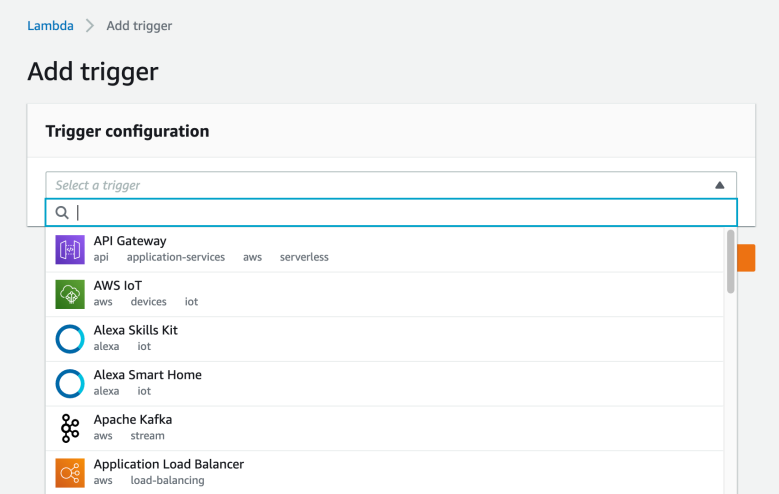
Ingesting data from a SSL mutual authentication enabled Kafka on-premises using lambda.

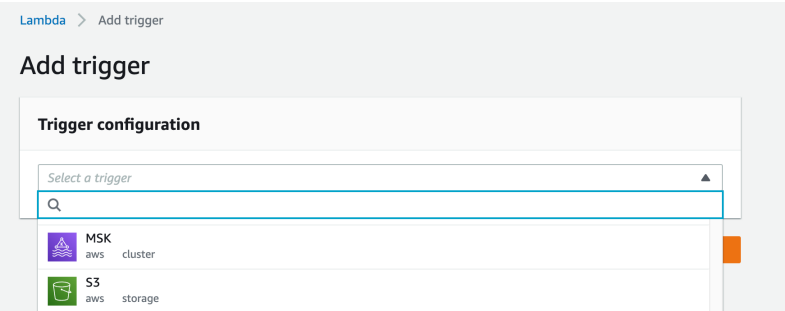
In some datalake project implemented by proserve there was the use case of ingesting data from different Kafka servers into a datalake. One Kafka was on-premises. This kafka was a confluent Kafka installation, with SSL enabled, configured with a customer internal CA certificate as root CA. Another Kafka was a MSK installation in another AWS account, without authentication. A third Kafka was also in another AWS account, but this Kafka had SSL Mutual authentication enabled.

One of the challenges was to create a low-cost, generic ingest solution to consume these kafka servers. For this, the option was to create a lambda based solution that costs USD $11 per month, over ec2 or eks solutions that costs minimum USD $110 per month.

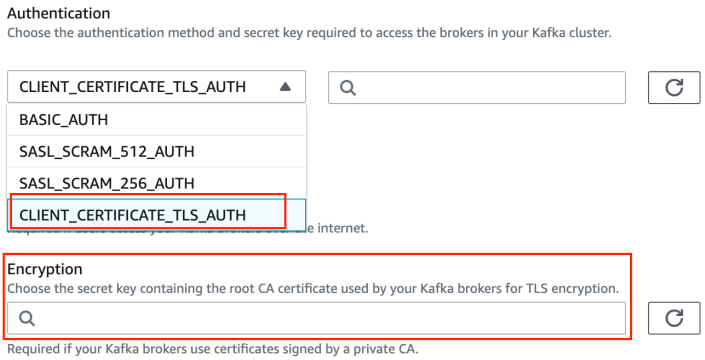
Lambda offers some lambda triggers to connect with some kafka implementations. For example, to connect with a custom kafka implementation , there is an Apache Kafka Trigger:



And another different implementation for Managed Kafka Service:

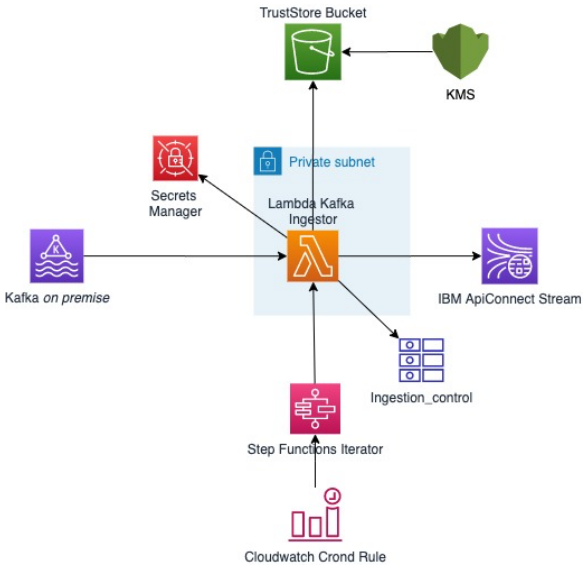


At the time the project was implemented (October 2021) , there was no possibility of authentication using digital certificates in these triggers. Today (January 2022), there are these authentication options :



These options allow to connect with a kafka using TLS encryption and authentication, but it enforces to use a secret to get the root CA certificate to validate trust between communications. This has a problem: when the CA certificate chain size is greater than the secret maximum size limit, we can't use this solution.

for this reason, the following solution has been implemented:

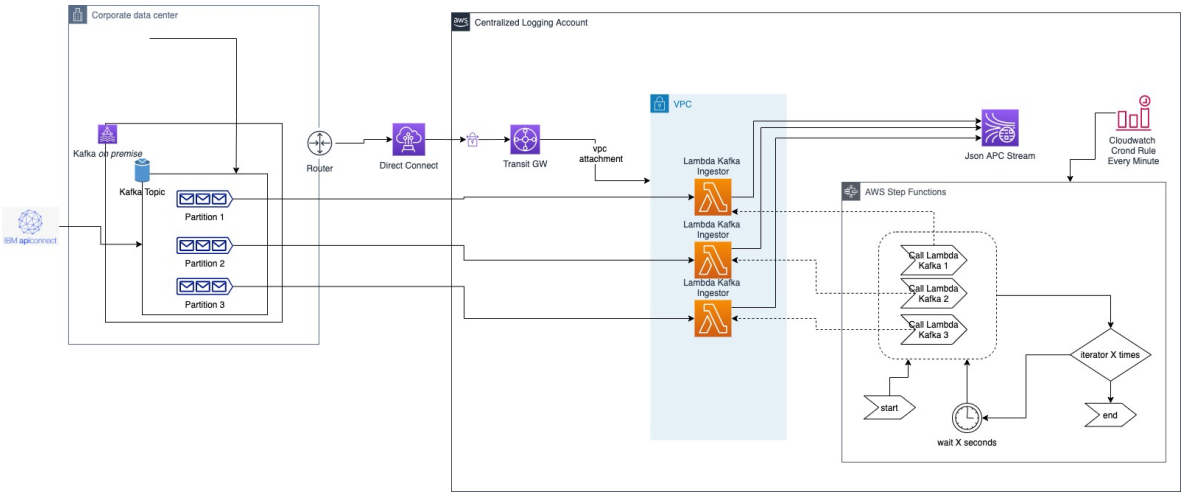


In this solution, a python lambda uses the confluent-kafka library to consume messages from kafka servers. This lambda has the ability to read the digital certificates required for communication from secrets manager or from a KMS-secured s3 bucket.

This lambda is inside a VPC to ensure communication on premises and with other aws accounts. This lambda sends the messages it receives to a configurable kinesis data stream. In the case of the project in question, the messages come from a centralizing system of Web APIs called "IBM API Connect" which collects logs of API calls and sends them to a kafka on-premises.

This lambda uses a control dynamodb table, which acts as a lock table to prevent the lambda from processing duplicate messages from the kafka.

This lambda is invoked from a step functions which is in turn invoked from a cloudwatch rule. The detailed procedure of this step functions is explained below:



In this flow, the API Connect application send events to the Kafka server that is on-premises, in a kafka topic, which is currently consumed by several consumers in the customer.

Communication between on-premises and the AWS account is done thanks to the direct connect and transit gateway services, which provide communication between the on-premises networks and the VPC where the ingestor Kafka lambda runs. This detail will not be explained in this document, since this configuration is done by default for all AWS accounts provisioned by the customer's cloud group.

Kafka topic is divided into 3 partitions. Partition is the concept of distribution that Kafka uses to be scalable. This Kafka server has 3 brokers or message processing nodes. The idea is that each partition lives within a Kafka broker.

