

# Lab Report: Spark Structured Streaming with Kafka

## Connecting to Confluent Cloud and Word Count Processing

OUEDRAOGO Kiswendsida Ulrich A.

October 23, 2025

## 1 Introduction

This report documents the implementation of a Spark Structured Streaming application that connects to a Kafka cluster (Confluent Cloud) and performs word count operations on streaming data. The lab involved setting up Apache Spark on Windows, configuring the Python environment, and resolving various compatibility issues.

**Source Code Repository:** All lab assignments and code for this course are hosted on GitHub at: <https://github.com/oulrich-ops/kafka-confluence>

## 2 Environment Setup

### 2.1 Apache Spark Installation

We installed Apache Spark version 3.5.7 (prebuilt for Hadoop 3.3) on Windows. The installation was performed following the tutorial provided in the lab instructions.

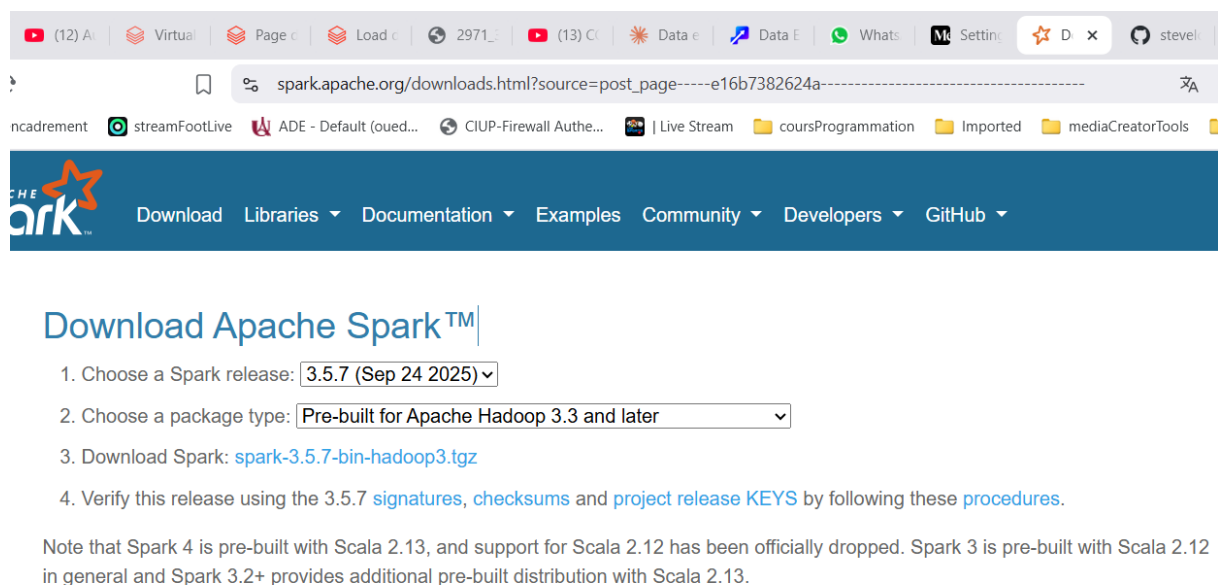


Figure 1: Apache Spark 3.5.7 installation verification

### 2.2 Environment Variables Configuration

After installing Java and Spark, we set up three Windows environment variables: `JAVA_HOME` (Java JDK path), `SPARK_HOME` (Spark installation path), and `HADOOP_HOME` (Hadoop utilities path).



Figure 2: Windows environment variables configuration

## 2.3 Windows Compatibility: Hadoop Winutils

**Important Note:** The first GitHub link mentioned in the Medium tutorial for Hadoop Winutils was outdated and incompatible with Spark 3.5.x. We used the second link provided in the lab description instead:

```
1 https://github.com/kontext-tech/winutils/tree/master
```

Listing 1: Correct Winutils Repository

This repository provides compatible versions of `winutils.exe` and `hadoop.dll` for Spark 3.5.x (prebuilt with Hadoop 3.3.x).

