Lab 3: Keyed Streams and Stateful Timers

**Goal:** Introduce stateful stream processing by partitioning a data stream, applying stateful logic with a ProcessFunction, and using Flink's timer service for time-based event detection.

# 

# 

# 

# 

# INDEX

[**Purpose of this Lab**](#_fnof67l9qv1p) **3**

[**Prerequisites**](#_qd5yf2ti7wjv) **3**

[**Project Structure**](#_ysghzp10649u) **4**

[Part 1: Project and Environment Setup](#_6a8qc5emw3ew) 4

[Part 2: Setting Up the Kafka Cluster](#_ajuft9b21bhj) 6

[Part 3: Developing the Stateful Flink Application](#_ho1tyxym1o7b) 7

[Part 4: Executing the End-to-End Pipeline](#_bjdob46fbhck) 12

[Part 5: Verification](#_mhjsyibhvqa4) 13

[Part 6: Cleanup](#_9atjcm21g6iz) 15

[Part 7: Next Steps](#_87qhajjntprn) 16

# 

# Purpose of this Lab

This lab transitions from the stateless ETL of Lab 2 to stateful stream processing, a core strength of Apache Flink. You will count "add-to-cart" events for each product and, more importantly, detect when a product becomes "idle" (hasn't received an event for a specified duration).

This is achieved by partitioning the data stream with keyBy and then implementing a ProcessFunction, which gives you fine-grained control over both state and time. By completing this lab, you will:

**Partition a DataStream:** Logically group an unbounded stream into distinct partitions using the keyBy transformation.

**Implement a ProcessFunction:** Define and apply a ProcessFunction for advanced, stateful stream manipulation.

**Manage State:** Declare, access, and update Flink's managed ValueState within a keyed context.

**Use Timers:** Register and react to processing-time timers to implement custom, time-based application logic.

**Understand Stateful Execution:** Observe how Flink isolates state per key and manages the lifecycle of state and timers.

# Prerequisites

This lab assumes you have successfully completed Lab 1 and Lab 2 and are using an **Ubuntu** environment. Your Flink cluster should already have the Kafka connector JAR in its lib directory from the previous lab.

# Project Structure

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

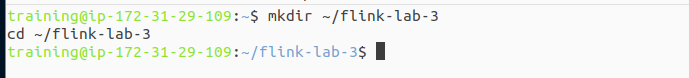
| ~/flink-lab-3/ ├── venv/ # The isolated Python virtual environment ├── docker-compose.yaml # Defines our Kafka service ├── producer.py # The script to generate mock data └── stateful\_timers.py # The Flink stateful job script |
| --- |

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

**Step 1: Create Project Directory and Virtual Environment** We'll create a new, separate directory for this lab.

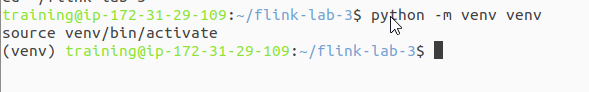
Create and navigate to the new lab directory

| mkdir ~/flink-lab-3 cd ~/flink-lab-3 |
| --- |



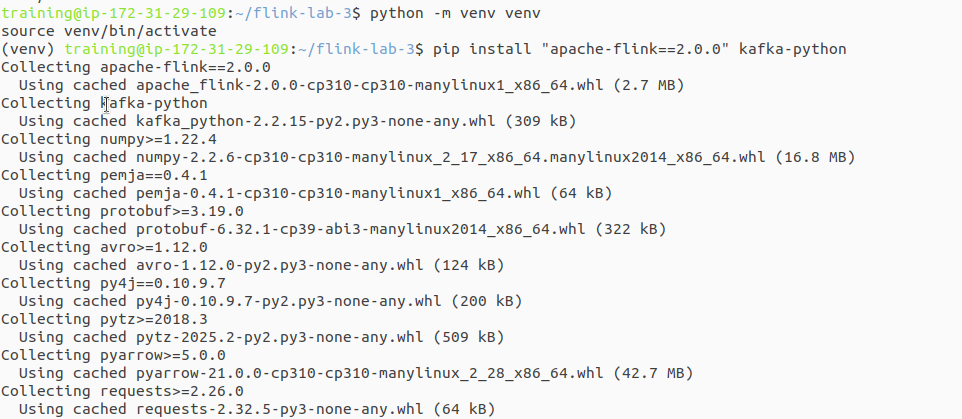
Initialize and activate a Python virtual environment