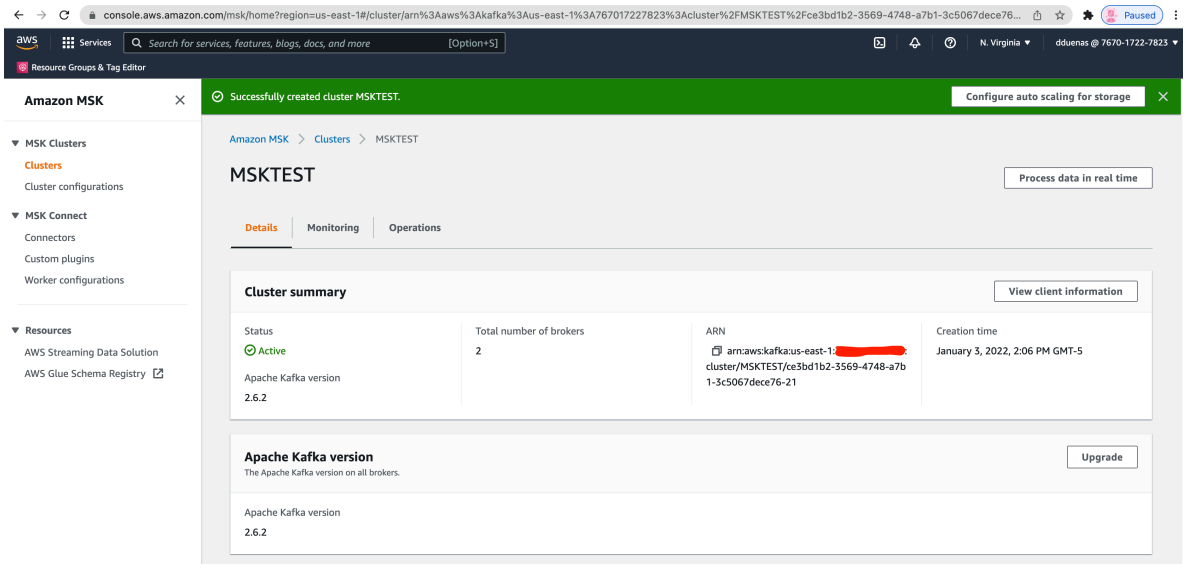
Each partition in the Kafka topic processes a different set of messages, where each message is marked with a unique identifier known as an "offset."

The diagram shows the invocation logic of the ingesting lambda Kafka, which is invoked from the orchestration step functions, where it is invoked in parallel 1 time for each partition in the Kafka. After each invocation there is a set of configurable loops that are executed in the step functions. This step functions is invoked from a cloudwatch rule that runs every minute.

Below is an example of how the lambda kafka ingestor is implemented and used.

Configuring test environment

To show an example execution of the lambda, a test kafka cluster has been created with the MKS service, called “msktest”:



You can create a cluster with or without authentication or TLS enabled for the test. The lambda configuration supports any scenario.

The MSK cluster creation can take up to 15 mins. After that, you can create an example kafka topic. For this, it was created an EC2 instance and downloaded the opensource kafka client from https://kafka.apache.org/downloads . Also, you need to install the AWS command line tool if not installed.

Once downloaded and decompressed, you need to know first the zookeeper connect string to run some kafka admin commands. For this, you can run the “aws kafka describe-cluster” :

```
[root@ip-10-0-0-142 ec2-user]# ls
kafka_2.12-2.2.1  kafka_2.12-2.2.1.tgz  python_test
[root@ip-10-0-0-142 ec2-user]# cd kafka_2.12-2.2.1/
[root@ip-10-0-0-142 kafka_2.12-2.2.1]# ls
bin  client-cert-sign-request  client.properties  config  libs  LICENSE  NOTICE  signed-certificate-from-acm  site-docs
[root@ip-10-0-0-142 kafka_2.12-2.2.1]# sudo su ec2-user
[ec2-user@ip-10-0-0-142 kafka_2.12-2.2.1]$ aws kafka describe-cluster --region us-east-1 --cluster-arn "arn:aws:kafka:us-east-1:[redacted]:cluster/MSKTEST/ce3bd1b2-3569-4748-a7b1-3c5
067dece76-21" --region us-east-1
{
    "ClusterInfo": {
        "LoggingInfo": {
            "BrokerLogs": {
                "S3": {
                    "Enabled": false
                },
                "Firehose": {
                    "Enabled": false
                },
                "CloudWatchLogs": {
                    "Enabled": false
                }
            }
        },
        "EncryptionInfo": {
```

You can obtain the cluster ARN from the MSK AWS web console. Then, from the json result you can take the ZookeeperConnectString property:

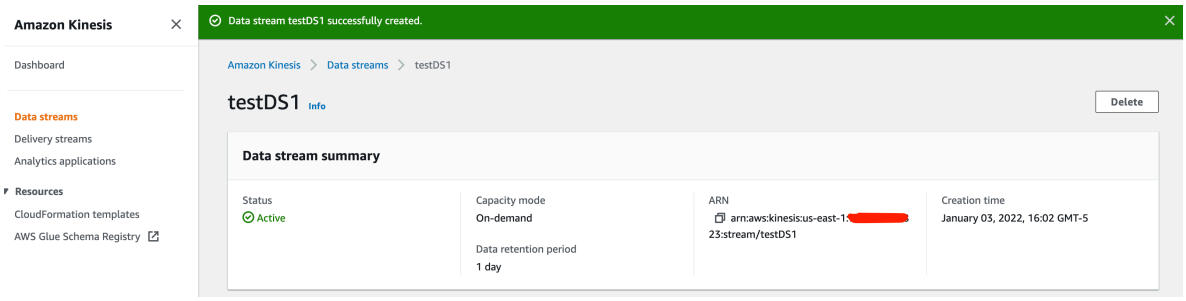
```
        "ClusterName": "MSKTEST",
        "CurrentBrokerSoftwareInfo": {
            "KafkaVersion": "2.6.2"
        },
        "Tags": {},
        "CreationTime": "2022-01-03T19:06:29.993Z",
        "NumberOfBrokerNodes": 2,
        "ZookeeperConnectString": "z-2.msktest.qx098q.c21.kafka.us-east-1.amazonaws.com:2181,z-3.msktest.qx098q.c21.kafka.us-east-1.amazonaws.com:2181,z-1.msktest.qx098q.c21.kafka.us-east-1.amazonaws.com:2181",
        "State": "ACTIVE",
```


With this information you can create an example topic using the downloaded kafka client:

```
[ec2-user@ip-10-0-0-142 kafka_2.12-2.2.1]$ bin/kafka-topics.sh --create --zookeeper "z-2.msktest.qx098q.c21.kafka.us-east-1.amazonaws.com:2181,z-3.msktest.qx098q.c21.kafka.us-east-1.amazonaws.com:2181,z-1.msktest.qx098q.c21.kafka.us-east-1.amazonaws.com:2181" --replication-factor 2 --partitions 3 --topic ExampleTopic
Created topic ExampleTopic.
```

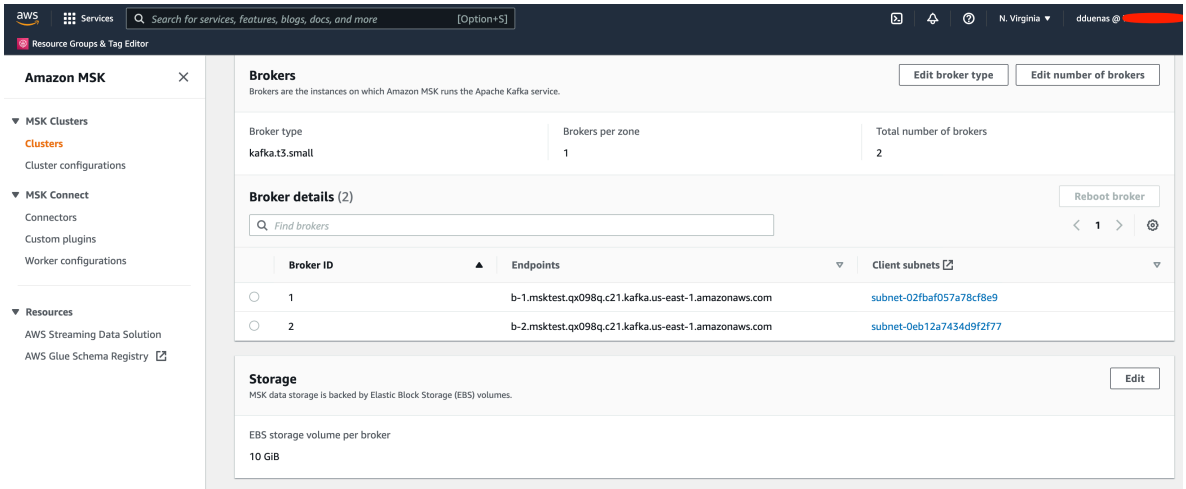
To run this command, you need to be in the root of the kafka client, you need to provide the previous zookeeper string, and for this example we are creating a topic called “ExampleTopic” with 3 partitions.

