Lab 11: Stateful Recovery and Application Upgrades

**Goal:** To understand and implement Flink’s core fault tolerance mechanisms using a local Flink cluster. You will configure a stateful PyFlink job with checkpointing, simulate a failure to observe automatic recovery, and then perform a planned application upgrade using a savepoint without any data loss.

### **INDEX**

* Purpose of this Lab
* Prerequisites
* Project Structure
* Part 1: Checkpoints vs. Savepoints
* Part 2: Project and Environment Setup
* Part 3: Configuring the Local Flink Cluster
* Part 4: Developing the Stateful Application (V1 & V2)
* Part 5: Scenario 1 - Automatic Recovery from Failure
* Part 6: Scenario 2 - Planned Upgrade with Savepoints
* Part 7: Cleanup
* Part 8: Next Steps

### **Purpose of this Lab**

Running a streaming application 24/7 in production requires robust mechanisms to handle both unexpected failures and planned maintenance. This lab explores Flink's two primary features for ensuring state consistency and operational flexibility.

First, you will configure **checkpointing**, which allows Flink to automatically recover the application's state after a failure (e.g., a machine crash or network issue). Second, you will use **savepoints** to perform a controlled upgrade of your application's code, demonstrating how to evolve your business logic without resetting your progress.

By completing this lab, you will:

* **Configure Checkpointing:** Enable and configure Flink's automatic state snapshotting mechanism on your local cluster.
* **Use Managed Keyed State:** Implement a stateful KeyedProcessFunction using ValueState to store data per key.
* **Simulate and Recover from Failure:** Manually kill a local TaskManager process and observe the job automatically restart from the last successful checkpoint.
* **Create and Restore from a Savepoint:** Take a portable, manual snapshot of the application's state and use it to launch an upgraded version of the job.
* **Understand Operational Best Practices:** Differentiate between the use cases for automated checkpoints (for recovery) and manual savepoints (for operations).

### **Prerequisites**

* An Ubuntu-based environment with Docker and Docker Compose installed.
* Successful completion of Labs 1, 2, and 3.
* A local Flink 2.0.0 installation ($FLINK\_HOME) configured, with the Kafka connector already available in the $FLINK\_HOME/lib directory from Lab 2.

### **Project Structure**

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

| ~/flink-lab-12/ ├── venv/ # Python virtual environment ├── docker-compose.yaml # Defines the Kafka service ONLY ├── producer.py # Script to generate mock data ├── stateful\_job\_v1.py # The initial version of our Flink app └── stateful\_job\_v2.py # The "upgraded" version of the Flink app |
| --- |

### **Part 1: Checkpoints vs. Savepoints**

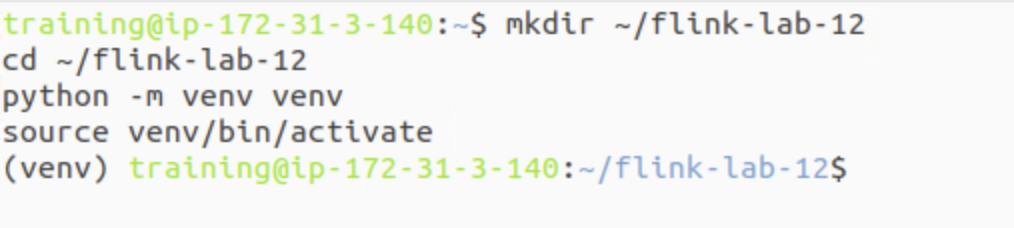
It's crucial to understand the distinction:

* **Checkpoints:**
* **Purpose:** Automatic recovery from unexpected job failures.
* **Mechanism:** Flink automatically takes periodic snapshots of the state.
* **Ownership:** Managed by Flink. Older checkpoints are often pruned automatically.
* **Savepoints:**
* **Purpose:** Planned manual operations (stopping/resuming, upgrades, migrations).
* **Mechanism:** Manually triggered by the user.
* **Ownership:** Managed by you. They are never deleted automatically.

