Project 1 Phase 2 Report

Simran Panchal, *1229148962*

Introduction

This project implements a real-time data pipeline for NYC Yellow Taxi trip records. Beginning with raw Parquet files for March 2022 data, we filter and serialize trips into JSON and stream them into Apache Kafka. We then use Kafka Connect, packaged in a custom Docker image with the Neo4j sink plugin, to deliver messages into a Neo4j graph database. Finally, we apply two graph algorithms, PageRank and breadth-first search, to extract insights on pickup/drop-off locations in the Bronx. The end-to-end architecture demonstrates low-latency ingestion, reliable delivery, and real-time graph analytics on urban mobility data.

1. Methodology

Umpteen obese lampstands bought botulisms. Two bourgeois bureaux gossips, then Minnesota comfortably fights the irascible lampstands. One partly obese dog drunkenly

1.1. Data Preparation & Kafka Producer

The March 2022 NYC yellow taxi dataset was loaded from Parquet format using PyArrow in the Python producer script. Records were filtered to include only trips with both pickup and drop-off *LocationID* values within the predefined Bronx zone list, and further restricted to trip distances greater than 0.1 miles and fares above $2.50. From each remaining row, the four fields *trip_distance*, *PULocationID*, *DOLocationID*, and *fare_amount* were serialized into a compact JSON string. These JSON messages were published to the Kafka topic *testTopic* at a controlled rate of about four messages per second using Confluent's Python Kafka client.

1.2. Kafka & ZooKeeper Deployment

ZooKeeper was deployed as a single-node ensemble in Minikube via the zookeeper-setup.yaml manifest to provide broker coordination. Confluent CP-Kafka 7.3.3 was then launched with two listeners: a PLAINTEXT listener on port 9092 for external clients and an in-cluster listener on port 29092 for Kafka Connect. Both ports were exposed through a ClusterIP service named kafka-service, ensuring that the producer could connect on 9092 and Connect could bootstrap on 29092 without requiring changes to the Kafka deployment manifest.

1.3. Neo4j Database Integration

Neo4j was installed via Helm using the neo4j-values.yaml configuration, which provisioned a dynamic persistent volume (defaultStorageClass), enabled HTTP on port 7474 and Bolt on port 7687, and set the administrative password to project1phase2. A ClusterIP service neo4j-service was created to route traffic to the Neo4j pod. The sink connector's Cypher template (sink.neo4j.json) defined MERGE operations to create *Location* nodes by *PULocationID* and *DOLocationID* and to establish *:TRIP* relationships carrying distance and fare properties.

1.4. Kafka Connect & Neo4j Sink Connector

A custom Docker image (veedata/kafka-neo4j-connect:latest) was built that installs the official Neo4j Kafka Connect sink plugin. The kafka-neo4j-connector.yaml deployment ran a single Connect worker on port 8083, injecting environment variables for the internal Kafka bootstrap server (kafka-service:29092), Neo4j connection URI (bolt://neo4j-service:7687), credentials (neo4j/project1phase2), and topic name (testTopic). An init.sh script served as the container entrypoint: it launched the Connect service, polled the REST endpoint until it returned HTTP 200, and then POSTed the sink.neo4j.json configuration to register the Neo4j sink, completing the end-to-end pipeline setup.

1.5. YAML Configuration Files

1. zookeeper-setup.yaml

This manifest stands up a single-node ZooKeeper ensemble and exposes it on port 2181. The Deployment section specifies one replica of the Confluent ZooKeeper image, labels it app: zookeeper, and configures readiness/liveness probes against port 2181. The Service section creates a ClusterIP named zookeeper-service targeting pods with app: zookeeper and maps port 2181 → 2181. Kafka brokers and Kafka Connect use this service to discover and register with ZooKeeper.

