# Autoscaling Stream ETL & Window Aggregation using Spark Structured Streaming

Ahmed Yar, Humayun Junaid, Muhammad Asjad Kashif
*Department of Computer Science*
*National University of Sciences & Technology (NUST)*
Pakistan
Course Code: CS-315 – Cloud Computing

***Abstract*—In the era of Big Data, static infrastructure often leads to either resource wastage (during low traffic) or performance bottlenecks (during high traffic). This project implements a robust, real-time streaming pipeline using Apache Spark Structured Streaming and Apache Kafka. The primary objective was to demonstrate Infrastructure Elasticity (Autoscaling). By enabling Spark's Dynamic Allocation, our system automatically provisions additional computing resources (Executors) during load spikes and releases them during idle periods. The pipeline processes simulated real-time logs from Computer Science students, calculating metrics such as "Total Lines of Code" and "Average CPU Usage" in 10-second tumbling windows.**

***Index Terms*—Apache Spark, Structured Streaming, Kafka, Autoscaling, Cloud Computing, ETL.**

## I. Introduction

### A. Problem Statement

Traditional batch processing systems cannot provide real-time insights. Furthermore, streaming systems deployed on static clusters face a financial dilemma:

- *Over-provisioning:* Paying for peak capacity 24/7 results in high cloud costs.
- *Under-provisioning:* Using a small cluster results in high latency and potential data loss during traffic spikes.

### B. Proposed Solution

We implemented a Stream ETL (Extract, Transform, Load) job that utilizes Spark Dynamic Allocation. This feature allows the application to communicate with the cluster manager to request or remove executors based on the current backlog of tasks, ensuring optimal resource utilization and cost efficiency.

## II. System Architecture

The pipeline consists of three main stages: Ingestion, Processing, and Visualization.

### A. Data Ingestion Layer (Apache Kafka)

The ingestion layer serves as the entry point for data.

- **Source:** A custom Python Producer script (`producer.py`) uses the Faker library to generate synthetic JSON logs.
- **Scenario:** The data simulates CS students working on assignments. Fields include `student_id`, `course`, `activity` (coding, debugging), `lines_of_code`, and `cpu_usage`.
- **Message Broker:** These logs are pushed to a Kafka topic named `cs_student_logs`. Kafka acts as a high-throughput buffer, decoupling the data generation from processing [4].

### B. Processing Layer (Spark Structured Streaming)

- **Engine:** Apache Spark 3.5.0 (PySpark).
- **Logic:**
  1) **Read:** Consumes data from Kafka.
  2) **Parse:** Converts JSON strings into a structured Schema.
  3) **Watermarking:** Handles late-arriving data with a 1-minute threshold to ensure data consistency [1].
  4) **Aggregation:** Performs a Tumbling Window aggregation every 10 seconds to calculate the sum of `lines_of_code` and average `cpu_usage` grouped by Course.

### C. Autoscaling Mechanism

The system is configured with the following Dynamic Allocation parameters:

- `spark.dynamicAllocation.enabled`: True
- `spark.dynamicAllocation.minExecutors`: 0 (scales down to zero if idle)
- `spark.dynamicAllocation.maxExecutors`: 4 (prevents infinite resource consumption)
- `spark.dynamicAllocation.schedulerBacklogTimeout` 1s (triggers scaling quickly)

## III. Implementation Details

### A. Technology Stack

The technologies utilized in this project are outlined in Table I.

TABLE I
TECHNOLOGY STACK

| Component | Technology Used |
|---|---|
| Language | Python 3.13 (PySpark) |
| Streaming Engine | Spark Structured Streaming |
| Message Broker | Apache Kafka (Docker) |
| Cluster Manager | Spark Standalone |
| OS | Windows 10/11 |

### B. Key Code Logic

The core logic aggregates data based on event time windows. The code snippet below demonstrates the windowing strategy:

```
windowed_aggs = parsed_df \
    .withWatermark("event_time", "1 minute") \
    .groupBy(
        window(col("event_time"), "10 seconds"),
        col("course")
    ) \
    .agg(
        sum("lines_of_code").alias("total_loc"),
        avg("cpu_usage_pct").alias("avg_cpu")
    )
```

Listing 1. Tumbling Window Aggregation

## IV. Experimental Results & Autoscaling Demo

To evaluate the system, we performed a load test simulating a traffic spike.