Also, we need to create a kinesis data stream destination for the lambda. In this case, the stream is called “testDS1”:



Writing a test program to generate API Connect like events

The second step to run the example is to write a test program that emulates the kafka source events. In this case it will be a python program. This program needs to know the list of kafka brokers, that we can obtain from the MSK web console:



The program uses the python library called “confluent-kafka” (https://docs.confluent.io/clients-confluent-kafka-python/current/overview.html).

Below you can find the example python code that creates a confluent kafka producer, and sends some mock messages to a configured kafka, based on a json template. (in the code folder attached , in “testProducerCode” folder you can find the python source code and the json template file) :

```
from confluent_kafka import Producer
from time import sleep
import json
import time
import random
from random import randrange
from datetime import datetime, timedelta, timezone

# ----------------------------------------
# Kafka configuration
# ----------------------------------------
def config_listener(broker):
    # Start listener
    producer_config = {
        'bootstrap.servers': broker,
        # SSL configs
        "security.protocol": "SSL",
        "ssl.ca.location": "ca_certificate_chain.pem",
        "ssl.certificate.location": "certificate.pem",
        "ssl.key.location": "key.pem"
    }
    return Producer(producer_config)

# ----------------------------------------
# Log Generator Class
# ----------------------------------------
class DataGenerator():
    def __init__(self):
        pass

    def app_name_generator(self):
        apps = ['APP1',
         'APP2',
         'APP3',
         'APP4']
        self.app_name = random.choice(apps)
        return self.app_name

    def api_name_generator(self):
        api_names = ['api-name-1', 'api-name-2', 'api-name-3',
                     'api-name-4']
        self.api_name = random.choice(api_names)
        return self.api_name

    def status_code_generator(self, anomaly=False):
        status_codes = ['200 OK', '404', '500']
        normal_weights = [1, 0, 0]
        anomaly_weights = [0, 0.5, 0.5]
        weights = anomaly_weights if anomaly else normal_weights

        self.status_code = random.choices(status_codes, weights)[0]
        return self.status_code

    def time_to_serve_generator(self, anomaly=False):
        time_population = [20, 22, 26, 82, 70, 25]
        normal_weights = [0.245, 0.245, 0.245, 0.01, 0.01, 0.245]
        anomaly_weights = [0.025, 0.025, 0.025, 0.45, 0.45, 0.025]
        weights = anomaly_weights if anomaly else normal_weights

        self.time_to_serve_request = random.choices(time_population, weights)[0]
        return self.time_to_serve_request

    def bytes_sent_generator(self, anomaly=False):
```

```
        bytes_sent_population = [1590, 1560, 1350, 1400, 3250, 4000]
        normal_weights = [0.245, 0.245, 0.245, 0.245, 0.01, 0.01]
        anomaly_weights = [0.025, 0.025, 0.025, 0.025, 0.45, 0.45]
        weights = anomaly_weights if anomaly else normal_weights

        self.bytes_sent = random.choices(bytes_sent_population, weights)[0]
        return self.bytes_sent

    def bytes_received_generator(self, anomaly=False):
        bytes_received_population = [458, 468, 534, 500, 490, 0]
        normal_weights = [0.2, 0.2, 0.2, 0.2, 0.19, 0.01]
        anomaly_weights = [0.025, 0.025, 0.025, 0.025, 0.025, 0.875]
        weights = anomaly_weights if anomaly else normal_weights

        self.bytes_received = random.choices(bytes_received_population, weights)[0]
        return self.bytes_received

    def datetime_generator(self):
        now = datetime.now(timezone.utc)
        now_minus_10 = datetime.now(timezone.utc) - timedelta(minutes=10)
        start = datetime.strptime(now_minus_10.strftime('%Y-%m-%dT%H:%M:%S.%fZ'),
                                  '%Y-%m-%dT%H:%M:%S.%fZ')
        end = datetime.strptime(now.strftime('%Y-%m-%dT%H:%M:%S.%fZ'), '%Y-%m-
%dT%H:%M:%S.%fZ')
        delta = end - start
        int_delta = (delta.days * 24 * 60 * 60) + delta.seconds
        random_second = randrange(int_delta)
        self.datetime = (start + timedelta(seconds=random_second)).strftime(
            "%Y-%m-%dT%H:%M:%S.%f")
        self.datetime = self.datetime[:-3] + "Z"
        return self.datetime

data_generator = DataGenerator()

# ----------------------------------------
# Build mock log function
# ----------------------------------------

def generate_log(template_log):
    new_log = template_log
    anomaly_options = [True, False]
    anomaly_weights = [5, 95]

    anomaly = random.choices(anomaly_options, anomaly_weights, k=1)[0]
    print(anomaly)
    new_log['app_name'] = data_generator.app_name_generator()
    new_log['api_name'] = data_generator.api_name_generator()
    new_log['status_code'] = data_generator.status_code_generator(anomaly=anomaly)
    new_log['time_to_serve_request'] =
data_generator.time_to_serve_generator(anomaly=anomaly)
    new_log['bytes_sent'] = data_generator.bytes_sent_generator(anomaly=anomaly)
    new_log['bytes_received'] = data_generator.bytes_received_generator(anomaly=anomaly)

    return new_log


def log_producer(broker, topic):
    producer = config_listener(broker)
    # Read log template
    f = open('template_event.json', )
    log_data = json.load(f)

    flush_counter = 0
    try:
        date = data_generator.datetime_generator()
        for i in range(0, 30000):
            new_log = generate_log(log_data)
            new_log["transaction_id"] = str('transaction' + str(i))
            new_log['datetime'] = date
            #print(new_log)
            producer.produce(topic, json.dumps(new_log))
```

```
            if flush_counter == 100:
                print("############## " + str(i))
                sleep(1)
                date = data_generator.datetime_generator()
                print(date, i)
                producer.flush()
                flush_counter = 0

            flush_counter += 1

    except Exception as ex:
        print("Kafka Exception : {}", ex)
    finally:
        print("closing producer")
        producer.flush()

# ----------------------------------------
# Main Invocation
# ----------------------------------------

broker = "b-1.msktest.qx098q.c21.kafka.us-east-1.amazonaws.com:9094,b-
2.msktest.qx098q.c21.kafka.us-east-1.amazonaws.com:9094"
topic = "ExampleTopic"

my_producer = log_producer(broker, topic)
```

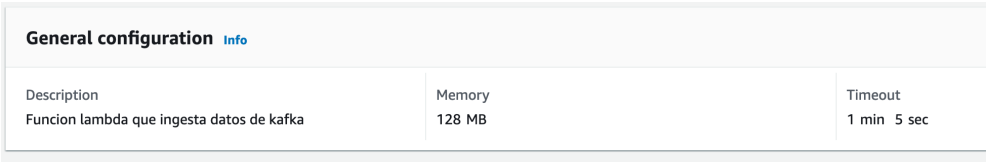
The main method in this code is “log_producer” and it contains a for that sends 30000 messages to the kafka, in batches of 100 records every second. Note that at the end there are the constants “broker” and “topic” to configure the kafka brokers and the name of the topic created previously.

NOTE: The code presented above is for reference only, and is intended to show the functionality of the libraries presented.

You can run this python code with the instruction “python3 <file_name>.py” , to create the test messages.

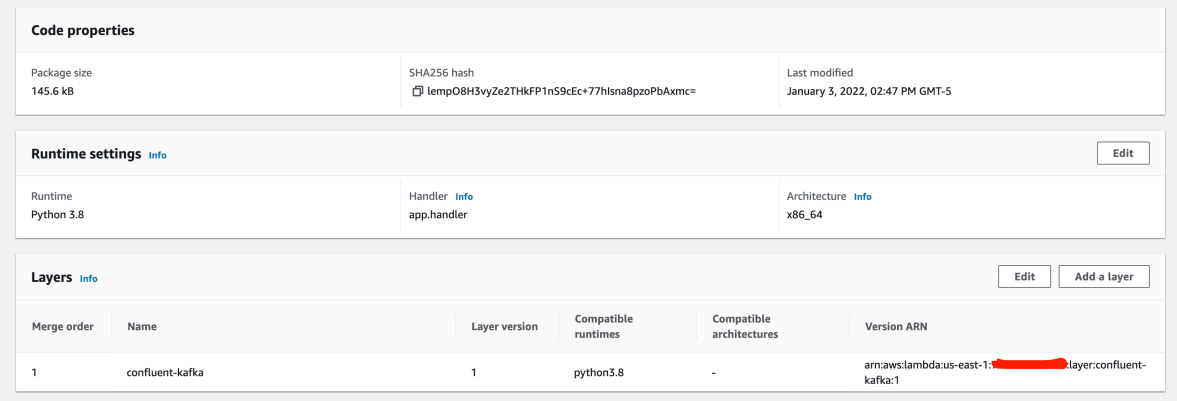
Writing the lambda kafka ingestor

For this example it was created a lambda called “testLambdaMSKSSLV2”. These are the default resource values. Since the lambda will be orchestrated by a cloudwarch rule running every minute, its timeout is 1 min and 5 seconds.

