ tự do của chính đoạn đường đó. Nếu speed_ratio cao, giao thông tương đối thông thoáng; nếu thấp, đoạn đường có dấu hiệu tắc nghẽn.

2.2.3. Quy tắc phân loại 3 mức tắc nghẽn

Cơ sở ngưỡng phân loại được tham chiếu từ Highway Capacity Manual 2010. Trong HCM 2010, đối với Urban Street Segments, mức phục vụ LOS được xác định theo Travel Speed as a Percentage of Base Free-Flow Speed. Cụ thể, Exhibit 16-4 chia mức phục vụ thành các nhóm A đến F: LOS A khi tốc độ lớn hơn 85% tốc độ tự do, LOS B khi tốc độ nằm trong khoảng trên 67% đến 85%, LOS C khi tốc độ nằm trong khoảng trên 50% đến 67%, LOS D khi tốc độ nằm trong khoảng trên 40% đến 50%, LOS E khi

tốc độ nằm trong khoảng trên 30% đến 40%, và LOS F khi tốc độ nhỏ hơn hoặc bằng 30% tốc độ tự do[2].

Dựa trên các ngưỡng này, đồ án rút gọn sáu mức LOS A/B/C/D/E/F thành ba lớp tắc nghẽn để phù hợp với bài toán phân loại bằng mô hình học máy. Cách rút gọn này cũng tương đồng với hướng tiếp cận của nghiên cứu MTP: Exploring Multimodal Urban Traffic Profiling with Modality Augmentation and Spectrum Fusion, trong đó tác giả hợp nhất sáu mức LOS tiêu chuẩn thành ba nhóm Low Congestion, Moderate Congestion và High Congestion nhằm thích nghi với bài toán phân loại trong học máy. Nghiên cứu này cũng sử dụng phương pháp Percentage of Free-Flow Speed để xây dựng nhãn giao thông, bảo đảm nhãn phân loại có khả năng diễn giải và phù hợp với nguyên lý kỹ thuật giao thông[3].

Từ cơ sở trên, đồ án xây dựng quy tắc phân loại current_congestion thành ba lớp như sau:

Class	Điều kiện theo speed_ratio	HCM 2010 LOS
0 - Thông thoáng	speed_ratio ≥ 0.67	A/B
1 - Tắc nghẽn	0.40 ≤ speed_ratio < 0.67	C/D
2 - Nghiêm trọng	speed_ratio < 0.40	E/F

Với cách phân loại này, ranh giới 0.67 được dùng để tách nhóm Thông thoáng khỏi các trạng thái bắt đầu có dấu hiệu hạn chế dòng phương tiện, tương ứng với ranh giới giữa LOS B và LOS C trong HCM 2010. Ranh giới 0.40 được dùng để tách nhóm Tắc nghẽn khỏi nhóm Nghiêm trọng, tương ứng với ranh giới giữa LOS D và LOS E. Như vậy, ba lớp đầu ra của đồ án vẫn giữ được ý nghĩa kỹ thuật từ hệ thống LOS, đồng thời được đơn giản hóa để phù hợp với mục tiêu huấn luyện mô hình RandomForestClassifier và trực quan hóa kết quả trên dashboard.

[2] Transportation Research Board. (2010). *Highway Capacity Manual 2010 (HCM2010)*. Transportation Research Board of the National Academies. Retrieved from https://sitesreservoirproject.riptideweb.com/references/REF23/Volume%201/Ch18_Nav_Trans_Traffic/Transportation%20Research%20Board%202010_HCM_2010_References.pdf

[3] Xiang, H., Wang, P., Xu, X., Yi, K., Zhang, X., Sheng, Q. Z., Beheshti, A., & Fan, W. (2025). *MTP: Exploring Multimodal Urban Traffic Profiling with Modality Augmentation and Spectrum Fusion*. arXiv preprint arXiv:2511.10218.

2.2.4. Ý nghĩa các lớp: Thông thoáng, Tắc nghẽn, Nghiêm trọng

Lớp 0 - Thông thoáng biểu diễn trạng thái giao thông ổn định. Khi speed_ratio >= 0.67, tốc độ hiện tại vẫn đạt ít nhất 67% so với tốc độ tự do của đoạn đường. Điều này cho thấy dòng phương tiện di chuyển tương đối thuận lợi, ít bị cản trở.

Lớp 1 - Tắc nghẽn biểu diễn trạng thái giao thông bắt đầu bị hạn chế. Khi speed_ratio nằm trong khoảng từ 0.40 đến dưới 0.67, tốc độ phương tiện đã giảm đáng kể so với điều kiện thông thoáng. Đây là trạng thái cho thấy đoạn đường có dấu hiệu ùn tắc hoặc lưu thông chậm.

Lớp 2 - Nghiêm trọng biểu diễn trạng thái tắc nghẽn nặng. Khi speed_ratio < 0.40, tốc độ hiện tại rất thấp so với tốc độ tự do, phản ánh tình trạng di chuyển chậm, gián đoạn hoặc gần với trạng thái stop-and-go.

2.3. Nguồn dữ liệu sử dụng

Đề tài sử dụng hai nguồn dữ liệu chính từ NYC DOT: dữ liệu lịch sử để huấn luyện mô hình và dữ liệu gần thời gian thực để dự đoán.

Do hạn chế về vùng sử dụng nên khi truy cập vào 2 nguồn dữ liệu này, nhóm phải bật VPN sang các vùng thuộc Mỹ mới có thể lấy data và sử dụng.

2.3.1. Dữ liệu lịch sử: NYC Open Data - DOT Traffic Speeds NBE

Dữ liệu lịch sử được lấy từ bộ dữ liệu NYC Open Data - DOT Traffic Speeds NBE. Đây là nguồn dữ liệu lưu trữ các bản ghi tốc độ giao thông trong quá khứ, được sử dụng để xây dựng tập huấn luyện, tập validation và tập test cho mô hình.

URL dùng để call api (bật VPN Mỹ): https://data.cityofnewyork.us/resource/i4gi-tjb9.json

Link truy cập (bật VPN Mỹ): https://data.cityofnewyork.us/Transportation/DOT-Traffic-Speeds-NBE/i4gi-tjb9

2.3.2. Dữ liệu gần thời gian thực: NYC DOT Real-Time Traffic Speed Feed

Dữ liệu gần thời gian thực được lấy từ NYC DOT Real-Time Traffic Speed Feed. Nguồn này cung cấp thông tin tốc độ giao thông mới nhất theo từng link_id, phục vụ cho giai đoạn dự đoán khi hệ thống đã được triển khai.

URL: https://linkdata.nyctmc.org/data/LinkSpeedQuery.txt

Nguồn chính thức:

https://www.nyc.gov/html/dot/html/about/datafeeds.shtml#realtime

(mục "Traffic Speed (txt)")

Dữ liệu real-time có cấu trúc tương tự dữ liệu lịch sử, gồm các trường quan trọng như linkId, Speed, DataAsOf, Borough và linkName. Trong hệ thống, Job A sẽ thu thập dữ liệu real-time, tính speed_ratio và current_congestion, sau đó đẩy snapshot lên Kafka topic nyc-traffic-snapshots. Job B đọc dữ liệu từ Kafka, tạo đủ đặc trưng cần thiết và đưa vào mô hình để dự đoán mức độ tắc nghẽn trong 15 phút tiếp theo.

2.4. Mô tả cấu trúc dữ liệu

2.4.1. Các cột dữ liệu ban đầu

Dữ liệu lịch sử từ NYC Open Data - DOT Traffic Speeds NBE gồm 13 cột: id, speed, travel_time, status, data_as_of, link_id, link_points, encoded_poly_line, encoded_poly_line_lvls, owner, transcom_id, borough và link_name.

Dữ liệu real-time từ NYC DOT cũng có cấu trúc tương tự, gồm các trường như id, Speed, TravelTime, Status, DataAsOf, linkId, linkPoints, EncodedPolyLine, EncodedPolyLineLvls, Owner, Transcom_id, Borough và linkName.

2.4.2. Các cột được sử dụng trong đề tài

Các cột chính được sử dụng gồm link_id, speed, data_as_of, borough và link_name.

Cột link_id dùng để định danh đoạn đường, nhóm dữ liệu, tính lag/lead và dự đoán theo từng đoạn đường. Cột speed dùng để tính free_flow_speed, speed_ratio và trạng thái tắc nghẽn. Cột data_as_of dùng để sắp xếp dữ liệu theo thời gian, trích xuất đặc trưng thời gian và tạo các đặc trưng trễ. Cột borough được dùng làm đặc trưng đầu vào cho mô hình và phục vụ phân tích theo quận. Cột link_name chỉ dùng để hiển thị kết quả trên dashboard.

2.4.3. Các cột bị loại bỏ

Các cột status, travel_time, id, transcom_id, owner, encoded_poly_line, encoded_poly_line_lvls và link_points không được đưa vào mô hình dự đoán.

Cột status không mang giá trị phân tích hữu ích. Cột travel_time bị loại vì có quan hệ phụ thuộc với tốc độ và chiều dài đoạn đường, trong khi speed đã đủ để xây dựng các chỉ số cần thiết. Các cột còn lại chủ yếu là thông tin định danh, mô tả hoặc phục vụ hệ thống nguồn nên không dùng làm đặc trưng huấn luyện. Riêng link_points

có thể được dùng trong bước thiết lập ban đầu để trích xuất tọa độ cho link_coordinates.parquet, phục vụ hiển thị bản đồ trên Power BI.

2.4.4. Đặc điểm cập nhật của dữ liệu

Dữ liệu lịch sử trên NYC Open Data được cập nhật thông qua quá trình ETL định kỳ. Khi ổn định, các bản ghi thường cách nhau khoảng 5 phút, nhưng vẫn có thể xuất hiện khoảng trống 10 đến 60 phút hoặc lâu hơn.

Dữ liệu real-time từ NYC DOT có tần suất cập nhật nhanh hơn, thường dưới 2 phút trong điều kiện bình thường. Vì vậy, dữ liệu lịch sử phù hợp để huấn luyện mô hình, còn dữ liệu real-time phù hợp để phục vụ dự đoán khi hệ thống vận hành.

2.4.5. Vấn đề độ trễ, khoảng trống thời gian và dữ liệu cảm biến lỗi

Dữ liệu giao thông thực tế có thể gặp các vấn đề như độ trễ cập nhật, thiếu bản ghi theo thời gian hoặc lỗi cảm biến. Những vấn đề này ảnh hưởng trực tiếp đến việc tính lag, lead và nhãn dự đoán.

Để đảm bảo chất lượng dữ liệu, đề tài kiểm tra khoảng cách thời gian thực tế khi tính các đặc trưng trễ 15, 30, 45, 60 phút và nhãn sau 15 phút. Các dòng có khoảng cách thời gian lệch quá mức cho phép sẽ bị loại bỏ.

Ngoài ra, hệ thống chỉ giữ các bản ghi có speed > 0 và speed < 100 để loại bỏ dữ liệu cảm biến lỗi. Đồng thời, chỉ các link_id đang hoạt động trong active_link_ids.json mới được giữ lại nhằm bảo đảm dữ liệu huấn luyện phù hợp.

2.5. Phạm vi dữ liệu huấn luyện

Phạm vi dữ liệu huấn luyện được lựa chọn từ tháng 01/2024 đến tháng 06/2026. Đây là giai đoạn đủ gần với thời điểm triển khai hệ thống, đồng thời đủ dài để mô hình học được các quy luật giao thông theo thời gian như giờ trong ngày, ngày trong tuần, tháng và sự thay đổi theo mùa.

2.5.1. Lý do chọn dữ liệu từ 01/2024 đến 2026

Đề tài sử dụng dữ liệu từ 01/2024 đến 06/2026 vì đây là giai đoạn dữ liệu còn mới, phản ánh tốt hơn tình trạng giao thông hiện tại tại thành phố New York. Nếu sử dụng dữ liệu quá cũ, mô hình có thể học các quy luật không còn phù hợp do thay đổi về hạ tầng, luồng di chuyển, chính sách giao thông hoặc hành vi người dân.

Ngoài ra, khoảng thời gian này đủ dài để xây dựng tập Train, Validation và Test theo thứ tự thời gian. Cụ thể, dữ liệu từ 01/2024 đến 09/2025 được dùng để huấn luyện,

dữ liệu từ 10/2025 đến 12/2025 dùng để validation, và dữ liệu từ 01/2026 đến 06/2026 dùng để đánh giá mô hình.

2.5.2. Lý do chỉ sử dụng các link_id đang hoạt động

Hệ thống chỉ sử dụng các link_id đang hoạt động trong nguồn dữ liệu real-time để bảo đảm dữ liệu huấn luyện phù hợp với môi trường dự đoán thực tế. Một số đoạn đường trong các năm trước có thể đã bị thay đổi, ngừng hoạt động hoặc không còn xuất hiện trong feed real-time.

Việc lọc theo active_link_ids.json giúp loại bỏ các đoạn đường không còn phù hợp, đồng thời bảo đảm mô hình chỉ học trên những link_id có khả năng xuất hiện khi hệ thống vận hành. Điều này giúp giảm nhiễu dữ liệu và tránh việc mô hình học các mẫu giao thông của những đoạn đường không còn sử dụng.

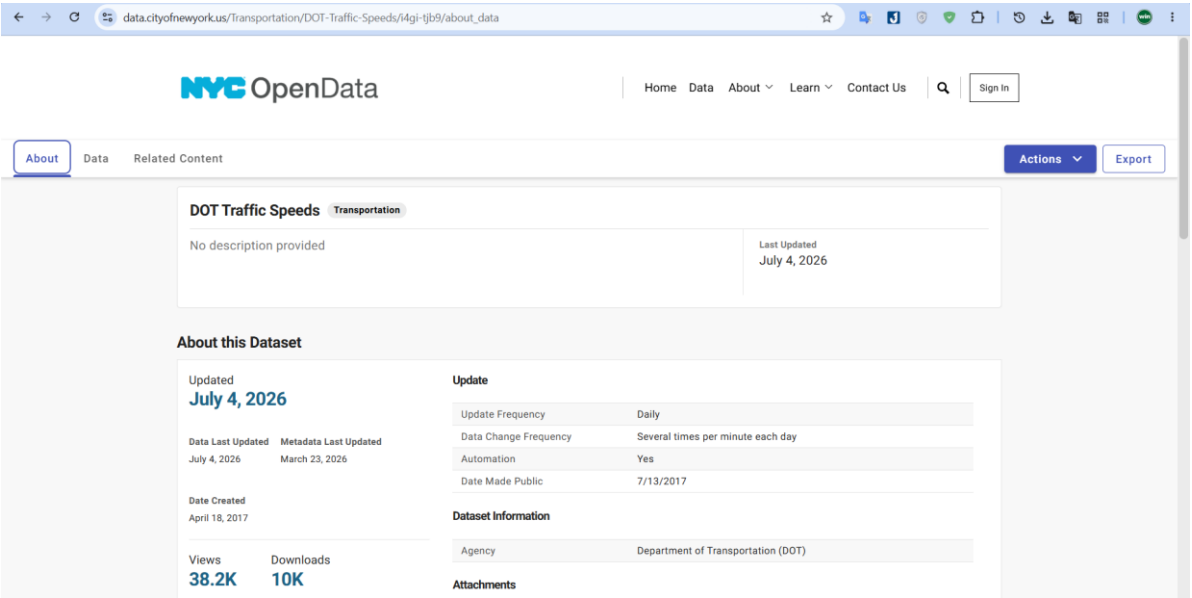
CHƯƠNG 3. THIẾT KẾ, THIẾT LẬP VÀ TRIỂN KHAI KIẾN TRÚC DATA LAKEHOUSE

3.1. Yêu cầu thiết kế hệ thống

3.1.1. Yêu cầu thu thập dữ liệu

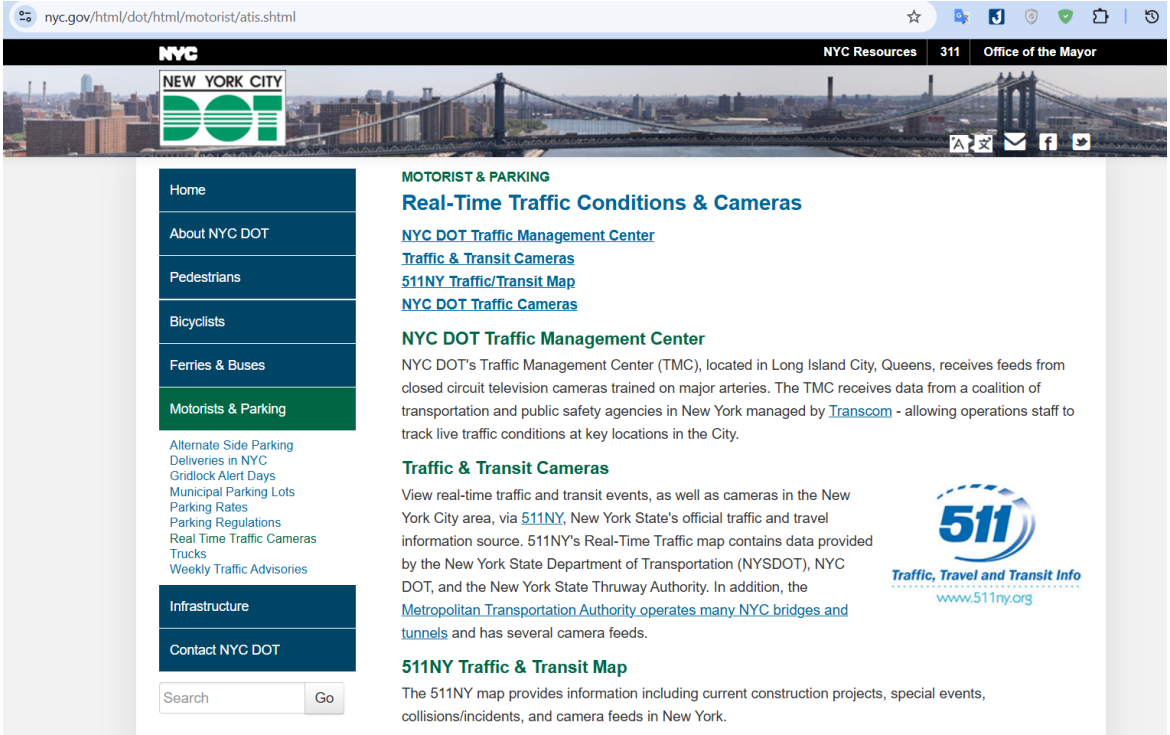
Hệ thống cần thu thập dữ liệu từ hai nguồn chính:

Nguồn thứ nhất là dữ liệu lịch sử từ NYC Open Data - DOT Traffic Speeds NBE, dùng để xây dựng tập huấn luyện, validation và test cho mô hình.



Hình 3.1: Trang web NYC Open Data – DOT Traffic Speeds NBE

Nguồn thứ hai là NYC DOT Real-Time Traffic Speed Feed, dùng để lấy dữ liệu gần thời gian thực phục vụ dự đoán.



Hình 3.2: Trang NYC DOT Real-Time Traffic Speed Feed

Hai bộ dữ liệu này đều thuộc về tổ chức NYC DOT (New York City DOT) vì vậy khi cần truy cập hoặc tải dataset về thì các công cụ VPN đã được sử dụng nhằm chuyển vùng dữ liệu sang bên Mỹ.

Quá trình thu thập dữ liệu cần bảo đảm lấy được các trường quan trọng và đồng nhất, đặc biệt với dữ liệu dùng để huấn luyện model và dự đoán như link_id, speed, data_as_of, borough và link_name. Khi thực hiện xử lý dữ liệu real-time, Job A phải được thiết kế để gọi feed định kỳ, lọc dữ liệu lỗi, tính các chỉ số cần thiết và đẩy snapshot hợp lệ vào Kafka sao cho khớp với lại thông tin mà Job B nhận được dùng để dự đoán.

3.1.2. Yêu cầu lưu trữ dữ liệu lớn

Hệ thống cần lưu trữ dữ liệu trên Azure Blob Storage theo kiến trúc Medallion, bảo đảm tách biệt dữ liệu thô, dữ liệu đã làm sạch, dữ liệu đặc trưng và kết quả dự đoán.

3.1.3. Yêu cầu xử lý dữ liệu bằng Apache Spark

Apache Spark được sử dụng để xử lý dữ liệu lớn trong các bước chuyển đổi từ Bronze sang Silver và từ Silver sang Gold. Spark cần thực hiện các thao tác như đọc dữ liệu từ Azure Blob Storage, lọc dữ liệu lỗi, chuẩn hóa kiểu dữ liệu, chuyển đổi múi giờ, tạo đặc trưng thời gian và ghi dữ liệu ở định dạng Parquet.

Trong bước xử lý Gold, Spark cần tính free_flow_speed, speed_ratio, current_congestion, các đặc trưng trễ 15, 30, 45, 60 phút, xu hướng tắc nghẽn và nhãn future_congestion_15min. Ngoài ra, hệ thống cần kiểm tra khoảng cách thời gian thực tế giữa các bản ghi để loại bỏ các dòng dữ liệu không hợp lệ.

3.1.4. Yêu cầu huấn luyện mô hình dự đoán

Hệ thống cần huấn luyện mô hình dự đoán mức độ tắc nghẽn giao thông sau 15 phút cho từng link_id. Mô hình được sử dụng là RandomForestClassifier trong Spark MLlib.

Dữ liệu huấn luyện cần được chia theo trục thời gian thay vì chia ngẫu nhiên để tránh rò rỉ dữ liệu và phản ánh đúng bối cảnh dự đoán trên dữ liệu tương lai.

3.1.5. Yêu cầu dự đoán gần thời gian thực

Hệ thống cần hỗ trợ dự đoán gần thời gian thực bằng cách tiếp nhận snapshot giao thông mới, tạo đủ đặc trưng đầu vào và ghi kết quả dự đoán vào tầng Gold.

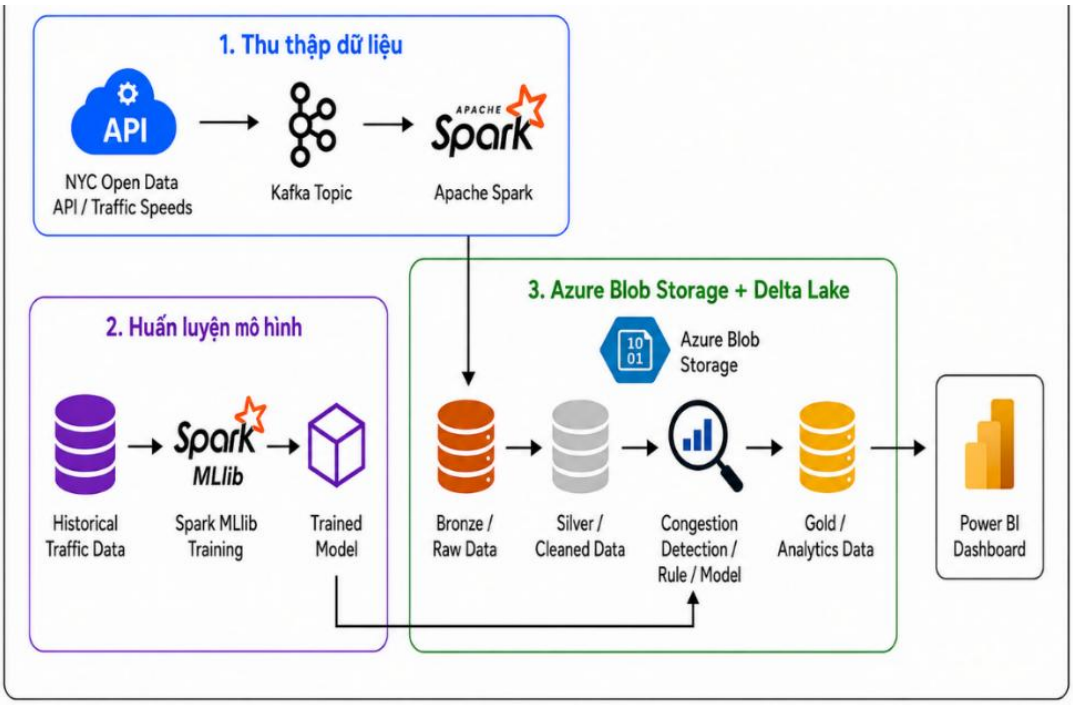
3.1.6. Yêu cầu trực quan hóa kết quả

Hệ thống cần trực quan hóa kết quả dự đoán trên Power BI. Dữ liệu đầu vào cho dashboard được lấy từ tầng Gold trên Azure Blob Storage.

Dashboard cần hỗ trợ quan sát tình trạng giao thông theo đoạn đường, quận, thời gian và mức độ tắc nghẽn. Các thông tin chính cần hiển thị gồm link_id, borough, link_name, timestamp, target_time, predicted_congestion và confidence. Ngoài ra, dữ liệu tọa độ từ link_coordinates.parquet có thể được sử dụng để hiển thị các đoạn đường trên bản đồ, giúp người dùng dễ dàng nhận diện khu vực có nguy cơ ùn tắc.

3.2. Tổng quan kiến trúc hệ thống

Kiến trúc hệ thống được xây dựng theo mô hình Data Lakehouse Streaming, kết hợp giữa xử lý dữ liệu lịch sử và dữ liệu gần thời gian thực. Hệ thống sử dụng Azure Blob Storage làm nơi lưu trữ trung tâm, Apache Spark để xử lý dữ liệu và huấn luyện mô hình, Apache Kafka để truyền dữ liệu real-time, Spark MLlib để xây dựng mô hình dự đoán và Power BI để trực quan hóa kết quả.



Hình 3.3: Tổng quan kiến trúc hệ thống

3.2.1. Luồng dữ liệu lịch sử phục vụ huấn luyện

Luồng dữ liệu lịch sử bắt đầu từ nguồn NYC Open Data - DOT Traffic Speeds NBE. Dữ liệu được thu thập và lưu vào tầng Bronze trên Azure Blob Storage dưới dạng dữ liệu thô. Ở tầng này, hệ thống giữ lại toàn bộ dữ liệu gốc để phục vụ kiểm tra và xử lý lại khi cần.

Sau đó, Apache Spark xử lý dữ liệu từ Bronze sang Silver bằng cách làm sạch dữ liệu, lọc các bản ghi lỗi, chuẩn hóa kiểu dữ liệu, chuyển đổi múi giờ và tạo các đặc trưng thời gian như hour, day_of_week, month, is_weekend và is_holiday.

Từ tầng Silver, hệ thống tiếp tục xử lý sang tầng Gold để tạo dữ liệu sẵn sàng cho huấn luyện mô hình. Các bước chính gồm tính free_flow_speed, speed_ratio, current_congestion, các đặc trưng trễ, xu hướng tắc nghẽn và nhãn future_congestion_15min. Dữ liệu Gold này được dùng để huấn luyện mô hình RandomForestClassifier bằng Spark MLlib.

3.2.2. Luồng dữ liệu real-time phục vụ dự đoán

Luồng dữ liệu real-time được lấy từ NYC DOT Real-Time Traffic Speed Feed. Job A có nhiệm vụ gọi nguồn dữ liệu này định kỳ, lọc các bản ghi không hợp lệ, tính speed_ratio và current_congestion, sau đó đẩy snapshot hợp lệ vào Kafka topic nyc-traffic-snapshots.

Kafka đóng vai trò là hàng đợi trung gian, lưu các snapshot giao thông trong 90 phút gần nhất. Việc lưu dữ liệu trong khoảng thời gian này giúp Job B có đủ dữ liệu để tạo các đặc trưng trễ 15, 30, 45 và 60 phút.

Job B chạy định kỳ 5 phút/lần, đọc dữ liệu từ Kafka, lấy snapshot hiện tại và các snapshot quá khứ theo từng link_id. Nếu dữ liệu đủ điều kiện, Job B tạo 19 đặc trưng đầu vào, đưa vào mô hình RandomForest đã huấn luyện và sinh ra kết quả predicted_congestion cho 15 phút tiếp theo.

3.2.3. Luồng dữ liệu Gold phục vụ Power BI

Kết quả dự đoán từ Job B được ghi vào tầng Gold tại thư mục gold/predictions/ trên Azure Blob Storage. Dữ liệu đầu ra gồm các trường như link_id, borough, link_name, timestamp, target_time, predicted_congestion và confidence.

Power BI sử dụng dữ liệu ở tầng Gold để xây dựng dashboard trực quan hóa. Dashboard cho phép người dùng theo dõi tình trạng giao thông theo đoạn đường, theo quận, theo thời gian và theo mức độ tắc nghẽn. Ngoài ra, file link_coordinates.parquet có thể được sử dụng để hiển thị vị trí các đoạn đường trên bản đồ.

3.3. Ánh xạ 5 lớp Lakehouse vào đề tài

Trong đề tài, kiến trúc Data Lakehouse được ánh xạ thành 5 lớp chính: Ingestion, Storage, Metadata, Processing/API và Consumption. Mỗi lớp đảm nhiệm một vai trò riêng trong quá trình thu thập, lưu trữ, xử lý, huấn luyện, dự đoán và trực quan hóa dữ liệu giao thông.

3.3.1. Ingestion: NYC Open Data API, NYC DOT Feed và Apache Kafka

Lớp Ingestion chịu trách nhiệm đưa dữ liệu vào hệ thống. Dữ liệu lịch sử được thu thập từ NYC Open Data - DOT Traffic Speeds NBE để phục vụ huấn luyện mô hình. Dữ liệu gần thời gian thực được lấy từ NYC DOT Real-Time Traffic Speed Feed để phục vụ dự đoán.

Đối với luồng real-time, Job A gọi feed định kỳ, lọc dữ liệu lỗi, tính speed_ratio và current_congestion, sau đó đẩy snapshot hợp lệ vào Kafka topic nyc-traffic-snapshots. Kafka lưu các snapshot trong 90 phút gần nhất để Job B sử dụng khi tạo đặc trưng dự đoán.

3.3.2. Storage: Azure Blob Storage, Delta Lake và Parquet

Lớp Storage được triển khai trên Azure Blob Storage. Dữ liệu được tổ chức theo mô hình Medallion gồm Bronze, Silver và Gold.

Tầng Bronze lưu dữ liệu thô, tầng Silver lưu dữ liệu đã làm sạch và chuẩn hóa, còn tầng Gold lưu dữ liệu sẵn sàng cho huấn luyện mô hình và kết quả dự đoán. Dữ liệu được lưu chủ yếu dưới định dạng Parquet để hỗ trợ xử lý bằng Apache Spark.

3.3.3. Metadata: schema, partition, lookup table và model artifacts

Metadata và artifacts của hệ thống gồm schema, cấu trúc partition, các bảng lookup và mô hình đã huấn luyện phục vụ cho quá trình dự đoán.

3.3.4. Processing/API: Apache Spark, Spark MLlib, Job A và Job B

Lớp Processing/API gồm Apache Spark cho xử lý batch, Spark MLlib cho huấn luyện mô hình, Job A cho thu thập realtime và Job B cho dự đoán.

3.3.5. Consumption: Power BI Dashboard

Lớp Consumption phục vụ người dùng cuối thông qua Power BI Dashboard. Power BI đọc dữ liệu dự đoán từ tầng Gold, đặc biệt là thư mục gold/predictions/ trên Azure Blob Storage.

Dashboard hiển thị các thông tin như link_id, borough, link_name, timestamp, target_time, predicted_congestion và confidence. Người dùng có thể theo dõi tình trạng giao thông theo đoạn đường, theo quận, theo thời gian và theo mức độ tắc nghẽn.

3.4. Cấu trúc thư mục trên Azure Blob Storage

Toàn bộ dữ liệu và artifact của hệ thống được tổ chức trong container Azure Blob Storage có tên nyc-traffic-lakehouse. Cấu trúc thư mục được thiết kế theo đúng ba tầng Bronze, Silver và Gold, đồng thời tách riêng thư mục artifacts để lưu các thành phần hỗ trợ mô hình và quá trình vận hành.

nyc-traffic-lakehouse/

```
│
├── bronze/
│   └── dot_traffic_speeds/
│       ├── historical/
│       │   ├── year=2024/
│       │   │   ├── month=01/
│       │   │   │   └── raw_2024_01.parquet
│       │   │   ├── month=02/
```

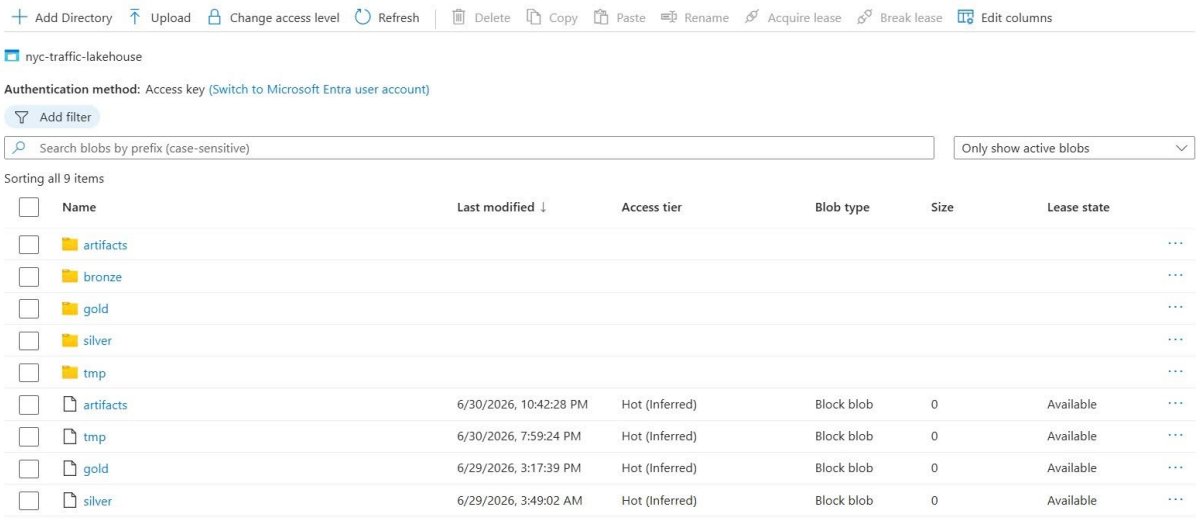
```
│   │   │   └── ...
│   │   ├── year=2025/
│   │   └── year=2026/
│   │
│   └── realtime/
│       ├── date=2026-06-24/
│       │   ├── hour=15/
│       │   │   └── part-0001.parquet
│       │   └── hour=16/
│       └── ...
│
├── silver/
│   └── traffic_cleaned/
│       ├── _delta_log/
│       ├── year=2024/
│       │   ├── month=01/
│       │   └── ...
│       ├── year=2025/
│       └── year=2026/
│
├── gold/
│   ├── training_features/
│   │   ├── _delta_log/
│   │   ├── year=2024/
│   │   │   ├── month=01/
│   │   │   └── ...
│   │   ├── year=2025/
│   │   └── year=2026/
│   │
│   └── predictions/
│       ├── _delta_log/
│       ├── date=2026-06-24/
│       │   ├── hour=15/
│       │   │   └── part-0001.parquet
│       │   └── hour=16/
│       └── ...
│
└── artifacts/
    ├── model/
    │   └── random_forest_model/
```

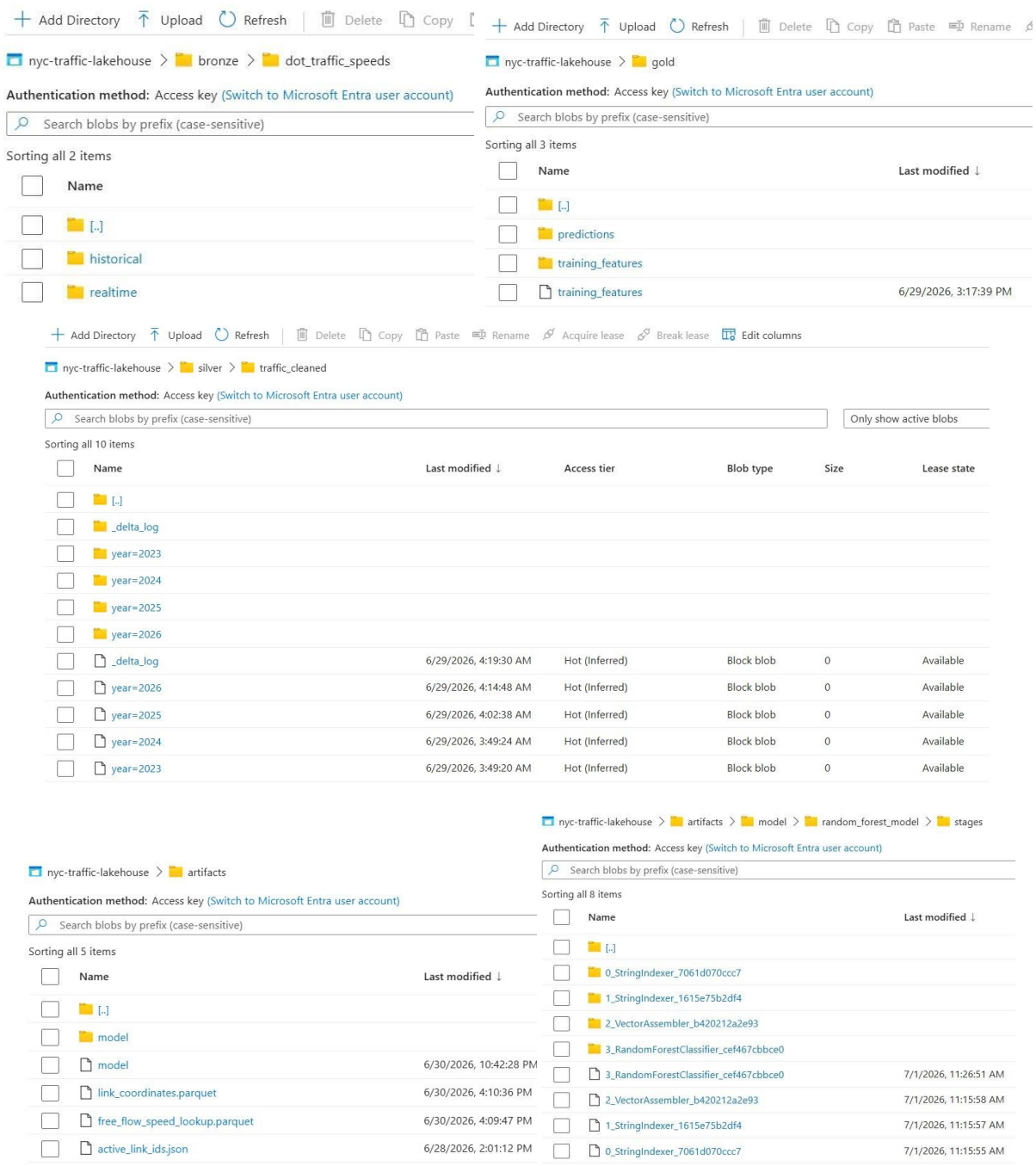
```
│       ├── metadata/
│       └── stages/
│
├── free_flow_speed_lookup.parquet
├── active_link_ids.json
└── link_coordinates.parquet
```

Trong cấu trúc trên, thư mục bronze/dot_traffic_speeds/historical lưu dữ liệu lịch sử theo phân vùng year và month, còn bronze/dot_traffic_speeds/realtime lưu dữ liệu gần thời gian thực theo date và hour.

