CONFLUENT

Grainger GTG 2023 -

Confluent Cloud Disaster Recovery Workshop

This workshop will familiarize you with the tools Confluent provides its customers to enable data streaming and geo-replication across regions through Cluster Linking. During the workshop you'll create topics, produce and consume messages from an Apache Kafka® cluster, and see how Cluster Linking can be used to replicate data seamlessly to any other Confluent Cloud cluster in the world.

Confluent Cloud is a resilient, scalable data streaming service based on Apache Kafka® and delivered as a fully managed service. Customers interact with Confluent Cloud through a web interface, a local command line interface, or through REST API calls. In this workshop we will primarily interact with the Confluent Cloud clusters via the command line.
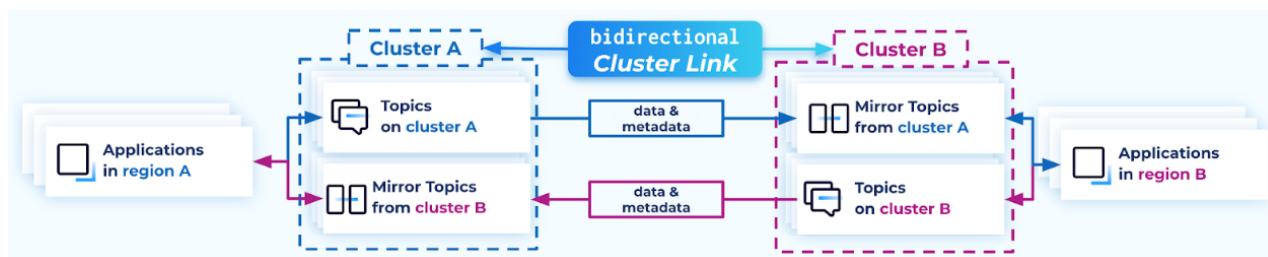
Prerequisites

The Confluent Cloud clusters to be used during this workshop are available via the internet (i.e. secure public endpoints) and some resources will be shared to minimize the complexity to work with the platform during this workshop. In a production environment additional private IP and authentication safeguards would be enabled. If you have trouble accessing the cloud environment or any steps in the lab please try disabling your organization's VPN (if one is in use). If problems persist please let someone from Confluent know to help troubleshoot.

Workshop files (including the python producer, python consumer and this workbook) are available via github. To download:

```
git clone https://github.com/ctechter/Grainger_GTG_2023.git
```
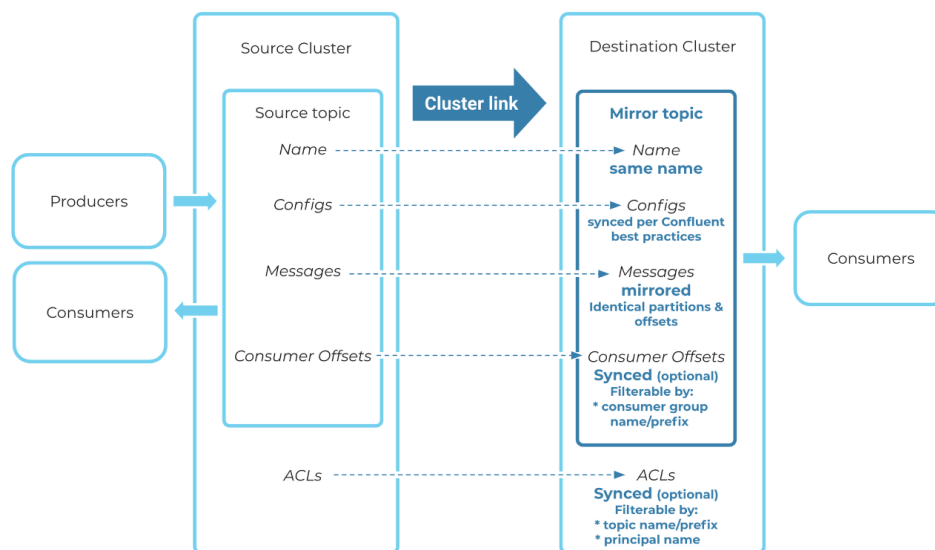
Overview

During this workshop we're going to briefly work through how Confluent Cloud enables both active-passive and active-active architectures through the use of Cluster Linking, allowing both data and metadata to flow **bi-directionally** between two or more clusters. A high-level topology of bi-directional cluster linking is below

CONFLUENT

bidirectional *Cluster Link*

Cluster A

Topics on cluster A

data & metadata

Mirror Topics from cluster A

Cluster B

Applications in region A

Mirror Topics from cluster B

data & metadata

Topics on cluster B

Applications in region B

For the purposes of this workshop we're going to simulate an active/passive Disaster Recovery scenario where Cluster Linking is used to move data only in one direction (from the "Source" cluster to the "Destination" cluster. With bi-directional cluster linking there is no Source or Destination cluster; both are treated as equals.

