Building a Data Pipeline Using Streaming Data Processing: Final Report

Maureen Ekwebelem & Andre Chuabio
CISC 5950
May 11, 2025

What have we learned in this project?

In this project, we aimed to build a real-time data pipeline using Airflow, Kafka, Spark Streaming, and Cassandra , all running in Docker. These tools work together to have real time streaming data moving from one step to the next. Airflow schedules and sends the data in a Kagka topic, Spark Streaming will read and process the streaming data, and the final transformed output is put into Cassandra for storage. We also gained insight into the dependencies of the different technologies within each other. The Control Center listens for events on the Schema registry to visualize what's happening in Kafka, which is managed by the Zookeeper. We also learned how to use Spark with external JARs to connect Kafka and Cassandra as well as SQL.

Furthermore, we gained a lot of insight on how to orchestrate an end to end pipeline, it was interesting to learn about Airflow's role as a type of "conductor", how airflow DAGs can reliably schedule and trigger each stage of a streaming workflow. Defining task dependencies (generate-> publish->process-> persist) not only ensures correct ordering, but makes retries and backfills straightforward when individual steps fail. Playing with Kafka as a decoupling layer was also interesting, it was rewarding to decouple data generation from processing by inserting Kafka between producers and consumers. By doing so, it allowed us to independently scale producers(airflow tasks) and consumers (sparkjobs), and buffer spikes in event volume without losing messages.

Also, attempting the articles single-node docker compose setup on a three node GCP cluster exposed the need for explicit service placement and network segmentation. Simply lifting and shifting containers led to port conflicts and unbalanced resource usage: all three Kafka brokers attempted to become leaders simultaneously, and spark workers competed for the same Docker bridge network. We learned to assign each service a dedicated host-orchestrating the ZooKeeper across all nodes, placing kKafka brokers one per node, and co-locating spark master and seed cassandra on the most powerful node- so that election and replication could function predictably

What were the challenges?

This project came with numerous challenges. First, the biggest challenge we ran into was getting Spark Streaming to push the data into Cassandra. We were seeing the data just fine in Kafka, but somehow it wasn't making it to Cassandra indicating some error with the Spark Streaming + Cassandra jobs. Prior to the creation of the video demonstration, we spent a good amount of time debugging, but our streaming data wasn't appearing, even though the pipeline seemed to be working.

Another challenge was the inconsistency between the Youtube tutorial, the Github README, and the project site. The README made the set up seem poverty straightforward with roughly 3 steps, yet in reality it wasn't. On top of that, we had to fix our code to sort out a naming mismatch, the 'user_created' vs 'users_created' which kept appearing. Even though it was a small difference, it caused confusion in our pipeline and gave us issues and it would have been nice if it was consistent across the scripts and the documents.