Thư mục silver/traffic_cleaned lưu dữ liệu đã được làm sạch, chuẩn hóa và bổ sung đặc trưng thời gian. Đây là nguồn dữ liệu đầu vào cho quá trình xây dựng Gold Layer. Thư mục gold/training_features lưu tập dữ liệu đặc trưng dùng cho huấn luyện, kiểm định và kiểm tra mô hình. Thư mục gold/predictions lưu kết quả dự đoán gần thời gian thực, được Power BI sử dụng để trực quan hóa tình trạng giao thông.

Thư mục artifacts được đặt tách biệt khỏi ba tầng dữ liệu vì đây không phải là dữ liệu nghiệp vụ chính, mà là các thành phần hỗ trợ pipeline và mô hình. Trong đó, random_forest_model là mô hình đã huấn luyện; free_flow_speed_lookup.parquet lưu tốc độ tự do của từng link_id; active_link_ids.json lưu danh sách đoạn đường còn hoạt động; còn link_coordinates.parquet lưu tọa độ phục vụ hiển thị bản đồ trên dashboard.



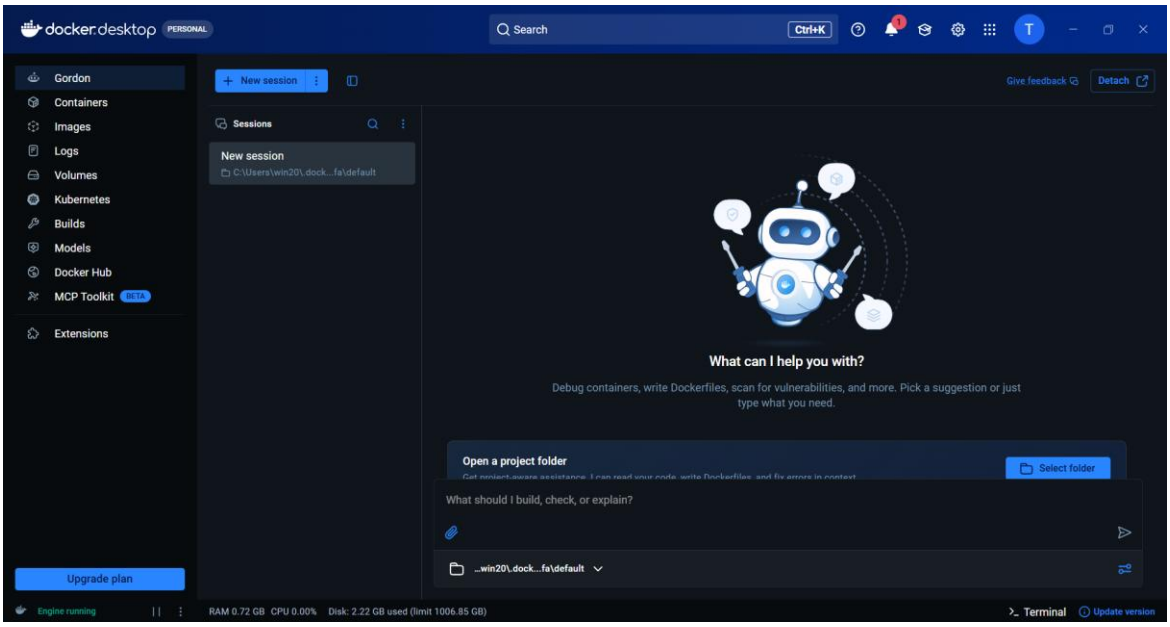


Hình 3.4: Cấu trúc thư mục lưu trữ Azure Blob Storage

3.5. Thiết lập môi trường triển khai

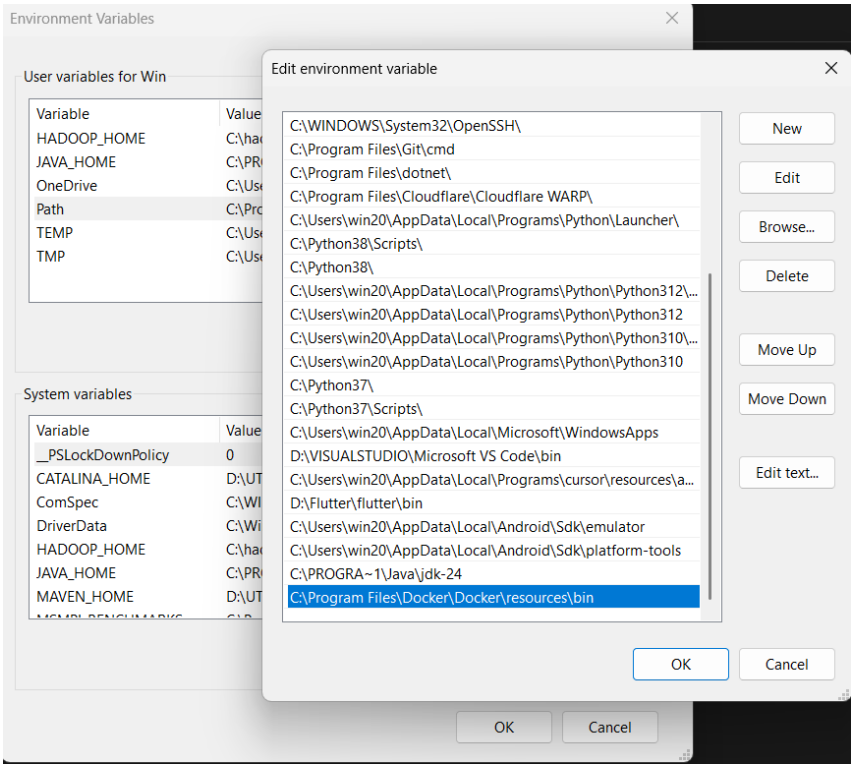
3.5.1. Cài đặt Docker Desktop

Bước đầu tiên trong quá trình thiết lập môi trường là cài đặt Docker Desktop và Git trên máy phát triển nhằm phục vụ cho mục đích đồng nhất môi trường làm việc. Docker Desktop được sử dụng để tạo, quản lý và vận hành các container của hệ thống. Đối với môi trường Windows, Docker Desktop cần được cấu hình sử dụng WSL2 backend để hỗ trợ tốt các container Linux.



Hình 3.5: Giao diện Docker Desktop

Tiếp theo là thêm vào Path của user và system



Hình 3.6: Biến môi trường

Kiểm tra Docker bằng các lệnh docker --version và docker compose version

```
Administrator: C:\WINDOWS\system32\cmd.exe
Microsoft Windows [Version 10.0.26200.8737]
(c) Microsoft Corporation. All rights reserved.

C:\Windows\System32>docker --version
Docker version 29.5.3, build d1c06ef

C:\Windows\System32>docker compose version
Docker Compose version v5.1.4

C:\Windows\System32>
```

Hình 3.7: Kiểm tra Docker version

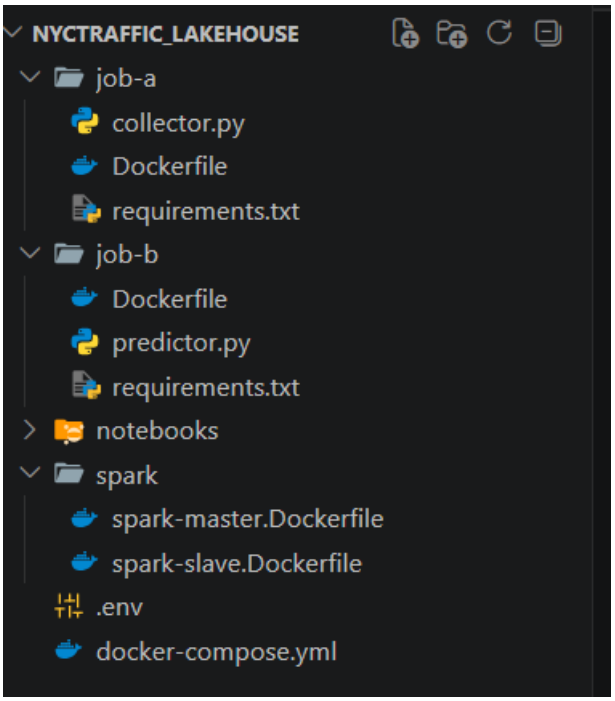
3.5.2. Tạo cấu trúc thư mục project

Cấu trúc thư mục project được tổ chức theo hướng tách riêng từng nhóm chức năng. Thư mục job-a/ chứa mã nguồn của collector, bao gồm file collector.py, Dockerfile và requirements.txt. Đây là thành phần chịu trách nhiệm gọi dữ liệu real-time từ NYC DOT, xử lý bước đầu và đẩy snapshot hợp lệ vào Kafka.

Thư mục job-b/ chứa mã nguồn của predictor, bao gồm file predictor.py, Dockerfile và requirements.txt. Thành phần này đọc dữ liệu từ Kafka, xây dựng đặc trưng đầu vào và thực hiện dự đoán bằng mô hình đã huấn luyện.

Thư mục notebooks/ được sử dụng để lưu các notebook phục vụ quá trình xử lý dữ liệu batch, bao gồm tạo danh sách link đang hoạt động, thu thập dữ liệu lịch sử, xử lý Bronze sang Silver, xử lý Silver sang Gold và huấn luyện mô hình.

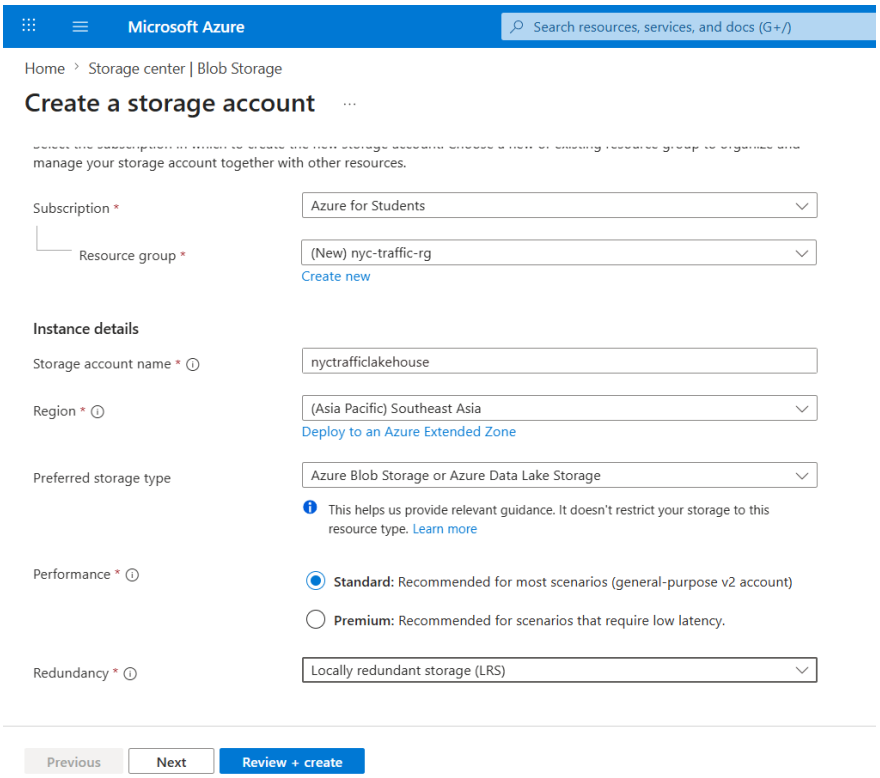
Thư mục spark/ chứa các Dockerfile tùy chỉnh cho Spark Master và Spark Worker, nhằm bảo đảm phiên bản Python trong Spark cluster tương thích với môi trường Jupyter. Ngoài ra, project còn có file .env để lưu biến môi trường và file docker-compose.yml để định nghĩa các service triển khai.

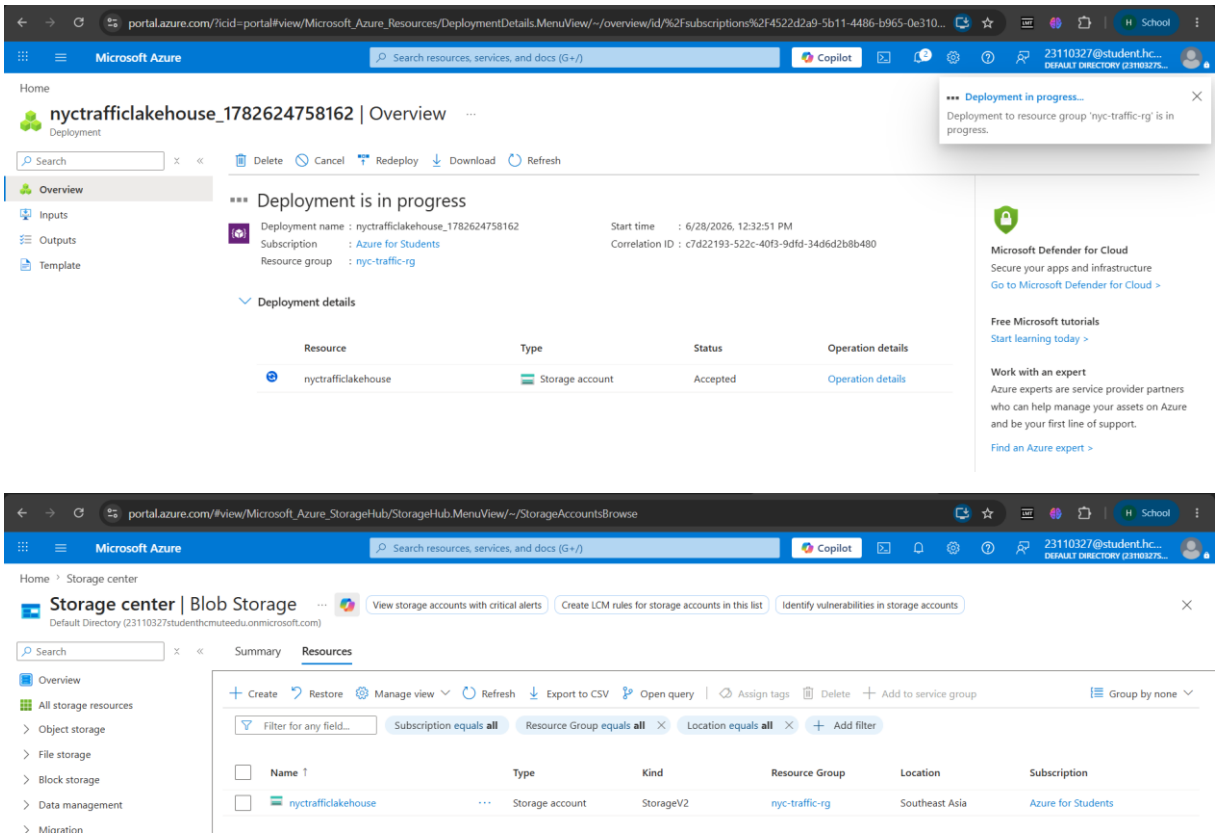


Hình 3.8: Cấu trúc thư mục project

3.5.3. Tạo Azure Blob Storage

Trên Azure Portal, nhóm tạo một Storage Account có tên nyctrafficlakehouse, thuộc resource group nyc-traffic-rg, đặt tại region Southeast Asia, sử dụng hiệu năng Standard và cơ chế dự phòng LRS. Cấu hình này phù hợp với phạm vi đồ án vì đáp ứng được nhu cầu lưu trữ dữ liệu lớn, đồng thời vẫn tối ưu chi phí triển khai.

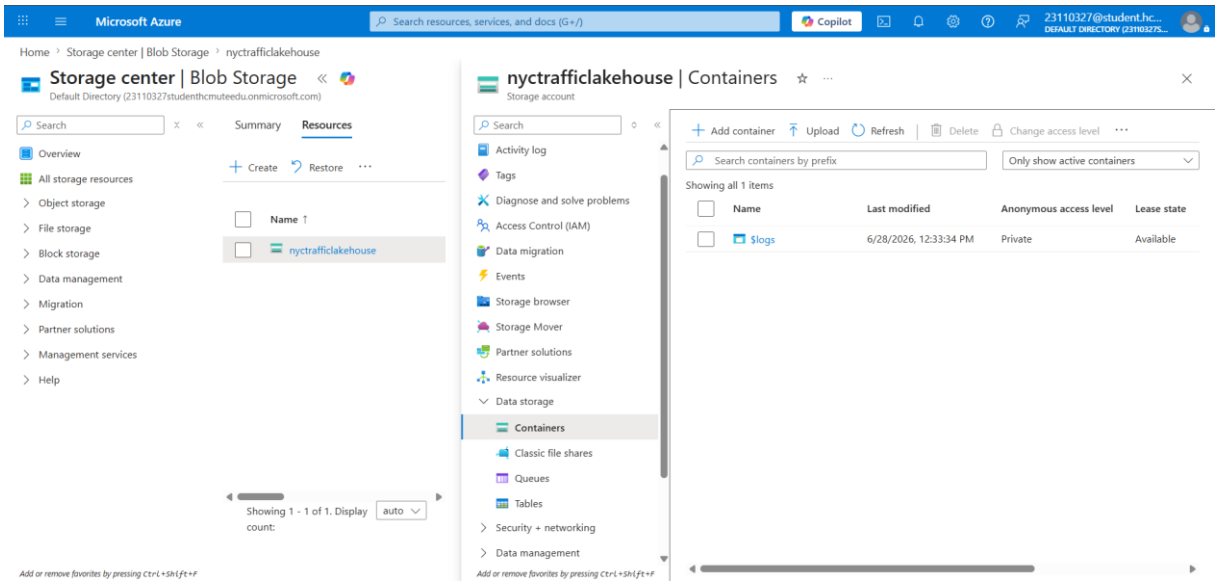


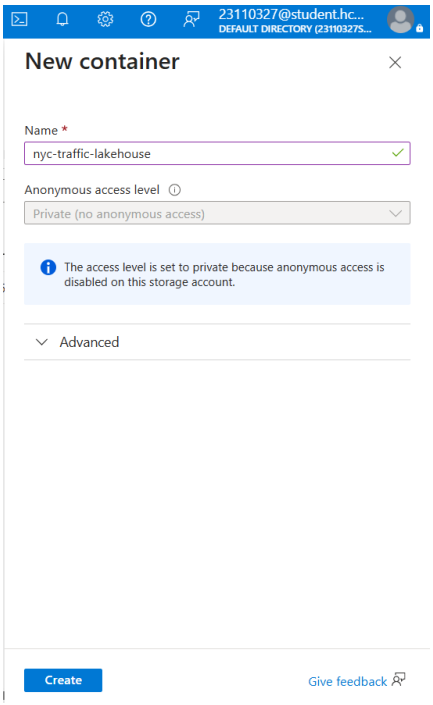


Hình 3.9: Tạo Azure Blob Storage

Sau khi tạo Storage Account, nhóm tạo container có tên nyc-traffic-lakehouse và đặt chế độ truy cập là Private.

Chọn Add container, điền thông tin và create.

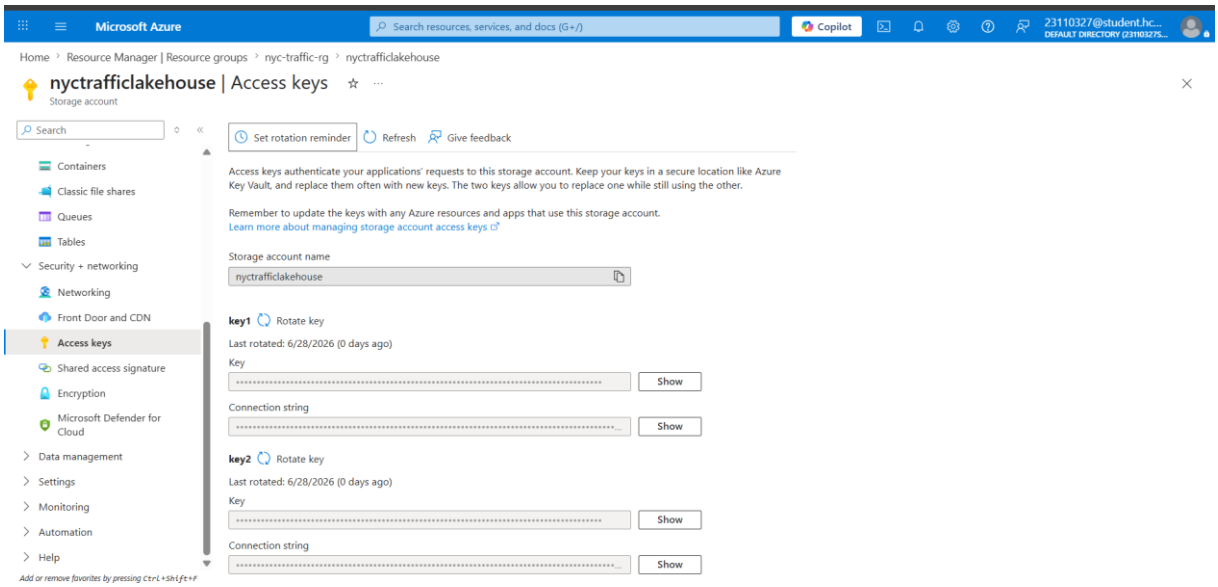




Hình 3.10: Tạo container

Container này đóng vai trò tương tự một bucket lưu trữ object, là nơi chứa toàn bộ dữ liệu của hệ thống, bao gồm dữ liệu Bronze, Silver, Gold và các artifact phục vụ mô hình. Việc đặt container ở chế độ Private giúp hạn chế truy cập công khai, bảo đảm dữ liệu chỉ được đọc ghi thông qua các thông tin xác thực hợp lệ.

Kiểm tra thông tin đăng nhập, vào Storage account nyctrafficlakehouse đến menu trái tìm Access keys (trong mục Security + networking).



Hình 3.11: Kiểm tra thông tin đăng nhập

3.5.4. Cấu hình file .env

File .env được sử dụng để quản lý các biến môi trường cần thiết cho quá trình triển khai. Trong các biến môi trường:

- AZURE_STORAGE_CONNECTION_STRING và AZURE_STORAGE_ACCOUNT_KEY được lấy từ mục Access keys của Storage Account
- AZURE_STORAGE_ACCOUNT_NAME là tên Storage Account đã tạo (nyctrafficlakehouse)
- AZURE_CONTAINER_NAME được đặt là nyc-traffic-lakehouse
- Biến AZURE_STORAGE_ACCOUNT_NAME đặc biệt cần thiết khi khai báo cấu hình Spark để truy cập Azure Blob Storage, do Spark yêu cầu key cấu hình phải gắn kèm tên account theo định dạng *spark.hadoop.fs.azure.account.key.<account_name>.blob.core.windows.net*
- Ngoài ra trong môi trường còn có thêm biến SOCRATA_APP_TOKEN: Đây là token của NYC Open Data nhằm cho việc gọi API lấy dữ liệu suôn sẻ và mượt mà hơn mà không bị gián đoạn so với dùng cách gọi thông thường.

Các biến này cho phép các container truy cập Azure Blob Storage. Ngoài nhóm biến Azure, file .env còn chứa các cấu hình phục vụ giao tiếp nội bộ giữa các service.

```
.env
.env
1   # Azure Blob Storage
2   AZURE_STORAGE_CONNECTION_STRING= <hidden>
3   AZURE_CONTAINER_NAME=nyc-traffic-lakehouse
4   AZURE_STORAGE_ACCOUNT_NAME=nyctrafficlakehous
5   AZURE_STORAGE_ACCOUNT_KEY= <hidden>
6
7   # Kafka
8   KAFKA_BOOTSTRAP_SERVERS=kafka:9092
9
10  # Spark
11  SPARK_MASTER=spark://spark-master:7077
12
13  # NYC OPEN DATA API
14  SOCRATA_APP_TOKEN=app_token
```

Hình 3.12: Cấu hình file .env

3.5.5. Thiết lập file docker-compose.yml

File docker-compose.yml định nghĩa các service chính của hệ thống, bao gồm kafka, spark-master, spark-slave, job-a và job-b. Tất cả các service được đặt trong cùng một Docker network để có thể giao tiếp với nhau thông qua tên container.

Kafka được triển khai bằng image Apache Kafka và chạy ở chế độ KRaft, do đó không cần sử dụng Zookeeper. Service Kafka được cấu hình với broker listener tại cổng 9092, đồng thời thiết lập retention log để phục vụ việc lưu giữ dữ liệu snapshot trong một khoảng thời gian nhất định. Đây là thành phần trung gian để Job A đẩy dữ liệu real-time và Job B đọc dữ liệu phục vụ dự đoán.

Spark được triển khai theo mô hình Spark Standalone với một Spark Master và một Spark Worker. Spark Master được cấu hình chạy lệnh /opt/spark/sbin/start-master.sh và mở cổng 7077 để các Spark application kết nối vào cluster. Đồng thời, Spark Master mở cổng 8080 để theo dõi trạng thái cụm thông qua Spark UI. Spark Worker được cấu hình chạy lệnh /opt/spark/sbin/start-worker.sh spark://spark-master:7077 để đăng ký vào Spark Master.

Trong quá trình thiết lập, nhóm tạo Dockerfile tùy chỉnh cho spark-master và spark-slave, cài đặt Python 3.13 và ánh xạ python3 về Python 3.13. Cách xử lý này giúp môi trường Spark đồng nhất với Jupyter Notebook và giảm lỗi khi chạy các pipeline PySpark.

.env | docker-compose.yml | spark-master.Dockerfile

spark > spark-master.Dockerfile > ...

```
FROM spark:4.0.0-scala2.13-java21-python3-ubuntu

USER root
RUN apt-get update && \
    apt-get install -y software-properties-common && \
    add-apt-repository ppa:deadsnakes/ppa -y && \
    apt-get update && \
    apt-get install -y python3.13 && \
    ln -sf /usr/bin/python3.13 /usr/bin/python3

USER spark
```

.env | docker-compose.yml | spark-master.Dockerfile

spark > spark-slave.Dockerfile > ...

```
FROM spark:4.0.0-scala2.13-java21-python3-ubuntu

USER root
RUN apt-get update && \
    apt-get install -y software-properties-common && \
    add-apt-repository ppa:deadsnakes/ppa -y && \
    apt-get update && \
    apt-get install -y python3.13 && \
    ln -sf /usr/bin/python3.13 /usr/bin/python3

USER spark
```

Hình 3.13: Dockerfile tùy chỉnh cho spark-master và spark-slave

Ngoài ra, trong docker-compose.yml, Job A được cấu hình phụ thuộc vào Kafka, còn Job B phụ thuộc vào Kafka và Spark Master. Điều này phản ánh đúng quan hệ vận hành của hệ thống: Job A cần Kafka để gửi message, trong khi Job B cần cả Kafka để đọc snapshot và Spark để thực hiện xử lý, dự đoán.

```yaml
networks:
  nyc-traffic-net:
    driver: bridge

Run All Services
services:

  Run Service
  kafka:
    image: apache/kafka:4.3.1
    container_name: kafka
    restart: unless-stopped
    environment:
      KAFKA_NODE_ID: 1
      KAFKA_PROCESS_ROLES: broker,controller
      KAFKA_LISTENERS: PLAINTEXT://:9092,CONTROLLER://:9093
      KAFKA_ADVERTISED_LISTENERS: PLAINTEXT://kafka:9092
      KAFKA_CONTROLLER_QUORUM_VOTERS: 1@kafka:9093
      KAFKA_LISTENER_SECURITY_PROTOCOL_MAP: PLAINTEXT:PLAINTEXT,CONTROLLER:PLAINTEXT
      KAFKA_CONTROLLER_LISTENER_NAMES: CONTROLLER
      KAFKA_OFFSETS_TOPIC_REPLICATION_FACTOR: 1
      KAFKA_LOG_RETENTION_MS: 5400000
      KAFKA_HEAP_OPTS: "-Xmx512m -Xms512m"
    networks:
      - nyc-traffic-net
    healthcheck:
      test: ["CMD-SHELL", "/opt/kafka/bin/kafka-broker-api-versions.sh --bootstrap-server localhost:9092 || exit 1"]
      interval: 15s
      timeout: 15s
      retries: 5
      start_period: 90s
    deploy:
      resources:
        limits:
          memory: 1g
```

```yaml
  Run Service
  spark-master:
    build:
      context: ./spark
      dockerfile: spark-master.Dockerfile
    container_name: spark-master
    restart: unless-stopped
    environment:
      SPARK_NO_DAEMONIZE: "true"
      SPARK_HOME: /opt/spark
    env_file: .env
    command: /opt/spark/sbin/start-master.sh
    ports:
      - "8080:8080"
      - "7077:7077"
    networks:
      - nyc-traffic-net
```

```yaml
  Run Service
  spark-slave:
    build:
      context: ./spark
      dockerfile: spark-slave.Dockerfile
    container_name: spark-slave
    restart: unless-stopped
    environment:
      SPARK_NO_DAEMONIZE: "true"
      SPARK_HOME: /opt/spark
    env_file: .env
    command: /opt/spark/sbin/start-worker.sh spark://spark-master:7077
    depends_on:
      - spark-master
    networks:
      - nyc-traffic-net
```

```yaml
  Run Service
  job-a:
    build: ./job-a
    container_name: job-a
    restart: unless-stopped
    env_file: .env
    depends_on:
      kafka:
        condition: service_healthy
    networks:
      - nyc-traffic-net
```

```yaml
  Run Service
  job-b:
    build: ./job-b
    container_name: job-b
    restart: unless-stopped
    env_file: .env
    depends_on:
      kafka:
        condition: service_healthy
      spark-master:
        condition: service_started
    networks:
      - nyc-traffic-net
```

Hình 3.14: Thiết lập file docker-compose.yml

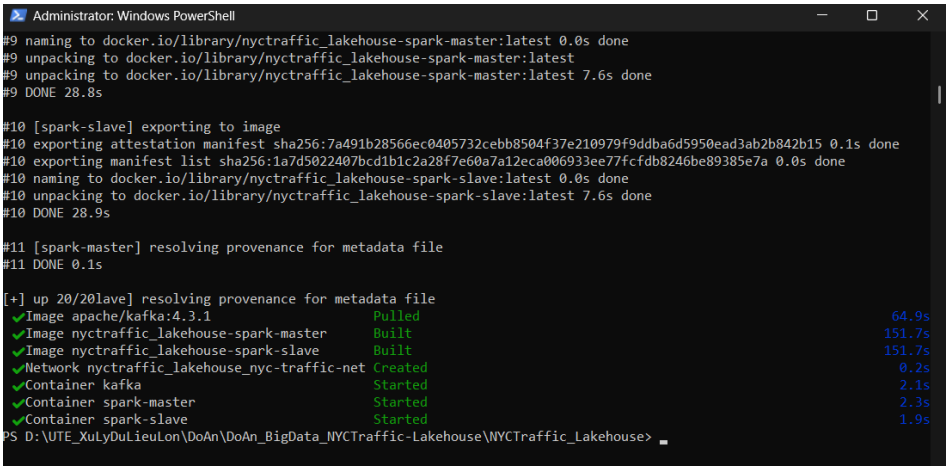
Tên network được khai báo trong docker-compose.yml là nyc-traffic-net. Tuy nhiên, khi Docker Compose khởi tạo network, tên thực tế sẽ được tự động gắn thêm tiền

tố là tên thư mục project (ví dụ nyctraffic_lakehouse_nyc-traffic-net), do Compose mặc định đặt tên network theo định dạng <project_name>_<network_name>. Đây là lý do tên network xuất hiện trong lệnh docker run của Jupyter Notebook ở mục 3.5.8 khác với tên khai báo gốc.

3.5.6. Khởi động Kafka, Spark Master và Spark Worker

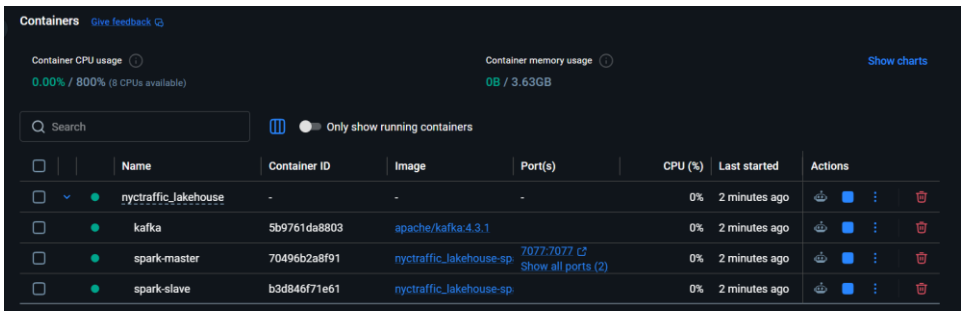
Sau khi hoàn tất cấu hình Docker Compose, nhóm tiến hành khởi động trước ba service nền tảng gồm Kafka, Spark Master và Spark Worker. Việc chạy riêng ba container này ở giai đoạn đầu nhằm kiểm tra hạ tầng cơ bản trước khi triển khai các job xử lý dữ liệu.

Lệnh khởi động *docker compose up kafka spark-master spark-slave -d* được thực hiện trong thư mục gốc của project bằng Docker Compose. Khi lệnh được thực thi, Docker sẽ tạo network, build các image Spark tùy chỉnh nếu cần và khởi động các container theo cấu hình đã khai báo. Kafka được khởi động như broker trung tâm, Spark Master khởi động dịch vụ điều phối tại spark://spark-master:7077, còn Spark Worker đăng ký tài nguyên xử lý với Spark Master.



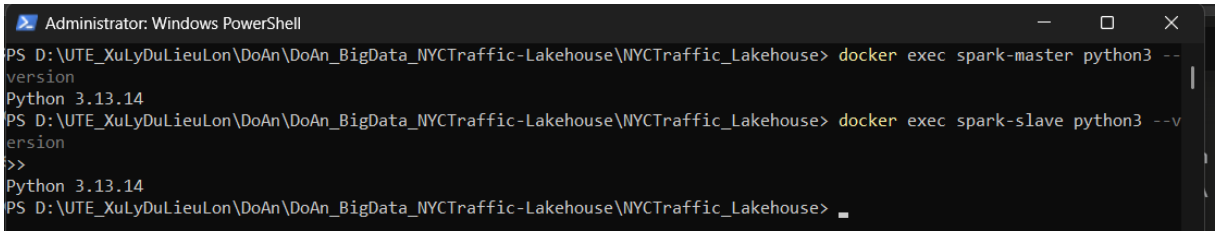
Hình 3.15: Khởi động Kafka, Spark Master và Spark Worker

Sau khi các container chạy, nhóm kiểm tra trạng thái trong Docker Desktop để bảo đảm Kafka, Spark Master và Spark Worker đều đang hoạt động.