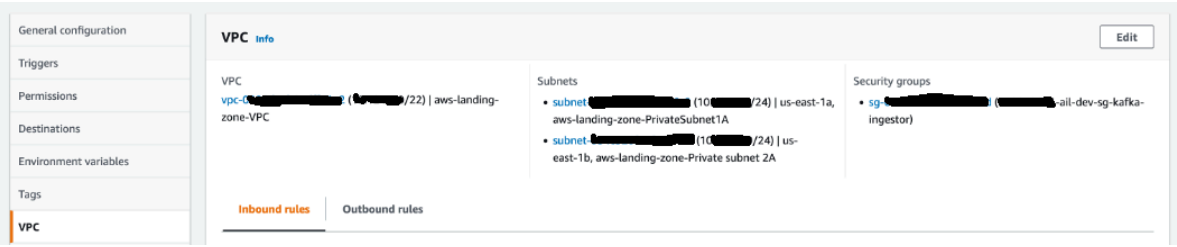


This lambda is configured with the minimum amount of memory, which is 128 Mb. This means that only one CPU core is assigned. The average execution time of this lambda is 5 seconds, with a maximum of 10 seconds in an AWS deployed kafka.

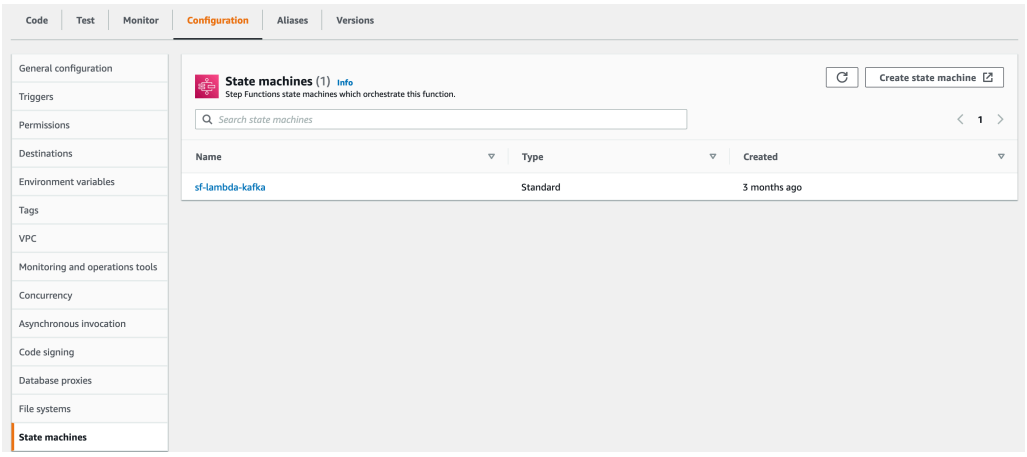
The lambda also uses a lambda layer, which you can find attached in the source code folder “LambdaKafkaCode” -> ”confluentKafkaLayer”. This is a python 3.8 or 3.9 lambda layer:



Since this lambda must connect to a private customer network, it is mandatory that this lambda live within a private subnet in an AWS VPC. All AWS customer accounts come with a VPC and 4 subnets, 2 public and 2 privates by default. In this case, the lambda has been instantiated in the 2 private subnets, and a security group has been created to control the inflow and outflow of data from / to on-premises at the firewall level.

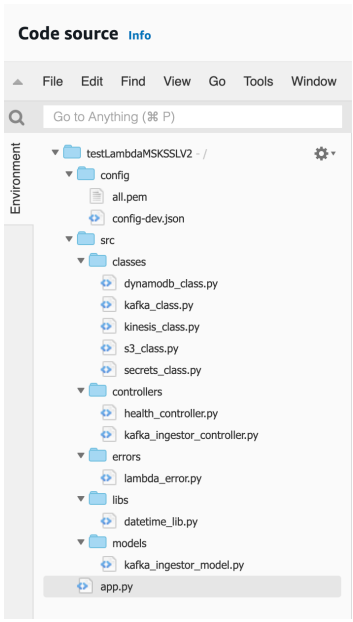


Finally, when the infrastructure of this lambda is deployed completely, it will have a reference to the step functions that invokes the lambda. It should appear in the “state machines” section of the lambda console:



This step functions will be explained in detail later in this document.

At the code level, the lambda has the following structure:



All the source code associated with this lambda can be found in the attached folder at: “LambdaKafkaCode” -> “lambdaKafkaIngestor”.

Each of its components is described below:

Component	Description
config	In this directory are the files all.pem, which is the generic java 8 truststore file (required if a connection is to be made to an MSK (AWS Managed Kafka) cluster and config-dev.json that contains all the configuration properties of the lambda that will be explained later.
app.py	This class is the entry point of the lambda. It has a method to read the json config file, and it also decides which controller it should call. The standard customer lambdas have 2 controllers, one called the health controller, which is used for smoke test purposes and the lambda's own controller.
Src/clases/dynamodb_class.py	This class handles communication with the dynamodb ingest control table. It has a method that inserts a record in this control table with its respective ttl, and another method that allows you to delete the control record from the table.

Src/clases/kafka_class.py	This class is the core of the lambda, and will be explained in more detail later. It uses the “confluent-kafka” library which is the most robust Python library to communicate with a Kafka server. This library was used since it allows the authentication scheme via TLS managed by the Kafka onpremises. It mainly has a method that reads messages from the Kafka and prepares them to be sent in the format expected by kinesis data streams.
Src/clases/kinesis_class.py	This class is responsible for sending a group of messages in batches to kinesis data streams, using the put_records method of boto3. It has a mechanism for retrying shipments.
Src/clases/s3_class.py	This class is in charge of communicating with the s3 bucket where the truststore is stored with the customer's SSL certificates. It has a method that reads the certificate from the bucket and copies it to a temporary space on the lambda disk.
Src/clases/secrets_class.py	This class is responsible for communicating with the AWS secrets manager service. It can be used against a Kafka with mutual TLS authentication, where the use of a certificate and a private key is required, these must be protected in secrets and this class allows you to retrieve them from that service.
Src/controllers/health_controller.py	This class is generic in all customer lambdas, and contains a method that always returns a successful response, and is used to do smoke tests.
Src/controllers/kafka_ingestor_controller.py	This class is a controller that is maintained as part of the MVC pattern in which lambdas are developed in the customer. In this lambda it is a bypass towards the kafka_ingestor_controller.py class and additionally formats the outputs of the lambda according to the customer's code standard.
Src/errors/lambda_error.py	This class is an extension of the python Exception class, to model an error that is thrown from this lambda.

Src/libs/datetime_lib.py	This class has utility methods for handling dates in the lambda.
Src/models/Kafka_ingestor_model.py	This class is the one that contains the orchestration of the calls to the classes found in src / classes. after this it returns the response to the controller.

The config-dev.json file contains the following configuration properties:

```
{
  "aws_region_name": "us-east-1",
  "aws_access_key_id": "",
  "aws_secret_access_key": "",
  "events": {
    "health": "health"
  },
  "status_message_codes": {
    "fully_processed": {
      "code": 200,
      "message": "Kafka messages found and fully processed."
    },
    "partially_processed": {
      "code": 206,
      "message": "Kafka messages were found but some failing. See logs and detail field for
more information."
    },
    "completely_failed": {
      "code": 503,
      "message": "Kafka messages were found and all failed. See logs and detail field for
more information."
    },
    "not_found_records": {
      "code": 204,
      "message": "No Kafka messages were found."
    },
    "another_lambda_processing_records": {
      "code": 203,
      "message": "Another lambda processg records in partition, skipping."
    }
  },
  "kinesis": {
     "retry_count": 10,
     "retry_wait_in_sec": 0.1,
     "stream_name": "testDS1"
  },
  "kafka" : {
      "bootstrap_servers": "b-1.msktest.qx098q.c21.kafka.us-east-1.amazonaws.com:9094,b-
2.msktest.qx098q.c21.kafka.us-east-1.amazonaws.com:9094",
      "topic_name": "ExampleTopic",
      "group_id": "ExampleTopic-G1",
      "enable_auto_commit": "true",
      "records_batch_size": 300,
      "poll_seconds": 4,
      "security_protocol": "SSL",
      "use_authentication": "false",
      "use_truststore_only": "true",
      "ssl_ca_location": "config/all.pem",
      "secrets_ssl_certificate_location": "dev-kafka-public-cert",
      "secrets_ssl_key_location": "dev-kafka-private-key"
  },
  "dynamodb": {
     "use_control_table": "true",
     "control_table_name": "inn007-dynamo-kafka-control",
     "primary_key_name": "topic_partition_id",
     "time_to_live_time_mins": "3"
  },
  "lambda": {
```

```
    "temp_folder": "/tmp/"
  }
}
```