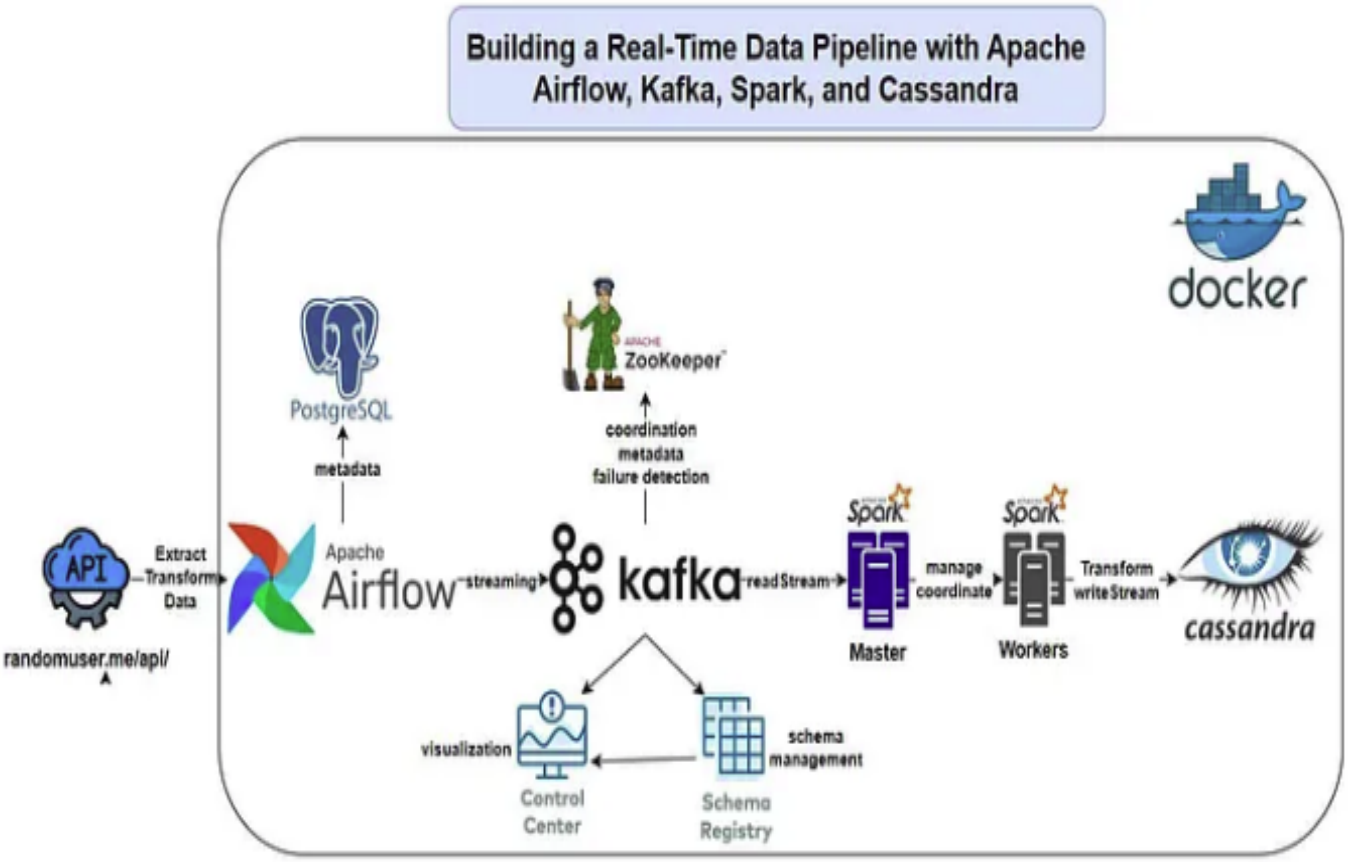
Another obstacle arose from disk-capacity constraints on our VM hosts. As containers generated logs and temporary data, the root volumes filled up rapidly, leading to container crashes and degraded cluster performance. At first, we attempted manual cleanups and log pruning, but this was neither sustainable nor reliable. We ended up increasing the size of the disks attached to each node.

Furthermore, we encountered network configuration errors that prevented Spark workers from registering with the master. By default, our GCP firewall rules did not include spark's RPC port or the web UI port, so worker nodes repeatedly timed out when attempting to connect, or in some cases connection was refused. To solve this we updated our network policies and firewall rules on the Cloud.

How does window size affect our programs?

Having more data over a longer interval can lead to longer delays in processing. If the window is too short, you might not see enough data show up in Cassandra at once. But if it's too long, Spark can take longer to write the data, which can make the system feel like it's lagging. It's important to ensure you find a decent window length to avoid either of these issues.

The choice of window size in spark streaming taught us that it represents a trade-off between data completeness and system latency. Larger windows aggregate more events before triggering a computation, which could improve the statistical performance of our results but we found that it also introduces greater end-to-end delay before each output appears in Cassandra. On the other hand, small windows produce more frequent outputs , but individual windows contain too few events to yield meaningful aggregations, and the increased scheduling overhead can strain the streaming job.


Building a Real-Time Data Pipeline with Apache Airflow, Kafka, Spark, and Cassandra
docker
PostgreSQL
metadata
ZooKeeper
coordination
metadata
failure detection
API
Extract
Transform
Data
randomuser.me/api/
Apache
Airflow
streaming
kafka
readStream
Spark
Master
manage
coordinate
Spark
Workers
Transform
writeStream
cassandra
visualization
Control
Center
schema
management
Schema
Registry