bidirectional *Cluster Link*

Cluster A

local = cluster A

remote = cluster B

Cluster B

local = cluster B

remote = cluster A
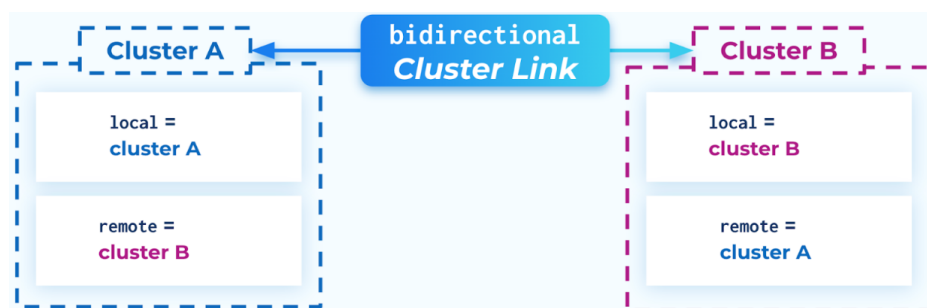
One of the key differentiators of Cluster Linking is the ability to not only replicate data "byte-for-byte" but also keep consumer offsets and/or ACLs in sync between clusters.

Source Cluster

Source topic

Cluster link

Destination Cluster

Mirror topic

Producers

Consumers

Name

Configs

Messages

Consumer Offsets

ACLs

Name **same name**

Configs **synced per Confluent best practices**

Messages **mirrored Identical partitions & offsets**

Consumer Offsets **Synced** (optional) **Filterable by:** * consumer group name/prefix

ACLs **Synced** (optional) **Filterable by:** * topic name/prefix * principal name

Consumers

Step 1 - Login and Create a Topic

1. Sign into Confluent Cloud at https://confluent.cloud
   a. Login Credentials:
   i. email: ctechter+gtg@confluent.io

ii. Password: GtG#2023$!

b. ***PLEASE NOTE*** all participants in this workshop will use a single shared account to access the UI (Please be kind to your fellow workshop participants!), environments, etc. To minimize any duplication of asset names please follow the naming convention outlined in the workshop steps

2. Click into the tile marked "Confluent Cloud" and navigate to the environment name "Grainger_GTG_Primary"
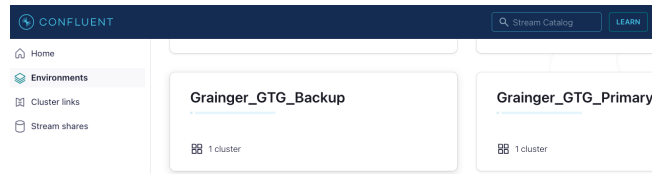
3. Click on the only cluster in the environment titled "GTG_Primary"

4. From here select "Topics" from the left-side menu and then "Add Topic"
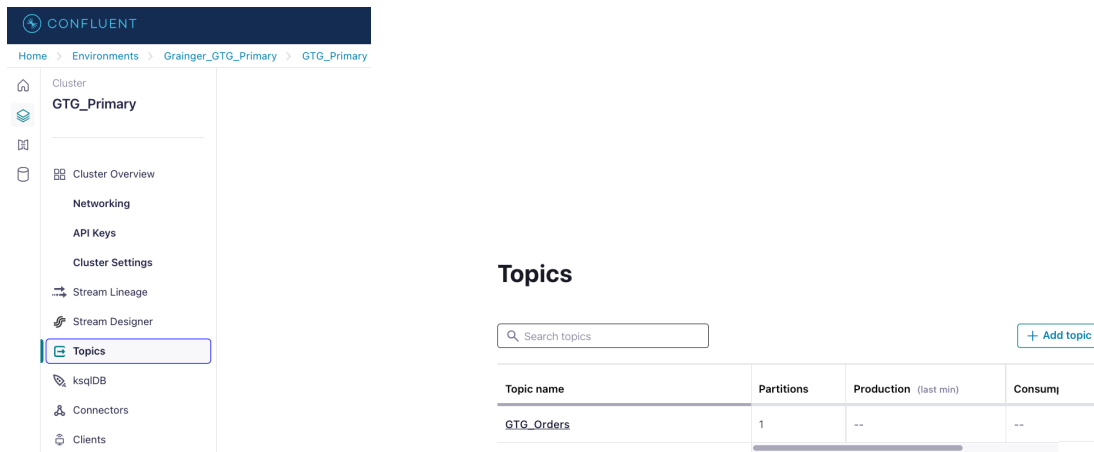
5. Create the topic with the following settings:

a. Name: "***GTG_Workshop_[XXX]***". Replace "XXX" with your initials or some unique combination of letters and numbers (for example the instructor will be creating a topic named "***GTG_Workshop_CWT***")

b. Partitions: Change from the default of 6 to 1.

c. Click on "Create with defaults" to create the topic.

i. NOTE: If you would like to review all the available options for topic creation click on "Show advanced settings," however please don't change any of the default configurations.

d. When prompted to define a data contract select "Skip"