The description of the properties is below:

Component	Description
Aws_region_name	AWS Region where the lambda is deployed
Events.health	If the lambda receives in the input event "type": "health" in its parameters, the lambda is executed with the health controller. It is for lambda smoke test purposes.
Status_message_codes	This is a json array with all the possible outputs that the lambda can have, in code - description pairs. These codes are generally homologated to traditional HTTP response codes, although this is not required.
Kinesis.retry_count	Number of times the lambda will retry sending messages to kinesis data streams if it finds an error in the message delivery.
Kinesis.retry_wait_in_sec	Timeout in seconds, between retry and retry, sending messages to kinesis.
Kinesis.stream_name	Name of the kinesis data stream where the destination data will be sent.
Kafka.bootstrap_servers	String with the chain of Kafka bootstrap servers from which the messages will be extracted.
Kafka.topic_name	Name of the kafka topic to which the lambda will connect to listen to the messages.
Kafka.group_id	Name of Kafka's consumer group. The idea is that all lambda instances will belong to the same consumer group.
Kafka.enable_auto_commit	"True" to indicate if Kafka is responsible for handling the commit of the offsets of each partition in a topic or "false" otherwise.
Kafka.records_batch_size	Maximum number of kafka records to be brought in at each kafka reading. At this moment it is configured every 300 records, since the limit of 1 mb per second that kinesis data streams has should not be exceeded too much, and 300 records is approx 1 mb taking into account that an api connect record weight is 3.5 k in average.
Kafka.poll_seconds	Number of seconds the lambda will wait for records in Kafka's queue before releasing it. In this case it is set to 4 seconds, so the lambda connects to Kafka, waits 4 seconds for the arrival of new messages (unless there are 300 that it is requesting) and ends its process.

Kafka.security_protocol	Data-in-transit communication security protocol used by Kafka. SSL indicates that communication will take place using digital certificates in X509 format.
kafka.use_authentication	"True" if the lambda uses authentication against the source Kafka. "False" otherwise.
kafka.use_truststore_only	"True" if the lambda uses authentication, but does not use mutual authentication but only TLS communication in transit (which is the case of the customer).
kafka.s3_ssl_ca_location	This property indicates the path in s3 where the truststore file that contains the bank's trusted certificates for the SSL connection lives. This property has 2 variants: “ssl_ca_location” to indicate that the truststore will take it from the lambda file system, and “secrets_ssl_ca_location” to indicate that the truststore will take it from the secrets service.
kafka.secrets_ssl_certificate_location	This property indicates the name of the secret in the secrets manager service, where the certificate that the lambda will use to authenticate itself against the Kafka in case of mutual TLS authentication is configured.
kafka.secrets_ssl_key_location	This property indicates the name of the secret in the secrets manager service, where the private key that the lambda will use to authenticate itself against the Kafka in case of mutual TLS authentication is configured.
dynamo.use_control_table	"True" if the lambda uses the dynamodb ingest control table. "False" otherwise.
dynamo.control_table_name	Name of the ingest control table dynamodb. This table acts as a table of locks, to avoid reading the same partition of a Kafka topic at the same time.
dynamo.primary_key_name	Name of the primary key field in the dynamodb table.
dynamo.time_to_live_time_mins	Time to live of a record in the dynamo table, in case the lambda cannot eliminate it due to some error in the system.
lambda.temp_folder	Temporary folder that uses the lambda to write temporary files by default. Usually it is / tmp / which is the temporary default space of a lambda and has a capacity of 512 mb.

As mentioned before, the core class of this lambda is the Kafka_class.py. This class uses the confluent Kafka library to communicate with Kafka. Here is a code snippet for this class:

```
topic_partition = TopicPartition(self.topic, self.kafka_partition)
self.consumer.assign([topic_partition])
records = []
for i in range(0, self.batch_size):
    msg = self.consumer.poll(self.poll_seconds)
    if msg is None:
        self.logger.info("No more messages in queue in poll time")
        break
    elif msg.error() is not None \
            and msg.error().code() is not None \
            and msg.error().code() == KafkaError._PARTITION_EOF:
        self.logger.info("End of messages in partition processed")
        break
    else:
        original_log = msg.value().decode("utf-8")
        data = original_log.replace("\n", "") + "\n"
        self.logger.debug("Offset:" + str(msg.offset()))
        self.logger.debug("Partition:" + str(msg.partition()))
        encode_data = data.encode('utf-8')
        record = {"PartitionKey": str(random.randrange(100)), "Data": encode_data}
        records.append(record)
```

In this code the "assign" method is used to subscribe the lambda to a specific topic, to a specific partition. The alternative is to use the "subscribe" method, but this method can fetch information from several partitions in one invocation, which is not desirable in this case.

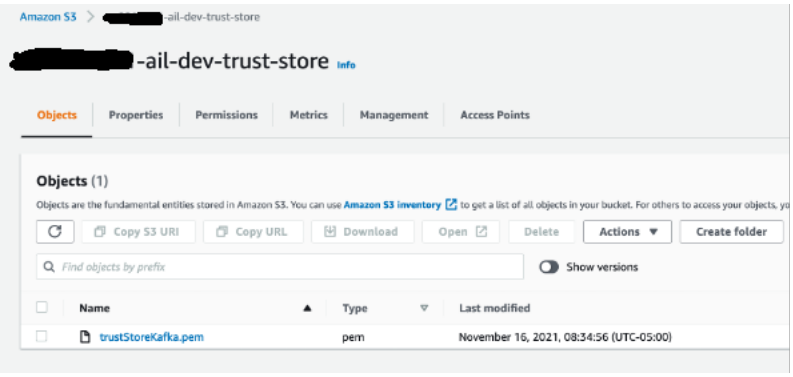
Another important method is the "poll" method, which is the one that allows you to wait in Kafka's queue for a certain number of seconds for new messages, before continuing the process.

In the final part of the method, the content of the message is taken, and a new record is created with a random “Partition Key”, and this is to prepare it for sending to kinesis that this field requires.

With the code attached at: “LambdaKafkaCode” -> “lambdaKafkaIngestor” and with the shown configurations, you can create and deploy the lambda in a test AWS account.

Bucket S3 Truststore

This bucket will contain the truststore file required by the ingesting kafka lambda in order to establish SSL communication with the source kafka. This is what the content of this bucket looks like:



This file that appears in the image must be in PEM format, which is the certificate format supported by Python. Typically, the truststore file is a certificate chain, which is in the form:

```
-----BEGIN CERTIFICATE-----
MIIFOTCCBCGgAwIBAgITTAAAABHFdgrN/8c+swAAAAAAETANBgkqhkiG9w0BAQsF
ADAeMRwwGgYDVQQDExNCQU5DT0xPTUJJQVJPT1RDQUJDMB4XDTIwMDgyNjE3MTE1
.....
iTI/566Hg5unY3jA1SLaDdmKBpdD4UIs8ctijbNc9jQSsBb73Y41HIHrpy7i
-----END CERTIFICATE-----
-----BEGIN CERTIFICATE-----
MIIFOTCCBCGgAwIBAgITTAAAABHFdgrN/8c+swAAAAAAETANBgkqhkiG9w0BAQsF
ADAeMRwwGgYDVQQDExNCQU5DT0xPTUJJQVJPT1RDQUJDMB4XDTIwMDgyNjE3MTE1
.....
BEw0Pz9n+4OAxHoCeqAEGNjTE+UzkuX5Tp5jYb9RTGO+yD/V6OyGS57Hox39v3dt
iTI/566Hg5unY3jA1SLaDdmKBpdD4UIs8ctijbNc9jQSsBb73Y41HIHrpy7i
-----END CERTIFICATE-----
-----BEGIN CERTIFICATE-----
MIIDFzCCAf+gAwIBAgIQXEiYSmsirI5Ov/AsqmfyHjANBgkqhkiG9w0BAQsFADAe
MRwwGgYDVQQDExNCQU5DT0xPTUJJQVJPT1RDQUJDMB4XDTE4MDQxMTAzNDYyNVoX
DTM4MDQxMTAzNTYyNFowHjEcMBoGA1UEAxMTQkFOQ09MT01CSUFST09UQ0FCQzCC
.....
CVr8c432pK0Cl2thII07D6N0vJ6iLgB4Rb49UyKu7fW3v79ItIGPaJKaL4oRCzpv
3YYAf6swmbiJ6k9e0Pj9IDv+/nC/ddKeUIb190b5G1gs2i6E3whc4GDRG9lBr7CU
e27Q3BQFPMk02pbQAnbeRGePO9scc1u9gD9N
-----END CERTIFICATE-----
```

In the case of the example, the red ellipsis are to shorten the space, but basically each certificate begins with the text ----- BEGIN CERTIFICATE ----- and ends with the text ----- END CERTIFICATE-- ---, where the internal content is a certificate in X509 format encoded in base 64.

