

NYC Taxi Graph Pipeline: Dockerized Neo4j and Kubernetes-Based Stream Processing

Siddharth Axaykumar Mehta

Arizona State University

smehta74@asu.edu

ABSTRACT

This paper presents the design, implementation, and evaluation of a two-phase graph-data processing pipeline built around the March 2022 NYC Yellow Taxi dataset, scoped to Bronx taxi zones. Phase 1 establishes a fully automated Dockerized Neo4j environment that ingests 1,530 validated trip records as a directed property graph and executes PageRank and Breadth-First Search (BFS) through the Neo4j Graph Data Science (GDS) library. Phase 2 replaces the static batch loader with a live, Kubernetes-orchestrated streaming pipeline comprising Apache Zookeeper, Apache Kafka, a Neo4j Kafka Sink Connector, and Neo4j deployed via Helm. A Python producer streams trip records to Kafka in real time; the connector persists each event to Neo4j, where the same graph algorithms are executed through a shared Python interface. Both phases achieve a perfect automated test score (Phase 1: 60/60; Phase 2: 100/100), validating both infrastructure correctness and algorithm fidelity.

1. INTRODUCTION

Graph databases offer a natural representation of transportation networks, where geographic locations correspond to nodes and individual transit events to directed, weighted edges. This correspondence makes them particularly attractive for the New York City Taxi and Limousine Commission (TLC) dataset, which captures every yellow-cab trip in the city with high fidelity. Restricting analysis to Bronx taxi zones yields a tractable yet realistic subgraph that is well-suited for evaluating graph-analytic algorithms at moderate scale.

This project addresses two complementary questions. First, given a static parquet snapshot, how can that data be loaded into a graph database inside a fully reproducible container and subjected to classical graph algorithms? Second, how can the same analytical pipeline be extended to a continuous data stream using cloud-native tooling on Kubernetes?

To answer these questions, we design and implement two distinct pipeline phases that share a common graph schema and a common Python interface for PageRank and BFS. This architectural decision enforces a clean separation between infrastructure concerns and algorithm logic, allowing each phase to be evaluated independently

while ensuring that algorithm results remain directly comparable across deployment environments.

The remainder of this paper is organized as follows. Section 2 details the Phase 1 Dockerized Neo4j pipeline. Section 3 describes the Phase 2 Kubernetes streaming architecture. Section 4 discusses cross-phase observations, algorithm insights, and scalability considerations. Section 5 concludes with engineering lessons learned.

2. PHASE 1: DOCKERIZED NEO4J

2.1 Dataset and Graph Schema

The source data is the March 2022 NYC Yellow Taxi trip record file, distributed as a Parquet archive approximately 53 MB in size and containing roughly 3.6 million rows. We apply two filters: (i) both the pickup and dropoff TLC location IDs must fall within the 42 designated Bronx zones, and (ii) records with null or non-positive trip distances and fares are discarded. After filtering, 1,530 valid trip records remain.

The graph schema maps cleanly onto the cleaned data. Each unique TLC zone identifier becomes a `Location` node carrying an integer `name` property. Each trip record becomes a directed `TRIP` relationship with four properties: `distance` (miles), `fare` (USD), `pickup_dt` (datetime), and `dropoff_dt` (datetime). Node creation uses `MERGE` to guarantee uniqueness, while relationship creation uses `CREATE` to preserve parallel trips between the same zone pair. The resulting graph contains 42 nodes and 1,530 relationships.

2.2 Dockerfile Design

Reproducibility is the primary design goal of Phase 1. The Dockerfile extends the official `neo4j` base image and performs all configuration and data loading at build time, requiring no manual post-build intervention. Table 1 summarizes the build steps.

Neo4j is configured to listen on `0.0.0.0:7687` (Bolt protocol) and `0.0.0.0:7474` (HTTP). GDS and APOC procedures are whitelisted via targeted `neo4j.conf` appends during the build, ensuring that graph algorithm calls succeed without additional runtime configuration.

Table 1: Dockerfile build steps for Phase 1.

Step	Action
1	Create /cse511/ working directory
2	Download March 2022 NYC Taxi Parquet via wget
3	Install OpenJDK 21 via apt
4	pip install neo4j pandas pyarrow
5	Install GDS plugin v2.21.0 JAR to Neo4j plugins directory
6	Append network and procedure settings to neo4j.conf
7	Set default Neo4j password (graphprocessing)
8	Execute data_loader.py to populate graph at build time

2.3 Data Loader

data_loader.py connects to Neo4j via the official Python driver, reads the Parquet file using pyarrow, applies the Bronx zone filter, and streams rows to Neo4j in batches using parameterized UNWIND statements. The core ingestion Cypher is as follows:

```
UNWIND $rows AS row
MERGE (p:Location {name: toInteger(row.
    PULocationID)})
MERGE (d:Location {name: toInteger(row.
    DOLocationID)})
CREATE (p)-[:TRIP {
    distance: row.trip_distance,
    fare: row.fare_amount,
    pickup_dt: datetime(row.pickup_dt),
    dropoff_dt: datetime(row.dropoff_dt)
}]->(d)
```

Batch-based UNWIND substantially reduces round-trip overhead compared to row-by-row insertion, a critical consideration when ingesting on a single-node container without connection pooling.

