Tailored Training - Kafka Developer & Flink SQL on Confluent Cloud

Exercise Book

Version 7.6.0-v1.0.0

Table of Contents

Copyright & Trademarks

Lab 01 Fundamentals of Apache Kafka

a. Introduction

This document provides Hands-On Exercises for the course **Tailored Training - Kafka Developer & Flink SQL on Confluent Cloud**. You will use a setup that includes a virtual machine (VM) configured as a Docker host to demonstrate the distributed nature of Apache Kafka.

The main Kafka cluster includes the following components, each running in a container:

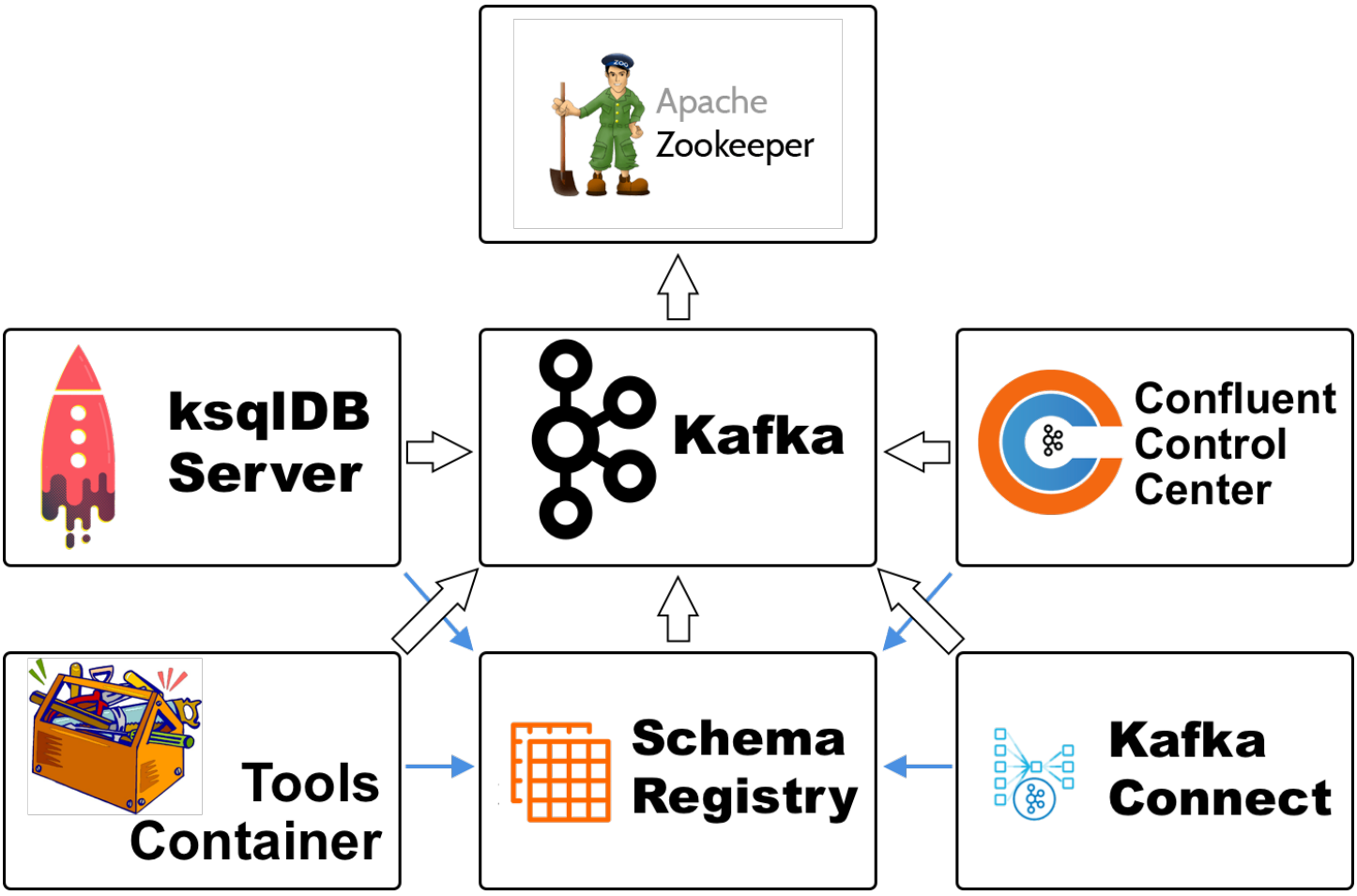


Table 1. Components of the Confluent Platform

Alias	Description
zookeeper	ZooKeeper
kafka	Kafka Broker

Alias	Description
schema-registry	Schema Registry
connect	Kafka Connect
ksqldb-server	ksqlDB Server
control-center	Confluent Control Center
tools	secondary location for tools run against the cluster

As you progress through the exercises you will selectively turn on parts of your cluster as they are needed.

You will use Confluent Control Center to monitor the main Kafka cluster. To achieve this, we are also running the Control Center service which is backed by the same Kafka cluster.

In this course we are using Confluent Platform version 7.0.0 which includes Kafka 3.0.0.

In production, Control Center should be deployed with its own dedicated Kafka cluster, separate from the cluster with production traffic. Using a dedicated metrics cluster is more resilient because it continues to provide system health monitoring even if the production traffic cluster experiences issues.

Alternative Lab Environments

As an alternative you can also download the VM to your laptop and run it in VirtualBox. Make sure you have the newest version of VirtualBox installed. Download the VM from this link:

- https://s3.amazonaws.com/confluent-training-images-us-east-1/training-ubuntu-20-04-jan2022.ova

If you have installed Docker for Desktop on your Mac or Windows 10 Pro machine then you can run the labs there. But please note that your trainer might not be able to troubleshoot any potential problems if you are running the labs locally. If you choose to do this, follow the instructions at → Running Labs in Docker for Desktop.

Command Line Examples

Most exercises contain commands that must be run from the command line. These commands will look like this:

```
$ pwd
/home/training
```

Commands you should type are shown in **bold**; non-bold text is an example of the output produced as a result of the command.

Preparing the Labs

Welcome to your lab environment! You are connected as user **training**, password **training**.

If you haven't already done so, you should open the **Exercise Guide** that is located on the lab virtual machine. To do so, open the **Confluent Training Exercises** folder that is located on the lab virtual machine desktop. Then double-click the shortcut that is in the folder to open the **Exercise Guide**.

Copy and paste works best if you copy from the Exercise Guide on your lab virtual machine.

- Standard Ubuntu keyboard shortcuts will work: `Ctrl+C` → Copy, `Ctrl+V` → Paste
- In a Terminal window: `Ctrl+Shift+C` → Copy, `Ctrl+Shift+V` → Paste.

If you find these keyboard shortcuts are not working you can use the right-click context menu for copy and paste.

1. Open a terminal window
2. Clone the source code repository to the folder `confluent-dev` in your **home** directory:

```
$ cd ~
$ git clone --depth 1 --branch 7.0.0-v1.0.5 \
    https://github.com/confluentinc/training-developer-src.git \
    confluent-dev
```

If you chose to select another folder for the labs then note that many of our samples assume that the lab folder is `~/confluent-dev`. You will have to adjust all those command to fit your specific environment.

3. Navigate to the `confluent-dev` folder:

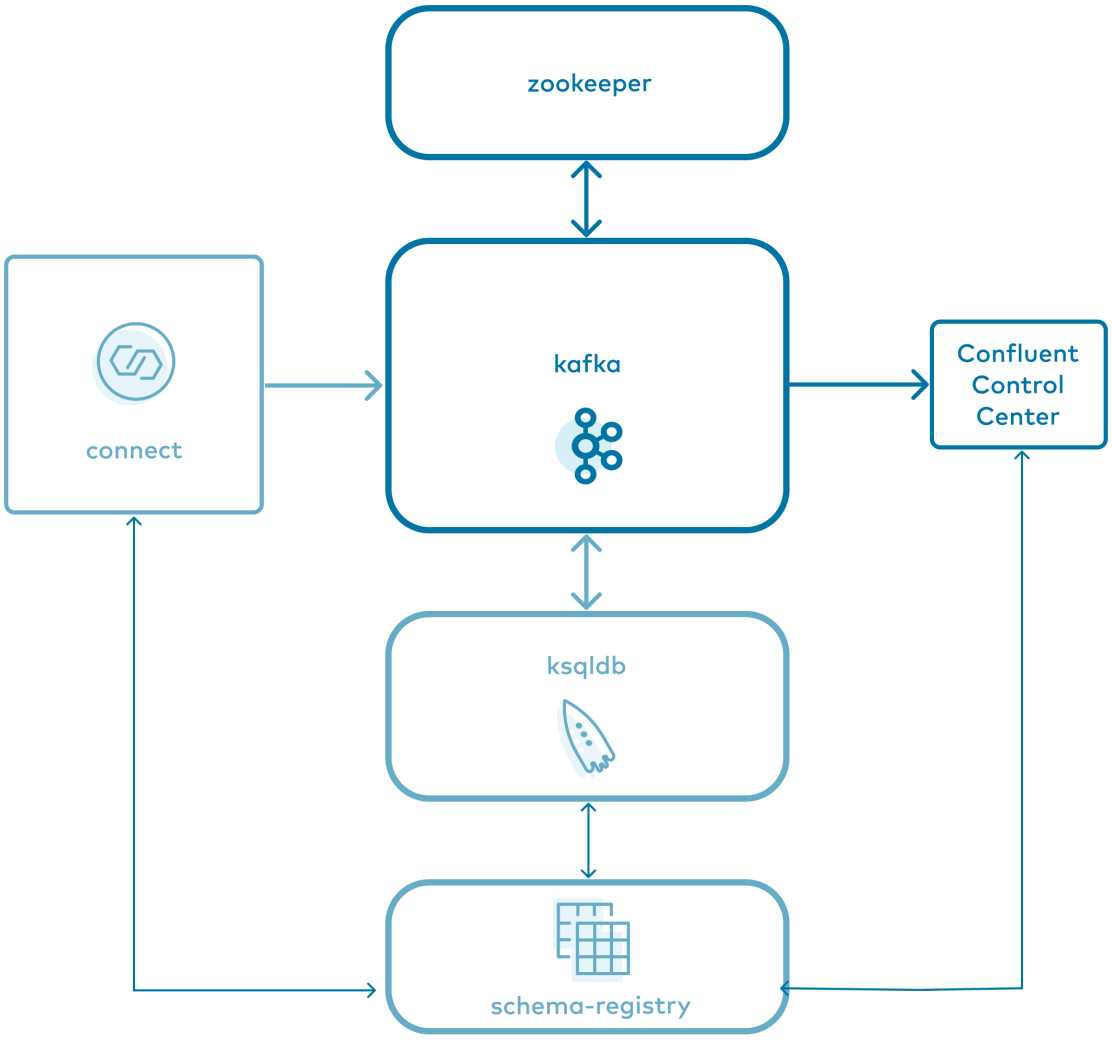
```
$ cd ~/confluent-dev
```

4. Start the Kafka cluster:

```
$ docker-compose up -d zookeeper kafka control-center
```

You should see something similar to this:

```
[+] Running 4/4
 ✔ Network confluent-dev_default  Creat...
0.1s
 ✔ Container zookeeper            Started
3.5s
 ✔ Container kafka                Started
3.5s
 ✔ Container control-center       Started
3.5s
```



In the first steps of each exercise, you launch the containers needed for the exercise with `docker-compose up`.

If at any time you want to get your environment back to a clean state use `docker-compose down` to end all of your containers. Then return to your last `docker-compose up` to get back to the beginning of an exercise.

Exercises do not need to be completed in order. You can start from the beginning of any exercise at any time.

If you want to completely clear out your docker environment use the script on the VM at `~/docker-nuke.sh`. The nuke script will forcefully end all of your running docker containers.

5. Monitor the cluster with:

```
$ docker-compose ps

NAME                COMMAND                  SERVICE
STATUS              PORTS
control-center      "/etc/confluent/dock…"   control-center
running             0.0.0.0:9021->9021/tcp, :::9021->9021/tcp
kafka               "/etc/confluent/dock…"   kafka
running             0.0.0.0:9092->9092/tcp, :::9092->9092/tcp
zookeeper           "/etc/confluent/dock…"   zookeeper
running             0.0.0.0:2181->2181/tcp, :::2181->2181/tcp
```

All services should have **State** equal to **Up**.

6. You can also observe the stats of Docker on your VM:

```
$ docker stats

CONTAINER ID   NAME              CPU %     MEM USAGE / LIMIT     MEM %
NET I/O           BLOCK I/O         PIDS
04d013943313   control-center    7.39%     599.6MiB / 15.18GiB   3.86%
1.16MB / 497kB    23.5MB / 5.36MB   91
f0da4b92f565   kafka             12.59%    623.9MiB / 15.18GiB   4.01%
1.39MB / 1.86MB   528kB / 3.08MB    112
fae4862a650a   zookeeper         0.07%     102.9MiB / 15.18GiB   0.66%
712kB / 886kB     3.18MB / 3.05MB   47
```

Press **Ctrl+C** to exit the Docker statistics.

Testing the Installation

1. Use the **zookeeper-shell** command to verify that all Brokers have registered with ZooKeeper. You should see a single Broker listed as **[101]** in the penultimate line of the output.

```
$ zookeeper-shell zookeeper:2181 ls /brokers/ids
Connecting to zookeeper:2181

WATCHER::

WatchedEvent state:SyncConnected type:None path:null
[101]
[2022-12-13 12:14:08,310] ERROR Exiting JVM with code 0
(org.apache.zookeeper.util.ServiceUtils)
```

Ignore the ERROR in the last line. It is a "good" error indicating the command is exiting the Zookeeper shell.

OPTIONAL: Analyzing the Docker Compose File

1. Open the file `docker-compose.yml` in your editor and:
   a. locate the various services that are listed in the table earlier in this section
   b. note that the container name (e.g. `zookeeper` or `kafka`) are used to resolve a particular service
   c. note how the broker (kafka)
      i. gets a unique ID assigned via environment variable `KAFKA_BROKER_ID`
      ii. defines where to find the ZooKeeper instance

```
KAFKA_ZOOKEEPER_CONNECT: zookeeper:2181
```

      iii. sets the replication factor for the offsets topic to 1:

```
KAFKA_OFFSETS_TOPIC_REPLICATION_FACTOR: 1
```

      iv. configures the broker to send metrics to Confluent Control Center:

```
KAFKA_METRIC_REPORTERS:
"io.confluent.metrics.reporter.ConfluentMetricsReporter"
CONFLUENT_METRICS_REPORTER_BOOTSTRAP_SERVERS: "kafka:9092"
```

   d. note how various services use the environment variable `..._BOOTSTRAP_SERVERS` to define the list of Kafka brokers that serve as bootstrap servers (in our case it's only one instance):

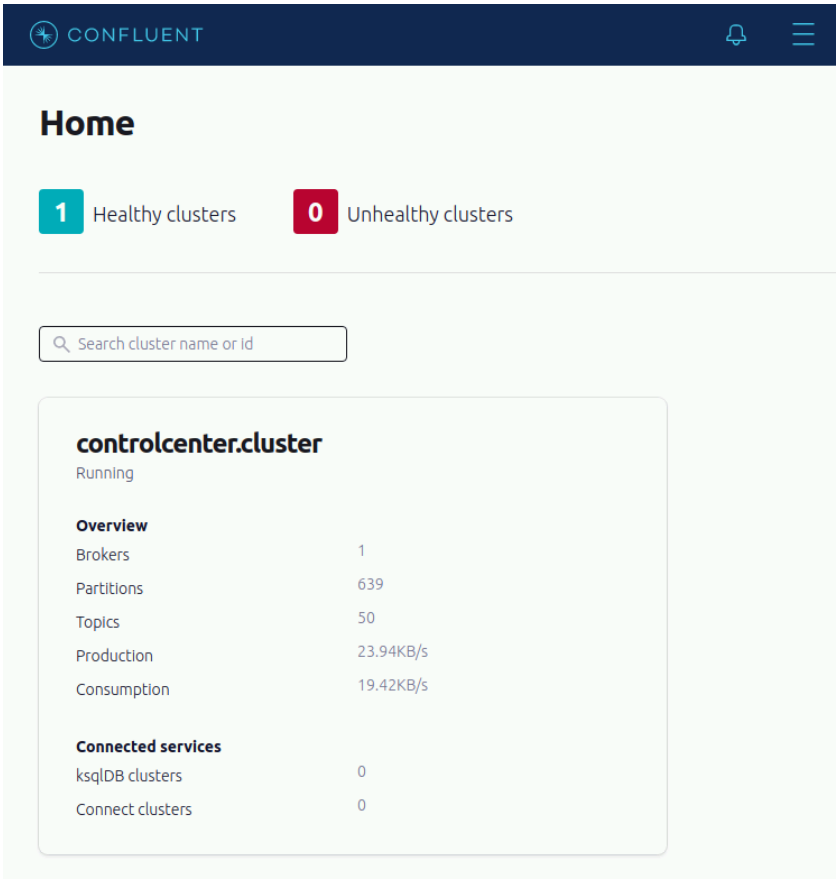
```
..._BOOTSTRAP_SERVERS: kafka:9092
```

   e. note how e.g. the `connect` service and the `ksqldb-server` service define producer and consumer interceptors that produce data which can be monitored in Confluent Control Center:

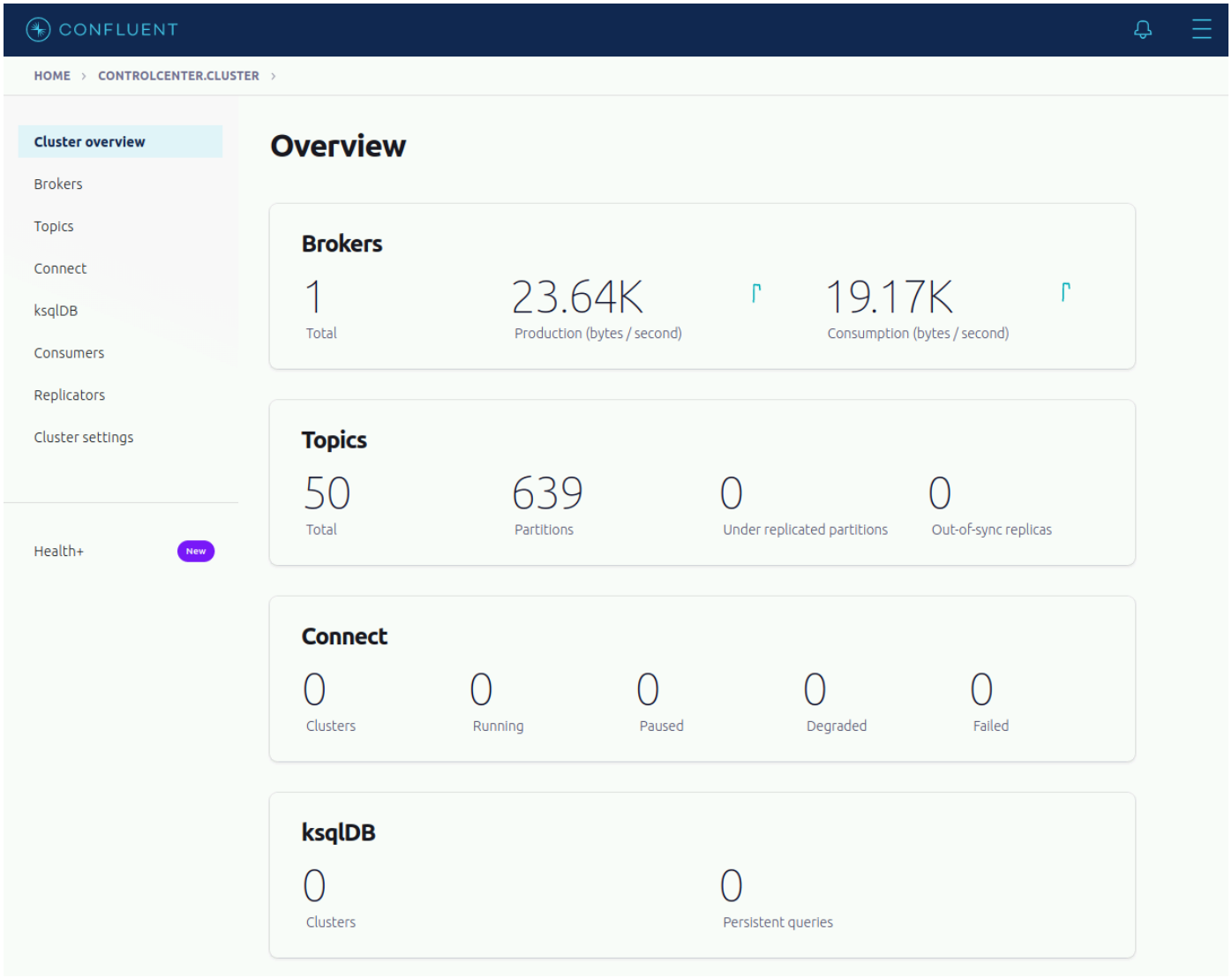
```
io.confluent.monitoring.clients.interceptor.MonitoringProducerInte
rceptor
io.confluent.monitoring.clients.interceptor.MonitoringConsumerInte
rceptor
```

Using Confluent Control Center

1. On your host machine, open a new browser tab in Google Chrome.
2. Navigate to Control Center at the URL http://localhost:9021:



3. Select the cluster **controlcenter.cluster** and you will see this:



We have a single broker in our cluster. Also note the other important metrics of our Kafka cluster on this view.

4. Optional: Explore the other tabs of Confluent Control Center, such as **Topics** or **Cluster settings**.

b. Using Kafka's Command-Line Tools

In this Hands-On Exercise you will start to become familiar with some of Kafka's command-line tools. Specifically you will:

- Use a tool to **create** a topic
- Use a console program to **produce** a message

- Use a console program to **consume** a message
- Use a tool to explore data stored in ZooKeeper

Prerequisites

1. Navigate to the `confluent-dev` folder:

```
$ cd ~/confluent-dev
```

2. Run the Kafka cluster, including Confluent Control Center:

```
$ docker-compose up -d zookeeper kafka control-center
```

   If your containers are running from the previous exercise this command will simply tell you each container is running.

Console Producing and Consuming

Kafka has built-in command line utilities to produce messages to a Topic and read messages from a Topic. These are extremely useful to verify that Kafka is working correctly, and for testing and debugging.

1. Before we can start writing data to a topic in Kafka, we need to first create that topic using a tool called `kafka-topics`. From within the terminal window run the command:

```
$ kafka-topics
```

   This will bring up a list of parameters that the `kafka-topics` program can receive. Take a moment to look through the options.

2. Now execute the following command to create the topic `testing`:

```
$ kafka-topics --bootstrap-server kafka:9092 \
    --create \
    --partitions 1 \
    --replication-factor 1 \
    --topic testing
```

We create the topic with a single partition and `replication-factor` of one.

We could have configured Kafka to allow **auto-creation** of topics. In this case we would not have had to do the above step and the topic would automatically be created when the first record is written to it. But this behavior is **strongly discouraged** in production. Always create your topics explicitly!

3. Now let's move on to start writing data into the topic just created. From within the terminal window run the command:

```
$ kafka-console-producer
```

This will bring up a list of parameters that the `kafka-console-producer` program can receive. Take a moment to look through the options. We will discuss many of their meanings later in the course.

4. Run `kafka-console-producer` again with the required arguments:

```
$ kafka-console-producer --bootstrap-server kafka:9092 --topic
testing
```

The tool prompts you with a `>`.

5. At this prompt type:

```
> some data
```

And click **Enter**.

6. Now type:

```
> more data
```

And click **Enter**.

7. Type:

```
> final data
```

   And click **Enter**.

8. Now we will use a Consumer to retrieve the data that was produced. Open a new terminal window and run the command:

```
$ kafka-console-consumer
```

   This will bring up a list of parameters that the `kafka-console-consumer` can receive. Take a moment to look through the options.

9. Run `kafka-console-consumer` again with the following arguments:

```
$ kafka-console-consumer \
    --bootstrap-server kafka:9092 \
    --from-beginning \
    --topic testing
```

   After a short moment you should see all the messages that you produced using `kafka-console-producer` earlier:

```
some data
more data
final data
```

10. Press `Ctrl+D` to exit the `kafka-console-producer` program.

11. Press `Ctrl+C` to exit `kafka-console-consumer`.

OPTIONAL: Working with record keys

By default, `kafka-console-producer` and `kafka-console-consumer` assume null keys. They can also be run with appropriate arguments to write and read keys as well as values.

1. Re-run the Producer with additional arguments to write (key,value) pairs to the Topic:

```
$ kafka-console-producer \
    --bootstrap-server kafka:9092 \
    --topic testing \
    --property parse.key=true \
    --property key.separator=,
```

2. Enter a few values such as:

```
> 1,my first record
> 2,another record
> 3,Kafka is cool
```

3. Press `Ctrl+D` to exit the producer.

4. Now run the **Consumer** with additional arguments to print the key as well as the value:

```
$ kafka-console-consumer \
    --bootstrap-server kafka:9092 \
    --from-beginning \
    --topic testing \
    --property print.key=true

null    some data
null    more data
null    final data
1       my first record
2       another record
3       Kafka is cool
```

   Note the `NULL` values for the first 3 records that we entered earlier...

5. Press `Ctrl+C` to exit the consumer.

The ZooKeeper Shell

1. Kafka's data in ZooKeeper can be accessed using the `zookeeper-shell` command:

```
$ zookeeper-shell zookeeper
Connecting to zookeeper
Welcome to ZooKeeper!
JLine support is disabled

WATCHER::

WatchedEvent state:SyncConnected type:None path:null
```

2. From within the **zookeeper-shell** application, type **ls /** to view the directory structure in ZooKeeper. Note the **/** is required.

```
ls /
[admin, brokers, cluster, config, consumers, controller,
controller_epoch, isr_change_notification, latest_producer_id_block,
log_dir_event_notification, zookeeper]
```

3. Type **ls /brokers** to see this next level of the directory structure.

```
ls /brokers
[ids, seqid, topics]
```

4. Type **ls /brokers/ids** to see the broker ids for the Kafka cluster.

```
ls /brokers/ids
[101]
```

Note the output **[101]**, indicating that we have a single broker with ID **101** in our cluster.

5. Type **get /brokers/ids/101** to see the metadata for broker 101.

```
get /brokers/ids/101
{"features":{},"listener_security_protocol_map":{"PLAINTEXT":"PLAINTE
XT"},"endpoints":["PLAINTEXT://kafka:9092"],"jmx_port":-
1,"port":9092,"host":"kafka","version":5,"tags":{},"timestamp":"16709
33479389"}
```

6. Type **get /brokers/topics/testing/partitions/0/state** to see the metadata for partition 0 of topic **testing**.

```
get /brokers/topics/testing/partitions/0/state
{"controller_epoch":1,"leader":101,"version":1,"confluent_is_unclean_
leader":false,"leader_epoch":0,"isr":[101]}
```

Note: During client startup, it requests cluster metadata from a broker in the `bootstrap.servers` list. The output of the two previous commands reflects a bit of this cluster metadata included in the broker response. We will cover this metadata request in more detail later in this course.

7. Press `Ctrl+D` to exit the ZooKeeper shell.

Conclusion

In this lab you have used Kafka command line tools to create a topic, write and read from this topic. Finally you have used the ZooKeeper shell tool to access data stored within ZooKeeper.

STOP HERE. THIS IS THE END OF THE EXERCISE.

Lab 02 Producing Messages to Kafka

a. Kafka Producer (Java, C#, Python)

The goal of this lab is to create a simple producer. The producer application reads a file of latitude and longitude values and writes to the topic **`driver-positions`**. See an example file at **`~/confluent-dev/challenge/java-producer/drivers/driver-1.csv`**. For each entry in the file the application writes a Kafka record. When the application reaches the end of the file it loops back to the top. The record is created with the driver id as the key, and the comma separated latitude and longitude as the value. For example:

Key	Value
driver-1	47.5952,-122.3316

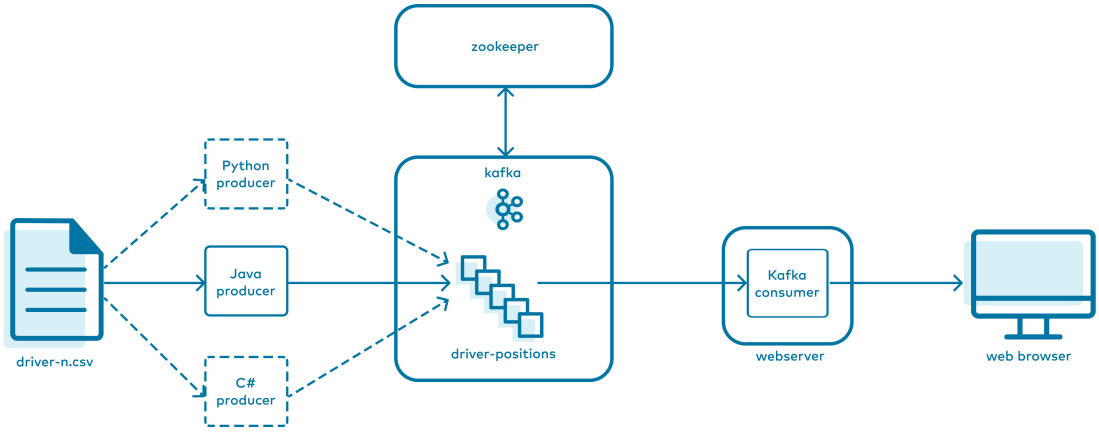
Prerequisites

1. Use the command in the table below to navigate to the project folder for your language. Click the associated hyperlink to open the API reference:

Language	Command	API Reference
Java	**`cd ~/confluent-dev/challenge/java-producer`**	Class KafkaProducer<K,V>
C#	**`cd ~/confluent-dev/challenge/dotnet-producer`**	Interface IProducer<TKey, TValue>
Python	**`cd ~/confluent-dev/challenge/python-producer`**	class confluent_kafka.Producer

2. Run the Kafka cluster:

```
$ docker-compose up -d zookeeper kafka control-center create-topics webserver
```



The **`create-topics`** container creates the topics for all of the upcoming exercises and then exits. The **`webserver`** container is running a web application that is consuming from the **`driver-positions`** and displaying the position of each driver.

3. View the application at http://localhost:3001. You will see a driver appear on the map when you complete the code challenges in this exercise.