### **Part 2: Project and Environment Setup**

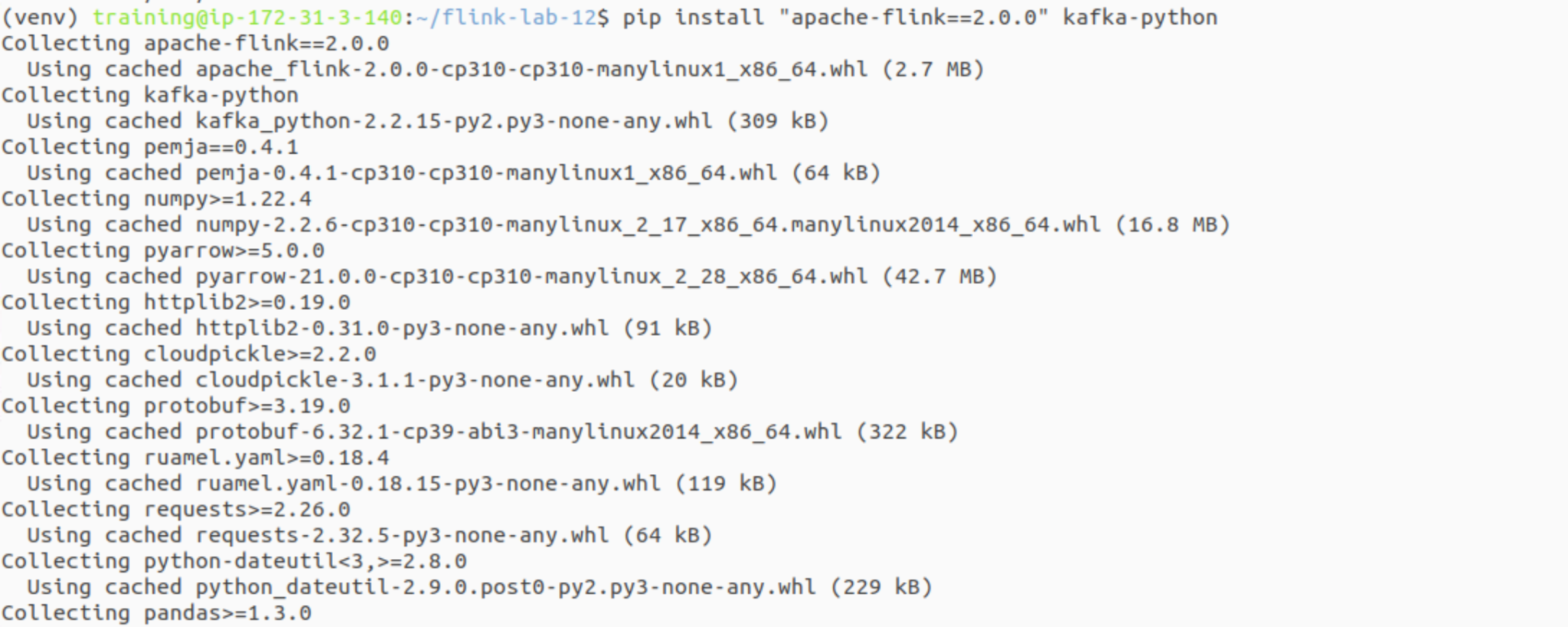
**Step 1: Create Project Directory and Virtual Environment**

| mkdir ~/flink-lab-12 cd ~/flink-lab-12 python -m venv venv source venv/bin/activate |
| --- |



**Step 2: Install Python Dependencies**

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



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

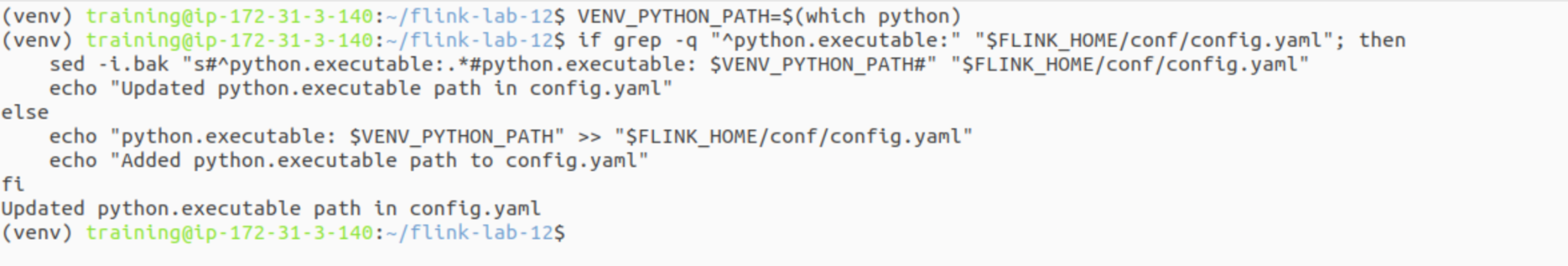
Since we created a new virtual environment, we must update Flink's configuration to point to the correct Python executable for this lab.

