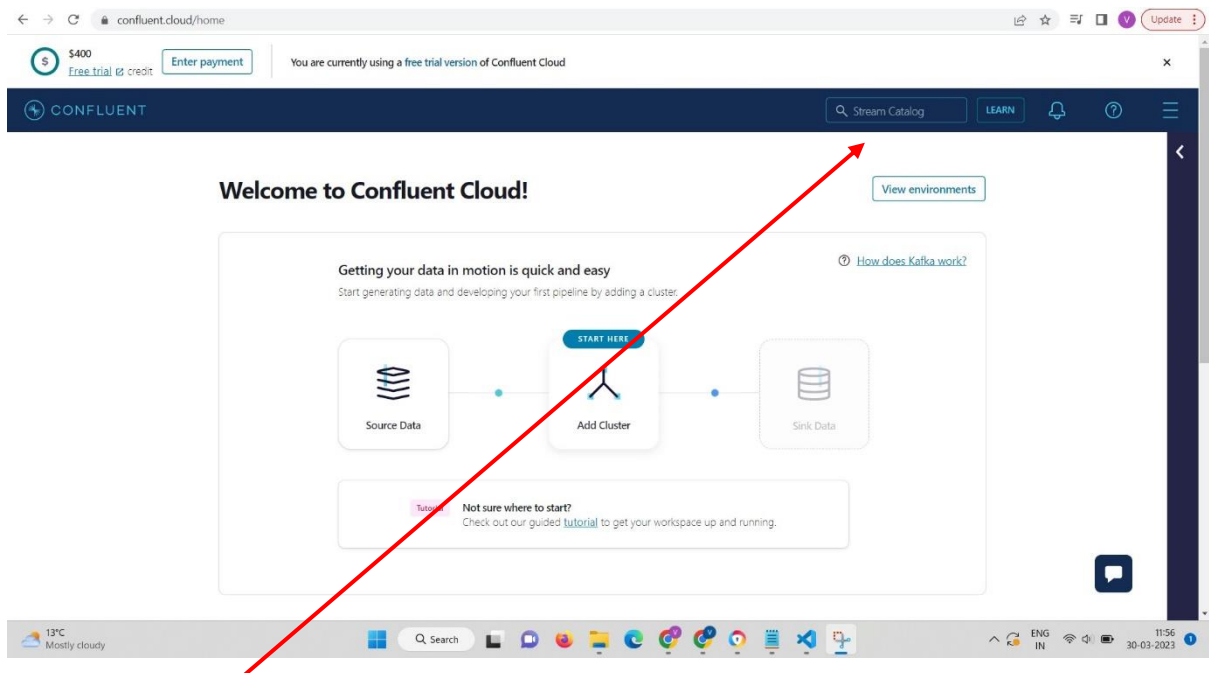
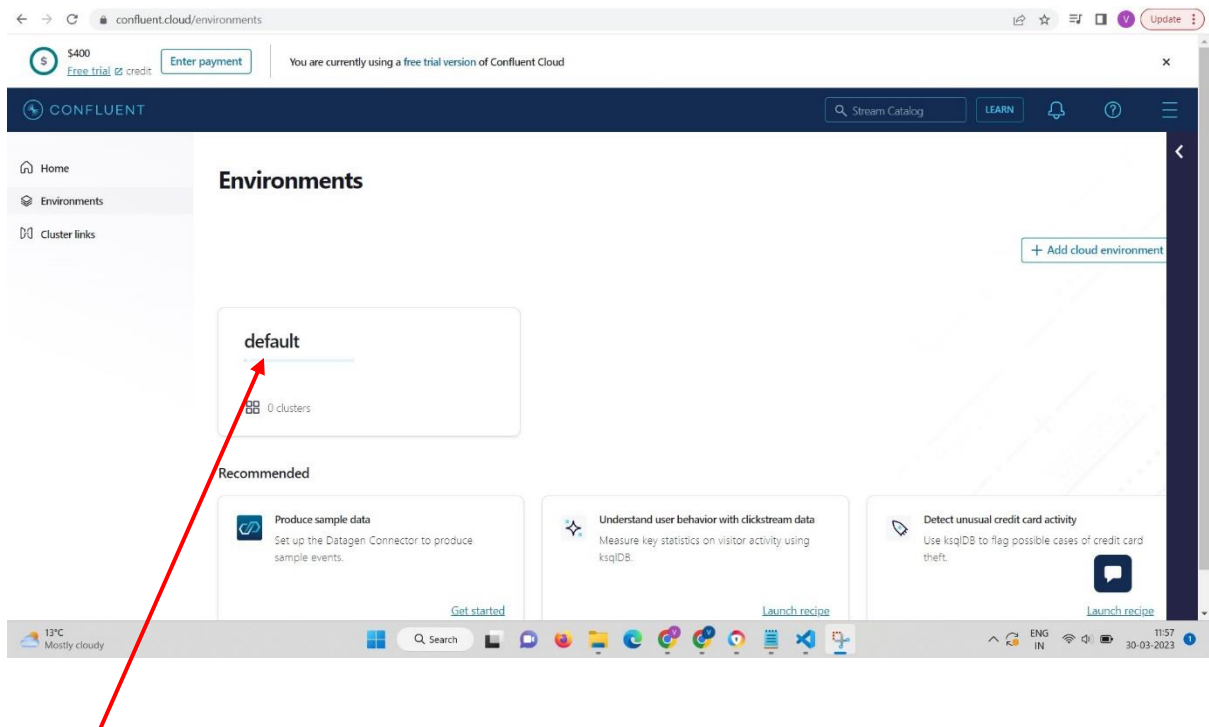