4. Run the **`kafka-topics`** command to see the new topics. All of the new topics are prefixed with **`driver`**:

```
$ kafka-topics --bootstrap-server kafka:9092 --describe | grep
'driver'
Topic: driver-positions TopicId: dWTPP4A1QB-0r3yt4jP2pg
PartitionCount: 3   ReplicationFactor: 1    Configs:
segment.bytes=536870912,retention.bytes=536870912
    Topic: driver-positions Partition: 0    Leader: 101 Replicas: 101
Isr: 101    Offline:
    Topic: driver-positions Partition: 1    Leader: 101 Replicas: 101
Isr: 101    Offline:
    Topic: driver-positions Partition: 2    Leader: 101 Replicas: 101
Isr: 101    Offline:
...
```

If any replicas were listed as offline, this would be an indication that the corresponding broker is also offline.

5. If you are completing the C# or Python exercise, install the dependencies.

   a. For C#:

   First, install dotnet running this command. You'll be prompted to enter the password: **training**

```
$ ~/confluent-dev/dotnet-install.sh
```

Then, run:

```
$ dotnet restore
```

b. For Python:

```
$ pip3 install -r requirements.txt
```

6. Open the project in Visual Studio Code. As always, make sure you open VS Code in the correct project folder as specified in step 1.

```
$ code .
```

Writing the Producer

1. Open the implementation file for your language of choice
   - Java `src/main/java/clients/Producer.java`
   - C#: `Program.cs`
   - Python: `main.py`.
2. Locate the `TODO` comments in your implementation file. Use the API reference for your language to attempt each challenge. Solutions are provided at the end of this lab and in the `~/confluent-dev/solution` folder.
3. At any time run the application by selecting the menu **Run** → **Start Debugging** in VS Code. As you complete the challenges try to produce a similar output from your application:

```
Starting Java producer.
Sent Key:driver-1 Value:47.618579,-122.355081
Sent Key:driver-1 Value:47.618577152452055,-122.35520620652974
Sent Key:driver-1 Value:47.61857902704408,-122.35507321130525
Sent Key:driver-1 Value:47.618579488930855,-122.35494018791431
Sent Key:driver-1 Value:47.61857995081763,-122.35480716452278
...
```

Are you having issues with VS code debugging?

Here are some common issues you can check for:

- Did you launch `code` from the correct directory? Ensure you've followed the `cd` command above to change to the project folder.
- Do you miss an earlier step? Quit VS Code, run the missing step, and relaunch VS Code.
- Java users: if VS Code is still having an issue after relaunching try cleaning your workspace. Click the cog wheel icon at the bottom left of VS Code, click **Command Palette**, search for and select **Java: Clean the Java language server workspace**, click **Restart and delete**.

4. When you have the application producing data, leave the application running, and return to the web application at http://localhost:3001.

If you are interested in the inner workings of the web application the source code is available at `~/confluent-dev/webserver`. The web application is a simple Node.js Express web application that contains a Kafka consumer reading from a topic and delivering the records via a Socket.IO websocket to the front end.

5. Use the `kafka-console-consumer` tool to view the data on the `driver-positions` topic:

```
$ kafka-console-consumer --bootstrap-server kafka:9092 \
    --topic driver-positions \
    --property print.key=true \
    --from-beginning
```

Exit the consumer with `Ctrl+C`.

6. When you have completed the challenges, stop the debugger in VS Code.

Extra Challenges and Questions

1. The producer application takes the driver ID from an environment variable `DRIVER_ID`. From new terminal windows run the producer with several different driver IDs. Observe the web application at http://localhost:3001. When you have finished exit your producers with `Ctrl+C`.

   a. Java

   ```
   $ cd ~/confluent-dev/solution/java-producer && \
      DRIVER_ID=driver-3 ./gradlew run --console plain
   ```

   b. C#

   ```
   $ cd ~/confluent-dev/solution/dotnet-producer && \
      DRIVER_ID=driver-3 dotnet run
   ```

   c. Python

   ```
   $ cd ~/confluent-dev/solution/python-producer && \
      DRIVER_ID=driver-3 python3 main.py
   ```

2. Experiment with producer settings of `batch.size` (default: 16384) and `linger.ms` (default: 0 ms). How would you expect the producer to behave with the settings below? Observe the web application at http://localhost:3001.

   a. Java

   ```
   settings.put(ProducerConfig.BATCH_SIZE_CONFIG, 16384);
   settings.put(ProducerConfig.LINGER_MS_CONFIG, 5000);
   ```

   b. C#

   ```
   BatchNumMessages = 16384,
   LingerMs = 5000
   ```

   c. Python

```
'batch.num.messages': 16384,
'linger.ms': 5000
```

Java Solution

solution/java-producer/src/main/java/clients/Producer.java

```
// TODO: configure the location of the bootstrap server
settings.put(ProducerConfig.BOOTSTRAP_SERVERS_CONFIG, "kafka:9092");
```

```
// TODO: populate the message object
final ProducerRecord<String, String> record = new ProducerRecord
<>(KAFKA_TOPIC, key, value);
```

```
// TODO: write the lat/long position to a Kafka topic
// TODO: print the key and value in the callback lambda
producer.send(record, (md, e) -> {
  System.out.printf("Sent Key:%s Value:%s\n", key, value);
});
```

C# Solution

solution/dotnet-producer/Program.cs

```
// TODO: configure the location of the bootstrap server
BootstrapServers = "kafka:9092",
```

```
// TODO: populate the message object
var message = new Message<string, string> { Key = driverId, Value = line
};
```

```
// TODO: write the lat/long position to a Kafka topic
// TODO: configure handler as a callback to print the key and value
producer.Produce(KafkaTopic, message, handler);
```

Python Solution

solution/python-producer/main.py

```
#TODO: configure the location of the bootstrap server
'bootstrap.servers': 'kafka:9092',
```

```
#TODO: write the lat/long position to a Kafka topic
#TODO: configure delivery_report as a callback to print the key and
value
producer.produce(
    KAFKA_TOPIC,
    key=DRIVER_ID,
    value=line,
    on_delivery=delivery_report)
```

Extra Challenges and Questions Solutions

1. If you followed the steps successfully you will see multiple cars driving on the map.
2. With the supplied settings for **`batch.size`** and **`linger.ms`** the consumer is batching up records. The amount of batched data never exceeds the **`batch.size`**, so the batch is sent to the broker when the **`linger.ms`** of seconds is met. From the producer documentation:

> linger.ms: ...This setting gives the upper bound on the delay for batching: once we get batch.size worth of records for a partition it will be sent immediately regardless of this setting, however if we have fewer than this many bytes accumulated for this partition we will 'linger' for the specified time waiting for more records to show up...

STOP HERE. THIS IS THE END OF THE EXERCISE.

Lab 04 Consuming Messages from Kafka

a. Kafka Consumer (Java, C#, Python)

The goal of this lab is to create a simple Kafka consumer, that consumes records from the `driver-positions` topic.

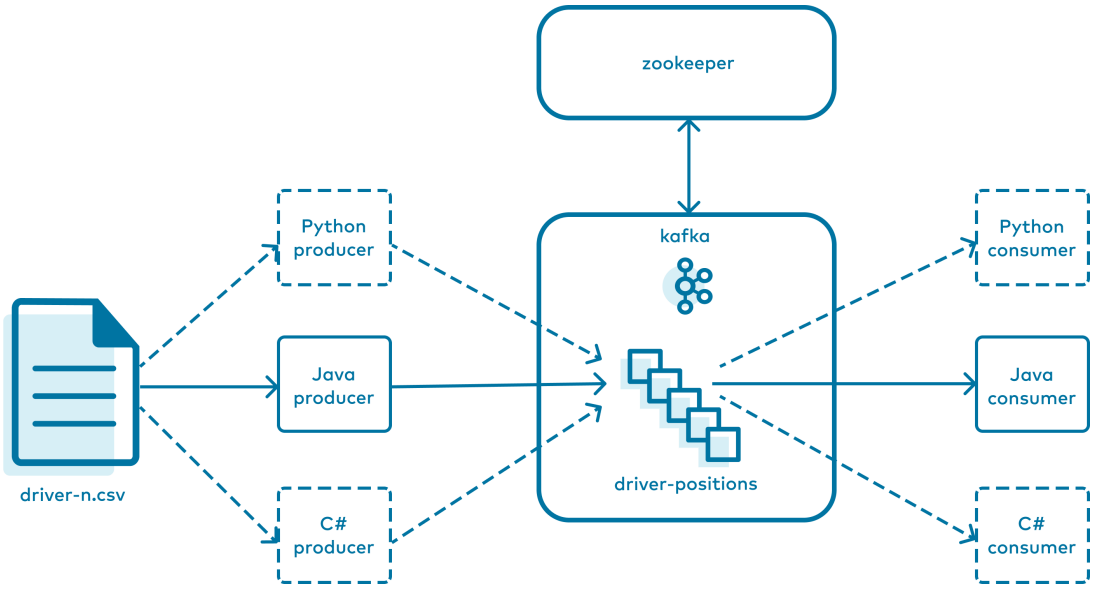
Prerequisites

1. Use the command in the table below to navigate to the project folder for your language. Click the associated hyperlink to open the API reference:

Language	Command	API Reference
Java	`cd ~/confluent-dev/challenge/java-consumer`	Class KafkaConsumer<K,V>
C#	`cd ~/confluent-dev/challenge/dotnet-consumer`	Interface IConsumer<TKey, TValue>
Python	`cd ~/confluent-dev/challenge/python-consumer`	class confluent_kafka.Consumer

2. Run the Kafka cluster:

```
$ docker-compose up -d zookeeper kafka control-center create-topics webserver
```



3. If you are completing the C# or Python exercise, install the dependencies.

   a. For C#:

   ```
   $ dotnet restore
   ```

   b. For Python:

   ```
   $ pip3 install -r requirements.txt
   ```

4. Open the project in Visual Studio Code:

   ```
   $ code .
   ```

Writing the Consumer

1. Run the producer solution from the previous exercise in a terminal window. This will give you live data in the `driver-positions` topic. From a terminal window run:

   ```
   $ cd ~/confluent-dev/solution/java-producer && \
     ./gradlew run --console plain
   ```

2. Open the implementation file for your language of choice

   - Java: `src/main/java/clients/Consumer.java`

- C#: `Program.cs`
- Python: `main.py`.

3. Locate the `TODO` comments in your implementation file. Use the API reference for your language to attempt each challenge. Solutions are provided at the end of this lab and in the `~/confluent-dev/solution` folder.

4. At any time run the application by selecting the menu **Run** → **Start Debugging** in VS Code. As you complete the challenges try to produce a similar output from your application:

```
Starting Java Consumer.
Key:driver-1 Value:47.618579,-122.355081 [partition 1]
Key:driver-1 Value:47.618577152452055,-122.35520620652974 [partition
1]
Key:driver-1 Value:47.61857902704408,-122.35507321130525 [partition
1]
Key:driver-1 Value:47.618579488930855,-122.35494018791431 [partition
1]
Key:driver-1 Value:47.61857995081763,-122.35480716452278 [partition
1]
...
```

5. Leave your consumer running. In a terminal window run this command to inspect the status of your consumer group:

```
$ kafka-consumer-groups \
    --bootstrap-server kafka:9092 \
    --describe \
    --group java-consumer \
    --group csharp-consumer \
    --group python-consumer

GROUP           TOPIC            PARTITION  CURRENT-OFFSET  LOG-END-
OFFSET  LAG ...
csharp-consumer driver-positions 0          -               0
-
csharp-consumer driver-positions 1          2148            2153
5
csharp-consumer driver-positions 2          -               0
-
```

You can see some interesting metrics for your topic consumption. `CURRENT-OFFSET` is the last *committed* offset from your consumers. `LOG-END-OFFSET` is the last offset in each partition. `LAG` is *how far behind* the consumption is, or in other words `LOG-END-OFFSET -`

`CURRENT-OFFSET`. These metrics are very useful when checking if your consumption is keeping up with production.

6. When you have completed the challenges, stop the debugger in VS Code.
7. Return to the terminal window running the producer solution. Press `Ctrl+C` to exit the producer.

Extra Challenges and Questions

1. End your processing, and launch the consumer again. You'll see that the second time you run the application processing begins from a non-zero offset. Does `auto.offset.reset` apply the second time the application is run?
2. How do consumers know where to begin their processing?
3. Can you think of a way to make your next run of the application begin at the offset at the start of each partition?
4. Experiment with consumer settings of `fetch.max.wait.ms` (default: 500ms) and `fetch.min.bytes` (default: 1 byte). How would you expect the consumer to behave with the settings below?

   a. Java

   ```
   settings.put(ConsumerConfig.FETCH_MAX_WAIT_MS_CONFIG, "5000");
   settings.put(ConsumerConfig.FETCH_MIN_BYTES_CONFIG, "5000000");
   ```

   b. C#

   ```
   FetchWaitMaxMs = 5000,
   FetchMinBytes = 5000000,
   ```

   c. Python

   ```
   "fetch.wait.max.ms": "5000",
   "fetch.min.bytes": "5000000"
   ```

Java Solution

solution/java-consumer/src/main/java/clients/Consumer.java

```
// TODO: Poll for available records
final ConsumerRecords<String, String> records = consumer.poll(Duration
.ofMillis(100));
```

```
// TODO: print the contents of the record
System.out.printf("Key:%s Value:%s [partition %s]\n",
    record.key(), record.value(), record.partition());
```

C# Solution

solution/dotnet-consumer/Program.cs

```
// TODO: Consume available records
var cr = consumer.Consume(cts.Token);
```

```
// TODO: print the contents of the record
Console.WriteLine($"Key:{cr.Message.Key} Value:{cr.Message.Value}
[partition {cr.Partition.Value}]");
```

Python Solution

solution/python-consumer/main.py

```
#TODO: Poll for available records
msg = consumer.poll(1.0)
```

```
#TODO: print the contents of the record
print("Key:{} Value:{} [partition {}]".format(
    msg.key().decode('utf-8'),
    msg.value().decode('utf-8'),
    msg.partition()
))
```

Extra Challenges and Questions Solutions

1. `auto.offset.reset` would not be used on the second launch of your consumer. `auto.offset.reset` is used when there is no committed position (e.g. the group is first

initialized) or when an offset is out of range.

2. An instance in a consumer group sends its offset commits and fetches to a group coordinator broker. The group coordinators read from and write to special compacted Kafka topic named *__consumer_offsets*.

   Curious about the *__consumer_offsets* topic? You can consume message on this topic while your consumer runs with the command below:

```
$ kafka-console-consumer --bootstrap-server kafka:9092 \
--topic __consumer_offsets \
--formatter
"kafka.coordinator.group.GroupMetadataManager\$OffsetsMessageFormatte
r" \
| grep 'driver-positions'
```

3. You could begin consumption at the start of each partition simply by updating your consumer to use a new `GROUP_ID`. Also, the utility `kafka-consumer-groups` has a parameter `--to-earliest` which will set offsets to earliest offset. In the next exercise we will see how to programmatically *seek* to an offset for consumption.

4. With the supplied settings for `fetch.max.wait.ms` and `fetch.min.bytes` we see results roughly every 5 seconds. From the consumer documentation:

   > fetch.max.wait.ms: The maximum amount of time the server will block before answering the fetch request if there isn't sufficient data to immediately satisfy the requirement given by fetch.min.bytes.

STOP HERE. THIS IS THE END OF THE EXERCISE.

Lab 07 Schema Management in Apache Kafka

a. Schema Registry, Avro Producer and Consumer (Java, C#, Python)

The goal of this lab is to update our simple producer to write to an AVRO serialized topic `driver-positions-avro`. The code is very similar to your previous producer lab. The code will now communicate with Schema Registry to store and retrieve schemas, and will serialize the structured data in AVRO format.

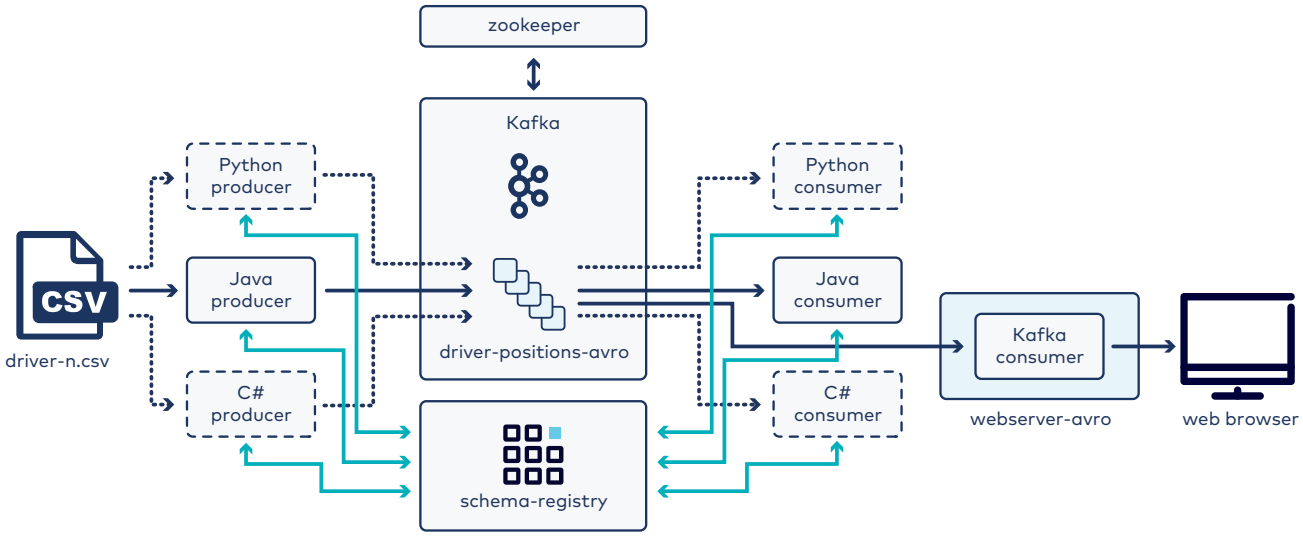
Prerequisites

1. Use the command in the table below to navigate to the project folder for your language:

Language	Command
Java	`cd ~/confluent-dev/challenge/java-producer-avro`
C#	`cd ~/confluent-dev/challenge/dotnet-producer-avro`
Python	`cd ~/confluent-dev/challenge/python-producer-avro`

2. Run the Kafka cluster:

```
$ docker-compose up -d zookeeper kafka control-center create-topics \
     schema-registry webserver-avro
```



You have some new containers you haven't seen before, Schema Registry and an updated version of the web application that can deserialize AVRO serialized data.

Writing the Avro Producer

1. If you are completing the **Java** exercise:
   a. Inspect the schema file at `src/main/avro/position_value.avsc`.
   b. The supplied `build.gradle` file contains Avro plugin `com.commercehub.gradle.plugin.avro` which includes a task `generateAvroJava` to generate POJOs (Plain Old Java Objects/classes) from any Avro schemas in the project.
   c. Use `gradle` to generate the AVRO class:

```
$ ./gradlew build
```

2. If you are completing the **C#** exercise:
   a. Inspect the schema file at `position_value.avsc`.
   b. Install the `avrogen` tool:

```
$ dotnet tool install -g Confluent.Apache.Avro.AvroGen
```

You may need to restart the VM to use `avrogen` in the next step. After restarting, you'll need to run again:

```
$ docker-compose up -d zookeeper kafka control-center
create-topics schema-registry webserver-avro
```

c. Use the `avrogen` to generate the AVRO class:

```
$ avrogen -s position_value.avsc .
```

d. Restore dependencies for your project:

```
$ dotnet restore
```

3. If you are completing the **Python** exercise:

   a. Inspect the schema file at `position_value.avsc`.

   b. Install the dependencies:

```
$ pip3 install -r requirements.txt
```

4. Open the project in Visual Studio Code:

```
$ code .
```

5. Open the implementation file for your language of choice. Can you determine what has changed from the previous producer exercise?

   - Java `src/main/java/clients/Producer.java`
   - C#: `Program.cs`
   - Python: `main.py`

6. Run the application by selecting the menu **Run** → **Start Debugging** in VS Code. You will see your application output:

```
Starting Java Avro producer.
...
Sent Key:driver-1 Latitude:47.618579 Longitude:-122.355081
Sent Key:driver-1 Latitude:47.618577152452055 Longitude:-
122.35520620652974
Sent Key:driver-1 Latitude:47.61857902704408 Longitude:-
122.35507321130525
Sent Key:driver-1 Latitude:47.618579488930855 Longitude:-
122.35494018791431
...
```

7. Leave your Avro producer application running. You can view the web application at http://localhost:3002.
8. Stop the debugger in VS Code.

OPTIONAL: Inspecting the Schema Registry REST API

Next you will inspect the contents and settings of Schema Registry via the REST API. See more details about the API at Schema Registry API Reference.

1. Find all the **subjects** in your Schema Registry:

```
$ curl schema-registry:8081/subjects
["driver-positions-avro-value"]
```

2. How many versions do you see for your subject?

```
$ curl schema-registry:8081/subjects/driver-positions-avro-
value/versions
[1]
```

3. View the contents of version 1 of the schema:

```
$ curl -s schema-registry:8081/subjects/driver-positions-avro-
value/versions/1
{"subject":"driver-positions-avro-
value","version":1,"id":1,"schema":"{\"type\":\"record\",\"name\":\"P
ositionValue\",\"namespace\":\"clients.avro\",\"fields\":[{\"name\":\
"latitude\",\"type\":\"double\"},{\"name\":\"longitude\",\"type\":\"d
ouble\"}]}"}
```

You can get the schema for a specific version of a subject with the `/subjects/(string:`

`subject)/versions/(versionId: version)/schema` path. You can pipe this to `jq` for pretty printing:

```
$ curl -s schema-registry:8081/subjects/driver-positions-avro-
value/versions/1/schema \
  | jq .
{
  "type": "record",
  "name": "PositionValue",
  "namespace": "clients.avro",
  "fields": [
    {
      "name": "latitude",
      "type": "double"
    },
    {
      "name": "longitude",
      "type": "double"
    }
  ]
}
```

4. Our Avro producer self-registered the `driver-positions-avro-value` schema subject when it produced its first record. In a production environment, we would typically have pre-registered the schema subject using the Schema Registry REST API. Let's run the command to do so now and observe the result.

```
$ curl -XPOST  -H "Content-Type:
application/vnd.schemaregistry.v1+json"  schema-
registry:8081/subjects/driver-positions-avro-value/versions/  -d '{
"schema":"{\"type\":\"record\",\"name\":\"PositionValue\",\"namespace
\":\"clients.avro\",\"fields\":[{\"name\":\"latitude\",\"type\":\"dou
ble\"},{\"name\":\"longitude\",\"type\":\"double\"}]}"}'
{"id":1}
```

   The command responds with the schema ID.

5. Check the default compatibility setting:

```
$ curl schema-registry:8081/config
{"compatibilityLevel":"BACKWARD"}
```

Writing the Avro Consumer

1. Use the command in the table below to navigate to the project folder for your language:

Language	Command
Java	`cd ~/confluent-dev/challenge/java-consumer-avro`
C#	`cd ~/confluent-dev/challenge/dotnet-consumer-avro`
Python	`cd ~/confluent-dev/challenge/python-consumer-avro`

2. Complete the same initialization steps you did for the producer exercise.

   a. Java

   ```
   $ ./gradlew build
   ```

   b. C#

   ```
   $ avrogen -s position_value.avsc . ; dotnet restore
   ```

   c. Python

   ```
   $ pip3 install -r requirements.txt
   ```

3. Open the project in Visual Studio Code:

   ```
   $ code .
   ```

4. Run the application by selecting the menu **Run** → **Start Debugging** in VS Code. You will see your application output:

```
Starting Java Avro Consumer.
Key:driver-1 Latitude:47.618579 Longitude:-122.355081 [partition 1]
Key:driver-1 Latitude:47.618577152452055 Longitude:-
122.35520620652974 [partition 1]
Key:driver-1 Latitude:47.61857902704408 Longitude:-122.35507321130525
[partition 1]
Key:driver-1 Latitude:47.618579488930855 Longitude:-
122.35494018791431 [partition 1]
Key:driver-1 Latitude:47.61857995081763 Longitude:-122.35480716452278
[partition 1]
...
```

Extra Challenges and Questions

1. Inspect the logs of your Schema Registry docker container:

   ```
   $ docker-compose logs schema-registry | grep '/schemas/ids/1'
   ```

   How many requests to `GET /schemas/ids/1` do you see? Can you explain the number of requests?

2. Modify the earlier `curl -XPOST` command to register a new schema version that doesn't meet the current Schema Registry compatibility setting.

3. Advanced challenge: Try adding a field with a default value to your AVRO producer, for example:

   ```
   {"name": "latitude", "type": "double"},
   {"name": "longitude", "type": "double"},
   {"name": "altitude", "type": "double", "default": 0.0}
   ```

   Would this be BACKWARD compatible? Would this be FORWARD compatible? See the documentation for Compatibility Types. Try producing data to your existing topic with a dummy value for altitude (fun fact: Seattle's highest point is 512ft). Can the consumer application or web application still consume from this topic?

Extra Challenges and Questions Solutions

1. A consumer loads a schema when it first sees a record for the schema id, and caches the result for subsequent records.

2. Adding a field without a default value would not meet the BACKWARD compatibility requirement, for example:

```
$ curl -XPOST  -H "Content-Type:
application/vnd.schemaregistry.v1+json"  schema-
registry:8081/subjects/driver-positions-avro-value/versions/  -d '{
"schema":"{\"type\":\"record\",\"name\":\"PositionValue\",\"namespace
\":\"clients.avro\",\"fields\":[{\"name\":\"latitude\",\"type\":\"dou
ble\"},{\"name\":\"longitude\",\"type\":\"double\"},{\"name\":\"new_f
ield\",\"type\":\"double\"}]}"}'
{"error_code":409,"message":"Schema being registered is incompatible
with an earlier schema for subject \"driver-positions-avro-value\",
details: [Incompatibility{type:READER_FIELD_MISSING_DEFAULT_VALUE,
location:/fields/2, message:new_field,
reader:{\"type\":\"record\",\"name\":\"PositionValue\",\"namespace\":
\"clients.avro\",\"fields\":[{\"name\":\"latitude\",\"type\":\"double
\"},{\"name\":\"longitude\",\"type\":\"double\"},{\"name\":\"new_fiel
d\",\"type\":\"double\"}]},
writer:{\"type\":\"record\",\"name\":\"PositionValue\",\"namespace\":
\"clients.avro\",\"fields\":[{\"name\":\"latitude\",\"type\":\"double
\"},{\"name\":\"longitude\",\"type\":\"double\"}]}}]"}
```

3. Adding a field with a default value is both FORWARD and BACKWARD compatible. If you were to produce data to the `driver-positions-avro` topic with a value for altitude consumers built with version 1 of the schema would ignore the values for altitude, making this update FORWARD compatible.

STOP HERE. THIS IS THE END OF THE EXERCISE.

Lab 08 Data Pipelines with Kafka Connect

a. Kafka Connect - Database to Kafka

The goal of this lab is to build a data pipeline that captures all the changes to a database table `driver-profiles` and writes the changes to a Kafka topic `driver-profiles-avro`. This can all be automated using the Kafka Connect and the JDBC source connector.

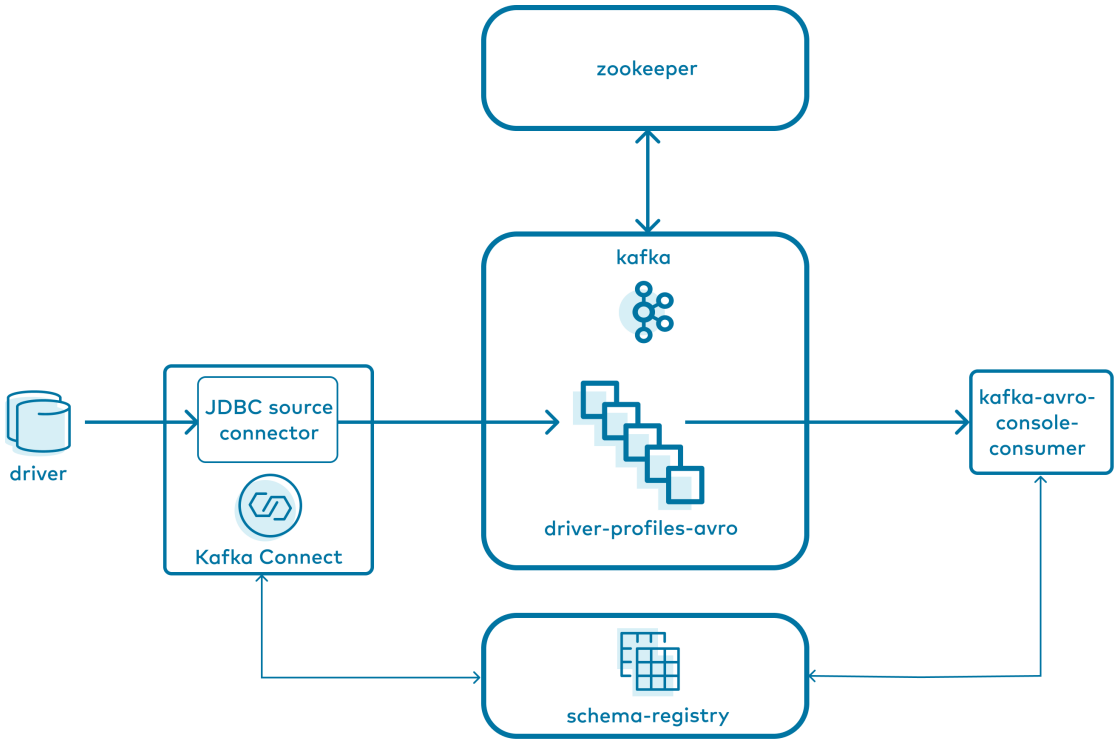
Prerequisites

1. Navigate to the project folder:

```
$ cd ~/confluent-dev
```

2. Run the Kafka cluster:

```
$ docker-compose up -d zookeeper kafka control-center create-topics \
    schema-registry connect postgres
```



You have now turned on containers for Kafka Connect and a Postgres database. If you look in `postgres/docker-entrypoint-initdb.d/001-driver.sql` you can see the SQL script used to create and populate the `driver-profiles` table in the Postgres database.

Inspecting Postgres

1. Inspect the contents of the `driver-profiles` table by first connecting to the Postgres database:

```
$ psql -h postgres -U postgres
psql (11.2)
Type "help" for help.

postgres=#
```

2. At the postgres prompt use a SQL select statement to view the contents of the `driver-profiles` table:

```
postgres=# select * from driver;
 id | driverkey | firstname | lastname |  make   |  model   |
timestamp
----+-----------+-----------+----------+---------+----------+
---------------------------
  1 | driver-1  | Randall   | Palmer   | Toyota  | Offset   | 2020-
01-26 01:11:31.707991
  2 | driver-2  | Razı      | İnönü    | Nissan  | Narkhede | 2020-
01-26 01:11:31.709005
...
```

3. Press `Q` to exit the Table View.
4. Exit `psql` by pressing `Ctrl+D`.

Install the Kafka Connect JDBC Connector

We use the Kafka Connect JDBC connector in this exercise so we need to install it on the worker.