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



**Step 4: Create a Docker Compose File for Kafka**

This file is now much simpler and only defines the services we need from Docker: Zookeeper and Kafka.

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

Add the following content:

| # docker-compose.yaml services:  zookeeper:  image: confluentinc/cp-zookeeper:7.3.2  container\_name: zookeeper  ports: ["2181:2181"]  environment:  ZOOKEEPER\_CLIENT\_PORT: 2181   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\_OFFSETS\_TOPIC\_REPLICATION\_FACTOR: 1 |
| --- |

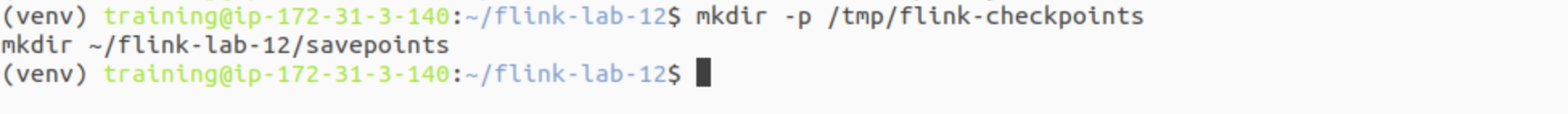
### 

### **Part 3: Configuring the Local Flink Cluster**

**Step 1: Create Directories for State Storage**

We need local directories where Flink can store its checkpoint and savepoint data.

| mkdir -p /tmp/flink-checkpoints mkdir ~/flink-lab-12/savepoints |
| --- |



**Step 2: Configure Flink for Checkpointing**

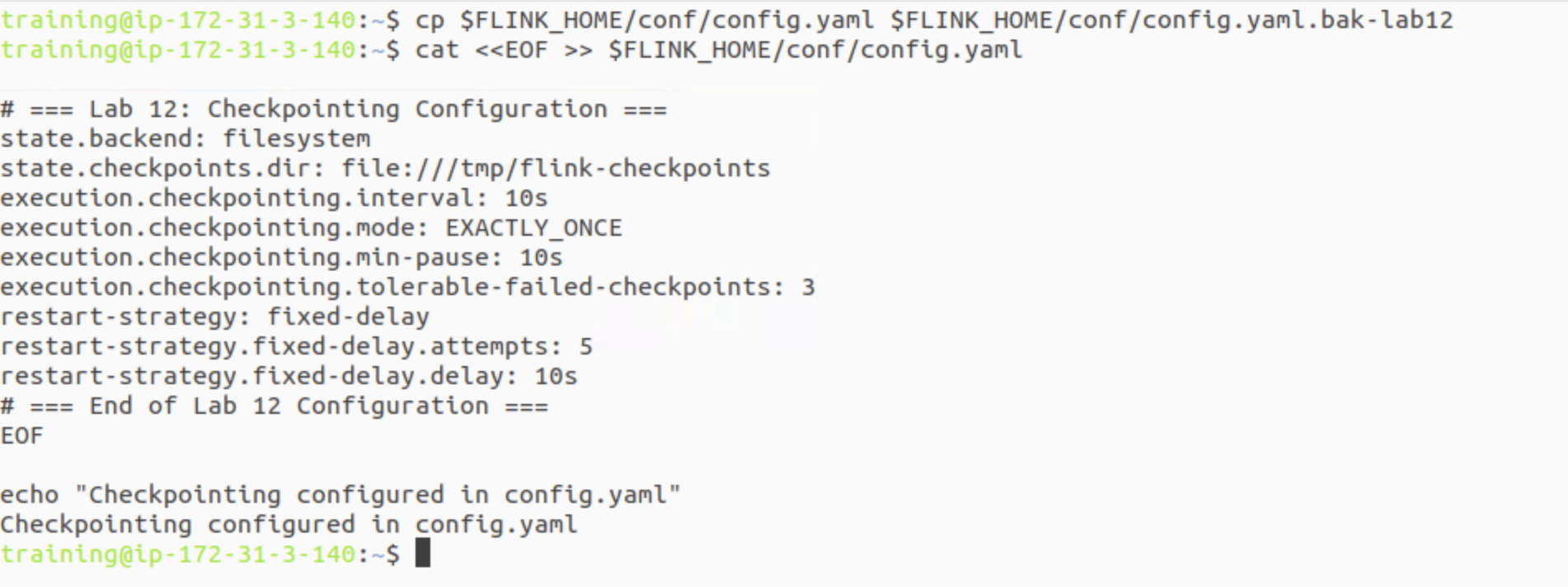
We will add the necessary configuration to your main Flink configuration file. This script appends the settings and creates a backup of the original file.