In this case, it is 3 certificates concatenated in the file, which means that the customer uses 3 trusted certificates in its certification chain.

When building a trustsore file, you must use the same certificates that Kafka has configured in its truststore. If in Kafka these certificates are in a java keystore (.jks) you have to do a process to transform certificates from .jks to .pem format. to be able to use them from Python.

This bucket complies with the security standards required by the customer, which is basically that it is encrypted using a CMK key on the AWS account:

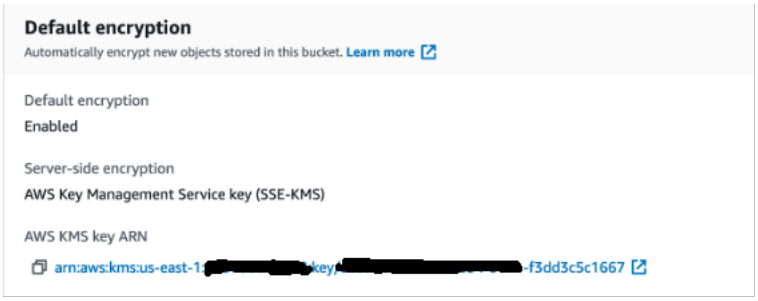
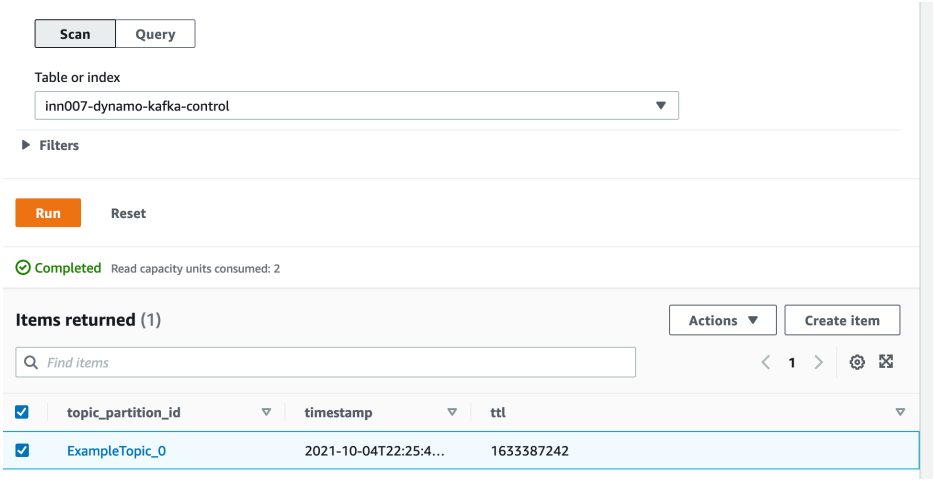


Tabla DynamoDB Control

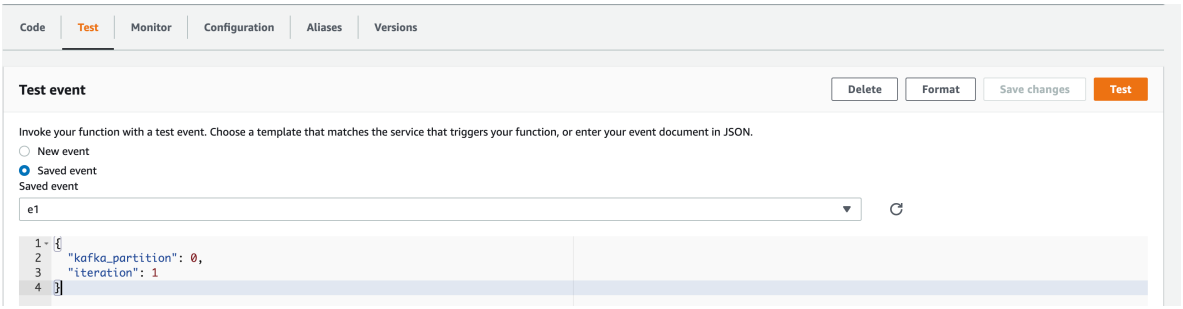
This table was created with the purpose of controlling the consumption of the same partition in the same Kafka topic by two lambdas at the same time, which would produce duplicate records. In this context, the lambda that executes on a specific partition creates a record in the table, and as the value of the key it uses a concatenated of the topic name + "_" + partition id:



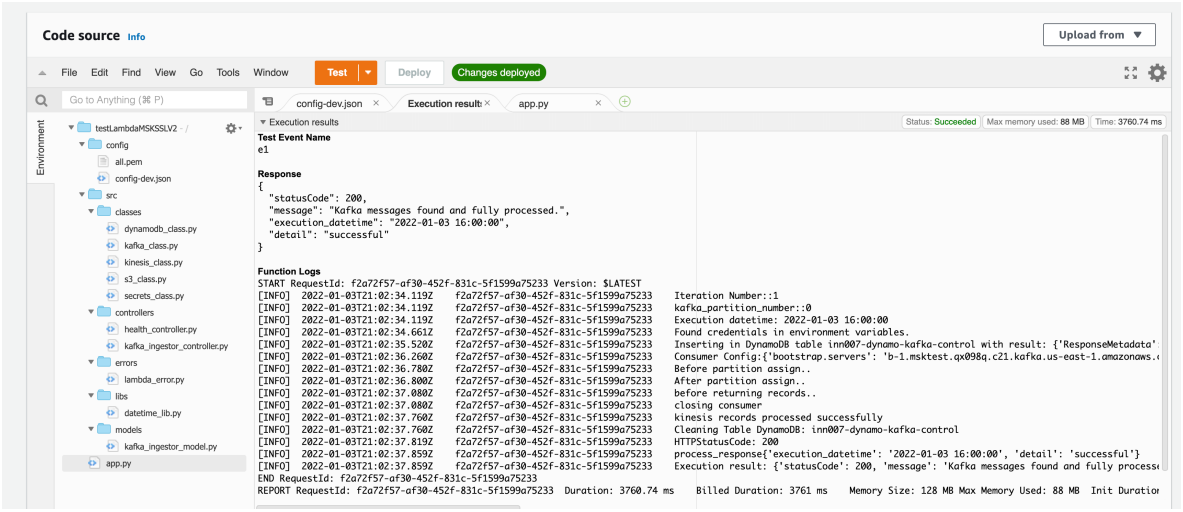
At the end of the process, the lambda clears the record. If by chance there is an error in the lambda, and in the next minute another lambda is invoked to the same partition, this lambda will realize that there is already a record in this lock table, and it will not do anything. This record will be deleted after time to live configured.

Running the lambda kafka ingestor

This lambda receives an input parameter from the step functions which is the partition of kafka that it should consume (they are numbered from 0 to n). In this case there are 3 partitions (from 0 to 2). It also receives a parameter called "iteration" that is more informative, to know in what iteration number the lambda is executed within the minute of execution of the step functions:



After running the lambda, and running the test producer code at the same time as the lambda, you will find records to process in the kafka queue:



In this case it will process maximum 300 records in 1 invocation. Let's see an example. You can use the kafka utility "kafka-consumer-groups.sh" to check the current offset number in each partition in a kafka topic for a given consumer group:

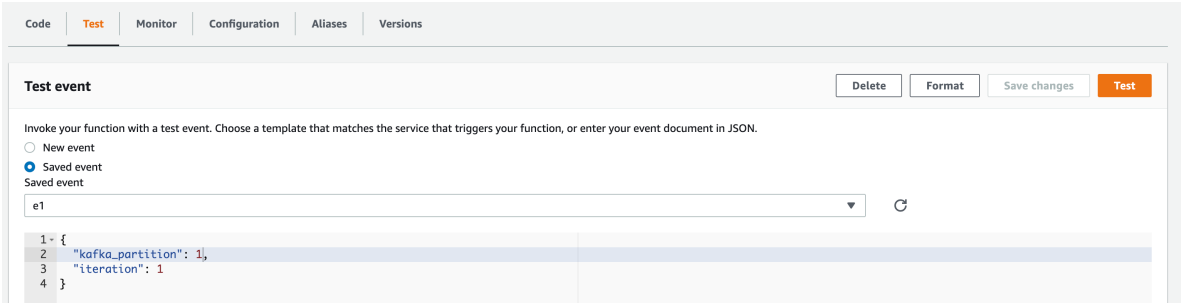
```
[ec2-user@ip-10-0-0-142 kafka_2.12-2.2.1]$ bin/kafka-consumer-groups.sh --bootstrap-server "b-1.msktest.qx098q.c21.kafka.us-east-1.amazonaws.com:9094,b-2.msktest.qx098q.c21.kafka.us-ea
st-1.amazonaws.com:9094" --describe --group "ExampleTopic-G1" --command-config client.properties
Consumer group 'ExampleTopic-G1' has no active members.

TOPIC           PARTITION  CURRENT-OFFSET  LOG-END-OFFSET  LAG             CONSUMER-ID     HOST            CLIENT-ID
ExampleTopic    0          966             9128            8162            -               -               -
[ec2-user@ip-10-0-0-142 kafka_2.12-2.2.1]$
```