1. Install the connector with the following command (and expected response):

```
$ docker-compose exec -u root connect confluent-hub install
confluentinc/kafka-connect-jdbc:10.6.0
The component can be installed in any of the following Confluent
Platform installations:
  1. / (installed rpm/deb package)
  2. / (where this tool is installed)
Choose one of these to continue the installation (1-2):
```

2. At the prompt, type `1` and press **Enter**:

```
Choose one of these to continue the installation (1-2): 1
```

3. You'll be prompted again. At the prompt, type `y` and press **Enter**.

```
Do you want to install this into /usr/share/confluent-hub-components?
(yN) y
```

4. You'll be prompted again. At the prompt, type `y` and press **Enter**.

```
Component's license:
Confluent Community License
https://www.confluent.io/confluent-community-license
I agree to the software license agreement (yN) y
```

5. You'll be prompted again. At the prompt, type `y` and press **Enter**.

```
Downloading component Kafka Connect JDBC 10.0.0, provided by
Confluent, Inc. from Confluent Hub and installing into
/usr/share/java/kafka
Detected Worker's configs:
  1. Standard: /etc/kafka/connect-distributed.properties
  2. Standard: /etc/kafka/connect-standalone.properties
  3. Standard: /etc/schema-registry/connect-avro-
distributed.properties
  4. Standard: /etc/schema-registry/connect-avro-
standalone.properties
  5. Used by Connect process with PID : /etc/kafka-connect/kafka-
connect.properties
Do you want to update all detected configs? (yN) y
```

The installation completes.

```
Adding installation directory to plugin path in the following files:
  /etc/kafka/connect-distributed.properties
  /etc/kafka/connect-standalone.properties
  /etc/schema-registry/connect-avro-distributed.properties
  /etc/schema-registry/connect-avro-standalone.properties
  /etc/kafka-connect/kafka-connect.properties

Completed
```

6. To complete the installation, we need to restart the `connect` container:

```
$ docker-compose restart connect
```

7. Verify that the Connect Worker successfully restarted prior to continuing to the next step:

```
$ docker-compose logs connect | grep -i "INFO .* Finished starting
connectors and tasks"
connect              | [2022-04-07 18:16:59,032] INFO [Worker
clientId=connect-1, groupId=connect] Finished starting connectors and
tasks
(org.apache.kafka.connect.runtime.distributed.DistributedHerder)
connect              | [2022-04-07 18:32:14,011] INFO [Worker
clientId=connect-1, groupId=connect] Finished starting connectors and
tasks
(org.apache.kafka.connect.runtime.distributed.DistributedHerder)
```

Repeat this command until the **Finished starting connectors and tasks** message appears twice.

Configure the AVRO source connector

1. Add a JDBC source connector via command line with the `curl` command below. Let's focus on the transformations that are happening in the connector settings. You can read more about transformations in the documentation for Kafka Connect Transformations.

   a. **RegexRouter** By default the records would be written to a topic with the same name as the table. The setting here will update record topic to `driver-profiles-avro`.

   b. **ValueToKey** The connector is configured to use the `driverkey` property as the record key. At this point the key in the record would look like `{driverkey=driver-5}`.

c. **ExtractField$Key** The connector extracts the `driverkey` field from the key and replaces the entire key with the extracted field. A key of `{driverkey=driver-5}` would be replaced with `driver-5`.

```
$ curl -X POST \
  -H "Content-Type: application/json" \
  --data '{
    "name": "Driver-Connector-Avro",
    "config": {
      "connector.class":
"io.confluent.connect.jdbc.JdbcSourceConnector",
      "connection.url":
"jdbc:postgresql://postgres:5432/postgres",
      "connection.user": "postgres",
      "table.whitelist": "driver",
      "topic.prefix": "",
      "mode":"timestamp+incrementing",
      "incrementing.column.name": "id",
      "timestamp.column.name": "timestamp",
      "table.types": "TABLE",
      "numeric.mapping": "best_fit",
      "key.converter":
"org.apache.kafka.connect.storage.StringConverter",
      "value.converter":
"io.confluent.connect.avro.AvroConverter",
      "value.converter.schema.registry.url": "http://schema-
registry:8081",
      "transforms": "suffix,createKey,extractKey",

"transforms.suffix.type":"org.apache.kafka.connect.transforms.Rege
xRouter",
      "transforms.suffix.regex":"(.*)",
      "transforms.suffix.replacement":"$1-profiles-avro",
      "transforms.createKey.type":
"org.apache.kafka.connect.transforms.ValueToKey",
      "transforms.createKey.fields": "driverkey",
      "transforms.extractKey.type":
"org.apache.kafka.connect.transforms.ExtractField$Key",
      "transforms.extractKey.field": "driverkey"
    }
}' http://connect:8083/connectors
```

the answer should be something like this:

```
{"name":"Driver-Connector-
Avro","config":{"connector.class":"io.confluent.connect.jdbc.JdbcS
ourceConnector","connection.url":"jdbc:postgresql://postgres:5432/
postgres","connection.user":"postgres","table.whitelist":"driver",
"topic.prefix":"","mode":"timestamp+incrementing","incrementing.co
lumn.name":"id","timestamp.column.name":"timestamp","table.types":
"TABLE","numeric.mapping":"best_fit","key.converter":"org.apache.k
afka.connect.storage.StringConverter","value.converter":"io.conflu
ent.connect.avro.AvroConverter","value.converter.schema.registry.u
rl":"http://schema-
registry:8081","transforms":"suffix,createKey,extractKey","transfo
rms.suffix.type":"org.apache.kafka.connect.transforms.RegexRouter"
,"transforms.suffix.regex":"(.*)","transforms.suffix.replacement":
"$1-profiles-
avro","transforms.createKey.type":"org.apache.kafka.connect.transf
orms.ValueToKey","transforms.createKey.fields":"driverkey","transf
orms.extractKey.type":"org.apache.kafka.connect.transforms.Extract
Field$Key","transforms.extractKey.field":"driverkey","name":"Drive
r-Connector-Avro"},"tasks":[],"type":"source"}
```

2. Let's see what we get:

```
$ kafka-avro-console-consumer \
    --bootstrap-server kafka:9092 \
    --property schema.registry.url=http://schema-registry:8081 \
    --topic driver-profiles-avro \
    --property print.key=true \
    --key
-deserializer=org.apache.kafka.common.serialization.StringDeserialize
r \
    --from-beginning
```

and we should see something like this:

```
driver-2    {"id":2,"driverkey":"driver-
2","firstname":"Razı","lastname":"İnönü",...
driver-6    {"id":6,"driverkey":"driver-
6","firstname":"William","lastname":"Peterson",...
driver-10   {"id":10,"driverkey":"driver-
10","firstname":"Aaron","lastname":"Gill",...
...
```

It may take several seconds for the records to appear.

3. Exit the consumer with `Ctrl+C`.

Configure the Protobuf source connector

Schema Registry 5.5 added support for Protobuf and JSON Schema along with Avro. You can now add a connector using a `ProtobufConverter` class. The connector will source the same data from the Postgres database, register a Protobuf schema, and write the Protobuf serialized bytes to the Kakfa topic `driver-profiles-protobuf`.

1. The command below is nearly the same as your previous command. We have changed the: name, value.converter, and topic suffix.

```
$ curl -X POST \
  -H "Content-Type: application/json" \
  --data '{
    "name": "Driver-Connector-Protobuf",
    "config": {
      "connector.class":
"io.confluent.connect.jdbc.JdbcSourceConnector",
      "connection.url": "jdbc:postgresql://postgres:5432/postgres",
      "connection.user": "postgres",
      "table.whitelist": "driver",
      "topic.prefix": "",
      "mode":"timestamp+incrementing",
      "incrementing.column.name": "id",
      "timestamp.column.name": "timestamp",
      "table.types": "TABLE",
      "numeric.mapping": "best_fit",
      "key.converter":
"org.apache.kafka.connect.storage.StringConverter",
      "value.converter":
"io.confluent.connect.protobuf.ProtobufConverter",
      "value.converter.schema.registry.url": "http://schema-
registry:8081",
      "transforms": "suffix,createKey,extractKey",

"transforms.suffix.type":"org.apache.kafka.connect.transforms.RegexRo
uter",
      "transforms.suffix.regex":"(.*)",
      "transforms.suffix.replacement":"$1-profiles-protobuf ",
      "transforms.createKey.type":
"org.apache.kafka.connect.transforms.ValueToKey",
      "transforms.createKey.fields": "driverkey",
      "transforms.extractKey.type":
"org.apache.kafka.connect.transforms.ExtractField$Key",
      "transforms.extractKey.field": "driverkey"
    }
}' http://connect:8083/connectors
```

the answer should be something like this:


```
{"name":"Driver-Connector-
Protobuf","config":{"connector.class":"io.confluent.connect.jdbc.Jdbc
SourceConnector","connection.url":"jdbc:postgresql://postgres:5432/po
stgres","connection.user":"postgres","table.whitelist":"driver","topi
c.prefix":"","mode":"timestamp+incrementing","incrementing.column.nam
e":"id","timestamp.column.name":"timestamp","table.types":"TABLE","nu
meric.mapping":"best_fit","key.converter":"org.apache.kafka.connect.s
torage.StringConverter","value.converter":"io.confluent.connect.proto
buf.ProtobufConverter","value.converter.schema.registry.url":"http://
schema-
registry:8081","transforms":"suffix,createKey,extractKey","transforms
.suffix.type":"org.apache.kafka.connect.transforms.RegexRouter","tran
sforms.suffix.regex":"(.*)","transforms.suffix.replacement":"$1-
profiles-protobuf
","transforms.createKey.type":"org.apache.kafka.connect.transforms.Va
lueToKey","transforms.createKey.fields":"driverkey","transforms.extra
ctKey.type":"org.apache.kafka.connect.transforms.ExtractField$Key","t
ransforms.extractKey.field":"driverkey","name":"Driver-Connector-
Protobuf"},"tasks":[],"type":"source"}
```


2. The results will look the same as the AVRO topic. This is because **`kafka-protobuf-console-consumer`** is deserializing the bytes.

```
$ kafka-protobuf-console-consumer \
    --bootstrap-server kafka:9092 \
    --property schema.registry.url=http://schema-registry:8081 \
    --topic driver-profiles-protobuf \
    --property print.key=true \
    --key
-deserializer=org.apache.kafka.common.serialization.StringDeserialize
r \
    --from-beginning
```

3. Exit the consumer with **`Ctrl+C`**.

4. The bytes in the **`driver-profiles-protobuf`** topic are Protobuf serialized. We can see the raw bytes using **`kafkacat`** and piping the results to **`hexdump`**. See more about kafkacat at https://github.com/edenhill/kafkacat. From the kafkacat documentation:

> kafkacat is a generic non-JVM producer and consumer for Apache Kafka >=0.8, think of it as a netcat for Kafka.

5. Run the command below to see the raw bytes of one message on the **`driver-profiles-protobuf`** topic. Let's focus on the options we will be using:

`-b` Bootstrap broker(s).

`-C` Consume mode.

`-c1` Exit after consuming 1 message.

`-t` Topic to consume from.

```
$ kafkacat -b kafka:9092 -C -c1 -t driver-profiles-protobuf | hexdump
-C
00000000  00 00 00 00 03 00 08 09  12 08 64 72 69 76 65 72
|..........driver|
00000010  2d 39 1a 06 e5 8a a0 e5  a5 88 22 06 e5 b0 8f e6  |-
9........".....|
00000020  9e 97 2a 02 47 4d 32 08  42 65 72 67 6c 75 6e 64
|..*.GM2.Berglund|
00000030  3a 0c 08 af dd bf f6 05  10 c0 e2 b9 e3 01 0a
|:..............|
0000003f
```

The wire format documentation explains the format of the bytes. The 0th byte is `00` for the format version number. The next 4 bytes `00 00 00 03` tell us the schema id. The remaining bytes are the Protobuf serialized data.

6. For comparison you can inspect the raw bytes on the `driver-profiles-avro` topic. You can see the the format version number, schema id and *AVRO* serialized data.

```
$ kafkacat -b kafka:9092 -C -c1 -t driver-profiles-avro | hexdump -C
00000000  00 00 00 00 01 12 10 64  72 69 76 65 72 2d 39 0c
|.......driver-9.|
00000010  e5 8a a0 e5 a5 88 0c e5  b0 8f e6 9e 97 04 47 4d
|..............GM|
00000020  10 42 65 72 67 6c 75 6e  64 ae 8b a2 a0 ce 5c 0a
|.Berglund.....\.|
00000030
```

7. We can request the contents of schema just created via the Schema Registry REST API:

```
$ curl schema-registry:8081/subjects/driver-profiles-protobuf-
value/versions/1/schema
syntax = "proto3";

import "google/protobuf/timestamp.proto";

message driver {
  int32 id = 1;
  string driverkey = 2;
  string firstname = 3;
  string lastname = 4;
  string make = 5;
  string model = 6;
  google.protobuf.Timestamp timestamp = 7;
}
```

Extra Challenges and Questions

1. Leave the `kafka-avro-console-consumer` reading from the `driver-profiles-avro` topic in a terminal window. Use `psql` to update a row in the `driver` table, and see the update appear on the `driver-profiles-avro` topic. Hint: the `timestamp` column will need to be updated for connect to detect the changes, set timestamp to `now()` for the current date time.
2. Can you use `kafka-topics` to determine the log cleaning policy for the `driver-profiles-avro` topic? Why would this policy have been chosen?

Extra Challenges and Questions Solutions

1. This `UPDATE` statement in `psql` will update a single row in the drivers table:

   ```
   UPDATE driver SET firstname='Bill', timestamp=now() WHERE id = 6;
   ```

   You will see the update appear on the `driver-profiles-avro` topic:

   ```
   driver-6 {"id":6,"driverkey":"driver-
   6","firstname":"Bill","lastname":"Peterson","make":"GM","model":"Berg
   land","timestamp":"1584128781527"}
   ```

2. We can see the topic property with `kafka-topics`:

   ```
   $ kafka-topics --bootstrap-server kafka:9092 --describe --topic
   driver-profiles-avro
   Topic: driver-profiles-avro PartitionCount: 3   ReplicationFactor: 1
   Configs: cleanup.policy=compact
   ```

   Log compaction means we will always retain at least the last known value for each message. We are only interested in the most recent record for a key, and can clean out any previous values.

Conclusion

In this exercise you have used a Kafka Connect JDBC source connector to import data residing in PostgreSQL database into the topics `driver-profiles-avro` and `driver-`

`profiles-protobuf` in the Kafka cluster. This data can now, e.g., be used to enrich our driver position data in the next exercise.

STOP HERE. THIS IS THE END OF THE EXERCISE.

Lab 09 Setting up the Lab and Cloud Environment

a. Setting up the Lab and Cloud Environment

1. Open a terminal window
2. Clone the source code repository to the folder `confluent-flink` in your **home** directory:

```
$ cd ~
$ git clone --depth 1 --branch 1.19.0-v1.0.0 \
    https://github.com/borjahernandez/training-flk-sql-cc.git \
    confluent-flink
```

If you chose to select another folder for the labs then note that many of our samples assume that the lab folder is `~/confluent-flink`. You will have to adjust all those command to fit your specific environment.

3. Navigate to the `confluent-flink` folder:

```
$ cd ~/confluent-flink
```

4. List the content of the `confluent-flink` folder:

```
$ ls -l
```

Doublecheck that the content is:

```
total 4
drwxr-xr-x 6 training users 4096 Jul 15 09:39 terraform
```

Creating your Confluent Cloud account

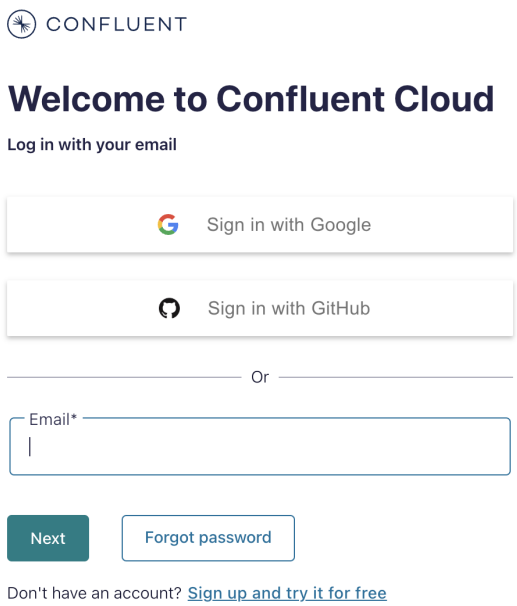
During this course, you will perform all exercises using your own Confluent Cloud account.

As the use of Confluent Cloud resources have a cost associated, you will be provided of **voucher code** to plenty cover the cost of performing all course exercises.

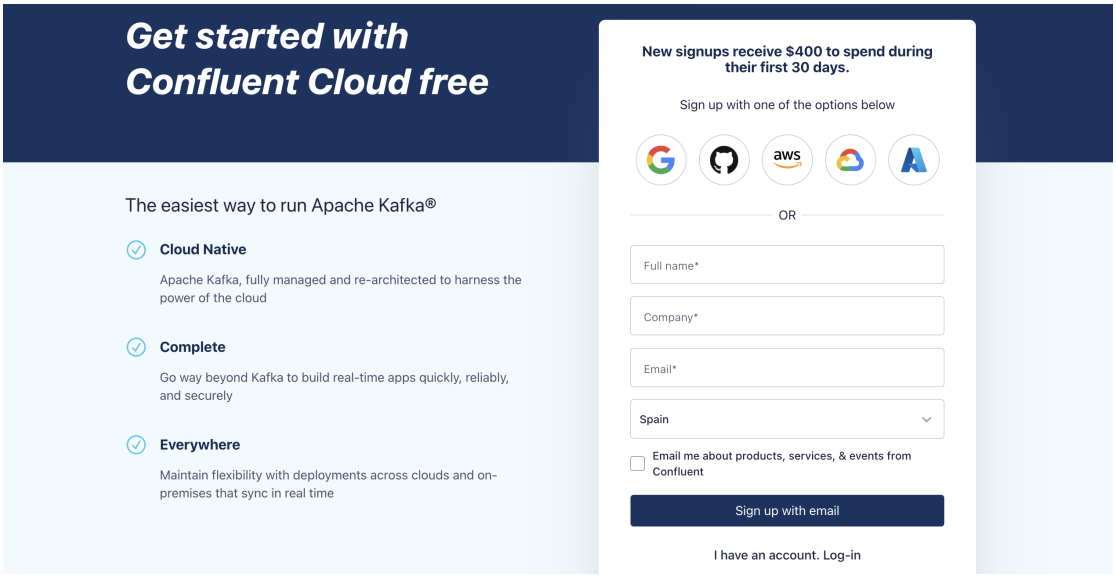
Make sure you delete all Confluent Cloud resources at the end of the exercises. We will guide through the deletion process. If you don't complete the deletion, you will keep incurring in more costs and potentially consuming all the free credit.

After the free credit is over, your Organization will be suspended stopping further charges. **Unless you previously added a credit card**. More information here.

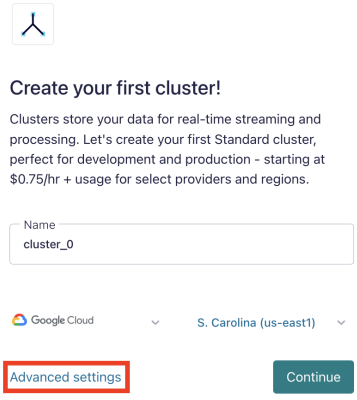
7. Open a new browser tab.
8. Navigate to Confluent Cloud website at the URL https://confluent.cloud and select **Sign up and try it for free**:



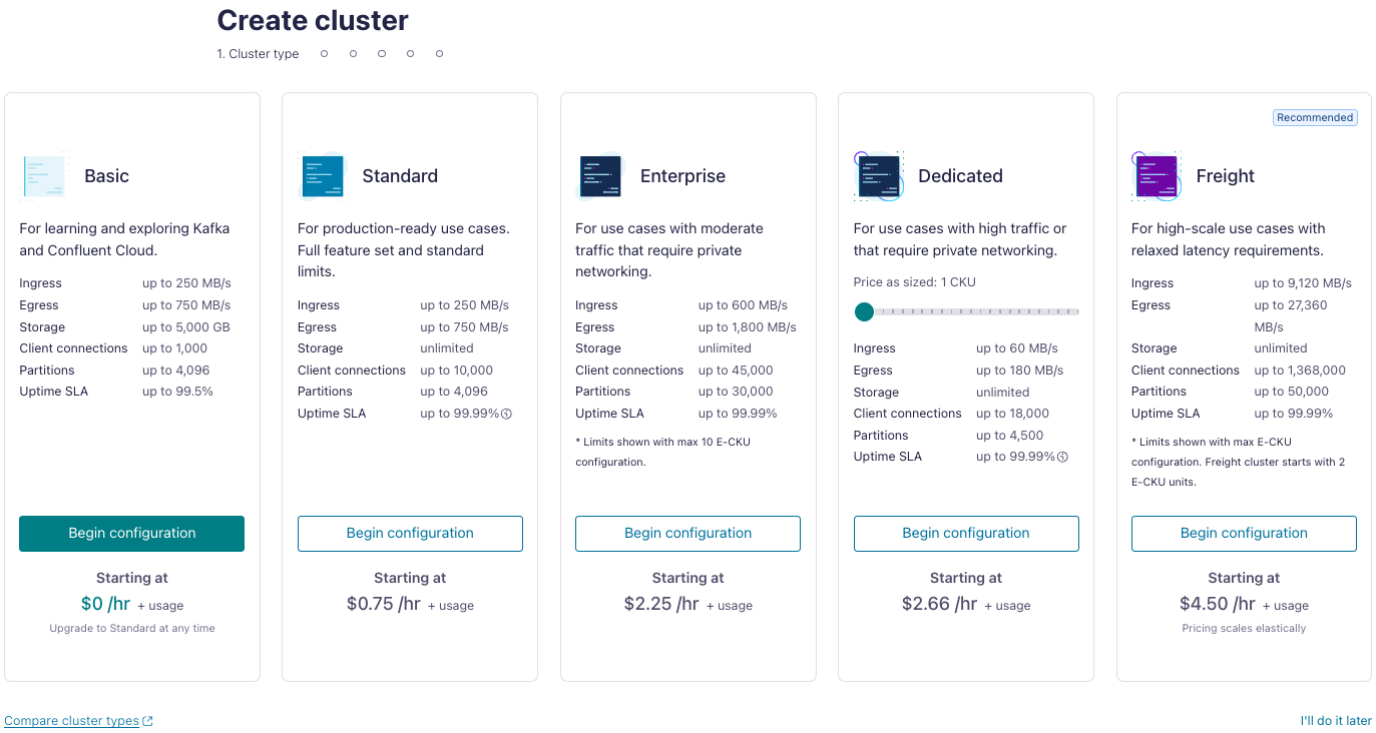
9. Sign up providing your name, company, email and country:



10. Click the **Sign up with email** button.
11. Watch your inbox for a confirmation email. Once you get the email, follow the link to create a password and proceed to the next step.
12. Once you are logged in, **Skip** the "Invite your teammates" page and then answer the survey to finish setting your account.
13. You are now asked to create your first cluster, click on **Advanced settings**.



14. Choose a **Basic cluster** by selecting **Begin configuration** to start:



15. Choose a **Cloud Provider**, **Region**, and **Uptime SLA**. Then, click on **Continue**.

Select one of these regions:

AWS

```
Oregon (us-west-2)
Ohio (us-east-2)
N. Virginia (us-east-1)
Frankfurt (eu-central-1)
Ireland (eu-west-1)
London (eu-west-2)
Mumbai (ap-south-1)
Singapore (ap-southeast-1)
Sydney (ap-southeast-2)
```

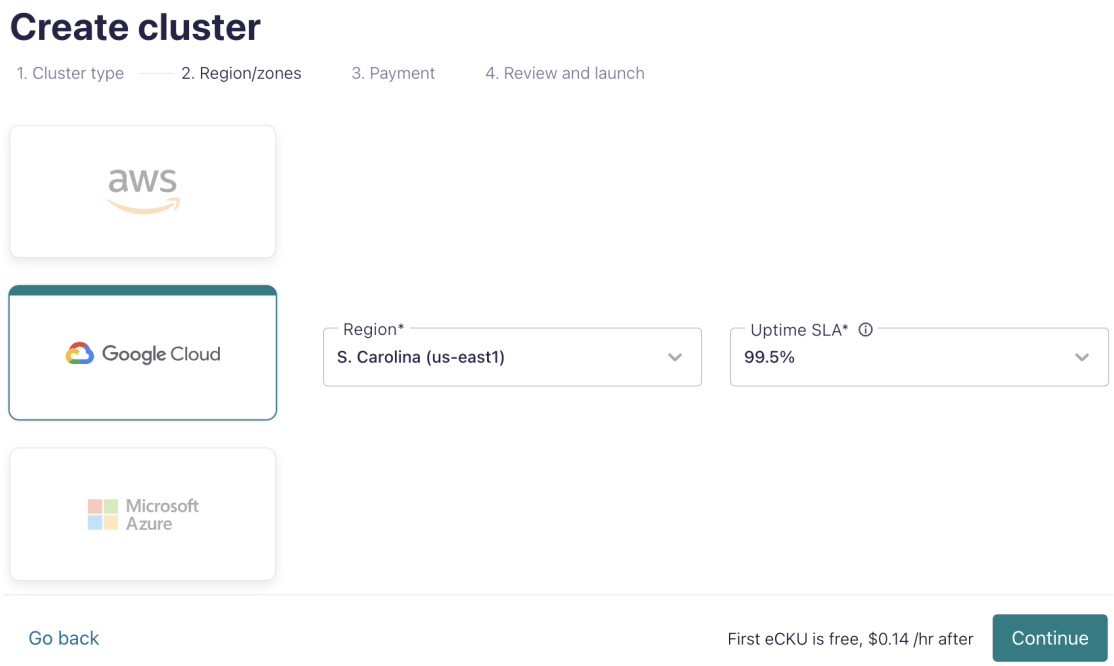
Google Cloud

```
Iowa (us-central1)
Las Vegas (us-west4)
N. Virginia (us-east4)
S. Carolina (us-east1)
Belgium (europe-west1)
Frankfurt (europe-west3)
Mumbai (asia-south1)
Delhi (asia-south2)
Singapore (asia-southeast1)
Sydney (australia-southeast1)
```

Microsoft Azure

```
Virginia (eastus)
Virginia (eastus2)
Washington (westus2)
Netherlands (westeurope)
Pune (centralindia)
Singapore (southeastasia)
```

Costs will vary with these choices, but they are clearly shown on the dropdown, so you'll know what you're getting.

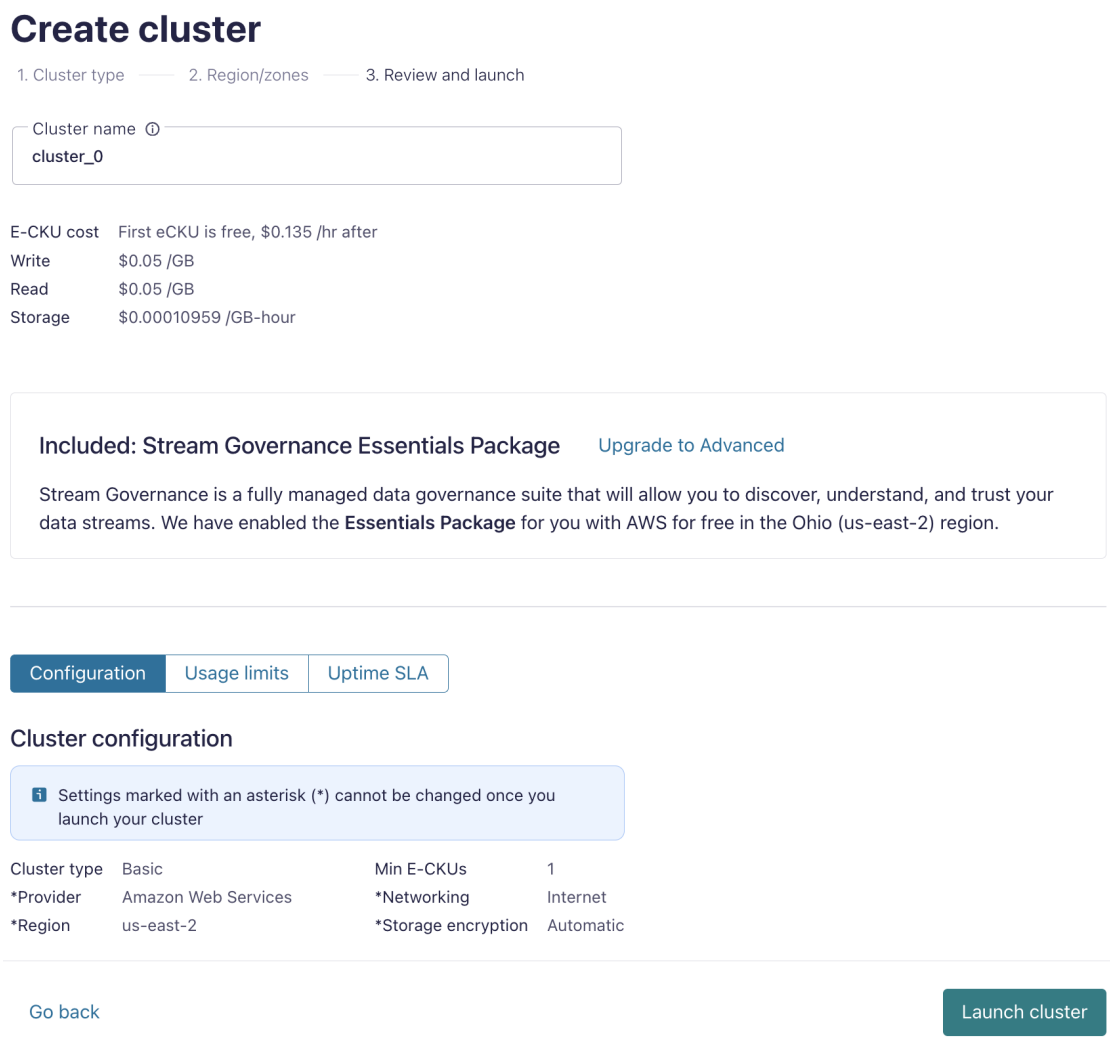


16. In the payment page:

    a. Scroll down and expand the **Promo codes** section.

    b. Enter the code that starts with **POPTOUT** and click **Apply**.