Hình 3.16: Giao diện docker sau khi khởi động các container

Đồng thời, nhóm kiểm tra phiên bản Python trong hai container Spark bằng lệnh docker exec spark-master python3 --version và docker exec spark-slave python3 --version.

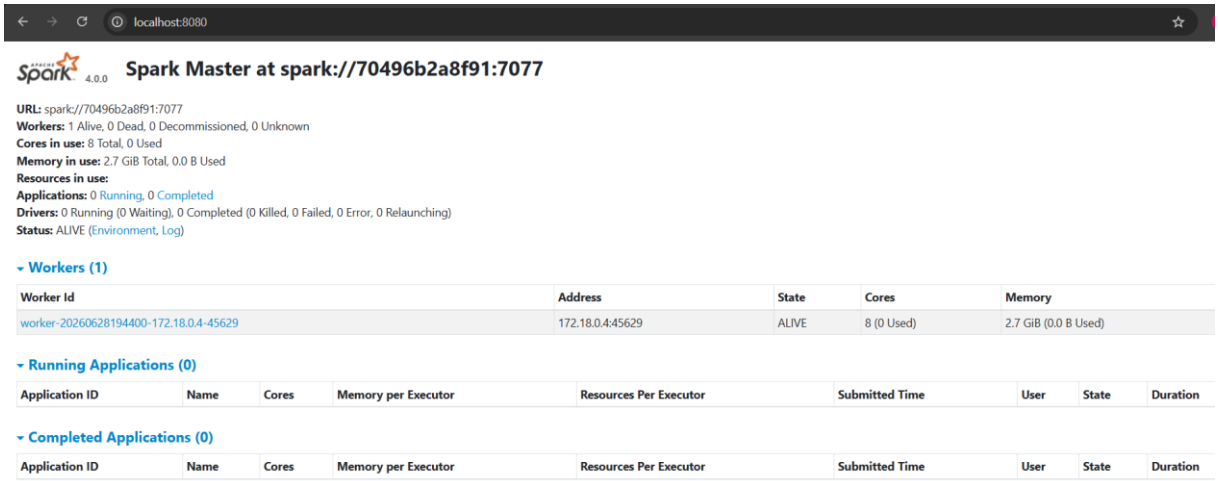


Hình 3.17: Kiểm tra phiên bản python của master và slave

3.5.7. Kiểm tra Spark UI

Sau khi khởi động các service nền tảng, nhóm kiểm tra Spark UI bằng cách truy cập http://localhost:8080 trên trình duyệt. Nếu Spark Master hoạt động đúng, giao diện sẽ hiển thị thông tin của cụm Spark, bao gồm địa chỉ master, trạng thái worker, số core, bộ nhớ và các ứng dụng đang chạy. Trạng thái mong đợi là Spark Master đang hoạt động và có một Spark Worker kết nối thành công.

Đối với các Spark job chạy trong quá trình xử lý dữ liệu, giao diện Spark application có thể được theo dõi thông qua các cổng như 4040, 4041 hoặc 4042, tùy theo số lượng ứng dụng Spark đang chạy. Việc mở các cổng này giúp nhóm quan sát quá trình thực thi job, kiểm tra stage, task và tài nguyên sử dụng khi chạy notebook hoặc job dự đoán.



Hình 3.18: Giao diện Spark UI

→ Thấy "Spark Master" và 1 worker connected

3.5.8. Khởi động Jupyter Notebook trong cùng Docker network

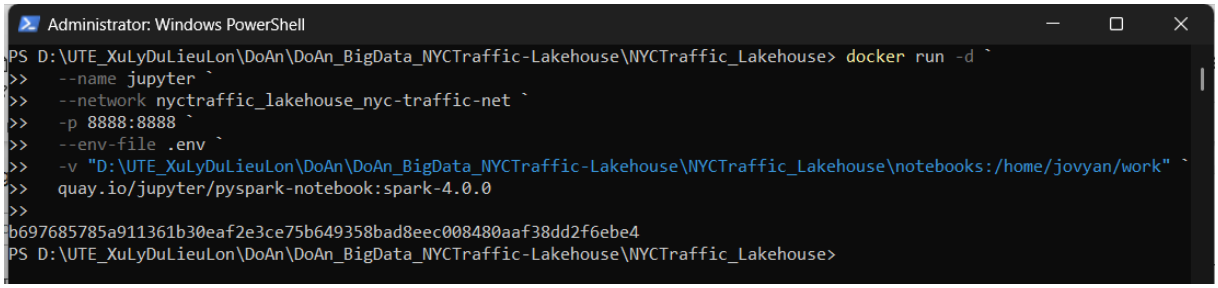
Jupyter Notebook được sử dụng làm môi trường thực thi các notebook xử lý dữ liệu trong giai đoạn xây dựng pipeline. Container này không được khai báo trong docker-compose.yml vì chỉ phục vụ giai đoạn xây dựng pipeline một lần khi setup, không phải service chạy thường trực như 5 container còn lại.

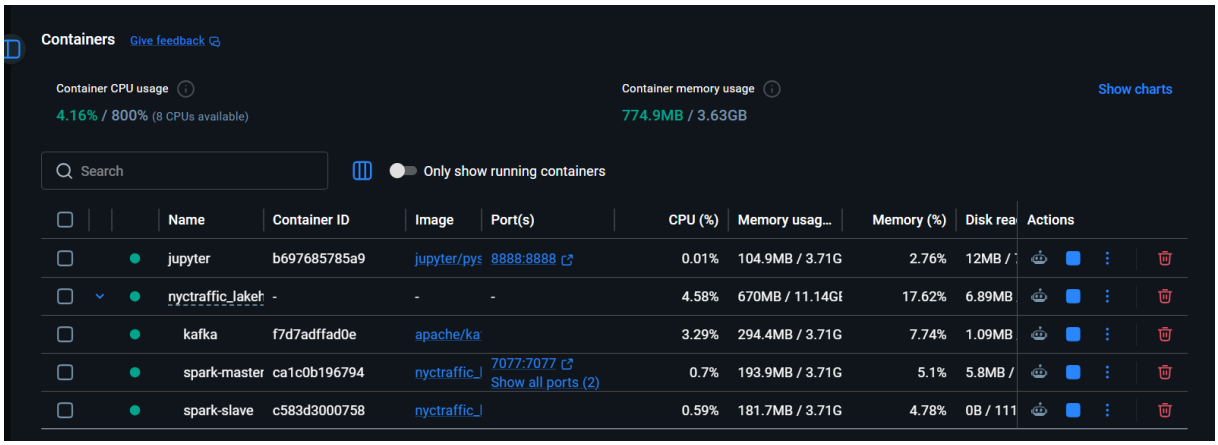
Để Jupyter có thể giao tiếp với Spark Master và sử dụng cùng cấu hình Azure Blob Storage, container Jupyter được khởi động trong cùng Docker network với Kafka và Spark. Trong quá trình chạy, Jupyter được gắn file .env để nhận các biến môi trường cần thiết, đồng thời mount thư mục notebooks/ của project vào thư mục làm việc trong container.

Khi khởi động Jupyter, nhóm ánh xạ cổng 8888 để truy cập giao diện JupyterLab từ trình duyệt. Ngoài ra, các cổng 4040, 4041 và 4042 cũng được ánh xạ để theo dõi Spark application UI khi chạy notebook.

Lệnh chạy (trên powershell với quyền admin) truy cập vào thư mục project:

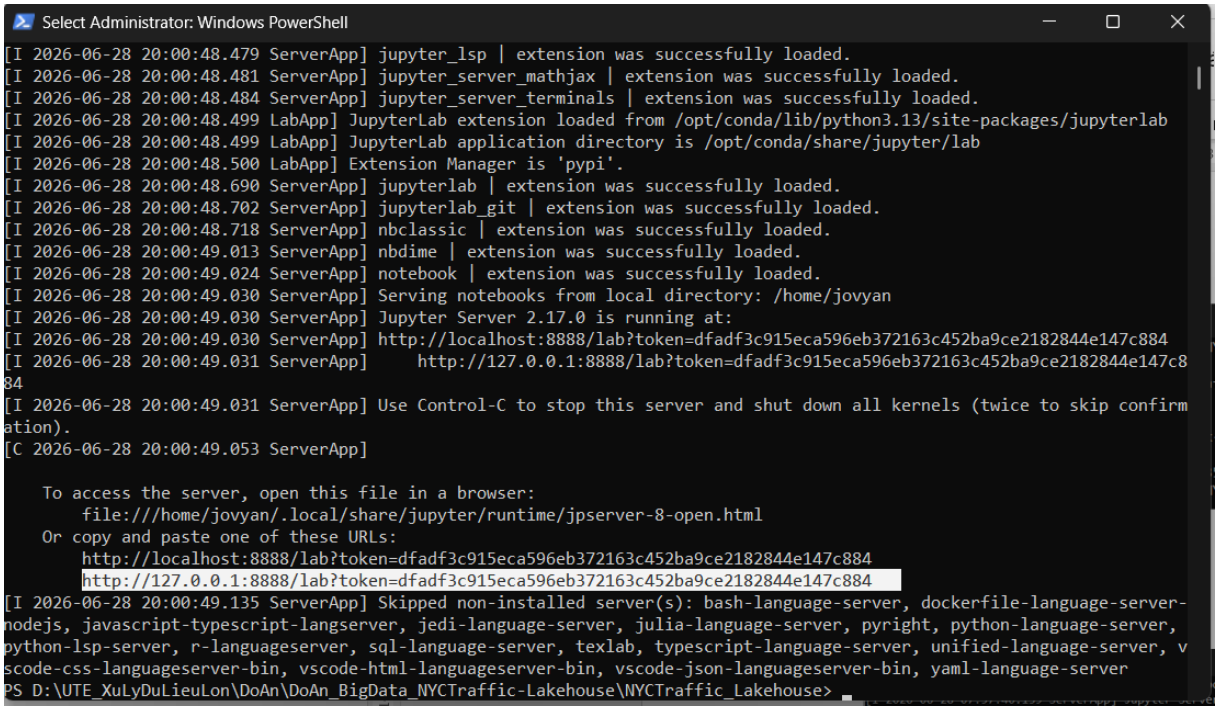
```
docker run -d `
  --name jupyter `
  --network nyctraffic_lakehouse_nyc-traffic-net `
  -p 8888:8888 `
  -p 4040:4040 `
  -p 4041:4041 `
  -p 4042:4042 `
  --env-file .env `
  -v "D:\UTE_XuLyDuLieuLon\DoAn\DoAn_BigData_NYCTraffic-Lakehouse\NYCTraffic_Lakehouse\notebooks:/home/jovyan/work" `
  quay.io/jupyter/pyspark-notebook:spark-4.0.0
```

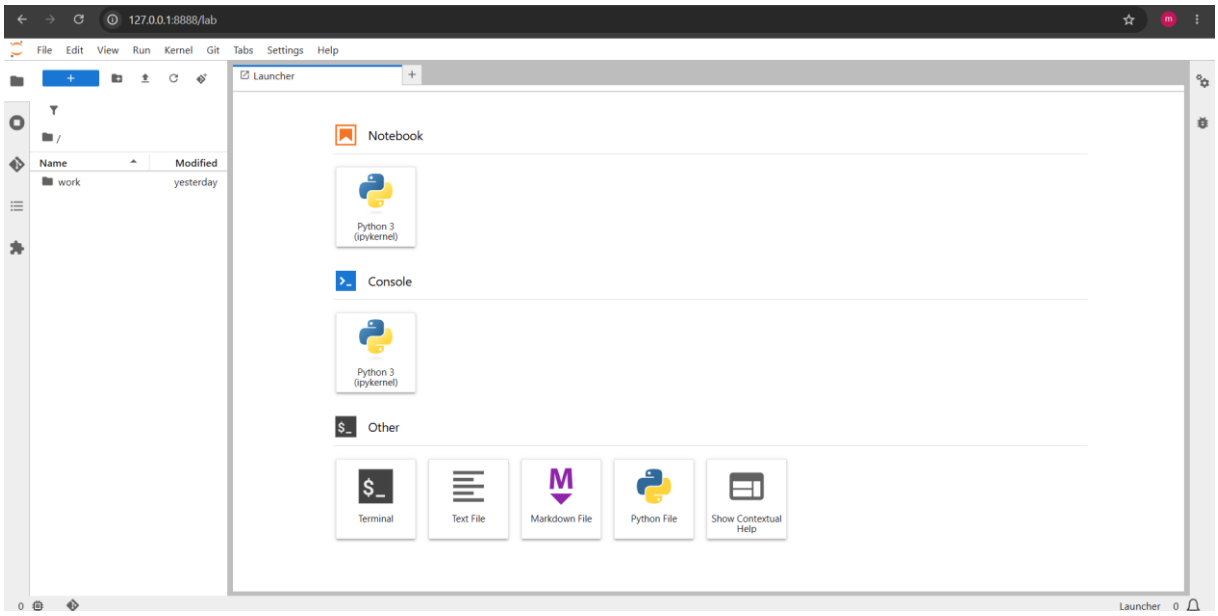




Hình 3.19: Khởi động Jupyter Notebook trong cùng Docker network

Lấy log truy cập bằng lệnh: docker logs jupyter



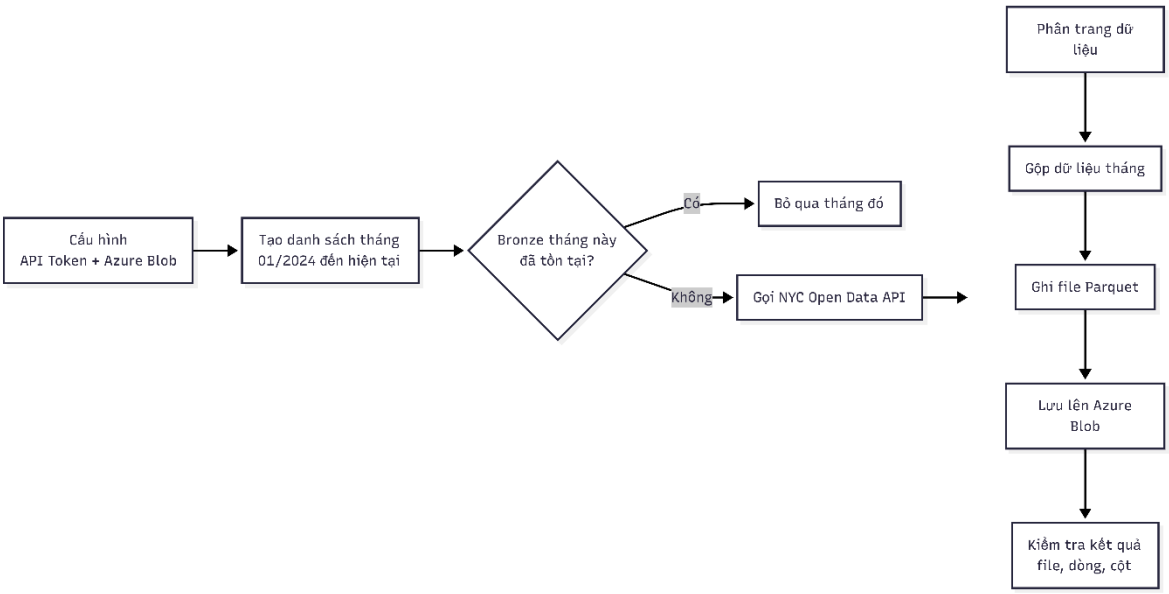


Hình 3.20: Truy cập Jupyter Notebook thông qua trình duyệt web

CHƯƠNG 4: XỬ LÝ DỮ LIỆU, HUẤN LUYỆN MÔ HÌNH VÀ ĐÁNH GIÁ

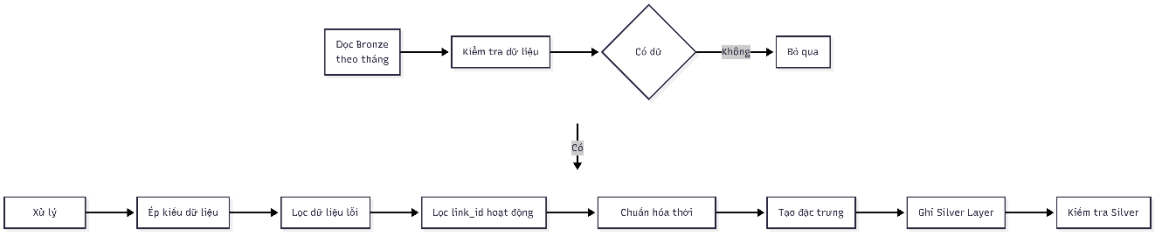
4.1. Quy trình thu thập, lưu Bronze và xử lý dữ liệu từ Bronze sang Silver

Minh họa quá trình thu thập dữ liệu lịch sử và lưu Bronze



Hình 4.1: Luồng xử lý lưu dữ liệu vào tầng Bronze

Minh họa luồng hoạt động xử lý từ Bronze sang Silver



Hình 4.2: Luồng xử lý từ Bronze sang Silver

4.1.1. Thu thập dữ liệu lịch sử từ NYC Open Data

Quá trình thu thập dữ liệu lịch sử được thực hiện trong notebook 02_bronze_historical.ipynb. Hệ thống gọi API của NYC Open Data theo từng tháng, sử dụng điều kiện thời gian trên trường data_as_of và cơ chế phân trang bằng $limit và $offset. Mỗi lần gọi API lấy tối đa 50.000 bản ghi, sau đó tiếp tục tăng offset cho đến khi lấy hết dữ liệu của tháng đó.

```python
APP_TOKEN = os.environ["SOCRATA_APP_TOKEN"]
CONN_STR = os.environ["AZURE_STORAGE_CONNECTION_STRING"]
CONTAINER = os.environ.get("AZURE_CONTAINER_NAME", "nyc-traffic-lakehouse")

API_URL = "https://data.cityofnewyork.us/resource/i4gi-tjb9.json"
LIMIT = 50000  # max per request

client = BlobServiceClient.from_connection_string(CONN_STR)
print("Config xong")
```

```
Config xong
```

```python
def pull_month(year, month):
    start = f"{year}-{month:02d}-01T00:00:00"
    if month == 12:
        end = f"{year+1}-01-01T00:00:00"
    else:
        end = f"{year}-{month+1:02d}-01T00:00:00"

    all_rows = []
    offset = 0

    while True:
        params = {
            "$where": f"data_as_of >= '{start}' AND data_as_of < '{end}'",
            "$limit": LIMIT,
            "$offset": offset,
            "$order": "data_as_of ASC",
            "$$app_token": APP_TOKEN
        }
        resp = requests.get(API_URL, params=params, timeout=60)
        resp.raise_for_status()
        rows = resp.json()

        if not rows:
            break

        all_rows.extend(rows)
        offset += LIMIT
        print(f"  {year}-{month:02d}: pulled {len(all_rows)} rows...", end="\r")

        if len(rows) < LIMIT:
            break

    return all_rows
```

Hình 4.3: Hàm thu thập dữ liệu lịch sử từ NYC Open Data

Sau khi dữ liệu của từng tháng được lấy về, hệ thống chuyển dữ liệu sang DataFrame và ghi xuống Bronze Layer dưới định dạng Parquet, giữ nguyên toàn bộ 13 cột gốc trả về từ API (id, speed, travel_time, status, data_as_of, link_id, link_points, encoded_poly_line, encoded_poly_line_lvls, owner, transcom_id, borough, link_name).

Bronze Layer được thiết kế theo nguyên tắc append-only, lưu trữ dữ liệu thô đúng như nguồn gốc để có thể truy vết lại khi cần, và không sử dụng cơ chế Delta Lake (_delta_log) như Silver và Gold.

Dữ liệu được lưu theo cấu trúc bronze/dot_traffic_speeds/historical/year=/month=/.

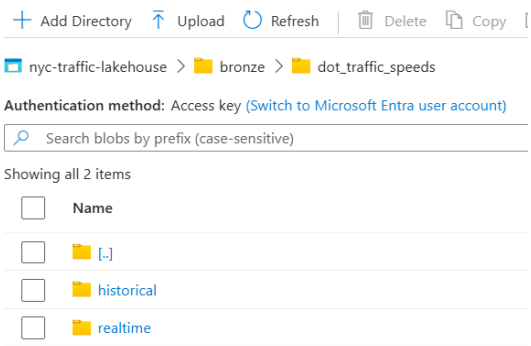
```python
def upload_parquet(df, year, month):
    blob_path = f"bronze/dot_traffic_speeds/historical/year={year}/month={month:02d}/raw_{year}_{month:02d}.parquet"

    table = pa.Table.from_pandas(df, preserve_index=False)
    buf = io.BytesIO()
    pq.write_table(table, buf)
    buf.seek(0)

    blob = client.get_blob_client(container=CONTAINER, blob=blob_path)
    blob.upload_blob(buf, overwrite=True)
    print(f"Uploaded: {blob_path} ({len(df)} rows)")
```

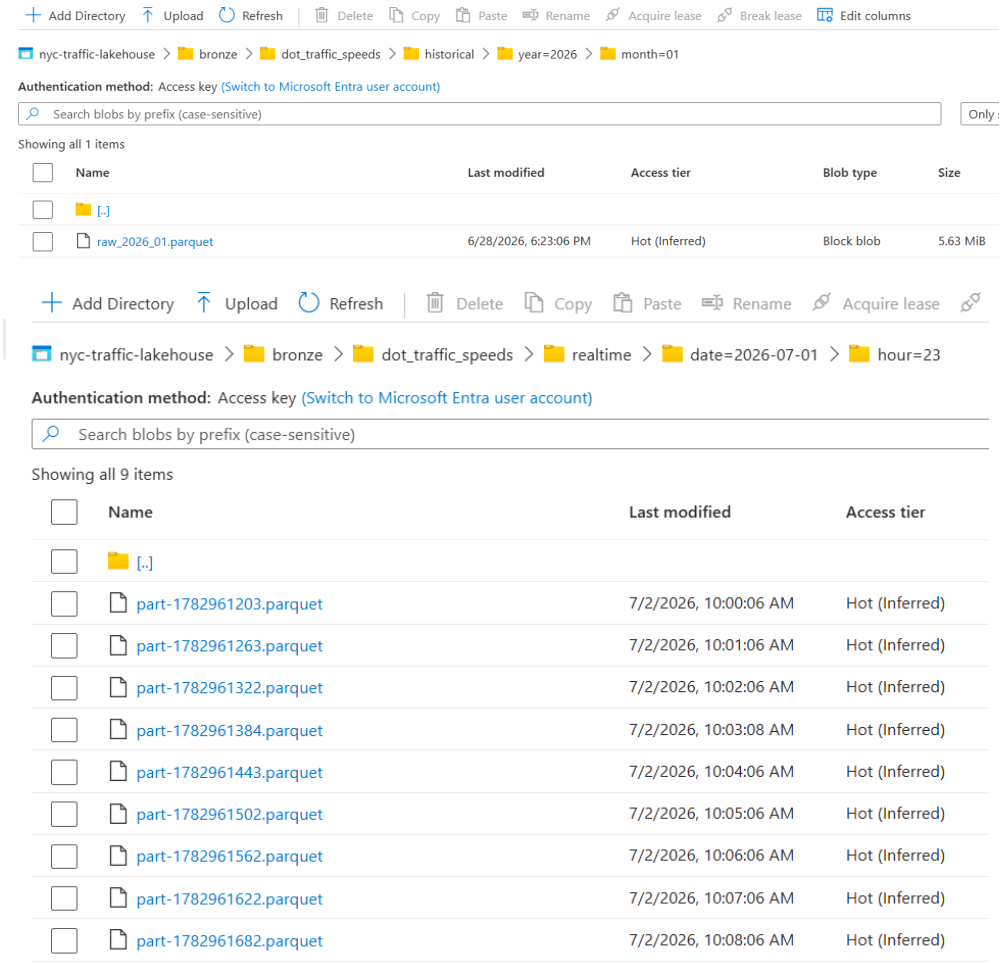
```python
# Kiểm tra
for year, month in months[:30]:
    blob_path = f"bronze/dot_traffic_speeds/historical/year={year}/month={month:02d}/raw_{year}_{month:02d}.parquet"
    blob = client.get_blob_client(container=CONTAINER, blob=blob_path)
    props = blob.get_blob_properties()
    print(f"{blob_path}: {props.size:,} bytes")
```

```
bronze/dot_traffic_speeds/historical/year=2024/month=01/raw_2024_01.parquet: 6,566,301 bytes
bronze/dot_traffic_speeds/historical/year=2024/month=02/raw_2024_02.parquet: 3,960,147 bytes
bronze/dot_traffic_speeds/historical/year=2024/month=03/raw_2024_03.parquet: 7,209,525 bytes
bronze/dot_traffic_speeds/historical/year=2024/month=04/raw_2024_04.parquet: 6,984,106 bytes
bronze/dot_traffic_speeds/historical/year=2024/month=05/raw_2024_05.parquet: 7,359,568 bytes
bronze/dot_traffic_speeds/historical/year=2024/month=06/raw_2024_06.parquet: 7,135,940 bytes
bronze/dot_traffic_speeds/historical/year=2024/month=07/raw_2024_07.parquet: 7,159,674 bytes
bronze/dot_traffic_speeds/historical/year=2024/month=08/raw_2024_08.parquet: 6,641,495 bytes
bronze/dot_traffic_speeds/historical/year=2024/month=09/raw_2024_09.parquet: 7,143,603 bytes
bronze/dot_traffic_speeds/historical/year=2024/month=10/raw_2024_10.parquet: 6,205,979 bytes
bronze/dot_traffic_speeds/historical/year=2024/month=11/raw_2024_11.parquet: 6,422,266 bytes
bronze/dot_traffic_speeds/historical/year=2024/month=12/raw_2024_12.parquet: 4,651,859 bytes
bronze/dot_traffic_speeds/historical/year=2025/month=01/raw_2025_01.parquet: 5,823,061 bytes
bronze/dot_traffic_speeds/historical/year=2025/month=02/raw_2025_02.parquet: 5,201,663 bytes
bronze/dot_traffic_speeds/historical/year=2025/month=03/raw_2025_03.parquet: 6,465,231 bytes
bronze/dot_traffic_speeds/historical/year=2025/month=04/raw_2025_04.parquet: 5,294,074 bytes
bronze/dot_traffic_speeds/historical/year=2025/month=05/raw_2025_05.parquet: 6,368,740 bytes
bronze/dot_traffic_speeds/historical/year=2025/month=06/raw_2025_06.parquet: 6,100,002 bytes
bronze/dot_traffic_speeds/historical/year=2025/month=07/raw_2025_07.parquet: 5,841,242 bytes
bronze/dot_traffic_speeds/historical/year=2025/month=08/raw_2025_08.parquet: 5,929,793 bytes
bronze/dot_traffic_speeds/historical/year=2025/month=09/raw_2025_09.parquet: 5,733,897 bytes
bronze/dot_traffic_speeds/historical/year=2025/month=10/raw_2025_10.parquet: 5,894,218 bytes
bronze/dot_traffic_speeds/historical/year=2025/month=11/raw_2025_11.parquet: 6,477,330 bytes
bronze/dot_traffic_speeds/historical/year=2025/month=12/raw_2025_12.parquet: 5,855,001 bytes
```

Hình 4.4: Hàm ghi dữ liệu lịch sử xuống Bronze Layer và kiểm tra

Sau khi ghi xong, nhóm kiểm tra trực tiếp trên Azure Blob Storage để xác nhận dữ liệu được lưu đúng cấu trúc bronze/dot_traffic_speeds/historical/, phân vùng theo year/month, giữ đầy đủ 13 cột gốc như thiết kế.





Hình 4.5: Kiểm tra dữ liệu Bronze trên Azure Blob Storage

```
2024-01: 996,717 rows, 13 cols
Columns: ['id', 'speed', 'travel_time', 'status', 'data_as_of', 'link_id', 'link_points', 'encoded_poly_line', 'encoded_poly_line_lvls', 'owner
link_id unique: 122
data_as_of range: 2024-01-01T00:03:03.000 → 2024-01-29T09:07:12.000
speed range: 0.0 → 108.7
Null counts:
Series([], dtype: int64)
Sample row:
id                                                                  330
speed                                                             39.76
travel_time                                                         126
status                                                                0
data_as_of                                          2024-01-01T00:03:03.000
link_id                                                           4329507
link_points              40.75719,-73.99724 40.76017,-74.00382 40.76185...
encoded_poly_line                                 mkwwFvqsbMsQbh@oIlTmYp}@
encoded_poly_line_lvls                                              BBBB
owner                                                     PA -Lincoln Tunnel
transcom_id                                                       4329507
borough                                                         Manhattan
link_name                                LINCOLN TUNNEL W NORTH TUBE NY - NJ
Name: 0, dtype: object
```

Hình 4.6: Kiểm tra định dạng của dữ liệu Bronze

4.1.2. Thu thập dữ liệu real-time từ NYC DOT Feed

Dữ liệu real-time được xử lý theo hai mục đích: tạo danh sách active_link_ids.json trong giai đoạn chuẩn bị và phục vụ Job A trong giai đoạn vận hành.

Đối với mục đích thứ nhất, hệ thống gọi NYC DOT Feed (linkdata.nyctmc.org), giữ lại những link_id có trường DataAsOf nằm trong vòng 48 giờ gần nhất và lưu danh sách này ra file active_link_ids.json.

```
import requests

url = "https://linkdata.nyctmc.org/data/LinkSpeedQuery.txt"
response = requests.get(url, timeout=15)
lines = response.text.strip().split("\n")

header = lines[0].split("\t")
rows = [dict(zip(header, line.split("\t"))) for line in lines[1:]]

print("Header:", header)
print()
print("Row 0:", rows[0])
```

```
Header: ['"id"', '"Speed"', '"TravelTime"', '"Status"', '"DataAsOf"', '"linkId"'

Row 0: {'"id"': '"1"', '"Speed"': '"0"', '"TravelTime"': '"0"', '"Status"': '"-1
```

Hình 4.7: Gọi real-time feed và kiểm tra format của NYC DOT

```
import pytz

eastern = pytz.timezone("America/New_York")
now_et = datetime.now(eastern)
cutoff = now_et - timedelta(hours=48)

active_link_ids = []
for row in rows:
    try:
        data_as_of_str = row['"DataAsOf"'].strip().strip('"')
        link_id = row['"linkId"'].strip().strip('"')

        dt_naive = datetime.strptime(data_as_of_str, "%m/%d/%Y %H:%M:%S")
        dt_et = eastern.localize(dt_naive)

        if dt_et >= cutoff:
            active_link_ids.append(link_id)
    except Exception as e:
        continue

active_link_ids = sorted(set(active_link_ids))
print(f"Số link active: {len(active_link_ids)}")
print(active_link_ids[:5])
```

```
Số link active: 125
['4329472', '4329473', '4329499', '4329507', '4329508']
```

```
# Upload thẳng lên Azure
conn_str = os.environ["AZURE_STORAGE_CONNECTION_STRING"]
container = os.environ.get("AZURE_CONTAINER_NAME", "nyc-traffic-lakehouse")

client = BlobServiceClient.from_connection_string(conn_str)
blob = client.get_blob_client(container=container, blob="artifacts/active_link_ids.json")

data = json.dumps(active_link_ids).encode("utf-8")
blob.upload_blob(data, overwrite=True)
```

Hình 4.8: Lọc ra active_link_ids và upload lên Azure Blob Storage

Đối với mục đích thứ hai, dùng cho Job A khi gọi NYC DOT Feed định kỳ mỗi 1 phút trong giai đoạn vận hành cho việc lọc snapshot hợp lệ, tính speed_ratio và current_congestion, đẩy dữ liệu vào Kafka để phục vụ Job B dự đoán, đồng thời ghi bổ sung snapshot thô xuống bronze/dot_traffic_speeds/realtime/ trên Azure Blob Storage.

```python
KAFKA_TOPIC = "nyc-traffic-snapshots"
REALTIME_FEED_URL = "https://linkdata.nyctmc.org/data/LinkSpeedQuery.txt"
EASTERN = pytz.timezone("America/New_York")
```

NYCTraffic_Lakehouse > job-a > collector.py > ...

```python

# Helper: parse TSV feed real-time
def fetch_realtime_feed():
    # Gọi linkdata.nyctmc.org. Timeout 15s, KHÔNG retry (intentional design)
    resp = requests.get(REALTIME_FEED_URL, timeout=15)
    resp.raise_for_status()
    lines = resp.text.strip().splitlines()
    header = lines[0].split("\t")
    rows = [dict(zip(header, line.split("\t"))) for line in lines[1:]]
    return rows

def parse_row(row):
    # Parse 1 row TSV -> dict sạch. Key có dấu nháy kép bao quanh
    link_id    = row['"linkId"'].strip().strip('"')
    speed_str  = row['"Speed"'].strip().strip('"')
    daof_str   = row['"DataAsOf"'].strip().strip('"')
    link_name  = row.get('"linkName"', '').strip().strip('"')

    speed = float(speed_str) if speed_str else None

    dt_naive = datetime.strptime(daof_str, "%m/%d/%Y %H:%M:%S")
    dt_et = EASTERN.localize(dt_naive)

    return {
        "link_id": link_id,
        "speed": speed,
        "data_as_of": dt_et,
        "link_name": link_name
    }
```

Hình 4.9: Vận hành real-time

Sau khi Job A vận hành, nhóm kiểm tra trực tiếp trên Azure Blob Storage để xác nhận nhánh bronze/dot_traffic_speeds/realtime/ được ghi liên tục theo phân vùng date/hour đúng như thiết kế.

nyc-traffic-lakehouse > bronze > dot_traffic_speeds > realtime > date=2026-06-30 > hour=23

Authentication method: Access key (Switch to Microsoft Entra user account)

Search blobs by prefix (case-sensitive)

Showing all 32 items

Name	Last modified	Access tier
[..]		
part-1782876435.parquet	7/1/2026, 10:27:17 AM	Hot (Inferred)
part-1782876500.parquet	7/1/2026, 10:28:22 AM	Hot (Inferred)
part-1782876560.parquet	7/1/2026, 10:29:22 AM	Hot (Inferred)
part-1782876621.parquet	7/1/2026, 10:30:23 AM	Hot (Inferred)
part-1782876681.parquet	7/1/2026, 10:31:23 AM	Hot (Inferred)
part-1782876742.parquet	7/1/2026, 10:32:25 AM	Hot (Inferred)
part-1782876802.parquet	7/1/2026, 10:33:24 AM	Hot (Inferred)
part-1782876863.parquet	7/1/2026, 10:34:25 AM	Hot (Inferred)
part-1782876922.parquet	7/1/2026, 10:35:25 AM	Hot (Inferred)
part-1782876983.parquet	7/1/2026, 10:36:26 AM	Hot (Inferred)

Hình 4.10: Kiểm tra dữ liệu Bronze realtime trên Azure Blob Storage

4.1.3. Chuẩn hóa schema dữ liệu, kiểu dữ liệu, múi giờ, lọc dữ liệu cảm biến lỗi, lọc các link_id đang hoạt động và trích xuất đặc trưng thời gian

Dựa trên danh sách cột đã xác định ở mục 2.4, pipeline Bronze to Silver chỉ giữ lại các trường phục vụ trực tiếp cho bài toán gồm link_id, borough, link_name, data_as_of và speed, loại bỏ hoàn toàn các cột không mang giá trị sử dụng (id, travel_time, status, link_points, encoded_poly_line, encoded_poly_line_lvls, owner, transcom_id).

Quy trình xử lý được thực hiện theo thứ tự sau:

1. Lọc giá trị speed bất thường (speed > 0 AND speed < 100) để loại bỏ lỗi cảm biến.
2. Lọc theo danh sách active_link_ids.json, chỉ giữ lại các đoạn đường đang hoạt động.
3. Chuẩn hóa múi giờ của trường data_as_of về America/New_York.
4. Sắp xếp dữ liệu theo link_id và data_as_of, làm tiền đề cho việc tính các đặc trưng lag/lead ở bước Silver -> Gold sau này.
5. Trích xuất các đặc trưng thời gian hour, day_of_week, month, is_weekend và is_holiday từ trường data_as_of đã chuẩn hóa múi giờ.

   Trong đó, day_of_week được tính theo quy ước của Spark (dayofweek(): 1 = Chủ Nhật, 7 = Thứ Bảy) để đảm bảo nhất quán với cách tính ở Job B trong giai đoạn predict; is_weekend nhận giá trị 1 nếu day_of_week thuộc {1, 7}.