| 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 |
| --- |



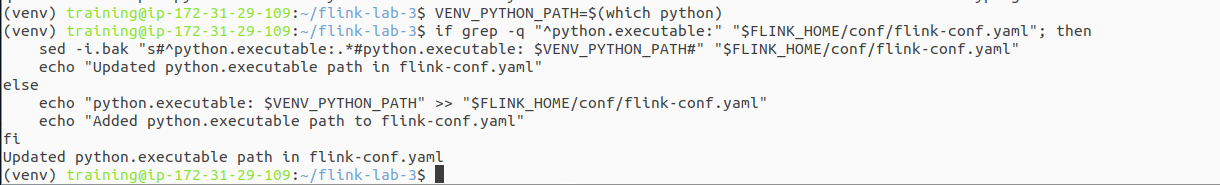
**Step 3: Configure Flink for the New Project** Since we created a new virtual environment, we must update Flink's configuration to point to the correct Python executable for this lab.

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 |
| --- |



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

**Step 1: Define the Kafka Service** Create a file named docker-compose.yaml in the ~/flink-lab-3 directory. This is identical to the file from Lab 2.

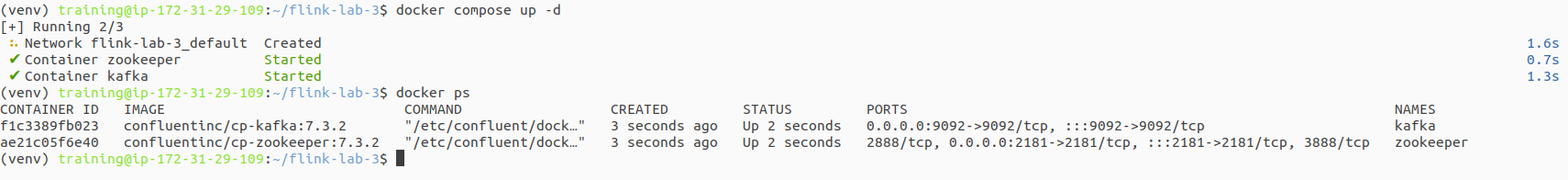
| code docker-compose.yaml |
| --- |

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-3 directory, start the services.

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



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

**Step 1: Implement the Kafka Producer** Create a file named producer.py. We will use the same robust producer with retry logic from Lab 2.

| 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 Stateful Flink Script** Create the main Flink application file, stateful\_timers.py.

| code stateful\_timers.py |
| --- |

Add the following code:

| # stateful\_timers.py from pyflink.common import Types from pyflink.datastream import StreamExecutionEnvironment from pyflink.datastream.connectors.kafka import KafkaSource, KafkaOffsetsInitializer from pyflink.datastream.formats.json import JsonRowDeserializationSchema from pyflink.common.watermark\_strategy import WatermarkStrategy from pyflink.datastream.functions import KeyedProcessFunction, RuntimeContext from pyflink.datastream.state import ValueStateDescriptor  class IdleProductDetector(KeyedProcessFunction):  """  A KeyedProcessFunction that maintains a count for each product\_id and  detects when a product has been "idle" (no events for 1 minute).  """   def \_\_init\_\_(self):  self.count\_state = None  self.timer\_state = None   def open(self, runtime\_context: RuntimeContext):  # Initialize state descriptors. This is called once per parallel instance.  count\_descriptor = ValueStateDescriptor("event\_count", Types.INT())  self.count\_state = runtime\_context.get\_state(count\_descriptor)   timer\_descriptor = ValueStateDescriptor("idle\_timer", Types.LONG())  self.timer\_state = runtime\_context.get\_state(timer\_descriptor)   def process\_element(self, value, ctx: 'KeyedProcessFunction.Context'):  # 1. Retrieve and update count state  current\_count = self.count\_state.value()  if current\_count is None:  current\_count = 0  new\_count = current\_count + 1  self.count\_state.update(new\_count)   # 2. Manage timers to detect idle products  # Delete the previous timer if it exists  previous\_timer\_ts = self.timer\_state.value()  if previous\_timer\_ts is not None:  ctx.timer\_service().delete\_processing\_time\_timer(previous\_timer\_ts)   # Register a new timer for 1 minute in the future  new\_timer\_ts = ctx.timer\_service().current\_processing\_time() + 60 \* 1000  ctx.timer\_service().register\_processing\_time\_timer(new\_timer\_ts)  self.timer\_state.update(new\_timer\_ts)   # Emit the current product and its updated count  yield f"Product: {ctx.get\_current\_key()}, Count: {new\_count}"   def on\_timer(self, timestamp: int, ctx: 'KeyedProcessFunction.OnTimerContext'):  # This method is invoked when a processing-time timer fires.  # Check if the timer that fired is the one we have stored.  stored\_timer\_ts = self.timer\_state.value()  if stored\_timer\_ts is not None and stored\_timer\_ts == timestamp:  product\_key = ctx.get\_current\_key()  event\_count = self.count\_state.value()  yield f"[IDLE ALERT] Product '{product\_key}' has been idle for 1 minute. Last count was {event\_count}."    # Clean up state for this key as it's now considered idle  self.count\_state.clear()  self.timer\_state.clear()  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-stateful-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"  )   # Filter for 'add\_to\_cart' events, key by product\_id, and apply the process function  keyed\_stream = data\_stream \  .filter(lambda event: event.event\_type == 'add\_to\_cart') \  .key\_by(lambda event: event.product\_id)   # Apply the custom KeyedProcessFunction  output\_stream = keyed\_stream.process(IdleProductDetector(), output\_type=Types.STRING())   output\_stream.print()  env.execute("stateful\_timers\_and\_keyed\_streams")  if \_\_name\_\_ == '\_\_main\_\_':  main() |
| --- |

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

You will need three separate terminal windows.

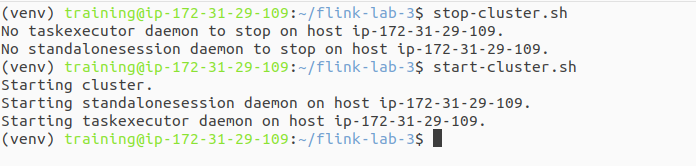
**Terminal 1: Start the Flink Cluster** If your Flink cluster is not already running, restart it to ensure it's using the latest configuration.

If running, stop it first

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

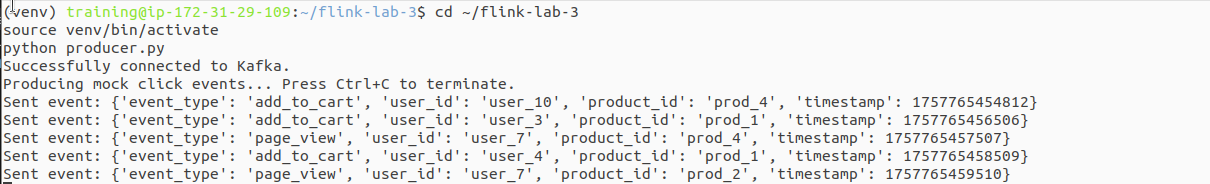
Start the cluster

| 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-3 source venv/bin/activate python producer.py |
| --- |