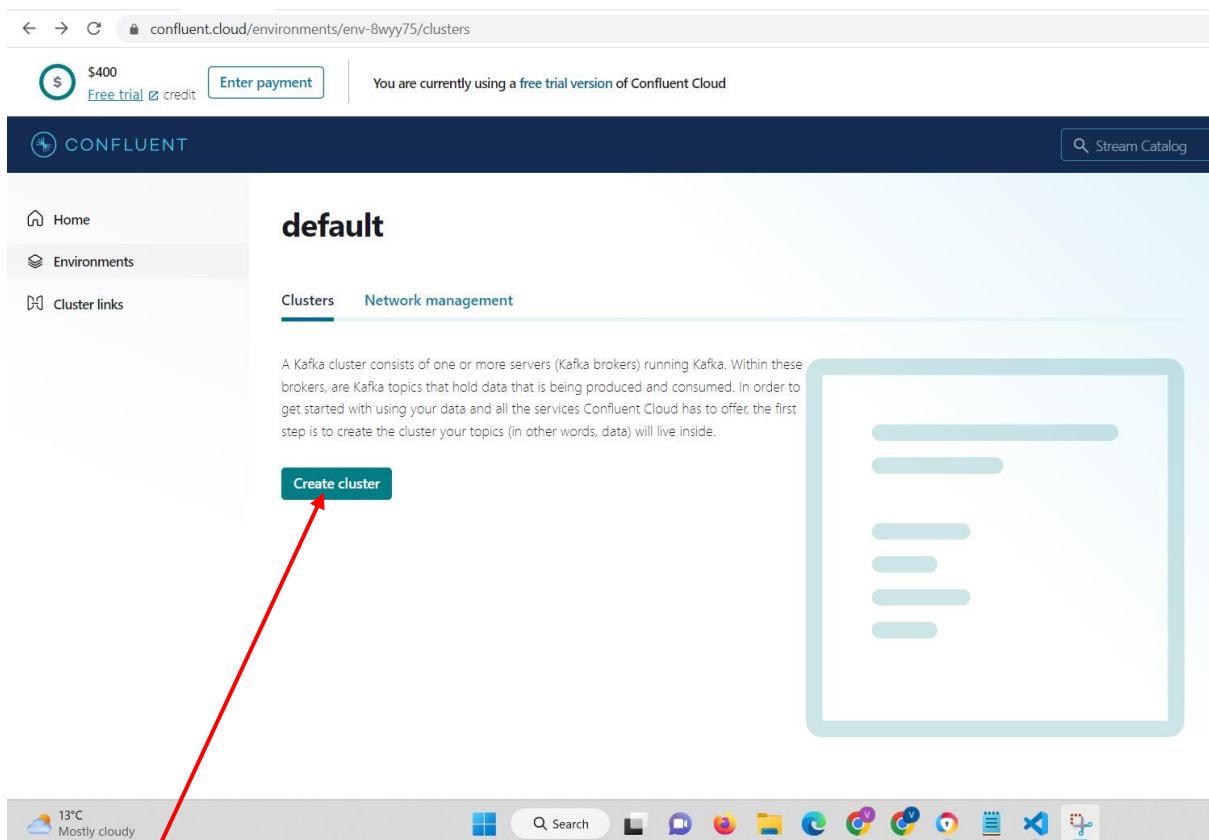
## Kafka code 1:



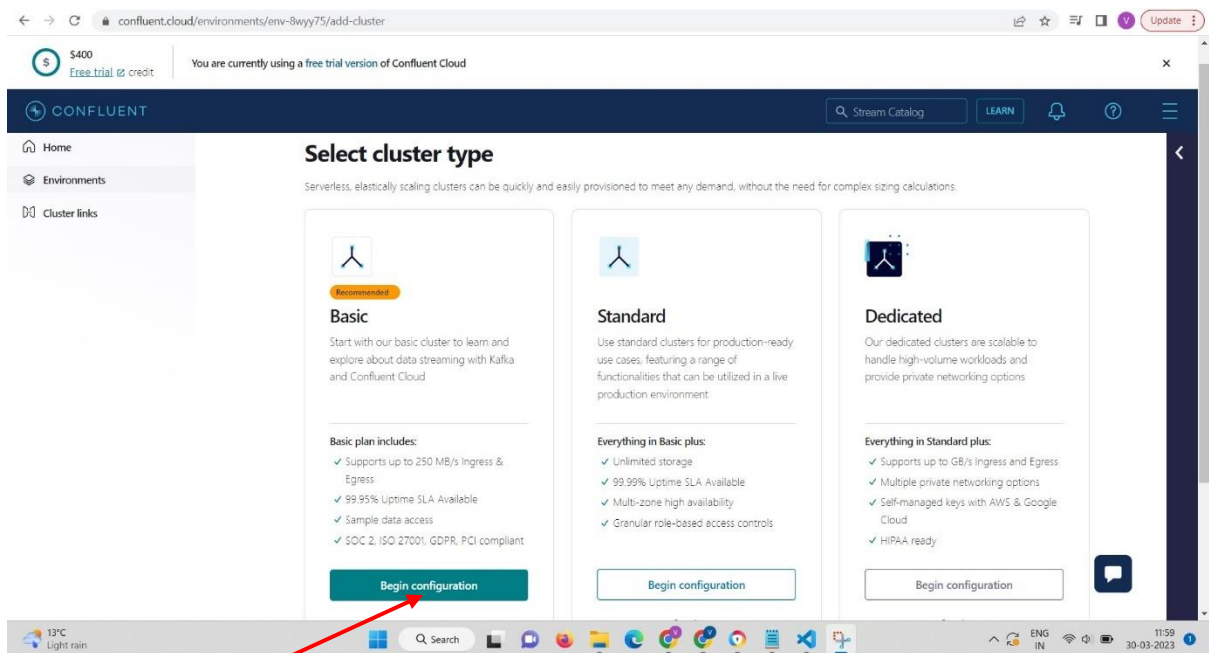
Click on view environments.