### A. Test Scenario

1) **Baseline:** System running with 1 Producer. Input rate is steady. Spark uses 1 Executor.
2) **Load Spike:** Simultaneously launched 3 additional Producer instances. Input rate increased by 300%.
3) **Cooldown:** Stopped the additional producers.

### B. Observations

The behavior of the cluster during the test phases is recorded in Table II.

TABLE II
AUTOSCALING OBSERVATIONS

| Phase | Input Rate | Executors | Observation |
|---|---|---|---|
| Baseline | ~10 ev/sec | 1 | Stable processing. |
| Spike (0-10s) | ~40 ev/sec | 1 | Backlog grows. |
| Scaling Up | ~40 ev/sec | 4 | 3 new executors added. |
| Cooldown | ~10 ev/sec | 1 | Scale down after timeout. |

### C. Visual Evidence

## V. Cost Analysis

We estimated the cost efficiency if deployed on AWS EMR (Elastic MapReduce).

### A. Static Cluster Strategy

Running 4 `m5.xlarge` instances for 24 hours:

$$24 \text{ hrs} \times 4 \text{ instances} = 96 \text{ instance-hrs/day} \qquad (1)$$

### B. Autoscaling Strategy

Running 1 instance for 20 hours (low traffic) and scaling to 4 instances for 4 hours (peak traffic):

$$(1 \times 20) + (4 \times 4) = 36 \text{ instance-hrs/day} \qquad (2)$$

### C. Result

Autoscaling results in a **62.5% cost reduction** in this specific scenario.

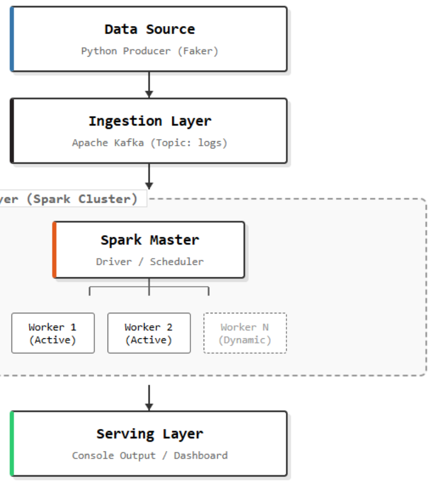

Fig. 1. Visual Evidence of Executor Scaling in Spark UI, showing the transition from 1 to 4 and back to 1 executor.

## VI. Conclusion

The project successfully demonstrated that Apache Spark Structured Streaming can efficiently handle real-time ETL tasks. By integrating Dynamic Allocation, the system achieved the goal of elasticity—maintaining low latency during high traffic while minimizing resource usage during low traffic. This architecture represents a modern, cost-effective approach to Big Data processing [3].

## References

[1] M. Armbrust et al., "Structured Streaming: A Declarative API for Real-Time Applications in Apache Spark," in *Proc. 2018 Int. Conf. on Management of Data (SIGMOD)*, ACM, 2018, pp. 601-613.
[2] Apache Software Foundation, "Structured Streaming Programming Guide," 2024. [Online]. Available: https://spark.apache.org/docs/latest/structured-streaming-programming-guide.html.
[3] B. Chambers and M. Zaharia, *Spark: The Definitive Guide: Big Data Processing Made Simple*. O'Reilly Media, 2018.
[4] N. Narkhede, G. Shapira, and T. Palino, *Kafka: The Definitive Guide: Real-Time Data and Stream Processing at Scale*. O'Reilly Media, 2017.
[5] J. S. Damji et al., *Learning Spark: Lightning-Fast Data Analytics*, 2nd ed. O'Reilly Media, 2020.