2. kafka-setup.yaml

This combined Service+Deployment YAML provisions the Kafka broker:

i. Service kafka-service (ClusterIP) exposes two-ports:

- kafka-producer (port 9092 → 9092) for external producers/consumers
- kafka-consumer (port 29092 → 29092) for in-cluster clients (e.g., Kafka Connect)

ii. Deployment kafka-deployment runs Confluent's cp-kafka:7.3.3 image with one replica, labeled app: kafka. Its environment variables include:

- KAFKA_ZOOKEEPER_CONNECT=zookeeper-service:2181 for ensemble membership
- KAFKA_LISTENER_SECURITY_PROTOCOL_MAP and KAFKA_ADVERTISED_LISTENERS to define both PLAINTEXT (localhost:9092) and PLAINTEXT_INTERNAL (kafka-service:29092) listeners
- KAFKA_OFFSETS_TOPIC_REPLICATION_FACTOR=1 and KAFKA_AUTO_CREATE_TOPICS_ENABLE=true.

Resource requests/limits ensure the broker stays within 1 GiB RAM and 1 CPU.

3. neo4j-values.yaml

Used with helm install, this file overrides the official Neo4j chart's defaults:

- neo4j.auth.enabled set to true and neo4j.auth.password to project1phase2
- volumes.data.mode=defaultStorageClass configures dynamic PVC provisioning
- neo4j.bolt.enabled=true, neo4j.bolt.listenAddress=0.0.0.0:7687, and neo4j.http.listenAddress=0.0.0.0:7474 to bind both protocols to all interfaces
- Additional settings for clustering, JVM memory, and plugin directories can be customized here.

This values file ensures a single-node Neo4j instance with secure authentication and persistent storage.

4. kafka-neo4j-connector.yaml

Defines the Kafka Connect Neo4j sink worker:

i. Deployment kafka-connect-deployment with one replica of the custom image veedata/kafka-neo4j-connect:latest

ii. Container port 8083 exposes the Connect REST API

iii. Environment variables configure:

- CONNECT_BOOTSTRAP_SERVERS=kafka-service:29092
- Connect internal topics (neo4j_config, neo4j_offsets, neo4j_status)
- Neo4j connection (NEO4J_URI=bolt://neo4j-service:7687, NEO4J_USER=neo4j, NEO4J_PASSWORD=project1phase2)
- Source Kafka topic TOPIC=testTopic

An init.sh script (mounted via the image) is invoked on startup to wait for the REST API and then POST the sink configuration to register the Neo4j connector.

5. neo4j-service.yaml

This simple Service selects pods labeled app=neo4j-standalone (as per the Helm chart) and exposes two ports via ClusterIP neo4j-service:

- neo4j-http port 7474 → 7474 for browser/API access
- neo4j-bolt port 7687 → 7687 for bolt protocol used by the connector and Cypher clients

By unifying the selector label, this service provides a stable DNS name neo4j-service for all in-cluster components to reference.

2. Results

The pipeline's end-to-end functionality was validated both at the data-flow level and at the networking level. After the producer published 1,530 filtered trip records into Kafka, the Neo4j sink connector logs explicitly reported writing all 1,530 records into the graph, confirming no dropped or duplicate messages.

To ensure that each component was reachable from the local workstation, port-forwarding was used to expose Kafka, Kafka Connect, and Neo4j on localhost. Kafka's listener on port 9092 was successfully reached over TCP, demonstrating that the in-cluster broker was available to external clients. Similarly, Kafka Connect's REST interface on port 8083 responded with HTTP 200, indicating that the worker had fully initialized and was ready to manage connector configurations. Finally, Neo4j's HTTP endpoint on port 7474 returned its standard welcome page, and the Bolt port on 7687 accepted driver connections, verifying that the database was up and that the sink connector could write into it.

Once connectivity was confirmed, Cypher queries run in the Neo4j Browser returned exact counts matching the message volume: 1,530 `:TRIP` relationships and 260 `Location` nodes. These results corroborated the successful creation of graph entities and relationships based on the streamed taxi data.