Step 2 - Log into CLI and create API keys for primary and backup clusters

Now that our topic is created let's log into the Confluent CLI and create an API key through the command line. We will then use this API key to run the python producers and consumers to send/receive data through the Confluent Kafka® cluster. An overview of the CLI can be found at:

https://docs.confluent.io/confluent-cli/current/overview.html

1. To download and install the latest version in the default directory (./bin) run
   a. `curl -sL --http1.1 https://cnfl.io/cli | sh -s -- latest`
2. For convenience add the ./bin directory to your $PATH:
   a. `export PATH=$(pwd)/bin:$PATH`
3. Next we'll log in and enter the same credentials used to log into the UI listed in Step 1 above
   a. `confluent login`
      i. NOTE: if you want to save the credentials for non-interactive re-authentication you can use the --save flag)
   b. Enter in the email address and password listed in Step 1 above
4. Once logged in our next step is to navigate to the GTG Primary environment and cluster (there are many different environments and clusters in this organization as this is a Confluent technical sales sandbox environment). To navigate to the GTG Primary environment execute the following commands through the CLI:
   a. Search for the environment ID using the following command:
      i. `confluent environment list`

```
Grainger_GTG_2023: confluent environment list
  Current |     ID      |                  Name
 ---------+-------------+-------------------------------------------
    *      | env-09q50q | Grainger_GTG_Primary
           | env-2rp5mm | Grainger_GTG_Backup
           | env-3n860  | Keith
           | env-3rd2qo | Sandbox
           | env-3rjrj2 | Wolter
           | env-6pr73  | jrolfe
           | env-gqv07v | gdappili
           | env-knxxpm | Kafcongo
           | env-m8k10x | charmon
           | env-m8p10q | yemalin
           | env-mvp30q | Rolfe-DND-environment
           | env-njkv3  | smaripadaga
           | env-nw0g23 | Ammar
           | env-nz3x6  | showard
           | env-o0z2o  | kkind
           | env-o5pqo  | bart
           | env-p6r85  | ctechter
           | env-qr5vnp | ccloud-stack-sa-y6x686-kafkageodemo
           | env-y35jo  | Britton
           | env-yon9go | pv-sa-8mz8w0-dnd
           | env-yovp7j | prodtest
           | env-zg9xyd | Staging
Grainger_GTG_2023:
```

b. Search for the ID next to the name **"Grainger_GTG_Primary"** to enter into the following command:

    i. `confluent environment use env-09q50q`

c. Typing `confluent environment list` again will then show an asterisk next to **"Grainger_GTG_Primary"** indicating the command line is using that environment.

5. Next we need to select the cluster we'll be using (in this case there is only one cluster in the environment). Execute the following commands

    a. `confluent kafka cluster list` (only one cluster should be returned)

    b. `confluent kafka cluster use lkc-0jvoy6`

    c. Just like the environment commands executing `confluent kafka cluster list` again will show an asterisk next to the cluster name indicating the CLI is executing further commands against this cluster.

```
Grainger_GTG_2023: confluent kafka cluster list
  Current |     ID     |     Name    |   Type    | Provider |  Region   | Availability | Status
 ---------+------------+-------------+-----------+----------+-----------+--------------+---------
    *      | lkc-0jvoy6 | GTG_Primary | DEDICATED | aws      | us-east-1 | single-zone  | UP
Grainger_GTG_2023: confluent kafka cluster use lkc-0jvoy6
Set Kafka cluster "lkc-0jvoy6" as the active cluster for environment "env-09q50q".
```

6. Next we'll be creating two API keys for use in the rest of the workshop exercises (one each for the primary and backup clusters. A few things of note:

- **BE SURE TO RECORD THE API KEY AND SECRET** when it's generated. A secret cannot be retrieved from the system once generated; if the secret is lost the key must be deleted and a new key /secret generated

- For simplicity we'll be creating a key that has full permissions within the Confluent Cloud kafka® cluster. There are many security options available to ensure proper resource

access and control, the setup and configuration of which are outside the scope of this workshop.

a. To create the api key/secret in the Primary cluster:

　　i. Be sure you have confirmed you're working in the "***Grainger_GTG_Primary***" environment and cluster. Be sure to note the cluster ID.

```
Grainger_GTG_2023: confluent environment use env-09q50q
Using environment "env-09q50q".
Grainger_GTG_2023: confluent kafka cluster list
  Current |    ID     |    Name     |   Type    | Provider |  Region   |  Availability | Status
----------+-----------+-------------+-----------+----------+-----------+---------------+--------
    *      | lkc-0jvoy6 | GTG_Primary | DEDICATED | aws      | us-east-1 | single-zone   | UP
Grainger_GTG_2023:
```

　　ii. Create the API key/secret by using the following command. The `--resource lkc-0jvoy9` flag is used to provide full access of the cluster to the API key.