As the lambda only ran for the partition "0", only a record in the consumer group has been created. In the next image it was an example of the command, before and after a lambda execution:

```
[ec2-user@ip-10-0-0-142 kafka_2.12-2.2.1]$ bin/kafka-consumer-groups.sh --bootstrap-server "b-1.msktest.qx098q.c21.kafka.us-east-1.amazonaws.com:9094,b-2.msktest.qx098q.c21.kafka.us-ea
st-1.amazonaws.com:9094" --describe --group "ExampleTopic-G1" --command-config client.properties
Consumer group 'ExampleTopic-G1' has no active members.

TOPIC           PARTITION  CURRENT-OFFSET  LOG-END-OFFSET  LAG             CONSUMER-ID     HOST            CLIENT-ID
ExampleTopic    0          966             9128            8162            -               -               -
[ec2-user@ip-10-0-0-142 kafka_2.12-2.2.1]$ bin/kafka-consumer-groups.sh --bootstrap-server "b-1.msktest.qx098q.c21.kafka.us-east-1.amazonaws.com:9094,b-2.msktest.qx098q.c21.kafka.us-ea
st-1.amazonaws.com:9094" --describe --group "ExampleTopic-G1" --command-config client.properties
Consumer group 'ExampleTopic-G1' has no active members.

TOPIC           PARTITION  CURRENT-OFFSET  LOG-END-OFFSET  LAG             CONSUMER-ID     HOST            CLIENT-ID
ExampleTopic    0          1266            9128            7862            -               -               -
[ec2-user@ip-10-0-0-142 kafka_2.12-2.2.1]$
```

As you can see, the first time the current offset was 966, and the lag of messages to attend in the partition was 8162. After the execution that is configured with 300 messages, the new lag is 7862, and the new offset is 1266, so it means the lambda processed the 300 messages successfully.

You can do the same with the partition 1:



When executed over partition 1, it will create a new record in kafka consumer group , this time for that partition:

```
[ec2-user@ip-10-0-0-142 kafka_2.12-2.2.1]$ bin/kafka-consumer-groups.sh --bootstrap-server "b-1.msktest.qx098q.c21.kaf
st-1.amazonaws.com:9094" --describe --group "ExampleTopic-G1" --command-config client.properties
Consumer group 'ExampleTopic-G1' has no active members.

TOPIC           PARTITION  CURRENT-OFFSET  LOG-END-OFFSET  LAG             CONSUMER-ID     HOST            CLIENT-ID
ExampleTopic    0          1266            10425           9159            -               -               -
ExampleTopic    1          12213           12820           607             -               -               -
[ec2-user@ip-10-0-0-142 kafka_2.12-2.2.1]$
```

Note that those records are created the first time a consumer consumes the partition. In this case we have 2 records, one for each partition, and each with its own offset and lag information. If we execute with the third partition, we will have the third record:

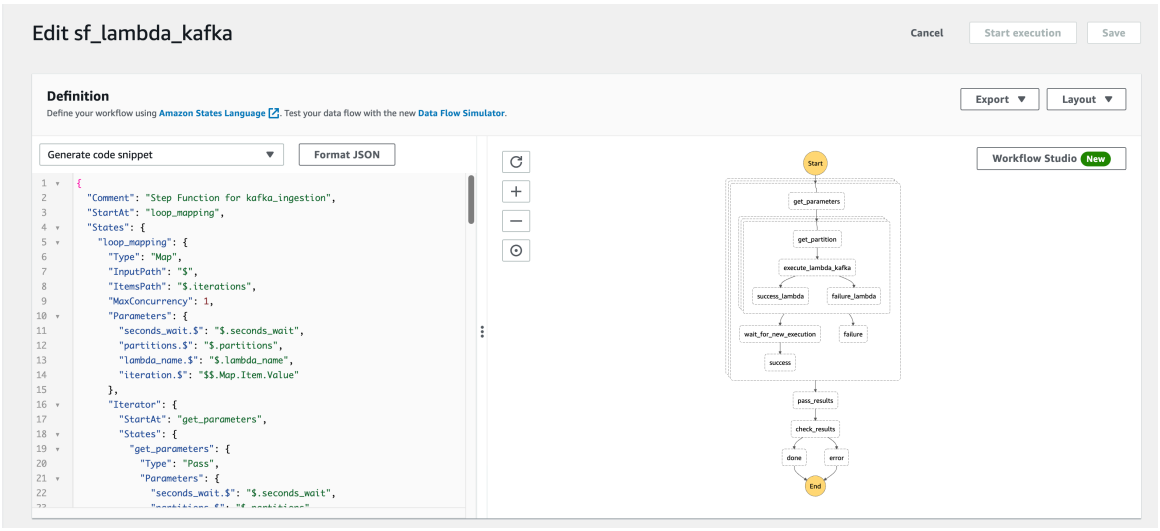
```
[ec2-user@ip-10-0-0-142 kafka_2.12-2.2.1]$ bin/kafka-consumer-groups.sh --bootstrap-server "b-1.msktest.qx098q.c21.kafka.us-east-1.amazonaws.com:9094,b-2.msktest.qx098q.c21.kafka.us-ea
st-1.amazonaws.com:9094" --describe --group "ExampleTopic-G1" --command-config client.properties
Consumer group 'ExampleTopic-G1' has no active members.

TOPIC           PARTITION  CURRENT-OFFSET  LOG-END-OFFSET  LAG             CONSUMER-ID     HOST            CLIENT-ID
ExampleTopic    0          1266            12439           11173           -               -               -
ExampleTopic    2          12611           12792           181             -               -               -
ExampleTopic    1          12213           15070           2857            -               -               -
[ec2-user@ip-10-0-0-142 kafka_2.12-2.2.1]$
```

Writing the step functions controller

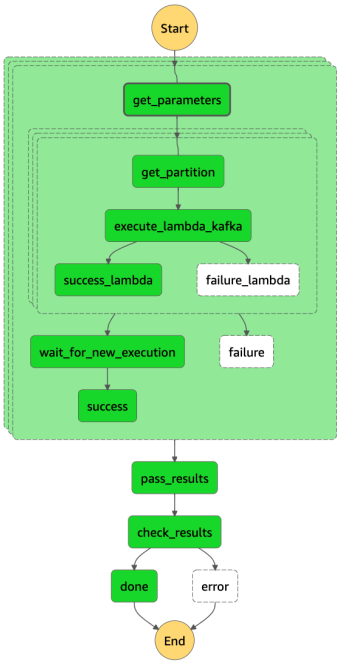
As is mentioned in the architecture explanation, the step functions controller is which invokes the lambda kafka ingestor. This invocation happens N times per minute (called iterations) and each iteration calls the lambda in parallel 3 times (1 per each partition).

You can create the step functions from the AWS web console, using the code attached that you can find in folder “LambdaKafkaCode” -> “StepFunctionsController”. This code is a json file, with the state machine definition required to orchestrate the lambda invocation.

In this example we created a step functions called “sf_lambda_kafka”.



This step functions has the following structure:



Each box in the graph corresponds to a step of step functions. Conceptually, this flow first gets and processes the input that comes to it as a parameter from the cloudwatch rule. The entry is as follows:

```
{
  "iterations": [1,2,3,4] ,
  "partitions":[0,1,2],
  "seconds_wait":5 ,
  "lambda_name": "testLambdaMSKSSLV2"
}
```

So the process takes an array called iterations, and create a loop of 4 iterations in this case. In each cycle, execute the ingestor Kafka lambda as many times as there are elements in the “partitions” array. Depending on the result of each of the 3 lambda instances launched in this case, this flow ends in "success" or "failure". Then wait the number of seconds specified in “seconds_wait” parameter before starting a new cycle. Finally, the parameter “lambda_name” is the name of the lambda to invoke, in this case the ingestor Kafka's lambda.