Create a backup

| cp $FLINK\_HOME/conf/config.yaml $FLINK\_HOME/conf/config.yaml.bak-lab12 |
| --- |

Append the checkpointing configuration

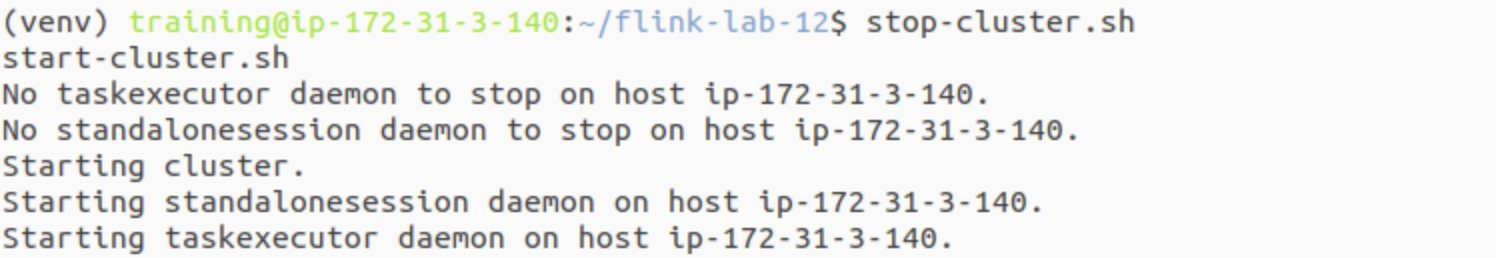
| cat <<EOF >> $FLINK\_HOME/conf/config.yaml  # === Lab 12: Checkpointing Configuration ===  state.backend: filesystem  state.checkpoints.dir: file:///tmp/flink-checkpoints  execution.checkpointing.interval: 10s  execution.checkpointing.mode: EXACTLY\_ONCE  execution.checkpointing.min-pause: 10s  execution.checkpointing.tolerable-failed-checkpoints: 3  restart-strategy: fixed-delay  restart-strategy.fixed-delay.attempts: 5  restart-strategy.fixed-delay.delay: 10s  # === End of Lab 12 Configuration ===  EOF  echo "Checkpointing configured in config.yaml" |
| --- |

****

**Step 3: Restart the Flink Cluster to Apply Changes**

For the new configuration to take effect, you must restart your Flink cluster.

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



### **Part 4: Developing the Stateful Application (V1 & V2)**

Step 1: Create the Kafka Producer

This is the same producer from previous labs.

| code producer.py |
| --- |

Add the following code:

| # producer.py import json, time, random from kafka import KafkaProducer  KAFKA\_TOPIC = 'product\_clicks' KAFKA\_BROKERS = 'localhost:9092'  if \_\_name\_\_ == '\_\_main\_\_':  producer = KafkaProducer(  bootstrap\_servers=KAFKA\_BROKERS,  value\_serializer=lambda v: json.dumps(v).encode('utf-8')  )  product\_ids = [f'prod\_{i}' for i in range(1, 6)]  print("Producing mock click events... Press Ctrl+C to stop.")  try:  while True:  event = {'product\_id': random.choice(product\_ids), 'timestamp': int(time.time() \* 1000)}  producer.send(KAFKA\_TOPIC, value=event)  print(f"Sent: {event}")  time.sleep(random.uniform(0.5, 2.0))  except KeyboardInterrupt:  print("\nStopping producer.")  finally:  producer.flush()  producer.close() |
| --- |