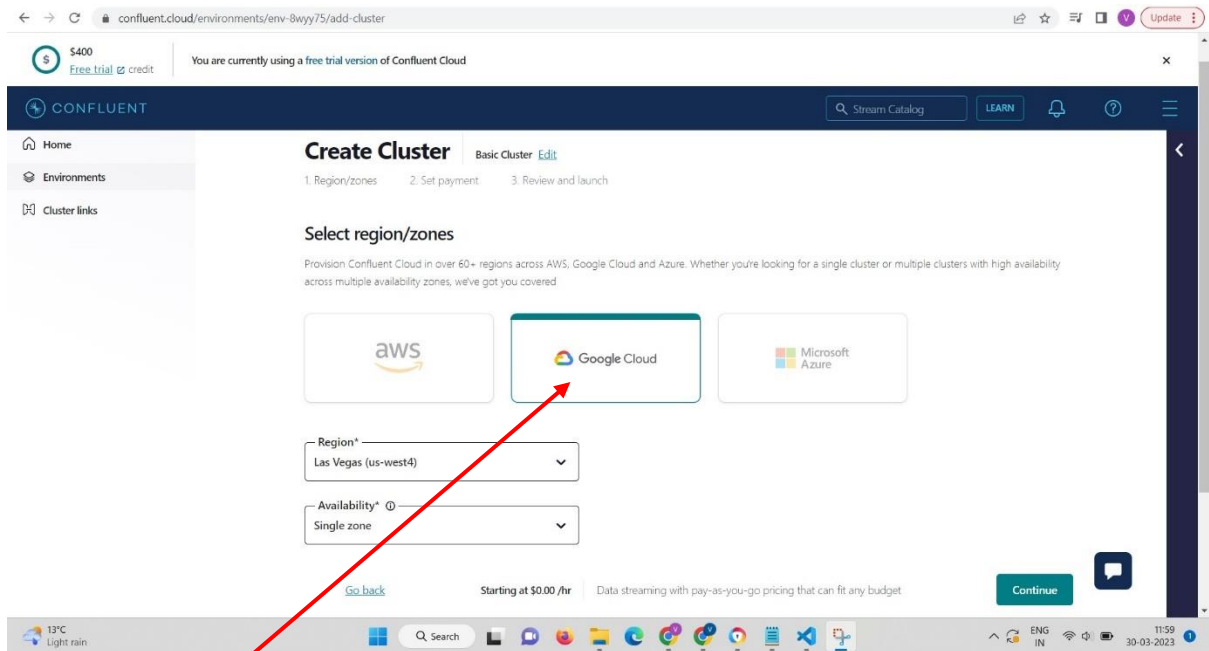
Click on default.



Click on create cluster.

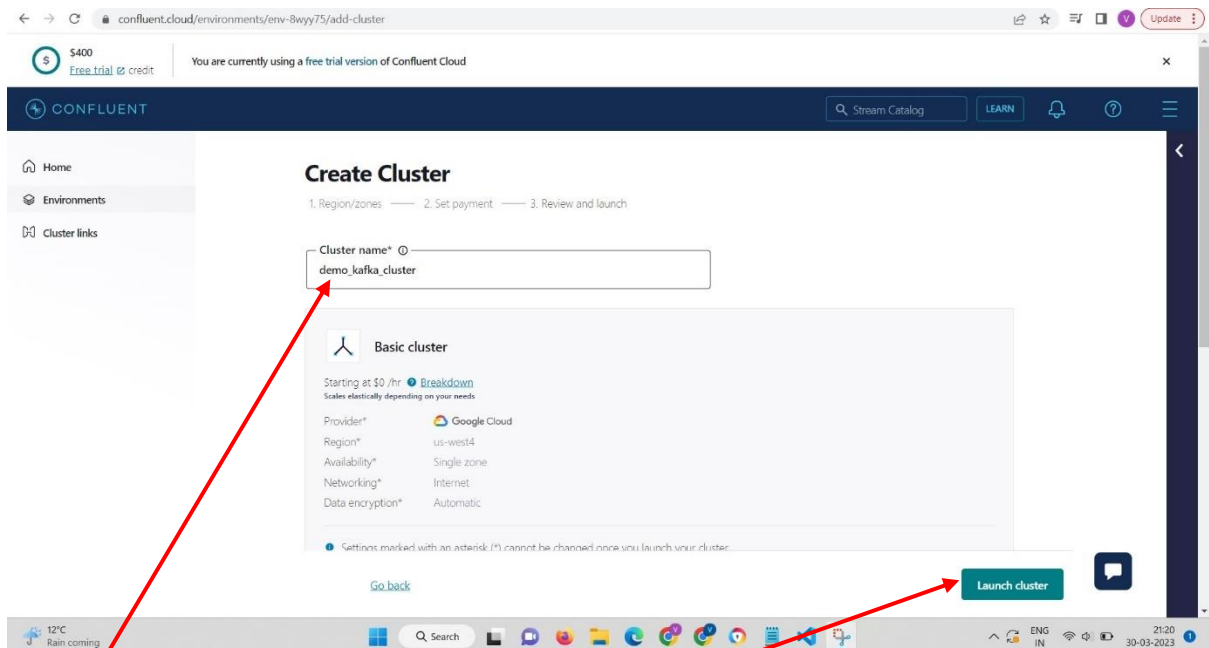


Click on begin configuration.

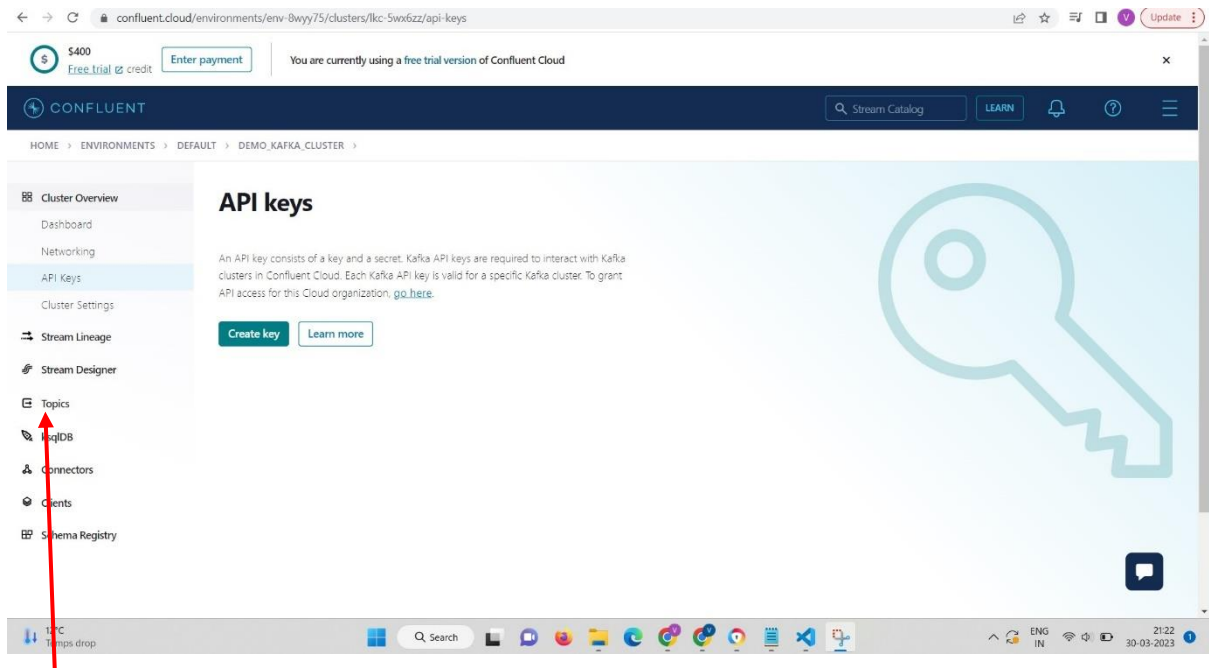


Click on Google Cloud.