　　　　a. `confluent api-key create --resource lkc-0jvoy9`

　　iii. The command line will return the API key and secret. ***MAKE SURE TO RECORD THESE VALUES AS THE SECRET CANNOT BE RETRIEVED AGAIN***.
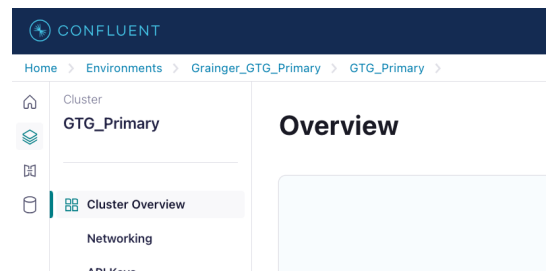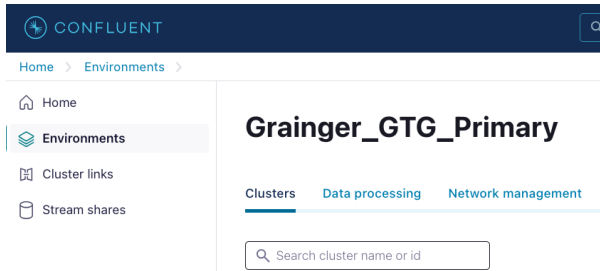
```
Grainger_GTG_2023: confluent api-key create --resource lkc-0jvoy6
It may take a couple of minutes for the API key to be ready.
Save the API key and secret. The secret is not retrievable later.
+-----------+--------------------------------------------------------------------+
| API Key   | 7                                    N                             |
| API Secret | X                                                           /f |
+-----------+--------------------------------------------------------------------+
```
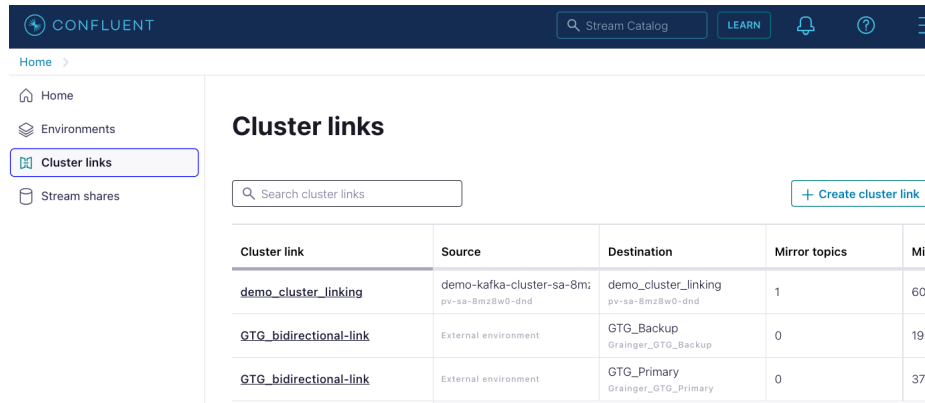
b. To create the api key/secret in the DR cluster:

　　i. Switch over to the DR environment and cluster (you can retrieve the environment ID from the command you executed earlier) and then create the API key/secret using the same set of commands when creating the primary cluster.

　　　　1. `confluent environment use env-2rp5mm`
　　　　2. `confluent kafka cluster list`
　　　　3. `confluent kafka cluster use lkc-j81qrw`
　　　　4. `confluent api-key create --resource lkc-j81qrw`
　　　　　　a. Record the API key and secret

c. Once the key and secret are created and recorded for both clusters we're ready to leverage Cluster Linking for seamless data replication from one cluster to another

Step 3 - Setup Cluster Linking for seamless data replication

A bi-directional Cluster Link has already been setup by the Confluent team. Within the Confluent Cloud UI click on the "Cluster links" section on the left of the UI (note if you are in a cluster view there is only an icon, not the name "Cluster links"

1. You'll notice two cluster links labeled "GTG_bidirectional-link."

(There is a known bug in the UI where the Source cluster is not visually rendering correctly)

2. As previously outlined Bi-directional Cluster Linking is a new feature within Confluent Cloud enabling both active/passive AND active/active Disaster Recovery scenarios. Benefits of this architecture include:
   a. In an active/passive setup, a bidirectional cluster link ensures consumer offsets from the DR region get synced to the primary region, so that consumer applications from the DR region can be moved or failed over to the primary region.
   b. In an active/active setup, a bidirectional cluster link ensures that consumer offsets are synced to both clusters, so that consumers and producers can easily failover to the other cluster.