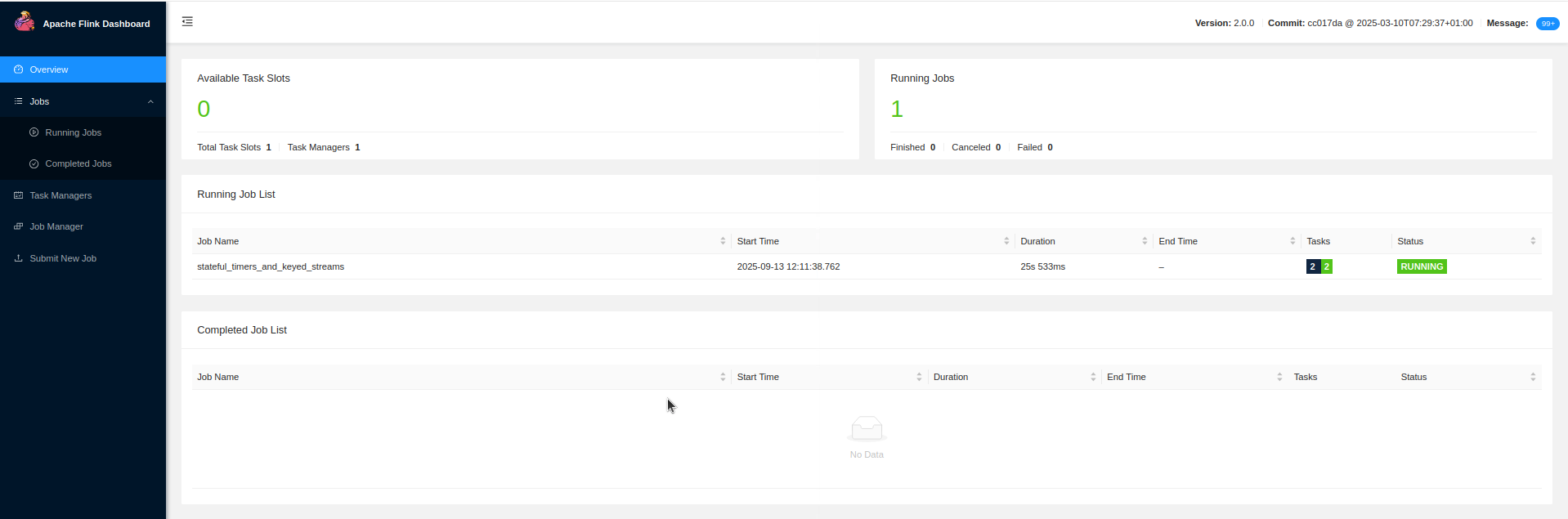
**Terminal 3: Submit the Flink Application** Because we placed the Kafka connector in Flink's lib directory, we can submit the job with a simple command.

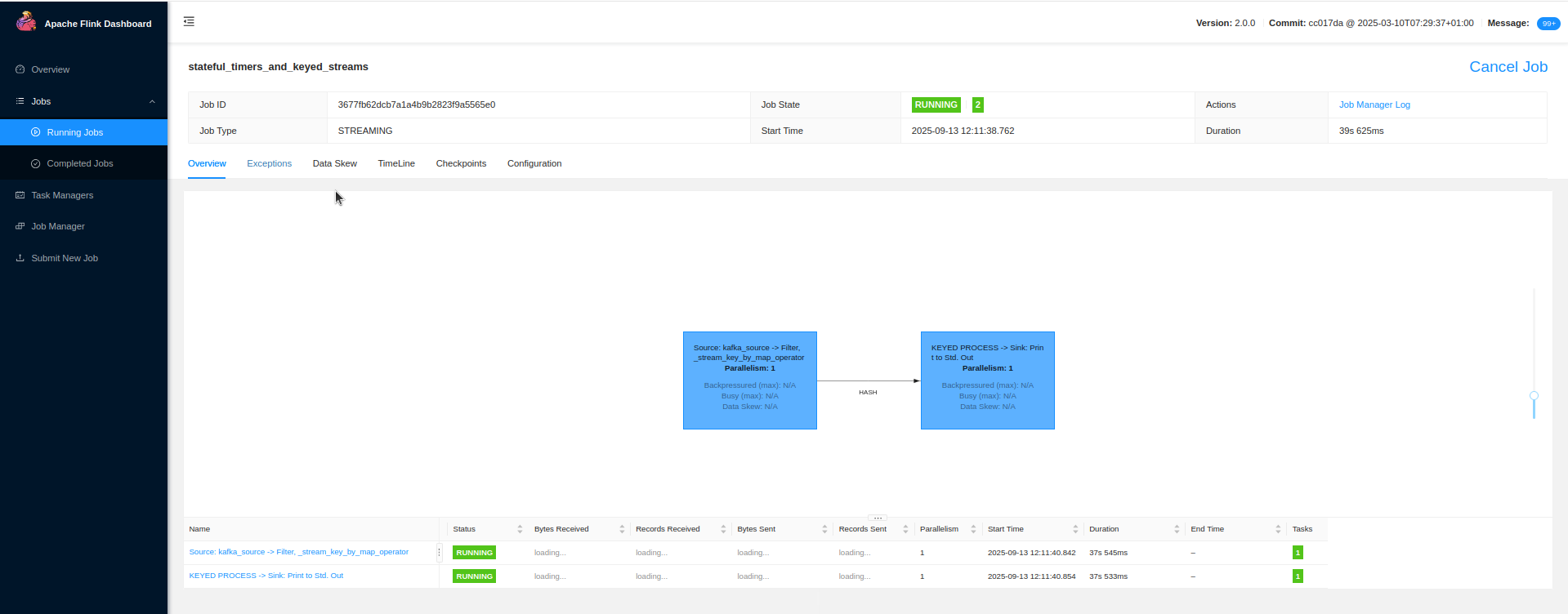
| cd ~/flink-lab-3 source venv/bin/activate flink run -py stateful\_timers.py |
| --- |