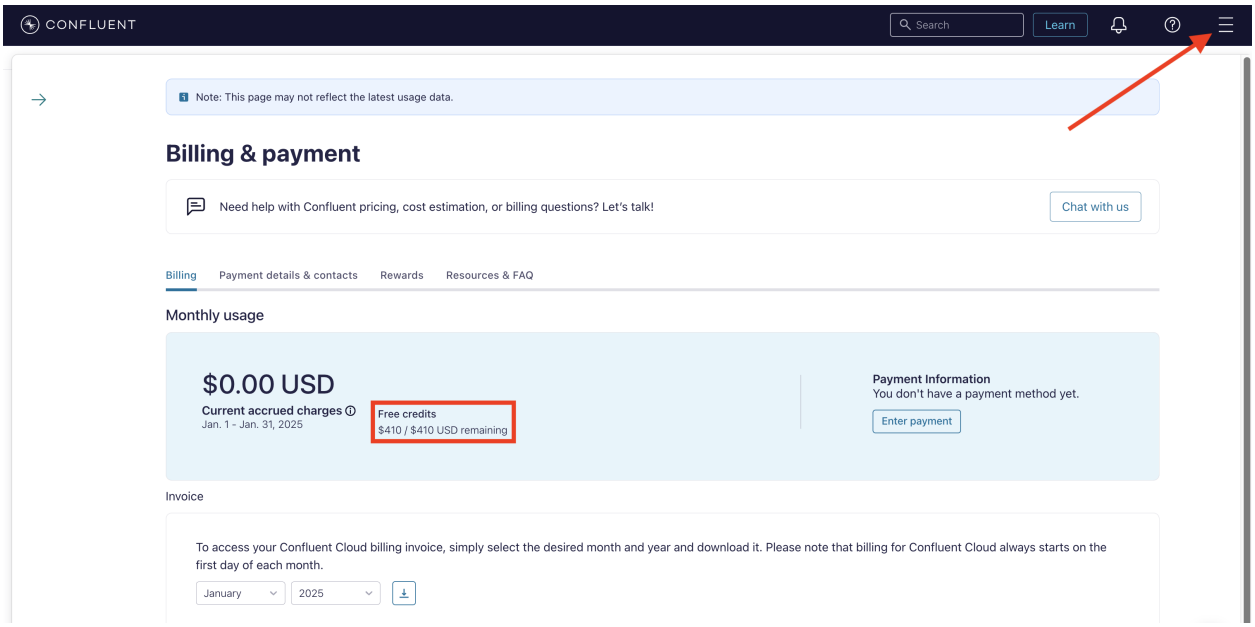
17. Give your cluster a **name** and review **Configuration**, **Usage limits** and **Uptime SLA**. Then, select **Launch cluster**.



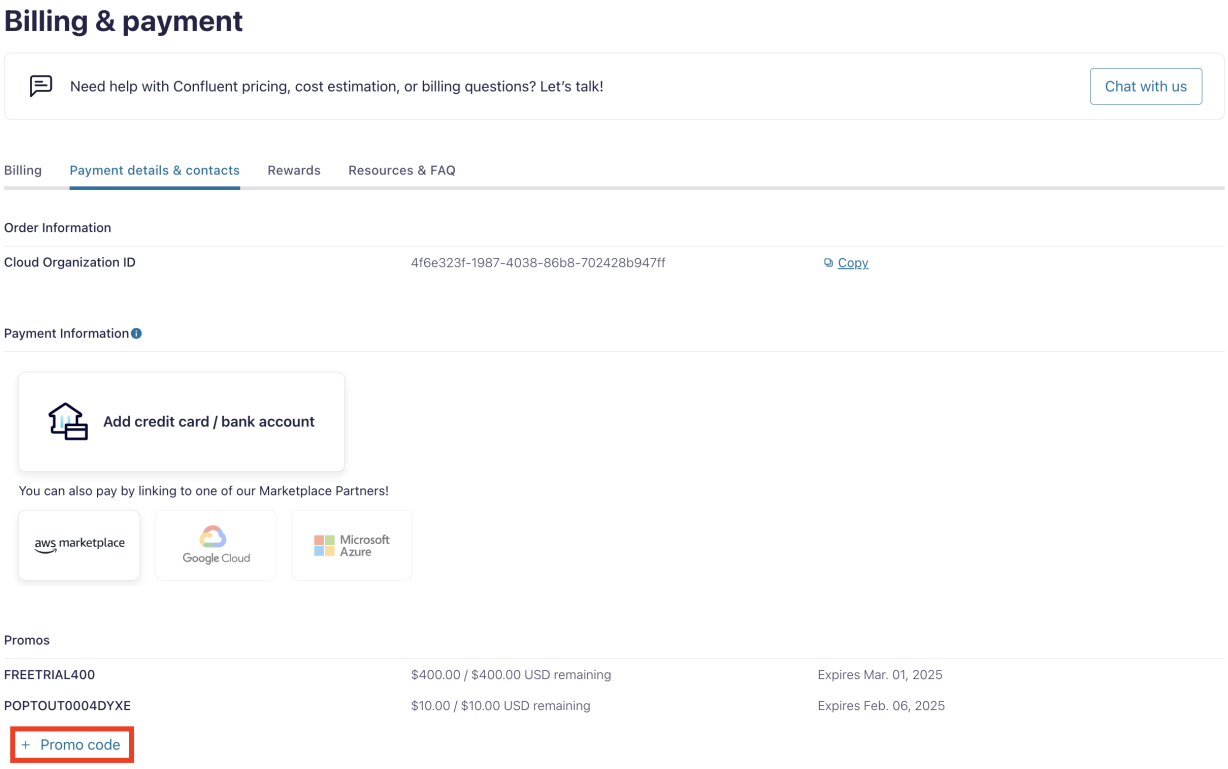
18. In a few seconds your new Kafka cluster will be provisioned.

Managing your Free Credit

19. Go to **Main Menu** (three lines icon) and click on **Billing & payment**. In this window you can check your free credit balance:



20. Now, click on the tab **Payment details & contacts**.



21. To add a new code to your Organization, click on **+ Promo code** button.

22. Enter the other **Promo Code** provided by the instructor which should be an 8 character code. Then, click **Save**.

Great! You have now created your own Confluent Cloud cluster. You will use this new cluster later in the course.

Setting up the Confluent CLI

Confluent CLI is already installed in the provided Virtual Machine.

23. Go to the Virtual Machine, open a new terminal window and run the following command to check its version:

```
$ confluent --version
confluent version v3.47.0
```

24. (Optional) You can update the Confluent CLI version to the latest version by running the following command:

    Password: **training**

```
$ sudo confluent update
```

25. Run the following command and log into the CLI by entering your email address and password you used to create the Confluent Cloud account:

```
$ confluent login
Enter your Confluent Cloud credentials:
Email: <enter email>
Password: <enter password>
```

If you use the command **`confluent login --save`**, your login credentials are stored locally in this file **`~/.confluent/config.json`** for non-interactive re-authentication. The password is stored encrypted.

26. Next, get a list of the Kafka clusters in your account, which should be just one:

```
$ confluent kafka cluster list
 Current |     ID     |     Name      | Type  | Provider |    Region
| ... | Status
---------+------------+---------------+-------+----------+
--------------+     +--------
         | lkc-7o8k9j | kafka_cluster | BASIC | aws      | eu-
central-1 | ... | UP
```

27. Set your cluster as the default so you don't need to keep referring to it every time you run a command. Run this command using your **cluster Id** from the previous step:

```
$ confluent kafka cluster use {{ CLUSTER_ID }}
```

Sample

```
$ confluent kafka cluster use lkc-7o8k9j
Set Kafka cluster "lkc-7o8k9j" as the active cluster for
environment "env-5qzg3n".
```

28. List again your Kafka clusters. Notice that now there is an **asterisk** in the **Current** column indicating the CLI is currently using this cluster:

```
$ confluent kafka cluster list
 Current |     ID     |     Name      | Type  | Provider |    Region
| ... | Status
---------+------------+---------------+-------+----------+
--------------+     +--------
 *       | lkc-7o8k9j | kafka_cluster | BASIC | aws      | eu-
central-1 | ... | UP
```

Conclusion

In this lab you have prepared and tested the Lab Environment. You have also created your Confluent Cloud account and a Kafka cluster that will be used in subsequent exercises.

(OPTIONAL) Installing Confluent CLI in your local machine

1. The installation of the Confluent CLI in your machine depends on your OS:

   a. For macOS and Linux, run:

   ```
   $ brew install confluentinc/tap/cli
   ```

   b. For Windows:

i. Download the latest Windows ZIP file from https://github.com/confluentinc/cli/releases/latest

ii. Unzip the file `confluent_X.X.X_windows_amd64.zip`

iii. Run `confluent.exe`

STOP HERE. THIS IS THE END OF THE EXERCISE.

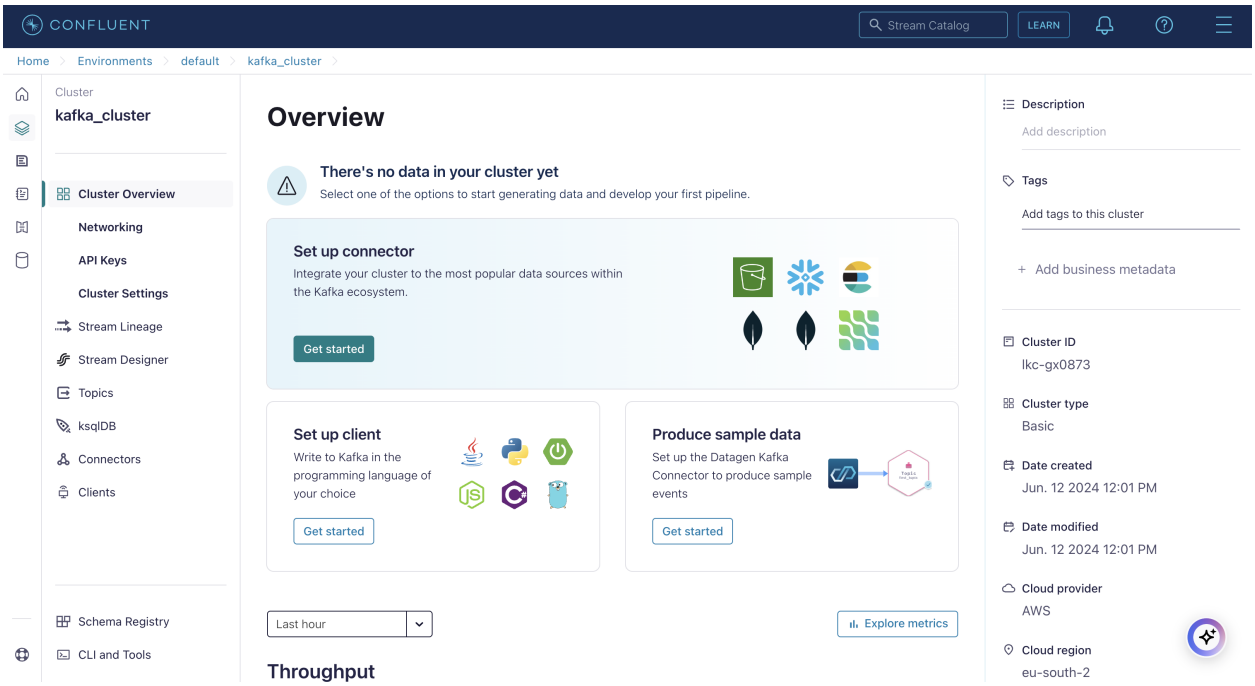
Lab 10 Working with Flink in Confluent Cloud

a. Working with Flink in Confluent Cloud

The goal of this lab is to familiarize yourself with the Confluent Cloud Web UI and the Confluent CLI for performing common tasks such as running Flink queries, navigating through catalogs and databases, managing Flink statements, and viewing compute pool information.

Creating your first Kafka topic

1. Go to Confluent Cloud website and log in to your Confluent Cloud account.
2. Navigate to the cluster you created in Lab 1, which should be inside the Environment called **default**.



3. In the left pane, click on **Topics**.
4. Now, click **Create topic** and use the following settings:
    a. Topic name: **orders**

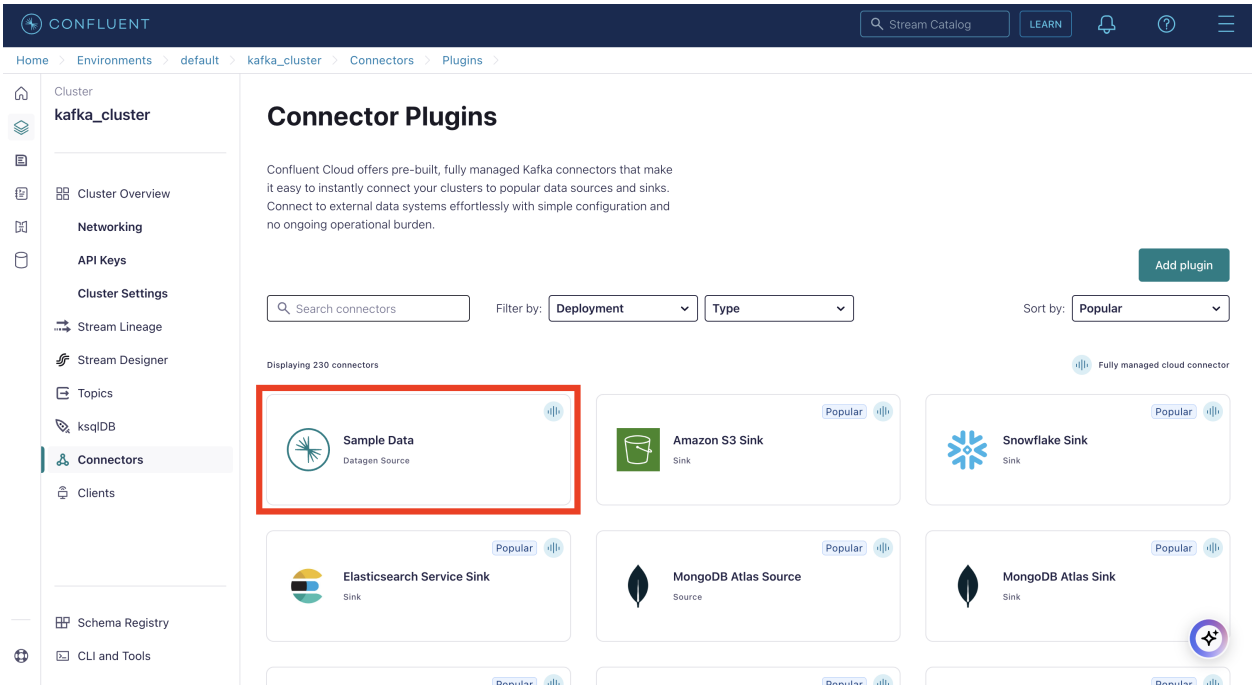
b. Partitions: **1**

5. Click on **Create with defaults** to complete the topic creation.
6. In the "**Define a data contract**" window, click on **Skip**. The schema for this topic will be created later.

Producing Data to your Kafka Topic

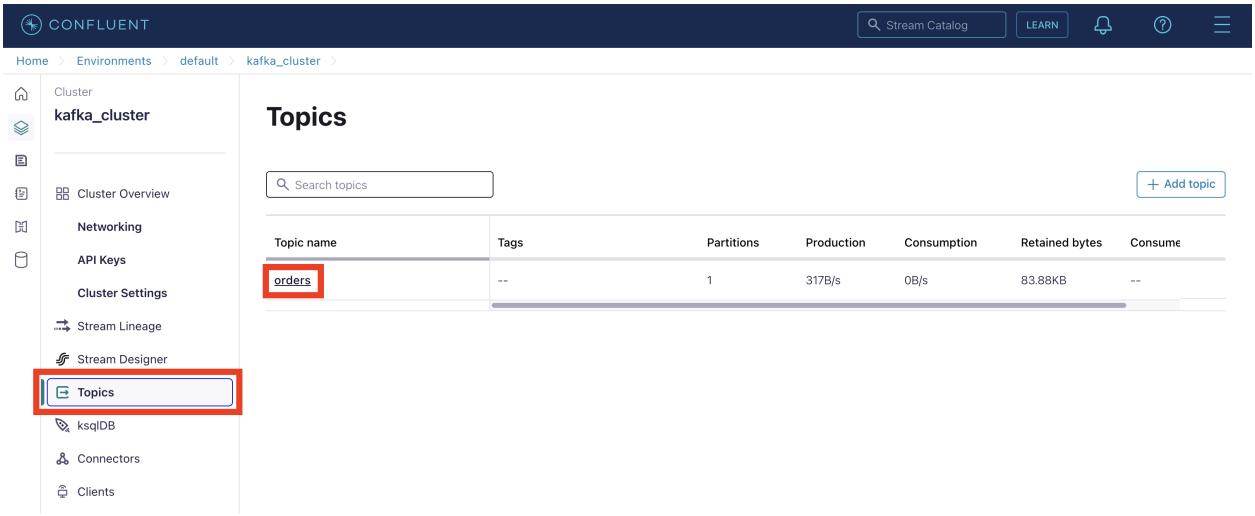
Let's create a DataGen connector to produce fake data to our topic.

7. In the left pane, click on **Connectors**.
8. In this window, you can view what connectors you have available in Confluent Cloud. Now, select the connector **Sample Data**.

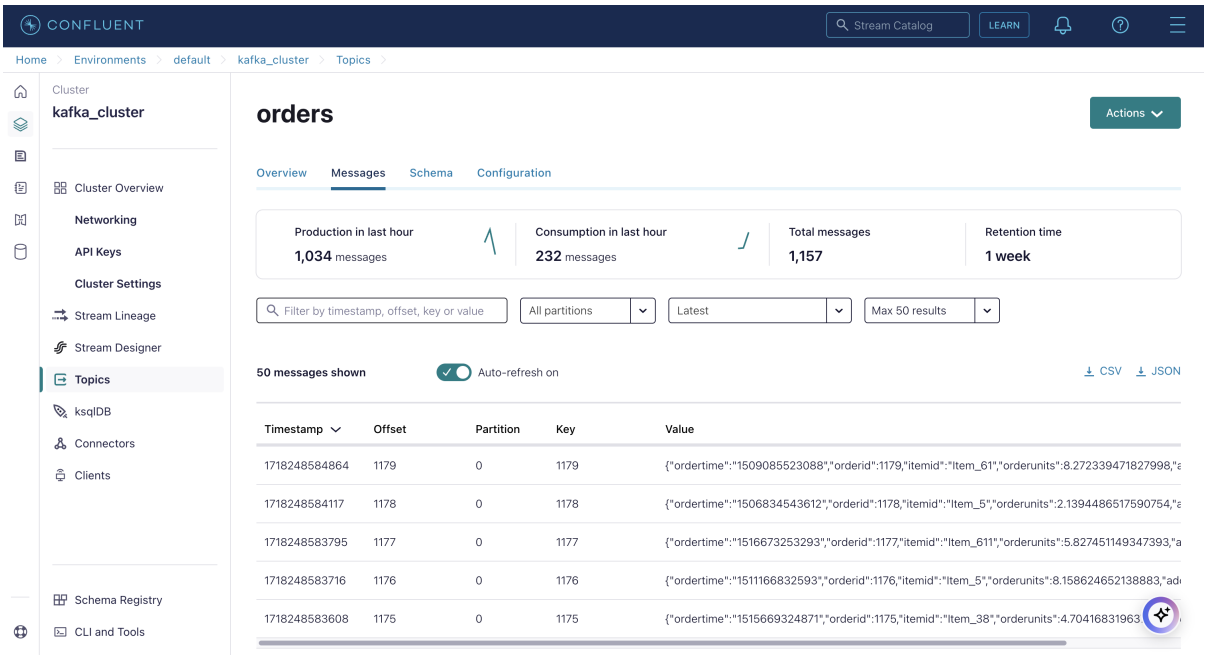


9. In the "**Launch Sample Data**" window, click on **Additional configuration**.
10. Select the topic **orders** and click **Continue**.
11. In Kafka Credentials, select **Service Account** and create a new service account:
    - New service account name: **sa-orders-connector**
    - Description: **Service Account for the orders Datagen connector**
    - **Check the box** for "Adding all required ACLs" and click on **Continue**

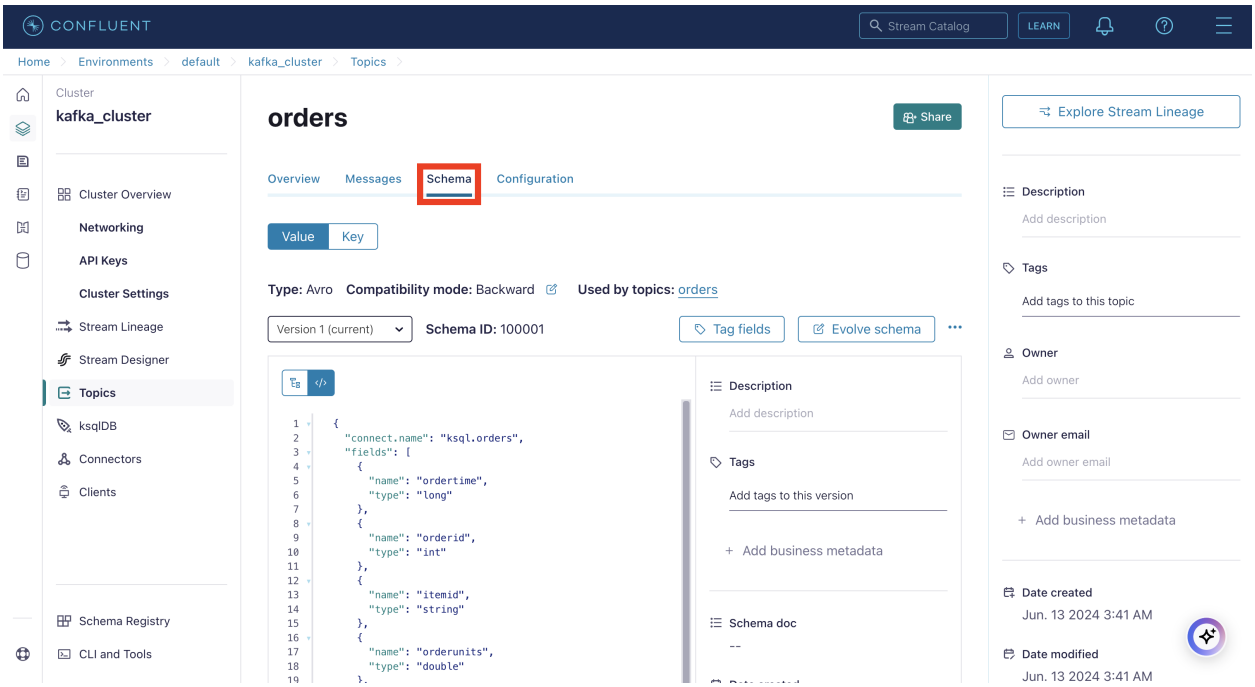
12. In Configuration, select **AVRO** as the output record value format and select **Orders** as the schema, then click on **Continue**.
13. In Sizing, leave the number of tasks to 1 and click **Continue**.
14. In Review and launch, set the name of the connector to **OrdersConnector** and click **Continue**.
15. After a few seconds, your new DataGen connector will start running.
16. Let's check that messages are being produced to topic **orders**. Go back to **Topics** and then click on **orders**.



17. Now, click on the tab **Messages**.
    a. Here, you view the messages that are currently being produced.
    b. You can also view messages in particular partitions and offsets by using the filters.
    c. If you click on a message you can view more details like the message headers or the timestamp in more readable format.



18. Finally, check the schema of the messages by clicking on the tab **Schema**.



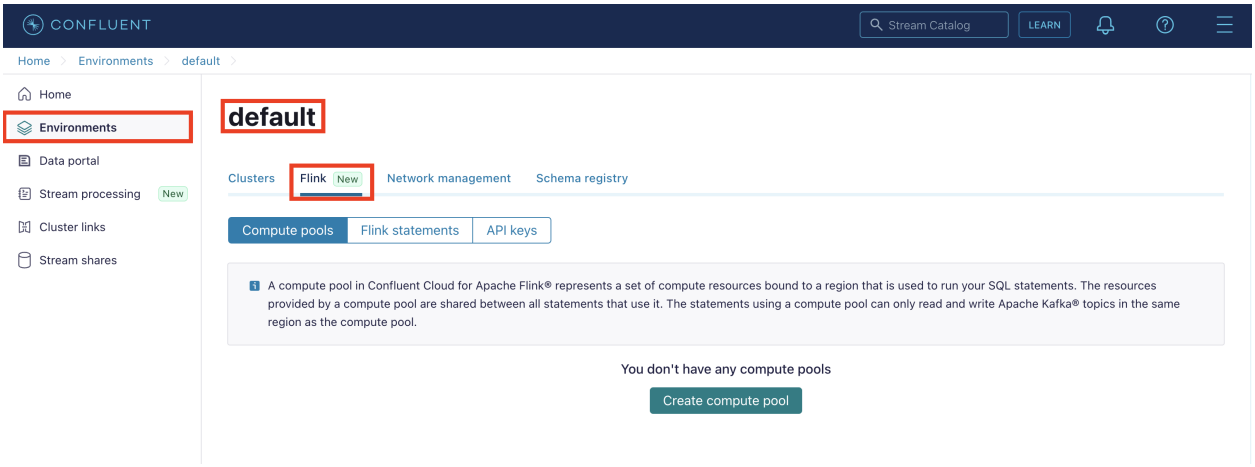
The AVRO schema has been automatically created by the DataGen connector.

Flink is going to use this schema in combination with the topic data in order to create a **Dynamic Table**.

Kafka Topic + Schema = Flink Table

Creating a Flink Compute Pool

19. Go to the environment **default** page and choose the **Flink** tab.



20. From here you can create a new compute pool. Click on **Create compute pool**:

> **Choose the same cloud provider and region as the ones you selected earlier for the Kafka cluster**.
> The Flink compute pool can only access the data from any Kafka cluster located in the same region as the compute pool.

   a. Choose the cloud provide and region, then click **Continue**

   b. Enter the Pool name: **flink_training_compute_pool** and choose **10 CFUs** (Confluent Flink Units)

   c. Finally, click **Create**

> **CFU definition**: It is a logical unit of processing power that is used to measure the resources consumed by Confluent Cloud for Apache Flink.
>
> Each Flink statement consumes a minimum of 1 CFU-minute but may consume more depending on the needs of the workload.

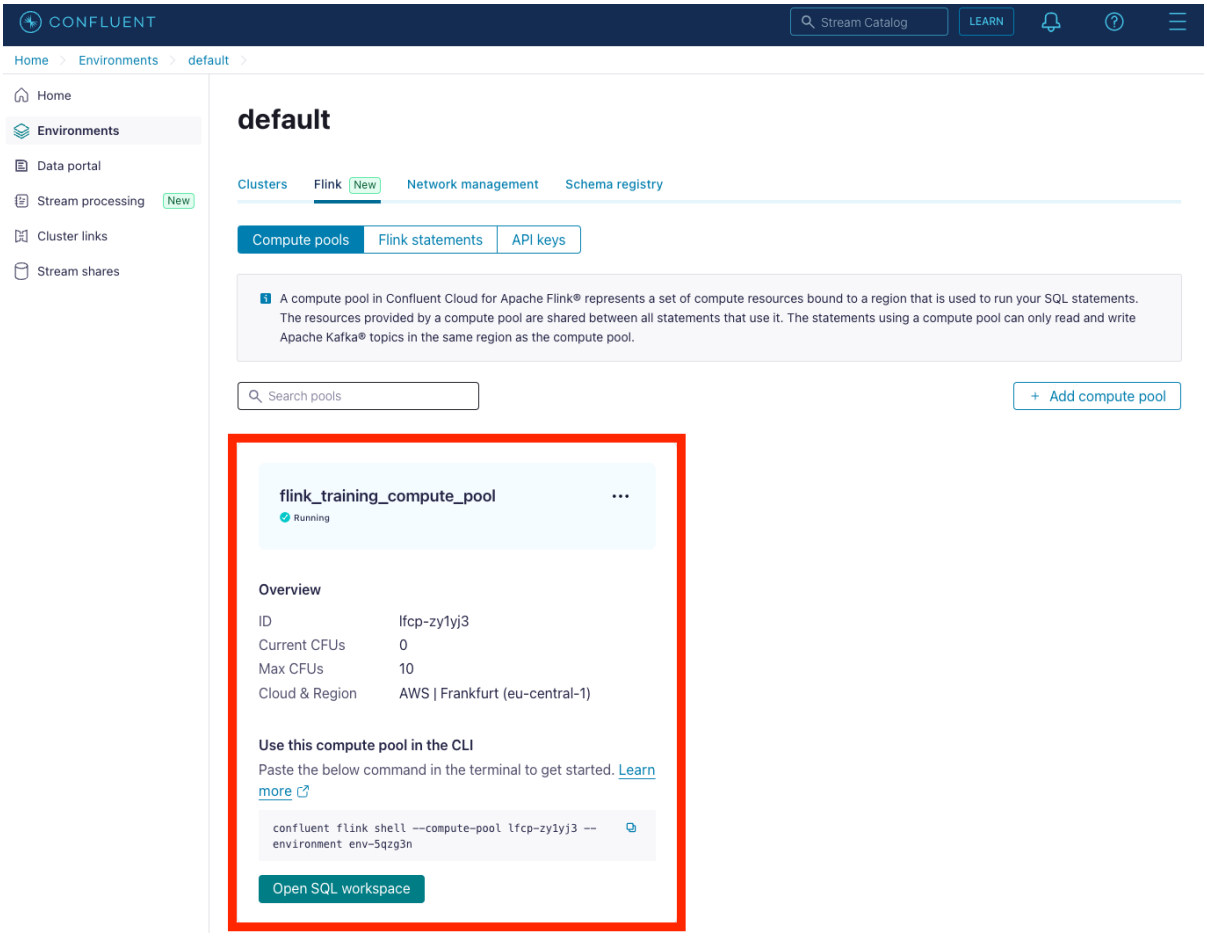
The pool will be provisioned and ready to use in a few moments.