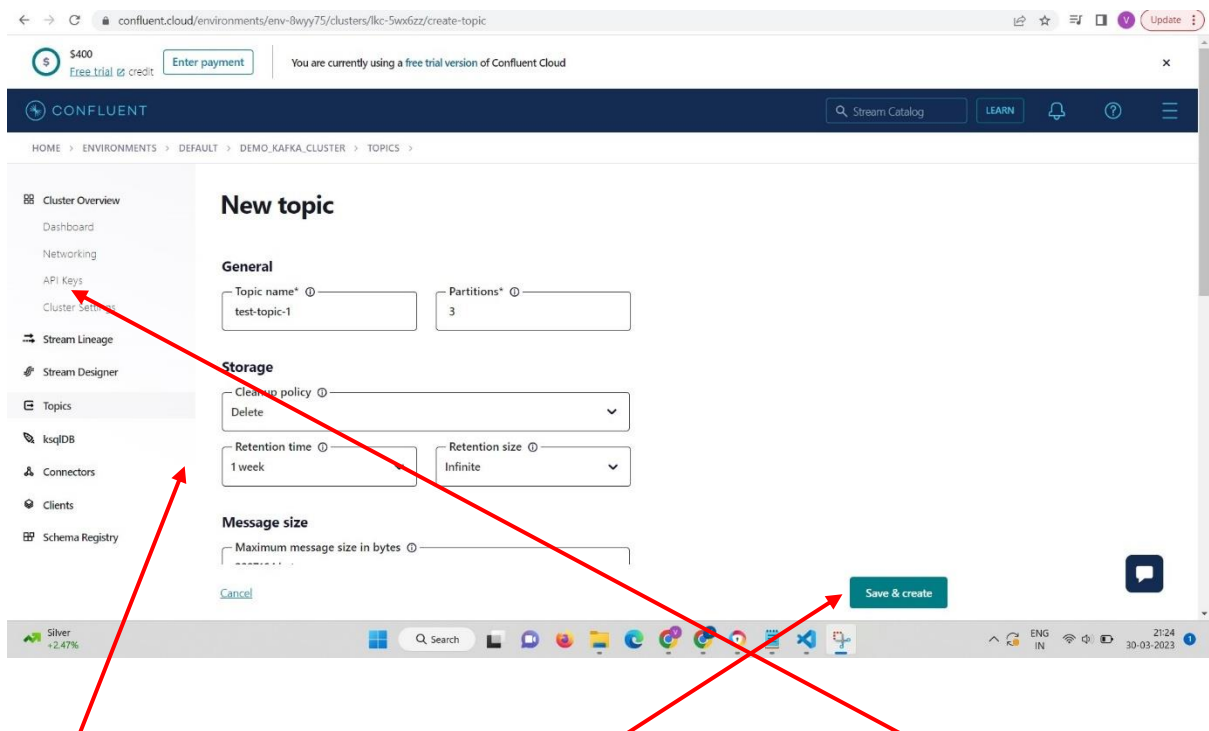
Note: Any cloud platform can be used.



Edit the cluster name and click on the launch cluster.

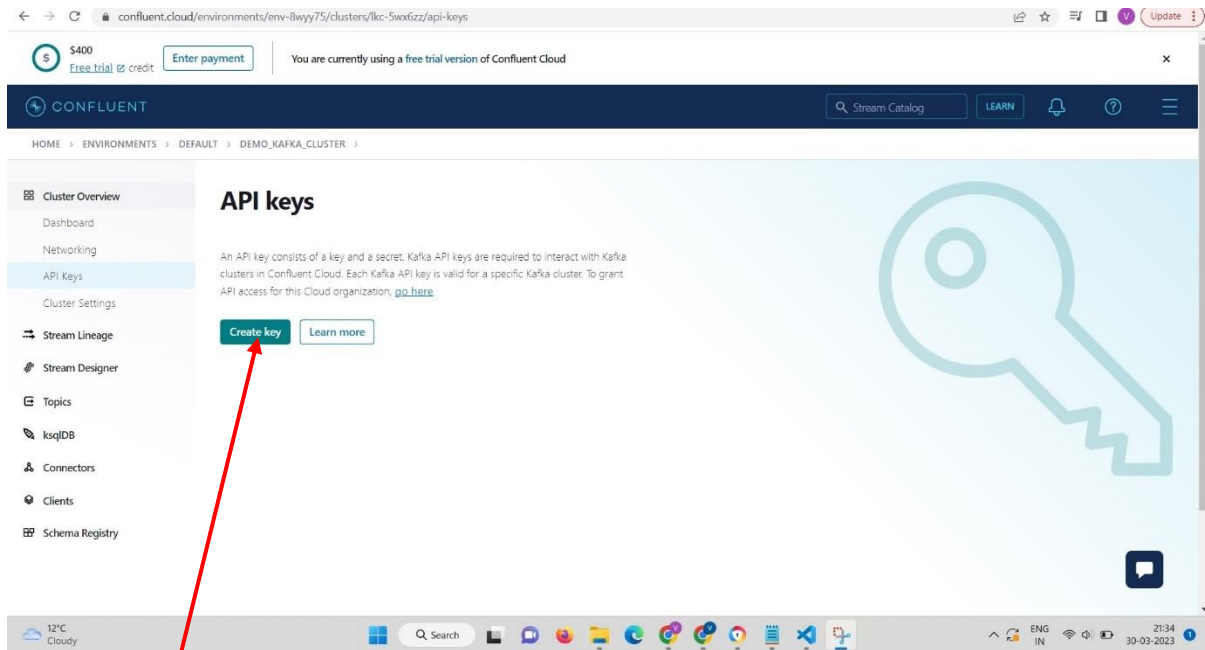


Click on the topics.

