Lab 2: Basic ETL - Filtering and Routing a Data Stream

**Goal:** Connect a PyFlink application to a live Kafka data stream, process JSON events, and perform basic ETL (Extract, Transform, Load) operations like filtering and mapping.

# 

# INDEX

[**Purpose of this Lab**](#_h6emjt6wnn5a) **3**

[**Prerequisites**](#_gnnu8yll5snm) **3**

[**Project Structure**](#_s9ozho8ajv7f) **4**

[Part 1: Project and Environment Setup](#_nehp7g9g18j) 5

[Part 2: Setting Up the Kafka Cluster](#_rmjizubdp2yp) 8

[Part 3: Developing the Flink ETL Application](#_5bsu9je1ss7d) 9

[Part 4: Executing the End-to-End Pipeline](#_bhzbmd1nc63n) 12

[Part 5: Verification](#_6ptotpeulne9) 13

[Part 6: Cleanup](#_ptniatdtxd1v) 14

[Part 7: Next Steps](#_dobobxfgq3e0) 15

# 

# Purpose of this Lab

This lab builds on the local cluster from Lab 1 by introducing a real-world streaming data pipeline. You will consume a continuous stream of simulated website clickstream events from a Kafka topic, a common pattern in data engineering. By completing this lab, you will:

* **Provision a Data Source:** Deploy a local Zookeeper and Kafka instance using Docker Compose.
* **Generate Real-Time Data:** Write and run a Python script to produce a continuous stream of JSON events to a Kafka topic.
* **Consume & Deserialize:** Configure Flink's KafkaSource to connect to Kafka and correctly parse incoming JSON data into structured objects.
* **Transform Data:** Apply filter and map transformations to the data stream to isolate specific events and extract relevant fields.
* **Execute an End-to-End Pipeline:** Manage all the moving parts (Flink Cluster, Kafka, Producer, Flink Job) to see the full data flow in action.

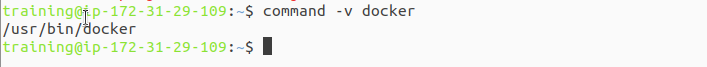
# Prerequisites

This lab assumes you have successfully completed Lab 1 and are using an **Ubuntu** environment.

**Docker and Docker Compose**

We need Docker to run our local Kafka cluster. First, check if you have Docker installed:

|  |
| --- |
| command -v docker |



If this command prints a path, you can move on. If not, follow these steps to install Docker Engine and Docker Compose.

Set up Docker's apt repository.

|  |
| --- |
| sudo apt-get update sudo apt-get install ca-certificates curl gnupg sudo install -m 0755 -d /etc/apt/keyrings curl -fsSL [https://download.docker.com/linux/ubuntu/gpg](https://download.docker.com/linux/ubuntu/gpg) | sudo gpg --dearmor -o /etc/apt/keyrings/docker.gpg sudo chmod a+r /etc/apt/keyrings/docker.gpg  echo \  "deb [arch=$(dpkg --print-architecture) signed-by=/etc/apt/keyrings/docker.gpg] [https://download.docker.com/linux/ubuntu](https://download.docker.com/linux/ubuntu) \  $(. /etc/os-release && echo "$VERSION\_CODENAME") stable" | \  sudo tee /etc/apt/sources.list.d/docker.list > /dev/null |

Install Docker Engine and Compose

|  |
| --- |
| sudo apt-get update sudo apt-get install docker-ce docker-ce-cli containerd.io docker-buildx-plugin docker-compose-plugin -y |

### 

# Project Structure

By the end of this lab, your new project directory will be structured as follows:

|  |
| --- |
| ~/flink-lab-2/ ├── jars/ # A temporary directory for downloading connectors │ └── flink-sql-connector-kafka-4.0.0-2.0.jar ├── venv/ # The isolated Python virtual environment ├── docker-compose.yaml # Defines our Kafka service ├── kafka\_etl.py # The Flink ETL job script └── producer.py # The script to generate mock data |

### **Part 1: Project and Environment Setup**

**Step 1: Create Project Directory and Virtual Environment**

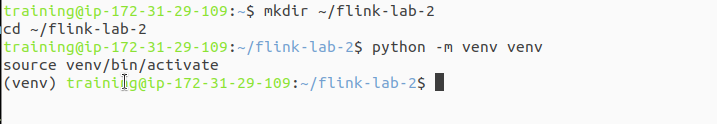
We'll create a new, separate directory for this lab to keep things organized.