2.4 Graph Algorithm Implementation

PageRank. A named in-memory graph projection (pr_graph) is created over Location nodes and TRIP relationships. The GDS pageRank.stream procedure is invoked with caller-supplied max_iterations and weight_property parameters and a fixed damping factor of 0.85. The procedure returns a score for every node; the implementation extracts and returns the single highest-scoring and lowest-scoring nodes.

BFS. A separate projection (bfs_graph) is created, and node internal IDs for the specified start_node and target_nodes are resolved via lookup queries. The GDS bfs.stream procedure traverses the graph and returns the sequence of visited nodes as an ordered list of name dictionaries, representing the traversal path.

2.5 Phase 1 Results

All five PageRank and all five BFS test cases passed without error. The automated tester awarded full marks

across every category: data loading 10/10, PageRank 25/25, and BFS 25/25, for a total of 60/60.

3. PHASE 2: KUBERNETES STREAMING PIPELINE

3.1 Architecture Overview

Phase 2 replaces the static Docker container with a distributed, event-driven architecture deployed on Minikube. Four principal components interact over a Kubernetes cluster network:

1. **Python Kafka Producer** — reads the filtered trip records and publishes each as a JSON message to a Kafka topic.
2. **Apache Zookeeper + Kafka** — provide distributed messaging infrastructure; deployed as Kubernetes Deployment and Service objects.
3. **Neo4j Kafka Sink Connector** — consumes messages from Kafka and writes each event to Neo4j using a per-event Cypher template.
4. **Neo4j (Helm)** — stores the streaming graph and serves GDS algorithm queries through the same Python interface as Phase 1.

3.2 Kubernetes Manifests

Zookeeper. A ClusterIP Service is exposed on port 2181. The Deployment runs confluentinc/cp-zookeeper:7.3.3 with resource limits of 1 CPU and 1 GB of memory.

Kafka. Two advertised listeners are configured: an external listener on port 9092 (port-forwarded to the host for the producer) and an internal listener on port 29092 (for connector-to-broker communication inside the cluster). This dual-listener configuration allows both the external Python producer and the internal sink connector to reach the same broker without additional network bridging.

Neo4j. Deployed via the official Helm chart: helm install my-neo4j-release neo4j/neo4j. The GDS plugin is enabled through the NEO4J_PLUGINS environment variable. A critical fix involved correcting the Service selector from app=neo4j-standalone to app=my-neo4j-release to match the labels emitted by the Helm chart.

Connector. The connector pod uses the image veedata/kafka-neo4j-connect:latest2. A ConfigMap volume mount injects a corrected sink.neo4j.json configuration, overriding the credentials baked into the base image without requiring a rebuild. The container memory limit was raised to 2500 Mi to prevent JVM out-of-memory termination.

3.3 Sink Connector Cypher

The connector applies the following Cypher template to each consumed Kafka event:

```
MERGE (p:Location {name: toInteger(event.
  PULocationID)})
MERGE (d:Location {name: toInteger(event.
  DOLocationID)})
CREATE (p)-[:TRIP {
  distance: toFloat(event.trip_distance),
  fare:      toFloat(event.fare_amount)
}]->(d)
```

Both key and value converters are set to `JsonConverter` without schema enforcement. Setting `errors.tolerance=all` prevents the connector from stalling on malformed or incomplete messages, maintaining pipeline throughput under imperfect input conditions.

3.4 Debugging and Engineering Challenges

Three non-trivial inter-component misconfigurations were encountered and resolved during integration:

Service selector mismatch. The Neo4j Service manifest initially specified `app=neo4j-standalone` as its pod selector. The Helm-deployed pods carry the label `app=my-neo4j-release`, so `kubectl port-forward` failed because no pods matched the selector, causing port-forward to hang. Correcting the selector to match the Helm chart labels immediately restored connectivity.

Connector JVM OOMKill. The connector container was configured with a 1000 Mi memory limit, while the JVM defaulted to a 2 GB maximum heap (`-Xmx2G`). The discrepancy caused the Linux OOM killer to terminate the container during startup. Raising the Kubernetes resource limit to 2500 Mi resolved the issue without requiring JVM heap tuning.