3. You can explore the available options by clicking on either of the two cluster links (please do NOT change any settings). Available options within a cluster link include:
   a. *Auto-create mirror topics*
      i. When enabled, all topics (or a user-defined subset of topics) will be automatically created in the secondary/DR cluster
      ii. NOTE: For this workshop you will create the mirrored topic via the CLI
   b. *Add prefix to mirror topics*
      i. When enabled will append the entered string to the topic created in the secondary/DR cluster
   c. *Enabling Synchronization of consumer offsets*
      i. This allows consumer applications to failover and restart very close to the point where they left off, drastically reducing the chance of missing any messages that have not been consumed yet.
   d. *Enabling Synchronization of access control lists (ACLs)*
      i. Syncing ACLs ensures that any applications and users who are authorized with ACLs maintain that authorization in the DR region

CONFIDENTIAL

4. Next we'll mirror the topic you created earlier to provide data replication to the secondary/DR cluster
   a. NOTE: In the CLI make sure you're entering commands in the secondary/DR cluster.
      i. `confluent environment use env-2rp5mm`
      ii. `confluent kafka cluster use lkc-j81qrw`
   b. Next create the mirror topic on the secondary/DR cluster (substitute the topic name shown below with the topic you created at the beginning of this workshop).
      i. `confluent kafka mirror create GTG_Workshop_CWT --link GTG_bidirectional-link`
   c. In the Confluent Cloud UI, when looking at "topics" in the secondary/DR cluster you'll see your mirrored topic (a mirrored topic is represented with the [icon] icon
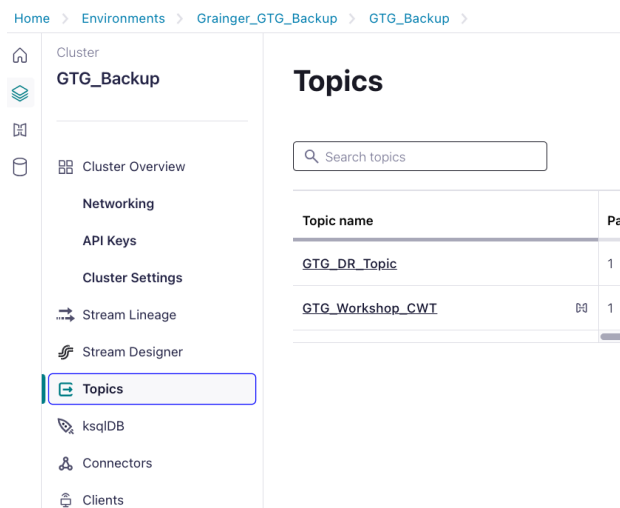
Step 4 - Run a python producer to send data to Confluent Cloud

1. The next step is to run the supplied python producer that will generate mock data and send it to your Confluent Cloud topic (where it will also be seamlessly sent to the DR cluster/topic). We will also simultaneously run two instances of the Python Kafka® consumer to consume messages from both the primary cluster and the DR cluster which is receiving the replicated messages in real time.
2. Pre-Requisites:
   a. Download the Python producer and consumer code from github. You should see the following files:
      i. *GTG_CC_Python*
         - *GTG_CC_Primary.ini*
         - *GTG_CC_Backup.ini*
         - *GTG_CC_python_producer.py*
         - *GTG_CC_python_consumer.py*

b. This simple python producer will pull the connection information from the .ini file, connect to the relevant Confluent Cloud cluster and then create JSON messages based on a random set of values. The python consumer code simply pulls messages from a Kafka® topic and displays them on the command-line terminal.

c. You'll need 3 terminal instances/tabs open for running the producer instance along with two consumer instances.

d. To initialize the python virtual environment and execute the program type in the following commands in each of the 3 terminals you have open (you can choose to use '**ccloud-gtg**' or use any virtual name you'd like):

    i. `virtualenv ccloud-gtg`

    ii. `source ccloud-gtg/bin/activate` (this only needs to be run the first time you initialize the environment)

    iii. `pip install confluent-kafka` (this only needs to be run the first time you initialize the environment)

e. Note there are two .ini files in the root folder; one for the primary Confluent Cloud cluster and another for the DR cluster.

f. Many settings have already been filled in, please confirm and/or fill in the following sections of the "***GTG_CC_Primary.ini***" file:

    i. sasl.username   [API Key previously created for Primary cluster]

    ii. sasl.password   [API Secret previously created for Primary cluster]

    iii. topic   [Enter in the topic name you created in Step 1]

```
GTG_CC_Python >  ≡ GTG_CC_Primary.ini
 1   [ConfluentCloudEndpoint]
 2   bootstrap.servers=pkc-5p1o2.us-east-1.aws.confluent.cloud:9092
 3   security.protocol=SASL_SSL
 4   sasl.mechanisms=PLAIN
 5   sasl.username=
 6   sasl.password=
 7   linger.ms=100
 8   max.in.flight.requests.per.connection=50
 9
10   [Schema_Registry]
11   # Required connection configs for Confluent Cloud Schema Registry
12   schema_registry_url=https://psrc-6zww3.us-east-2.aws.confluent.cloud
13   basic_auth_credentials_source=USER_INFO
14   basic_auth_user_info={{SR_API_KEY}}:{{SR_API_SECRET}}
15
16   [misc]
17   topic=GTG_Workshop_[XXX]
18   # Best practice for higher availability in librdkafka clients prior to 1.7
19   session_timeout_ms=45000
20
21   [consumer]
22   group.id=python_GTG_ConsumerGroup_1
```