Create and navigate to the new lab directory

|  |
| --- |
| mkdir ~/flink-lab-2 cd ~/flink-lab-2 |

Initialize and activate a Python virtual environment

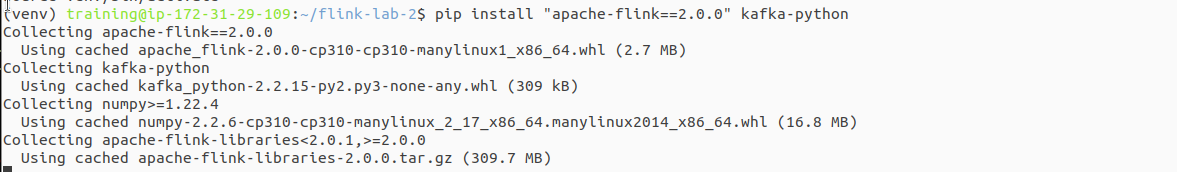
|  |
| --- |
| python -m venv venv source venv/bin/activate |



**Step 2: Install Python Dependencies**

With the venv active, install apache-flink and the Python client for Kafka.

|  |
| --- |
| pip install "apache-flink==2.0.0" kafka-python |



**Step 3: Configure Flink for the New Project**

Since we created a new virtual environment, we must update Flink's configuration to point to the correct Python executable for this lab.

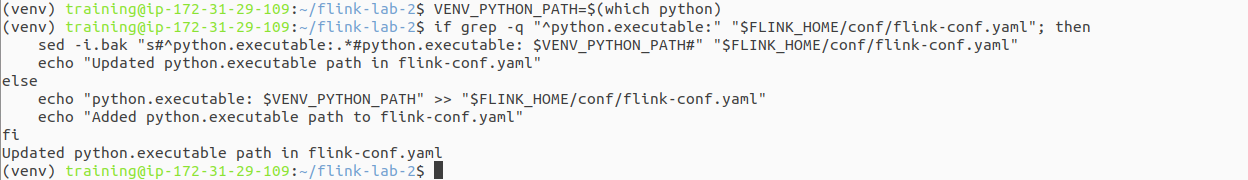
Get the absolute path to the Python executable in the new venv

|  |
| --- |
| VENV\_PYTHON\_PATH=$(which python) |



This command finds and replaces the 'python.executable' line, or adds it if not present.

|  |
| --- |
| if grep -q "^python.executable:" "$FLINK\_HOME/conf/flink-conf.yaml"; then  sed -i.bak "s#^python.executable:.\*#python.executable: $VENV\_PYTHON\_PATH#" "$FLINK\_HOME/conf/flink-conf.yaml"  echo "Updated python.executable path in flink-conf.yaml" else  echo "python.executable: $VENV\_PYTHON\_PATH" >> "$FLINK\_HOME/conf/flink-conf.yaml"  echo "Added python.executable path to flink-conf.yaml" fi |

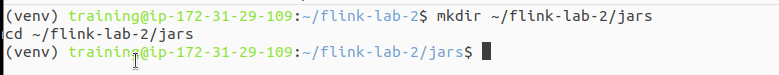


**Step 4: Download the Flink Kafka Connector**

Flink needs a special connector to communicate with Kafka. We will download the required .jar file into a temporary jars directory.

Create a directory to hold the connector jar

|  |
| --- |
| mkdir ~/flink-lab-2/jars cd ~/flink-lab-2/jars |



Download the specific Kafka connector for Flink 2.0

|  |
| --- |
| wget https://repo.maven.apache.org/maven2/org/apache/flink/flink-sql-connector-kafka/4.0.0-2.0/flink-sql-connector-kafka-4.0.0-2.0.jar |



Go back to the main project directory

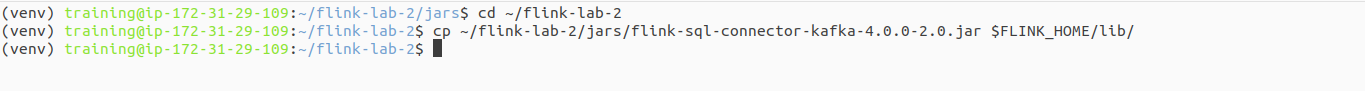
|  |
| --- |
| cd ~/flink-lab-2 |

**Step 5: Make the Connector Available to the Cluster**

This is the most reliable way to add a connector for local development. We will copy the JAR file directly into your Flink installation's lib folder. Any JARs in this folder are automatically loaded when the cluster starts.