**Step 2: Create the Flink Application (Version 1)**

Note the bootstrap\_servers now points to localhost:9092.

| code stateful\_job\_v1.py |
| --- |

Add the V1 code:

| # stateful\_job\_v1.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 ProductCounter(KeyedProcessFunction):  def \_\_init\_\_(self):  self.count\_state = None   def open(self, runtime\_context: RuntimeContext):  descriptor = ValueStateDescriptor("product\_count", Types.INT())  self.count\_state = runtime\_context.get\_state(descriptor)   def process\_element(self, value, ctx: 'KeyedProcessFunction.Context'):  current\_count = self.count\_state.value() or 0  new\_count = current\_count + 1  self.count\_state.update(new\_count)  product\_id = value.product\_id  yield f"[V1] Product: {product\_id}, Count: {new\_count}"  def main():  env = StreamExecutionEnvironment.get\_execution\_environment()  type\_info = Types.ROW\_NAMED(["product\_id", "timestamp"], [Types.STRING(), Types.LONG()])  deserializer = JsonRowDeserializationSchema.builder().type\_info(type\_info).build()   kafka\_source = KafkaSource.builder() \  .set\_bootstrap\_servers('localhost:9092') \  .set\_topics('product\_clicks') \  .set\_group\_id('flink-recovery-group') \  .set\_starting\_offsets(KafkaOffsetsInitializer.latest()) \  .set\_value\_only\_deserializer(deserializer) \  .build()   stream = env.from\_source(kafka\_source, WatermarkStrategy.no\_watermarks(), "Kafka Source")  stream.key\_by(lambda row: row.product\_id) \  .process(ProductCounter(), output\_type=Types.STRING()) \  .print()  env.execute("Stateful Recovery Job V1")  if \_\_name\_\_ == '\_\_main\_\_':  main() |
| --- |

Step 3: Create the Flink Application (Version 2)

The only changes are the output string and the job name.

| code stateful\_job\_v2.py |
| --- |

Copy the V1 code and modify the yield line and the env.execute line:

| # stateful\_job\_v2.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 ProductCounter(KeyedProcessFunction):  def \_\_init\_\_(self):  self.count\_state = None   def open(self, runtime\_context: RuntimeContext):  descriptor = ValueStateDescriptor("product\_count", Types.INT())  self.count\_state = runtime\_context.get\_state(descriptor)   def process\_element(self, value, ctx: 'KeyedProcessFunction.Context'):  current\_count = self.count\_state.value() or 0  new\_count = current\_count + 1  self.count\_state.update(new\_count)  product\_id = value.product\_id  # --- THE ONLY LOGIC CHANGE IS HERE ---  yield f"[V2 UPGRADED] Product: {product\_id}, Total Clicks: {new\_count}"  def main():  env = StreamExecutionEnvironment.get\_execution\_environment()  type\_info = Types.ROW\_NAMED(["product\_id", "timestamp"], [Types.STRING(), Types.LONG()])  deserializer = JsonRowDeserializationSchema.builder().type\_info(type\_info).build()   kafka\_source = KafkaSource.builder() \  .set\_bootstrap\_servers('localhost:9092') \  .set\_topics('product\_clicks') \  .set\_group\_id('flink-recovery-group') \  .set\_starting\_offsets(KafkaOffsetsInitializer.latest()) \  .set\_value\_only\_deserializer(deserializer) \  .build()   stream = env.from\_source(kafka\_source, WatermarkStrategy.no\_watermarks(), "Kafka Source")  stream.key\_by(lambda row: row.product\_id) \  .process(ProductCounter(), output\_type=Types.STRING()) \  .print()  # --- CHANGE THE JOB NAME ---  env.execute("Stateful Recovery Job V2")  if \_\_name\_\_ == '\_\_main\_\_':  main() |
| --- |