g. Repeat the same steps for the "***GTG_CC_Backup.ini***" file

3. First we'll execute the two consumers to poll your topic from both the Primary and Backup Confluent Cloud clusters. To run the consumers:

a. To start the consumer for the Primary cluster, In a command-line terminal execute the following command (leave this running, at any time you can stop the consumer using ***control+c***):

          i.   `./GTG_CC_python_consumer.py GTG_CC_Primary.ini`

    b. To start the consumer for the Backup cluster, In a second command-line terminal execute the following command (also leave this running):

          i.   `./GTG_CC_python_consumer.py GTG_CC_Backup.ini`

4. To run the producer:

    a. To execute the producer program simply run

          i.   `./GTG_CC_python_producer.py GTG_CC_Primary.ini`

          ii.   Upon execution there will be a prompt for the number of messages you would like to send to the topic. Selecting 0 (zero) will send a continuous stream of messages until you stop the program using **control+c** on the keyboard.

5. Observe you'll see messages being sent from the producer and being displayed in both the primary and backup consumers being run in the other terminals.

6. Now try to produce to the cluster-linked topic. Run the following command and choose a small number of records to produce:

    a. `./GTG_CC_python_producer.py GTG_CC_Backup.ini`

    b. The program will return an error each time it attempts to produce a record:

```
(ccloud-gtg) GTG_CC_Python: ./GTG_CC_python_producer.py GTG_CC_Backup.ini
Enter the # of records to generate.  Enter '0' for a continuous stream of records [use control+c to interrupt]: 3
ERROR: Message failed delivery: KafkaError{code=INVALID_REQUEST,val=42,str="Broker: Invalid request"}
ERROR: Message failed delivery: KafkaError{code=INVALID_REQUEST,val=42,str="Broker: Invalid request"}
ERROR: Message failed delivery: KafkaError{code=INVALID_REQUEST,val=42,str="Broker: Invalid request"}
program complete
```

    c. This is expected behavior as the topic is a mirror topic.

7. For now leave both consumer applications running.

Step 5 - Disaster!!!

Now that we've seen data is flowing from the primary cluster to the backup cluster let's simulate a Disaster Recovery scenario. In this case we'll "promote" the backup cluster to be the primary cluster and change over the producer to send data through the newly promoted cluster.

NOTE: We have consumer instances running against both clusters for illustrative purposes of this workshop. In a real-world scenario both producer and consumer applications would need to be migrated over.

1. Let's go ahead and have data continuously streaming while failover occurs. Run the producer application and select '0' (zero):

    a. `./GTG_CC_python_producer.py GTG_CC_Primary.ini`

2. With data flowing into your topic you will now initiate failover of your topic by issuing the following command:

   a. First we'll execute a dry run of the failover by adding a `--dry-run` flag at the end of the command.

b. Switch over to the terminal running the Confluent CLI, then test the failover in a dry run by executing the following command (the command must be executed in the DR environment):

      i. `confluent environment use env-2rp5mm`

      ii. `confluent kafka mirror failover [topic name] --link GTG_bidirectional-link --cluster lkc-j81qrw --dry-run`

```
Grainger_GTG_2023: confluent environment use env-2rp5mm
Using environment "env-2rp5mm".
Grainger_GTG_2023: confluent kafka mirror failover GTG_Workshop_CWT --link GTG_bidirectional-link --cluster lkc-j81qrw --dry-run
 Mirror Topic Name | Partition | Partition Mirror Lag | Last Source Fetch Offset | Error Message | Error Code
-------------------+-----------+----------------------+--------------------------+---------------+------------
 GTG_Workshop_CWT  |        0 |                   0 |                      19 |               |
```

c. To execute the failover and "promotion" of the topic in the DR cluster issue the same command without the `--dry-run` flag.

d. `confluent kafka mirror failover GTG_Workshop_CWT --link GTG_Grainger_CL --cluster lkc-j81qrw`

3. Let's observe what's happened after the `mirror failover` command is executed. You'll notice data is still being produced and consumed in the primary cluster, however no data is now being received in the consumer attached to the backup cluster. Also, if we look in the UI at the topic in the Backup cluster we'll see there is no longer a 🔗 icon next to the topic name.

    a. Leave both consumers running but stop the producer.

    b. Re-run the producer using the backup .ini file

      i. `./GTG_CC_python_producer.py GTG_CC_Backup.ini`

    c. You should now observe data being read from the backup cluster consumer.

      i. NOTE: Before executing the `mirror failover` command remember the python client was not able to produce records into the backup cluster topic.

4. At this point the topic has been "failed over" and both produce and consume operations can continue through the backup cluster.

Step 6 - Failing back

1. Let's say that after a period of time you'd like to return to using the Primary data cluster. This time we'll create a cluster link from the secondary/DR cluster to the primary cluster.

NOTE: Before creating the cluster link we need to delete the topic in the primary cluster.

NOTE: This would NOT be a production strategy; deletion and migration of data to the original topic is only done in this workshop for illustrative purposes.