## 2.4 PySpark Verification

After installation, we verified the setup by running the `pyspark` command in the console:

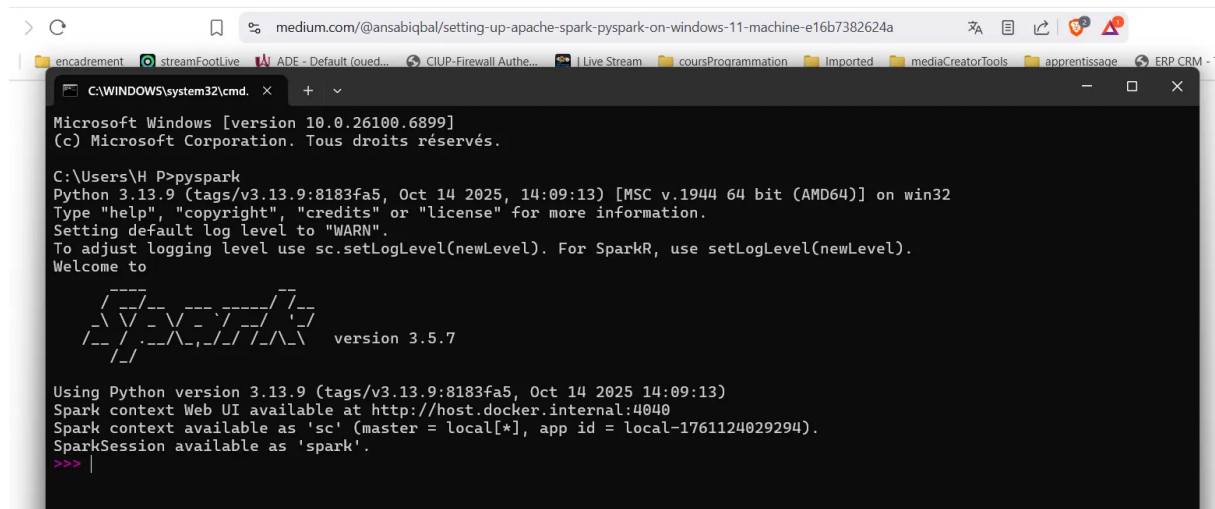


Figure 3: PySpark console showing version 3.5.7

# 3 Python Environment Setup

## 3.1 Miniconda Installation

We installed Miniconda for Windows to manage our Python environment. Following the lab instructions, we created a virtual environment using:

```
1 conda create -n cenv python=3.10 pyspark
```

Listing 2: Initial Environment Creation (from Lab Instructions)

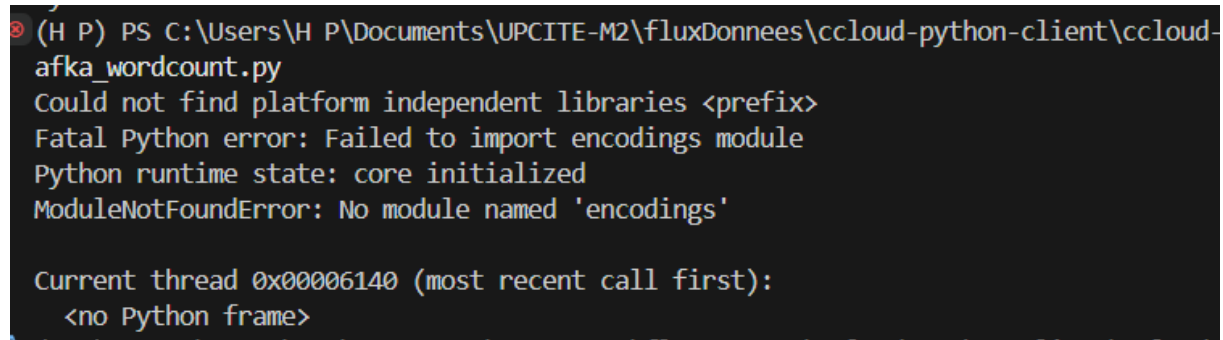
## 3.2 Version Conflicts and Resolution

### 3.2.1 Problem 1: PySpark Version Conflict

The initial installation installed PySpark version 4.0, which was incompatible with our Spark 3.5.7 installation. This caused JavaPackage errors when trying to connect to Kafka.