   Riêng is_holiday được xác định bằng cách xây dựng một bảng ánh xạ ngày lễ (dict {date: 0/1}) từ thư viện holidays (holidays.US(subdiv="NY")) cho giai đoạn 2024–2026, sau đó chuyển thành một Spark DataFrame nhỏ và thực hiện broadcast join vào Silver theo cột ngày.

```python
import pytz

eastern = pytz.timezone("America/New_York")
now_et = datetime.now(eastern)
cutoff = now_et - timedelta(hours=48)

active_link_ids = []
for row in rows:
    try:
        data_as_of_str = row['"DataAsOf"'].strip().strip('"')
        link_id = row['"linkId"'].strip().strip('"')

        dt_naive = datetime.strptime(data_as_of_str, "%m/%d/%Y %H:%M:%S")
        dt_et = eastern.localize(dt_naive)

        if dt_et >= cutoff:
            active_link_ids.append(link_id)
    except Exception as e:
        continue

active_link_ids = sorted(set(active_link_ids))
print(f"Số link active: {len(active_link_ids)}")
print(active_link_ids[:5])
```

```
Số link active: 125
['4329472', '4329473', '4329499', '4329507', '4329508']
```

```python
# Verify
downloaded = blob.download_blob().readall()
ids = json.loads(downloaded)
print(f"✅ Verify: {len(ids)} link trên Azure")
print(ids[:5])
```

```
✅ Verify: 125 link trên Azure
['4329472', '4329473', '4329499', '4329507', '4329508']
```

```python
def process_bronze_to_silver(df):
    # 1. Cast và filter speed
    df = df.withColumn("speed", F.col("speed").cast("double"))
    df = df.filter((F.col("speed") > 0) & (F.col("speed") < 100))

    # 2. Lọc link_id active
    df = df.filter(F.col("link_id").isin(active_link_ids))

    # 3. Parse data_as_of: UTC naive string → timestamp → convert ET tường minh
    df = df.withColumn(
        "data_as_of",
        F.to_timestamp("data_as_of", "yyyy-MM-dd'T'HH:mm:ss.SSS")
    )
    # Convert tường minh UTC → America/New_York
    df = df.withColumn(
        "data_as_of",
        F.from_utc_timestamp(F.col("data_as_of"), "America/New_York")
    )

    # 4. Extract time features từ ET (session.timeZone=America/New_York đảm bảo nhất quán)
    df = df.withColumn("hour",        F.hour("data_as_of"))
    df = df.withColumn("day_of_week", F.dayofweek("data_as_of"))
    df = df.withColumn("month",       F.month("data_as_of"))
    df = df.withColumn("is_weekend",
        F.when(F.col("day_of_week").isin(1, 7), 1).otherwise(0).cast(IntegerType())
    )
```

```python
    # 5. is_holiday: broadcast join theo date string
    df = df.withColumn("date_str", F.date_format("data_as_of", "yyyy-MM-dd"))
    df = df.join(holiday_broadcast, df["date_str"] == holiday_df["holiday_date"], "left")
    df = df.withColumn(
        "is_holiday",
        F.when(F.col("holiday_date").isNotNull(), 1).otherwise(0).cast(IntegerType())
    )
    df = df.drop("date_str", "holiday_date")

    # 6. Giữ đúng schema Silver
    silver_cols = ["link_id", "borough", "link_name", "data_as_of", "speed",
                   "hour", "day_of_week", "month", "is_weekend", "is_holiday"]
    return df.select(silver_cols)

print("process_bronze_to_silver() sẵn sàng")
```

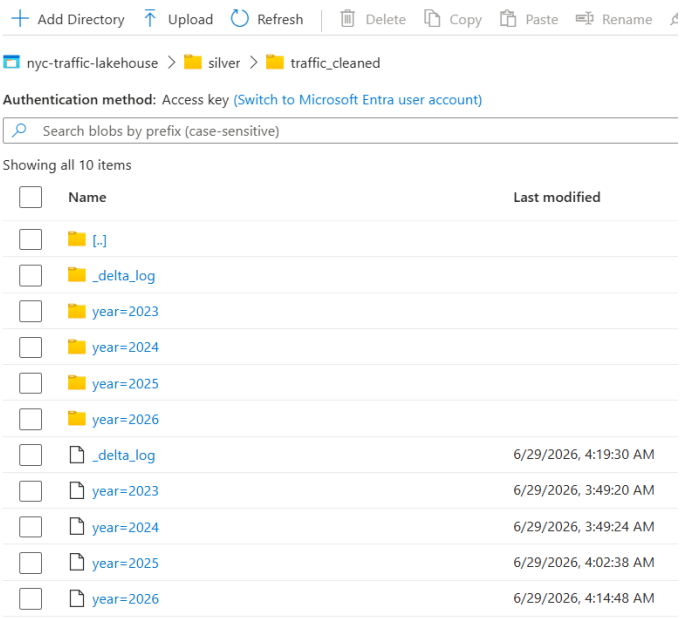
Hình 4.11: Hàm các bước xử lý Bronze sang Silver

```python
# Đọc Bronze tháng 01/2024
df_bronze_test = spark.read.parquet(
    f"{BASE_PATH}/bronze/dot_traffic_speeds/historical/year=2024/month=01/raw_2024_01.parquet"
)

print(f"Bronze rows: {df_bronze_test.count()}")
print(f"Columns: {df_bronze_test.columns}")

# Process
df_silver_test = process_bronze_to_silver(df_bronze_test)

print(f"\nSilver rows: {df_silver_test.count()}")
print(f"Link unique: {df_silver_test.select('link_id').distinct().count()}")
df_silver_test.printSchema()
df_silver_test.show(3, truncate=False)
```

```
Bronze rows: 996717
Columns: ['id', 'speed', 'travel_time', 'status', 'data_as_of', 'link_id', 'link_points', 'encoded_poly_lin

Silver rows: 801252
Link unique: 105
root
 |-- link_id: string (nullable = true)
 |-- borough: string (nullable = true)
 |-- link_name: string (nullable = true)
 |-- data_as_of: timestamp (nullable = true)
 |-- speed: double (nullable = true)
 |-- hour: integer (nullable = true)
 |-- day_of_week: integer (nullable = true)
 |-- month: integer (nullable = true)
 |-- is_weekend: integer (nullable = false)
 |-- is_holiday: integer (nullable = false)
```

Hình 4.12: Kiểm tra định dạng dữ liệu sau khi thực hiện xử lý Bronze sang Silver

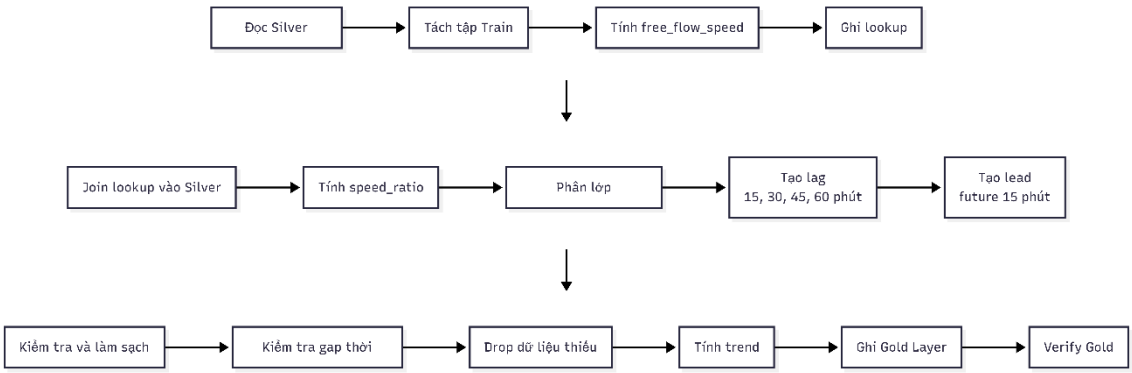
Ngoài việc kiểm tra định dạng dữ liệu output trực tiếp trong notebook, nhóm còn xác nhận lại trên Azure Blob Storage rằng thư mục silver/traffic_cleaned/ được ghi đúng dưới dạng Delta Lake (có _delta_log/), phân vùng theo year/month, và dữ liệu đã có đủ các cột phái sinh hour, day_of_week, month, is_weekend, is_holiday như thiết kế.



```
Tổng rows Silver: 23522760
Link unique: 116
Tháng có: [Row(year=2023, month=12), Row(year=2024, month=1), Row(year=2024, month=2), Row(year=2024, month=3), Row(year=
+-------+-------------+--------------------+-------------------+-----+----+-----------+-----+----------+----------+----+
|link_id|      borough|           link_name|         data_as_of|speed|hour|day_of_week|month|is_weekend|is_holiday|year|
+-------+-------------+--------------------+-------------------+-----+----+-----------+-----+----------+----------+----+
|4329499|Staten Island|MLK N WALKER STRE...|2024-04-01 00:04:03|54.68|   0|          2|    4|         0|         0|2024|
|4329507|     Manhattan|LINCOLN TUNNEL W ...|2024-04-01 00:04:03|44.73|   0|          2|    4|         0|         0|2024|
|4362249|       Queens|LIE WB LITTLE NEC...|2024-04-01 00:04:03|54.05|   0|          2|    4|         0|         0|2024|
+-------+-------------+--------------------+-------------------+-----+----+-----------+-----+----------+----------+----+
only showing top 3 rows
```

Hình 4.13: Kiểm tra dữ liệu Silver trên Azure Blob Storage và định dạng của dữ liệu

4.2. Xử lý từ Silver sang Gold



Hình 4.14: Minh họa luồng xử lý từ Silver sang Gold

Sau khi dữ liệu đã được làm sạch và chuẩn hóa ở Silver Layer, hệ thống tiếp tục xây dựng các đặc trưng phục vụ mô hình trong notebook 04_silver_to_gold.ipynb.

Đầu tiên là bước khởi tạo môi trường xử lý dữ liệu Silver -> Gold.

Chương trình đọc thông tin Azure Blob Storage từ biến môi trường, tạo SparkSession kết nối đến Spark Master trong Docker, cấu hình múi giờ New York để xử lý chính xác dữ liệu giao thông, bổ sung thư viện hadoop-azure để truy cập Azure Blob Storage và delta-spark để hỗ trợ Delta Lake.

Sau khi SparkSession được tạo thành công, chương trình in ra phiên bản Spark là 4.0.0, chứng minh môi trường Spark đã sẵn sàng.

Các biến này đã có cấu hình key từ file .env

```python
import os
import sys
from pyspark.sql import SparkSession

ACCOUNT_NAME = os.environ["AZURE_STORAGE_ACCOUNT_NAME"]
ACCOUNT_KEY  = os.environ["AZURE_STORAGE_ACCOUNT_KEY"]
CONTAINER    = os.environ.get("AZURE_CONTAINER_NAME", "nyc-traffic-lakehouse")
CONN_STR     = os.environ["AZURE_STORAGE_CONNECTION_STRING"]

spark = (SparkSession.builder
    .appName("Silver_to_Gold")
    .master("spark://spark-master:7077")
    .config("spark.sql.session.timeZone", "America/New_York")
    .config("spark.pyspark.python", "/usr/bin/python3")
    .config("spark.executorEnv.PYSPARK_PYTHON", "/usr/bin/python3")
    .config("spark.jars.packages",
            "org.apache.hadoop:hadoop-azure:3.3.4,"
            "io.delta:delta-spark_2.13:4.0.0")
    .config("spark.sql.extensions", "io.delta.sql.DeltaSparkSessionExtension")
    .config("spark.sql.catalog.spark_catalog", "org.apache.spark.sql.delta.catalog.DeltaCatalog")
    .config(f"spark.hadoop.fs.azure.account.key.{ACCOUNT_NAME}.blob.core.windows.net", ACCOUNT_KEY)
    .getOrCreate())

spark.sparkContext.setLogLevel("WARN")
print(f"Spark version: {spark.version}")
```

```
Spark version: 4.0.0
```

Hình 4.15: Thiết lập SparkSession, các biến môi trường trong 04_silver_to_gold.ipynb

Sau đó thiết lập một số đường dẫn để có thể làm việc trên Azure Blob Storage

```python
from pyspark.sql import functions as F
from pyspark.sql.window import Window

BASE_PATH = f"wasbs://{CONTAINER}@{ACCOUNT_NAME}.blob.core.windows.net"
SILVER_PATH = f"{BASE_PATH}/silver/traffic_cleaned"
LOOKUP_PATH = f"{BASE_PATH}/artifacts/free_flow_speed_lookup.parquet"
GOLD_PATH   = f"{BASE_PATH}/gold/training_features"

print(f"BASE_PATH: {BASE_PATH}")
```

```
BASE_PATH: wasbs://nyc-traffic-lakehouse@nyctrafficlakehouse.blob.core.windows.net
```

Hình 4.16: Khai báo các đường dẫn để làm việc với Azure Blob Storage

Để thử xem cấu hình đường dẫn đúng chưa thì tiến hành đọc dữ liệu trên silver.

```python
df_silver = spark.read.format("delta").load(SILVER_PATH)

print("Tổng rows Silver:", df_silver.count())
print("Số link_id unique:", df_silver.select("link_id").distinct().count())

df_silver.printSchema()
df_silver.show(5, truncate=False)
```

```
Tổng rows Silver: 23522760
Số link_id unique: 116
```

```
root
 |-- link_id: string (nullable = true)
 |-- borough: string (nullable = true)
 |-- link_name: string (nullable = true)
 |-- data_as_of: timestamp (nullable = true)
 |-- speed: double (nullable = true)
 |-- hour: integer (nullable = true)
 |-- day_of_week: integer (nullable = true)
 |-- month: integer (nullable = true)
 |-- is_weekend: integer (nullable = true)
 |-- is_holiday: integer (nullable = true)
 |-- year: integer (nullable = true)

+-------+-------------+---------------------------------+-------------------+-----+----+-----------+-----+----------+----------+----+
|link_id|borough      |link_name                        |data_as_of         |speed|hour|day_of_week|month|is_weekend|is_holiday|year|
+-------+-------------+---------------------------------+-------------------+-----+----+-----------+-----+----------+----------+----+
|4329499|Staten Island|MLK N WALKER STREET - NJ ROUTE 169 |2024-04-01 00:04:03|54.68|0   |2          |4    |0         |0         |2024|
|4329507|Manhattan    |LINCOLN TUNNEL W NORTH TUBE NY - NJ|2024-04-01 00:04:03|44.73|0   |2          |4    |0         |0         |2024|
|4362249|Queens       |LIE WB LITTLE NECK PKWY - NB CIP   |2024-04-01 00:04:03|54.05|0   |2          |4    |0         |0         |2024|
|4362251|Queens       |LIE WB LITTLE NECK PKWY - NB CVE   |2024-04-01 00:04:03|55.92|0   |2          |4    |0         |0         |2024|
|4362314|Queens       |TNB S Qns Anchorage - CIP S @ TNB  |2024-04-01 00:04:03|52.19|0   |2          |4    |0         |0         |2024|
+-------+-------------+---------------------------------+-------------------+-----+----+-----------+-----+----------+----------+----+
only showing top 5 rows
```

Hình 4.17: Đọc Silver và kiểm tra tổng quan

Kiểm tra thành công thì tách riêng tập train cho phần tính toán để tránh data leak (train lấy từ 1/1/2024 – tháng 9/2025).

```
# Train: 01/2024 -> 09/2025
df_train = df_silver.filter(
    (F.col("year") == 2024) |
    ((F.col("year") == 2025) & (F.col("month") <= 9))
)

print(f"Train rows: {df_train.count()}")
print(f"Train link_id unique: {df_train.select('link_id').distinct().count()}")

Train rows: 16914113
Train link_id unique: 116
```

Hình 4.18: Tách riêng các tháng train để tính toán

Sau khi tách riêng được tập train thì tiến hành tính toán các thông số cần thiết phục vụ giai đoạn sau là huấn luyện mô hình.

4.2.1. Tính free_flow_speed theo P85 của từng link_id

Đầu tiên là tính free_flow_speed được tính bằng percentile_approx(speed, 0.85) theo từng link_id trên tập Train.

```python
df_p85 = (df_train.groupBy("link_id")
    .agg(F.expr("percentile_approx(speed, 0.85)").alias("free_flow_speed"))
)

# borough, link_name là thuộc tính tĩnh của mỗi link -> lấy first() là đủ
df_link_meta = (df_train.groupBy("link_id")
    .agg(F.first("borough").alias("borough"),
         F.first("link_name").alias("link_name"))
)

df_lookup = df_p85.join(df_link_meta, "link_id", "left")

n_lookup = df_lookup.count()
print(f"Số link trong lookup: {n_lookup}")
df_lookup.orderBy("link_id").show(10)
```

```
Số link trong lookup: 116
+-------+---------------+-------------+--------------------+
|link_id|free_flow_speed|      borough|           link_name|
+-------+---------------+-------------+--------------------+
|4329472|           29.2|    Manhattan|LINCOLN TUNNEL E ...|
|4329473|          31.68|    Manhattan|LINCOLN TUNNEL E ...|
|4329499|          54.05|Staten Island|MLK N WALKER STRE...|
|4329507|          39.14|    Manhattan|LINCOLN TUNNEL W ...|
|4329508|          33.55|    Manhattan|LINCOLN TUNNEL W ...|
|4362244|           55.3|       Queens|CVE NB LIE - WILL...|
|4362249|          50.95|       Queens|LIE WB LITTLE NEC...|
|4362250|          60.27|       Queens|CVE NB GCP - WILL...|
|4362251|          53.43|       Queens|LIE WB LITTLE NEC...|
|4362314|          50.95|       Queens|TNB S Qns Anchora...|
+-------+---------------+-------------+--------------------+
only showing top 10 rows
```

Hình 4.19: Tính free_flow_speed bằng P85 trên tập Train

Ghi file parquet lên azure theo đường dẫn đã định nghĩa và kiểm tra.

```python
df_lookup.write.mode("overwrite").parquet(LOOKUP_PATH)
print(f"Ghi free_flow_speed_lookup.parquet → {LOOKUP_PATH} ({n_lookup} rows)")
```

Python

```
_speed_lookup.parquet → wasbs://nyc-traffic-lakehouse@nyctrafficlakehouse.blob.core.windows.net/artifacts/free_flow_speed_lookup.parquet (116 rows)
```

nyc-traffic-lakehouse > artifacts

Authentication method: Access key (Switch to Microsoft Entra user account)

Search blobs by prefix (case-sensitive)

Showing all 5 items

Name	Last modified	Access tier
[..]		
model		
active_link_ids.json	6/28/2026, 2:01:12 PM	Hot (Inferred)
free_flow_speed_lookup.parquet	6/30/2026, 4:09:47 PM	Hot (Inferred)
link_coordinates.parquet	6/30/2026, 4:10:36 PM	Hot (Inferred)
model	6/30/2026, 10:42:28 PM	Hot (Inferred)

Hình 4.20: Ghi file free_flow_speed_lookup lên Azure Blob Storage

Và join thêm cột vào silver theo link_id

```python
# Silver đã có borough, link_name sẵn -> chỉ cần lấy free_flow_speed từ lookup
df_gold = df_silver.join(
    df_lookup.select("link_id", "free_flow_speed"),
    on="link_id",
    how="left"
)

print(f"Rows sau join: {df_gold.count()}")
print(f"Rows free_flow_speed null (link thiếu lookup): "
      f"{df_gold.filter(F.col('free_flow_speed').isNull()).count()}")
```

```
Rows sau join: 23522760
Rows free_flow_speed null (link thiếu lookup): 0
```

Hình 4.21: Join thêm cột vào lookup

4.2.2. Tính speed_ratio và xác định current_congestion

Sau khi join free_flow_speed, hệ thống tính

speed_ratio = speed / free_flow_speed

Và gán current_congestion theo quy tắc phân loại đã trình bày ở mục 2.2. Kết quả của bước này là hai đặc trưng quan trọng dùng cho cả huấn luyện mô hình. Từ output có thể thấy được số lượng dữ liệu sau khi phân loại thành từng lớp.

```python
df_gold = df_gold.withColumn(
    "speed_ratio",
    F.col("speed") / F.col("free_flow_speed")
)

# Ngưỡng 0.67 / 0.40 -> 3 lớp:
#   0 = thông thoáng   (speed_ratio >= 0.67)
#   1 = tắc trung bình (0.40 <= speed_ratio < 0.67)
#   2 = tắc nặng       (speed_ratio < 0.40)
# QUAN TRỌNG: phải check isNull() trước, nếu không Spark sẽ rơi vào .otherwise(2) sai cho các giá trị null
df_gold = df_gold.withColumn(
    "current_congestion",
    F.when(F.col("speed_ratio").isNull(), F.lit(None))
     .when(F.col("speed_ratio") >= 0.67, F.lit(0))
     .when(F.col("speed_ratio") >= 0.40, F.lit(1))
     .otherwise(F.lit(2))
     .cast("int")
)

df_gold.groupBy("current_congestion").count().orderBy("current_congestion").show()
```

```
+-----------------+--------+
|current_congestion|   count|
+-----------------+--------+
|                0|15915889|
|                1| 3465650|
|                2| 4141221|
+-----------------+--------+
```

Hình 4.22: Tính tỷ lệ tốc độ và phân loại trạng thái tắc nghẽn hiện tại thành ba lớp.

4.2.3. Tạo đặc trưng trễ 15, 30, 45 và 60 phút, nhãn future_congestion_15min

Sau khi tính toán và chia thành 3 lớp thì ở mỗi mốc thời gian, hệ thống tạo hai nhóm đặc trưng: past_congestion_xmin biểu diễn trạng thái tắc nghẽn tại thời điểm quá khứ, còn past_speed_ratio_xmin biểu diễn tỷ lệ tốc độ tại cùng thời điểm đó.

Nhãn mục tiêu của bài toán là future_congestion_15min, biểu diễn mức độ tắc nghẽn của cùng một link_id sau 15 phút.

```python
window_link = Window.partitionBy("link_id").orderBy("data_as_of")

lag_map = {15: 3, 30: 6, 45: 9, 60: 12}  # phút : số dòng lag (giả định ~5 phút/dòng)

for minutes, offset in lag_map.items():
    df_gold = df_gold.withColumn(
        f"past_congestion_{minutes}min",
        F.lag("current_congestion", offset).over(window_link)
    )
    df_gold = df_gold.withColumn(
        f"past_speed_ratio_{minutes}min",
        F.lag("speed_ratio", offset).over(window_link)
    )
    # cột phụ để validate gap thời gian thực tế, sẽ drop sau
    df_gold = df_gold.withColumn(
        f"_past_ts_{minutes}min",
        F.lag("data_as_of", offset).over(window_link)
    )

# Label: future_congestion_15min = lead 3 dòng (~15 phút sau)
df_gold = df_gold.withColumn(
    "future_congestion_15min",
    F.lead("current_congestion", 3).over(window_link)
)
df_gold = df_gold.withColumn(
    "_future_ts_15min",
    F.lead("data_as_of", 3).over(window_link)
)
```

Hình 4.23: Tạo các đặc trưng trễ 15, 30, 45, 60 phút và nhãn dự đoán sau 15 phút

Sau khi có được các đặc trưng trễ thì tiến hành kiểm tra độ hợp lệ của các mốc thời gian quá khứ và tương lai trước khi đưa dữ liệu vào bước huấn luyện mô hình.

Hàm valid_gap() được xây dựng để kiểm tra xem một khoảng cách thời gian có hợp lệ hay không. Nếu giá trị khoảng cách bị null, bản ghi tạm thời vẫn được cho qua vì có thể một số link không có đủ dữ liệu quá khứ hoặc tương lai. Các giá trị null này sẽ được xử lý ở bước loại bỏ dữ liệu thiếu sau đó. Nếu không null, khoảng cách thời gian phải nằm trong khoảng dung sai đã quy định. Cuối cùng, chương trình lọc lại DataFrame df_gold, chỉ giữ các dòng thỏa mãn điều kiện hợp lệ về khoảng cách thời gian.

```python
tolerance = {15: (10, 20), 30: (25, 35), 45: (40, 50), 60: (55, 65)}

# Tính gap (phút) cho từng lag
for minutes in [15, 30, 45, 60]:
    df_gold = df_gold.withColumn(
        f"_gap_{minutes}min",
        (F.col("data_as_of").cast("long") - F.col(f"_past_ts_{minutes}min").cast("long")) / 60
    )

# Gap cho lead (future)
df_gold = df_gold.withColumn(
    "_gap_future_15min",
    (F.col("_future_ts_15min").cast("long") - F.col("data_as_of").cast("long")) / 60
)

# Điều kiện hợp lệ: nếu lag/lead null (do ở biên link) -> tạm cho qua, sẽ bị loại ở bước Drop NaN sau.
# Nếu lag/lead có giá trị -> gap phải nằm trong khung dung sai.
def valid_gap(col_gap, lo, hi):
    return F.col(col_gap).isNull() | ((F.col(col_gap) >= lo) & (F.col(col_gap) <= hi))

cond = valid_gap("_gap_future_15min", 10, 20)
for minutes, (lo, hi) in tolerance.items():
    cond = cond & valid_gap(f"_gap_{minutes}min", lo, hi)

rows_before = df_gold.count()
df_gold = df_gold.filter(cond)
rows_after = df_gold.count()
print(f"Trước validate gap: {rows_before:,} | Sau: {rows_after:,} | Đã loại: {rows_before - rows_after:,}")
```

```
Trước validate gap: 23,522,760 | Sau: 21,888,843 | Đã loại: 1,633,917
```

Hình 4.24: Lọc lại DataFrame df_gold

Tiếp theo là loại bỏ các dòng bị thiếu dữ liệu quan trọng (tức là đã bị gán null ở bước trước).

```
required_for_dropna = (
    [f"past_congestion_{m}min" for m in [15, 30, 45, 60]] +
    [f"past_speed_ratio_{m}min" for m in [15, 30, 45, 60]] +
    ["future_congestion_15min", "speed_ratio", "current_congestion"]
)

rows_before = df_gold.count()
df_gold = df_gold.dropna(subset=required_for_dropna)
rows_after = df_gold.count()
print(f"Trước dropna: {rows_before:,} | Sau: {rows_after:,} | Đã loại: {rows_before - rows_after:,}")

Trước dropna: 21,888,843 | Sau: 21,887,219 | Đã loại: 1,624
```

Hình 4.25: Loại bỏ các dòng bị thiếu dữ liệu quan trọng

4.2.4. Tạo đặc trưng xu hướng tắc nghẽn và xu hướng tốc độ

Sau khi lọc xử lý xong thì tiến hành tính 2 cột xu hướng

- congestion_trend được tính bằng:

congestion_trend = current_congestion - past_congestion_15min

Nếu giá trị này dương, tình trạng giao thông đang có xu hướng xấu đi so với 15 phút trước. Nếu giá trị âm, tình trạng giao thông đang có xu hướng cải thiện. Nếu bằng 0, trạng thái tắc nghẽn gần như không thay đổi.

- Tương tự, speed_ratio_trend được tính bằng:

speed_ratio_trend = speed_ratio - past_speed_ratio_15min

Đặc trưng này phản ánh xu hướng tăng hoặc giảm của tỷ lệ tốc độ. Nếu speed_ratio_trend âm, tốc độ tương đối đang giảm, có thể báo hiệu nguy cơ ùn tắc tăng lên. Nếu giá trị dương, tốc độ tương đối đang tăng, cho thấy giao thông có xu hướng thông thoáng hơn.

```
df_gold = df_gold.withColumn(
    "congestion_trend",
    F.col("current_congestion") - F.col("past_congestion_15min")
)
df_gold = df_gold.withColumn(
    "speed_ratio_trend",
    F.col("speed_ratio") - F.col("past_speed_ratio_15min")
)
```

Hình 4.26: Tạo đặc trưng xu hướng tắc nghẽn và xu hướng tốc độ

4.2.5. Danh sách đặc trưng đầu vào và biến mục tiêu đầu ra

Sau khi xử lý và tính toán đầy đủ thì tiến hành lấy tập đặc trưng đầu vào gồm 19 cột, được chia thành ba nhóm chính. Nhóm thứ nhất là đặc trưng định danh và không

gian, gồm link_id và borough. Nhóm thứ hai là đặc trưng trạng thái hiện tại và thời gian, gồm hour, day_of_week, month, is_weekend, is_holiday, speed_ratio và current_congestion. Nhóm thứ ba là đặc trưng quá khứ và xu hướng, gồm các biến trễ 15, 30, 45, 60 phút và hai biến xu hướng.

Biến mục tiêu của mô hình là future_congestion_15min; danh sách 19 đặc trưng đầu vào được tổng hợp như trong Hình 4.27.

```
feature_cols = [
    "link_id", "borough", "hour", "day_of_week", "month", "is_weekend", "is_holiday",   # 7
    "speed_ratio", "current_congestion",                                                 # 9
    "past_congestion_15min", "past_speed_ratio_15min",                                   # 11
    "past_congestion_30min", "past_speed_ratio_30min",                                   # 13
    "past_congestion_45min", "past_speed_ratio_45min",                                   # 15
    "past_congestion_60min", "past_speed_ratio_60min",                                   # 17
    "congestion_trend", "speed_ratio_trend",                                             # 19
]
assert len(feature_cols) == 19, f"Đếm lại! Hiện có {len(feature_cols)} cột"

df_gold_final = df_gold.select(
    *feature_cols,
    "future_congestion_15min",   # label
    "data_as_of",                 # mốc thời gian, không phải feature
    "year"                         # giữ để partition (month đã có trong feature_cols)
)
```

Hình 4.27: Đặc trưng đầu vào và biến mục tiêu đầu ra

```
df_gold_final.printSchema()
print(f"Tổng cột (19 feature + 1 label + data_as_of + year): {len(df_gold_final.columns)}")

root
 |-- link_id: string (nullable = true)
 |-- borough: string (nullable = true)
 |-- hour: integer (nullable = true)
 |-- day_of_week: integer (nullable = true)
 |-- month: integer (nullable = true)
 |-- is_weekend: integer (nullable = true)
 |-- is_holiday: integer (nullable = true)
 |-- speed_ratio: double (nullable = true)
 |-- current_congestion: integer (nullable = true)
 |-- past_congestion_15min: integer (nullable = true)
 |-- past_speed_ratio_15min: double (nullable = true)
 |-- past_congestion_30min: integer (nullable = true)
 |-- past_speed_ratio_30min: double (nullable = true)
 |-- past_congestion_45min: integer (nullable = true)
 |-- past_speed_ratio_45min: double (nullable = true)
 |-- past_congestion_60min: integer (nullable = true)
 |-- past_speed_ratio_60min: double (nullable = true)
 |-- congestion_trend: integer (nullable = true)
 |-- speed_ratio_trend: double (nullable = true)
 |-- future_congestion_15min: integer (nullable = true)
 |-- data_as_of: timestamp (nullable = true)
 |-- year: integer (nullable = true)

Tổng cột (19 feature + 1 label + data_as_of + year): 22
```

Hình 4.28: Output tổng số cột

Sau khi có được đặc trưng và biến đầu ra thì tiến hành ghi vào GOLD theo đường dẫn đã định nghĩa trên Azure Blob Storage.

```python
import time

# 1. Cache trước khi ghi — vì ta sẽ dùng df_gold_final 2 lần (đếm + ghi),
#    cache giúp lần ghi không phải tính lại toàn bộ window function/join từ đầu
df_gold_final = df_gold_final.repartition(64, "year", "month").cache()

# 2. Materialize cache bằng action đầu tiên + đo thời gian
t0 = time.time()
total_rows = df_gold_final.count()
print(f"[1/4] Đã tính xong toàn bộ pipeline: {total_rows:,} rows ({time.time()-t0:.1f}s)")

# 3. Log breakdown theo year/month TRƯỚC khi ghi — để biết khối lượng từng partition,
#    phát hiện sớm partition nào quá to (dễ OOM) trước khi bắt đầu ghi
print("[2/4] Phân bố rows theo year/month sẽ ghi:")
(df_gold_final.groupBy("year", "month")
    .count()
    .orderBy("year", "month")
    .show(40, truncate=False))

# 4. Ghi Delta + đo thời gian
print(f"[3/4] Bắt đầu ghi Gold Delta → {GOLD_PATH} ...")
t1 = time.time()

(df_gold_final.write.format("delta").mode("overwrite")
    .partitionBy("year", "month")
    .option("maxRecordsPerFile", 2_000_000)
    .save(GOLD_PATH))

print(f"Ghi xong sau {time.time()-t1:.1f}s")

# Giải phóng cache, không cần giữ trong memory nữa
df_gold_final.unpersist()
```

```
[1/4] Đã tính xong toàn bộ pipeline: 21,887,219 rows (129.5s)
[2/4] Phân bố rows theo year/month sẽ ghi:
```

```
[2/4] Phân bố rows theo year/month sẽ ghi:
+----+-----+------+
|year|month|count |
+----+-----+------+
|2023|12   |4512  |
|2024|1    |766767|
|2024|2    |462213|
|2024|3    |791258|
|2024|4    |782066|
|2024|5    |776141|
|2024|6    |782608|
|2024|7    |776870|
|2024|8    |741714|
|2024|9    |800545|
|2024|10   |817888|
|2024|11   |794750|
|2024|12   |666384|
|2025|1    |824472|
|2025|2    |691435|
|2025|3    |776612|
|2025|4    |758096|
|2025|5    |747874|
|2025|6    |688638|
|2025|7    |761101|
|2025|8    |816226|
|2025|9    |767461|
|2025|10   |779963|
|2025|11   |728302|
|2025|12   |762818|
```

```
|2025|1    |824472|
|2025|2    |691435|
|2025|3    |776612|
|2025|4    |758096|
|2025|5    |747874|
|2025|6    |688638|
|2025|7    |761101|
|2025|8    |816226|
|2025|9    |767461|
|2025|10   |779963|
|2025|11   |728302|
|2025|12   |762818|
|2026|1    |815809|
|2026|2    |688147|
|2026|3    |640131|
|2026|4    |496033|
|2026|5    |690488|
|2026|6    |489897|
+----+-----+------+

[3/4] Bắt đầu ghi Gold Delta → wasbs://nyc-traffic-lakehouse@nyctrafficlakehouse.blob.core.windows.net/gold/training_features ...
Ghi xong sau 204.4s

DataFrame[link_id: string, borough: string, hour: int, day_of_week: int, month: int, is_weekend: int, is_holiday: int, speed_ratio:
```

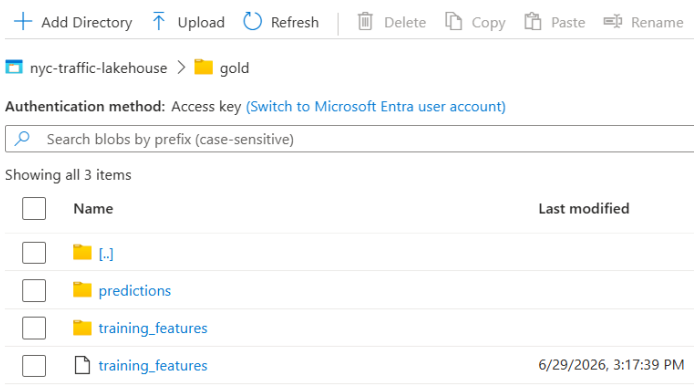
Hình 4.29: Tiến hành ghi vào gold

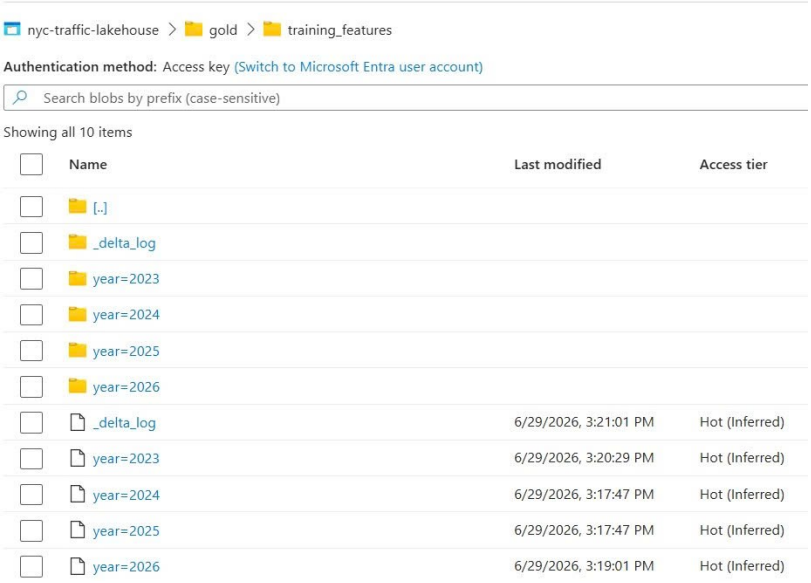
Đã ghi xong vào đường dẫn trên azure blob storage tầng gold/training_features. Load lại từ gold để kiểm tra.

```
df_check = spark.read.format("delta").load(GOLD_PATH)
print(f"Tổng rows: {df_check.count():,}")
print(f"Link unique: {df_check.select('link_id').distinct().count()}")
df_check.groupBy("year", "month").count().orderBy("year", "month").show(40)
df_check.groupBy("future_congestion_15min").count().orderBy("future_congestion_15min").show()

Tổng rows: 21,887,219
Link unique: 116
+----+-----+------+
|year|month| count|
+----+-----+------+
|2023|   12|  4512|
|2024|    1|766767|
|2024|    2|462213|
|2024|    3|791258|
```

Hình 4.30: Load lại từ tầng gold kiểm tra

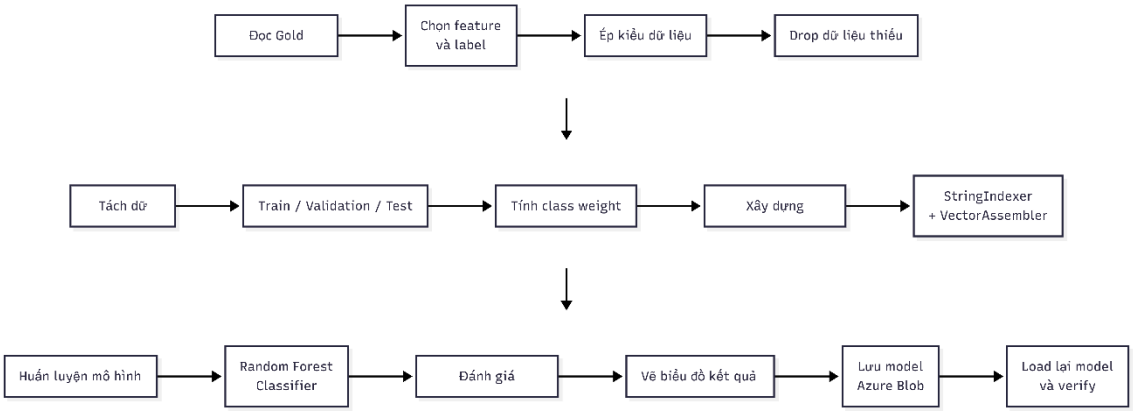
Truy cập Azure Blob Storage để xác nhận đã có





Hình 4.31: Kiểm tra gold trên Azure Blob Storage

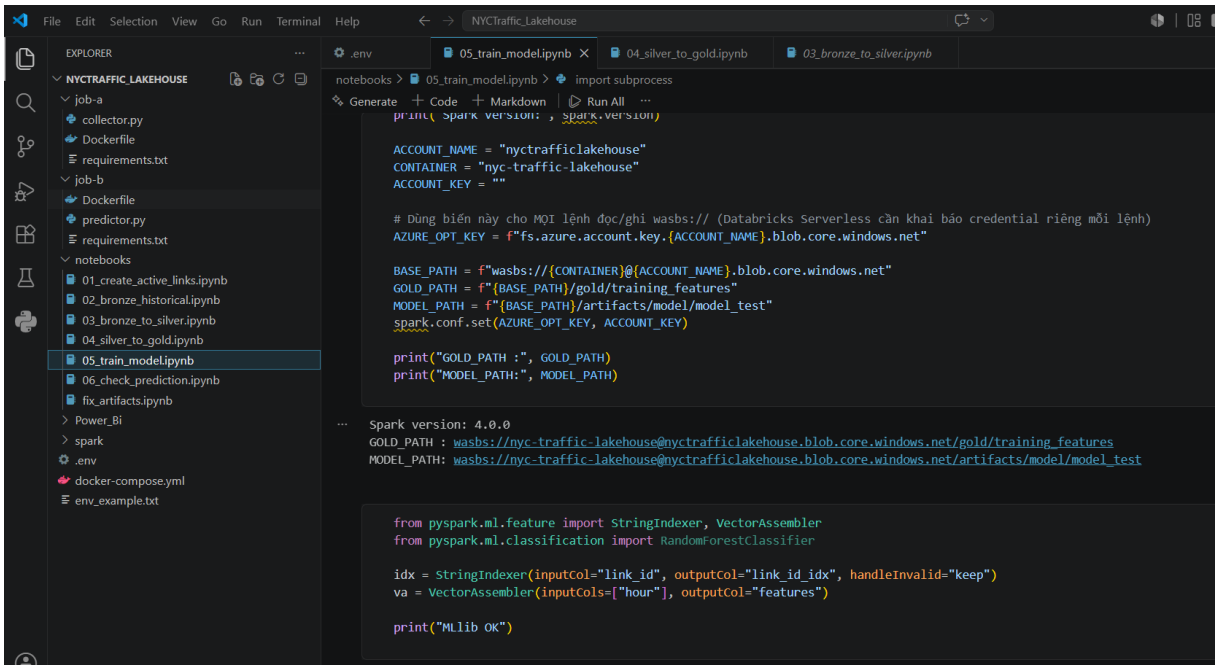
4.3. Xử lý và huấn luyện mô hình RandomForestClassifier bằng Spark MLlib



Hình 4.32: Minh họa luồng hoạt động trong xử lý và huấn luyện

RandomForestClassifier được lựa chọn vì phù hợp với bài toán phân loại nhiều lớp và có khả năng xử lý tốt các đặc trưng phi tuyến.

