Real-Time Transit Data Pipeline
A Streaming Architecture for Swedish Public Transport

Pranav Rajan David Tanudin

October 18, 2025

1 Introduction

This project implements a real-time data processing pipeline for Swedish public transport data. The system demonstrates practical applications of streaming architectures, NoSQL databases, and distributed processing frameworks. We take in live transit data from Trafiklab's API and make a UI visualization through an interactive web dashboard, enabling users to monitor real-time departures and delay patterns across Sweden's public transport network.

2 Dataset

2.1 Data Source

We utilize Trafiklab's real-time departures API (`https://www.trafiklab.se/`), which provides transit data for Swedish public transport. The API offers many different endpoints, but our project focused on two key endpoints: (1) a stops search API that returns transit stop metadata including stop IDs, names, and transport modes, and (2) a departures API that returns scheduled and real-time departure information.

2.2 Data Structure

Each departure record contains the following fields:

- **Operator**: Transit agency name (e.g., SL, Västtrafik)
- **Line**: Route designation (e.g., "14", "Blue Line")
- **Destination**: Terminal station or direction
- **Scheduled Time**: Planned departure timestamp
- **Real-time Time**: Actual departure based on vehicle tracking
- **Delay**: Difference in seconds between scheduled and real-time
- **Stop ID**: Unique identifier for the transit stop

The API returns all departures within a 60-minute window from the query time, with a monthly quota of 100,000 calls, which is sufficient for continuous monitoring of multiple stops.

3 Methodology

3.1 Architecture

Our pipeline follows a standard streaming architecture: `Trafiklab API → Producer → Kafka → Spark Streaming → MongoDB → Web Dashboard`. Each component serves a distinct purpose:

- **Producer**: A Python script polls the Trafiklab API every 60 seconds and publishes JSON messages to Kafka. The producer accepts a stop ID as a command-line argument, allowing monitoring of any transit stop in Sweden.
- **Kafka**: Acts as the message broker, sepreating data ingestion from processing. Messages are published to the `sl_stream` topic.
- **Spark Streaming**: Consumes messages from Kafka in micro-batches, applies schema validation, and writes to MongoDB. Spark ensures fault tolerance through checkpointing and enables scalable processing.
- **MongoDB**: Stores all departure records in a document-oriented format. The flexible schema accommodates varying data structures from different operators.
- **Web Dashboard**: A Flask application with automatic producer management. Users can search for stops, select a stop to monitor, and view real-time departures with pagination and auto-refresh.

3.2 Mini-Batch Processing

We employ a 60-second mini-batch interval, which is in line with the API's data characteristics. Since the API returns departures within a 60-minute window, sub-minute API calls would be excessive. However, delay information updates continuously based on vehicle GPS positions, making the 60-second refresh optimal for capturing real-time changes without excessive API calls.

3.3 Deduplication Strategy

The API's 60-minute window creates duplicate records across consecutive fetches. We implement MongoDB pipelines that group departures by composite key (operator, line, destination, scheduled time) and only keep the most recent record. This makes sure that users see unique departures with the latest delay estimates.

3.4 Producer Management

The web dashboard uses an automatic producer management. When a user selects a transit stop, the system spawns a dedicated producer subprocess for that stop and terminates any previous producers. This optimizes resource usage while maintaining historical data in MongoDB, making it possible for instant switching between stops.

4 Running the Code

Detailed instructions for running the code are available in the README.md file in the project repository on GitHub. A complete copy of these instructions is also provided in Appendix A for reference.

A Installation and Setup Instructions

A.1 Prerequisites

- Python 3.11+
- Docker Desktop with Docker Compose
- Trafiklab API key is included in the repository's `.env` file

A.2 Installation Steps

1. Clone repository and install dependencies:

```
git clone <repository-url>
cd id2221-data-intensive-traffic
pip install -r requirements.txt
```

2. Verify API key:
The `.env` file with the Trafiklab API key is already included in the GitHub repository. Verify it exists after cloning:

```
cat .env
```

3. Start the pipeline:

```
python main.py
```

This command starts Docker containers (Kafka, Spark, MongoDB), initializes the producer, and begins Spark streaming.

4. Start the web dashboard (separate terminal):

```
python web_dashboard.py
```

5. Access the dashboard:
Open `http://localhost:5432` in a web browser. Search for a transit stop (e.g., "Stockholm", "Göteborg"), select it, and wait 60 seconds for the first data to appear.

A.3 Available Services

- **Web Dashboard**: `http://localhost:5432`
- **Kafka UI (Kafdrop)**: `http://localhost:9000`
- **MongoDB**: `mongodb://localhost:27017`

A.4 Querying MongoDB

To query data directly:

```
python mongo_query.py
```

Or use MongoDB shell:

```
docker exec -it mongodb mongosh
use trafiklab
db.departures.find().limit(5).pretty()
```