### 3.2.2 Problem 2: Python Version Conflict

Additionally, we encountered conflicts with Python version 3.13 on our system when the virtual environment was activated, causing compatibility issues.



```
(H P) PS C:\Users\H P\Documents\UPCITE-M2\fluxDonnees\ccloud-python-client\ccloud-afka_wordcount.py
Could not find platform independent libraries <prefix>
Fatal Python error: Failed to import encodings module
Python runtime state: core initialized
ModuleNotFoundError: No module named 'encodings'

Current thread 0x00006140 (most recent call first):
<no Python frame>
```

Figure 4: Console showing fatal python error

### 3.2.3 Solution: Conda Init and Environment Recreation

To resolve the Python version conflict, we first ran:

```
1 conda init
```

Listing 3: Making Conda Paths Priority

This command configures the shell to prioritize Conda paths when the environment is activated.

Then, we completely recreated the environment with the correct versions:

```
1 # Deactivate current environment
2 conda deactivate
3
4 # Remove the problematic environment
5 conda remove -n cvenv --all
6
7 # Create new environment with Python 3.10
8 conda create -n cvenv python=3.10 -y
9
10 # Activate the environment
11 conda activate cvenv
12
13 # Install PySpark 3.5.0 (matching our Spark installation)
14 conda install -c conda-forge pyspark=3.5.0 -y
```

Listing 4: Environment Recreation with Correct Versions

After these steps, the environment was stable and compatible with our Spark installation.

## 4 Kafka Integration

### 4.1 Loading the Kafka Driver

A critical step for connecting Spark to Kafka is loading the appropriate Kafka connector driver. PySpark does not include Kafka support by default, so we must explicitly load the `spark-sql-kafka` package.

We configured this using the `PYSPARK_SUBMIT_ARGS` environment variable, which must be set **before** importing PySpark:

```
1 import os
2 os.environ['PYSPARK_SUBMIT_ARGS'] = \
3     '--packages org.apache.spark:spark-sql-kafka-0-10_2.12:3.5.0 pyspark-shell'
```

Listing 5: Loading Kafka Driver via Environment Variable

Mismatched versions will result in `JavaPackage` or `NoClassDefFoundError` exceptions.

#### 4.1.1 Alternative Method

Alternatively, the driver can be loaded directly in the `SparkSession` configuration:

```
1 spark = SparkSession \
2     .builder \
3     .config("spark.jars.packages",
4            "org.apache.spark:spark-sql-kafka-0-10_2.12:3.5.0") \
5     .getOrCreate()
```

Listing 6: Alternative: Loading via SparkSession Config

Both methods are functionally equivalent.

However, the configuration method is more explicit and centralizes all Spark settings in one location.

This tells PySpark to download and load the Kafka connector package that matches our Spark version (3.5.0) and Scala version (2.12).

### 4.2 Confluent Cloud Configuration

Our Kafka cluster is hosted on Confluent Cloud, which requires SASL/SSL authentication. The connection parameters include:

- Bootstrap servers: `pkc-60py3.europe-west9.gcp.confluent.cloud:9092`
- Security protocol: `SASL_SSL`
- SASL mechanism: `PLAIN`
- Authentication credentials (username and password)