Copy the downloaded JAR into the Flink lib directory

|  |
| --- |
| cp ~/flink-lab-2/jars/flink-sql-connector-kafka-4.0.0-2.0.jar $FLINK\_HOME/lib/ |



You may need to restart the Flink cluster (stop-cluster.sh then start-cluster.sh) if it was already running when you performed this step.

### **Part 2: Setting Up the Kafka Cluster**

**Step 1: Define the Kafka Service**

Create a file named docker-compose.yaml in the ~/flink-lab-2 directory.

|  |
| --- |
| code docker-compose.yaml |

Add the following content to the file:

|  |
| --- |
| # docker-compose.yaml services:  zookeeper:  image: confluentinc/cp-zookeeper:7.3.2  container\_name: zookeeper  ports: ["2181:2181"]  environment:  ZOOKEEPER\_CLIENT\_PORT: 2181  ZOOKEEPER\_TICK\_TIME: 2000  kafka:  image: confluentinc/cp-kafka:7.3.2  container\_name: kafka  ports: ["9092:9092"]  depends\_on: [zookeeper]  environment:  KAFKA\_BROKER\_ID: 1  KAFKA\_ZOOKEEPER\_CONNECT: zookeeper:2181  KAFKA\_ADVERTISED\_LISTENERS: PLAINTEXT://kafka:29092,PLAINTEXT\_HOST://localhost:9092  KAFKA\_LISTENER\_SECURITY\_PROTOCOL\_MAP: PLAINTEXT:PLAINTEXT,PLAINTEXT\_HOST:PLAINTEXT  KAFKA\_INTER\_BROKER\_LISTENER\_NAME: PLAINTEXT  KAFKA\_OFFSETS\_TOPIC\_REPLICATION\_FACTOR: 1 |

**Step 2: Launch the Kafka Cluster**

From the ~/flink-lab-2 directory, start the services.

|  |
| --- |
| docker compose up -d |



### **Part 3: Developing the Flink ETL Application**

Step 1: Implement the Kafka Producer

Create a file named producer.py. This script includes retry logic to wait for Kafka to be ready.

|  |
| --- |
| code producer.py |

Add the following code:

|  |
| --- |
| # producer.py import json import time import random from kafka import KafkaProducer from kafka.errors import NoBrokersAvailable  KAFKA\_TOPIC = 'clicks' KAFKA\_BROKERS = 'localhost:9092'  def create\_producer():  """Creates a KafkaProducer with retry logic."""  retries = 10  while retries > 0:  try:  producer = KafkaProducer(  bootstrap\_servers=KAFKA\_BROKERS,  value\_serializer=lambda v: json.dumps(v).encode('utf-8')  )  print("Successfully connected to Kafka.")  return producer  except NoBrokersAvailable:  retries -= 1  print(f"Kafka not available, retrying in 5 seconds... ({retries} retries left)")  time.sleep(5)  raise RuntimeError("Failed to connect to Kafka after multiple retries.")  if \_\_name\_\_ == '\_\_main\_\_':  producer = create\_producer()    print("Producing mock click events... Press Ctrl+C to terminate.")  user\_ids = [f'user\_{i}' for i in range(1, 11)]  product\_ids = [f'prod\_{i}' for i in range(1, 6)]  event\_types = ['page\_view', 'page\_view', 'page\_view', 'add\_to\_cart']  try:  while True:  event = {  'event\_type': random.choice(event\_types),  'user\_id': random.choice(user\_ids),  'product\_id': random.choice(product\_ids),  'timestamp': int(time.time() \* 1000)  }  producer.send(KAFKA\_TOPIC, value=event)  print(f"Sent event: {event}")  time.sleep(1)  except KeyboardInterrupt:  print("\nStopping producer.")  finally:  producer.flush()  producer.close() |