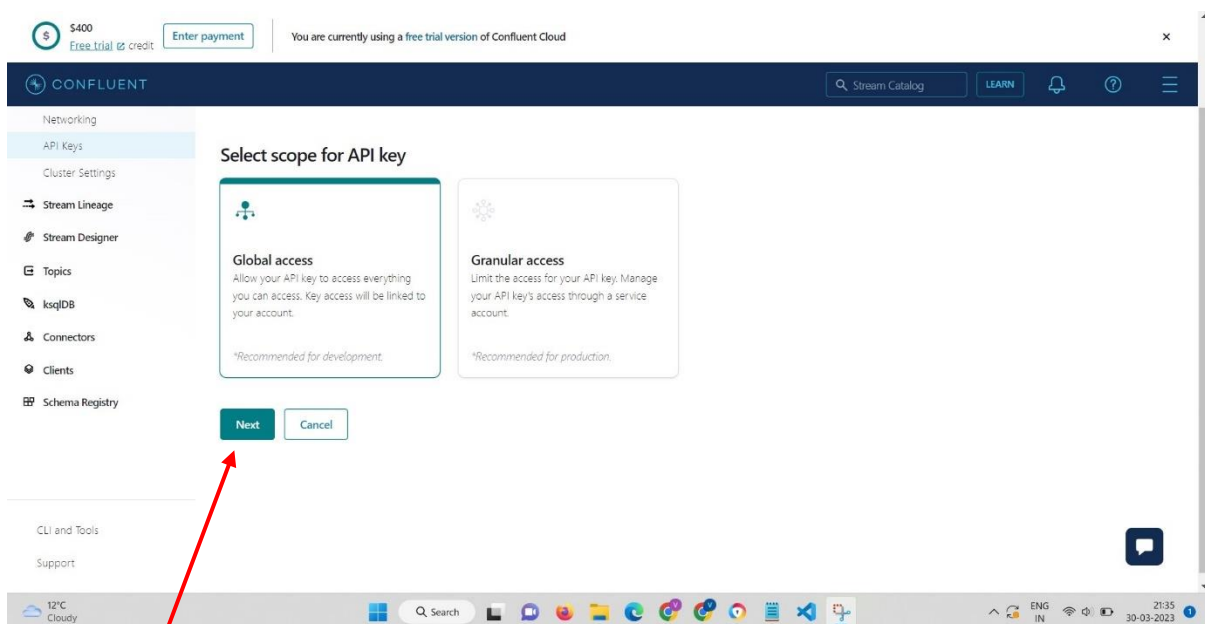


Change the topic configurations. Then click on save & create. The click on API keys.

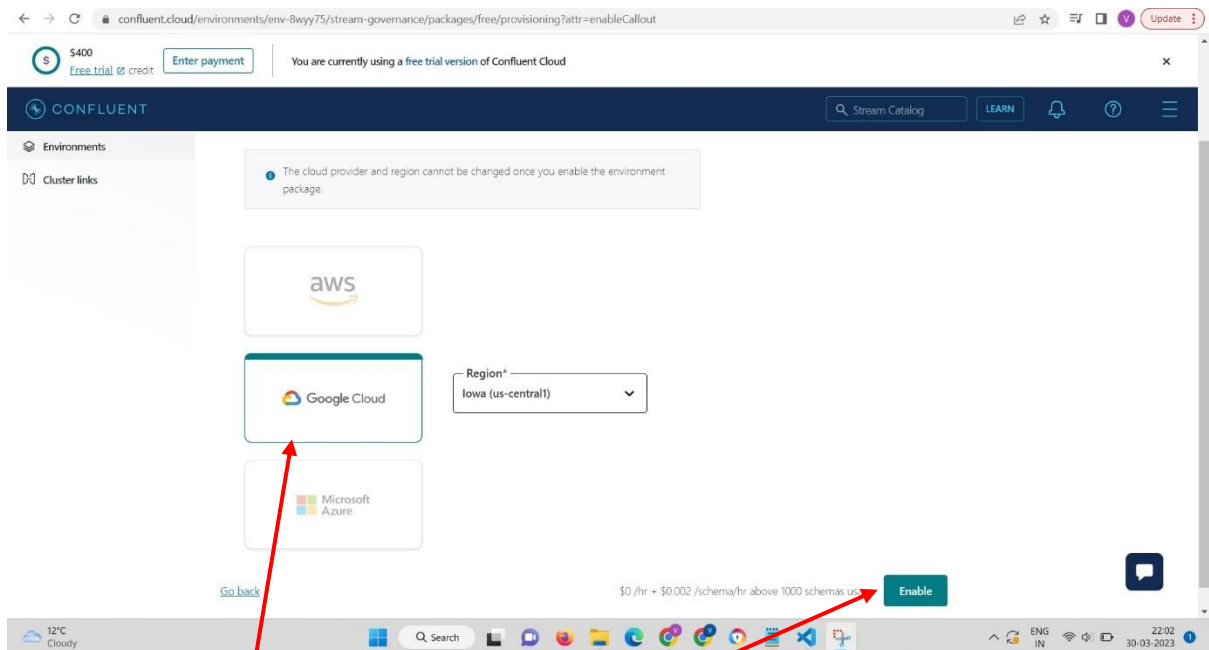
Note: In the program the name is test\_topic.



Click on create key.