Incorrect credentials in baked-in config. The `sink.neo4j.json` bundled inside the connector image was configured with the Phase 1 Neo4j password. Rather than rebuilding the image, a `ConfigMap` volume mount was used to overlay the corrected configuration file at runtime, demonstrating an important operational pattern for overriding third-party image configurations.

3.5 Phase 2 Results

The Python producer published all 1,530 trip records to Kafka. The sink connector consumed every message and persisted them to Neo4j, resulting in 42 `Location` nodes. Table 2 shows the complete automated test scores.

4. DISCUSSION

4.1 Phase Comparison

Phase 1 optimizes for reproducibility: a single Docker image encapsulates the entire environment, and a fresh `docker build` produces a fully populated, queryable graph with no external dependencies. Phase 2 optimizes

Table 2: Phase 2 automated test scores.

Category	Score
Zookeeper infrastructure	10 / 10
Kafka infrastructure	10 / 10
Kafka connectivity	10 / 10
Neo4j deployment	8 / 8
Neo4j connectivity	7 / 7
Kafka-Neo4j connector	15 / 15
Data file + producer	20 / 20
End-to-end pipeline	20 / 20
Total	100 / 100

for scalability and operational flexibility through component isolation, at the cost of three inter-component configuration mismatches that required systematic debugging. The trade-off is characteristic of distributed systems more broadly: greater modularity enables independent scaling and replacement of components, but introduces additional integration surface area that must be carefully managed.

4.2 PageRank Observations

With trip distance as the edge weight, the highest-PageRank Bronx zone corresponds to a major transit hub that receives many long-distance incoming trips. High fare-weighted PageRank is consistent with a zone that serves as a primary destination from distant parts of the borough. Conversely, the lowest-ranked zone is a peripheral location with few and short inbound trips, indicating low centrality within the Bronx taxi network.

4.3 BFS Observations

BFS from zone 159 (Mott Haven/Port Morris) to zone 212 (Pelham Bay Park) returns a direct, single-hop path, confirming the existence of at least one recorded direct trip between these zones. When BFS is invoked for zone pairs lacking a direct trip record, multi-hop paths emerge that route through intermediate zones, reflecting the connectivity structure of the Bronx sub-graph.

4.4 Scalability Considerations

The current deployment is intentionally single-node and suitable for experimentation. A production-grade deployment would require several targeted enhancements: increasing Kafka topic partitions to enable parallel consumption; enabling Neo4j Enterprise cluster mode with leader election for write availability; configuring multiple connector tasks to match the partition count; and migrating from Minikube to a managed Kubernetes service (e.g., Amazon EKS, Google GKE, or Azure AKS) for auto-scaling, managed storage, and network load balancing. Additionally, a schema registry (e.g., Confluent Schema Registry) would enforce message contracts and reduce the need for `errors.tolerance=all` as a

defensive measure.

5. CONCLUSION

This project demonstrates a principled progression from a self-contained, single-image Docker pipeline to a cloud-native, event-driven Kubernetes streaming architecture for graph data processing. The shared graph schema and shared Python algorithm interface validated the value of abstraction: algorithm logic was written once and executed correctly in two substantially different infrastructure environments.

Three concrete engineering lessons emerged from the implementation. First, Kubernetes label selectors must match pod labels exactly—a one-character discrepancy can silently break port forwarding and inter-service communication. Second, JVM-based containers must be sized with the JVM heap budget explicitly in mind; the default maximum heap frequently exceeds the container memory limit, leading to OOM termination. Third, ConfigMap volume mounts provide a clean, rebuild-free mechanism for overriding configuration files baked into third-party container images, a pattern of broad practical utility.

Phase 1 achieved a score of 60/60 and Phase 2 a score of 100/100 on the automated test suite, confirming the correctness of both the infrastructure implementation and the graph algorithm logic.

REFERENCES

- [1] Neo4j, Inc. Neo4j Graph Data Science Library Documentation. <https://neo4j.com/docs/graph-data-science/>, 2024.
- [2] Neo4j, Inc. Confluent Neo4j Sink Connector Documentation. <https://neo4j.com/docs/kafka/current/sink/>, 2024.
- [3] New York City Taxi and Limousine Commission. TLC Trip Record Data. <https://www.nyc.gov/site/tlc/about/tlc-trip-record-data.page>, 2024.
- [4] Apache Software Foundation. Apache Kafka Documentation. <https://kafka.apache.org/documentation/>, 2024.
- [5] The Kubernetes Authors. Kubernetes Documentation. <https://kubernetes.io/docs/>, 2024.
- [6] The Helm Authors. Helm Documentation. <https://helm.sh/docs/>, 2024.
- [7] L. Page, S. Brin, R. Motwani, and T. Winograd. The PageRank Citation Ranking: Bringing Order to the Web. Technical Report 1999-66, Stanford InfoLab, 1999.