Phần này làm trên 1 notebook riêng là 05_train_model.ipynb



Hình 4.33: File 05_train_model.ipynb

Đầu tiên tương tự như các phần khác khi làm trên 1 notebook mới sẽ phải tạo lại SparkSession và cấu hình các thông số cũng như key kết nối.

```
from pyspark.sql import functions as F

spark.conf.set("spark.sql.session.timeZone", "America/New_York")
print("Spark version:", spark.version)

ACCOUNT_NAME = "nyctrafficlakehouse"
CONTAINER = "nyc-traffic-lakehouse"
ACCOUNT_KEY = ""

# Dùng biến này cho MỌI lệnh đọc/ghi wasbs:// (Databricks Serverless cần khai báo credential riêng mỗi lệnh)
AZURE_OPT_KEY = f"fs.azure.account.key.{ACCOUNT_NAME}.blob.core.windows.net"

BASE_PATH = f"wasbs://{CONTAINER}@{ACCOUNT_NAME}.blob.core.windows.net"
GOLD_PATH = f"{BASE_PATH}/gold/training_features"
MODEL_PATH = f"{BASE_PATH}/artifacts/model/model_test"
spark.conf.set(AZURE_OPT_KEY, ACCOUNT_KEY)

print("GOLD_PATH :", GOLD_PATH)
print("MODEL_PATH:", MODEL_PATH)

Spark version: 4.0.0
GOLD_PATH : wasbs://nyc-traffic-lakehouse@nyctrafficlakehouse.blob.core.windows.net/gold/training_features
MODEL_PATH: wasbs://nyc-traffic-lakehouse@nyctrafficlakehouse.blob.core.windows.net/artifacts/model/model_test
```

Hình 4.34: Thiết lập đường dẫn và các biến môi trường làm việc

Thiết lập xong thì phải test xong đọc được dữ liệu từ gold hay không (đọc từ đường dẫn GOLD_PATH đã cấu hình).

```python
df = (spark.read
    .format("delta")
    .option(AZURE_OPT_KEY, ACCOUNT_KEY)
    .load(GOLD_PATH))

print(f"Tổng rows Gold: {df.count():,}")

label_col = "future_congestion_15min"

feature_cols = [
    "link_id", "borough",
    "hour", "day_of_week", "month", "is_weekend", "is_holiday",
    "speed_ratio", "current_congestion",
    "past_congestion_15min", "past_speed_ratio_15min",
    "past_congestion_30min", "past_speed_ratio_30min",
    "past_congestion_45min", "past_speed_ratio_45min",
    "past_congestion_60min", "past_speed_ratio_60min",
    "congestion_trend", "speed_ratio_trend"
]

required_cols = list(dict.fromkeys(feature_cols + [label_col, "data_as_of", "year"]))
missing_cols = [c for c in required_cols if c not in df.columns]
assert len(missing_cols) == 0, f"Thiếu cột trong Gold: {missing_cols}"

print("Số feature:", len(feature_cols))
print("Required cols:", required_cols)
```

```
Tổng rows Gold: 21,887,219
Số feature: 19
Required cols: ['link_id', 'borough', 'hour', 'day_of_week', 'month', 'is_weekend', 'is_holiday', 'speed_ratio', 'current_congestion',
```

Hình 4.35: Đọc dữ liệu từ Gold

Tiếp theo là chuẩn bị dữ liệu cho mô hình gồm;

- required_cols là danh sách các cột sẽ được dùng cho mô hình
- chuẩn hóa kiểu dữ liệu cho một số cột
- khai báo danh sách các cột số và ép kiểu
- lấy mẫu dữ liệu 35% (từng lớp như đã phân loại)

```python
df_model_raw = df.select(*required_cols)

df_model_raw = (
    df_model_raw
    .withColumn("link_id", F.col("link_id").cast("string"))
    .withColumn("borough", F.col("borough").cast("string"))
    .withColumn(label_col, F.col(label_col).cast("double"))
)

numeric_cast_cols = [
    "hour", "day_of_week", "month", "is_weekend", "is_holiday",
    "speed_ratio", "current_congestion",
    "past_congestion_15min", "past_speed_ratio_15min",
    "past_congestion_30min", "past_speed_ratio_30min",
    "past_congestion_45min", "past_speed_ratio_45min",
    "past_congestion_60min", "past_speed_ratio_60min",
    "congestion_trend", "speed_ratio_trend",
    "year"
]
for c in numeric_cast_cols:
    df_model_raw = df_model_raw.withColumn(c, F.col(c).cast("double"))

df_model_raw = df_model_raw.dropna(subset=feature_cols + [label_col, "year", "month"])

# SAMPLE 35% TOÀN BỘ dataset NGAY TỪ ĐẦU, giữ tỉ lệ 3 class
SAMPLE_FRACTION = 0.35
fractions = {0.0: SAMPLE_FRACTION, 1.0: SAMPLE_FRACTION, 2.0: SAMPLE_FRACTION}

df_model_sampled = df_model_raw.sampleBy(label_col, fractions=fractions, seed=42)
```

Hình 4.36: Chuẩn bị dữ liệu cho huấn luyện

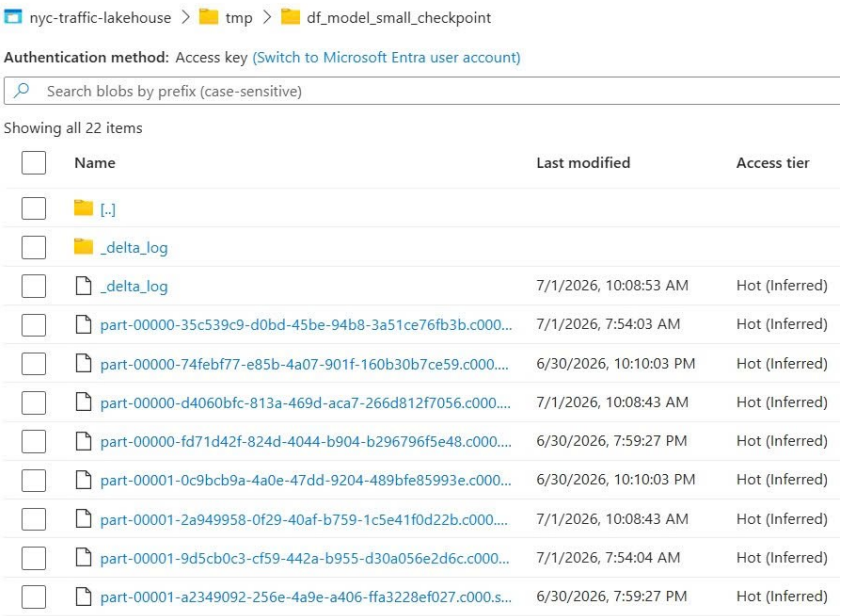
Sau đó ghi checkpoint lên Azure và xem số lượng data của từng lớp

```
# Checkpoint Delta (thay cho .cache(), vì Databricks Serverless không hỗ trợ .cache())
TMP_MODEL_CHECKPOINT = f"{BASE_PATH}/tmp/df_model_small_checkpoint"

(df_model_sampled.write
    .format("delta")
    .option(AZURE_OPT_KEY, ACCOUNT_KEY)
    .mode("overwrite")
    .save(TMP_MODEL_CHECKPOINT))

df_model = (spark.read
    .format("delta")
    .option(AZURE_OPT_KEY, ACCOUNT_KEY)
    .load(TMP_MODEL_CHECKPOINT))

n_model = df_model.count()
print(f"Rows sau sample {SAMPLE_FRACTION*100:.0f}%: {n_model:,}")
df_model.groupBy(label_col).count().orderBy(label_col).show()
```

```
Rows sau sample 35%: 7,663,509
+-----------------------+-------+
|future_congestion_15min|  count|
+-----------------------+-------+
|                    0.0|5252153|
|                    1.0|1129553|
|                    2.0|1281803|
+-----------------------+-------+
```

Hình 4.37: Ghi checkpoint lên Azure Blob Storage

Kiểm tra lại đã ghi checkpoint đúng chưa



Hình 4.38: Checkpoint đã ghi trên Azure Blob Storage

4.3.1. Chia Train, Validation và Test theo thời gian

Sau khi chuẩn bị data từng lớp xong thì tiến hành chia train/val/test

Trong pipeline, tập Train gồm dữ liệu từ tháng 01/2024 đến tháng 09/2025. Tập Validation gồm dữ liệu từ tháng 10/2025 đến tháng 12/2025. Tập Test gồm dữ liệu từ

tháng 01/2026 trở đi. Cách chia này giúp mô hình học từ dữ liệu lịch sử, kiểm tra tham số trên một giai đoạn kế tiếp và đánh giá cuối cùng trên giai đoạn mới nhất.

```
df_train = df_model.filter(
    (F.col("year") == 2024) | ((F.col("year") == 2025) & (F.col("month") <= 9))
)
df_val = df_model.filter(
    (F.col("year") == 2025) & (F.col("month").between(10, 12))
)
df_test = df_model.filter(F.col("year") >= 2026)

train_rows = df_train.count()
val_rows = df_val.count()
test_rows = df_test.count()
print(f"Train rows: {train_rows:,}")
print(f"Val rows  : {val_rows:,}")
print(f"Test rows : {test_rows:,}")

print("Train distribution:")
df_train.groupBy(label_col).count().orderBy(label_col).show()
print("Val distribution:")
df_val.groupBy(label_col).count().orderBy(label_col).show()
print("Test distribution:")
df_test.groupBy(label_col).count().orderBy(label_col).show()
```

```
Train rows: 5,528,651
Val rows  : 796,008
Test rows : 1,337,237
```

Hình 4.39: Chia dữ liệu Train, Validation và Test theo trục thời gian

4.3.2. Xử lý mất cân bằng lớp bằng weightCol

Chia xong thì thấy được sự mất cân bằng giữa các lớp nên cần tính class weight để model biết nên chú trọng vào class ít data.

Sau khi tính toán, trọng số được gán vào cột class_weight và truyền vào RandomForestClassifier thông qua tham số weightCol. Cách làm này giúp cải thiện khả năng học của mô hình trên cả ba lớp, đặc biệt là lớp tắc nghẽn nghiêm trọng.

```
num_classes = 3
class_counts = df_train.groupBy(label_col).count()
total_train = df_train.count()

weights_df = (
    class_counts
    .withColumn("class_weight", F.lit(total_train) / (F.lit(num_classes) * F.col("count")))
    .select(F.col(label_col).alias("label_for_weight"), F.col("class_weight").cast("double"))
)
print("Class weight tính trên Train:")
weights_df.orderBy("label_for_weight").show()

df_train_w = (
    df_train
    .join(F.broadcast(weights_df), df_train[label_col] == weights_df["label_for_weight"], "left")
    .drop("label_for_weight")
)
```

```
Class weight tính trên Train:
+----------------+------------------+
|label_for_weight|      class_weight|
+----------------+------------------+
|             0.0|0.4819638341287677|
|             1.0|2.2723652421666474|
|             2.0|2.0614925865105964|
+----------------+------------------+
```

Hình 4.40: Tính trọng số lớp theo inverse frequency

4.3.3. Xây dựng Pipeline trong Spark Mllib

Tính class weight xong thì xây dựng các bước huấn luyện mô hình RandomForest Classifier.

Quá trình huấn luyện được tổ chức dưới dạng Pipeline trong Spark MLlib. Pipeline gồm các bước chính: mã hóa biến phân loại, gộp đặc trưng thành vector và huấn luyện RandomForestClassifier.

```
from pyspark.ml import Pipeline
from pyspark.ml.feature import StringIndexer, VectorAssembler
from pyspark.ml.classification import RandomForestClassifier

categorical_indexers = [
    StringIndexer(inputCol="link_id", outputCol="link_id_idx", handleInvalid="keep"),
    StringIndexer(inputCol="borough", outputCol="borough_idx", handleInvalid="keep"),
]

numeric_features = [
    "hour", "day_of_week", "month", "is_weekend", "is_holiday",
    "speed_ratio", "current_congestion",
    "past_congestion_15min", "past_speed_ratio_15min",
    "past_congestion_30min", "past_speed_ratio_30min",
    "past_congestion_45min", "past_speed_ratio_45min",
    "past_congestion_60min", "past_speed_ratio_60min",
    "congestion_trend", "speed_ratio_trend",
]

assembler = VectorAssembler(
    inputCols=["link_id_idx", "borough_idx"] + numeric_features,
    outputCol="features",
    handleInvalid="keep"
)

# === Params cố định — model nhỏ ===
FINAL_PARAMS = {
    "numTrees": 80,
    "maxDepth": 12,
    "minInstancesPerNode": 5,
}

rf = RandomForestClassifier(
    featuresCol="features",
    labelCol=label_col,
    weightCol="class_weight",
    predictionCol="prediction",
    probabilityCol="probability",
    rawPredictionCol="rawPrediction",
    numTrees=FINAL_PARAMS["numTrees"],
    maxDepth=FINAL_PARAMS["maxDepth"],
    minInstancesPerNode=FINAL_PARAMS["minInstancesPerNode"],
    maxBins=132,
    impurity="gini",
    featureSubsetStrategy="sqrt",
    seed=42,
    subsamplingRate=0.8,
    maxMemoryInMB=400,
)

pipeline = Pipeline(stages=categorical_indexers + [assembler, rf])
print("Pipeline + params cố định:", FINAL_PARAMS)
```

```
Pipeline + params cố định: {'numTrees': 80, 'maxDepth': 12, 'minInstancesPerNode': 5}
```

Hình 4.41: Pipeline trong Spark Mllib

Hai biến link_id và borough là biến phân loại nên không thể đưa trực tiếp vào mô hình dưới dạng chuỗi. Hệ thống sử dụng StringIndexer để chuyển các giá trị chuỗi này thành chỉ số số học. link_id biểu diễn từng đoạn đường, còn borough biểu diễn quận hành chính của New York.

Sau khi mã hóa biến phân loại và chuẩn hóa các biến số, hệ thống sử dụng VectorAssembler để gộp toàn bộ đặc trưng đầu vào thành một vector duy nhất có tên features. Đây là định dạng bắt buộc của hầu hết thuật toán trong Spark MLlib.

Sau khi xây xong thì gọi hàm pipeline và huấn luyện mô hình

```
pipeline = Pipeline(stages=categorical_indexers + [assembler, rf])
print("Pipeline + params cố định:", FINAL_PARAMS)
```

```
Pipeline + params cố định: {'numTrees': 80, 'maxDepth': 12, 'minInstancesPerNode': 5}
```

```
import time

t0 = time.time()
model = pipeline.fit(df_train_w)
fit_time = time.time() - t0
print(f" Fit xong sau {fit_time/60:.2f} phút")
```

```
Fit xong sau 53.15 phút
```

Hình 4.42: Huấn luyện mô hình

4.3.4. Lưu mô hình lên Azure Blob Storage

Sau khi huấn luyện, mô hình được lưu lên Azure Blob Storage trong thư mục artifacts/model/random_forest_model/.

Do mô hình được lưu dưới dạng Spark PipelineModel, toàn bộ các bước xử lý trong pipeline đều được bảo toàn, bao gồm StringIndexer, VectorAssembler và RandomForestClassifier.

```python
MODEL_PATH = f"{BASE_PATH}/artifacts/model/random_forest_model"
```

```python
# Xóa model cũ trên Azure trước khi lưu mới — tránh rác do hash stage thay đổi giữa các lần fit
from azure.storage.blob import BlobServiceClient

CONN_STR = f"DefaultEndpointsProtocol=https;AccountName={ACCOUNT_NAME};AccountKey={ACCOUNT_KEY};EndpointSuffix=core.windows.net"
BLOB_MODEL_PREFIX = "artifacts/model/random_forest_model"

container_client = BlobServiceClient.from_connection_string(CONN_STR).get_container_client(CONTAINER)
old_blobs = list(container_client.list_blobs(name_starts_with=BLOB_MODEL_PREFIX))
for b in old_blobs:
    container_client.delete_blob(b.name)
print(f"Đã xóa {len(old_blobs)} file model cũ trên Azure")
```

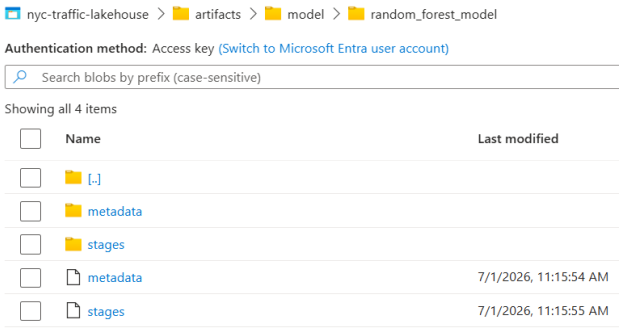
```
Đã xóa 36 file model cũ trên Azure
```

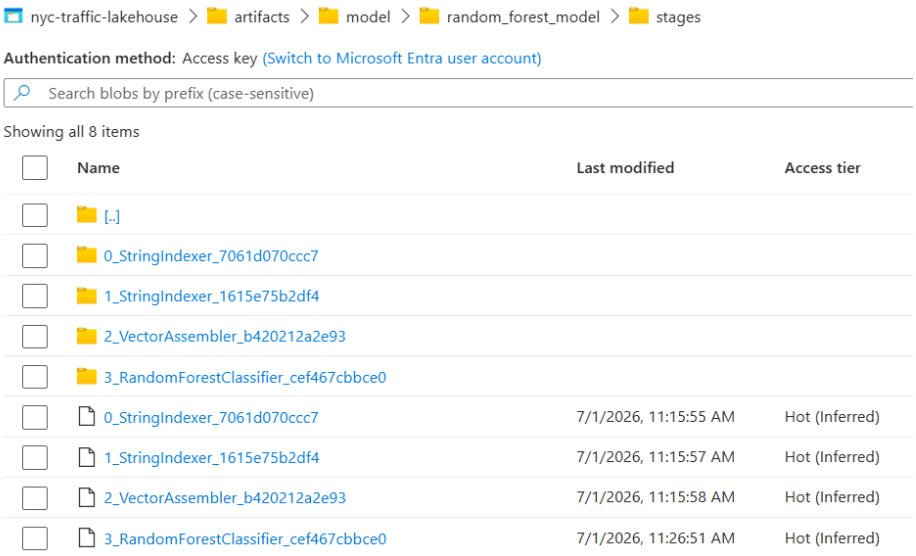
```python
# Save model trực tiếp lên Azure Blob
model.write().overwrite().save(MODEL_PATH)

print(" Đã save model lên:")
print(MODEL_PATH)
```

```
 Đã save model lên:
wasbs://nyc-traffic-lakehouse@nyctrafficlakehouse.blob.core.windows.net/artifacts/model/random_forest_model
```

Hình 4.43: Lưu model

Kiểm tra lại trên azure blob storage





Hình 4.44: Kiểm tra model đã ghi trên Azure Blob Storage

4.4. Đánh giá mô hình dự đoán

Bước tiếp theo sau khi huấn luyện mô hình RandomForestClassifier bằng Spark MLlib, nhóm tiến hành đánh giá mô hình trên hai tập dữ liệu riêng biệt là Validation set và Test set. Validation set được sử dụng để kiểm tra chất lượng mô hình trong giai đoạn lựa chọn cấu hình, còn Test set được sử dụng để đánh giá cuối cùng trên dữ liệu mới hơn theo trục thời gian.

Trong bài toán này, biến mục tiêu gồm ba lớp: lớp 0 biểu diễn trạng thái thông thoáng, lớp 1 biểu diễn trạng thái tắc vừa và lớp 2 biểu diễn trạng thái tắc nặng. Do phân phối dữ liệu giữa các lớp không hoàn toàn cân bằng, nhóm không chỉ sử dụng accuracy mà tập trung vào chỉ số Macro F1. Macro F1 tính trung bình F1-score của từng lớp, nhờ đó kết quả đánh giá không bị chi phối quá mạnh bởi lớp có số lượng bản ghi lớn nhất.

4.4.1. Phương pháp tính Macro F1

Để đánh giá mô hình, nhóm xây dựng hàm compute_macro_f1_fast() nhằm tính Precision, Recall và F1-score cho từng lớp dựa trên kết quả dự đoán. Hàm này trước hết nhóm dữ liệu theo nhãn thực tế và nhãn dự đoán để tạo ma trận nhầm lẫn ở dạng bảng đếm. Sau đó, với từng lớp 0, 1 và 2, hàm tính các giá trị tp, fp, fn, từ đó suy ra Precision, Recall và F1-score.

```
def compute_macro_f1_fast(pred_df, label_col=label_col, prediction_col="prediction"):
    cm_rows = pred_df.groupBy(label_col, prediction_col).count().collect()
    classes = [0.0, 1.0, 2.0]
    cm = {}
    for r in cm_rows:
        cm[(float(r[label_col]), float(r[prediction_col]))] = int(r["count"])

    total_rows = sum(cm.values())
    metrics = []
    for c in classes:
        tp = cm.get((c, c), 0)
        fp = sum(cm.get((other, c), 0) for other in classes if other != c)
        fn = sum(cm.get((c, other), 0) for other in classes if other != c)
        precision = tp / (tp + fp) if (tp + fp) > 0 else 0.0
        recall = tp / (tp + fn) if (tp + fn) > 0 else 0.0
        f1 = 2 * precision * recall / (precision + recall) if (precision + recall) > 0 else 0.0
        metrics.append((c, tp, fp, fn, precision, recall, f1))

    macro_f1 = sum(row[-1] for row in metrics) / len(classes)
    metrics_df = spark.createDataFrame(metrics, ["class", "tp", "fp", "fn", "precision", "recall", "f1"])
    return macro_f1, metrics_df, total_rows
```

Hình 4.45: Hàm tính Macro F1, Precision và Recall cho từng lớp

4.4.2. Kết quả đánh giá trên Validation set

Mô hình được đánh giá trước trên Validation set với 796.008 bản ghi. Kết quả cho thấy Macro F1 trên tập Validation đạt 0,761082. Đây là mức kết quả tương đối ổn định đối với bài toán phân loại ba lớp trên dữ liệu giao thông có tính biến động theo thời gian.

```
pred_val = model.transform(df_val).select(label_col, "prediction")
macro_f1_val, metrics_val_df, n_val_pred = compute_macro_f1_fast(pred_val)

print(f"Val rows: {n_val_pred:,}")
print(f"Macro F1 (Val): {macro_f1_val:.6f}")
metrics_val_df.orderBy("class").show()
```

```
Val rows: 796,008
Macro F1 (Val): 0.761082
+-----+------+-----+-----+------------------+------------------+------------------+
|class|    tp|   fp|   fn|         precision|            recall|                f1|
+-----+------+-----+-----+------------------+------------------+------------------+
|  0.0|467798|18553|76356|0.9618526537418449| 0.859679428985177|0.9079004953881835|
|  1.0| 82093|84581|33054|0.4925363284015503|0.7129408495227839| 0.582589657974388|
|  2.0|110863|32120|25844|0.7753579096815706|0.8109533527910056|  0.79275626586578|
+-----+------+-----+-----+------------------+------------------+------------------+
```

Hình 4.46: Kết quả đánh giá mô hình trên Validation set

Xét theo từng lớp, lớp 0 đạt F1-score khoảng 0,9079, cho thấy mô hình nhận diện khá tốt các trường hợp giao thông thông thoáng. Đây cũng là lớp có số lượng mẫu lớn, đồng thời có đặc trưng tốc độ rõ ràng hơn nên mô hình dễ học hơn. Lớp 2 đạt F1-score khoảng 0,7928, thể hiện mô hình có khả năng nhận diện tương đối tốt trạng thái tắc nặng. Trong khi đó, lớp 1 có F1-score khoảng 0,5826, thấp hơn hai lớp còn lại. Điều này cho thấy trạng thái tắc vừa là lớp khó phân biệt nhất, do nó nằm ở vùng trung gian giữa thông thoáng và tắc nặng.

4.4.3. Kết quả đánh giá trên Test set

Sau khi kiểm tra trên Validation set, mô hình được đánh giá trên Test set gồm 1.337.237 bản ghi. Đây là tập dữ liệu mới hơn theo thời gian, không được sử dụng trong quá trình huấn luyện và lựa chọn mô hình. Kết quả cho thấy Macro F1 trên Test set đạt 0,767470, cao hơn nhẹ so với Validation set.

```python
pred_test = model.transform(df_test).select(label_col, "prediction")
macro_f1_test, metrics_test_df, n_test_pred = compute_macro_f1_fast(pred_test)

print(f"Test rows: {n_test_pred:,}")
print(f"Macro F1 (Test): {macro_f1_test:.6f}")
metrics_test_df.orderBy("class").show()
```

```
Test rows: 1,337,237
Macro F1 (Test): 0.767470
+-----+------+------+------+------------------+------------------+------------------+
|class|    tp|    fp|    fn|         precision|            recall|                f1|
+-----+------+------+------+------------------+------------------+------------------+
|  0.0|755914| 31063|127128|0.9605287066839311| 0.856034027826536| 0.905275927998424|
|  1.0|145829|143388| 57474|0.5042200147294246|0.7172988101503667|0.5921749370583935|
|  2.0|206043| 55000| 44849|0.7893067425673165|0.8212418092246864|0.8049576606405111|
+-----+------+------+------+------------------+------------------+------------------+
```

Hình 4.47: Kết quả đánh giá mô hình trên Test set

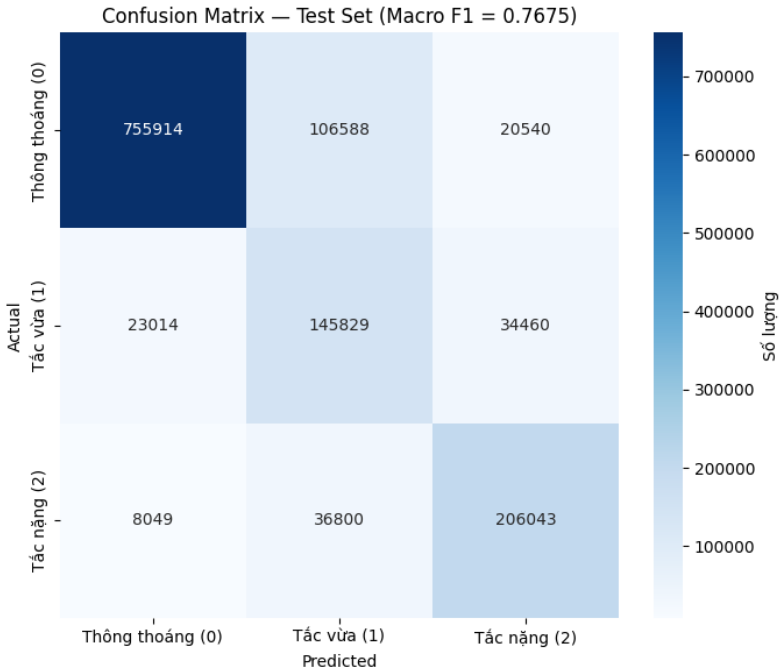
Kết quả trên Test set cho thấy mô hình không bị suy giảm khi chuyển sang giai đoạn dữ liệu mới hơn. Cụ thể, lớp 0 đạt F1-score khoảng 0,9053, lớp 1 đạt khoảng 0,5922 và lớp 2 đạt khoảng 0,8050. So với Validation set, F1-score của lớp 1 và lớp 2 đều cải thiện nhẹ, trong khi lớp 0 vẫn duy trì ở mức cao. Điều này cho thấy mô hình có khả năng tổng quát hóa tương đối tốt trên dữ liệu chưa từng thấy.

Tuy nhiên, kết quả cũng tiếp tục thể hiện hạn chế ở lớp 1. Lớp tắc vừa có Precision khoảng 0,5042 và Recall khoảng 0,7173, nghĩa là mô hình có xu hướng phát hiện được khá nhiều trường hợp tắc vừa nhưng vẫn tạo ra số lượng dự đoán nhầm tương đối lớn. Nói cách khác, lớp 1 đóng vai trò là vùng chuyển tiếp nên dễ bị nhầm giữa trạng thái thông thoáng và tắc nặng.

4.4.4. Phân tích Confusion Matrix trên Test set

Để quan sát chi tiết hơn các lỗi dự đoán, nhóm xây dựng Confusion Matrix trên Test set. Ma trận nhầm lẫn cho thấy mô hình dự đoán đúng 755.914 trường hợp thông thoáng, 145.829 trường hợp tắc vừa và 206.043 trường hợp tắc nặng.

```python
import matplotlib.pyplot as plt
import seaborn as sns
import numpy as np
import pandas as pd

# Lấy confusion matrix dạng đầy đủ (3x3) từ predictions Test
cm_pdf = (pred_test
    .groupBy(label_col, "prediction")
    .count()
    .toPandas())

classes = [0.0, 1.0, 2.0]
class_names = ["Thông thoáng (0)", "Tắc vừa (1)", "Tắc nặng (2)"]

cm_matrix = np.zeros((3, 3))
for _, row in cm_pdf.iterrows():
    true_idx = classes.index(row[label_col])
    pred_idx = classes.index(row["prediction"])
    cm_matrix[true_idx, pred_idx] = row["count"]

fig, ax = plt.subplots(figsize=(7, 6))
sns.heatmap(
    cm_matrix.astype(int),
    annot=True, fmt="d", cmap="Blues",
    xticklabels=class_names, yticklabels=class_names,
    cbar_kws={"label": "Số lượng"},
    ax=ax
)
ax.set_xlabel("Predicted")
ax.set_ylabel("Actual")
ax.set_title(f"Confusion Matrix — Test Set (Macro F1 = {macro_f1_test:.4f})")
plt.tight_layout()
plt.show()
```



Hình 4.48: Confusion Matrix trên Test set

Đối với lớp thông thoáng, phần lớn bản ghi được dự đoán đúng. Tuy nhiên, vẫn có 106.588 trường hợp thông thoáng bị dự đoán thành tắc vừa và 20.540 trường hợp bị dự đoán thành tắc nặng. Điều này cho thấy khi tốc độ giảm nhẹ hoặc có xu hướng xấu đi trong quá khứ gần, mô hình có thể đánh giá trạng thái hiện tại nghiêm trọng hơn thực tế.

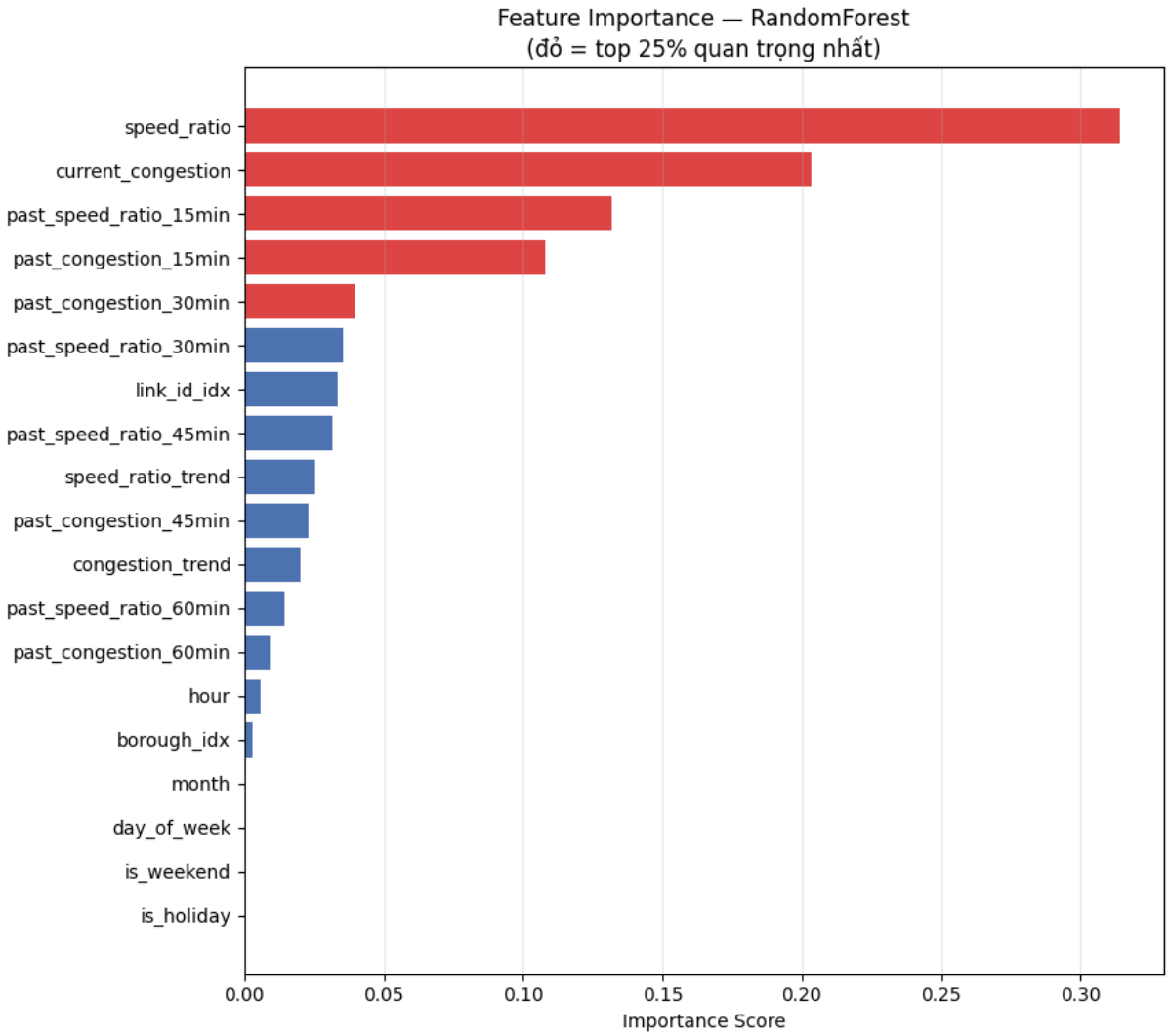
Đối với lớp tắc vừa, có 145.829 bản ghi được dự đoán đúng, nhưng cũng có 23.014 bản ghi bị dự đoán thành thông thoáng và 34.460 bản ghi bị dự đoán thành tắc nặng. Đây là lớp có mức độ nhập nhằng cao nhất vì nằm giữa hai trạng thái còn lại. Việc nhầm lẫn này là tương đối dễ hiểu, do ranh giới giữa tắc vừa và hai lớp lân cận phụ thuộc vào ngưỡng speed_ratio.