2. First log into the UI and navigate to the "Grainger_GTG_Primary" environment -> GTG_Primary cluster.

    a. From there select "topics",

      i. select your topic ("GTG_Workshop_CWT" for example)

      ii. Navigate to "Configuration" and then select "Delete topic"
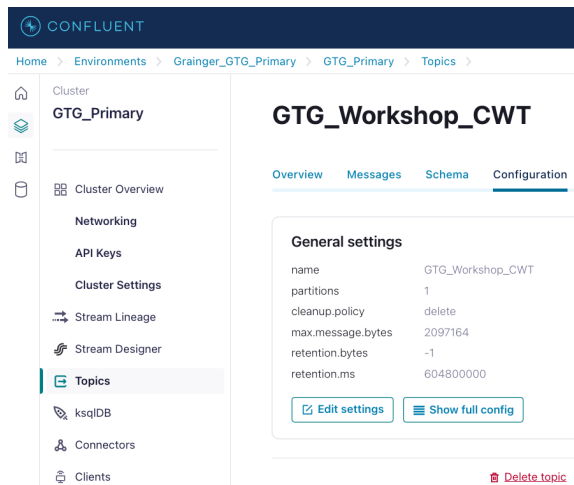
3. Next go back to the Confluent CLI and create the cluster link from the primary cluster back to the primary cluster.

   a. `confluent environment use env-09q50q`

   b. `confluent kafka mirror create GTG_Workshop_CWT --link GTG_bidirectional-link`

4. At this point we'll see the topic as a mirror topic in the GTG_Primary cluster. If your consumers are not running go ahead and start up both, and then let's try to produce records to the GTG_Primary cluster.

   a. In one terminal, `./GTG_CC_python_consumer.py GTG_CC_Primary.ini`

   b. In a second terminal, `./GTG_CC_python_consumer.py GTG_CC_Backup.ini`

   c. Now let's produce to the Backup cluster. In a 3rd terminal run

      i. `./GTG_CC_python_producer.py GTG_CC_Backup.ini`

   d. Confirm you're able to see data flow through to both consumer instances

   e. Next let's confirm we're NOT able to produce to the mirror topic in the GTG_Primary cluster. When you execute the following command you should receive errors stating "Invalid request"

      i. `./GTG_CC_python_producer.py GTG_CC_Primary.ini`

      ii.
      ```
      (ccloud-gtg) GTG_CC_Python: ./GTG_CC_python_producer.py GTG_CC_Primary.ini
      Enter the # of records to generate.  Enter '0' for a continuous stream of records [use control+c to interrupt]: 2
      ERROR: Message failed delivery: KafkaError{code=INVALID_REQUEST,val=42,str="Broker: Invalid request"}
      ERROR: Message failed delivery: KafkaError{code=INVALID_REQUEST,val=42,str="Broker: Invalid request"}
      program complete
      (ccloud-gtg) GTG_CC_Python:
      ```

5. Next let's go ahead and failover back to the topic in the GTG_Primary cluster which will allow our producers to once again send data to the GTG_Primary cluster.

   a. Switch over to the terminal running the Confluent CLI, then test the failover in a dry run by executing the following command (the command must be executed in the Primary environment):

      i. `confluent environment use env-09q50q`

      ii. `confluent kafka mirror failover [topic name] --link GTG_bidirectional-link --cluster lkc-0jvoy6 --dry-run`

b. Assuming no errors occur re-run the command without the `--dry-run` flag to complete the failover.

    i. `confluent kafka mirror failover [topic name] --link GTG_bidirectional-link --cluster lkc-0jvoy6`

c. After a short period of time re-run the producer against the GTG_Primary cluster, you will now notice data can be produced (and only the consumer attached to the primary cluster is receiving data)

    i. `./GTG_CC_python_producer.py GTG_CC_Primary.ini`

Appendix

Kafka Producer - Python Code:

```python
#!/usr/bin/env python

import sys
import json
from random import choice
from argparse import ArgumentParser, FileType
from configparser import ConfigParser
from confluent_kafka import Producer


if __name__ == '__main__':
# Parse the command line.
argparser = ArgumentParser()
argparser.add_argument('ConfigFile',type=FileType('r'))
args = argparser.parse_args()

# Parse the configuration.
# See https://github.com/edenhill/librdkafka/blob/master/CONFIGURATION.md
config_parser = ConfigParser()
config_parser.read_file(args.ConfigFile)
config = dict(config_parser['ConfluentCloudEndpoint'])
# Create Producer instance
producer = Producer(config)

# Optional per-message delivery callback (triggered by poll() or flush())
# when a message has been successfully delivered or permanently
# failed delivery (after retries).
def delivery_callback(err, msg):
if err:
print('ERROR: Message failed delivery: {}'.format(err))
else:
print("Produced event to topic {topic}: key = {key:12} value = {value:12}".format(
topic=msg.topic(), key=msg.key().decode('utf-8'), value=msg.value().decode('utf-8')))

# Produce data by selecting random values from these lists.
topic = config_parser["misc"]["topic"]
user_ids = ['eabara', 'jsmith', 'sgarcia', 'jbernard', 'htanaka', 'awalther']
```