21. Click on the **flink_training_compute_pool** box to view more details like:

    a. Number of CFUs that are currently being used (remember Confluent Flink scales up and down the CFUs depending on the workload)

    b. Number of messages in and out per minute

    c. Statements associated with this pool



Running your first Flink SQL query

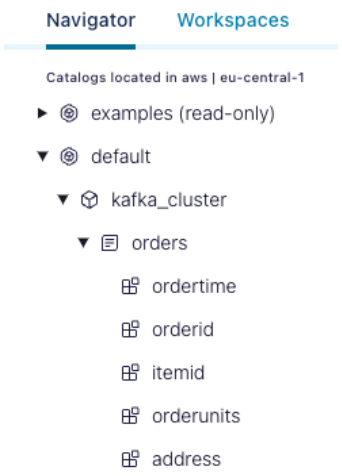
22. Now, click on the green button **Open SQL workspace**.

23. In the left pane, you can navigate through the available catalogs and databases for this compute pool.

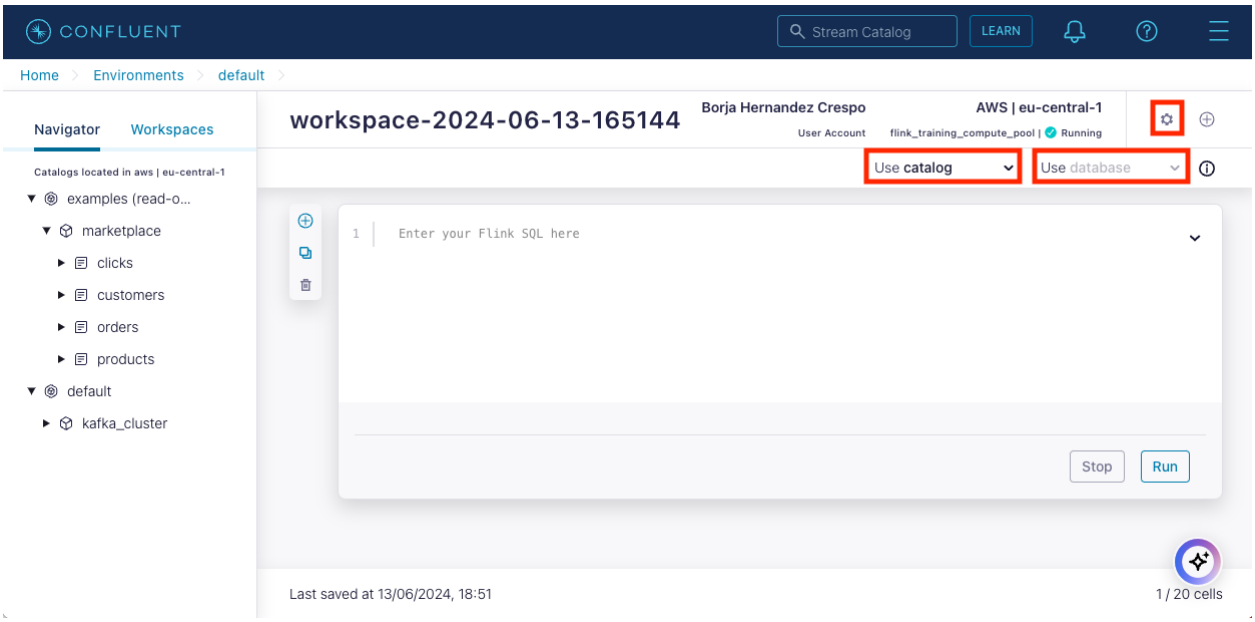
    As you can see, there are two catalogs:

    a. a built-in read-only ***examples*** catalog with one database ***marketplace*** and four tables, which can be used to start learning with Flink if you don't have any data

b. the ***default*** catalog (environment) with the database ***kafka_cluster*** and one table ***orders*** (topic). You can also expand the table ***orders*** to view the different columns



24. Click on the **Gear icon** in the top-right corner to open the settings window. Then, rename the workspace to "**lab2-workspace**" and click **Save changes**.

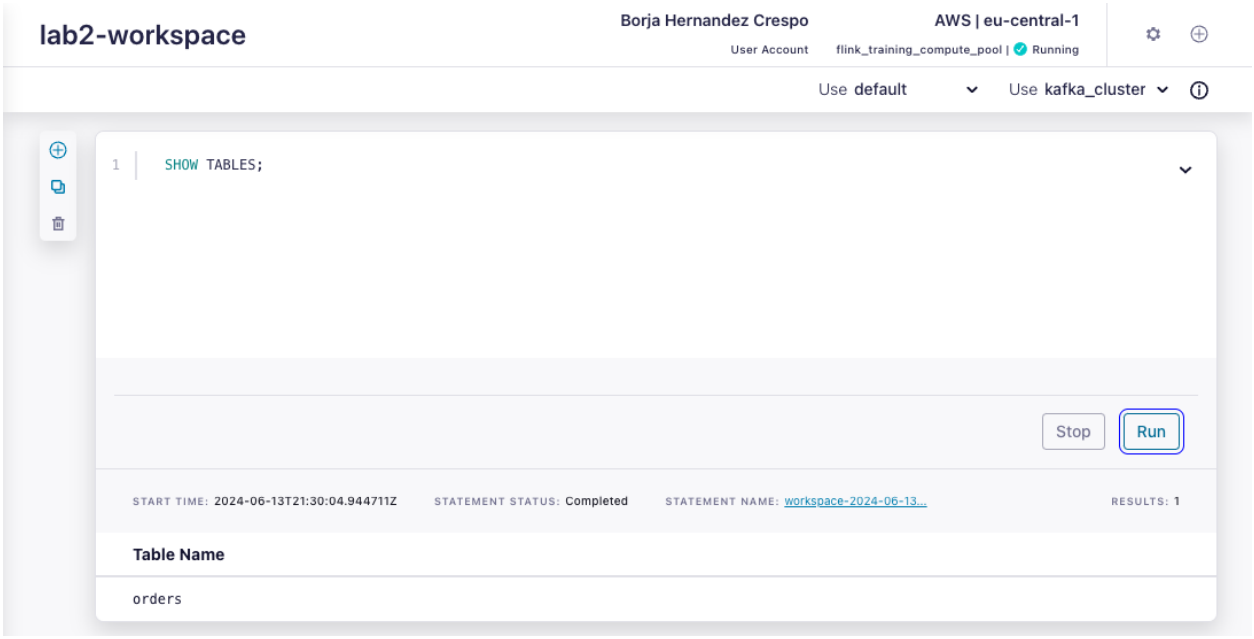


25. Select the catalog called **default** and the database called **kafka-cluster**.

    By default, this workspace will look for data in that database unless instructed otherwise.

26. Go to the central cell. Type the following command and hit **Run** to view all the tables in the **kafka-cluster** database:

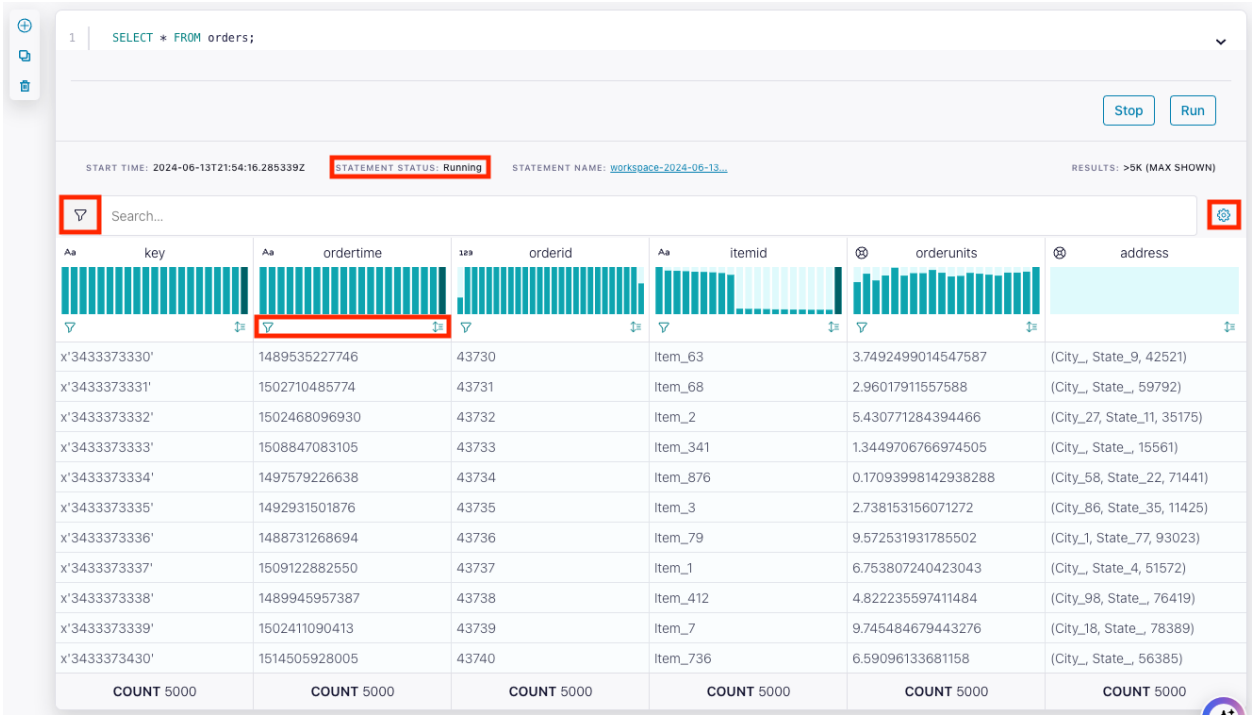
```
SHOW TABLES;
```



27. Now, add a new cell by clicking this icon ⊕

28. Run the following continuous query:

```
SELECT * FROM orders;
```



Note that the statement status is **Running**, it is a continuous query and will be running indefinitely unless you stop it.

Summary Statistics are provided to understand your data. Optionally, you can disable

the statistics by clicking on the **Gear icon**.

29. You can also **filter** and **sort** the results. Feel free to play around with these options.
30. Leave the statement running and add a new cell by clicking the **Plus icon**. Run the following command:

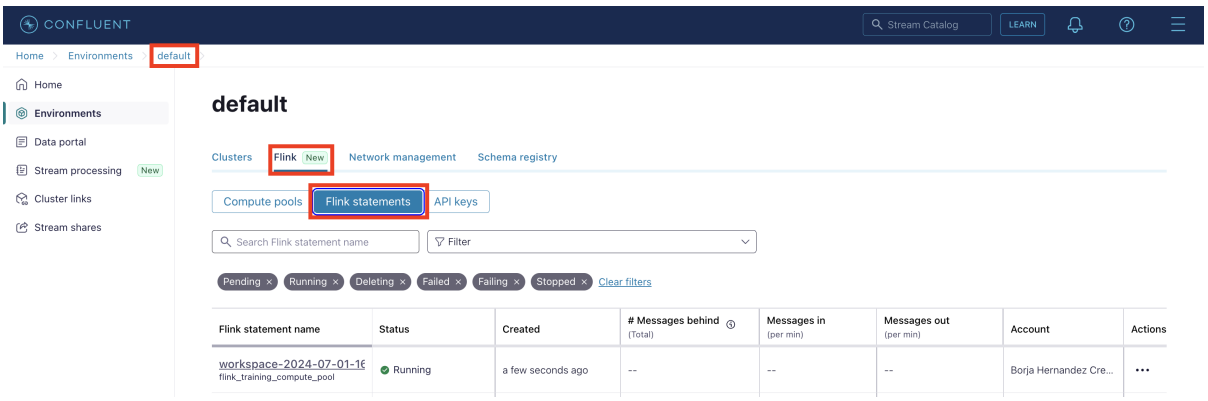
```
DESCRIBE orders;
```

Column Name	Data Type	Nullable	Extras
key	BYTES	NULL	BUCKET KEY
ordertime	BIGINT	NOT NULL	*Empty*
orderid	INT	NOT NULL	*Empty*
itemid	STRING	NOT NULL	*Empty*
orderunits	DOUBLE	NOT NULL	*Empty*
address	ROW<`city` STRING NOT NULL, `state` STRING NOT NULL, `zipcode` BIGINT NOT NULL>	NOT NULL	*Empty*

DESCRIBE shows table properties like:

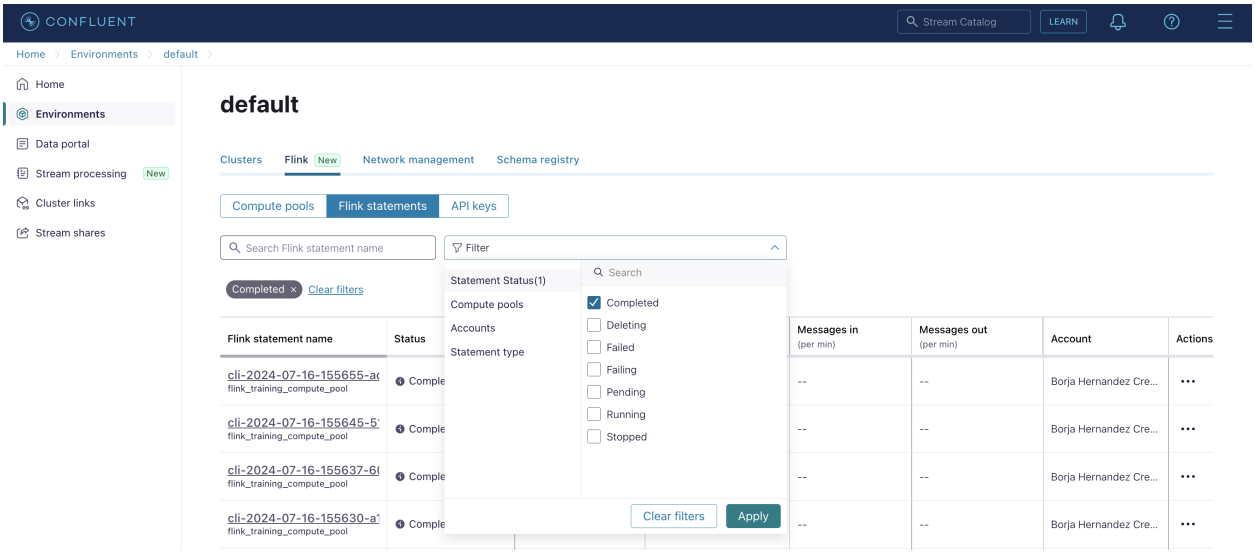
- Column names, data types and nullable columns
- Bucket keys (keys of distribution, how data is partitioned in the Kafka topic)
- Primary keys (if any)
- Custom watermarks (if any)

31. Let's now check the metrics of the running statement.
    a. Go to the environment **default** page
    b. Select the **Flink** tab
    c. Click on **Flink statements**



32. If you click on the statement name "**workspace-2024-06-13-1...**", a new window will open with more details about the statement.

33. You can also view Completed Statements by clicking on the "**Filter**" box, checking the Completed option and clicking **Apply**.



34. From here, you can **STOP** or **DELETE** any statement. Let's stop the running statement by clicking on the three dots under **Actions**, then click on **Stop statement** and **Confirm**.

    The statement status will transition to **Stopped**.

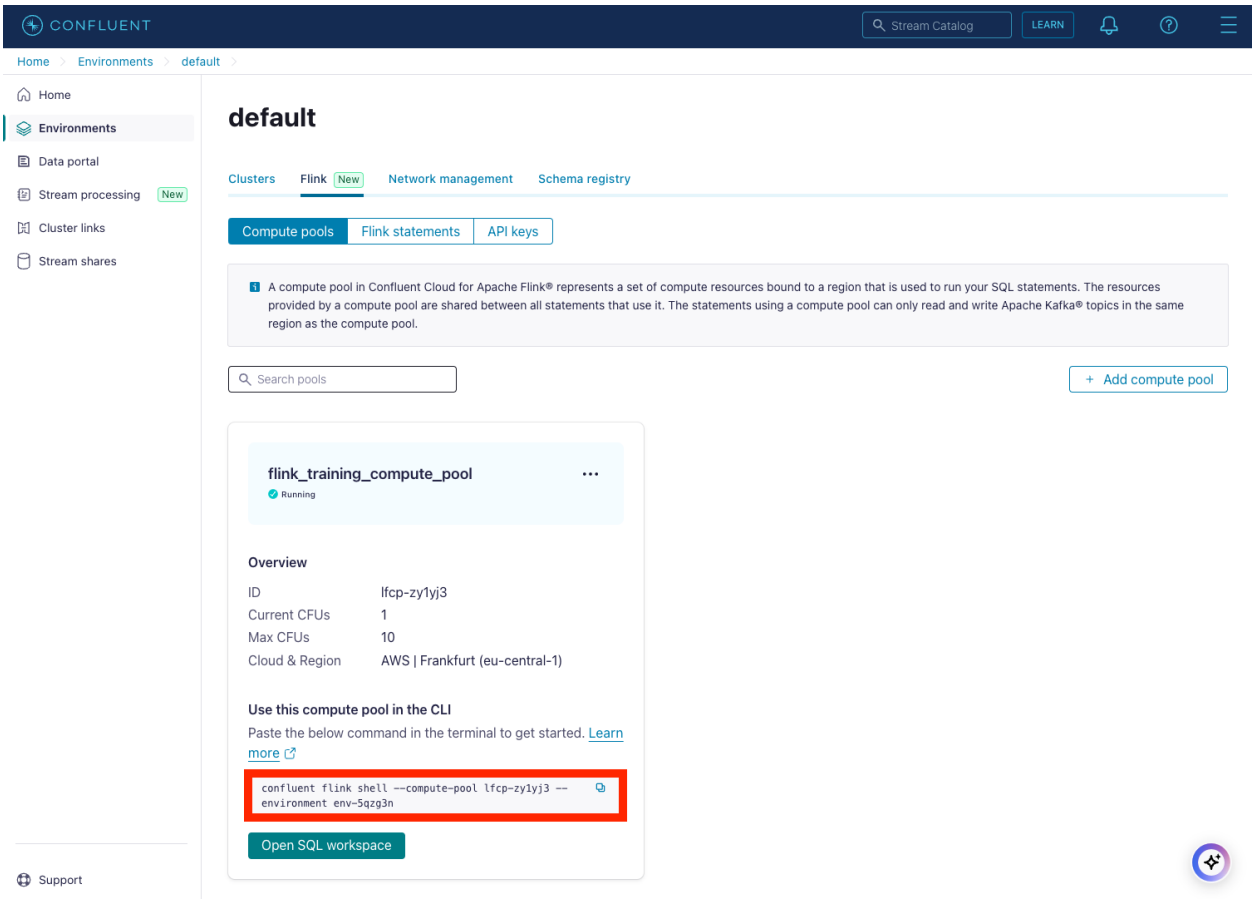
Using the Confluent CLI to run Flink SQL queries

35. Go to the Virtual Machine, open a new terminal window, and run the following command to log into your Confluent Cloud account:

> Optionally, you can use `confluent login --save` to avoid session timeouts.

```
$ confluent login
Enter your Confluent Cloud credentials:
Email: <enter email>
Password: <enter password>
```

36. In the Confluent Cloud UI, go to the **Flink** tab under the **default** environment and copy the command highlighted in the screenshot below:



37. Come back to the terminal, paste and run the command.

```
$ confluent flink shell --compute-pool <lfcp-xxxxxx> --environment
<env-xxxxxx>
Welcome!
To exit, press Ctrl-Q or type "exit".

[Ctrl-Q] Quit [Ctrl-S] Toggle Smart Completion
>
```

38. First, you need to define the default catalog and database. Run the following to view the available catalogs (environments);

```
> SHOW CATALOGS;

Statement name: cli-2024-06-14-110404-f0657d37-149f-4d18-82b7-
d21a207bcecb
Statement successfully submitted.
Waiting for statement to be ready. Statement phase is PENDING.
Statement phase is COMPLETED.
+--------------+--------------+
| Catalog Name |  Catalog ID  |
+--------------+--------------+
| examples     | cat-examples |
| default      | env-5qzg3n   |
+--------------+--------------+
```

39. Select the catalog **default** by running this command:

```
> USE CATALOG default;

Statement successfully submitted.
Statement phase is COMPLETED.
configuration updated successfully.
+---------------------+---------+
|         Key         |  Value  |
+---------------------+---------+
| sql.current-catalog | default |
+---------------------+---------+
```

40. Let's do the same to choose the database. Run:

```
> SHOW DATABASES;

Statement name: cli-2024-06-14-110839-556d8ac4-ada7-4964-bcde-
fdf055e1bb24
Statement successfully submitted.
Waiting for statement to be ready. Statement phase is PENDING.
Statement phase is COMPLETED.
+---------------+-------------+
| Database Name | Database ID |
+---------------+-------------+
| kafka_cluster | lkc-7o8k9j  |
+---------------+-------------+
```

```
> USE kafka_cluster;

Statement successfully submitted.
Statement phase is COMPLETED.
configuration updated successfully.
+----------------------+---------------+
|         Key          |     Value     |
+----------------------+---------------+
| sql.current-database | kafka_cluster |
+----------------------+---------------+
```

41. Now, let's run a continuous query to count the number of orders per item. Run:

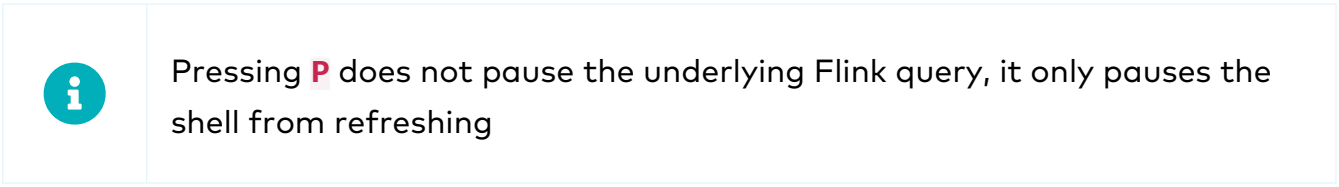
```
> SELECT itemid, COUNT(*) AS `count` FROM orders GROUP BY itemid;

╒═ Table mode (cli-2024-06-14-112441-39e38bbb-60c3-44d7-8fa2-
68a65dc7109a) ═╕
│itemid   count │
│               │
│Item_453 63    │
│               │
│Item_734 64    │
│               │
│Item_663 65    │
│               │
│Item_233 65    │
│               │
│Item_178 44    │
│               │
│Item_698 68    │
│               │
│Item_388 65    │
│               │
│Item_310 53    │
│               │
│Item_281 66    │
│               │
│Item_203 68    │
│               │
│Item_107 69    │
│               │
│Item_183 70    │
│               │
│Item_777 57    │
│               │
│Item_306 62    │
│               │
│Item_298 74    │
│               │
│Item_780 64    │
│               │
│Item_914 61    │
│               │
│Item_514 72    │
│               │
╘═══════════════╛
Refresh: Running                                    Last refresh:
11:21:15.765
[Q] Quit  [M] Show changelog  [P] Pause  [U/D] Jump up/down
```

42. In this interactive shell, you can:

    a. Press `M` to switch to changelog mode to view all the different updates in the table (we will cover this in the next module)

b. Press **P** to pause the shell from refreshing



c. Press **UP** and **DOWN** arrows to move the cursor between rows. Press **ENTER** to view all columns of the selected row more clearly

d. Press **U** or **D** to jump quickly up and down between rows

e. Press **Q** to quit the shell and implicitly stop the query

43. Press **Q** to quit the shell.

44. Let's create a new table running this command:

```
> CREATE TABLE orders_item_1 (
  `orderid` INT,
  `itemid` STRING,
  `orderunits` DOUBLE
);

Statement name: cli-2024-06-14-120146-2fa1d690-82d1-4c8c-875e-
ba60f27f2f89
Statement successfully submitted.
Waiting for statement to be ready. Statement phase is PENDING.
Statement phase is RUNNING.
Listening for execution errors. Press Enter to detach.
Statement phase is COMPLETED.
Table 'orders_item_1' created.
```

45. Now, insert some messages into this new table using data from the orders table, filtering specifically for orders of **Item_1**.

```
> INSERT INTO orders_item_1
SELECT orderid, itemid, orderunits
FROM orders
WHERE itemid = 'Item_1';

Statement name: cli-2024-06-14-120742-b08a4ed4-cb7d-40a5-91d3-
d3fb61189068
Statement successfully submitted.
Waiting for statement to be ready. Statement phase is PENDING.
Waiting for statement to be ready. Statement phase is PENDING.
(Timeout 6s/600s)
Waiting for statement to be ready. Statement phase is PENDING.
(Timeout 13s/600s)
Statement phase is RUNNING.
Listening for execution errors. Press Enter to detach.
```

46. Hit **ENTER** to detach, meaning to return to the shell without terminating the underlying query.

47. Check that the previous statement is still running.

    a. First, exit the Confluent Flink shell by pressing **Ctrl + D**

    b. Run the following command to set the cloud provider and region of your compute pool as default:

    Replace **cloud** and **region** with the corresponding values of your compute pool.

    ```
    $ confluent flink region use --cloud <aws|azure|gcp> --region
    <region>
    ```

    Example

    ```
    $ confluent flink region use --cloud aws --region eu-central-1
    Using Flink region "Frankfurt (eu-central-1)".
    ```

    c. Run the following command to list all the **RUNNING** statements:

```
$ confluent flink statement list --status RUNNING

          Creation Date       |      Name     |           Statement
| Compute Pool | Status
------------------------------+---------------+
--------------------------------+--------------+--------
 2024-06-14 12:07:42.752039   | cli-2024-0... | INSERT INTO
orders_item_1       | lfcp-zy1yj3  | RUNNING
 +0000 UTC                    |               | SELECT orderid,
itemid,         |              |
                              |               | orderunits FROM
orders WHERE    |              |
                              |               | itemid = 'Item_1';
|               |
```

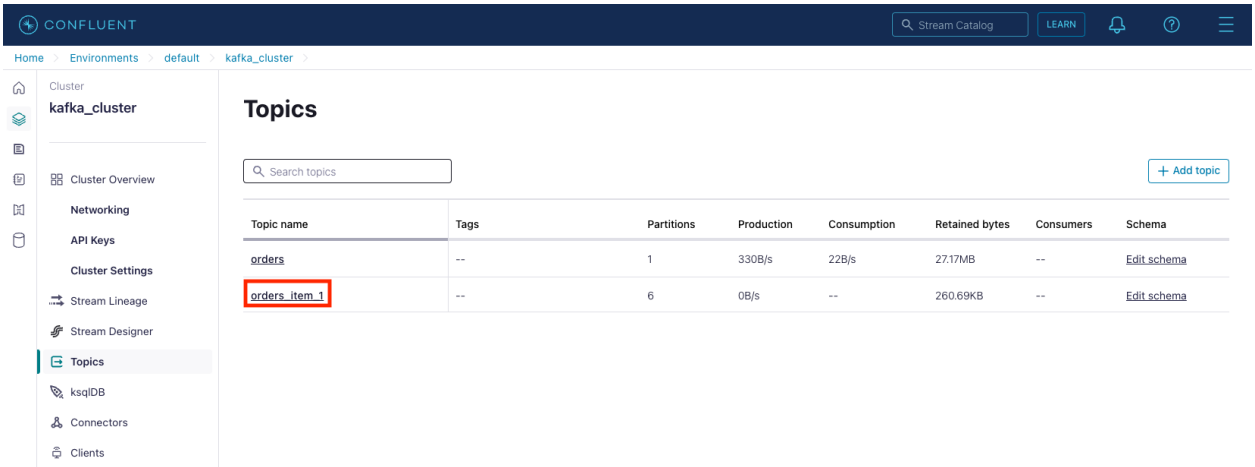
48. Let's stop the **RUNNING** statement using this command:

```
$ confluent flink statement stop <statement-name>
```

Example

```
$ confluent flink statement stop cli-2024-06-14-120742-b08a4ed4-
cb7d-40a5-91d3-d3fb61189068
Requested to stop Flink SQL statement "cli-2024-06-14-120742-
b08a4ed4-cb7d-40a5-91d3-d3fb61189068".
```

49. Finally, go to the Confluent Cloud Web UI to verify that creating the new Flink table **orders_item_1** has also created the Kafka topic **orders_item_1**.



Conclusion

In this lab, you created a topic and a DataGen connector to generate synthetic data in Confluent Cloud. Then, you learned how to set up a Flink compute pool and gained an understanding of how a Flink workspace is organized. Now, you are familiar with using both the Web UI and the Confluent CLI to execute Flink queries and manage statements effectively.

STOP HERE. THIS IS THE END OF THE EXERCISE.

Lab 11 Working with Dynamic Tables

a. Working with Dynamic Tables

The goal of this lab is to familiarize yourself with dynamic tables. You will learn how dynamic tables are created from Kafka topics, alter existing tables, and understand the differences between upsert and retract tables. Additionally, you will gain experience in manipulating system and metadata columns.

Deploying the Confluent Cloud Environment using Terraform

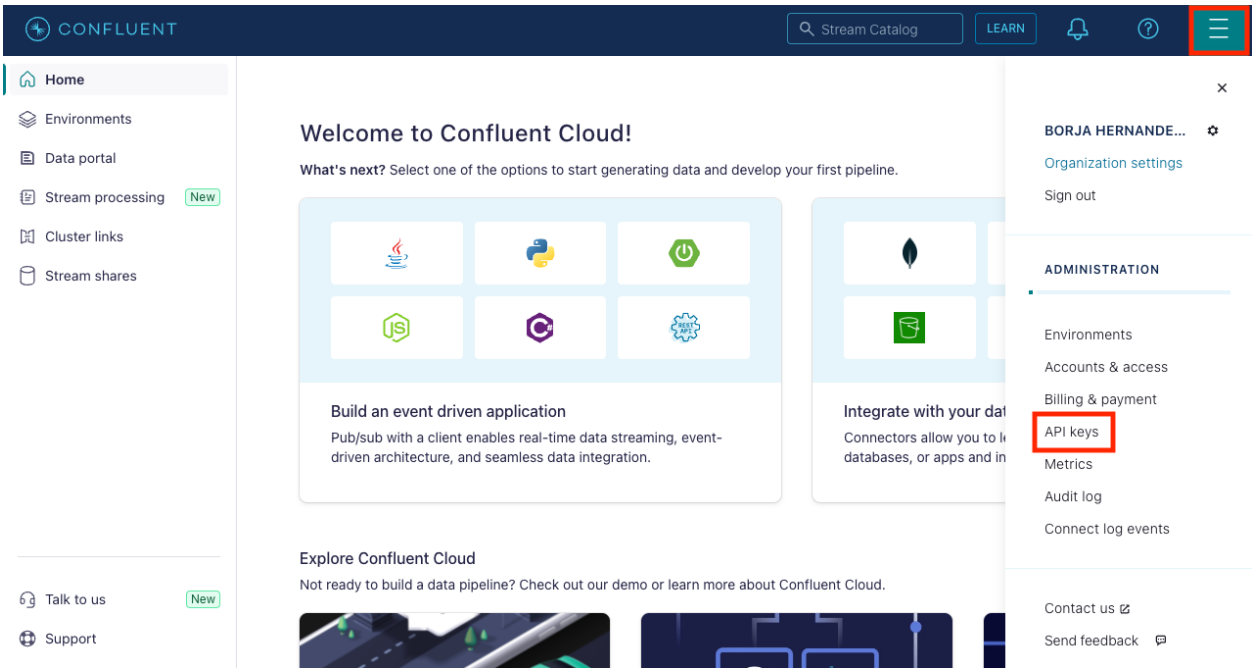
Terraform is an infrastructure as code tool, allowing users to define and provision infrastructure through code. In combination with the Confluent Terraform Provider, it enables the automated management of Confluent Cloud resources like Kafka clusters, topics, or Flink compute pools.