**Step 2: Implement the Flink ETL Script**

Create the main Flink application file, kafka\_etl.py.

|  |
| --- |
| code kafka\_etl.py |

Add the following code:

|  |
| --- |
| # kafka\_etl.py from pyflink.datastream import StreamExecutionEnvironment from pyflink.datastream.connectors.kafka import KafkaSource, KafkaOffsetsInitializer from pyflink.common.watermark\_strategy import WatermarkStrategy from pyflink.datastream.formats.json import JsonRowDeserializationSchema from pyflink.common.typeinfo import Types  def main():  env = StreamExecutionEnvironment.get\_execution\_environment()    type\_info = Types.ROW\_NAMED(  ["event\_type", "user\_id", "product\_id", "timestamp"],  [Types.STRING(), Types.STRING(), Types.STRING(), Types.LONG()]  )   json\_deserializer = JsonRowDeserializationSchema.builder() \  .type\_info(type\_info) \  .build()   kafka\_source = KafkaSource.builder() \  .set\_bootstrap\_servers('localhost:9092') \  .set\_topics('clicks') \  .set\_group\_id('flink-etl-group') \  .set\_starting\_offsets(KafkaOffsetsInitializer.latest()) \  .set\_value\_only\_deserializer(json\_deserializer) \  .build()   data\_stream = env.from\_source(  source=kafka\_source,  watermark\_strategy=WatermarkStrategy.no\_watermarks(),  source\_name="kafka\_source"  )   filtered\_stream = data\_stream.filter(  lambda event: event.event\_type == 'add\_to\_cart'  )   extracted\_stream = filtered\_stream.map(  lambda event: f"Add to Cart Event -> User: {event.user\_id}, Product: {event.product\_id}",  output\_type=Types.STRING()  )   extracted\_stream.print()  env.execute("kafka\_basic\_etl\_with\_schema")  if \_\_name\_\_ == '\_\_main\_\_':  main() |

### 

### **Part 4: Executing the End-to-End Pipeline**

You will need three separate terminal windows.

**Terminal 1: Start the Flink Cluster**

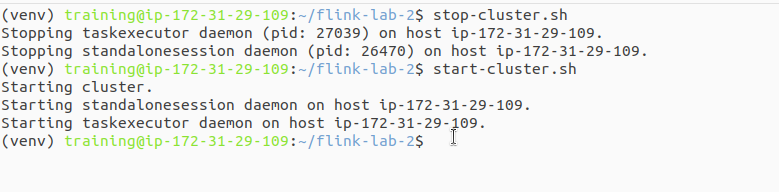
If your Flink cluster is not already running, start it now. (If you just added the JAR, restart it).

If running, stop it first

|  |
| --- |
| stop-cluster.sh |

Start the cluster

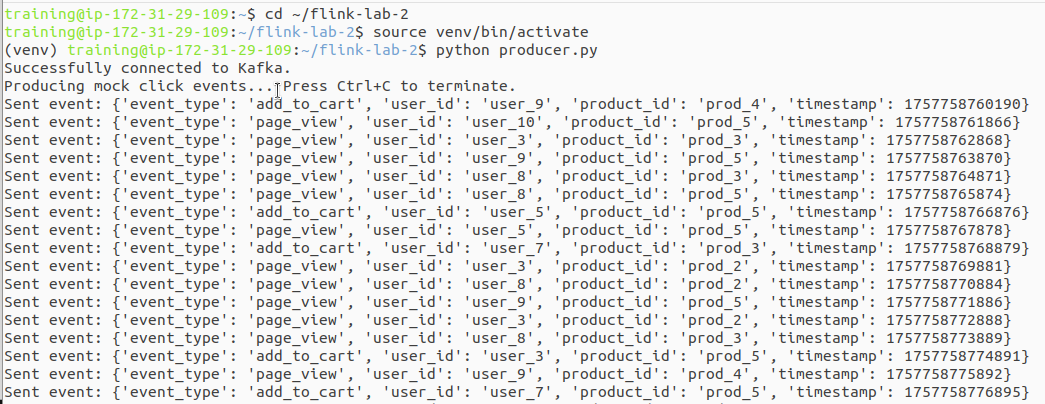
|  |
| --- |
| start-cluster.sh |



**Terminal 2: Launch the Data Producer**

Navigate to your lab directory, activate the environment, and start the producer script.