**Part 5: Scenario 1 - Automatic Recovery from Failure**

**Terminal 1: Start Kafka**

| **cd ~/flink-lab-12 docker compose up -d** |
| --- |

**Terminal 2: Start the Producer**

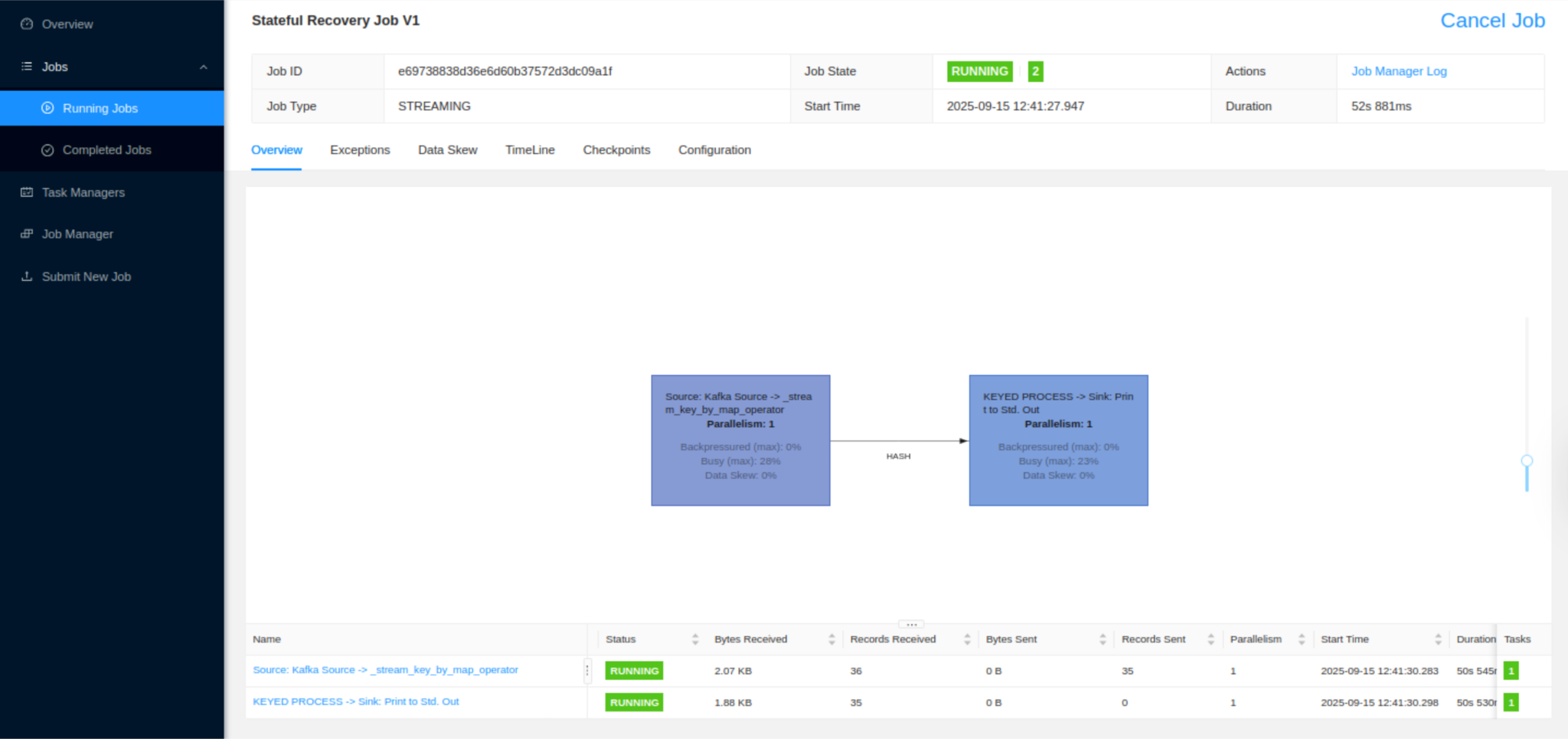
| **cd ~/flink-lab-12 source venv/bin/activate python producer.py** |
| --- |

**Terminal 3: Submit the Flink Job (V1)**

| **cd ~/flink-lab-12 source venv/bin/activate flink run -py stateful\_job\_v1.py** |
| --- |

**Step 1: Verify Initial Run**

Go to the Flink UI (<http://localhost:8081>). The job "Stateful Recovery Job V1" should be running. Check the TaskManager logs to see the counts:



**New Terminal:**

| tail -f $FLINK\_HOME/log/flink-\*-taskexecutor-\*.out | grep "\[V1\]" |
| --- |



In the Flink UI, click on the job and go to the "Checkpoints" tab. You will see checkpoints being completed every 10 seconds.

(Press Ctrl+C to exit the logs when you are finished.)