In this lab, you will automatically create a Confluent Cloud environment, a Kafka cluster, a Schema Registry, a Datagen connector, a Kafka topic, and a Flink Compute Pool using Terraform.

Creating a Cloud API Key

First, you need to create a Cloud API key to provide Terraform with access to manage resources in Confluent Cloud.

1. Go to Confluent Cloud website and log in to your Confluent Cloud account.
2. Click on the hamburger menu in the upper-right corner, then select **API keys**.



3. Click on **Add API key**. Then use the following instructions:

   a. Select **My account** as the owner of the new API key and click **Next**.

   > Note that **My Account** is only recommended for development environments like this training. In production, you should use **Service Accounts** as the owner of API keys and provide only the minimal required permissions.

   b. Select **Cloud resource management**, then click **Next**.

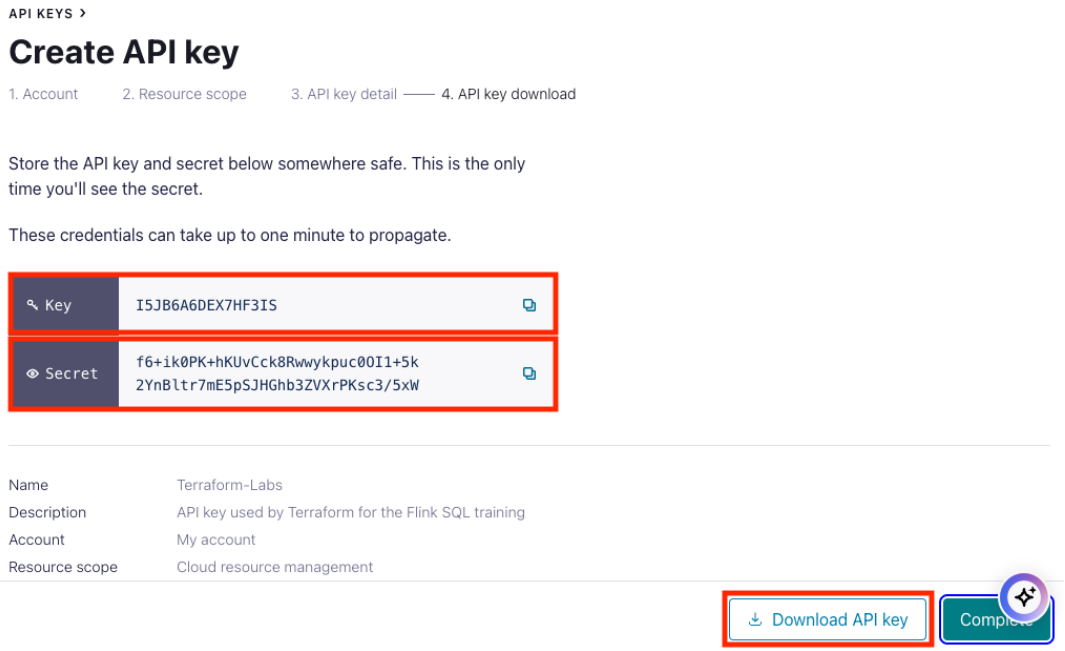
   c. Add a name to the new API key to identify it from others.

      - Name: **Terraform-Labs**
      - Description: **API key used by Terraform for the Flink SQL training**

   d. Click on **Create API key**.

   e. Click on **Download API key** to save it.

   > If you are on the VM, copy the **KEY** and **SECRET** externally to the VM. This way, if you need to destroy the VM in the future, you can retain the API key.



f. Finally, click on **Complete**.

Running Terraform

4. Now, go to the VM and open a Terminal window.
5. In the Terminal, change directory to `~/confluent-flink/terraform/lab-tables`. Run:

```
$ cd ~/confluent-flink/terraform/lab-tables
```

6. Open the file `terraform.tfvars` in VS Code. Run:

```
$ code terraform.tfvars
```

7. Replace the API key and the API secret with your values, ensuring they are enclosed in quotes (" ").
8. Press `Ctrl + S` to save the changes.
9. Return to the Terminal and initialize Terraform by running `terraform init`. Ensure you are in the `~/confluent-flink/terraform/lab-tables` directory:

```
$ terraform init
```

10. Create the Terraform execution plan by running:

```
$ terraform plan
```

11. Finally, run `terraform apply` to apply the execution plan and create all the Confluent Cloud resources:

```
$ terraform apply
```

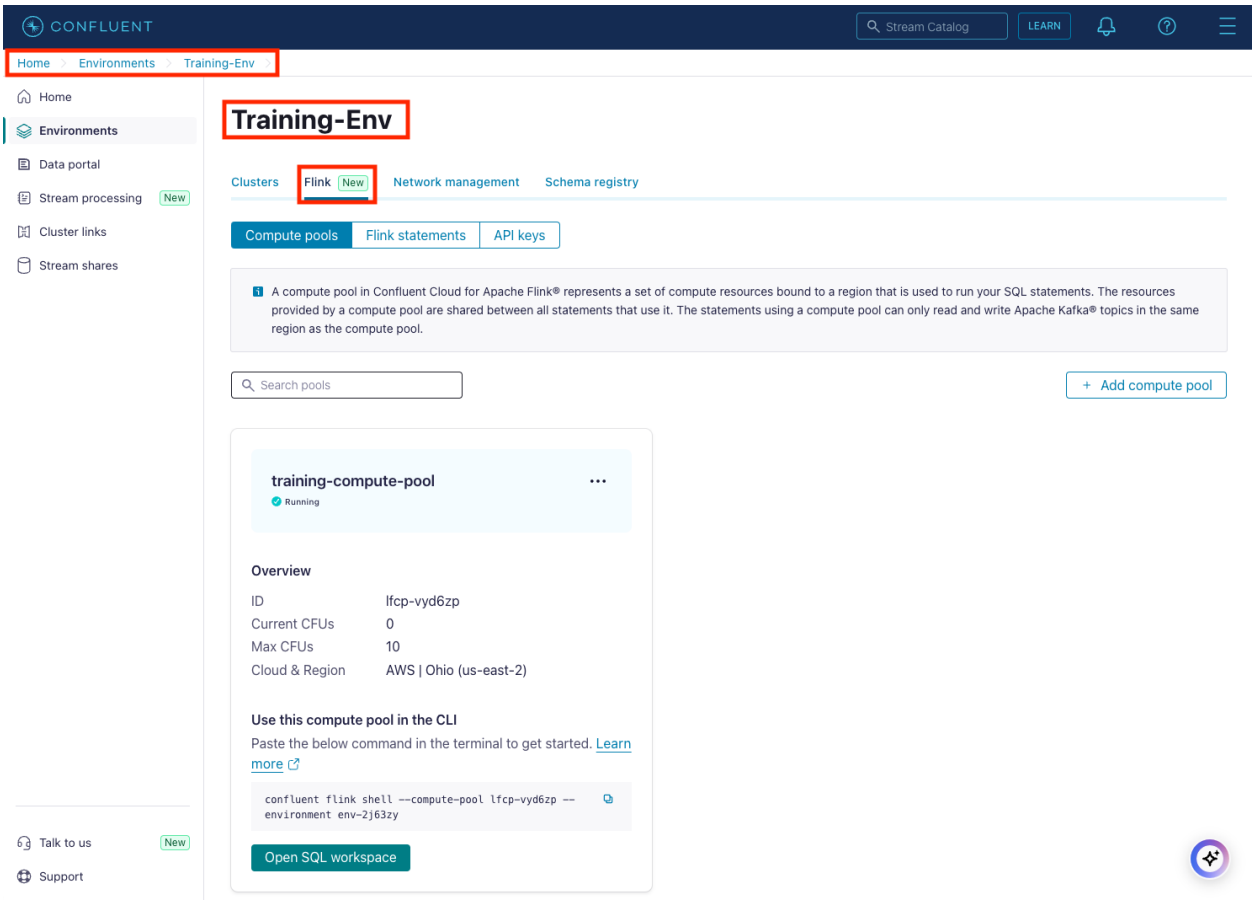
    Type `yes` to confirm.

Wait a few minutes for the new Confluent Cloud resources to become available:

- Confluent Cloud Environment: **Training-Env**
- Kafka Cluster: **training-kafka**
- Datagen Connector: **Orders_DatagenSourceConnector**
- Kafka Topic: **orders**
- Flink Compute Pool: **training-compute-pool**

Investigating the Dynamic Table `orders`

12. Go to Confluent Cloud Web UI and navigate to the **Flink** tab under the **Training-Env** environment.



13. Click on **Open SQL workspace**.

14. Configure the **Use catalog** and **Use database** as **Training-Env** and **training-kafka** respectively.

15. As you can see in the Navigator tree on the left side, **training-kafka** database only has one table, **orders**.

    This table **orders** was automatically created in Flink from the Kafka topic **orders**.

    Let's find out how this automatic table is defined. Run this Flink query:

```
> SHOW CREATE TABLE orders;

CREATE TABLE `Training-Env`.`training-kafka`.`orders` (
  `key` VARBINARY(2147483647),
  `ordertime` BIGINT NOT NULL,
  `orderid` INT NOT NULL,
  `itemid` VARCHAR(2147483647) NOT NULL,
  `orderunits` DOUBLE NOT NULL,
  `address` ROW<`city` VARCHAR(2147483647) NOT NULL, `state`
VARCHAR(2147483647) NOT NULL, `zipcode` BIGINT NOT NULL> NOT NULL
) DISTRIBUTED BY HASH(`key`) INTO 6 BUCKETS
WITH (
  'changelog.mode' = 'append',
  'connector' = 'confluent',
  'kafka.cleanup-policy' = 'delete',
  'kafka.max-message-size' = '2097164 bytes',
  'kafka.retention.size' = '0 bytes',
  'kafka.retention.time' = '7 d',
  'key.format' = 'raw',
  'scan.bounded.mode' = 'unbounded',
  'scan.startup.mode' = 'earliest-offset',
  'value.format' = 'avro-registry'
)
```

Output explanation:

- Tables can also be referenced using the full path Catalog.Database.Table (**`Training-Env`.`training-kafka`.`orders`**)
- The **key** of the messages is included in the table as a **`VARBINARY`** field (bytes). This is because there is no schema defined for the message key in the **orders** topic, so **`key.format`** is also defined as **`raw`**
- The data is distributed into 6 buckets (topic partitions) by hashing the **key** column. Remember that topic **orders** has 6 partitions and messages are partitioned based on the **key** of the message.
- **changelog.mode = 'append'**: The table operates in append-only mode.
- **connector = 'confluent'**: Uses the Confluent Kafka connector.
- **kafka.cleanup-policy = 'delete'**: Kafka messages are deleted based on retention.time and retention.size.
- **kafka.max-message-size = '2097164 bytes'**: Sets the maximum Kafka message size.
- **kafka.retention.size = '0 bytes'**: Kafka messages are not retained based on size (disabled).

- **kafka.retention.time = '7 d'**: Kafka messages are retained for 7 days.
- **key.format = 'raw'**: The key format is raw bytes.
- **scan.bounded.mode = 'unbounded'**: The table scans in an unbounded mode.
- **scan.startup.mode = 'earliest-offset'**: The table starts scanning from the earliest Kafka offset.
- **value.format = 'avro-registry'**: The value format is Avro, registered in the schema registry.

16. Let's take a look at how the data in the **orders** table looks. Run:

```
> SELECT * FROM orders;

key           ordertime     orderid itemid   orderunits
address
x'3133323235' 1506337744037 13225   Item_8   4.6354289634035615
(City_2, State_, 20513)
x'3133323237' 1488669690911 13227   Item_679 0.42492348246814593
(City_83, State_5, 46071)
x'3133323238' 1503406631332 13228   Item_8   6.713408415981801
(City_, State_9, 98245)
x'3133323239' 1490040017933 13229   Item_98  7.994281176569055
(City_, State_76, 80397)
x'3133323330' 1495205523397 13230   Item_97  9.791834679872236
(City_21, State_98, 77675)
x'3133323331' 1506243263409 13231   Item_376 7.535737416952441
(City_54, State_63, 93803)]
x'3133323332' 1509834781412 13232   Item_67  6.159164567778619
(City_69, State_96, 45565)
```

17. Imagine you need to process the **key** in a subsequent query, but it needs to be in `STRING` format instead of `VARBINARY`. Use this query to modify the data type:

```
> ALTER TABLE orders MODIFY (`key` STRING);
```

18. You also need to access the **headers** of the messages to process the data in subsequent queries. Run the following query to add a new column to the **orders** table:

```
> ALTER TABLE orders ADD (`headers` MAP<STRING, STRING> METADATA);
```

19. There is a final requirement for the **orders** table, which is to start consuming data from the latest Kafka offset of the **orders** topic. Run the following query to meet this

requirement:

```
> ALTER TABLE orders SET ('scan.startup.mode' = 'latest-offset');
```

20. Check how the table is currently defined:

```
> SHOW CREATE TABLE orders;

CREATE TABLE `Training-Env`.`training-kafka`.`orders` (
  `key` VARCHAR(2147483647),
  `ordertime` BIGINT NOT NULL,
  `orderid` INT NOT NULL,
  `itemid` VARCHAR(2147483647) NOT NULL,
  `orderunits` DOUBLE NOT NULL,
  `address` ROW<`city` VARCHAR(2147483647) NOT NULL, `state`
VARCHAR(2147483647) NOT NULL, `zipcode` BIGINT NOT NULL> NOT NULL
  `headers` MAP<VARCHAR(2147483647), VARCHAR(2147483647)> METADATA
) DISTRIBUTED BY HASH(`key`) INTO 6 BUCKETS
WITH (
  'changelog.mode' = 'append',
  'connector' = 'confluent',
  'kafka.cleanup-policy' = 'delete',
  'kafka.max-message-size' = '2097164 bytes',
  'kafka.retention.size' = '0 bytes',
  'kafka.retention.time' = '7 d',
  'key.format' = 'raw',
  'scan.bounded.mode' = 'unbounded',
  'scan.startup.mode' = 'latest-offset',
  'value.format' = 'avro-registry'
)
```

21. Check how the table data looks now by running:

```
> SELECT * FROM orders;

key   ordertime     orderid  ...   headers
15111 1500932570897 15111    ...   {task.id=0, task.generation=0,
current.iteration=15111}
15112 1518086690731 15112    ...   {task.id=0, task.generation=0,
current.iteration=15112}
15113 1511040924678 15113    ...   {task.id=0, task.generation=0,
current.iteration=15113}
15114 1499760925800 15114    ...   {task.id=0, task.generation=0,
current.iteration=15114}
15115 1488586078114 15115    ...   {task.id=0, task.generation=0,
current.iteration=15115}
15116 1518333934811 15116    ...   {task.id=0, task.generation=0,
current.iteration=15116}
15117 1497372872880 15117    ...   {task.id=0, task.generation=0,
current.iteration=15117}
```

Comparing UPSERT vs. RETRACT

Now, you are going to create two identical tables, but the **changelog.mode** will be **upsert** for one and **retract** for the other.

22. Run the following query to create the UPSERT table:

```
> CREATE TABLE items_sold_upsert (
    itemid STRING,
    total DOUBLE,
    PRIMARY KEY (itemid) NOT ENFORCED
  ) DISTRIBUTED INTO 1 BUCKETS
  WITH (
    'changelog.mode' = 'upsert'
  );
```

Note that the only way to create an UPSERT table is by defining a **PRIMARY KEY**.

If **PRIMARY KEY** is defined, automatically the table will be **DISTRIBUTED BY** the primary key column(s).

```
> SHOW CREATE TABLE items_sold_upsert;

CREATE TABLE `Training-Env`.`training-kafka`.`items_sold_upsert` (
  `itemid` VARCHAR(2147483647) NOT NULL,
  `total` DOUBLE,
  CONSTRAINT `PK_itemid` PRIMARY KEY (`itemid`) NOT ENFORCED
) DISTRIBUTED BY HASH(`itemid`) INTO 1 BUCKETS
WITH (
  'changelog.mode' = 'upsert',
  'connector' = 'confluent',
  'kafka.cleanup-policy' = 'delete',
  'kafka.max-message-size' = '2097164 bytes',
  'kafka.retention.size' = '0 bytes',
  'kafka.retention.time' = '7 d',
  'key.format' = 'avro-registry',
  'scan.bounded.mode' = 'unbounded',
  'scan.startup.mode' = 'earliest-offset',
  'value.format' = 'avro-registry'
)
```

23. For the RETRACT case, there are three ways to define the table:

    a. No **PRIMARY KEY** and default **DISTRIBUTED BY**:

```
> CREATE TABLE items_sold_retract_1 (
    itemid STRING,
    total DOUBLE
  ) DISTRIBUTED INTO 1 BUCKETS
  WITH (
    'changelog.mode' = 'retract'
  );
```

In this case, the distribution is by **HASH(entire value)**.

    b. No **PRIMARY KEY** and custom **DISTRIBUTED BY**:

```
> CREATE TABLE items_sold_retract_2 (
    itemid STRING,
    total DOUBLE
  ) DISTRIBUTED BY (itemid) INTO 1 BUCKETS
   WITH (
    'changelog.mode' = 'retract'
  );
```

In this case, the distribution is by `HASH(specified column(s))`.

c. `PRIMARY KEY` and default `DISTRIBUTED BY`:

```
> CREATE TABLE items_sold_retract_3 (
    itemid STRING,
    total DOUBLE,
    PRIMARY KEY(itemid) NOT ENFORCED
  ) DISTRIBUTED INTO 1 BUCKETS
  WITH (
    'changelog.mode' = 'retract'
  );
```

In this case, the distribution is by `HASH(primary key)`.

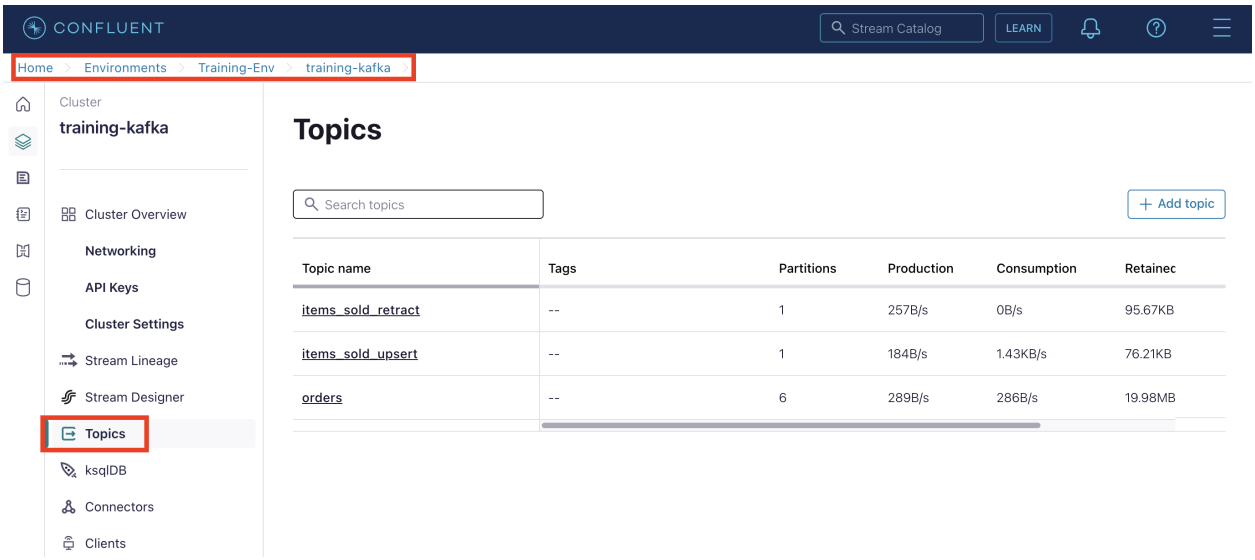
24. Let's use the second scenario (No `PRIMARY KEY` and custom `DISTRIBUTED BY`). Run this query:

```
> CREATE TABLE items_sold_retract (
  itemid STRING,
  total DOUBLE
) DISTRIBUTED BY (itemid) INTO 1 BUCKETS
 WITH (
  'changelog.mode' = 'retract'
);
```

25. Run the following query to insert data to both tables:

```
> EXECUTE STATEMENT SET
BEGIN
  INSERT INTO items_sold_upsert SELECT itemid, SUM(orderunits) AS
total
     FROM orders GROUP BY itemid;
  INSERT INTO items_sold_retract SELECT itemid, SUM(orderunits) AS
total
     FROM orders GROUP BY itemid;
END;
```

26. Go to Confluent Cloud web UI and navigate to the page **Topics** under the Kafka cluster **training-kafka**.



27. Both tables are receiving the same amount of data. However, if you check the metric `Retained bytes`, you will see that the amount of bytes for the **retract** table is greater than for the **upsert** table.

    This is because, when there is an update for the **retract** table, two messages are produced in the `items_sold_retract` topic (one for `UPDATE BEFORE` and another for `UPDATE AFTER`).

    On the other hand, for the **upsert** table, only one message is produced when there is an update (`UPDATE AFTER`), which reduces the amount of data produced in the topic.

28. Click on the `items_sold_retract` topic, then select the tab **Messages**. You will see the messages that are currently produced into this topic.

29. If you look into the **Key** column, you should see two consecutive messages with the same `itemid`. This means that the `itemid` has been updated, with the first message representing `UPDATE BEFORE` and the second message representing `UPDATE AFTER`.

30. Click on a message, and then on the **headers** tab in the window that opens on the right side.

    You should see something like this:

```
"key": "op",
"value": "\u0001"
```

    or

```
"key": "op",
"value": "\u0002"
```

They represent the `operation` of the message `UPDATE BEFORE` and `UPDATE AFTER` respectively.

If it is an `INSERT`, the headers are empty.

31. Finally, verify that the schema for the **Key** of the messages for the `items_sold_retract` topic is nullable, as `itemid` is not a `PRIMARY KEY`.

    However, if you check the **Key** schema for the `items_sold_upsert` topic, you will see that the Key (`itemid`) is NOT nullable and can only be `STRING`.

RETRACT

```
{
  "fields": [
    {
      "default": null,
      "name": "itemid",
      "type": [
        "null",
        "string"
      ]
    }
  ],
  "name": "record",
  "namespace": "org.apache.flink.avro.generated",
  "type": "record"
}
```

UPSERT

```
{
  "fields": [
    {
      "name": "itemid",
      "type": "string"
    }
  ],
  "name": "record",
  "namespace": "org.apache.flink.avro.generated",
  "type": "record"
}
```

CHALLENGE

Create a new table

Create a new table called `orders_new` with the following columns and types:

- **itemid** `STRING`

- **orderid** `INT`
- **ordertime** `BIGINT`
- **orderunits** `DOUBLE`
- **msgtime** `TIMESTAMP(3)`
- **headers** `MAP<STRING, STRING>` (metadata column)

Also, the table has to be configured with:

- Primary Key: **itemid** and **orderid**
- Changelog mode: **upsert**

And the new table has to be related to a Kafka topic with the following configurations:

- Number of partitions: **3**
- Format of the Key schema: **JSON**
- Cleanup policy: **compact**

Write the `CREATE TABLE …` query and run it.

TRY IT YOURSELF FIRST. *If you get stuck, check the solution*

```
CREATE TABLE orders_new (
  itemid STRING,
  orderid INT,
  ordertime BIGINT,
  orderunits DOUBLE,
  msgtime TIMESTAMP(3),
  headers MAP<STRING, STRING> METADATA,
  PRIMARY KEY (itemid, orderid) NOT ENFORCED
) DISTRIBUTED INTO 3 BUCKETS
WITH (
  'changelog.mode' = 'upsert',
  'key.format' = 'json-registry',
  'kafka.cleanup-policy' = 'compact'
);
```

Insert data to the table

Create a query to insert data from the table `orders` into the table `orders_new` so that a row looks like this:

```
itemid:
Item_3

orderid:
256571

ordertime:
1510293656832

orderunits:
6.602346977037862

msgtime:
2024-06-21 13:51:05.082

headers:
{task.generation=4, task.id=0, current.iteration=256571, zipcode=84582}
```

A couple of remarks about this table:

- The column `msgtime` is based on the System column `$rowtime`
- The column `headers` contains four keys in the Map object (`task.generation`, `task.id`, `current.iteration`, `zipcode`):
  - The first three keys come from the headers of the data from table `orders`
  - The forth key is new and the zipcode value comes from the column `address` of table `orders` which contains the field `zipcode`

  Reminder of the `orders` table:

```
TABLE orders:
    key STRING
    ordertime BIGINT
    orderid INT
    itemid STRING
    orderunits DOUBLE
    address ROW<city STRING, state STRING, zipcode BIGINT>
    headers MAP<STRING, STRING> METADATA
```

Write the `INSERT INTO …` query and run it.

TRY IT YOURSELF FIRST. *If you get stuck, check the solution*

```
INSERT INTO orders_new SELECT
  itemid,
  orderid,
  ordertime,
  orderunits,
  $rowtime AS msgtime,
  MAP['task.generation', headers['task.generation'],
      'task.id', headers['task.id'],
      'current.iteration', headers['current.iteration'],
      'zipcode', CAST(address.zipcode AS STRING)]
    AS headers
FROM orders;
```

Clean up

- Go to the VM and open a Terminal window, then run:

```
$ cd ~/confluent-flink/terraform/lab-tables
```

- Run the following command to destroy all the Confluent Cloud resources created before by Terraform:

```
$ terraform destroy
```

Conclusion

In this lab, you set up Terraform to manage Confluent Cloud resources. You created an environment, a Kafka cluster, a topic, a connector, and a Flink compute pool using Terraform. Then, you explored how dynamic tables are automatically created from Kafka topics and how to alter existing tables. You also created two new tables (upsert and retract) and examined the differences between them. Finally, you completed a challenge by creating a new table and inserting data while manipulating the system column `$rowtime` and the metadata column `headers`.

STOP HERE. THIS IS THE END OF THE EXERCISE.

Lab 12 Using Watermarks and Windows

a. Using Watermarks and Windows

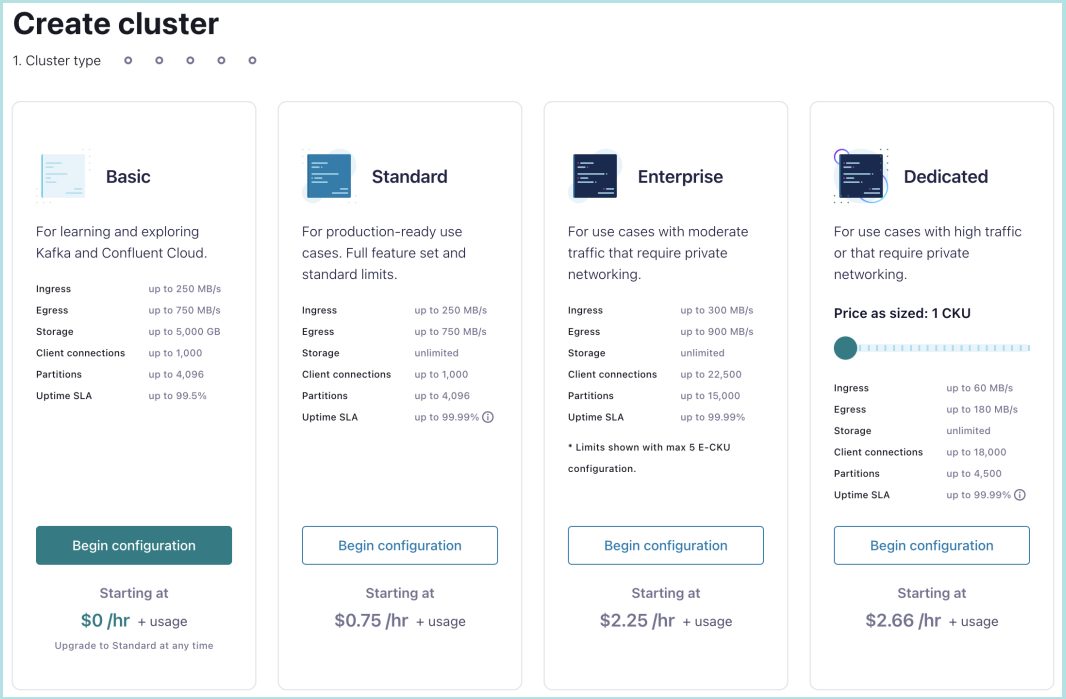
The goal of this lab is to familiarize yourself with timestamps and watermarks. You will learn how to view the default watermark and modify it to define a custom watermark. You will see the watermarks in action by inserting data into a table individually. Additionally, you will learn how to use various Time & Date functions. At the end, you will face a challenge to put everything you have learned into practice.

Creating a new Workspace

You should have the **default** environment, the **kafka_cluster** and the **flink_training_compute_pool** from the lab `Working with Flink in Confluent Cloud`.

If you don't have the **kafka_cluster** *and the* **flink_training_compute_pool**, *follow these instructions:*

- Go to the environment **default** page and click on "**Create your own cluster**" or "**Add cluster**".
- Choose a **Basic cluster** by clicking on **Begin configuration**.



- Choose a **cloud provider** and **region**.