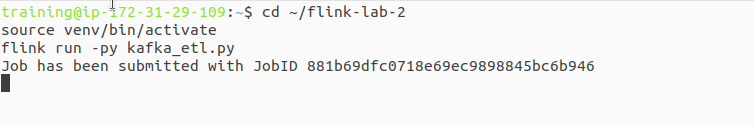
|  |
| --- |
| cd ~/flink-lab-2 source venv/bin/activate python producer.py |



**Terminal 3: Submit the Flink Application**

Now that the connector JAR is in Flink's lib folder, we can submit the job with a much simpler command.

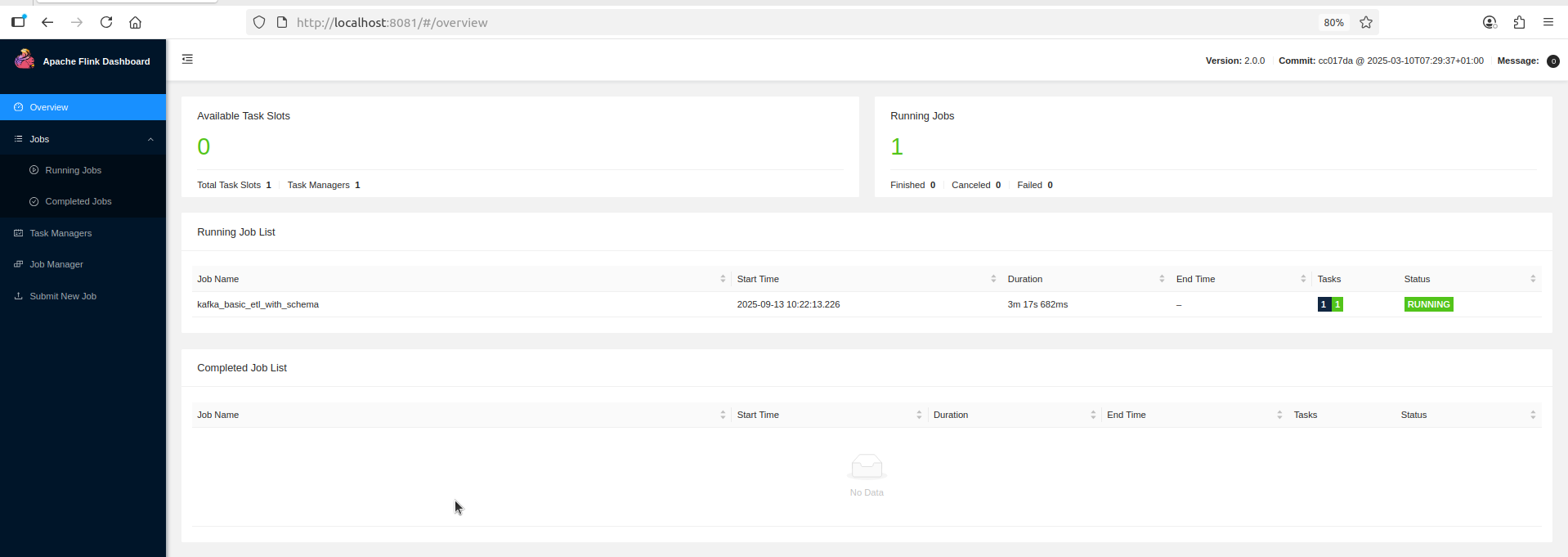
|  |
| --- |
| cd ~/flink-lab-2 source venv/bin/activate flink run -py kafka\_etl.py |