Đối với lớp tắc nặng, mô hình dự đoán đúng 206.043 bản ghi. Số trường hợp tắc nặng bị dự đoán thành thông thoáng là 8.049, tương đối thấp so với tổng số bản ghi lớp này. Đây là kết quả tích cực, vì trong bài toán giao thông, việc bỏ sót trạng thái tắc nặng thường nghiêm trọng hơn so với việc dự đoán dư mức độ tắc nghẽn.

4.4.5. Phân tích mức độ quan trọng của đặc trưng

Bên cạnh các chỉ số đánh giá, nhóm phân tích Feature Importance của RandomForestClassifier để xác định những đặc trưng có ảnh hưởng lớn nhất đến kết quả dự đoán. Kết quả cho thấy speed_ratio là đặc trưng quan trọng nhất, tiếp theo là current_congestion, past_speed_ratio_15min và past_congestion_15min.

```
# Feature Importance
rf_model = model.stages[-1]
importances = rf_model.featureImportances.toArray()

feature_names = ["link_id_idx", "borough_idx"] + numeric_features
importance_pdf = pd.DataFrame({
    "feature": feature_names,
    "importance": importances
}).sort_values("importance", ascending=True)

fig, ax = plt.subplots(figsize=(9, 8))
colors = ["#4C72B0" if imp < importance_pdf["importance"].quantile(0.75) else "#DD4444"
          for imp in importance_pdf["importance"]]
ax.barh(importance_pdf["feature"], importance_pdf["importance"], color=colors)
ax.set_xlabel("Importance Score")
ax.set_title("Feature Importance — RandomForest\n(đỏ = top 25% quan trọng nhất)")
ax.grid(axis="x", alpha=0.3)
plt.tight_layout()
plt.show()

print("\nTop 5 feature quan trọng nhất:")
print(importance_pdf.tail(5)[["feature", "importance"]].to_string(index=False))
```



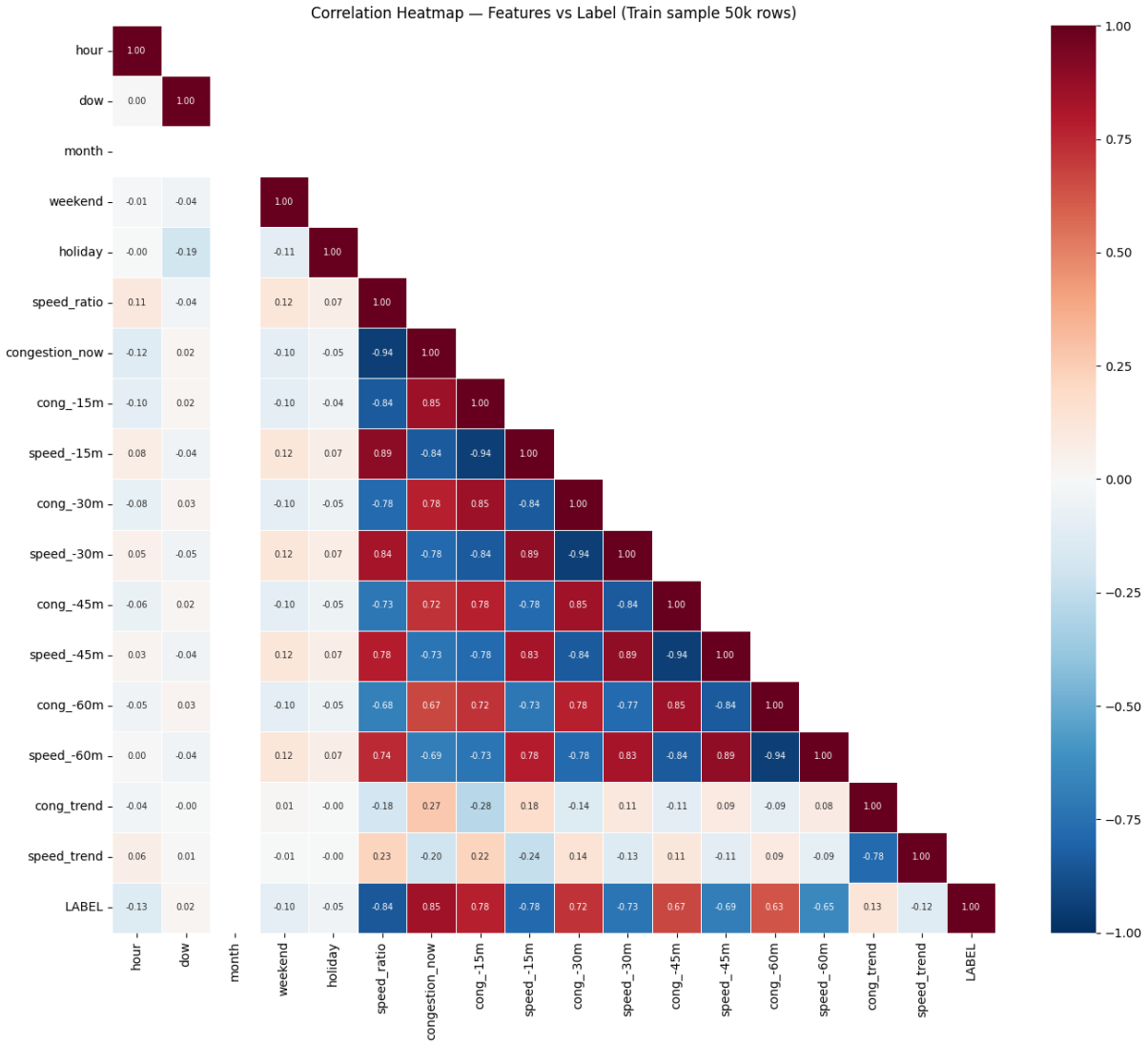

```
Top 5 feature quan trọng nhất:
                feature  importance
 past_congestion_30min    0.039878
 past_congestion_15min    0.108052
past_speed_ratio_15min    0.131761
    current_congestion    0.203324
           speed_ratio    0.314145
```

Hình 4.49: Mức độ quan trọng của các đặc trưng trong mô hình RandomForest

Kết quả này phù hợp với bản chất của bài toán. speed_ratio phản ánh trực tiếp mức độ suy giảm tốc độ so với tốc độ tự do của từng đoạn đường, do đó có khả năng phân biệt mạnh giữa các trạng thái giao thông. current_congestion là biến rời rạc được suy ra từ speed_ratio, giúp mô hình nắm bắt trạng thái hiện tại theo ba mức. Các đặc trưng trễ 15 phút cũng có vai trò quan trọng vì tình trạng giao thông trong tương lai gần thường chịu ảnh hưởng mạnh từ trạng thái hiện tại và quá khứ gần.

4.4.6. Phân tích tương quan giữa đặc trưng và nhãn

Nhóm cũng xây dựng heatmap tương quan giữa các đặc trưng và nhãn trên một mẫu dữ liệu Train gồm 50.000 dòng. Heatmap cho thấy nhãn LABEL có tương quan dương mạnh với congestion_now và các đặc trưng tắc nghẽn trong quá khứ, đồng thời có tương quan âm mạnh với speed_ratio và các đặc trưng tốc độ trong quá khứ.



Hình 4.50: Heatmap tương quan giữa đặc trưng và nhãn

Cụ thể, LABEL có tương quan dương cao với congestion_now, cong_15m, cong_30m, cong_45m và cong_60m. Điều này cho thấy khi trạng thái tắc nghẽn hiện tại và quá khứ tăng lên, khả năng nhãn tương lai rơi vào mức tắc cao hơn cũng tăng theo. Ngược lại, LABEL có tương quan âm mạnh với speed_ratio, speed_15m, speed_30m, speed_45m và speed_60m. Điều này phản ánh đúng logic giao thông: khi tỷ lệ tốc độ càng thấp, mức độ tắc nghẽn trong tương lai càng có xu hướng nghiêm trọng hơn.

Kết quả heatmap củng cố nhận xét từ Feature Importance: các đặc trưng liên quan đến tốc độ và trạng thái tắc nghẽn hiện tại/quá khứ gần có quan hệ rõ rệt nhất với nhãn dự đoán. Do current_congestion được suy ra từ speed_ratio, hai nhóm đặc trưng này có tương quan nghịch mạnh, nhưng vẫn được giữ lại vì Random Forest có thể khai thác cả giá trị liên tục và ngưỡng phân loại rời rạc.

4.5. Triển khai luồng dự đoán gần thời gian thực

Sau khi mô hình RandomForestClassifier được huấn luyện, hệ thống cần một luồng xử lý riêng để đưa mô hình vào vận hành thực tế, liên tục thu thập dữ liệu mới và sinh ra dự đoán cho 15 phút tiếp theo. Vì dữ liệu lịch sử từ NYC Open Data có độ trễ lớn nên không thể dùng cho mục đích dự đoán gần thời gian thực. Do đó, nhóm xây dựng một luồng xử lý độc lập với luồng huấn luyện, dựa trên dữ liệu real-time từ NYC DOT kết hợp với Apache Kafka làm lớp trung gian lưu trữ tạm thời. Luồng này gồm hai thành phần chính là Job A (collector) và Job B (predictor), giao tiếp với nhau thông qua Kafka topic nyc-traffic-snapshots.

4.5.1. Cấu trúc Kafka topic nyc-traffic-snapshots

Kafka được lựa chọn làm lớp trung gian giữa Job A và Job B vì hai lý do:

Thứ nhất, Job A và Job B chạy độc lập với chu kỳ khác nhau (1 phút và 5 phút), nên cần một cơ chế lưu trữ đệm cho phép Job B truy vấn lại dữ liệu trong một khoảng thời gian nhất định thay vì phải đồng bộ trực tiếp giữa hai tiến trình.

Thứ hai, việc tính các đặc trưng trễ (lag features) 15/30/45/60 phút đòi hỏi phải giữ lại lịch sử snapshot trong ít nhất 60–65 phút gần nhất, nên Kafka với cơ chế retention theo thời gian là lựa chọn phù hợp hơn so với việc Job B tự quản lý bộ nhớ đệm.

Topic được đặt tên *nyc-traffic-snapshots*, sử dụng key là link_id. Thời gian lưu trữ được cấu hình retention.ms = 5.400.000 ms lớn hơn khoảng thời gian cần thiết để tính đủ 4 mốc lag (tối đa 65 phút), tạo một khoảng đệm an toàn cho các trường hợp độ trễ dữ liệu hoặc lệch chu kỳ chạy giữa hai job.

```python
KAFKA_TOPIC = "nyc-traffic-snapshots"
REALTIME_FEED_URL = "https://linkdata.nyctmc.org/data/LinkSpeedQuery.txt"
EASTERN = pytz.timezone("America/New_York")

WARMUP_MIN_LINKS = 20   # < 20 distinct link trong Kafka 90 phút gần nhất -> warm-up
LINK_DEAD_MINUTES = 120  # DataAsOf cũ hơn 120 phút -> coi là link chết

blob_service = BlobServiceClient.from_connection_string(AZURE_CONN_STR)
producer = Producer({"bootstrap.servers": KAFKA_BOOTSTRAP})

# State giữ trong memory suốt vòng đời container
free_flow_speed_lookup = {}   # link_id -> {free_flow_speed, borough, link_name}
active_link_ids = set()
last_pushed_data_as_of = {}   # link_id -> data_as_of string (lần push gần nhất)
```

Hình 4.51: Topic được đặt tên nyc-traffic-snapshots

Message được cấu trúc dưới dạng JSON, chỉ chứa các trường tối thiểu cần thiết cho việc tính toán ở Job B:

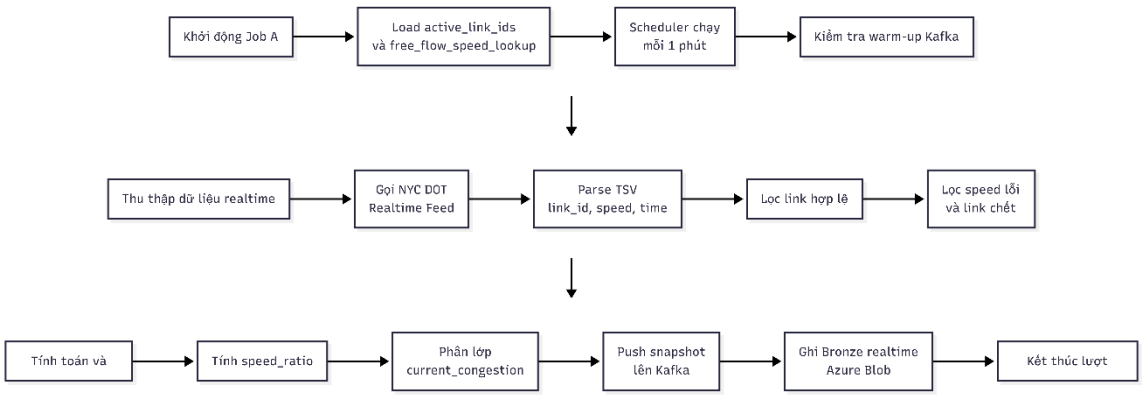
- Gồm: data_as_of, current_congestion, speed_ratio và link_name.

- Các trường như borough hay giá trị speed gốc không được đưa vào message vì Job B có thể lấy borough từ free_flow_speed_lookup đã nạp sẵn trong bộ nhớ, trong khi speed_ratio và current_congestion đã được Job A tính toán trước.

```python
# Push Kafka + ghi Azure Bronze realtime
def push_to_kafka(link_id: str, data_as_of: datetime, current_congestion: int, speed_ratio: float, link_name: str):
    value = {
        "data_as_of": data_as_of.isoformat(),
        "current_congestion": current_congestion,
        "speed_ratio": speed_ratio,
        "link_name": link_name
    }
    producer.produce(KAFKA_TOPIC, key=link_id, value=json.dumps(value))
```

Hình 4.52: Message được cấu trúc dưới dạng JSON

4.5.2. Cơ chế hoạt động của Job A – Collector



Job A đóng vai trò thu thập dữ liệu giao thông theo thời gian thực và đẩy vào Kafka. Khi container khởi động, Job A nạp một lần duy nhất free_flow_speed_lookup và active_link_ids.json từ Azure Blob Storage vào bộ nhớ, đồng thời khởi tạo Kafka Producer. Việc nạp một lần và giữ trong bộ nhớ suốt vòng đời container giúp tránh chi phí kết nối lại mỗi lần chạy, đảm bảo các lần chạy được căn theo đúng mốc giờ hệ thống (00:00, 00:01, 00:02...).

```
# Entry point
if __name__ == "__main__":
    logger.info("Job A starting...")
    load_lookup_and_active_links()

    scheduler = BlockingScheduler(timezone="America/New_York")
    scheduler.add_job(
        run_collector, "cron", minute="*", second=0,
        max_instances=1,      # nếu lần chạy trước chưa xong, skip lần mới thay vì chồng lấn
        misfire_grace_time=30  # cho phép trễ tối đa 30s nếu hệ thống bận
    )

    logger.info("Scheduler bắt đầu (mỗi 1 phút)...")
    try:
        scheduler.start()
    except (KeyboardInterrupt, SystemExit):
        logger.info("Job A dừng.")
```

Hình 4.53: Job A

Trong mỗi lần chạy, Job A gọi đến real-time feed linkdata.nyctmc.org với timeout 15 giây. Kết quả trả về gồm 153 link, được xử lý theo ba trường hợp:

- *Trường hợp 1*: Link đã ngừng hoạt động, được xác định khi trường DataAsOf của snapshot cũ hơn 120 phút so với thời điểm hiện tại. Với trường hợp này, Job A bỏ qua hoàn toàn, không xử lý và không đẩy dữ liệu lên Kafka.

- *Trường hợp 2:* Lỗi cảm biến tạm thời, xảy ra khi giá trị speed nhận về nhỏ hơn hoặc bằng 0, hoặc lớn hơn hoặc bằng 100. Đây là các giá trị không hợp lệ về mặt vật lý đối với tốc độ giao thông tại NYC. Trong trường hợp này, Job A không đẩy snapshot mới lên Kafka mà giữ nguyên các snapshot cũ trước đó của link này. Hiệu ứng của cách xử lý này là ở lần dự đoán tiếp theo, Job B sẽ tự động sử dụng snapshot hợp lệ gần nhất trước đó theo cơ chế LOCF (Last Observation Carried Forward), giúp hệ thống không bị gián đoạn hoàn toàn chỉ vì một lỗi cảm biến.

- *Trường hợp 3:* Dữ liệu hợp lệ, khi giá trị speed nằm trong khoảng (0, 100). Với trường hợp này, Job A trước tiên so sánh DataAsOf của snapshot hiện tại với DataAsOf

của lần push gần nhất cho link đó, được lưu trong một dict ở bộ nhớ trong tiến trình. Nếu DataAsOf không đổi, nghĩa là feed chưa cập nhật dữ liệu mới, Job A bỏ qua và không đẩy lại lên Kafka. Khi DataAsOf đã mới hơn, Job A tiếp tục kiểm tra xem link_id có tồn tại trong free_flow_speed_lookup hay không:

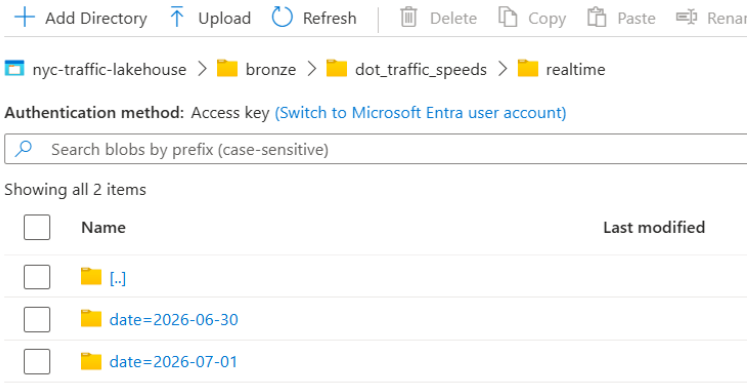
- Nếu link không có trong lookup, Job A xử lý như trường hợp lỗi cảm biến.

- Nếu link có trong lookup, Job A tính speed_ratio bằng cách chia speed cho free_flow_speed lấy từ bộ nhớ, sau đó xác định current_congestion theo ba ngưỡng (≥0,67; 0,40–0,67; <0,40), rồi đóng gói thành message và đẩy lên Kafka với key là link_id.

```
NYCTraffic_Lakehouse > job-a > collector.py > run_collector
288 def run_collector():
318         # CASE 1 - link chết
319         if data_as_of < dead_cutoff:
320             continue
321
322         # CASE 2 - sensor lỗi tạm thời
323         if speed is None or speed <= 0 or speed >= 100:
324             continue
325
326         # CASE 3 - data hợp lệ
327         prev_daof = last_pushed_data_as_of.get(link_id)
328         if prev_daof == data_as_of.isoformat():
329             continue  # DataAsOf không đổi, skip
330
331         if link_id not in free_flow_speed_lookup:
332             logger.warning(f"Link {link_id} không có trong free_flow_speed_lookup, bỏ qua")
333             continue
334
335         speed_ratio, current_congestion = compute_speed_ratio_congestion(speed, link_id)
336         if speed_ratio is None:
337             continue
338
339         try:
340             push_to_kafka(link_id, data_as_of, current_congestion, speed_ratio, link_name)
341         except Exception as e:
342             logger.warning(f"Push Kafka fail cho link {link_id}, bỏ qua: {e}")
343             continue
344
345         last_pushed_data_as_of[link_id] = data_as_of.isoformat()
346         pushed_count += 1
347
348         bronze_records.append({
349             "link_id": link_id,
350             "speed": speed,
351             "data_as_of": data_as_of,
352             "borough": free_flow_speed_lookup[link_id]["borough"],
353             "link_name": link_name
354         })
```

Hình 4.54: Job A - trường hợp xử lý

Job A cũng ghi lại snapshot thô ra tầng Bronze realtime trên Azure Blob Storage, theo cấu trúc thư mục phân vùng theo ngày và giờ (date=/hour=).



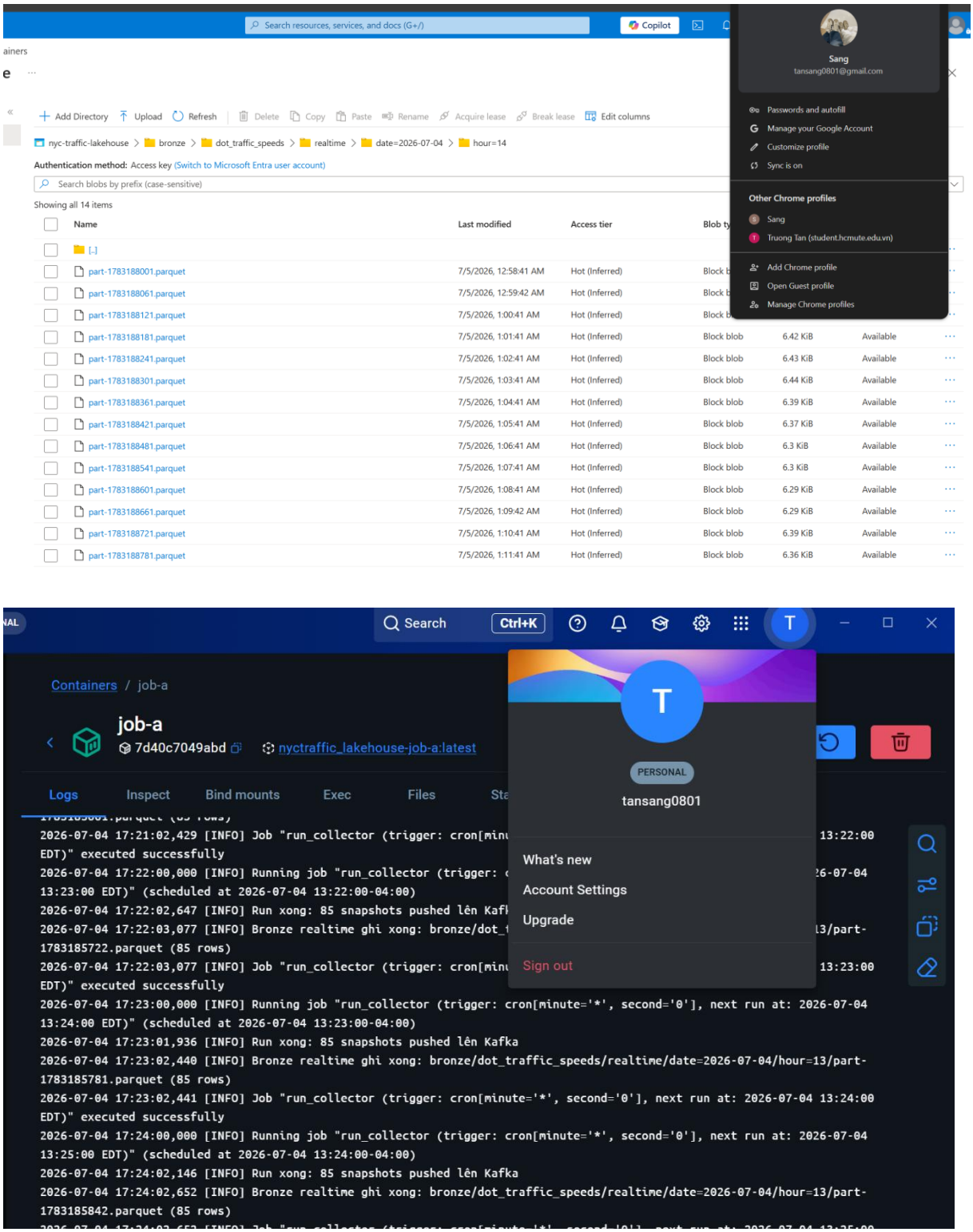
Hình 4.55: Bronze realtime trên Azure Blob Storage

Xem log Job A

Mở log job-a:

docker compose logs -f job-a

```
✔Container job-a                    Started                                                    7.5
PS D:\HK II Year3 (Second)\Big data\DoAn_BigData_NYCTraffic-Lakehouse\NYCTraffic_Lakehouse> docker compose logs -f job-a
job-a  | 2026-07-03 14:11:22,502 [INFO] Job A starting...
job-a  | 2026-07-03 14:11:25,117 [INFO] Loaded active_link_ids: 125 links
job-a  | 2026-07-03 14:11:25,584 [INFO] Loaded free_flow_speed_lookup: 116 links
job-a  | 2026-07-03 14:11:25,587 [INFO] Adding job tentatively -- it will be properly scheduled when the scheduler starts
job-a  | 2026-07-03 14:11:25,587 [INFO] Scheduler bắt đầu (mỗi 1 phút)...
job-a  | 2026-07-03 14:11:25,587 [INFO] Added job "run_collector" to job store "default"
job-a  | 2026-07-03 14:11:25,587 [INFO] Scheduler started
job-a  | 2026-07-03 14:12:00,001 [INFO] Running job "run_collector (trigger: cron[minute='*', second='0'], next run at: 2026-
7-03 10:12:00 EDT)" (scheduled at 2026-07-03 10:12:00-04:00)
job-a  | 2026-07-03 14:12:00,069 [WARNING] Không đếm được distinct link trong Kafka: KafkaError{code=_INVALID_ARG,val=-186,st
="Failed to get offsets: Local: Invalid argument or configuration"}
job-a  | 2026-07-03 14:12:00,070 [WARNING] Kafka chỉ có 0 distinct links (<20) - kiểm tra warm-up
job-a  | 2026-07-03 14:12:01,405 [INFO] Warm-up xong: pushed 0 snapshots lên Kafka
job-a  | 2026-07-03 14:12:03,422 [INFO] Run xong: 90 snapshots pushed lên Kafka
job-a  | 2026-07-03 14:12:03,894 [INFO] Bronze realtime ghi xong: bronze/dot_traffic_speeds/realtime/date=2026-07-03/hour=10/
art-1783087923.parquet (90 rows)
job-a  | 2026-07-03 14:12:03,894 [INFO] Job "run_collector (trigger: cron[minute='*', second='0'], next run at: 2026-07-03 10
13:00 EDT)" executed successfully
```



Hình 4.56 : Job A đã khởi động thành công và ghi dữ liệu vào tầng bronze

Job A chạy mỗi 1 phút, cần thấy các nội dung kiểu như:

Loaded free_flow_speed_lookup, Loaded active_link_ids, Kafka Producer initialized, Fetching NYC DOT realtime feed, Valid snapshots pushed to Kafka, Appended, realtime bronze parquet.

4.5.3. Cơ chế warm-up dữ liệu Kafka

Dựa trên các đặc trưng trễ với yêu cầu tối thiểu 60 phút lịch sử trong Kafka, hệ thống cần một cơ chế xử lý cho tình huống Kafka vừa được khởi động lại hoặc mất dữ liệu, khi đó không có đủ lịch sử để tính lag ngay lập tức. Trước mỗi lần đẩy snapshot mới, Job A kiểm tra số lượng link_id phân biệt có message trong 90 phút gần nhất trên Kafka. Nếu số lượng này đủ lớn, chứng tỏ Kafka đang có dữ liệu ổn định, Job A bỏ qua bước warm-up và tiếp tục chạy như bình thường.

Ngược lại, nếu số lượng link_id trong Kafka thấp hơn ngưỡng kiểm tra, hệ thống xem xét hai khả năng:

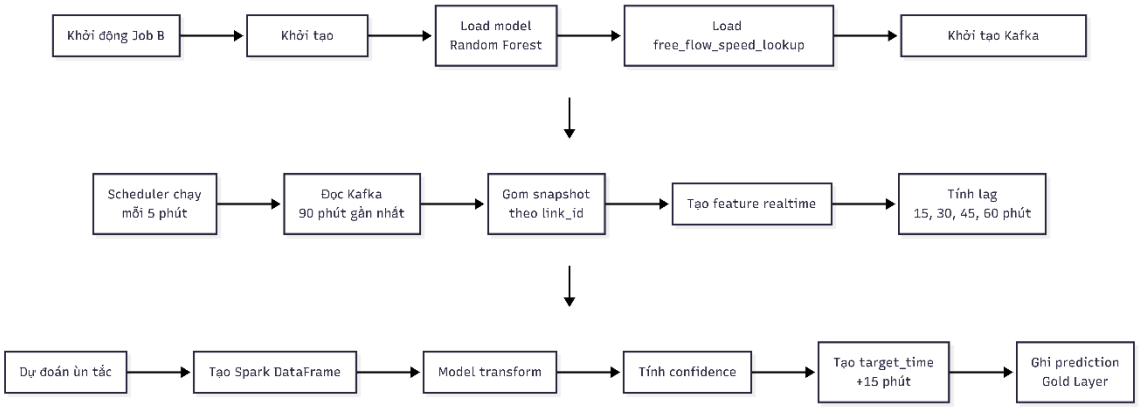
Khả năng thứ nhất là đây là lần đầu tiên hệ thống được triển khai và chưa từng có dữ liệu Bronze realtime nào được ghi trên Azure Blob Storage. Trong trường hợp này, Job A không thể khôi phục lịch sử vì không có nguồn nào để lấy lại, nên bỏ qua bước warm-up. Đây được xem là hành vi kỳ vọng của hệ thống mới triển khai: Job B sẽ tự động bỏ qua toàn bộ các link trong khoảng 60–70 phút đầu tiên cho đến khi tích lũy đủ lịch sử thực tế trong Kafka.

Khả năng thứ hai là Kafka bị mất dữ liệu tạm thời (ví dụ do container Kafka khởi động lại) trong khi Azure Blob Storage vẫn còn lưu trữ dữ liệu Bronze realtime trước đó. Trong trường hợp này, Job A đọc hai phân vùng liên tiếp gần nhất (giờ hiện tại và giờ trước đó, hoặc phân vùng cuối cùng của ngày hôm trước nếu đang ở giờ 0), lọc lấy các bản ghi có DataAsOf nằm trong 90 phút gần nhất, sau đó tính lại speed_ratio và current_congestion cho từng snapshot dựa trên free_flow_speed_lookup đang có sẵn trong bộ nhớ, rồi đẩy toàn bộ các snapshot này lên Kafka theo đúng cấu trúc message thông thường (đủ bốn trường data_as_of, current_congestion, speed_ratio, link_name) để đảm bảo tính nhất quán với luồng dữ liệu bình thường.

```python
warmup_attempted = False  # global, set True ngay sau lần thử đầu tiên (thành công hay không)
def maybe_warmup():
    global warmup_attempted
    if warmup_attempted:
        return   # đã thử 1 lần trong vòng đời container này, không lặp lại

    distinct_count = count_distinct_links_in_kafka_90min()
    if distinct_count >= WARMUP_MIN_LINKS:
        warmup_attempted = True
        return

    logger.warning(f"Kafka chỉ có {distinct_count} distinct links (<{WARMUP_MIN_LINKS}) - kiểm tra warm-up")

    container_client = blob_service.get_container_client(AZURE_CONTAINER)
    has_bronze_realtime = any(container_client.list_blobs(
        name_starts_with="bronze/dot_traffic_speeds/realtime/", results_per_page=1
    ))

    if not has_bronze_realtime:
        logger.warning("Bronze realtime trống (lần đầu deploy) - bỏ qua warm-up, hệ thống sẽ tự tích lũy data")
        warmup_attempted = True
        return

    warmup_from_bronze_realtime()
    warmup_attempted = True   # dù push được hay không, cũng không thử lại nữa
```

Hình 4.57: Cơ chế warm-up dữ liệu Kafka

4.5.4. Cơ chế hoạt động của Job B – Predictor



Job B chịu trách nhiệm đọc dữ liệu từ Kafka, tính toán đầy đủ các đặc trưng cần thiết và đưa vào mô hình đã huấn luyện để sinh ra dự đoán cho 15 phút tiếp theo. Khi container khởi động, Job B thực hiện một loạt bước nạp tài nguyên một lần duy nhất và giữ trong bộ nhớ suốt vòng đời container: khởi tạo SparkSession với master trỏ đến spark://spark-master:7077 để kết nối vào Spark Standalone cluster, nạp PipelineModel đã huấn luyện từ Azure Blob Storage, nạp free_flow_speed_lookup, đồng thời khởi tạo Kafka Consumer với group.id là job-b-predictor.

```python
# INIT: SparkSession
def init_spark():
    global spark
    spark = (SparkSession.builder
        .appName("Job_B_Predictor")
        .master("spark://spark-master:7077")
        .config("spark.sql.session.timeZone", "America/New_York")
        .config("spark.pyspark.python", "/usr/bin/python3")
        .config("spark.executorEnv.PYSPARK_PYTHON", "/usr/bin/python3")
        .config("spark.jars.packages", "org.apache.hadoop:hadoop-azure:3.3.4")
        .config(f"spark.hadoop.fs.azure.account.key.{ACCOUNT_NAME}.blob.core.windows.net", ACCOUNT_KEY)
        .config("spark.executor.memory", "2g")
        .config("spark.driver.memory", "1g")
        .getOrCreate())
    spark.sparkContext.setLogLevel("WARN")
    logger.info(f"Spark version: {spark.version}, master: {spark.sparkContext.master}")
```

```python
# INIT: load PipelineModel + free_flow_speed_lookup từ Azure
def load_model_and_lookup():
    global model, free_flow_speed_lookup

    model = PipelineModel.load(MODEL_PATH)
    logger.info(f"Loaded PipelineModel from {MODEL_PATH}")

    blob = blob_service.get_blob_client(container=CONTAINER, blob="artifacts/free_flow_speed_lookup.parquet")
    data = blob.download_blob().readall()
    df = pd.read_parquet(io.BytesIO(data))

    for _, row in df.iterrows():
        free_flow_speed_lookup[str(row["link_id"])] = {
            "free_flow_speed": float(row["free_flow_speed"]),
            "borough": row["borough"],
            "link_name": row["link_name"]
        }
    logger.info(f"Loaded free_flow_speed_lookup: {len(free_flow_speed_lookup)} links")
```

```python
# INIT: Kafka Consumer (giữ trong memory suốt vòng đời container)
def init_kafka_consumer():
    global kafka_consumer
    kafka_consumer = Consumer({
        "bootstrap.servers": KAFKA_BOOTSTRAP,
        "group.id": "job-b-predictor",
        "auto.offset.reset": "latest",
        "enable.auto.commit": False
    })
    logger.info("Kafka Consumer initialized")
```

Hình 4.58: Cơ chế hoạt động của Job B

Job B được lên lịch chạy mỗi 5 phút bằng APScheduler ở chế độ BlockingScheduler, căn theo mốc giờ hệ thống (00:00, 00:05, 00:10...). Trong mỗi lần chạy, Job B trước tiên truy vấn Kafka để lấy toàn bộ message trong 90 phút gần nhất. Việc đọc Kafka được thực hiện thủ công bằng confluent-kafka Consumer: assign trực tiếp các partition của topic, seek về offset tương ứng với mốc thời gian hiện tại trừ 90 phút, sau đó đọc liên tục cho đến offset hiện tại rồi dừng, giữ nguyên Consumer object trong bộ nhớ cho lần chạy kế tiếp.

```python
# Entry point
if __name__ == "__main__":
    logger.info("Job B starting...")
    init_spark()
    load_model_and_lookup()
    init_kafka_consumer()

    scheduler = BlockingScheduler(timezone="America/New_York")
    scheduler.add_job(
        run_predictor, "cron", minute="*/5", second=0,
        max_instances=1,
        misfire_grace_time=60
    )

    logger.info("Scheduler bắt đầu (mỗi 5 phút)...")
    try:
        scheduler.start()
    except (KeyboardInterrupt, SystemExit):
        logger.info("Job B dừng.")
        kafka_consumer.close()
        spark.stop()
```

Hình 4.59: Chạy job B

Sau khi có dữ liệu, với mỗi link_id xuất hiện trong tập message vừa lấy được, Job B lấy snapshot mới nhất của link đó để xác định current_congestion và speed_ratio hiện tại. Đồng thời, các đặc trưng thời gian gồm hour, day_of_week, month, is_weekend, is_holiday được tính từ chính timestamp của snapshot mới nhất, không phải từ thời điểm Job B đang chạy. Việc chuyển đổi timezone được thực hiện bằng thư viện pytz sang múi giờ America/New_York, tự động xử lý DST (giờ mùa hè/mùa đông). Riêng day_of_week được quy đổi từ hàm weekday() của Python (0 = Thứ Hai) sang quy ước của Spark (1 = Chủ Nhật, 7 = Thứ Bảy) thông qua bảng ánh xạ cố định.

```python
# Tính time features từ timestamp (mapping Spark convention)
def compute_time_features(dt_et: datetime):
    hour = dt_et.hour
    day_of_week = WEEKDAY_TO_SPARK_DOW[dt_et.weekday()]
    month = dt_et.month
    is_weekend = 1 if day_of_week in (1, 7) else 0
    date_str = dt_et.strftime("%Y-%m-%d")
    is_holiday = 1 if date_str in holiday_dates else 0
    return hour, day_of_week, month, is_weekend, is_holiday
```

Hình 4.60: Job B tính các đặc trưng

Xem log Job B:

Chạy:

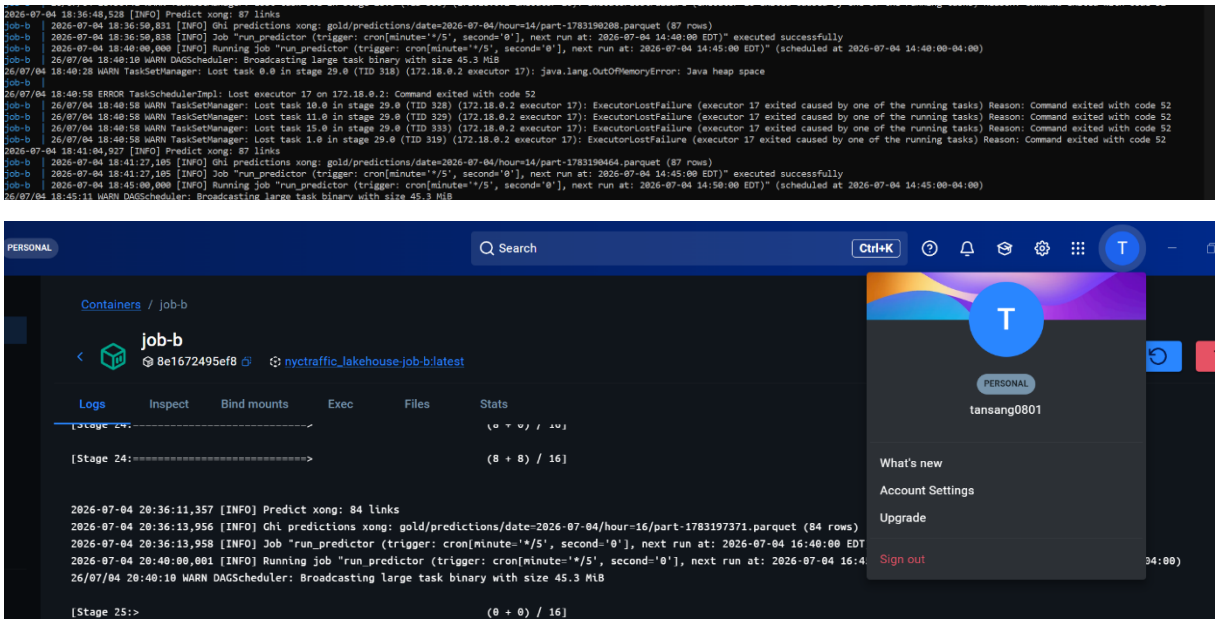
docker compose logs -f job-b

Scheduler bắt đầu mỗi 5 phút

Trong khoảng đầu, Job B có thể log:

Insufficient history, skipping all links - warming up

Đây là bình thường, không phải lỗi, vì Job B cần đủ lịch sử Kafka để tính 4 mốc lag:15 phút, 30 phút, 45 phút, 60 phút



Hình 4.61 : Job B – Predictor khi pipeline realtime đã bắt đầu dự đoán thành công

4.5.5. Cơ chế tìm lag features từ dữ liệu Kafka

Khác với giai đoạn huấn luyện, nơi các đặc trưng trễ được tính trực tiếp bằng hàm lag() trên DataFrame đã sắp xếp theo thời gian, trong giai đoạn dự đoán Job B phải tự tìm kiếm snapshot phù hợp trong tập message rời rạc lấy về từ Kafka, vốn không đảm bảo có đúng một snapshot tại mỗi mốc 5 phút do độ trễ hoặc gián đoạn của nguồn dữ liệu real-time.