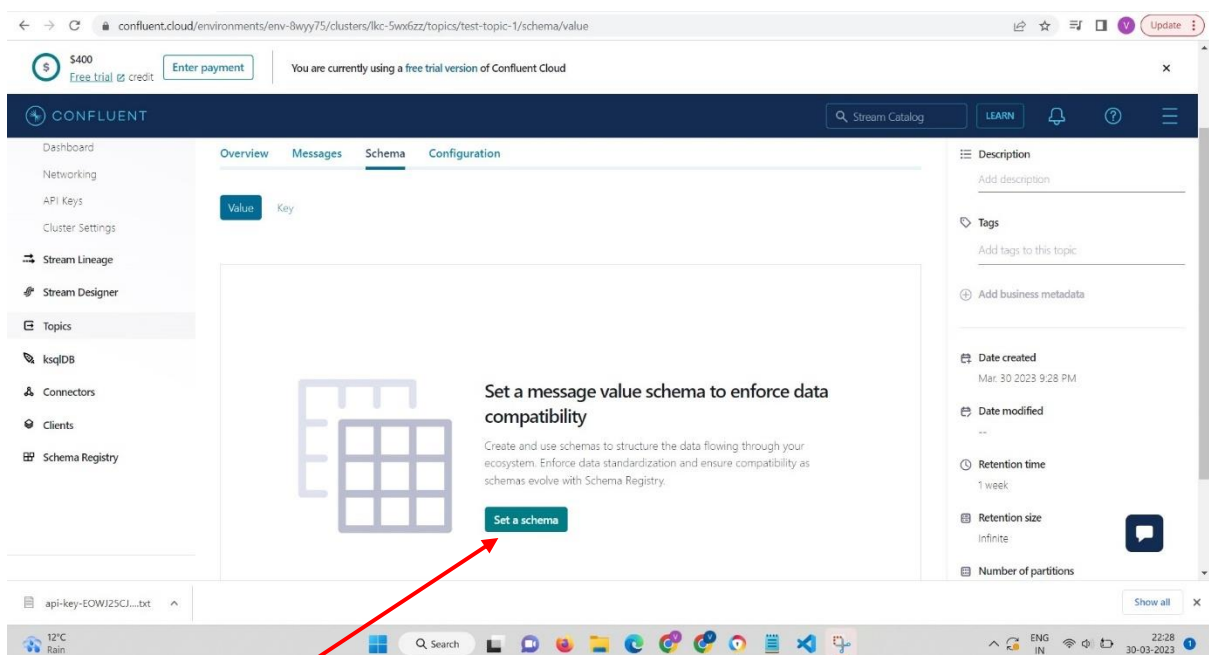


Select Global access and click on Next. After this api keys will be displayed. In that write description (kafka access keys) this description can be anything. Then click on download and continue.



Click on the Google Cloud. Then click on enable.

Note: Choice of cloud platform can be anything.



Click on Set a schema for value side. Then click on the JSON Schema and copy paste the below code:

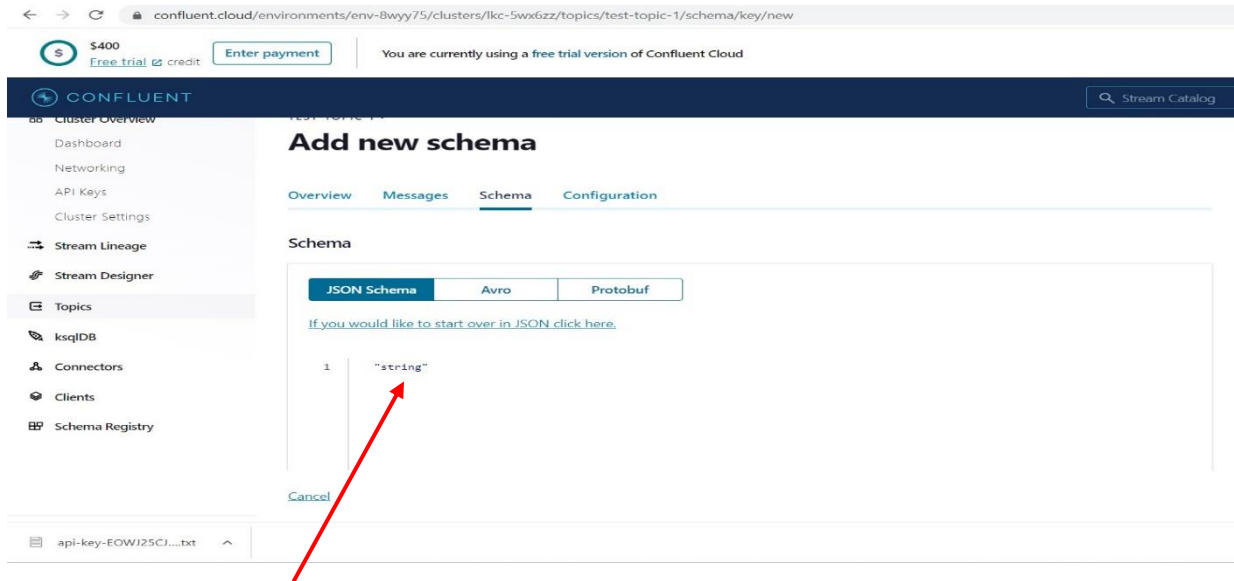
```
{
  "$id": "http://example.com/myURI.schema.json",
  "$schema": "http://json-schema.org/draft-07/schema#",
  "additionalProperties": false,
  "description": "Sample schema to help you get started.",
  "properties": {
    "brand": {
      "description": "The type(v) type is used.",
      "type": "string"
    },
    "car_name": {
      "description": "The type(v) type is used.",
      "type": "string"
    },
    "engine": {
      "description": "The type(v) type is used.",
      "type": "number"
    },
    "fuel_type": {
      "description": "The type(v) type is used.",
      "type": "string"
    },
    "km_driven": {
      "description": "The type(v) type is used.",
      "type": "number"
    },
    "max_power": {
      "description": "The type(v) type is used.",
      "type": "number"
    },
    "mileage": {
      "description": "The type(v) type is used.",
      "type": "number"
    },
    "model": {
      "description": "The type(v) type is used.",
      "type": "string"
    },
    "seats": {
      "description": "The type(v) type is used.",
      "type": "number"
    }
  }
}
```

```
    },
    "seller_type": {
      "description": "The type(v) type is used.",
      "type": "string"
    },
    "selling_price": {
      "description": "The type(v) type is used.",
      "type": "number"
    },
    "transmission_type": {
      "description": "The type(v) type is used.",
      "type": "string"
    },
    "vehicle_age": {
      "description": "The type(v) type is used.",
      "type": "number"
    }
  },
  "title": "SampleRecord",
  "type": "object"
}
```

#### Code explanation:

This is a JSON schema that defines the structure and data types of a data record for a car selling platform. The schema includes a title, description, and additional Properties set to false, which means that only the properties explicitly defined in the schema can be present in a data record, and no additional properties are allowed. The schema defines an object with 12 properties, including "brand", "car\_name", "engine", "fuel\_type", "km\_driven", "max\_power", "mileage", "model", "seats", "seller\_type", "selling\_price", "transmission\_type", and "vehicle\_age". Each property has a description that provides information about the data type expected, such as "string" or "number". Overall, this schema can be used to validate and ensure the consistency of data records for a car selling platform, ensuring that the data conforms to a specific structure and data types.





For key side type “string” in the JSON Schema.

Install following python packages in the terminal.