Select one of these regions:

AWS

```
Oregon (us-west-2)
Ohio (us-east-2)
N. Virginia (us-east-1)
Frankfurt (eu-central-1)
Ireland (eu-west-1)
London (eu-west-2)
Mumbai (ap-south-1)
Singapore (ap-southeast-1)
Sydney (ap-southeast-2)
```

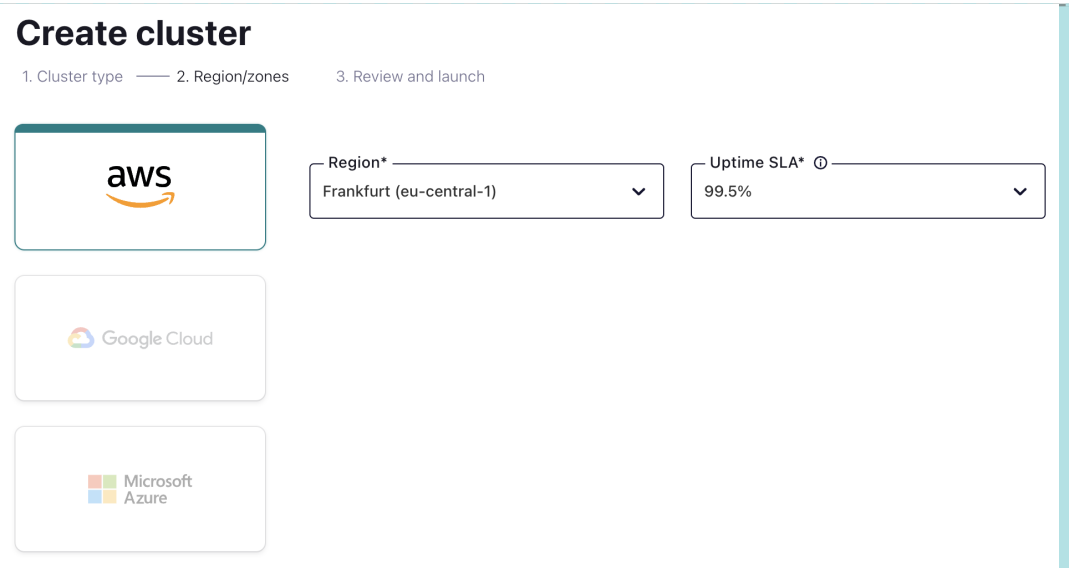
Google Cloud

```
Iowa (us-central1)
Las Vegas (us-west4)
N. Virginia (us-east4)
S. Carolina (us-east1)
Belgium (europe-west1)
Frankfurt (europe-west3)
Mumbai (asia-south1)
Delhi (asia-south2)
Singapore (asia-southeast1)
Sydney (australia-southeast1)
```

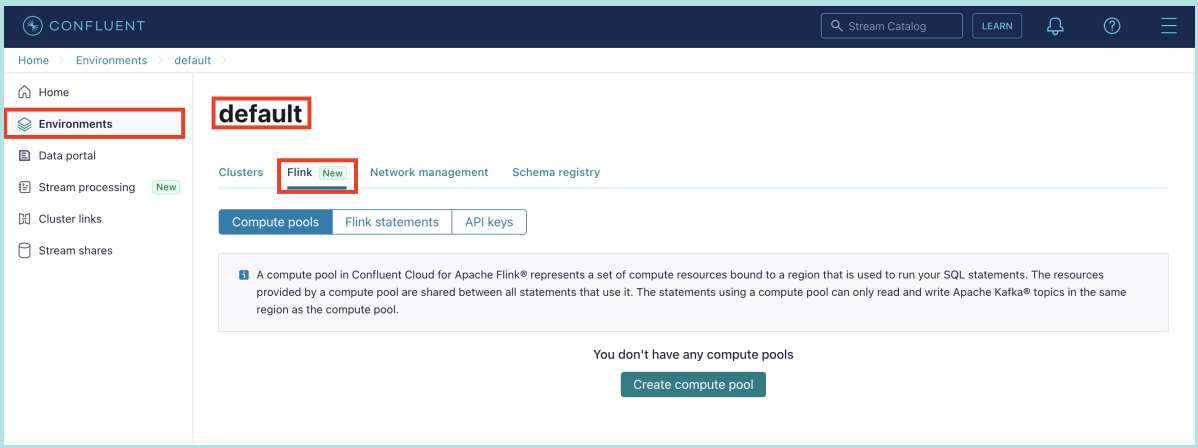
Microsoft Azure

```
Virginia (eastus)
Virginia (eastus2)
Washington (westus2)
Netherlands (westeurope)
Pune (centralindia)
Singapore (southeastasia)
```

Costs will vary with these choices, but they are clearly shown on the dropdown, so you'll know what you're getting. You can choose the cloud/region combination you prefer from the list above. Then, click **Continue**.



- In the "Payment" page, you should see a **Skip payment** button in the lower part of the screen. Click on that button.
  If **Skip payment** button is not shown, scroll down and click on **Promo codes**. Paste the code provided by your instructor that stars with "**POPTOUT...**" and then click **Apply**. Then you should see the **Skip payment** button.
- Give your cluster a name like **kafka_cluster** and review **Configuration & cost**, **Usage limits** and **Uptime SLA**. Then, select **Launch cluster**.
- Now, go to the environment **default** page and choose the **Flink** tab.



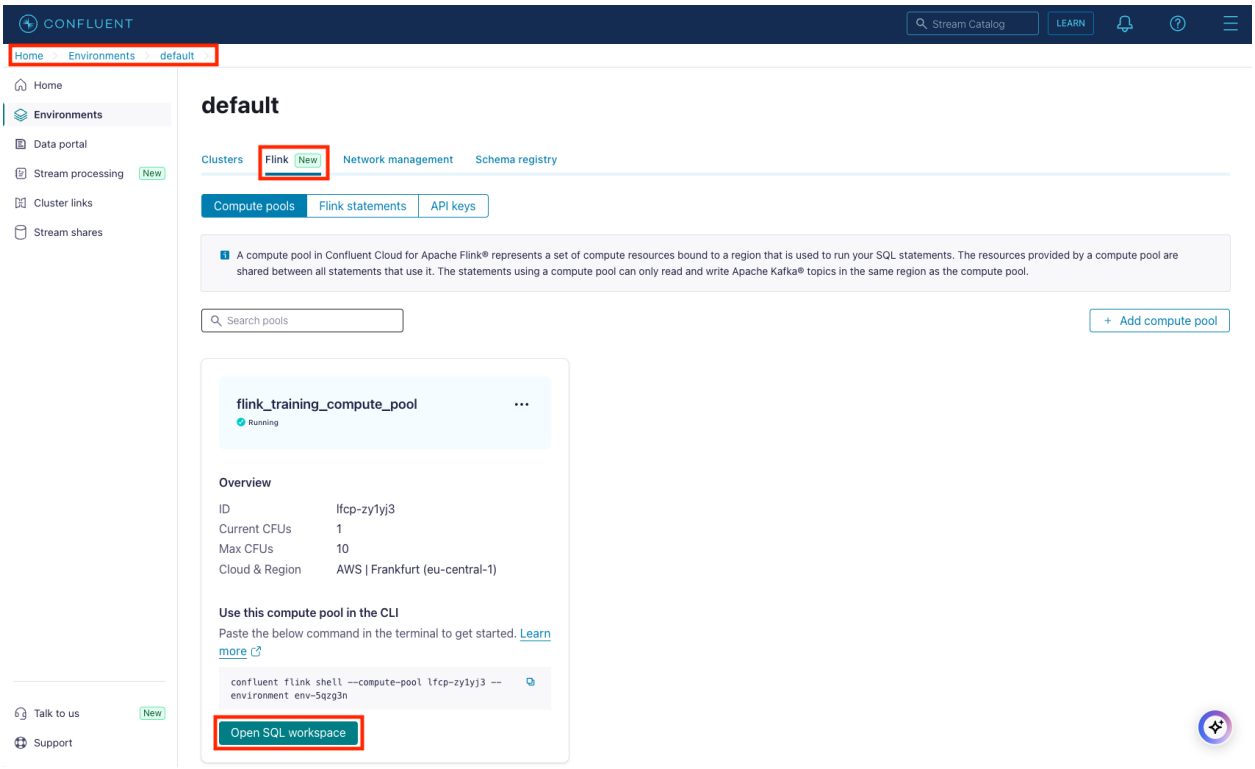
- From here you can create a new compute pool. Click on **Create compute pool**:

Choose the same cloud provider and region as the ones you selected earlier for the Kafka cluster.
The Flink compute pool can only access the data from any Kafka cluster located in the same region as the compute pool.

- Choose the cloud provide and region, then click **Continue**
- Enter the Pool name: **flink_training_compute_pool** and choose **10 CFUs** (Confluent Flink Units)
- Finally, click **Create**

  The pool will be provisioned and ready to use in a few moments.

1. In the Confluent Cloud UI, navigate to the **Flink** tab under the **default** environment and click on **Open SQL workspace** in the **flink_training_compute_pool**.



2. Click on the plus icon

in the upper right side to create a new workspace.

3. In the popup window, change the workspace name to `lab4-workspace` and select the Compute Pool from the dropdown list.
4. Click on **Create workspace**.

Creating a new Dynamic Table

5. Try to write a query in a cell of the new workspace to create a table with:
    - Table name `user_actions`
    - Three columns (**user** `STRING`, **action** `STRING`, **timestamp** `TIMESTAMP_LTZ(3)`)
    - Data distributed by the **user** column
    - Topic related to this topic with **2 partitions**

TRY IT YOURSELF FIRST. *If you get stuck, check the solution*

```
CREATE TABLE user_actions (
  `user` STRING,
  `action` STRING,
  `timestamp` TIMESTAMP_LTZ(3)
) DISTRIBUTED BY (`user`) INTO 2 BUCKETS;
```

6. Let's view the default Watermarking for this new table. Run this query:

```
> DESCRIBE EXTENDED user_actions;

+-------------+-----------------+----------
+-----------------------------------+---------+
| Column Name |    Data Type     | Nullable |              Extras
| Comment |
+-------------+-----------------+----------
+-----------------------------------+---------+
| user        | STRING           | NULL     | BUCKET KEY
|         |
| action      | STRING           | NULL     |
|         |
| timestamp   | TIMESTAMP_LTZ(3) | NULL     |
|         |
| $rowtime    | TIMESTAMP_LTZ(3) | NOT NULL | METADATA VIRTUAL,
| SYSTEM  |
|             | *ROWTIME*        |          | WATERMARK AS
`SOURCE_WATERMARK`() |         |
+-------------+-----------------+----------
+-----------------------------------+---------+
```

As you can see in the output:

- **user** is the `BUCKET KEY`
- **$rowtime** is used as the default WATERMARK for your new table:
  - The default WATERMARK is defined by a Confluent algorithm called `SOURCE_WATERMARK()`.
    It adapts automatically based on the timestamps of your data and incoming frequency of events.
    It is suitable for 80% of all use cases.

7. Now, you are going to customize the WATERMARK using `ALTER TABLE` so the WATERMARK is defined using the `timestamp` column with a delay of 2 seconds. Run the following query:

```
> ALTER TABLE user_actions MODIFY WATERMARK FOR `timestamp` AS
`timestamp` - INTERVAL '2' SECOND;
```

8. Let's check that the custom WATERMARK has been applied to the table. Run again this query:

```
> DESCRIBE EXTENDED user_actions;

+-------------+------------------+----------
+-----------------------------------+---------+
| Column Name |    Data Type     | Nullable |              Extras
| Comment |
+-------------+------------------+----------
+-----------------------------------+---------+
| user        | STRING           | NULL     | BUCKET KEY
|         |
|             |                  |          |
|         |
| action      | STRING           | NULL     |
|         |
|             |                  |          |
|         |
| timestamp   | TIMESTAMP_LTZ(3) | NULL     | WATERMARK AS
|         |
|             | *ROWTIME*        |          | `timestamp` - INTERVAL
'2' SECOND |         |
|             |                  |          |
|         |
| $rowtime    | TIMESTAMP_LTZ(3) | NOT NULL | METADATA VIRTUAL
| SYSTEM  |
+-------------+------------------+----------
+-----------------------------------+---------+
```

Note that we could have applied the custom WATERMARK directly in the `CREATE TABLE …` query instead of using the `ALTER TABLE …` command afterward. The query would look like this:

```
> CREATE TABLE user_actions (
  `user` STRING,
  `action` STRING,
  `timestamp` TIMESTAMP_LTZ(3),
  WATERMARK FOR `timestamp` AS `timestamp` - INTERVAL '2'
SECOND
) DISTRIBUTED BY (`user`) INTO 2 BUCKETS;
```

Evaluating the WATERMARK

9. First, let's run a query to count the number of actions per user every 1 minute.

```
> SELECT
    window_start,
    window_end,
    user,
    COUNT(*) AS number_actions
  FROM TABLE(
    TUMBLE(TABLE user_actions, DESCRIPTOR(`timestamp`), INTERVAL '1'
MINUTE))
  GROUP BY user, window_start, window_end;
```

Don't stop the query. Let it run continuously.

10. Now, insert some data to the `user_actions` table to get some results from the previous `COUNT()` query. Run the following query to insert events within a 1-minute interval:

```
> INSERT INTO user_actions VALUES
    ('Ariane', 'click', TIMESTAMP '2024-06-25 10:00:00'),
    ('Ariane', 'click', TIMESTAMP '2024-06-25 10:00:25'),
    ('Ariane', 'view',  TIMESTAMP '2024-06-25 10:00:51'),
    ('Ariane', 'click', TIMESTAMP '2024-06-25 10:01:01');
```

Question:

Have you got any result from the previous `COUNT()` query? If not, why?

Answer

A reason why you didn't receive any results is because the WATERMARK is defined as `timestamp - INTERVAL '2' SECOND`. Therefore, an event with a timestamp of `10:01:02` or later is required to consider the window `[10:00:00, 10:01:00)` as closed.

11. Let's insert a new event with a higher timestamp. Run:

```
> INSERT INTO user_actions VALUES
    ('Ariane', 'view', TIMESTAMP '2024-06-25 10:01:13');
```

Question:

Have you received any results now? If all previous messages were produced to the same partition, should you be getting any results?

12. Insert more messages to familiarize yourself with WATERMARKS and understand when the result of a WINDOW is output.

    Feel free to use the following examples or create your own:

```
('Mark', 'login', TIMESTAMP '2024-06-25 10:01:33')
('Kezia', 'view', TIMESTAMP '2024-06-25 10:01:54')
('Ariane', 'logout', TIMESTAMP '2024-06-25 10:02:00')
('Chris', 'click', TIMESTAMP '2024-06-25 10:02:02')
('Mark', 'click', TIMESTAMP '2024-06-25 10:05:10')
('Kezia', 'click', TIMESTAMP '2024-06-25 10:09:04')
('Kezia', 'view', TIMESTAMP '2024-06-25 10:09:15')
```

Manipulating Time Data

In this section, you need to use some Flink Time & Date functions to get the desired results. Check the documentation for assistance.

13. Using the column `timestamp` of the table `user_actions`, write a query to get this type of result:

```
user   action year month day
Ariane click  2024 6     25
Ariane click  2024 6     25
Ariane view   2024 6     25
```

TRY IT YOURSELF FIRST. *If you get stuck, check the solution*

```
SELECT
    `user`,
    `action`,
    YEAR(`timestamp`) AS `year`,
    MONTH(`timestamp`) AS `month`,
    DAYOFMONTH(`timestamp`) AS `day`
FROM user_actions;
```

14. Using the column `timestamp` of the table `user_actions`, write a query to get a result where `current_time` is a column with the current timestamp and `minutes_diff` is the difference in minutes between the columns `timestamp` and `current_time`:

```
user    action  timestamp                current_time
minutes_diff
Ariane  click   2024-06-25 10:00:00.000  2024-06-27 20:16:41.906
3496
Ariane  click   2024-06-25 10:00:25.000  2024-06-27 20:16:41.907
3496
Ariane  view    2024-06-25 10:00:51.000  2024-06-27 20:16:41.907
3495
```

TRY IT YOURSELF FIRST. *If you get stuck, check the solution*

```
SELECT
    user,
    action,
    `timestamp`,
    CURRENT_TIMESTAMP AS `current_time`,
    TIMESTAMPDIFF(MINUTE, `timestamp`, CURRENT_TIMESTAMP) AS
minutes_diff
FROM user_actions;
```

15. Using the column `timestamp` of the table `user_actions`, write a query to modify the format of the timestamp so the result is like:

```
user    action  formatted_timestamp
Ariane  click   Tue, 25 Jun 2024 10:00:00
Ariane  click   Tue, 25 Jun 2024 10:00:25
Ariane  view    Tue, 25 Jun 2024 10:00:51
```

TRY IT YOURSELF FIRST. *If you get stuck, check the solution*

```
SELECT
    user,
    action,
    DATE_FORMAT(`timestamp`, 'EEE, d MMM yyyy HH:mm:ss') AS
formatted_timestamp
FROM user_actions;
```

CHALLENGE

Setting Up with Terraform

16. Go to the VM and open a Terminal window.
17. In the Terminal, change directory to `~/confluent-flink/terraform/lab-time`. Run:

```
$ cd ~/confluent-flink/terraform/lab-time
```

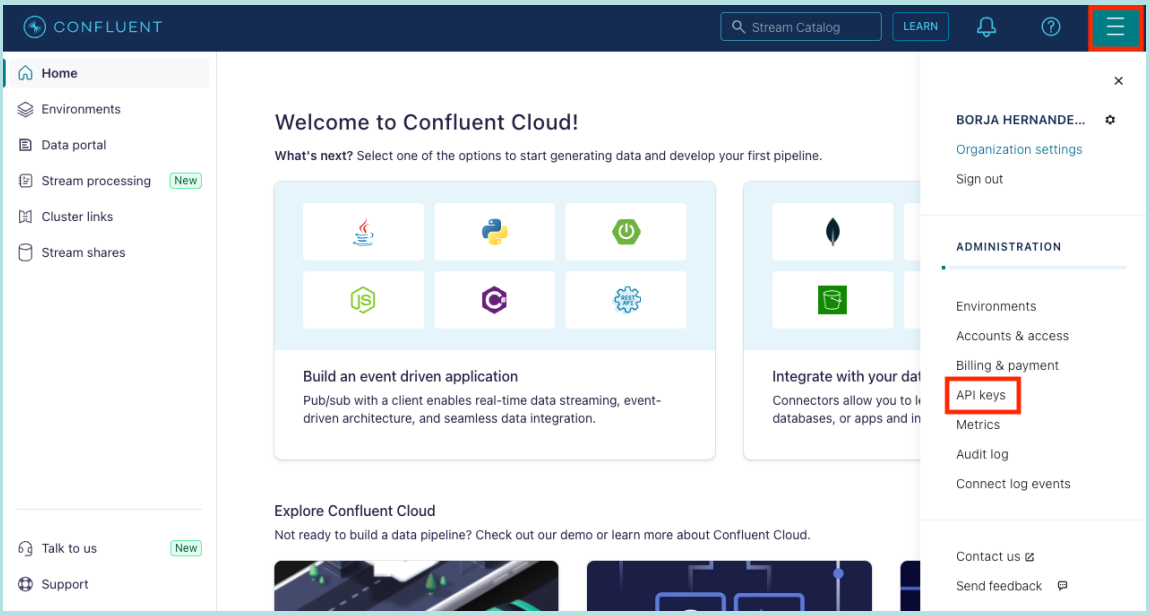
18. Open the file `terraform.tfvars` in VS Code. Run:

```
$ code terraform.tfvars
```

19. Replace the API key and the API secret with the credentials you obtained in Lab 3, ensuring they are enclosed in quotes (" ").

If you don't have the API Key and API Secret, follow these instructions

- Go to [Confluent Cloud website] and log in to your Confluent Cloud account.
- Click on the hamburger menu in the upper-right corner, then select **API keys**.

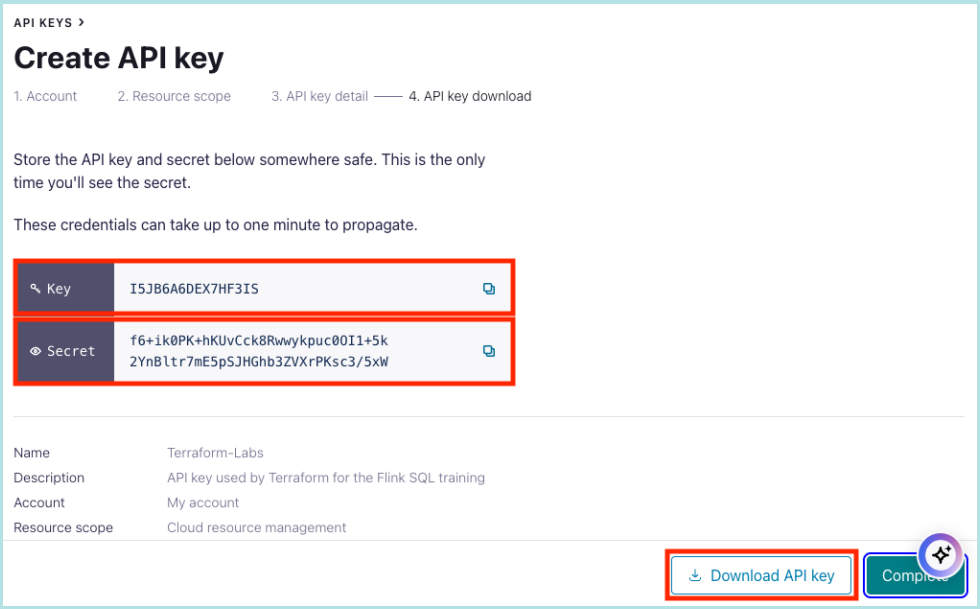


- Click on **Add API key**. Then use the following instructions:
  - Select **My account** as the owner of the new API key and click **Next**.

> Note that **My Account** is only recommended for development environments like this training. In production, you should use **Service Accounts** as the owner of API keys and provide only the minimal required permissions.

  - Select **Cloud resource management**, then click **Next**.
  - Add a name to the new API key to identify it from others.
    - Name: **Terraform-Labs**
    - Description: **API key used by Terraform for the Flink SQL training**
  - Click on **Create API key**.
  - Click on **Download API key** to save it.

If you are on the VM, copy the **KEY** and **SECRET** externally to the VM. This way, if you need to destroy the VM in the future, you can retain the API key.



- Finally, click on **Complete**.

20. Press `Ctrl + S` to save the changes.

21. Return to the Terminal and initialize Terraform by running `terraform init`. Ensure you are in the `~/confluent-flink/terraform/lab-time` directory:

```
$ terraform init
```

22. Create the Terraform execution plan by running:

```
$ terraform plan
```

23. Finally, run `terraform apply` to apply the execution plan and create all the Confluent Cloud resources:

```
$ terraform apply
```

Type `yes` to confirm.

Wait a few minutes for the new Confluent Cloud resources to become available:

- Confluent Cloud Environment: **Training-Env**
- Kafka Cluster: **training-kafka**
- Datagen Connector: **clicks_DatagenSourceConnector**
- Kafka Topic: **clicks**
- Flink Compute Pool: **training-compute-pool**

Monitoring User Activity

A company wants to monitor user activity on their website. There is a Kafka topic called `clicks` that contains data of user clicks from past to present.
We need to use this data to calculate the number of clicks each `userid` performs within a 10-second window, updated every 5 seconds (a hopping window), and identify only the `userid`'s who have performed more than 25 clicks within any such window.
This information should be output to a new Kafka topic called `high_activity_users`.

Tips:

- The `clicks` data includes a column called `time`, which contains the actual timestamp of the click in milliseconds since the Unix epoch.
- The messages are in strict time order.

TRY IT YOURSELF FIRST. *If you get stuck, check the solution*

```
ALTER TABLE clicks ADD `ts` AS TO_TIMESTAMP_LTZ(CAST(`time` AS
BIGINT), 3);
```

```
ALTER TABLE clicks MODIFY WATERMARK FOR `ts` AS `ts`;
```

```
CREATE TABLE high_activity_users (
    window_start TIMESTAMP_LTZ(3),
    window_end TIMESTAMP_LTZ(3),
    userid INT,
    clicks_count BIGINT
);
```

```
INSERT INTO high_activity_users
SELECT
    window_start,
    window_end,
    userid,
    COUNT(*) AS clicks_count
FROM TABLE(
  HOP(TABLE clicks, DESCRIPTOR(`ts`), INTERVAL '5' SECONDS, INTERVAL
'10' SECONDS))
GROUP BY userid, window_start, window_end
HAVING COUNT(*) > 25;
```

```
SELECT * FROM high_activity_users;
```

Clean-Up

24. Go back to the VM, open a terminal window and run the following command to change the directory to `~/confluent-flink/terraform/lab-time`

```
$ cd ~/confluent-flink/terraform/lab-time
```

25. Now, destroy all the resources created by Terraform using:

```
$ terraform destroy
```

Conclusion

In conclusion, you have gained familiarity with timestamps and watermarks by viewing and modifying the default watermark to define a custom one. You have seen how watermarks operate in real-time by inserting data into a table and have practiced using various Time & Date functions. Through the final challenge, you have reinforced your understanding and application of these concepts.

STOP HERE. THIS IS THE END OF THE EXERCISE.

Lab 13 Using Aggregations in a Practical Use Case

a. Using Aggregations in a Practical Use Case

The goal of this lab is to familiarize yourself with aggregations. You will begin with basic aggregations to count unique customers and gather insights on shoe products by brand. You will then perform historical price trends analysis, use `WITH` statements for customer filtering and implement top-N aggregation to identify top-spending customers. Finally, you will categorize customer loyalty levels based on spending and insert promotional campaign details into a promotions table.

Setting up the Environment using Terraform

1. Go to the VM and open a Terminal window.
2. In the Terminal, change directory to `~/confluent-flink/terraform/lab-aggregations`. Run:

```
$ cd ~/confluent-flink/terraform/lab-aggregations
```

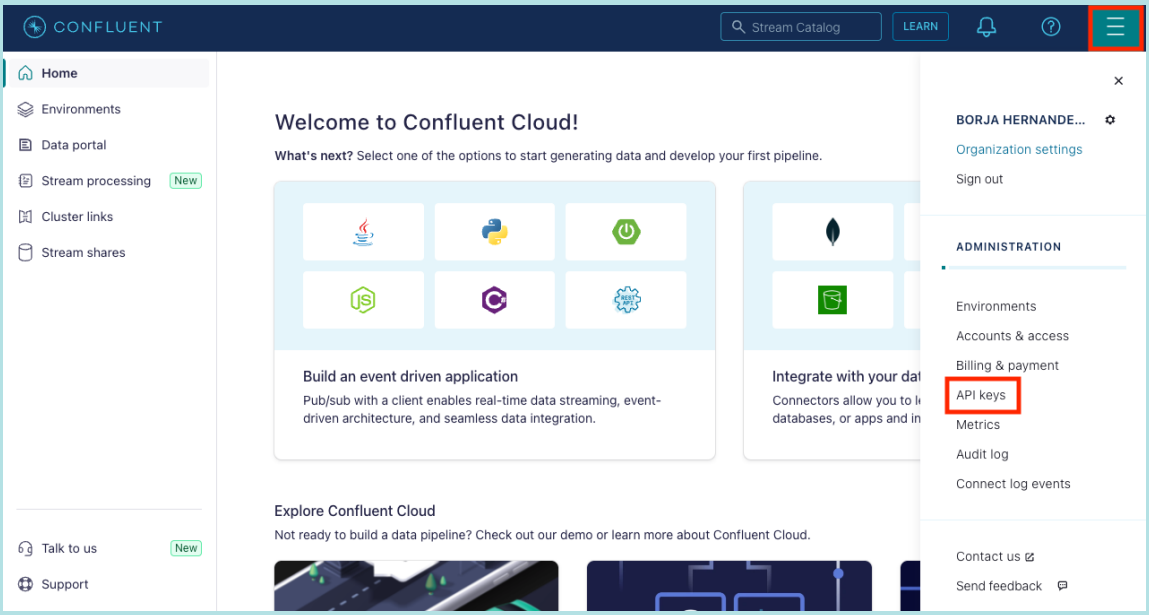
3. Open the file `terraform.tfvars` in VS Code. Run:

```
$ code terraform.tfvars
```

4. Replace the API key and the API secret with the credentials you obtained in Lab 3, ensuring they are enclosed in quotes (" ").

You can use the API Key/Secret created in Lab 3. If you didn't do it and you don't have the API Key and API Secret, follow these instructions

- Go to Confluent Cloud website and log in to your Confluent Cloud account.
- Click on the hamburger menu in the upper-right corner, then select **API keys**.

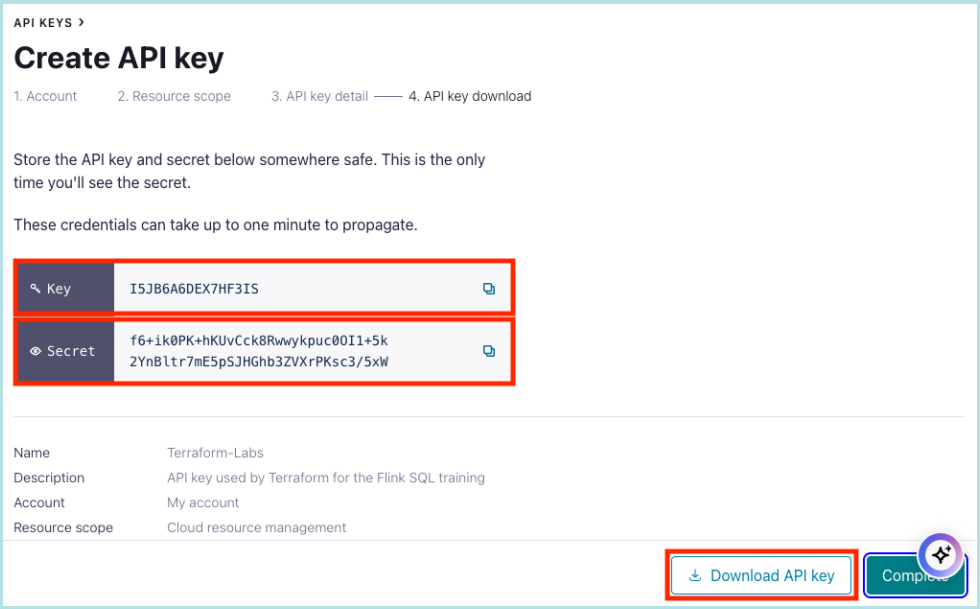


- Click on **Add API key**. Then use the following instructions:
  - Select **My account** as the owner of the new API key and click **Next**.

> Note that **My Account** is only recommended for development environments like this training. In production, you should use **Service Accounts** as the owner of API keys and provide only the minimal required permissions.

  - Select **Cloud resource management**, then click **Next**.
  - Add a name to the new API key to identify it from others.
    - Name: **Terraform-Labs**
    - Description: **API key used by Terraform for the Flink SQL training**
  - Click on **Create API key**.
  - Click on **Download API key** to save it.

If you are on the VM, copy the **KEY** and **SECRET** externally to the VM. This way, if you need to destroy the VM in the future, you can retain the API key.



- Finally, click on **Complete**.