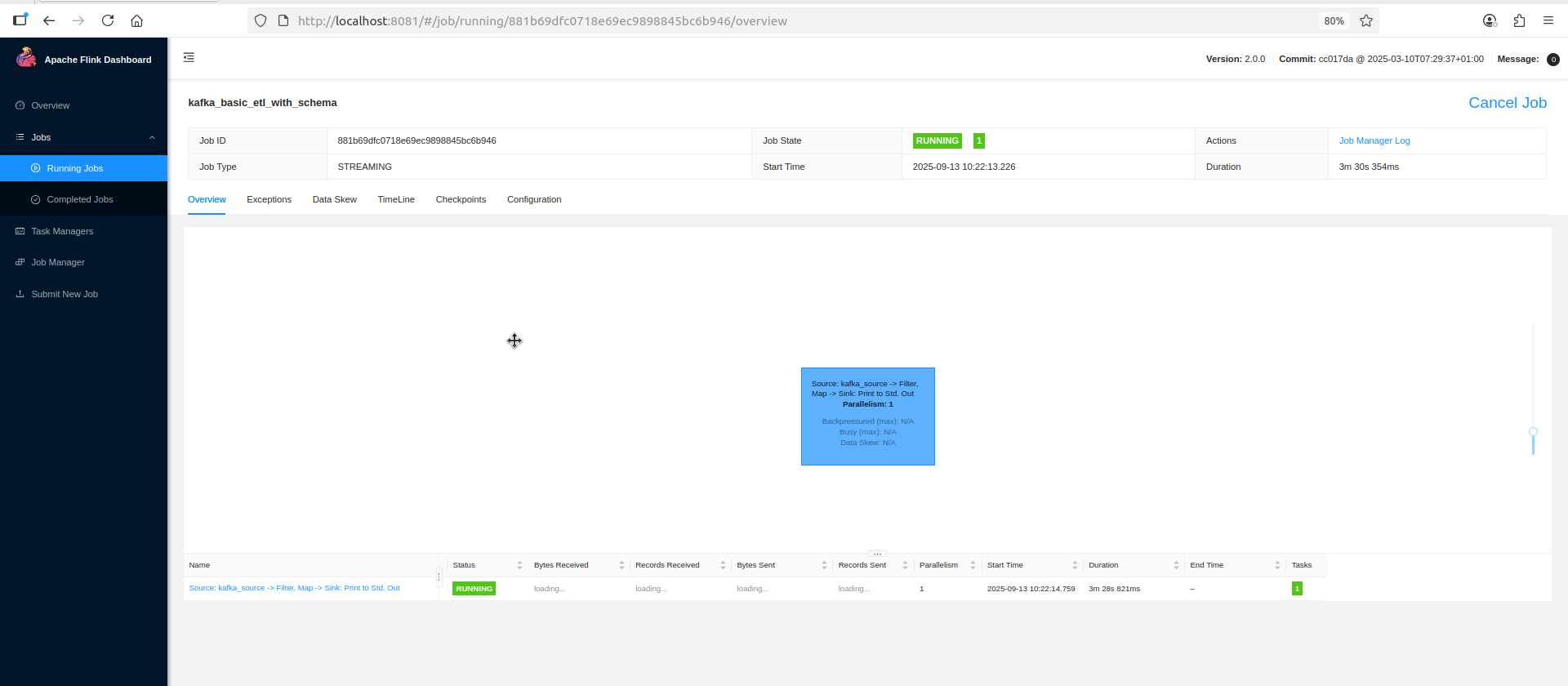


### **Part 5: Verification**

**Step 1: Verify Job Execution**

Go to the Flink UI at <http://localhost:8081>. The kafka\_basic\_etl\_with\_schema job should appear in the "Running Jobs" section.

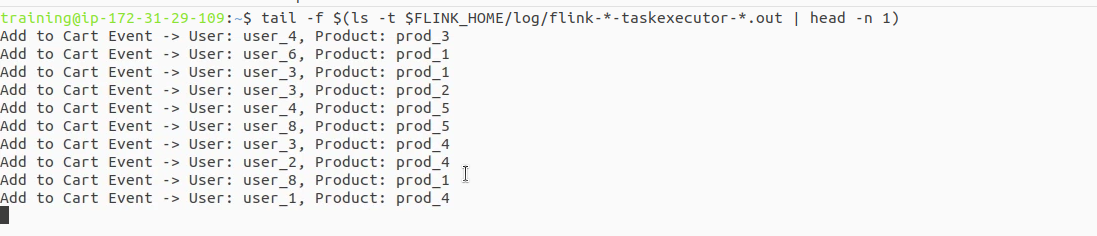




**Step 2: Inspect the Output**

View the TaskManager logs to see your processed data.