1. pip3 install confluent\_kafka
2. pip3 install pandas
3. pip3 install requests
4. pip3 install jsonschema

Python producer code:

```
#!/usr/bin/env python
# -*- coding: utf-8 -*-
#
# Copyright 2020 Confluent Inc.
#
# Licensed under the Apache License, Version 2.0 (the "License");
# you may not use this file except in compliance with the License.
# You may obtain a copy of the License at
#
# http://www.apache.org/licenses/LICENSE-2.0
#
# Unless required by applicable law or agreed to in writing, software
# distributed under the License is distributed on an "AS IS" BASIS,
# WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied.
# See the License for the specific language governing permissions and
# limitations under the License.
#
# A simple example demonstrating use of JsonSerializer.
```

```

import argparse
from uuid import uuid4
from six.moves import input
from confluent_kafka import Producer
from confluent_kafka.serialization import StringSerializer,
SerializationContext, MessageField
from confluent_kafka.schema_registry import SchemaRegistryClient
from confluent_kafka.schema_registry.json_schema import JSONSerializer
#from confluent_kafka.schema_registry import *
import pandas as pd
from typing import List

FILE_PATH = "your file path for the dataset"
columns=['car_name', 'brand', 'model', 'vehicle_age', 'km_driven',
'seller_type',
        'fuel_type', 'transmission_type', 'mileage', 'engine', 'max_power',
        'seats', 'selling_price']

API_KEY = 'your api key'
ENDPOINT_SCHEMA_URL = 'your endpoint schema url'
API_SECRET_KEY = 'your api secret key'
BOOTSTRAP_SERVER = 'your bootstrap server'
SECURITY_PROTOCOL = 'SASL_SSL'
SSL_MACHENISM = 'PLAIN'
SCHEMA_REGISTRY_API_KEY = 'your schema registry api key'
SCHEMA_REGISTRY_API_SECRET = 'your schema registry api secret'

def sasl_conf():

    sasl_conf = {'sasl.mechanism': SSL_MACHENISM,
                  # Set to SASL_SSL to enable TLS support.
                  # 'security.protocol': 'SASL_PLAINTEXT'}
                  'bootstrap.servers':BOOTSTRAP_SERVER,
                  'security.protocol': SECURITY_PROTOCOL,
                  'sasl.username': API_KEY,
                  'sasl.password': API_SECRET_KEY
                }
    return sasl_conf

def schema_config():
    return {'url':ENDPOINT_SCHEMA_URL,

            'basic.auth.user.info':f"{SCHEMA_REGISTRY_API_KEY}:{SCHEMA_REGISTRY_API_SE
CRET}"

```

```

    }

class Car:
    def __init__(self, record: dict):
        for k, v in record.items():
            setattr(self, k, v)

        self.record = record

    @staticmethod
    def dict_to_car(data: dict, ctx):
        return Car(record=data)

    def __str__(self):
        return f"{self.record}"

def get_car_instance(file_path):
    df = pd.read_csv(file_path)
    df = df.iloc[:, 1:]
    cars: List[Car] = []
    for data in df.values:
        car = Car(dict(zip(columns, data)))
        cars.append(car)
    yield car

def car_to_dict(car: Car, ctx):
    """
    Returns a dict representation of a User instance for serialization.
    Args:
        user (User): User instance.
        ctx (SerializationContext): Metadata pertaining to the serialization
            operation.
    Returns:
        dict: Dict populated with user attributes to be serialized.
    """

    # User._address must not be serialized; omit from dict
    return car.record

def delivery_report(err, msg):
    """
    Reports the success or failure of a message delivery.
    Args:
        err (KafkaError): The error that occurred on None on success.
        msg (Message): The message that was produced or failed.
    """

```

```

    if err is not None:
        print("Delivery failed for User record {}: {}".format(msg.key(), err))
        return
    print('User record {} successfully produced to {} [{}] at offset
    {}'.format(
        msg.key(), msg.topic(), msg.partition(), msg.offset()))

def main(topic):

    schema_str = """
    {
    "$id": "http://example.com/myURI.schema.json",
    "$schema": "http://json-schema.org/draft-07/schema#",
    "additionalProperties": false,
    "description": "Sample schema to help you get started.",
    "properties": {
        "brand": {
            "description": "The type(v) type is used.",
            "type": "string"
        },
        "car_name": {
            "description": "The type(v) type is used.",
            "type": "string"
        },
        "engine": {
            "description": "The type(v) type is used.",
            "type": "number"
        },
        "fuel_type": {
            "description": "The type(v) type is used.",
            "type": "string"
        },
        "km_driven": {
            "description": "The type(v) type is used.",
            "type": "number"
        },
        "max_power": {
            "description": "The type(v) type is used.",
            "type": "number"
        },
        "mileage": {
            "description": "The type(v) type is used.",
            "type": "number"
        },
        "model": {
            "description": "The type(v) type is used.",
            "type": "string"

```



```

except KeyboardInterrupt:
    pass
except ValueError:
    print("Invalid input, discarding record...")
    pass

print("\nFlushing records...")
producer.flush()

main("test_topic")

```

This is a Python script that demonstrates the use of Confluent Kafka Python library for producing JSON-serialized messages to a Kafka topic. The code is specifically designed for a sample dataset from Cardekho, where each record represents a car for sale. The script starts by defining the required configuration settings for Kafka, including API keys, endpoint URLs, and security protocols. It then defines a Car class that represents a single record in the Cardekho dataset. The `get_car_instance` function reads the dataset from a CSV file, creates instances of the Car class for each record, and yields them for use in producing messages. The `car_to_dict` function is used to serialize instances of the Car class into a JSON-serializable dictionary format. This function is used by the JSON Serializer object to convert Python objects into JSON-formatted strings that can be sent to Kafka. The main function initializes the Kafka producer, creates an instance of the JSON Serializer, and produces messages to a specified Kafka topic. The `delivery_report` function is used to report the success or failure of message delivery. Overall, this code demonstrates how to use the Confluent Kafka Python library to produce JSON-serialized messages to a Kafka topic, using a schema and serialization configuration provided by the Confluent Schema Registry.