### 

### **Part 5: Verification**

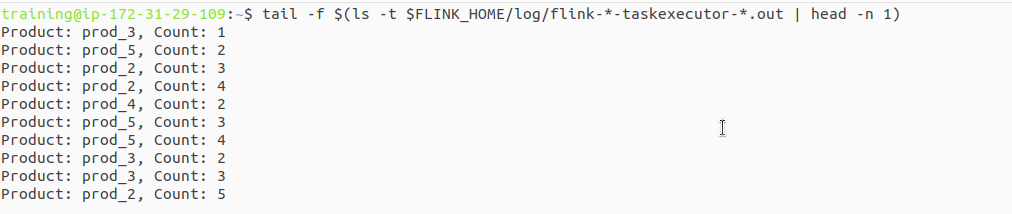
**Step 1: Verify Job Execution** Go to the Flink UI at http://localhost:8081. The stateful\_timers\_and\_keyed\_streams job should appear in the "Running Jobs" section.



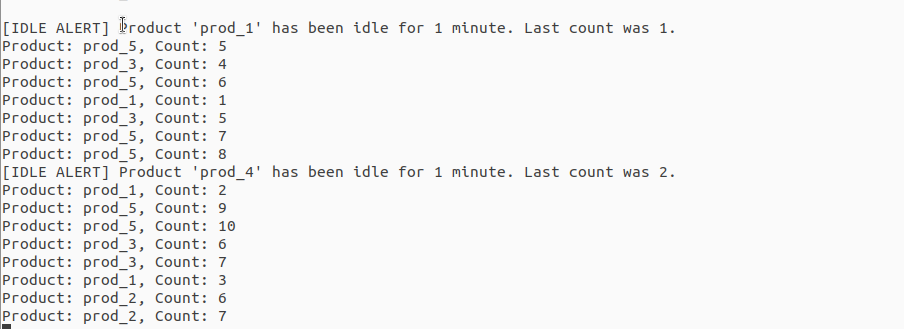


**Step 2: Inspect the Output** View the TaskManager logs to see the running counts for each product.

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



**Step 3: Wait for Idle Alerts** The producer script sends events for 5 different products. Wait for a minute or two. Due to the random nature of the producer, it is highly likely that at least one product will not receive an "add-to-cart" event for over 60 seconds. When this happens, you will see an idle alert in the logs: [IDLE ALERT] Product 'prod\_4' has been idle for 1 minute. Last count was 3.



### **Part 6: Cleanup**

Once you have verified the pipeline is working, shut down all the components to free up system resources.

**Stop the Flink job:** Press Ctrl+C in Terminal 3.

**Stop the producer:** Press Ctrl+C in Terminal 2.

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

**Stop the Kafka cluster:** cd ~/flink-lab-3 && docker compose down

**Deactivate the python environment:** deactivate in Terminals.

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

**Different State Types:** Explore other state types like ListState to store a history of events for each key.

**Event-Time Timers:** Modify the job to use event-time processing and timers, which are crucial for handling out-of-order data.

**Fault Tolerance:** Learn about Flink's checkpointing mechanism to see how it guarantees state consistency in case of failures.