**Step 2: Simulate a Failure**

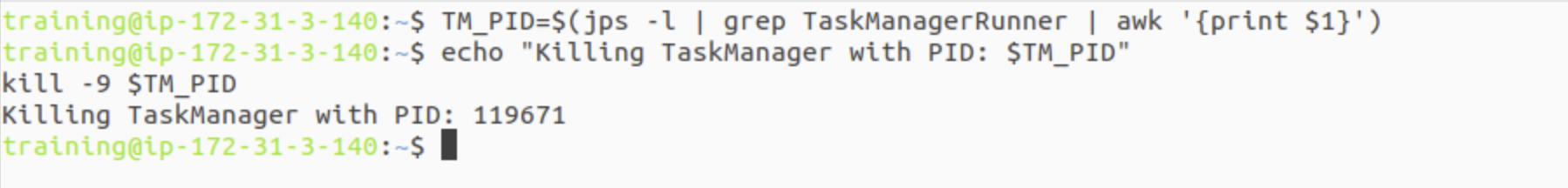
Let the job run for at least 30 seconds. Then, find the Process ID (PID) of the TaskManager and kill it.

Find the PID

| TM\_PID=$(jps -l | grep TaskManagerRunner | awk '{print $1}') |
| --- |

Kill the process

| echo "Killing TaskManager with PID: $TM\_PID" kill -9 $TM\_PID |
| --- |



**Step 3: Observe Automatic Recovery**

Go back to the Flink UI. You will see the job status change to "Failing" and then to "Restarting". Flink's standalone scripts will restart the TaskManager process, and the JobManager will redeploy the job, loading the state from the last successful checkpoint.

Once the job is "Running" again, check the logs. You will see that the counts **did not reset to 1**. They resumed from where they left off, proving recovery worked.

### **Part 6: Scenario 2 - Planned Upgrade with Savepoints**

Step 1: Take a Savepoint

With the V1 job still running, open a new terminal. Find your Job ID and trigger a savepoint, storing it in the directory we created.

# Get the Job ID  
JOB\_ID=$(flink list -r | grep 'RUNNING' | awk '{print $4}')  
  
# Trigger the savepoint  
flink savepoint $JOB\_ID ~/flink-lab-12/savepoints

This command will return a path to the savepoint. **Copy this full path.**

**Step 2: Stop the V1 Job**

flink cancel $JOB\_ID

Step 3: Submit the V2 Job from the Savepoint

In Terminal 3, submit the V2 script, using the -s flag and the path you just copied.

# Make sure to replace <SAVEPOINT\_PATH> with the actual path from Step 1  
SAVEPOINT\_PATH="<PASTE\_THE\_SAVEPOINT\_PATH\_HERE>"  
  
flink run -s $SAVEPOINT\_PATH -py stateful\_job\_v2.py

Step 4: Verify the Upgrade

Check the TaskManager logs again.

tail -f $FLINK\_HOME/log/flink-\*-taskexecutor-\*.out | grep "\[V2 UPGRADED\]"

You will see the output has the new ✨ [V2 UPGRADED] format, and the counts have continued seamlessly from where the V1 job left off.

### **Part 7: Cleanup**

Step 1: Stop All Processes

Stop the producer (Ctrl+C in Terminal 2), stop the Flink job (from the UI or with flink cancel), and shut down Kafka.

# In ~/flink-lab-12  
docker compose down -v

Step 2: Stop the Flink Cluster

stop-cluster.sh

Step 3: Restore Flink Configuration

It's important to remove the custom configuration so it doesn't interfere with other labs.

# Restore the original config file from the backup we made  
mv $FLINK\_HOME/conf/config.yaml.bak-lab12 $FLINK\_HOME/conf/config.yaml  
echo "Restored original config.yaml"

Step 4: Clean Up State Directories

rm -rf /tmp/flink-checkpoints  
rm -rf ~/flink-lab-12/savepoints

### **Part 8: Next Steps**

* **State Backends:** Our configuration used the filesystem state backend. Research and re-configure the lab to use the rocksdb state backend, which is recommended for very large state sizes.
* **High Availability:** The JobManager is currently a single point of failure. Research and implement a Flink High Availability (HA) setup using ZooKeeper.
* **Schema Evolution:** Investigate how Flink savepoints handle changes to the data types within your state (e.g., changing a counter from INT to LONG).