5. Press `Ctrl + S` to save the changes.

6. Return to the Terminal and initialize Terraform by running `terraform init`. Ensure you are in the `~/confluent-flink/terraform/lab-aggregations` directory:

```
$ terraform init
```

7. Create the Terraform execution plan by running:

```
$ terraform plan
```

8. Finally, run `terraform apply` to apply the execution plan and create all the Confluent Cloud resources:

```
$ terraform apply
```

Type `yes` to confirm.

Wait around 10 minutes for the new Confluent Cloud resources to become available:

- Confluent Cloud Environment: **Shoe-Env**
- Kafka Cluster: **shoe-kafka**
- Datagen Connectors: **Shoe_Customers_DatagenSourceConnector**, **Shoe_Orders_DatagenSourceConnector**, **Shoe_Products_DatagenSourceConnector**,
- Kafka Topics: **shoe_customers**, **shoe_orders**, **shoe_products**
- Flink Compute Pool: **shoe-compute-pool**
- Flink Tables: **shoe_customers_keyed**, **shoe_products_keyed**, **shoe_orders_enriched**

(OPTIONAL)
In the meantime, check the Terraform file `main.tf` in VS Code to familiarize yourself with the syntax. Run the following command:

```
$ code ~/confluent-flink/terraform/lab-aggregations/main.tf
```

Starting with Simple Aggregations

The table `shoe_customers` contains personal information about all customers that have ordered a pair of shoes through our website.

9. Run the following query to find the number of customers records:

```
> SELECT COUNT(id) AS num_records
  FROM shoe_customers;
```

10. The result of the previous query is not the actual number of unique customers. Think about how the query to obtain the number of unique customers should be written and run it.

    TRY IT YOURSELF FIRST. *If you get stuck, check the solution*

```
> SELECT COUNT(DISTINCT id) AS num_customers
  FROM shoe_customers;
```

Now, we want to get multiple insights from table `shoe_products`. This tables contains information about the shoe brand, model (`name`), price and customer rating.

11. Write and run a query to get the following information:
    - Brand name
    - Number of unique models per brand
    - Overall average rating per brand rounded to 2 decimal places
    - Highest price per brand
    - String listing all shoe models per brand

brand_name	number_models	avg_rating	max_price	model_names
Grady Group	35	2.76	24995	Sidekick Spectra5 267,Impreza Sidekick 956,Impreza Pro 596,Pe
Tillman, Effertz and Nikolaus	32	2.8	22995	Impreza Spectra5 227,Max Impreza 151,Pro TrailRunner 985,Perf
Beer, DAmore and Wintheiser	32	3.14	18995	Spectra5 Perf 274,Max Pro 453,TrailRunner Impreza 452,Pro Perf
Okuneva, McCullough and Re...	28	2.33	22995	Spectra5 TrailRunner 882,Sidekick TrailRunner 454,Sidekick Trail
Will Inc	31	2.88	22995	TrailRunner Pro 118,Perf Impreza 310,Sidekick Spectra5 445,Max
Williamson Group	26	3.37	18995	Pro Spectra5 709,Perf Impreza 910,Spectra5 Sidekick 743,Max P

TRY IT YOURSELF FIRST. *If you get stuck, check the solution*

```
> SELECT brand as brand_name,
       COUNT(DISTINCT name) as number_models,
       ROUND(AVG(rating),2) as avg_rating,
       MAX(sale_price) as max_price,
       LISTAGG(DISTINCT name) as model_names
  FROM shoe_products
  GROUP BY brand;
```

Another interesting query to gain insights from the `shoe_products` table is to add a new column for every new record showing the historical prices for shoes of the same brand and model over time.

12. Create a query to add a column with a collection (multiset) of historical prices for the same brand and model. The result should resemble this format:

id	brand	name	sale_price	rating	historical_prices
acfaf17f-05d3-41c2-b819-fd...	Jones-Stokes	Pro Spectra5 813	11995	5	[["11995","7"]]
c807959b-d7f4-4835-99ef-...	Jones-Stokes	Impreza TrailRunner 273	2397	4.1	[["2397","5"]]
cdc79931-139b-4e19-96cb-...	Macejkovic, Walter and Cum...	Sidekick TrailRunner 507	2495	4.3	[["2495","6"]]
7bb3751a-856c-41f4-b323-...	Tremblay LLC	Perf Impreza 773	8995	5	[["8995","7"]]
c8628d98-ff4f-4840-905b-...	Will Inc	Max Impreza 661	9995	0	[["9995","12"]]

TRY IT YOURSELF FIRST. *If you get stuck, check the solution*

```
> SELECT
      id,
      brand,
      name,
      sale_price,
      rating,
      COLLECT(sale_price) OVER (PARTITION BY brand,name
                                ORDER BY $rowtime
                               ) AS historical_prices
  FROM
      shoe_products;
```

Using WITH()

13. The following query utilizes a WITH statement to create a "temporary view", allowing us to query against this view and retrieve customers whose average order price exceeds $8,000 and whose shoes have an average rating greater than 4.5. Run the following query to execute this search:

```
> WITH CustomerStats AS (
    SELECT
      email,
      first_name,
      last_name,
      AVG(sale_price) AS avg_sale_price,
      AVG(rating) AS avg_rating
    FROM
      shoe_orders_enriched
    GROUP BY
      email, first_name, last_name
  )
  SELECT *
  FROM
    CustomerStats
  WHERE
    avg_sale_price > 8000 AND avg_rating > 4.5;
```

14. Technically, the previous query can be rewritten without using a WITH clause. Modify it to achieve the same results without using WITH.

    TRY IT YOURSELF FIRST. *If you get stuck, check the solution*

```
> SELECT
    email,
    first_name,
    last_name,
    AVG(sale_price) AS avg_sale_price,
    AVG(rating) AS avg_rating
  FROM
    shoe_orders_enriched
  GROUP BY
    email, first_name, last_name
  HAVING AVG(sale_price) > 8000 AND AVG(rating) > 4.5;
```

Using Top-N Aggregation

15. Create a query to find the top 3 customers who have spent the most. Use the `shoe_orders_enriched` table as the source.
    The result should have this structure:

first_name	last_name	email	total_sale	rownum
Colet	Dansken	kglindeo@rambler.ru	168537	1
Augustine	Daveridge	falcornl@virginia.edu	167736	2
Cullen	Feuell	voverall2e@fema.gov	166742	3

TRY IT YOURSELF FIRST. *If you get stuck, check the solution*

```
> SELECT *
  FROM (
    SELECT
      *,
      ROW_NUMBER() OVER (ORDER BY total_sale DESC) AS rownum
    FROM (
      SELECT first_name, last_name, email, SUM(sale_price) AS
total_sale
      FROM shoe_orders_enriched
      GROUP BY first_name, last_name, email
    )
  )
  WHERE rownum <= 3;
```

Calculating Customer Loyalty Level

16. Create a new table that will store the loyalty levels for the customers.

```
> CREATE TABLE shoe_loyalty_levels(
    email STRING,
    total BIGINT,
    loyalty_level STRING,
    PRIMARY KEY (email) NOT ENFORCED
  ) DISTRIBUTED INTO 1 BUCKETS;
```

17. Now, write a SQL query to insert into the new table the customer loyalty level per customer based on their total spending on shoe products. The loyalty levels are categorized as 'GOLD', 'SILVER', 'BRONZE', or 'NONE' depending on the total sales amount.

    The customer's loyalty level follows these criteria:

    - 'GOLD' for customers with total spending greater than $700,000.
    - 'SILVER' for customers with total spending greater than $70,000.
    - 'BRONZE' for customers with total spending greater than $7,000.
    - 'NONE' for all other customers.

    Hint: Use the `CASE` statement to categorize customers into different loyalty levels based

on their total spending.

email	total	loyalty_level
kswash5@reuters.com	63662	BRONZE
ffotherby4h@europa.eu	152286	SILVER
pbittlestone6r@twitpic.com	130634	SILVER
kpickburnb@ox.ac.uk	171219	SILVER

TRY IT YOURSELF FIRST. *If you get stuck, check the solution*

```
INSERT INTO shoe_loyalty_levels(
 email,
 total,
 loyalty_level)
SELECT
  email,
  SUM(sale_price) AS total,
  CASE
    WHEN SUM(sale_price) > 700000 THEN 'GOLD'
    WHEN SUM(sale_price) > 70000 THEN 'SILVER'
    WHEN SUM(sale_price) > 7000 THEN 'BRONZE'
    ELSE 'NONE'
  END AS loyalty_level
FROM shoe_orders_enriched
GROUP BY email;
```

Creating Promotional Campaigns

Finally, you will write SQL queries to create promotional campaigns based on customer purchasing behavior for specific brands of shoe products.

18. The table containing the promotion notifications is defined like this. Run this query:

```
> CREATE TABLE shoe_promotions(
    email STRING,
    promotion_name STRING,
    PRIMARY KEY (email) NOT ENFORCED
  ) DISTRIBUTED INTO 1 BUCKETS;
```

Currently, we have two active promotional campaigns:

1. Next Free Promotion:
   Customers qualify for the `next_free` promotion if they purchase 2 products from the brand 'Jones-Stokes'.
2. Bundle Offer Promotion:
   Customers qualify for the `bundle_offer` promotion if they purchase at least 3 products in total from both 'Braun-Bruen' and 'Will Inc' brands

19. Write two `INSERT INTO…` statements (one per promotion) to add these promotions into the `shoe_promotions` table.

    TRY IT YOURSELF FIRST. *If you get stuck, check the solution*

```
INSERT INTO shoe_promotions
SELECT
   email,
   'next_free' AS promotion_name
FROM shoe_orders_enriched
WHERE brand = 'Jones-Stokes'
GROUP BY email
HAVING COUNT(*) % 2 = 0;
```

```
INSERT INTO shoe_promotions
SELECT
   email,
   'bundle_offer' AS promotion_name
FROM shoe_orders_enriched
WHERE brand IN ('Braun-Bruen', 'Will Inc')
GROUP BY email
HAVING COUNT(DISTINCT brand) = 2 AND COUNT(brand) > 3;
```

Clean-Up

20. Go back to the VM, open a terminal window and run the following command to change the directory to `~/confluent-flink/terraform/lab-aggregations`

```
$ cd ~/confluent-flink/terraform/lab-aggregations
```

21. Now, destroy all the resources created by Terraform using:

```
$ terraform destroy
```

Conclusion

Through this lab, you've gained practical skills in SQL for aggregating data, creating temporary views with `WITH` statements, and deriving actionable insights from large datasets. Analyzing customer behavior, categorizing loyalty levels, and designing targeted promotions based on purchase patterns are key applications of these skills in business intelligence and data analysis.

Lab 14 Exploring Various Types of Joins

a. Exploring Various Types of Joins

The goal of this lab is to understand and practice using various types of SQL joins in Flink SQL. You will learn how to perform INNER JOINS, INTERVAL JOINS, TEMPORAL JOINS and WINDOW JOINS to analyze customer behavior by examining the products they viewed before making a purchase.

Setting up the Environment using Terraform

1. Go to the VM and open a Terminal window.
2. In the Terminal, change directory to `~/confluent-flink/terraform/lab-joins`. Run:

```
$ cd ~/confluent-flink/terraform/lab-joins
```

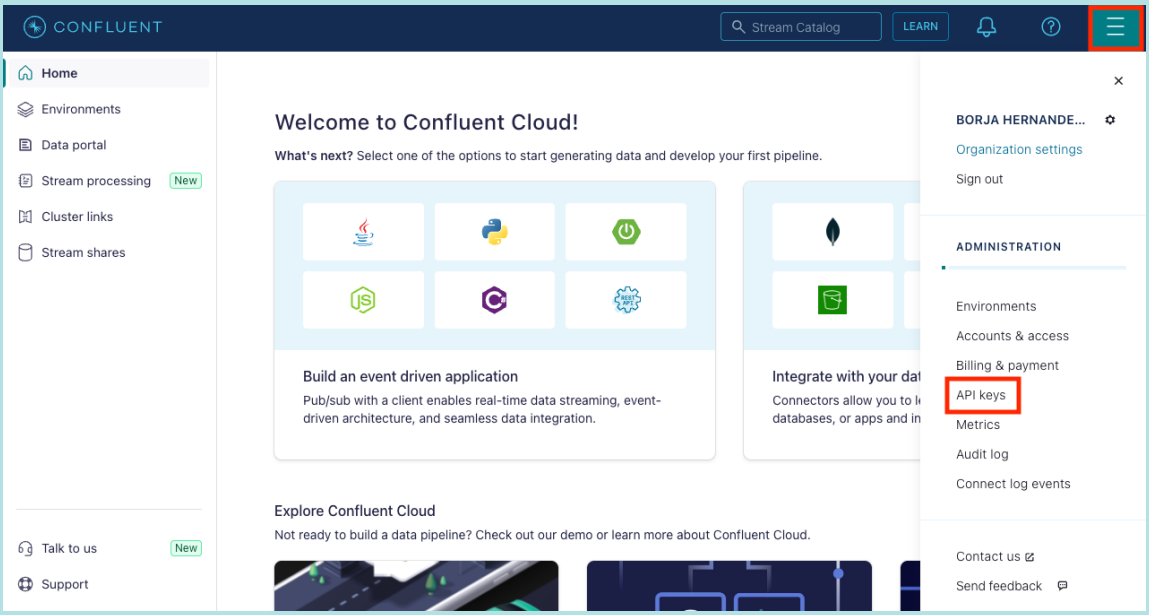
3. Open the file `terraform.tfvars` in VS Code. Run:

```
$ code terraform.tfvars
```

4. Replace the API key and the API secret with the credentials you obtained in Lab 3, ensuring they are enclosed in quotes (" ").

You can use the API Key/Secret created in Lab 3. If you didn't do it and you don't have the API Key and API Secret, follow these instructions

- Go to Confluent Cloud website and log in to your Confluent Cloud account.
- Click on the hamburger menu in the upper-right corner, then select **API keys**.

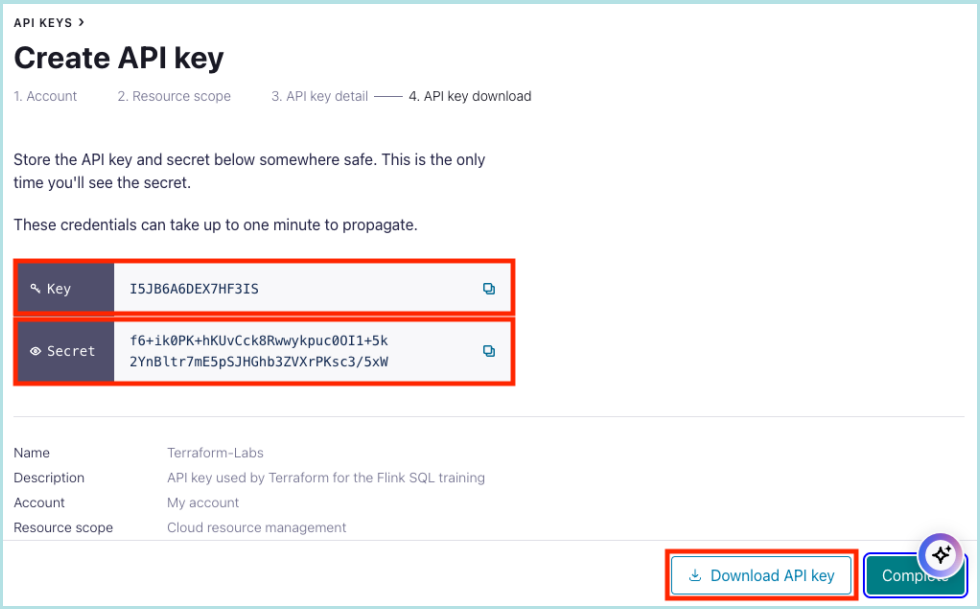


- Click on **Add API key**. Then use the following instructions:
  - Select **My account** as the owner of the new API key and click **Next**.

> Note that **My Account** is only recommended for development environments like this training. In production, you should use **Service Accounts** as the owner of API keys and provide only the minimal required permissions.

  - Select **Cloud resource management**, then click **Next**.
  - Add a name to the new API key to identify it from others.
    - Name: **Terraform-Labs**
    - Description: **API key used by Terraform for the Flink SQL training**
  - Click on **Create API key**.
  - Click on **Download API key** to save it.

If you are on the VM, copy the **KEY** and **SECRET** externally to the VM. This way, if you need to destroy the VM in the future, you can retain the API key.



- Finally, click on **Complete**.

5. Press `Ctrl + S` to save the changes.

6. Return to the Terminal and initialize Terraform by running `terraform init`. Ensure you are in the `~/confluent-flink/terraform/lab-joins` directory:

```
$ terraform init
```

7. Create the Terraform execution plan by running:

```
$ terraform plan
```

8. Finally, run `terraform apply` to apply the execution plan and create all the Confluent Cloud resources:

```
$ terraform apply
```

Type `yes` to confirm.

Wait around 10 minutes for the new Confluent Cloud resources to become available:

- Confluent Cloud Environment: **Joins-Env**
- Kafka Cluster: **joins-kafka**
- Datagen Connectors: **Shoe_Clickstream_DatagenSourceConnector**, **Shoe_Customers_DatagenSourceConnector**, **Shoe_Orders_DatagenSourceConnector**, **Shoe_Products_DatagenSourceConnector**,
- Kafka Topics: **shoe_clickstream**, **shoe_customers**, **shoe_orders**, **shoe_products**
- Flink Compute Pool: **shoe-compute-pool**

(In the meantime, go to the next section "**Description of the Exercise**" and familiarized yourself with the input tables)

Description of the Exercise

The objective of this lab exercise is to analyze customer behavior by examining the products they viewed before making a purchase.

Table: `shoe_clickstream` (append-only)

This table captures all user clicks on the shoe shop website and includes the following columns:
(Note: In this lab, you will use only the first three columns)

- `product_id`: ID of the product viewed
- `user_id`: ID of the user/customer
- `ts`: Click timestamp
- `view_time`
- `page_url`
- `ip`

Table: `shoe_orders` (append-only)

This table captures all orders on the shoe shop website and includes the following columns:

- `order_id`: ID of the order
- `product_id`: ID of the product ordered
- `customer_id`: ID of the user/customer
- `ts`: Order timestamp

Table: `shoe_products` (upsert)

This table contains the customer information and includes the following columns:
(Note: In this lab, you will use only the `key` and `name` columns)

- `key`: Product ID **(PRIMARY KEY - field in the message key)**
- `id`: Product ID (field in the message value)
- `brand`
- `name`: Name of the product
- `sale_price`
- `rating`

Table: `shoe_customers` (upsert)

This table contains the customer information and includes the following columns:
(Note: In this lab, you will use only the `key` and `email` columns)

- `key`: Customer ID **(PRIMARY KEY - field in the message key)**
- `id`: Customer ID (field in the message value)
- `first_name`
- `last_name`
- `email`: Email of the customer
- `phone`
- `street_address`
- `state`

- `zip_code`
- `country`
- `country_code`

Starting with Regular Joins

9. Once Terraform finished, open the SQL Workspace in the compute pool `shoe-compute-pool` under the `Joins-Env` environment.

10. Let's start with a simple `INNER JOIN` to join the tables `shoe_clickstream` and `shoe_orders` based on the ID of the user/customer. This will allow us to relate orders with views by the same customer.
    Copy and paste the following query into a cell in the SQL Workspace. Complete the blanks and run it:

```
> SELECT
    so.product_id AS product_id_ordered,
    sc.product_id AS product_id_viewed,
    so.customer_id,
    so.order_id,
    so.ts AS order_ts,
    sc.ts AS click_ts
  FROM
    shoe_clickstream AS sc
  _______
    shoe_orders AS so
  ON
    __________________;
```

TRY IT YOURSELF FIRST. *If you get stuck, check the solution*

```
> SELECT
    so.product_id AS product_id_ordered,
    sc.product_id AS product_id_viewed,
    so.customer_id,
    so.order_id,
    so.ts AS order_ts,
    sc.ts AS click_ts
  FROM
    shoe_clickstream AS sc
  INNER JOIN
    shoe_orders AS so
  ON
    sc.user_id = so.customer_id;
```

Using an Interval Join

The previous `INNER JOIN` joins any new record on the left side with all previous and future matched records on the right side. This means that the state will grow indefinitely, as past records must be kept indefinitely.

Since our objective is to analyze customer behavior just before a purchase, let's add a time constraint to the query. By converting the previous regular join into an interval join, we can benefit from the efficient state management of interval joins, which remove outdated records.

11. First, run the following query to create the table which will contain the result of the join:

```
> CREATE TABLE shoe_viewed_before_order (
    product_id_ordered STRING,
    product_id_viewed STRING,
    customer_id STRING,
    order_id INT,
    order_ts TIMESTAMP(3),
    click_ts TIMESTAMP(3)
  ) DISTRIBUTED INTO 1 BUCKETS;
```

12. Now, complete the following query to create an Interval Join with the time condition that the `click_ts` must be between 1 hour before the `order_ts` and the `order_ts`

```
> INSERT INTO shoe_viewed_before_order SELECT
    so.product_id AS product_id_ordered,
    sc.product_id AS product_id_viewed,
    so.customer_id,
    so.order_id,
    so.ts AS order_ts,
    sc.ts AS click_ts
  FROM
    shoe_clickstream AS sc,
    shoe_orders AS so
  _______
    sc.user_id = so.customer_id
  AND
    ________________________________________;
```

TRY IT YOURSELF FIRST. *If you get stuck, check the solution*

```
> INSERT INTO shoe_viewed_before_order SELECT
    so.product_id AS product_id_ordered,
    sc.product_id AS product_id_viewed,
    so.customer_id,
    so.order_id,
    so.ts AS order_ts,
    sc.ts AS click_ts
  FROM
    shoe_clickstream AS sc,
    shoe_orders AS so
  WHERE
    sc.user_id = so.customer_id
  AND
    sc.ts BETWEEN so.ts - INTERVAL '1' HOUR AND so.ts;
```

13. Feel free to run a `SELECT…` statement to view the content of table `shoe_viewed_before_order`

```
> SELECT * FROM shoe_viewed_before_order;
```

Enriching with Temporal Joins

In the table `shoe_viewed_before_order`, each row represents a product viewed by a customer within 1 hour before making a purchase.

Next, we want to enrich the `shoe_viewed_before_order` table by adding the product name (based on its product ID) and the customer email (based on its customer ID).

14. Create a table to store the results of the Temporal Joins by running the following query:

```
> CREATE TABLE shoe_viewed_before_order_enriched (
    product_ordered STRING,
    products_viewed STRING,
    email STRING,
    order_id INT,
    order_ts TIMESTAMP(3),
    clicks_ts STRING
  ) DISTRIBUTED INTO 1 BUCKETS
  WITH ('changelog.mode' = 'retract');
```

15. The following query is a multi-join query containing three Temporal Joins (lookup joins) to achieve the following:

    - Retrieve the product name based on the `product_id_ordered`
    - Retrieve the product name based on the `product_id_viewed`
    - Retrieve the customer email based on the `customer_id`

    Additionally, the query includes a `GROUP BY` clause to create a list of products viewed before the purchase and a list of `click_ts`.

    Complete the following query and run it:

```
> INSERT INTO shoe_viewed_before_order_enriched SELECT
    spo.`name` AS product_ordered,
    LISTAGG(spv.`name`) AS products_viewed,
    sc.email,
    svbo.order_id,
    svbo.order_ts,
    LISTAGG(CAST(svbo.click_ts AS STRING)) AS clicks_ts
  FROM
    shoe_viewed_before_order AS svbo
  LEFT JOIN shoe_products ________________________ AS spo
    ON ________________________
  LEFT JOIN shoe_products ________________________ AS spv
    ON ________________________
  LEFT JOIN shoe_customers ________________________ AS sc
    ON ________________________
  GROUP BY
    ________________________________________________;
```

TRY IT YOURSELF FIRST. *If you get stuck, check the solution*

```
> INSERT INTO shoe_viewed_before_order_enriched SELECT
    spo.`name` AS product_ordered,
    LISTAGG(spv.`name`) AS products_viewed,
    sc.email,
    svbo.order_id,
    svbo.order_ts,
    LISTAGG(CAST(svbo.click_ts AS STRING)) AS clicks_ts
  FROM
    shoe_viewed_before_order AS svbo
  LEFT JOIN shoe_products FOR SYSTEM_TIME AS OF svbo.`$rowtime` AS
spo
    ON svbo.product_id_ordered = spo.`key`
  LEFT JOIN shoe_products FOR SYSTEM_TIME AS OF svbo.`$rowtime` AS
spv
    ON svbo.product_id_viewed = spv.`key`
  LEFT JOIN shoe_customers FOR SYSTEM_TIME AS OF svbo.`$rowtime`
AS sc
    ON svbo.customer_id = sc.`key`
  GROUP BY
    svbo.order_id, spo.`name`, sc.email, svbo.order_ts;
```

16. Feel free to run a `SELECT…` statement to view the content of table `shoe_viewed_before_order_enriched`

```
> SELECT * FROM shoe_viewed_before_order_enriched;
```

Applying Window Joins

17. Finally, create a window join between the `shoe_clickstream` and `shoe_orders` tables to calculate the number of clicks and the number of orders per customer per day (getting updates of the daily count every hour).

    Complete and run the following query:

```
> SELECT
    sc.window_start,
    sc.window_end,
    so.customer_id,
    sc.num_products_viewed,
    so.num_products_ordered
  FROM
    (SELECT COUNT(1) AS num_products_viewed, user_id, window_start,
window_end
      FROM
        ____________________________________________________
      GROUP BY user_id, window_start, window_end
    ) AS sc
  INNER JOIN
    (SELECT ____________________________________________________
      FROM
        ____________________________________________________
      GROUP BY ____________________________________________________
    ) AS so
  ON
    ___________________
  AND
    ___________________
  AND
    ___________________;
```


TRY IT YOURSELF FIRST. *If you get stuck, check the solution*

```
> SELECT
    sc.window_start,
    sc.window_end,
    so.customer_id,
    sc.num_products_viewed,
    so.num_products_ordered
  FROM
    (SELECT COUNT(1) AS num_products_viewed, user_id,
window_start, window_end
      FROM
        TABLE(CUMULATE(TABLE shoe_clickstream, DESCRIPTOR(ts),
INTERVAL '1' HOUR, INTERVAL '1' DAY))
      GROUP BY user_id, window_start, window_end
    ) AS sc
  INNER JOIN
    (SELECT COUNT(1) AS num_products_ordered, customer_id,
window_start, window_end
      FROM
        TABLE(CUMULATE(TABLE shoe_orders, DESCRIPTOR(ts), INTERVAL
'1' HOUR, INTERVAL '1' DAY))
      GROUP BY customer_id, window_start, window_end
    ) AS so
  ON
    sc.user_id = so.customer_id
  AND
    sc.window_start = so.window_start
  AND
    sc.window_end = so.window_end;
```

Clean-Up

18. Go back to the VM, open a terminal window and run the following command to change the directory to `~/confluent-flink/terraform/lab-joins`

```
$ cd ~/confluent-flink/terraform/lab-joins
```

19. Now, destroy all the resources created by Terraform using:

```
$ terraform destroy
```

Conclusion

In conclusion, this lab provided a comprehensive hands-on experience with different types of SQL joins within Flink SQL. You successfully navigated through setting up the environment using Terraform, executing various joins to analyze customer behavior, and enriching join results with product and customer details. By completing this lab, you gained practical skills in leveraging SQL joins to derive meaningful insights, particularly in the context of customer purchase behavior analysis.

Appendix A: Running All Labs with Docker

Running Labs in Docker for Desktop

If you have installed Docker for Desktop on your Mac or Windows 10 Pro machine you are able to complete the course by building and running your applications from the command line.

- Increase the memory available to Docker Desktop to a minimum of 6 GiB. See the advanced settings for Docker Desktop for Mac, and Docker Desktop for Windows.
- Follow the instructions at → Preparing the Labs to `git clone` the source code. The exercise source code will now be on your host machine where you can use any editor to complete the exercises or experiment.
- From the location where you cloned the source code, launch a tools container:

```
$ docker-compose up -d tools
```

All the command line instructions will work from the tools container. This container has been preconfigured with all of the tools you use in the exercises, e.g. `kafka-topics`, `gradle`, `dotnet` and `python`.

```
$ docker-compose exec tools bash
root@tools:/#
```

The source code cloned onto your host machine is present in the tools container as a bind mount. You can see the source code in `~/confluent-dev`.

```
root@tools:/# ls ~/confluent-dev/
README.md  challenge  docker-compose.yml  postgres  solution  update-
hosts.sh  webserver  webserver-avro
```

Anywhere you are instructed to open additional terminal windows you can `exec` additional bash shells on the tools container with `docker-compose exec tools bash` on your host machine.

- The `docker` or `docker-compose` instructions are run on your host machine.

Running the Exercise Applications

From the `tools` container you can use command line alternatives to the VS Code steps used in the instructions. Complete the coding exercises with an editor of your choice on your host machine. When instructed to run code using the VS Code debugger, instead run your code from within the tools container using these terminal commands:

- For Java applications: `./gradlew run`
- For C# applications: `dotnet run`
- For Python applications: `python3 main.py`

Where you are instructed to stop debugging in VS code use `Ctrl+C` to end the running exercise.

Getting Started

Let's apply this to get you started with the **Introduction** exercise.

1. Open a two terminal windows. The first window will be used for `docker` and `docker-compose` commands on your host machine. The second window will be used for the commands run in the tools container.
2. In the first terminal window, clone the source code repository to the folder `confluent-dev`:

```
$ git clone --depth 1 --branch 1.19.0-v1.0.0 \
    https://github.com/confluentinc/training-developer-src.git \
    confluent-dev
```

3. Start the Kafka cluster:

```
$ docker-compose up -d zookeeper kafka control-center
```

4. In the second terminal window launch the tools container, and open a bash shell.

```
$ docker-compose up -d tools
$ docker-compose exec tools bash
```

5. The exercise command line instructions can now be run in the tools container. The first command we have in the **Introduction** exercise uses `zookeeper-shell`:

```
root@tools:/# zookeeper-shell zookeeper:2181 ls /brokers/ids
```

6. If we skip ahead to the **Kafka Producer** coding exercise, this begins with a command to change to the challenge directory. You can do this in the second terminal window where you are accessing the tools container. Using the Java challenge as an example:

```
root@tools:/# cd ~/confluent-dev/challenge/java-producer
```

7. The next command launches additional containers. Run this command in the first terminal window:

```
$ docker-compose up -d zookeeper kafka control-center create-topics
webserver
```

8. Complete the source code challenges in `confluent-dev/challenge/java-producer/src/main/java/clients/Producer.java` on your host machine. Now you can build and run in the second terminal window:

```
root@tools:/# cd ~/confluent-dev/challenge/java-producer
root@tools:/# ./gradlew run
```