Đối với mỗi mốc lag (15, 30, 45, 60 phút), Job B áp dụng chiến lược tìm kiếm hai bước. Bước ưu tiên thứ nhất là tìm một snapshot nằm trong khoảng ±5 phút quanh mốc thời gian mong muốn - ví dụ với mốc 15 phút, Job B tìm snapshot có thời gian nằm trong khoảng 10 đến 20 phút trước thời điểm hiện tại; nếu có nhiều snapshot thỏa mãn, Job B chọn snapshot gần mốc 15 phút nhất. Các đặc trưng past_speed_ratio tương ứng được lấy từ cùng snapshot với past_congestion cùng mốc, không cần tìm kiếm riêng.

Nếu không tìm được snapshot nào trong khoảng ±5 phút nêu trên, Job B chuyển sang bước ưu tiên thứ hai, áp dụng nguyên tắc LOCF (Last Observation Carried Forward): lấy snapshot hợp lệ gần nhất trước mốc thời gian đó, không giới hạn phạm vi tìm kiếm. Tuy nhiên, nếu khoảng cách giữa snapshot tìm được và mốc mong muốn vượt quá 15 phút, Job B coi mốc lag đó là bị thiếu (missing), vì khoảng cách quá xa sẽ khiến giá trị lag không còn phản ánh đúng trạng thái giao thông tại thời điểm cần thiết, có thể gây sai lệch lớn cho dự đoán.

```python
# Tìm lag feature: Ưu tiên 1 (±5 phút quanh mốc) -> Ưu tiên 2 (LOCF)
def find_lag_snapshot(snapshots: list, ref_time: datetime, target_minutes: int, lo: int, hi: int):
    # snapshots đã sort theo data_as_of tăng dần. Trả về snapshot dict hoặc None nếu MISSING
    target_time = ref_time - timedelta(minutes=target_minutes)

    window_start = ref_time - timedelta(minutes=hi)
    window_end = ref_time - timedelta(minutes=lo)

    candidates = [s for s in snapshots if window_start <= s["data_as_of"] <= window_end]
    if candidates:
        best = min(candidates, key=lambda s: abs((s["data_as_of"] - target_time).total_seconds()))
        return best

    # LOCF: snapshot hợp lệ gần nhất TRƯỚC mốc target_time
    before = [s for s in snapshots if s["data_as_of"] <= target_time]
    if not before:
        return None  # MISSING

    locf = max(before, key=lambda s: s["data_as_of"])
    gap_minutes = (target_time - locf["data_as_of"]).total_seconds() / 60
    if gap_minutes > 15:
        return None  # MISSING

    return locf
```

Hình 4.62: Tìm lag features từ dữ liệu Kafka

4.5.6. Cơ chế bỏ qua các link_id không đủ dữ liệu

Vì mô hình yêu cầu đầy đủ cả 19 đặc trưng đầu vào, trong đó có 4 mốc lag congestion và 4 mốc lag speed_ratio, nên nếu bất kỳ một trong bốn mốc lag (15, 30, 45, 60 phút) bị xác định là missing theo cơ chế đã trình bày, Job B sẽ bỏ qua việc dự đoán cho link_id đó trong lần chạy hiện tại, không ghi ra bất kỳ output nào cho link này.

Bên cạnh cơ chế bỏ qua do thiếu lag feature, hệ thống còn có hai trường hợp bỏ qua khác đã đề cập ở các mục trước: bỏ qua do link_id không tồn tại trong StringIndexer của PipelineModel, và việc 9 trong số 125 link active không bao giờ có snapshot trên Kafka do không có entry trong free_flow_speed_lookup, nên các link này sẽ không bao giờ xuất hiện trong kết quả dự đoán của hệ thống hiện tại.

```python
# Build feature row cho 1 link. Trả về None nếu thiếu lag (MISSING) hoặc link không có lookup.
def build_feature_row(link_id: str, snapshots: list):
    if link_id not in free_flow_speed_lookup:
        return None

    latest = max(snapshots, key=lambda s: s["data_as_of"])
    current_congestion = latest["current_congestion"]
    speed_ratio = latest["speed_ratio"]
    data_as_of = latest["data_as_of"]

    hour, day_of_week, month, is_weekend, is_holiday = compute_time_features(data_as_of)

    lag_values = {}
    for minutes, (lo, hi) in LAG_SPECS.items():
        snap = find_lag_snapshot(snapshots, data_as_of, minutes, lo, hi)
        if snap is None:
            return None  # MISSING -> bỏ qua link này
        lag_values[minutes] = snap

    past_congestion_15min = lag_values[15]["current_congestion"]
    past_speed_ratio_15min = lag_values[15]["speed_ratio"]

    congestion_trend = current_congestion - past_congestion_15min
    speed_ratio_trend = speed_ratio - past_speed_ratio_15min

    borough = free_flow_speed_lookup[link_id]["borough"]
    link_name = latest.get("link_name") or free_flow_speed_lookup[link_id].get("link_name")

    return {
        "link_id": link_id,
        "borough": borough,
        "link_name": link_name,
        "speed_ratio": speed_ratio,
        "current_congestion": current_congestion,
        "hour": hour,
        "day_of_week": day_of_week,
        "month": month,
        "is_weekend": is_weekend,
        "is_holiday": is_holiday,
        "past_congestion_15min": lag_values[15]["current_congestion"],
        "past_speed_ratio_15min": lag_values[15]["speed_ratio"],
        "past_congestion_30min": lag_values[30]["current_congestion"],
        "past_speed_ratio_30min": lag_values[30]["speed_ratio"],
        "past_congestion_45min": lag_values[45]["current_congestion"],
        "past_speed_ratio_45min": lag_values[45]["speed_ratio"],
        "past_congestion_60min": lag_values[60]["current_congestion"],
        "past_speed_ratio_60min": lag_values[60]["speed_ratio"],
        "congestion_trend": congestion_trend,
        "speed_ratio_trend": speed_ratio_trend,
        "data_as_of": data_as_of,
    }
```

Hình 4.63: Lọc lag từ dữ liệu Kafka

4.5.7. Ghi kết quả dự đoán vào Gold Layer

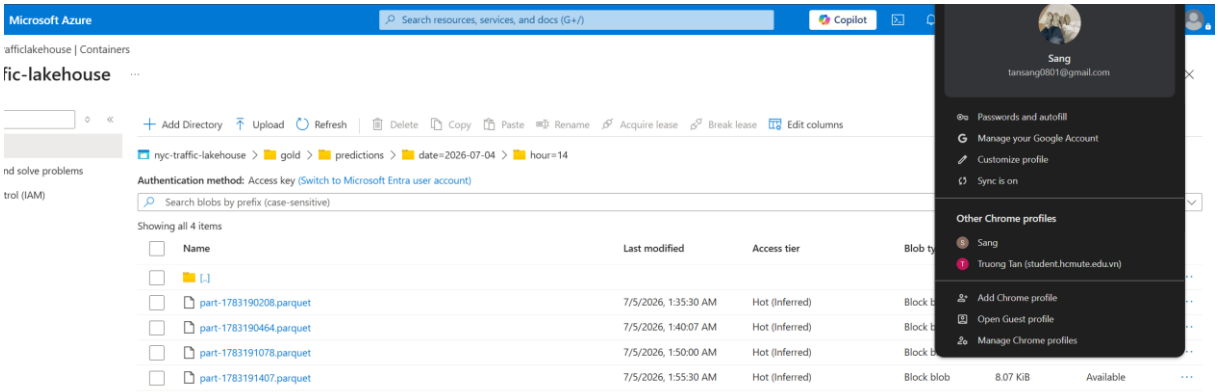
Sau khi hoàn tất tính toán đặc trưng cho các link đủ điều kiện, Job B tính thêm hai đặc trưng xu hướng congestion_trend và speed_ratio_trend dựa trên chênh lệch giữa giá trị hiện tại và giá trị tại mốc 15 phút trước, sau đó gộp toàn bộ 19 đặc trưng thành vector đầu vào và đưa vào PipelineModel để dự đoán. Kết quả trả về gồm nhãn dự đoán predicted_congestion (0, 1 hoặc 2) và giá trị confidence, được lấy là giá trị lớn nhất trong vector xác suất mà RandomForestClassifier sinh ra cho ba lớp.

```python
# Ghi kết quả ra Azure gold/predictions/, retry 1 lần nếu fail
def write_predictions(records: list):
    if not records:
        logger.info("Không có prediction nào để ghi")
        return

    now_et = datetime.now(EASTERN)
    date_str = now_et.strftime("%Y-%m-%d")
    hour_str = now_et.strftime("%H")
    blob_path = f"gold/predictions/date={date_str}/hour={hour_str}/part-{int(time.time())}.parquet"

    df = pd.DataFrame(records)

    for attempt in range(2):
        try:
            buf = io.BytesIO()
            df.to_parquet(buf, index=False)
            buf.seek(0)
            blob = blob_service.get_blob_client(container=CONTAINER, blob=blob_path)
            blob.upload_blob(buf, overwrite=True)
            logger.info(f"Ghi predictions xong: {blob_path} ({len(records)} rows)")
            return
        except Exception as e:
            logger.warning(f"Ghi predictions fail (attempt {attempt+1}/2): {e}")
            time.sleep(1)

    logger.error(f"Ghi predictions fail sau retry, bỏ qua batch: {blob_path}")
```

Hình 4.64: Ghi kết quả dự đoán vào Gold Layer

Kiểm tra trong Azure Blob Storage theo đường dẫn:

gold/predictions/date=/hour=/



Hình 4.65 : Kiểm tra kết quả trên Gold layer

4.6. Kết quả triển khai và kiểm thử hệ thống

4.6.1. Kiểm tra kết quả dự đoán trong Gold Layer

Sau khi hệ thống chạy ổn định qua giai đoạn warm-up, nhóm kiểm tra trực tiếp nội dung các file parquet trong gold/predictions/ để xác nhận tính đúng đắn của dữ liệu output. Mỗi file tương ứng với một lần chạy của Job B (mỗi 5 phút), chứa các dòng dữ liệu với đầy đủ bảy trường link_id, borough, link_name, timestamp, target_time, predicted_congestion và confidence.

nyc-traffic-lakehouse > gold > predictions > date=2026-07-01 > hour=13

Authentication method: Access key (Switch to Microsoft Entra user account)

Search blobs by prefix (case-sensitive) | Only sh

Showing all 3 items

Name	Last modified	Access tier	Blob type	Size
[..]				
part-1782925252.parquet	7/2/2026, 12:01:15 AM	Hot (Inferred)	Block blob	8.18 KiB
part-1782925853.parquet	7/2/2026, 12:11:02 AM	Hot (Inferred)	Block blob	8.21 KiB
part-1782926368.parquet	7/2/2026, 12:19:35 AM	Hot (Inferred)	Block blob	8.2 KiB

link_id	borough	link_name	timestamp	target_time	predicted_congestion	confidence
4362249	Queens	LIE WB LITTLE NECK PKWY - NB CIP	2026-06-30 14:46:02	2026-06-30 15:01:02	1	0.781482
4616298	Queens	VWE N MP4.63 (Exit 6 - Jamaica Ave) - MP6.39 (Exit 11 Jewel Ave)	2026-06-30 14:45:15	2026-06-30 15:00:15	2	0.639122
4616355	Queens	Belt Pkwy E 182nd St - Laurelton Pkwy N @ SSP	2026-06-30 14:45:15	2026-06-30 15:00:15	2	0.970597
4616356	Queens	Belt Pkwy W 182nd St - JFK Expressway	2026-06-30 14:45:15	2026-06-30 15:00:15	1	0.834013
4616357	Queens	Belt Pkwy W JFK Expressway - VWE N Jamaica Ave	2026-06-30 14:45:15	2026-06-30 15:00:15	1	0.674229
4616310	Bronx	CBE AMSTERDAM AVE (L/LVL) - MORRIS AVE	2026-06-30 14:45:15	2026-06-30 15:00:15	2	0.935365
4616309	Bronx	CBE E AMSTERDAM AVE(U/LVL) - MORRIS AVE	2026-06-30 14:45:15	2026-06-30 15:00:15	2	0.919456
4616352	Queens	CIP S Hempstead Tpk - Laurelton Pkwy @ SSP	2026-06-30 14:45:15	2026-06-30 15:00:15	1	0.868126
4616250	Bronx	HRP N LAFAYETTE AVENUE - E TREMONT AVENUE	2026-06-30 14:45:15	2026-06-30 15:00:15	1	0.754591
4616364	Queens	LIE W 108TH ST - 84TH ST	2026-06-30 14:45:15	2026-06-30 15:00:15	1	0.881249

Hình 4.66: Kiểm tra kết quả dự đoán trên Azure Blob Storage

4.6.2. Kiểm tra dashboard Power BI

Dashboard Power BI được xây dựng bằng cách kết nối tới Gold Layer thông qua Python Script. Bảng df_dashboard được tạo bằng cách kết hợp dữ liệu tọa độ đường, dữ liệu dự đoán Gold và lookup tốc độ tự do. Dashboard gồm bản đồ NYC hiển thị bubble theo predicted_congestion, bảng top link tắc nghẽn theo giờ, biểu đồ phân bố mức độ tắc nghẽn theo borough, cùng các bộ lọc theo quận, giờ và ngày.

DOAN_BIGDATA_NYCTRAFFIC-L...
- NYCTraffic_Lakehouse
  - job-a
  - job-b
  - notebooks
  - Power_Bi
    - Code
      - ketnoipowerbivoitanggold.py
      - link_coordinates.py
      - load_free_flow_speed_lookup.py
      - realtime.py
    - Nhom01_PowerBI.pbix
  - spark
    - spark-master.Dockerfile
    - spark-slave.Dockerfile
  - .env
  - docker-compose.yml
  - env_example.txt
  - .gitignore
  - LICENSE
  - README.md

OUTLINE

TIMELINE

NYCTraffic_Lakehouse > Power_Bi > Code > ketnoipowerbivoitanggold.py > ...

```python
import io
import pandas as pd
from azure.storage.blob import BlobServiceClient

# Azure Blob config
CONNECTION_STRING = """"""
CONTAINER_NAME = "nyc-traffic-lakehouse"

# Gold prediction path
# Bảng này có predicted_congestion + confidence
GOLD_PREFIX = "gold/predictions/"

# Test trước cho nhẹ. Nếu OK rồi có thể đổi thành None.
MAX_FILES = 100

# Connect Azure Blob
blob_service = BlobServiceClient.from_connection_string(CONNECTION_STRING)
container_client = blob_service.get_container_client(CONTAINER_NAME)

parquet_blobs = []

for blob in container_client.list_blobs(name_starts_with=GOLD_PREFIX):
    name = blob.name

    if name.endswith(".parquet"):
        parquet_blobs.append(name)

parquet_blobs = sorted(parquet_blobs, reverse=True)

if MAX_FILES is not None:
    parquet_blobs = parquet_blobs[:MAX_FILES]

# Read parquet files
if len(parquet_blobs) == 0:
    df_gold = pd.DataFrame({
        "link_id": [],
        "borough": [],
```

Hình 4.67: Code kết nối với tầng gold

Queries [6]

- df_gold
- df_lookup
- df_coords
- df_realtime
- df_realtime_dashboard
- df_dashboard

```
= Table.TransformColumnTypes(df_gold1,{{"link_id", type text}, {"borough", type text}, {"hour", Int64.Type}, {"day_of_week", Int64.Type},
```

	link_id	borough	link_name	timestamp	target_time	dat
1	4362249	Queens	LIE WB LITTLE NECK PKWY - NB CIP	2026-07-02 04:21:02	2026-07-02 04:36:02	
2	4616356	Queens	Belt Pkwy W 182nd St - JFK Expressway	2026-07-02 04:20:13	2026-07-02 04:35:13	
3	4616364	Queens	LIE W 108TH ST - 84TH ST	2026-07-02 04:20:13	2026-07-02 04:35:13	
4	4616250	Bronx	HRP N LAFAYETTE AVENUE - E TREMONT AVENUE	2026-07-02 04:20:13	2026-07-02 04:35:13	
5	4616354	Queens	Laurelton Pkwy S @ SSP - Belt Pkwy W 182nd St	2026-07-02 04:20:13	2026-07-02 04:35:13	
6	4616341	Manhattan	FDR S Catherine Slip - BKN Bridge Manhattan Side	2026-07-02 04:20:13	2026-07-02 04:35:13	
7	4616193	Staten Island	SIE E BRADLEY AVENUE - CLOVE ROAD	2026-07-02 04:20:13	2026-07-02 04:35:13	
8	4616333	Manhattan	FDR S 25th St - Catherine Slip	2026-07-02 04:20:13	2026-07-02 04:35:13	
9	4616352	Queens	CIP S Hempstead Tpk - Laurelton Pkwy @ SSP	2026-07-02 04:20:13	2026-07-02 04:35:13	
10	4616194	Staten Island	SIE E WOOLEY AVENUE - BRADLEY AVENUE	2026-07-02 04:20:13	2026-07-02 04:35:13	
11	4616353	Queens	Laurelton Pkwy N @ SSP - CIP N Hempstead Tpk	2026-07-02 04:20:13	2026-07-02 04:35:13	
12	4616309	Bronx	CBE E AMSTERDAM AVE(U/LVL) - MORRIS AVE	2026-07-02 04:20:13	2026-07-02 04:35:13	
13	4616310	Bronx	CBE AMSTERDAM AVE (L/LVL) - MORRIS AVE	2026-07-02 04:20:13	2026-07-02 04:35:13	
14	4616357	Queens	Belt Pkwy W JFK Expressway - VWE N Jamaica Ave	2026-07-02 04:20:13	2026-07-02 04:35:13	
15	4616298	Queens	VWE N MP4.63 (Exit 6 - Jamaica Ave) - MP6.39 (Exit 11 Jewel Ave)	2026-07-02 04:20:13	2026-07-02 04:35:13	
16	4616305	Bronx	BWB N Toll Plaza - HRP N Lafayatte Ave	2026-07-02 04:20:13	2026-07-02 04:35:13	
17	4616252	Bronx	BRP N WATSON AVENUE - FORDHAM ROAD	2026-07-02 04:20:13	2026-07-02 04:35:13	
18	4616259	Queens	VWE S MP6.39 (Exit 11 Jewel Ave) - MP4.63 (Exit 6 Jamaica Ave)	2026-07-02 04:20:13	2026-07-02 04:35:13	
19	4616340	Manhattan	BQE N Atlantic Ave - MAN Bridge Manhattan Side	2026-07-02 04:20:13	2026-07-02 04:35:13	
20	4362342	Queens	LIE E 84TH ST - 108TH ST	2026-07-02 04:20:07	2026-07-02 04:35:07	
21	4362250	Queens	CVE NB GCP - WILLETS PT BLVD	2026-07-02 04:20:07	2026-07-02 04:35:07	
22	4616211	Staten Island	SIE W RICHMOND AVENUE - SOUTH AVENUE	2026-07-02 04:20:04	2026-07-02 04:35:04	
23	4616212	Staten Island	WSE N VICTORY BLVD - SOUTH AVENUE	2026-07-02 04:20:04	2026-07-02 04:35:04	
24	4616241	Queens	VWE S MP8.65 (Exit 13 Northern Blvd) - MP6.39 (Exit 11 Jewel Ave)	2026-07-02 04:20:04	2026-07-02 04:35:04	
25	4616300	Queens	VWE N MP8.64 (Exit 13 Northern Blvd) - Whitestone Expwy Exit 14 (Lin...	2026-07-02 04:20:04	2026-07-02 04:35:04	

Hình 4.68: Bảng preview query của df_gold

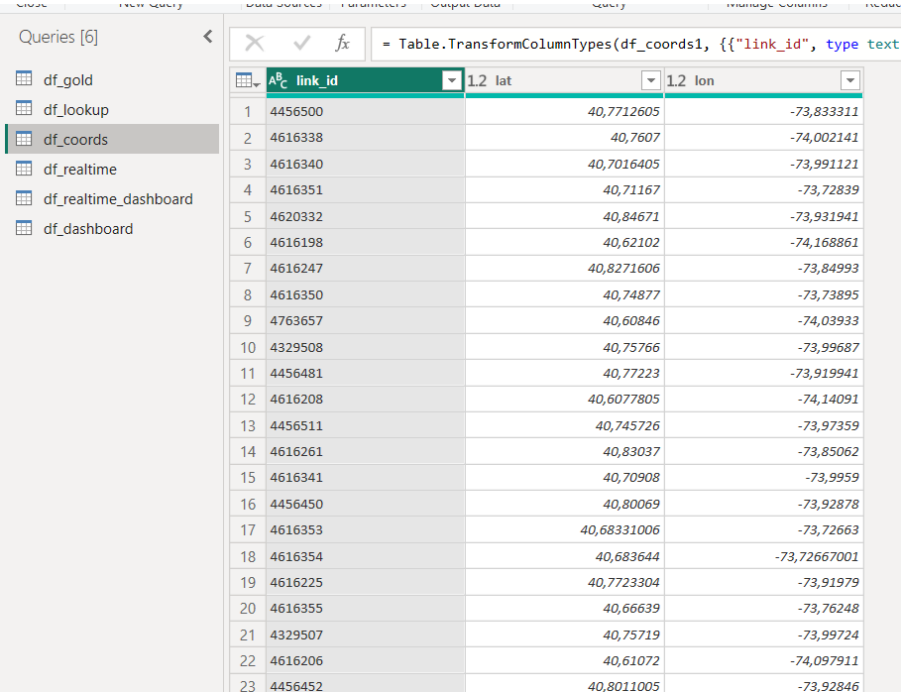
Bảng df_coords được sử dụng để bổ sung tọa độ cho từng link đường. Bảng này chứa các trường như link_id, lat và lon, giúp Power BI xác định vị trí của từng đoạn đường trên bản đồ. Khi tạo bảng df_dashboard, hệ thống thực hiện merge df_coords với df_gold theo khóa link_id bằng kiểu join Left Outer. Việc này giúp mỗi kết quả dự đoán có thêm thông tin tọa độ để hiển thị trên Azure Maps.

DOAN_BIGDATA_NYCTRAFFIC-L...
- NYCTraffic_Lakehouse
  - job-a
  - job-b
  - notebooks
  - Power_Bi
    - Code
      - ketnoipowerbivoitanggold.py
      - link_coordinates.py
      - load_free_flow_speed_lookup.py
      - realtime.py
    - Nhom01_PowerBI.pbix
  - spark
    - spark-master.Dockerfile
    - spark-slave.Dockerfile
  - .env
  - docker-compose.yml
  - env_example.txt
  - .gitignore
  - LICENSE
  - README.md

NYCTraffic_Lakehouse > Power_Bi > Code > link_coordinates.py > ...

```python
import io
import pandas as pd
from azure.storage.blob import BlobServiceClient

# Azure Blob config
CONNECTION_STRING = """"""
CONTAINER_NAME = "nyc-traffic-lakehouse"

# Tìm file link_coordinates.parquet
blob_service = BlobServiceClient.from_connection_string(CONNECTION_STRING)
container_client = blob_service.get_container_client(CONTAINER_NAME)

matched_blobs = []

for blob in container_client.list_blobs(name_starts_with="artifacts/"):
    name = blob.name

    if "link_coordinates" in name and name.endswith(".parquet"):
        matched_blobs.append(name)

if len(matched_blobs) == 0:
    df_coords = pd.DataFrame({
        "error": ["Không tìm thấy file link_coordinates.parquet trong Azure Blob"]
    })
else:
    dfs = []

    for blob_name in matched_blobs:
        blob_client = container_client.get_blob_client(blob_name)
        data = blob_client.download_blob().readall()

        df_part = pd.read_parquet(io.BytesIO(data))
        dfs.append(df_part)
```

Hình 4.69: Code lấy vị trí của đường



Queries [6]

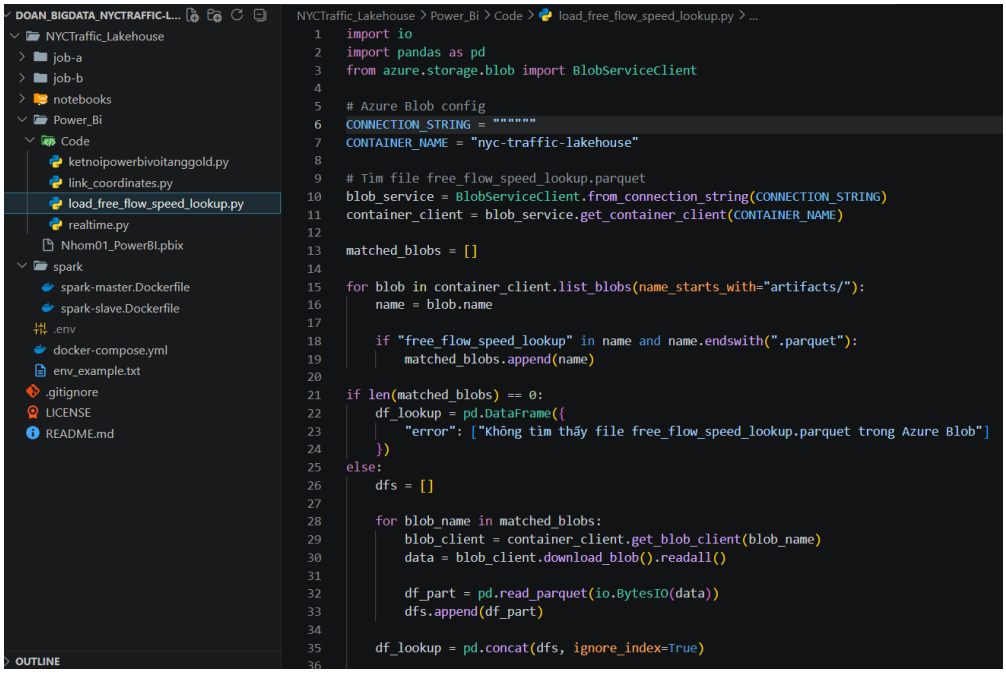
- df_gold
- df_lookup
- df_coords
- df_realtime
- df_realtime_dashboard
- df_dashboard

```
= Table.TransformColumnTypes(df_coords1, {{"link_id", type text
```

	link_id	lat	lon
1	4456500	40,7712605	-73,833311
2	4616338	40,7607	-74,002141
3	4616340	40,7016405	-73,991121
4	4616351	40,71167	-73,72839
5	4620332	40,84671	-73,931941
6	4616198	40,62102	-74,168861
7	4616247	40,8271606	-73,84993
8	4616350	40,74877	-73,73895
9	4763657	40,60846	-74,03933
10	4329508	40,75766	-73,99687
11	4456481	40,77223	-73,919941
12	4616208	40,6077805	-74,14091
13	4456511	40,745726	-73,97359
14	4616261	40,83037	-73,85062
15	4616341	40,70908	-73,9959
16	4456450	40,80069	-73,92878
17	4616353	40,68331006	-73,72663
18	4616354	40,683644	-73,72667001
19	4616225	40,7723304	-73,91979
20	4616355	40,66639	-73,76248
21	4329507	40,75719	-73,99724
22	4616206	40,61072	-74,097911
23	4456452	40,8011005	-73,92846

Hình 4.70: Bảng preview query của df_coords

Bảng df_lookup được dùng để bổ sung thông tin tra cứu của từng link đường, đặc biệt là free_flow_speed và link_name. free_flow_speed là tốc độ tự do được tính từ dữ liệu lịch sử, dùng làm cơ sở so sánh với tốc độ realtime để đánh giá mức độ tắc nghẽn hiện tại. Sau khi merge thêm df_lookup, bảng df_dashboard có đủ thông tin phục vụ bản đồ dự đoán, bảng chi tiết và biểu đồ phân bố theo khu vực.



DOAN_BIGDATA_NYCTRAFFIC-L...
- NYCTraffic_Lakehouse
  - job-a
  - job-b
  - notebooks
  - Power_Bi
    - Code
      - ketnoipowerbivoitanggold.py
      - link_coordinates.py
      - load_free_flow_speed_lookup.py
      - realtime.py
    - Nhom01_PowerBI.pbix
  - spark
    - spark-master.Dockerfile
    - spark-slave.Dockerfile
  - .env
  - docker-compose.yml
  - env_example.txt
- .gitignore
- LICENSE
- README.md

OUTLINE

NYCTraffic_Lakehouse > Power_Bi > Code > load_free_flow_speed_lookup.py > ...

```python
import io
import pandas as pd
from azure.storage.blob import BlobServiceClient

# Azure Blob config
CONNECTION_STRING = """"""
CONTAINER_NAME = "nyc-traffic-lakehouse"

# Tìm file free_flow_speed_lookup.parquet
blob_service = BlobServiceClient.from_connection_string(CONNECTION_STRING)
container_client = blob_service.get_container_client(CONTAINER_NAME)

matched_blobs = []

for blob in container_client.list_blobs(name_starts_with="artifacts/"):
    name = blob.name

    if "free_flow_speed_lookup" in name and name.endswith(".parquet"):
        matched_blobs.append(name)

if len(matched_blobs) == 0:
    df_lookup = pd.DataFrame({
        "error": ["Không tìm thấy file free_flow_speed_lookup.parquet trong Azure Blob"]
    })
else:
    dfs = []

    for blob_name in matched_blobs:
        blob_client = container_client.get_blob_client(blob_name)
        data = blob_client.download_blob().readall()

        df_part = pd.read_parquet(io.BytesIO(data))
        dfs.append(df_part)

    df_lookup = pd.concat(dfs, ignore_index=True)
```

Hình 4.71: Lấy tốc độ và tên của đường

Queries [6]

- df_gold
- df_lookup
- df_coords
- df_realtime
- df_realtime_dashboard
- df_dashboard

= Source{[Name="df_lookup"]}[Value]

	link_id	free_flow_speed	borough	link_name
1	4362251	53.43	Queens	LIE WB LITTLE NECK PKWY - NB CVE
2	4616192	58.4	Staten Island	SIE E CLOVE ROAD - FINGERBOARD ROAD
3	4616310	48.46	Bronx	CBE AMSTERDAM AVE (L/LVL) - MORRIS AVE
4	4616246	59.03	Bronx	BE N STRATFORD AVENUE - CASTLE HILL AVE
5	4616241	55.3	Queens	VWE S MP8.65 (Exit 13 Northern Blvd) - MP6.39 (Exit 11 Jewel Ave)
6	4456500	50.95	Queens	Whitestone Expwy N Exit 14 (Linden Pl) - BWB N Queens Anchorage
7	4620332	47.22	Bronx	CBE W MORRIS AVE - GWB W AMSTERDAM AVE (L/LVL)
8	4616340	11.18	Manhattan	BQE N Atlantic Ave - MAN Bridge Manhattan Side
9	4616351	57.16	Queens	CIP N Hempstead Tpk - LIE
10	4456505	54.05	Queens	TNB N Queens Anchorage - Toll Plaza
11	4616199	60.89	Staten Island	SIE W - WSE S SOUTH AVENUE - SOUTH AVENUE
12	4616320	56.54	Queens	CIP N TNB - Whitestone Expwy S Exit 14 (Linden Pl)
13	4456510	37.28	Queens	QMT W Toll Plaza - Manhattan Side
14	4616213	61.51	Staten Island	WSE N ARDEN AVENUE - VICTORY BLVD
15	4616215	56.54	Staten Island	WSE N TYRELLAN AVENUE - BLOOMINGDALE ROAD
16	4456501	44.73	Manhattan	BBT W Toll Plaza - Manhattan Portal
17	4616259	41.63	Queens	VWE S MP6.39 (Exit 11 Jewel Ave) - MP4.63 (Exit 6 Jamaica Ave)
18	4616197	57.16	Staten Island	SIE E SOUTH AVENUE - RICHMOND AVENUE
19	4575278	56.54	Bronx	MDE N VAN CORTLAND PARK - NYST N EXIT 1 (MP.48)
20	4616276	26.09	Queens	VWE S MP2.66 (Exit 2 Rockaway Blvd) - Belt Pkwy E 182nd St
21	4616200	57.78	Staten Island	WSE S SOUTH AVENUE - VICTORY BOULEVARD
22	4616357	15.53	Queens	Belt Pkwy W JFK Expressway - VWE N Jamaica Ave
23	4616356	47.84	Queens	Belt Pkwy W 182nd St - JFK Expressway
24	4616226	50.95	Brooklyn	GOW S 7TH AVENUE - 92ND STREET
25	4616305	62.13	Bronx	BWB N Toll Plaza - HRP N Lafayatte Ave
26	4620298	46.6	Manhattan	GWB E LOWER LEVEL PLAZA - CBE E LOWER LEVEL AMSTERDAM AVE
27	4329473	31.68	Manhattan	LINCOLN TUNNEL E CENTER TUBE NJ - NY

Hình 4.72: Bảng preview query của df_lookup

Ngoài dữ liệu dự đoán, dashboard còn có phần realtime. Bảng df_realtime được lấy từ API realtime của NYC Open Data, gồm các thông tin như link_id, speed, travel_time, data_as_of, link_name và borough.