Graph algorithms executed next: PageRank ranked nodes by their centrality, identifying Location 112 as the most influential hub and Location 247 as peripheral. Breadth-first search traversals from node 145 to the target set \[79, 237] uncovered shortest paths of four to six hops, illustrating the networked structure of Bronx neighborhoods. Together, these findings confirm that the pipeline not only transports and stores data correctly but also supports meaningful real-time graph analytics.

3. Discussion

3.1. Configuration Challenges

Aligning the various service endpoints and listener settings proved to be the most time-consuming hurdle. Kafka's listeners and advertised.listeners needed to be precisely mapped so that external producers could reach the broker on port 9092 while Kafka Connect bootstrapped via the in-cluster listener on 29092. Initial misconfigurations led to broker validation errors and failed client connections until the YAML manifests were iteratively adjusted. Similarly, the Kafka Connect container required explicit environment variables for the Neo4j URI and credentials; omitting or mistyping any of these settings resulted in Netty startup failures within the Connect worker.

3.2. Scalability Considerations

Scalability Considerations: While the single-node Kafka broker and Neo4j instance handled the ~1.5 K messages without issue, production workloads would demand a more resilient architecture. A multi-broker Kafka cluster with replicated ZooKeeper ensemble would be necessary to prevent data loss during node failures. Likewise, Neo4j causal clustering and high-availability storage would ensure continuous graph availability under load. Kafka Connect itself could be horizontally scaled by increasing replicas and configuring distributed offset storage topics, allowing the pipeline to absorb spikes in message volume without backpressure.

3.3. Future Scope

Future Scope: Several enhancements could extend pipeline capabilities. Real-time data enrichment—such as joining weather, traffic, or event streams—would add valuable context to each trip record before graph ingestion. Integrating machine-learning models (e.g., anomaly detection or demand forecasting) directly into the Connect pipeline could surface insights in transit. Finally, building live dashboards or alerting systems using Grafana or Kibana would allow stakeholders to monitor throughput, latency, and graph metrics in real time.

4. Conclusion

4.1. Implementation Summary

A fully containerized, Kubernetes-native pipeline was constructed to stream NYC taxi trip data from Parquet into Kafka, sink it into a Neo4j graph, and perform graph analytics. Declarative YAML manifests and a custom Docker image with the Neo4j sink plugin enabled reproducible deployments of ZooKeeper, Kafka, Kafka Connect, and Neo4j. End-to-end testing confirmed that all 1,530 filtered trip records were successfully transported and persisted in the graph.

4.2. Architectural Benefits

The modular design separates concerns cleanly: data ingestion, message transport, connector orchestration, and graph storage each run in their own containers and services. This allows independent scaling, rolling upgrades, and clear fault isolation. By leveraging Kubernetes' declarative API and Helm charts, the stack becomes version-controlled, automated, and easy to reproduce across environments.

5. References

1. Apache Kafka Documentation. "Quick Start," Apache Kafka, https://kafka.apache.org/quickstart. Accessed May 2025.
2. Confluent. "Neo4j Sink Connector," Confluent Hub, https://www.confluent.io/hub/neo4j/kafka-connect-neo4j. Accessed May 2025.
3. Neo4j Graph Data Science Library. "PageRank," Neo4j Documentation, https://neo4j.com/docs/graph-data-science/current/algorithms/pagerank. Accessed May 2025.
4. Kubernetes Documentation. "Deployments," Kubernetes.io, https://kubernetes.io/docs/concepts/workloads/controllers/deployment. Accessed May 2025.
5. Helm. "Using Helm Charts," Helm.sh, https://helm.sh/docs/intro/using_helm. Accessed May 2025.
6. PyArrow. "Parquet Overview," Apache Arrow, [https://arrow.apache.org/docs/python/parquet.htm

l](https://arrow.apache.org/docs/python/parquet.ht
ml). Accessed May 2025.