Python consumer code:

```

import argparse

from confluent_kafka import Consumer
from confluent_kafka.serialization import SerializationContext, MessageField
from confluent_kafka.schema_registry.json_schema import JSONDeserializer

API_KEY = 'your api key'
ENDPOINT_SCHEMA_URL = 'your endpoint schema url'
API_SECRET_KEY = 'your api secret key'
BOOTSTRAP_SERVER = 'your bootstrap server'
SECURITY_PROTOCOL = 'SASL_SSL'
SSL_MACHENISM = 'PLAIN'
SCHEMA_REGISTRY_API_KEY = 'your schema registry api key'
SCHEMA_REGISTRY_API_SECRET = 'your schema registry api secret'

```

```

def sasl_conf():

    sasl_conf = {'sasl.mechanism': SSL_MACHENISM,
                  # Set to SASL_SSL to enable TLS support.
                  # 'security.protocol': 'SASL_PLAINTEXT'}
                  'bootstrap.servers':BOOTSTRAP_SERVER,
                  'security.protocol': SECURITY_PROTOCOL,
                  'sasl.username': API_KEY,
                  'sasl.password': API_SECRET_KEY
                  }
    return sasl_conf


def schema_config():
    return {'url':ENDPOINT_SCHEMA_URL,

            'basic.auth.user.info':f"{SCHEMA_REGISTRY_API_KEY}:{SCHEMA_REGISTRY_API_SE
CRET}"

            }


class Car:
    def __init__(self,record:dict):
        for k,v in record.items():
            setattr(self,k,v)

        self.record=record

    @staticmethod
    def dict_to_car(data:dict,ctx):
        return Car(record=data)

    def __str__(self):
        return f"{self.record}"


def main(topic):

    schema_str = """
    {
    "$id": "http://example.com/myURI.schema.json",
    "$schema": "http://json-schema.org/draft-07/schema#",
    "additionalProperties": false,
    "description": "Sample schema to help you get started.",
    "properties": {

```

```
"brand": {
  "description": "The type(v) type is used.",
  "type": "string"
},
"car_name": {
  "description": "The type(v) type is used.",
  "type": "string"
},
"engine": {
  "description": "The type(v) type is used.",
  "type": "number"
},
"fuel_type": {
  "description": "The type(v) type is used.",
  "type": "string"
},
"km_driven": {
  "description": "The type(v) type is used.",
  "type": "number"
},
"max_power": {
  "description": "The type(v) type is used.",
  "type": "number"
},
"mileage": {
  "description": "The type(v) type is used.",
  "type": "number"
},
"model": {
  "description": "The type(v) type is used.",
  "type": "string"
},
"seats": {
  "description": "The type(v) type is used.",
  "type": "number"
},
"seller_type": {
  "description": "The type(v) type is used.",
  "type": "string"
},
"selling_price": {
  "description": "The type(v) type is used.",
  "type": "number"
},
"transmission_type": {
  "description": "The type(v) type is used.",
  "type": "string"
},
},
```



```

        "vehicle_age": {
            "description": "The type(v) type is used.",
            "type": "number"
        }
    },
    "title": "SampleRecord",
    "type": "object"
}

"""
json_deserializer = JSONDeserializer(schema_str,
                                     from_dict=Car.dict_to_car)

consumer_conf = sasl_conf()
consumer_conf.update({
    'group.id': 'group1',
    'auto.offset.reset': 'latest'})

consumer = Consumer(consumer_conf)
consumer.subscribe([topic])

while True:
    try:
        # SIGINT can't be handled when polling, limit timeout to 1 second.
        msg = consumer.poll(1.0)
        if msg is None:
            continue

        car = json_deserializer(msg.value(),
                                SerializationContext(msg.topic(), MessageField.VALUE))

        if car is not None:
            print("User record {}: car: {}\n"
                  .format(msg.key(), car))
    except KeyboardInterrupt:
        break

consumer.close()

main("test_topic")

```

Code explanation:

This code defines a Kafka consumer that reads messages from a Kafka topic named "test\_topic" and deserializes them using a JSON Deserializer. The JSON Deserializer is created using a JSON schema string that defines the schema of the message payload. The JSON schema defines the properties of a Car object, such as brand, car name, engine, fuel type, km driven, max power, mileage, model, seats, seller type, selling price, transmission type, and vehicle age. The

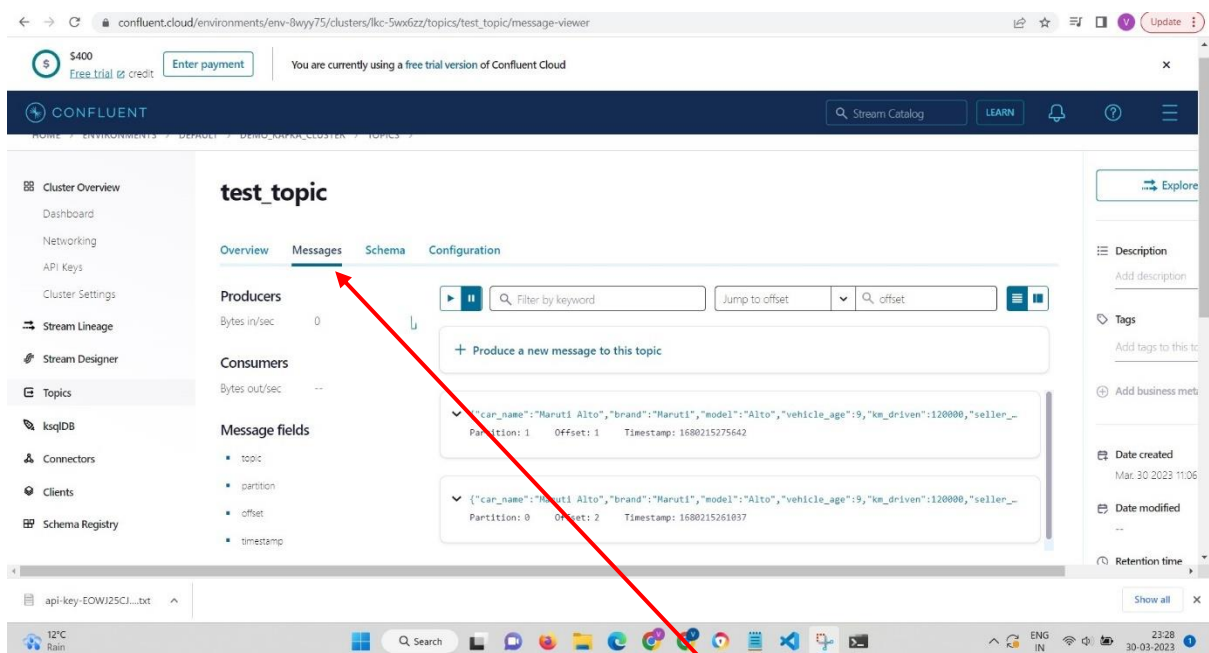
deserialized Car object is then printed to the console. The Kafka consumer is configured with security settings and subscribes to the specified topic. The consumer will keep running until interrupted by the user. When a message is received, it is deserialized and the Car object is printed to the console.

Run following code on terminal for producer side

```
python "your path for the producer code"
```

Run following code on terminal for consumer side

```
python "your path for the consumer code"
```



After running both the codes in the terminal click on the messages.