EXPLORER

DOAN_BIGDATA_NYCTRAFFIC-L...

- NYCTraffic_Lakehouse
  - job-a
  - job-b
  - notebooks
  - Power_Bi
    - Code
      - ketnoipowerbivoitanggold.py
      - link_coordinates.py
      - load_free_flow_speed_lookup.py
      - realtime.py
    - Nhom01_PowerBI.pbix
  - spark
    - spark-master.Dockerfile
    - spark-slave.Dockerfile
  - .env
  - docker-compose.yml
  - env_example.txt
  - .gitignore
  - LICENSE
  - README.md

OUTLINE

.env | collector.py | ketnoipowerbivoitanggold.py | link_coordinates.py | load_free_flow_s

NYCTraffic_Lakehouse > Power_Bi > Code > realtime.py > ...

```python
# NYC DOT real-time feed (linkdata.nyctmc.org) — TSV, <2 phút
# 13 cột: id, Speed, TravelTime, Status, DataAsOf, linkId,
# linkPoints, EncodedPolyLine, EncodedPolyLineLvls, Owner,
# Transcom_id, Borough, linkName
REALTIME_URL = "https://linkdata.nyctmc.org/data/LinkSpeedQuery.txt"

col_names = [
    "id", "Speed", "TravelTime", "Status", "DataAsOf", "linkId",
    "linkPoints", "EncodedPolyLine", "EncodedPolyLineLvls", "Owner",
    "Transcom_id", "Borough", "linkName"
]

needed_cols = ["link_id", "link_name", "borough", "speed", "data_as_of"]

# Load active_link_ids — nếu Azure lỗi thì bỏ qua filter, không crash
active_link_ids = None
try:
    blob_service = BlobServiceClient.from_connection_string(CONNECTION_STRING)
    container_client = blob_service.get_container_client(CONTAINER_NAME)

    for blob in container_client.list_blobs(name_starts_with="artifacts/"):
        if "active_link_ids" in blob.name and blob.name.endswith(".json"):
            blob_client = container_client.get_blob_client(blob.name)
            data = blob_client.download_blob().readall()
            import json
            active_link_ids = set(str(x) for x in json.loads(data))
            break
except Exception:
    active_link_ids = None  # Azure lỗi thì vẫn chạy tiếp, chỉ là không filter active link

try:
    df_realtime = pd.read_csv(
        REALTIME_URL,
        sep="\t",
        header=None,
```

Hình 4.73: Code lấy data realtime

Sau đó, bảng này được merge với df_coords để lấy tọa độ và merge với df_lookup để lấy free_flow_speed, tạo thành bảng df_realtime_dashboard.

Queries [6]

- df_gold
- df_lookup
- df_coords
- df_realtime
- df_realtime_dashboard
- df_dashboard

```
= Table.TransformColumnTypes(df_realtime1,{{"link_id", type text}})
```

	link_id	link_name	borough	speed	data_as_of
1	4362249	LIE WB LITTLE NECK PKWY - NB CIP	Queens	41.01	2026-07-02 09:44:02-04:00
2	4616250	HRP N LAFAYETTE AVENUE - E TREMONT AVENUE	Bronx	24.23	2026-07-02 09:43:16-04:00
3	4616276	VWE S MP2.66 (Exit 2 Rockaway Blvd) - Belt Pkwy E 182nd St	Queens	6.21	2026-07-02 09:43:16-04:00
4	4616353	Laurelton Pkwy N @ SSP - CIP N Hempstead Tpk	Queens	7.45	2026-07-02 09:43:16-04:00
5	4616364	LIE W 108TH ST - 84TH ST	Queens	34.79	2026-07-02 09:43:16-04:00
6	4616341	FDR S Catherine Slip - BKN Bridge Manhattan Side	Manhattan	22.99	2026-07-02 09:43:16-04:00
7	4616333	FDR S 25th St - Catherine Slip	Manhattan	22.99	2026-07-02 09:43:16-04:00
8	4616328	FDR N Catherine Slip - 25th St	Manhattan	31.68	2026-07-02 09:43:16-04:00
9	4616352	CIP S Hempstead Tpk - Laurelton Pkwy @ SSP	Queens	27.34	2026-07-02 09:43:16-04:00
10	4616298	VWE N MP4.63 (Exit 6 - Jamaica Ave) - MP6.39 (Exit 11 Jewel Ave)	Queens	19.26	2026-07-02 09:43:16-04:00
11	4616309	CBE E AMSTERDAM AVE(U/LVL) - MORRIS AVE	Bronx	23.61	2026-07-02 09:43:16-04:00
12	4616310	CBE AMSTERDAM AVE (L/LVL) - MORRIS AVE	Bronx	23.61	2026-07-02 09:43:16-04:00
13	4616357	Belt Pkwy W JFK Expressway - VWE N Jamaica Ave	Queens	8.07	2026-07-02 09:43:16-04:00
14	4616356	Belt Pkwy W 182nd St - JFK Expressway	Queens	27.34	2026-07-02 09:43:16-04:00
15	4616355	Belt Pkwy E 182nd St - Laurelton Pkwy N @ SSP	Queens	6.21	2026-07-02 09:43:16-04:00
16	4616354	Laurelton Pkwy S @ SSP - Belt Pkwy W 182nd St	Queens	27.34	2026-07-02 09:43:16-04:00
17	4616272	VWE S MP4.63 (Exit 6 Jamaica Ave) - MP2.66 (Exit 2 Roackaway Blvd)	Queens	6.21	2026-07-02 09:43:16-04:00
18	4616340	BQE N Atlantic Ave - MAN Bridge Manhattan Side	Manhattan	4.97	2026-07-02 09:43:16-04:00
19	4616259	VWE S MP6.39 (Exit 11 Jewel Ave) - MP4.63 (Exit 6 Jamaica Ave)	Queens	8.69	2026-07-02 09:43:16-04:00
20	4616252	BRP N WATSON AVENUE - FORDHAM ROAD	Bronx	32.31	2026-07-02 09:43:16-04:00
21	4616342	BKN Bridge Manhattan Side - FDR N Catherine Slip	Manhattan	31.68	2026-07-02 09:43:16-04:00
22	4616305	BWB N Toll Plaza - HRP N Lafayatte Ave	Bronx	24.23	2026-07-02 09:43:16-04:00
23	4620332	CBE W MORRIS AVE - GWB W AMSTERDAM AVE (L/LVL)	Bronx	8.07	2026-07-02 09:43:09-04:00
24	4620331	CBE W MORRIS AVE - GWB W AMSTERDAM AVE (U/LVL)	Bronx	6.83	2026-07-02 09:43:09-04:00
25	4362250	CVE NB GCP - WILLETS PT BLVD	Queens	40.38	2026-07-02 09:43:07-04:00
26	4362342	LIE E 84TH ST - 108TH ST	Queens	34.79	2026-07-02 09:43:07-04:00
27	4620330	MDE S HARLEM RIVER PARK - GWB W AMSTERDAM AVENUE LOWER L...	Bronx	4.97	2026-07-02 09:43:05-04:00
28	4329507	LINCOLN TUNNEL W NORTH TUBE NY - NJ	Manhattan	22.36	2026-07-02 09:43:05-04:00

Queries [6]

- df_gold
- df_lookup
- df_coords
- df_realtime
- df_realtime_dashboard
- df_dashboard

```
= Table.ExpandTableColumn(#"Merged Queries", "df_lookup", {"free_flow_speed"}, {"free_flow_speed"})
```

	link_id	lat	lon	borough	link_name	time
1	4456500	40,7712605	-73,833311	Queens	Whitestone Expwy N Exit 14 (Linden Pl) - BWB N Queens Anchorage	2026
2	4362249	40,7442206	-73,771661	Queens	LIE WB LITTLE NECK PKWY - NB CIP	2026
3	4616192	40,61052	-74,09769	Staten Island	SIE E CLOVE ROAD - FINGERBOARD ROAD	2026
4	4616338	40,7607	-74,002141	null	null	
5	4616310	40,82495	-73,836211	Bronx	CBE AMSTERDAM AVE (L/LVL) - MORRIS AVE	2026
6	4616356	40,6665206	-73,76246	Queens	Belt Pkwy W 182nd St - JFK Expressway	2026
7	4616246	40,8240706	-73,874311	Bronx	BE N STRATFORD AVENUE - CASTLE HILL AVE	2026
8	4616340	40,7016405	-73,991121	Manhattan	BQE N Atlantic Ave - MAN Bridge Manhattan Side	2026
9	4616241	40,7626006	-73,839671	Queens	VWE S MP8.65 (Exit 13 Northern Blvd) - MP6.39 (Exit 11 Jewel Ave)	2026
10	4616364	40,73744001	-73,85188001	Queens	LIE W 108TH ST - 84TH ST	2026
11	4616351	40,71167	-73,72839	Queens	CIP N Hempstead Tpk - LIE	2026
12	4620332	40,84671	-73,931941	Bronx	CBE W MORRIS AVE - GWB W AMSTERDAM AVE (L/LVL)	2026
13	4616354	40,683644	-73,72667001	Queens	Laurelton Pkwy S @ SSP - Belt Pkwy W 182nd St	2026
14	4616198	40,62102	-74,168861	Staten Island	WSE N-SIE E SOUTH AVENUE - SOUTH AVENUE	2026
15	4456505	40,797815	-73,79321	Queens	TNB N Queens Anchorage - Toll Plaza	2026
16	4616341	40,70908	-73,9959	Manhattan	FDR S Catherine Slip - BKN Bridge Manhattan Side	2026
17	4616199	40,60213	-74,187981	Staten Island	SIE W - WSE S SOUTH AVENUE - SOUTH AVENUE	2026
18	4616247	40,8271606	-73,84993	Bronx	BE N Castle Hill Avenue - Griswold Ave	2026
19	4616320	40,7894406	-73,786291	Queens	CIP N TNB - Whitestone Expwy S Exit 14 (Linden Pl)	2026
20	4616193	40,61052	-74,09769	Staten Island	SIE E BRADLEY AVENUE - CLOVE ROAD	2026
21	4456510	40,741534	-73,95406001	Queens	QMT W Toll Plaza - Manhattan Side	2026
22	4616350	40,74877	-73,73895	Queens	CIP S LIE - Hempstead Tpk	2026
23	4616213	40,5902	-74,19332	Staten Island	WSE N ARDEN AVENUE - VICTORY BLVD	2026
24	4616333	40,736971	-73,97477	Manhattan	FDR S 25th St - Catherine Slip	2026
25	4616215	40,52561	-74,23039	null	null	
26	4763657	40,60846	-74,03933	null	null	
27	4456501	40,68036	-74,00441001	Manhattan	BBT W Toll Plaza - Manhattan Portal	2026

Queries [6]

- df_gold
- df_lookup
- df_coords
- df_realtime
- df_realtime_dashboard
- df_dashboard

```
= Table.ExpandTableColumn(#"Merged Queries", "df_lookup", {"free_flow_speed"}, {"free_flow_speed"})
```

	link_id	link_name	borough	speed	data_as_of
1	4362249	LIE WB LITTLE NECK PKWY - NB CIP	Queens	41.01	2026-07-02 09:44:02-04:00
2	4362251	LIE WB LITTLE NECK PKWY - NB CVE	Queens	42.87	2026-07-02 09:43:04-04:00
3	4456500	Whitestone Expwy N Exit 14 (Linden Pl) - BWB N Queens Anchorage	Queens	18.01	2026-07-02 09:41:00-04:00
4	4616192	SIE E CLOVE ROAD - FINGERBOARD ROAD	Staten Island	52.81	2026-07-02 09:43:04-04:00
5	4616276	VWE S MP2.66 (Exit 2 Rockaway Blvd) - Belt Pkwy E 182nd St	Queens	6.21	2026-07-02 09:43:16-04:00
6	4616310	CBE AMSTERDAM AVE (L/LVL) - MORRIS AVE	Bronx	23.61	2026-07-02 09:43:16-04:00
7	4616340	BQE N Atlantic Ave - MAN Bridge Manhattan Side	Manhattan	4.97	2026-07-02 09:43:16-04:00
8	4616246	BE N STRATFORD AVENUE - CASTLE HILL AVE	Bronx	55.92	2026-07-02 09:43:04-04:00
9	4616353	Laurelton Pkwy N @ SSP - CIP N Hempstead Tpk	Queens	7.45	2026-07-02 09:43:16-04:00
10	4616241	VWE S MP8.65 (Exit 13 Northern Blvd) - MP6.39 (Exit 11 Jewel Ave)	Queens	52.19	2026-07-02 09:43:04-04:00
11	4616351	CIP N Hempstead Tpk - LIE	Queens	28.58	2026-07-02 09:43:04-04:00
12	4616364	LIE W 108TH ST - 84TH ST	Queens	34.79	2026-07-02 09:43:16-04:00
13	4620332	CBE W MORRIS AVE - GWB W AMSTERDAM AVE (L/LVL)	Bronx	8.07	2026-07-02 09:43:09-04:00
14	4456505	TNB N Queens Anchorage - Toll Plaza	Queens	36.03	2026-07-02 09:41:00-04:00
15	4616333	FDR S 25th St - Catherine Slip	Manhattan	22.99	2026-07-02 09:43:16-04:00
16	4616199	SIE W - WSE S SOUTH AVENUE - SOUTH AVENUE	Staten Island	53.43	2026-07-02 09:43:04-04:00
17	4616320	CIP N TNB - Whitestone Expwy S Exit 14 (Linden Pl)	Queens	47.84	2026-07-02 09:43:04-04:00
18	4616328	FDR N Catherine Slip - 25th St	Manhattan	31.68	2026-07-02 09:43:16-04:00
19	4456510	QMT W Toll Plaza - Manhattan Side	Queens	14.91	2026-07-02 09:41:00-04:00
20	4616213	WSE N ARDEN AVENUE - VICTORY BLVD	Staten Island	43.49	2026-07-02 09:43:04-04:00
21	4616298	VWE N MP4.63 (Exit 6 - Jamaica Ave) - MP6.39 (Exit 11 Jewel Ave)	Queens	19.26	2026-07-02 09:43:16-04:00
22	4456501	BBT W Toll Plaza - Manhattan Portal	Manhattan	36.03	2026-07-02 09:41:00-04:00
23	4616309	CBE E AMSTERDAM AVE(U/LVL) - MORRIS AVE	Bronx	23.61	2026-07-02 09:43:16-04:00
24	4616259	VWE S MP6.39 (Exit 11 Jewel Ave) - MP4.63 (Exit 6 Jamaica Ave)	Queens	8.69	2026-07-02 09:43:16-04:00
25	4616197	SIE E SOUTH AVENUE - RICHMOND AVENUE	Staten Island	42.25	2026-07-02 09:43:04-04:00
26	4616208	SIE W BRADLEY AVENUE - WOOLEY AVENUE	Staten Island	8.07	2026-07-02 09:43:04-04:00
27	4575278	MDE N VAN CORTLAND PARK - NYST N EXIT 1 (MP.48)	Bronx	21.12	2026-07-02 09:41:02-04:00
28					

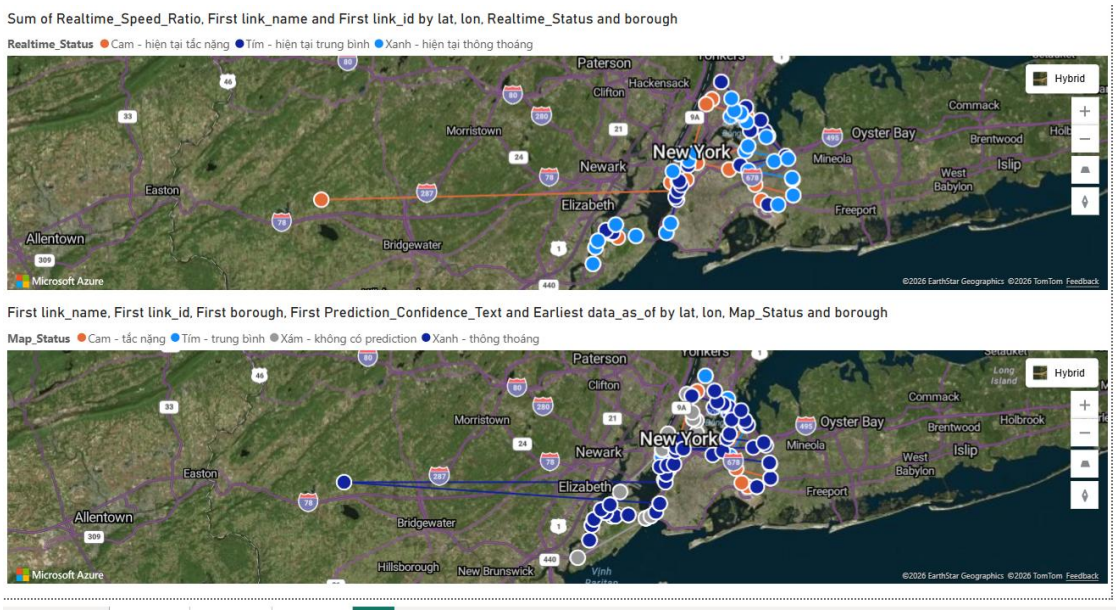
Hình 4.74: df_realtime, df_dashboard và df realtime_dashboard biểu thị thông tin hoàn chỉnh cho dự đoán 15 phút tới

Kết quả kiểm thử cho thấy dashboard có thể hiển thị các điểm giao thông trên bản đồ, phân loại màu theo mức độ tắc nghẽn dự đoán và cho phép người dùng lọc dữ liệu theo borough, hour và DateOnly. Ngoài ra, hệ thống hỗ trợ refresh thủ công trong Power BI để cập nhật dữ liệu mới nhất từ Azure Blob Storage.

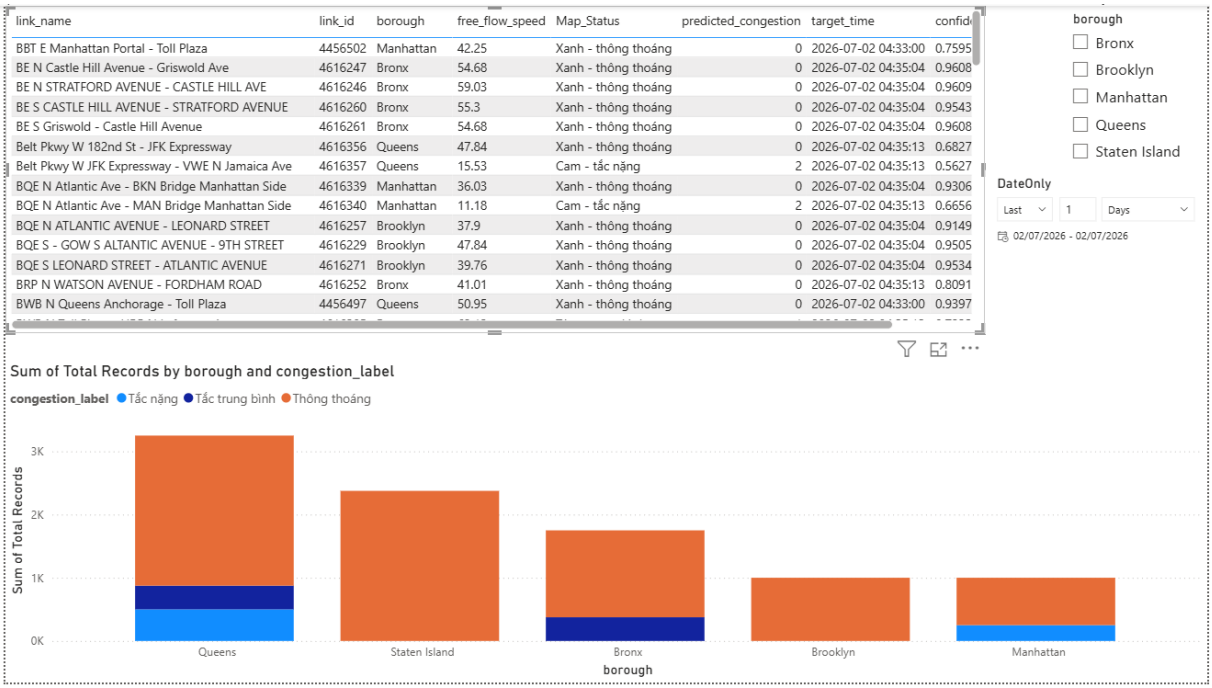
125
Current rows

Link Đường	Tên Đường	Quận	Độ tắc nghẽn 15p tới	Mức độ tin cậy của dự đoán	Thời điểm dữ liệu	Thời điểm dự đoán	Giờ
4362249	LIE WB LITTLE NECK PKWY - NB CIP	Queens	Thông thoáng	0.95883954803234	2026-07-02 04:21:02	2026-07-02 04:36:02	4
4616193	SIE E BRADLEY AVENUE - CLOVE ROAD	Staten Island	Thông thoáng	0.961300999509335	2026-07-02 04:20:13	2026-07-02 04:35:13	4
4616194	SIE E WOOLEY AVENUE - BRADLEY AVENUE	Staten Island	Thông thoáng	0.9543123872499425	2026-07-02 04:20:13	2026-07-02 04:35:13	4
4616250	HRP N LAFAYETTE AVENUE - E TREMONT AVENUE	Bronx	Tắc trung bình	0.7516943520832168	2026-07-02 04:20:13	2026-07-02 04:35:13	4
4616252	BRP N WATSON AVENUE - FORDHAM ROAD	Bronx	Thông thoáng	0.8091727296044056	2026-07-02 04:20:13	2026-07-02 04:35:13	4
4616259	VWE S MP6.39 (Exit 11 Jewel Ave) - MP4.63 (Exit 6 Jamaica Ave)	Queens	Tắc trung bình	0.7522358230839158	2026-07-02 04:20:13	2026-07-02 04:35:13	4
4616298	VWE N MP4.63 (Exit 6 - Jamaica Ave) - MP6.39 (Exit 11 Jewel Ave)	Queens	Tắc nặng	0.3814725104998604	2026-07-02 04:20:13	2026-07-02 04:35:13	4
4616305	BWB N Toll Plaza - HRP N Lafayatte Ave	Bronx	Tắc trung bình	0.7233460029954457	2026-07-02 04:20:13	2026-07-02 04:35:13	4
4616309	CBE E AMSTERDAM AVE(U/LVL) - MORRIS AVE	Bronx	Thông thoáng	0.7243487951286243	2026-07-02 04:20:13	2026-07-02 04:35:13	4
4616310	CBE AMSTERDAM AVE (L/LVL) - MORRIS AVE	Bronx	Thông thoáng	0.6619365234725977	2026-07-02 04:20:13	2026-07-02 04:35:13	4
4616333	FDR S 25th St - Catherine Slip	Manhattan	Thông thoáng	0.595005499025492	2026-07-02 04:20:13	2026-07-02 04:35:13	4
4616340	BQE N Atlantic Ave - MAN Bridge Manhattan Side	Manhattan	Tắc nặng	0.6656549543419635	2026-07-02 04:20:13	2026-07-02 04:35:13	4
4616341	FDR S Catherine Slip - BKN Bridge Manhattan Side	Manhattan	Thông thoáng	0.6692527817805247	2026-07-02 04:20:13	2026-07-02 04:35:13	4
4616352	CIP S Hempstead Tpk - Laurelton Pkwy @ SSP	Queens	Thông thoáng	0.6678466846910175	2026-07-02 04:20:13	2026-07-02 04:35:13	4
4616353	Laurelton Pkwy N @ SSP - CIP N Hempstead Tpk	Queens	Tắc nặng	0.9327017252539582	2026-07-02 04:20:13	2026-07-02 04:35:13	4
4616354	Laurelton Pkwy S @ SSP - Belt Pkwy W 182nd St	Queens	Thông thoáng	0.6196923169319103	2026-07-02 04:20:13	2026-07-02 04:35:13	4
4616356	Belt Pkwy W 182nd St - JFK Expressway	Queens	Thông thoáng	0.6827042297369827	2026-07-02 04:20:13	2026-07-02 04:35:13	4
4616357	Belt Pkwy W JFK Expressway - VWE N Jamaica Ave	Queens	Tắc nặng	0.5627524332393665	2026-07-02 04:20:13	2026-07-02 04:35:13	4
4616364	LIE W 108TH ST - 84TH ST	Queens	Thông thoáng	0.9347108732494072	2026-07-02 04:20:13	2026-07-02 04:35:13	4
4362250	CVE NB GCP - WILLETS PT BLVD	Queens	Tắc trung bình	0.7289686058149704	2026-07-02 04:20:07	2026-07-02 04:35:07	4
4362342	LIE E 84TH ST - 108TH ST	Queens	Thông thoáng	0.9302781665450054	2026-07-02 04:20:07	2026-07-02 04:35:07	4
4362244	CVE NB LIE - WILLETS PT BLVD	Queens	Thông thoáng	0.961820330049959	2026-07-02 04:20:04	2026-07-02 04:35:04	4
4362251	LIE WB LITTLE NECK PKWY - NB CVE	Queens	Thông thoáng	0.9579730941917652	2026-07-02 04:20:04	2026-07-02 04:35:04	4
4362314	TNB S Qns Anchorage - CIP S @ TNB	Queens	Tắc nặng	0.7870847899751097	2026-07-02 04:20:04	2026-07-02 04:35:04	4
4616192	SIE E CLOVE ROAD - FINGERBOARD ROAD	Staten Island	Thông thoáng	0.961820330049959	2026-07-02 04:20:04	2026-07-02 04:35:04	4
4616195	SIE E RICHMOND AVENUE - WOOLEY AVENUE	Staten Island	Thông thoáng	0.953190167443838	2026-07-02 04:20:04	2026-07-02 04:35:04	4
4616196	MLK S - SIE E WALKER STREET - WOOLEY AVENUE	Staten Island	Thông thoáng	0.8596523826303087	2026-07-02 04:20:04	2026-07-02 04:35:04	4

Hình 4.75: Bảng dự đoán tắc nghẽn trong 15 phút tiếp theo



Hình 4.76: Bản đồ độ tắc nghẽn hiện tại và trong 15 phút tiếp theo



link_name	link_id	borough	free_flow_speed	Map_Status	predicted_congestion	target_time	confid
BBT E Manhattan Portal - Toll Plaza	4456502	Manhattan	42.25	Xanh - thông thoáng	0	2026-07-02 04:33:00	0.7595
BE N Castle Hill Avenue - Griswold Ave	4616247	Bronx	54.68	Xanh - thông thoáng	0	2026-07-02 04:35:04	0.9608
BE N STRATFORD AVENUE - CASTLE HILL AVE	4616246	Bronx	59.03	Xanh - thông thoáng	0	2026-07-02 04:35:04	0.9609
BE S CASTLE HILL AVENUE - STRATFORD AVENUE	4616260	Bronx	55.3	Xanh - thông thoáng	0	2026-07-02 04:35:04	0.9543
BE S Griswold - Castle Hill Avenue	4616261	Bronx	54.68	Xanh - thông thoáng	0	2026-07-02 04:35:04	0.9608
Belt Pkwy W 182nd St - JFK Expressway	4616356	Queens	47.84	Xanh - thông thoáng	0	2026-07-02 04:35:13	0.6827
Belt Pkwy W JFK Expressway - VWE N Jamaica Ave	4616357	Queens	15.53	Cam - tắc nặng	2	2026-07-02 04:35:13	0.5627
BQE N Atlantic Ave - BKN Bridge Manhattan Side	4616339	Manhattan	36.03	Xanh - thông thoáng	0	2026-07-02 04:35:04	0.9306
BQE N Atlantic Ave - MAN Bridge Manhattan Side	4616340	Manhattan	11.18	Cam - tắc nặng	2	2026-07-02 04:35:13	0.6656
BQE N ATLANTIC AVENUE - LEONARD STREET	4616257	Brooklyn	37.9	Xanh - thông thoáng	0	2026-07-02 04:35:04	0.9149
BQE S - GOW S ALTANTIC AVENUE - 9TH STREET	4616229	Brooklyn	47.84	Xanh - thông thoáng	0	2026-07-02 04:35:04	0.9505
BQE S LEONARD STREET - ATLANTIC AVENUE	4616271	Brooklyn	39.76	Xanh - thông thoáng	0	2026-07-02 04:35:04	0.9534
BRP N WATSON AVENUE - FORDHAM ROAD	4616252	Bronx	41.01	Xanh - thông thoáng	0	2026-07-02 04:35:13	0.8091
BWB N Queens Anchorage - Toll Plaza	4456497	Queens	50.95	Xanh - thông thoáng	0	2026-07-02 04:33:00	0.9397

Hình 4.77: Bảng các Link_name tắc nghẽn theo giờ (Theo cơ chế lọc) và Biểu đồ phân bố tắc nghẽn theo quận

PHẦN KẾT LUẬN

1. Kết quả đạt được

Qua quá trình làm đồ án, nhóm đã xây dựng được hệ thống Data Lakehouse phục vụ phân tích và dự đoán mức độ ùn tắc giao thông tại thành phố New York. Hệ thống sử dụng dữ liệu lịch sử từ NYC Open Data và dữ liệu gần thời gian thực từ NYC DOT, sau đó tổ chức lưu trữ trên Azure Blob Storage theo kiến trúc Medallion gồm ba tầng Bronze, Silver và Gold. Quy trình xử lý dữ liệu được triển khai bằng Apache Spark, kết hợp Spark MLlib để huấn luyện mô hình RandomForestClassifier nhằm dự đoán mức độ tắc nghẽn giao thông trong 15 phút tiếp theo cho từng đoạn đường. Bên cạnh đó, hệ thống cũng xây dựng luồng dự đoán gần thời gian thực thông qua Apache Kafka và trực quan hóa kết quả bằng Power BI.

1.1. Ưu điểm

Xây dựng được một quy trình xử lý dữ liệu tương đối đầy đủ, từ thu thập, lưu trữ, làm sạch, xây dựng đặc trưng, huấn luyện mô hình đến trực quan hóa kết quả. Việc tổ chức dữ liệu theo kiến trúc Data Lakehouse giúp hệ thống phân tách rõ ràng giữa dữ liệu thô, dữ liệu đã xử lý và dữ liệu phục vụ phân tích.

Bên cạnh đó, hệ thống đã kết hợp được cả dữ liệu lịch sử và dữ liệu gần thời gian thực. Dữ liệu lịch sử được sử dụng để xây dựng tập huấn luyện và đánh giá mô hình, trong khi dữ liệu real-time được dùng để phục vụ luồng dự đoán đang vận hành. Các đặc trưng như tốc độ tự do, tỷ lệ tốc độ, trạng thái tắc nghẽn hiện tại và các đặc trưng trễ cũng giúp mô hình khai thác được xu hướng thay đổi của giao thông trong quá khứ gần.

Ngoài ra, việc sử dụng Docker, Kafka, Spark, Azure Blob Storage và Power BI giúp hệ thống có tính thực tiễn cao hơn so với một bài toán phân tích dữ liệu đơn lẻ. Kết quả dự đoán được đưa lên dashboard trực quan, giúp người dùng dễ quan sát tình trạng giao thông theo đoạn đường, quận và khung giờ.

1.2. Nhược điểm

Bên cạnh các kết quả đạt được, đồ án vẫn còn một số hạn chế. Trước hết, dữ liệu đầu vào chủ yếu dựa trên tốc độ giao thông, vì vậy hệ thống chưa xét đến nhiều yếu tố bên ngoài có thể ảnh hưởng trực tiếp đến ùn tắc như thời tiết, tai nạn, công trình đường

bộ, sự kiện lớn hoặc các ngày đặc biệt. Điều này có thể làm mô hình khó dự đoán chính xác trong những tình huống giao thông biến động bất thường.

Ngoài ra, mô hình hiện tại mới tập trung dự đoán mức độ tắc nghẽn trong mốc 15 phút tiếp theo và sử dụng RandomForestClassifier. Thuật toán này phù hợp với dữ liệu dạng bảng và có khả năng xử lý tốt trên Spark, nhưng vẫn còn hạn chế trong việc học các quan hệ chuỗi thời gian phức tạp hoặc quan hệ không gian giữa các đoạn đường.

2. Hướng phát triển

Trong tương lai, hệ thống có thể được tiếp tục phát triển theo hướng mở rộng dữ liệu đầu vào, cải thiện mô hình dự đoán và nâng cao khả năng vận hành. Việc bổ sung thêm các nguồn dữ liệu phụ trợ sẽ giúp hệ thống phản ánh đầy đủ hơn các yếu tố ảnh hưởng đến giao thông đô thị. Đồng thời, việc thử nghiệm các mô hình nâng cao cũng có thể giúp cải thiện độ chính xác và khả năng dự đoán trong các tình huống phức tạp.

Một hướng phát triển quan trọng là bổ sung thêm các nguồn dữ liệu liên quan như thời tiết, sự kiện và tai nạn giao thông. Thời tiết xấu, mưa lớn, tuyết, tai nạn hoặc các sự kiện đông người đều có thể làm thay đổi đáng kể tình trạng giao thông tại một khu vực. Khi kết hợp các dữ liệu này với dữ liệu tốc độ từ NYC DOT, hệ thống có thể hiểu rõ hơn nguyên nhân gây ùn tắc và đưa ra dự đoán sát với thực tế hơn.

Bên cạnh việc mở rộng dữ liệu, hệ thống có thể thử nghiệm thêm các mô hình nâng cao như XGBoost, LSTM, GRU hoặc Graph Neural Network. XGBoost có thể được sử dụng để cải thiện hiệu quả phân loại trên dữ liệu dạng bảng, đặc biệt trong trường hợp cần tối ưu độ chính xác và khả năng học các quan hệ phi tuyến. Trong khi đó, LSTM và GRU phù hợp với dữ liệu chuỗi thời gian, giúp mô hình khai thác tốt hơn xu hướng thay đổi của tốc độ giao thông theo thời gian.

TÀI LIỆU THAM KHẢO

[1] Vlahogianni, E. I., Karlaftis, M. G., & Golias, J. C. (2014). *Short-term traffic forecasting: Where we are and where we're going*. Transportation Research Part C: Emerging Technologies, 43, 3-19.

[2] New York State Department of Transportation. (2006). *Engineering Bulletin: Highway Design Manual Revision No. 50 - Revisions to Chapter 5 Basic Design (EB 06-036)*. New York State Department of Transportation. Retrieved from https://www.dot.ny.gov/divisions/engineering/design/dqab/hdm/hdm-repository/rev50.pdf

[3] Transportation Research Board. (2010). *Highway Capacity Manual 2010 (HCM2010)*. Transportation Research Board of the National Academies. Retrieved from https://sitesreservoirproject.riptideweb.com/references/REF23/Volume%201/Ch18_Nav_Trans_Traffic/Transportation%20Research%20Board%202010_HCM_2010_References.pdf

[4] Xiang, H., Wang, P., Xu, X., Yi, K., Zhang, X., Sheng, Q. Z., Beheshti, A., & Fan, W. (2025). *MTP: Exploring Multimodal Urban Traffic Profiling with Modality Augmentation and Spectrum Fusion*. arXiv preprint arXiv:2511.10218.

[5] Databricks. (n.d.). What is medallion architecture?. Retrieved from https://www.databricks.com/blog/what-is-medallion-architecture

[6] Singdata. (n.d.). Integrate machine learning with medallion architecture workloads. Retrieved from https://www.singdata.com/trending/integrate-machine-learning-medallion-architecture-workloads/

[7] Databricks. (2026). *What is a data lakehouse?* Databricks Documentation. https://docs.databricks.com/aws/en/lakehouse/

[8] Databricks. (2026). Databricks reference architectures. Databricks Documentation. https://docs.databricks.com/aws/en/lakehouse-architecture/reference

Nguồn github tham khảo:

[9] Nunez, J. E. (2015). DataMining_NewYork_Traffic_Speed [Source code]. GitHub. https://github.com/je-nunez/DataMining_NewYork_Traffic_Speed