## 5 Implementation

### 5.1 Complete Python Code

Below is the complete implementation of our Spark Structured Streaming application that reads from Kafka and performs word count:

```

1 from pyspark.sql import SparkSession
2 from pyspark.sql.functions import explode, split
3
4 # Create Spark Session with Kafka connector package loading
5 spark = SparkSession \
6     .builder \
7     .master("local[*]") \
8     .config("spark.jars.packages",
9            "org.apache.spark:spark-sql-kafka-0-10_2.12:3.5.0") \
10    .appName("StructuredKafkaWordCount") \
11    .getOrCreate()
12
13 # Read from Kafka with Confluent Cloud configuration
14 df = spark \
15     .readStream \
16     .format("kafka") \
17     .option("kafka.bootstrap.servers",
18            "pkc-60py3.europe-west9.gcp.confluent.cloud:9092") \
19     .option("kafka.security.protocol", "SASL_SSL") \
20     .option("kafka.sasl.mechanism", "PLAIN") \
21     .option("kafka.sasl.jaas.config",
22            'org.apache.kafka.common.security.plain.PlainLoginModule '
23            'required username="FCLAEESPDSIC7UXR" '
24            'password="cfltt3mqlGreq7TUTihrQURLGc89LrQPW//flojUU+1Npqh88mMXbE/'
25            'OHMZvfWYQ";') \
26     .option("subscribe", "log_web") \
27     .option("startingOffsets", "earliest") \
28     .load()
29
30 # Convert Kafka value to string
31 words = df.selectExpr("CAST(value AS STRING) as line")
32
33 # Split lines into words
34 words_exploded = words.select(
35     explode(split(words.line, " ")).alias("word")
36 )
37
38 # Count word occurrences
39 word_counts = words_exploded.groupBy("word").count()
40
41 # Write results to console
42 query = word_counts \
43     .writeStream \
44     .outputMode("complete") \
45     .format("console") \
46     .start()
47
48 # Wait for termination
49 query.awaitTermination()

```

Listing 7: Spark Structured Streaming Word Count Application

## 6 Results

### 6.1 Kafka Topic Data

Our Kafka topic `log_web` contains 112 messages with web log data in JSON format:

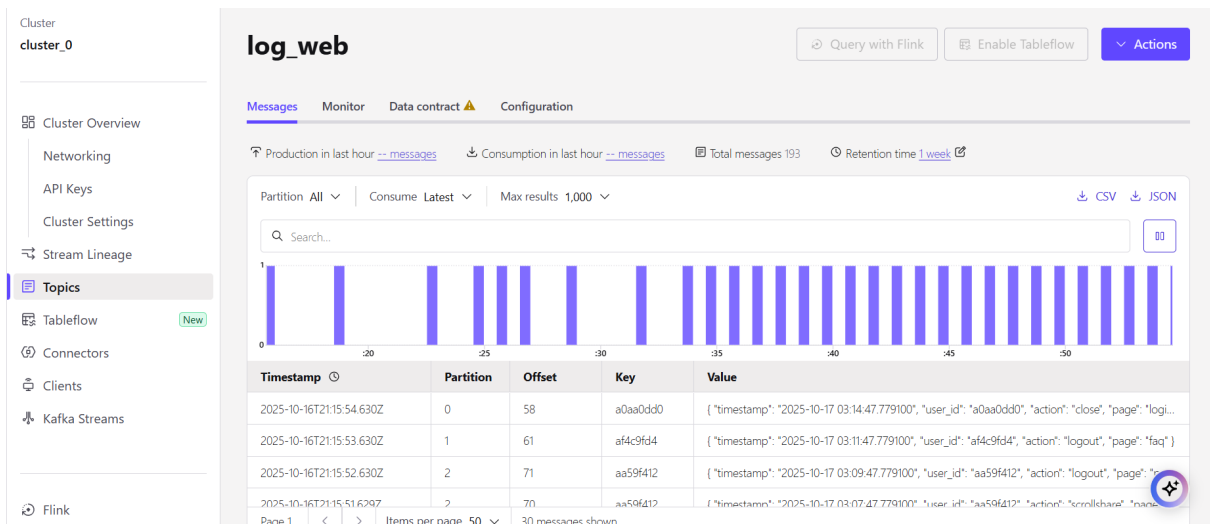


Figure 5: Kafka topic content showing web log messages

## 6.2 Console Output

When running the application, we successfully connected to Confluent Cloud and processed all messages. The console output shows the word count results:

```
(cvenv) PS C:\Users\H\P\Documents\UPCITE-M2\FluxDonnees\ccloud-python-client\ccloud-python-client\td03_Spark_Structured_Streaming> pyth
hon .\structured_kafka_wordcount.py
25/10/22 16:17:31 WARN ResolveWriteToStream: spark.sql.adaptive.enabled is not supported in streaming DataFrames/Datasets and will be
disabled.
25/10/22 16:17:32 WARN AdminClientConfig: These configurations '[key.deserializer, value.deserializer, enable.auto.commit, max.poll.re
cords, auto.offset.reset]' were supplied but are not used yet.
-----
Batch: 0
-----
+-----+-----+
|          word|count|
+-----+-----+
|01:39:47.779100",| 1|
|22:15:47.779100",| 1|
|  "1d6f1f65",| 9|
|  "cart"}| 18|
|00:52:47.779100",| 2|
|22:41:47.779100",| 2|
|01:34:47.779100",| 2|
|02:35:47.779100",| 1|
|21:38:47.779100",| 1|
|  "add_to cart",| 11|
|00:42:47.779100",| 1|
|02:21:47.779100",| 2|
|  "close",| 25|
|23:21:47.779100",| 1|
|23:10:47.779100",| 2|
|22:08:47.779100",| 2|
|22:29:47.779100",| 2|
|02:24:47.779100",| 1|
|21:26:47.779100",| 1|
|00:06:47.779100",| 1|
+-----+-----+
only showing top 20 rows
```

Figure 6: Spark Structured Streaming console output showing word counts

The output shows various words extracted from the JSON data, including timestamps, action types, pages, user\_id from the precedent lab.

## 7 Alternative Approach: Command Line Arguments

The lab instructions also suggested an alternative approach using command-line arguments, as shown in the provided template:

```

1 """
2 Consumes messages from one or more topics in Kafka and does wordcount.
3 Usage:
4     python structured_kafka_wordcount.py <bootstrap-servers>
5         <subscribe-type> <topics>
6
7 Where:
8     - bootstrap-servers: The Kafka "bootstrap.servers" configuration
9     - subscribe-type: 'assign', 'subscribe', or 'subscribePattern'
10    - topics: Topic specification (format depends on subscribe-type)
11 """

```

Listing 8: Alternative Approach from Lab Template

However, we chose to embed the configuration directly in the code for simplicity and to avoid exposing sensitive credentials in command-line arguments.

## 8 Challenges and Lessons Learned

### 8.1 Main Challenges

1. **Version Compatibility:** The biggest challenge was ensuring compatibility between Spark (3.5.7), PySpark (3.5.0), and the Kafka connector (3.5.0)
2. **Windows-Specific Issues:** Finding compatible Hadoop utilities for Windows required using an alternative GitHub repository
3. **Python Environment:** Managing Python versions and isolating dependencies using Conda
4. **Confluent Cloud Authentication:** Properly configuring SASL/SSL credentials for cloud-hosted Kafka

### 8.2 Key Takeaways

- Always match the version numbers across Spark, PySpark, and connector packages
- Use Conda environments to isolate project dependencies
- For Windows Spark installations, verify that Hadoop utilities match your Spark version
- Cloud Kafka services require additional security configuration compared to local installations

## 9 Conclusion

We successfully implemented a Spark Structured Streaming application that connects to Confluent Cloud Kafka and performs word count operations on streaming data. Despite several challenges related to version compatibility and Windows-specific issues, we were able to resolve all problems through systematic troubleshooting.

The final solution works reliably and processes 112 messages from the `log_web` topic, demonstrating the power of Spark Structured Streaming for real-time data processing.

## 10 Appendix: System Configuration

Component	Version/Configuration
Operating System	Windows
Java	JDK 8
Apache Spark	3.5.7 (prebuilt for Hadoop 3.3)
PySpark	3.5.0
Python	3.10
Kafka Connector	spark-sql-kafka-0-10_2.12:3.5.0
Scala	2.12
Conda	Miniconda (latest)
Kafka Cluster	Confluent Cloud (europe-west9.gcp)

Table 1: System configuration summary