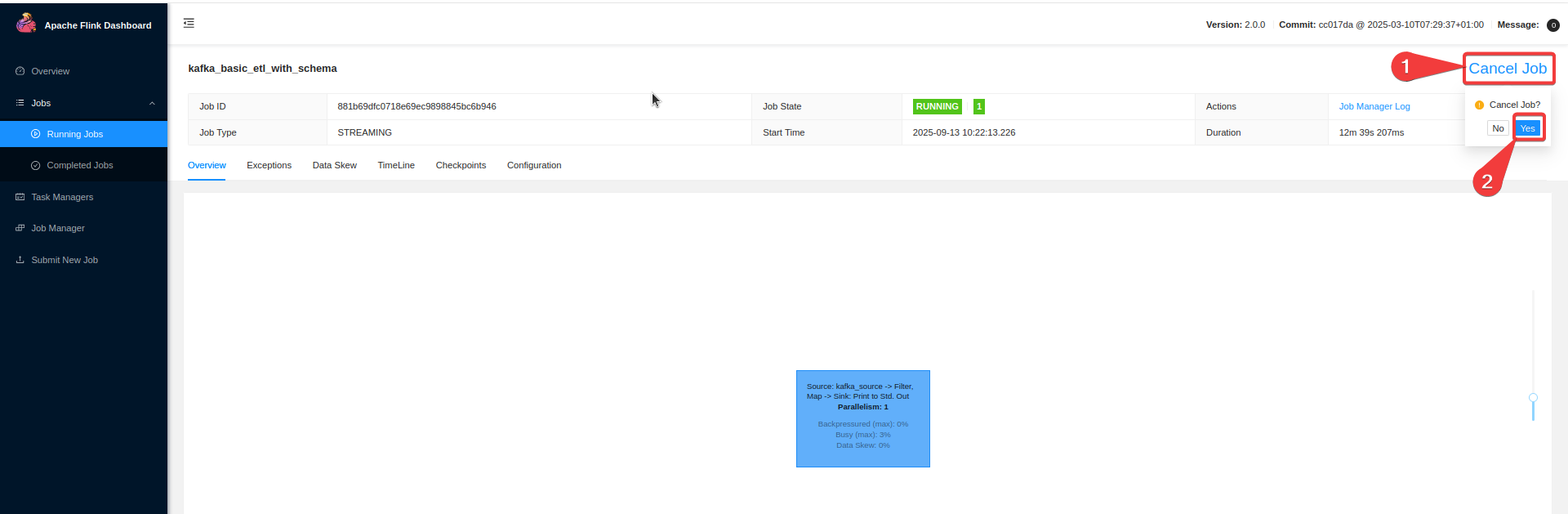
|  |
| --- |
| tail -f $(ls -t $FLINK\_HOME/log/flink-\*-taskexecutor-\*.out | head -n 1) |



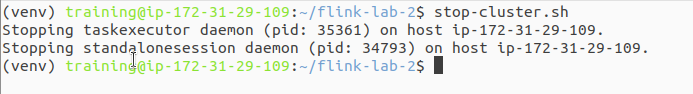
### **Part 6: Cleanup**

Once you have verified that the pipeline is working correctly, it's important to shut down all the components in the correct order to free up system resources.

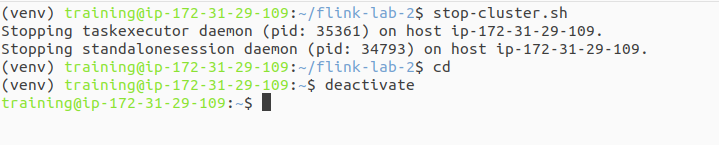
1. **Stop the producer:** Press Ctrl+C in Terminal 2.
2. **Stop the Flink job:** Cancel it from the Flink Web UI.



1. **Stop the Flink cluster:** stop-cluster.sh



1. **Stop the Kafka cluster:** cd ~/flink-lab-2 && docker compose down
2. To exit the Python virtual environment, simply run : deactivate



### **Part 7: Next Steps**

* **Stateful Transformations:** Explore stateful operations like key\_by() and reduce() to count events per user.
* **Windowing:** Use windowing functions to calculate metrics over specific time intervals.
* **Different Sinks:** Try writing the processed data to a different destination, like another Kafka topic or to files.