CONFIDENTIAL   14

```python
products = ['book', 'alarm clock', 't-shirts', 'gift card', 'batteries']
price = ['$5.99', '$10.25', '$2.49', '$4.83']
color = ['blue','green','yellow','red','orange']
num_records = int(input("Enter the # of records to generate. Enter '0' for a continuous stream of records [use control+c to interrupt]: "))
if num_records > 0:
for count in range(num_records):
record_json = json.dumps({'product': choice(products), 'price': choice(price), 'color': choice(color)})
producer.produce(topic, value=record_json, key=choice(user_ids), callback=delivery_callback)
count += 1
producer.poll(10000)
producer.flush()
else:
while True:
try:
record_json = json.dumps({'product': choice(products), 'price': choice(price), 'color': choice(color)})
producer.produce(topic, value=record_json, key=choice(user_ids), callback=delivery_callback)
producer.poll(10000)
producer.flush()
except KeyboardInterrupt:
break


print('program complete')
```

Kafka Consumer - Python Code:

```python
#!/usr/bin/env python

import sys
from argparse import ArgumentParser, FileType
from configparser import ConfigParser
from confluent_kafka import Consumer, OFFSET_BEGINNING

if __name__ == '__main__':
# Parse the command line.
argparser = ArgumentParser()
argparser.add_argument('config_file', type=FileType('r'))
argparser.add_argument('--reset', action='store_true')
```

```python
args = argparser.parse_args()

# Parse the configuration.
# See https://github.com/edenhill/librdkafka/blob/master/CONFIGURATION.md
config_parser = ConfigParser()
config_parser.read_file(args.config_file)
config = dict(config_parser['ConfluentCloudEndpoint'])
config.update(config_parser['consumer'])

# Create Consumer instance
consumer = Consumer(config)

# Set up a callback to handle the '--reset' flag.
def reset_offset(consumer, partitions):
if args.reset:
for p in partitions:
p.offset = OFFSET_BEGINNING
consumer.assign(partitions)

# Subscribe to topic
topic = config_parser["misc"]["topic"]
consumer.subscribe([topic], on_assign=reset_offset)

# Poll for new messages from Kafka and print them.
try:
while True:
msg = consumer.poll(1.0)
if msg is None:
# Initial message consumption may take up to
# `session.timeout.ms` for the consumer group to
# rebalance and start consuming
print("Waiting...")
elif msg.error():
print("ERROR: %s".format(msg.error()))
else:
# Extract the (optional) key and value, and print.

print("Consumed event from topic {topic}: key = {key:12} value = {value:12}".format(
topic=msg.topic(), key=msg.key().decode('utf-8'), value=msg.value().decode('utf-8')))
except KeyboardInterrupt:
```

```
pass
finally:
# Leave group and commit final offsets
consumer.close()
```

Primary Cluster - Login/Security .ini file:

```ini
[ConfluentCloudEndpoint]
bootstrap.servers=pkc-5p1o2.us-east-1.aws.confluent.cloud:9092
security.protocol=SASL_SSL
sasl.mechanisms=PLAIN
sasl.username=
sasl.password=
linger.ms=100
max.in.flight.requests.per.connection=50

[Schema_Registry]
# Required connection configs for Confluent Cloud Schema Registry
schema_registry_url=https://psrc-6zww3.us-east-2.aws.confluent.cloud
basic_auth_credentials_source=USER_INFO
basic_auth_user_info={{SR_API_KEY}}:{{SR_API_SECRET}}

[misc]
topic=GTG_Workshop_[XXX]
# Best practice for higher availability in librdkafka clients prior to 1.7
session_timeout_ms=45000

[consumer]
group.id=python_GTG_ConsumerGroup_1

# 'auto.offset.reset=earliest' to start reading from the beginning of
# the topic if no committed offsets exist.
auto.offset.reset=earliest
```

Secondary/DR Cluster - Login/Security .ini file:

```ini
[ConfluentCloudEndpoint]
bootstrap.servers=pkc-dd0xd.us-west-2.aws.confluent.cloud:9092
security.protocol=SASL_SSL
sasl.mechanisms=PLAIN
sasl.username=
```

```
sasl.password=
linger.ms=100
max.in.flight.requests.per.connection=50

[Schema_Registry]
# Required connection configs for Confluent Cloud Schema Registry
schema_registry_url=https://psrc-6zww3.us-east-2.aws.confluent.cloud
basic_auth_credentials_source=USER_INFO
basic_auth_user_info={{SR_API_KEY}}:{{SR_API_SECRET}}

[misc]
topic=GTG_Workshop_[XXX]
# Best practice for higher availability in librdkafka clients prior to 1.7
session_timeout_ms=45000

[consumer]
group.id=python_GTG_ConsumerGroup_1

# 'auto.offset.reset=earliest' to start reading from the beginning of
# the topic if no committed offsets exist.
auto.offset.reset=earliest
```