Each of the steps in the definition of this step functions is explained in detail below:

```
{
  "Comment": "Step Function for kafka_ingestion",
  "StartAt": "loop_mapping",
  "States": {
      "loop_mapping": {
          "Type": "Map",
          "InputPath": "$",
          "ItemsPath": "$.iterations",
          "MaxConcurrency": 1,
          "Parameters": {
            "seconds_wait.$": "$.seconds_wait",
            "partitions.$": "$.partitions",
            "lambda_name.$": "$.lambda_name",
            "iteration.$": "$$.Map.Item.Value"
          },
          "Iterator": {
            "StartAt": "get_parameters",
            "States": {
              "get_parameters":{
                  "Type": "Pass",
                  "Parameters": {
                      "seconds_wait.$": "$.seconds_wait",
                      "partitions.$": "$.partitions",
                      "lambda_name.$": "$.lambda_name",
                      "iteration.$": "$.iteration"
                  },
                  "Next":"parallel_lambda_processing"
              },
```

The process starts in the "loop_mapping" state. This state is of type "Map", which indicates that it is going to iterate over the array of objects specified in "ItemsPath", in this case $ .iterations.

The $ sign in step functions refers to the XPATH language, where $ indicates the root of a json. In the first step, the json is the one that enters the step functions presented above.

Note that the value of "MaxConcurrency" is 1, which indicates that each cycle will run 1 at a time. If this parameter were greater than 1, it would be executed in parallel.

Each "Map" type step has an associated "Iterator", which is a new state machine, which is executed within each iteration. The parameters "seconds_wait", "partitions" and "lambda_name" are passed to this new state machine, which it takes from the input json. The "iteration" parameter is taken from the array with the syntax "$$. Map.Item.Value. In step functions there are special values that start with $$, and indicate that they are context variables of step functions.

The first state of this sub state machine is "get_parameters". This step is only to obtain the variables that the iterator receives, and convert them into new variables within the

state submachine. It is like a variable declaration. Later the "parallel_lambda_processing" state is called:

```
"parallel_lambda_processing": {
        "Type": "Map",
        "InputPath": "$",
        "ItemsPath": "$.partitions",
        "MaxConcurrency": 3,
        "Parameters": {
          "partition.$": "$$.Map.Item.Value",
          "lambda_name.$": "$.lambda_name",
          "iteration.$": "$.iteration"
        },
        "Iterator": {
            "StartAt": "get_partition",
            "States": {
              "get_partition":{
                  "Type": "Pass",
                  "Parameters": {
                      "partition.$": "$.partition",
                      "lambda_name.$": "$.lambda_name",
                      "iteration.$": "$.iteration"
                  },
                  "Next":"execute_lambda_kafka"
              },
```

This state is of type "Map" again, and works exactly the same as the previous block, with the difference that it iterates over the array of "partitions", and in this case the value of "MaxConcurrency" is 3, which means that 3 tasks will be executed at the same time (1 for each partition).

In this case, the Iterator starts a new sub-state machine, and passes it as a parameter the partition, the name of the lambda that it must process and the iteration the process is in. The step “get_partition” makes the respective declaration of variables in this internal state submachine. Subsequently, the step that invokes the lambda Kafka ingestor is invoked:

```
"execute_lambda_kafka": {
      "Type": "Task",
       "Resource": "arn:aws:states:::lambda:invoke",
       "Parameters": {
           "FunctionName.$":"$.lambda_name",
           "Payload":{
             "kafka_partition.$": "$.partition",
             "iteration.$": "$.iteration"
           }
       },
      "Next": "success_lambda",
      "ResultPath": null,
      "Catch": [
          {
            "ErrorEquals": [
              "States.ALL"
            ],
            "ResultPath": "$.Error",
            "Next": "failure_lambda"
          }
      ]
  },
  "success_lambda": {
    "Type": "Pass",
    "Parameters": {
      "Passed.$": "States.Format('Task Succeed: {}', $)"
    },
    "OutputPath": "$.Passed",
    "End": true
```

```
  },
  "failure_lambda": {
    "Type": "Fail",
    "Cause": "Invalid response.",
    "Error": "Lambda Error"
  }
}
```

This is a "Task" type step. All such steps can invoke AWS resources. In this case, the resource arn must be specified in the “Resource” parameter. There are ARNs of special invocations to resources, in this case the arn "arn: aws: states ::: lambda: invoke" allows to invoke a lambda specifying its name as a parameter. In the "Payload" attribute, the input event is passed to the lambda. Later comes a "try / catch" in case the lambda is successful or failed. In this case create a step functions variable called "Passed" and format the output of the lambda together with the text "Task Succeed". Later this text is "returned" to the parent state machine, with the attribute "OutputPath".

At this point execution returns to the second state machine, which invokes the “wait_for_new_execution” step:

```
},
        "ResultPath": null,
        "Next": "wait_for_new_execution",
        "Catch": [
          {
                "ErrorEquals": [
                      "States.ALL"
                    ],
                    "ResultPath": "$.Error",
                    "Next": "failure"
                  }
              ]
          },
"wait_for_new_execution": {
  "Type": "Wait",
  "SecondsPath": "$.seconds_wait",
  "Next": "success"
},
"success": {
  "Type": "Pass",
  "Parameters": {
    "Passed.$": "States.Format('Task Succeed: {}', $)"
  },
  "OutputPath": "$.Passed",
  "End": true
},
"failure": {
  "Type": "Pass",
  "Parameters": {
    "Error.$": "States.Format('Task Failed: {}', $)"
  },
  "OutputPath": "$.Error",
  "End": true
}
```

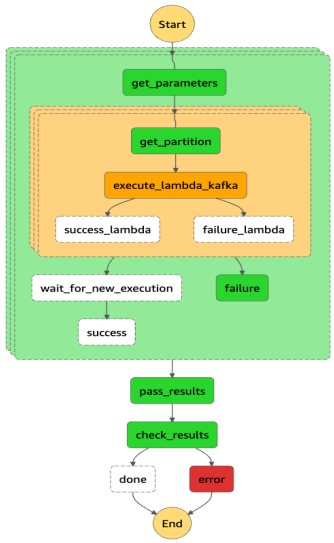
Note that this step is of type “Wait”, and it will wait the number of seconds specified in “seconds_wait”. Like the previous state machine, this machine formats the output text of the previous run, and returns it to the first state machine with the "OutputPath" attribute.

At this point the results of the internal state machines (1 for each partition) have been grouped into a single output variable. This variable is captured in the first state machine, with the attribute "ResultPath":

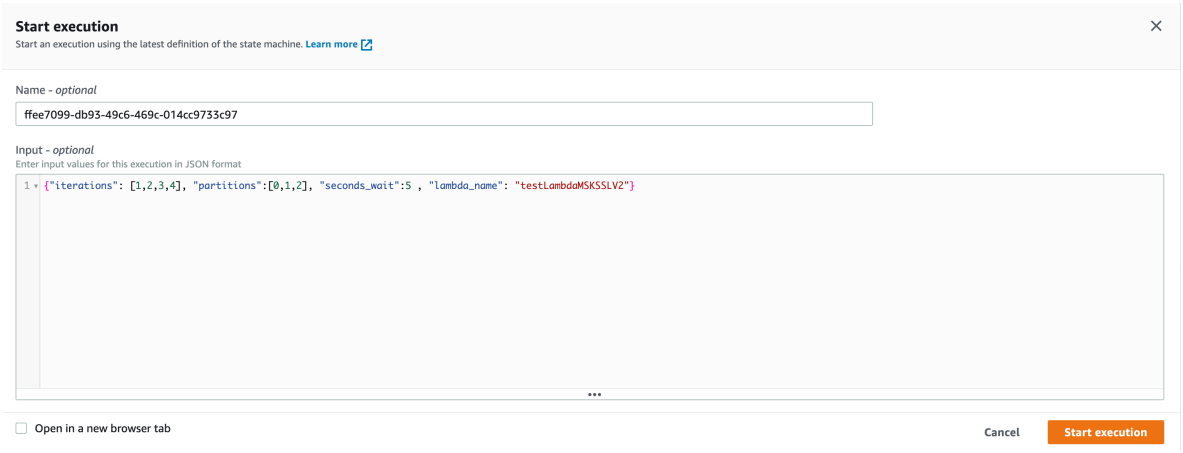
```
}
          },
          "ResultPath": "$",
          "Next": "pass_results"
      },
      "pass_results": {
        "Type": "Pass",
        "Parameters": {
          "output.$": "States.JsonToString($)"
        },
        "Next": "check_results"
      },
      "check_results": {
        "Type": "Choice",
        "Choices": [
          {
            "Variable": "$.output",
            "StringMatches": "*Task Failed*",
            "Next": "error"
          }
        ],
        "Default": "done"
      },
      "done": {
        "Type": "Succeed"
      },
      "error": {
        "Type": "Fail"
      }
  }
}
```

Finally, the step "pass_results" creates a new variable "output" converting all the json resulting from the previous process to String, and invokes "check_results". It basically searches the string for the text "Task Failed", and if it finds it, it goes down the error path, causing the whole step functions process to fail. If everything was successful, the process ends in green, thus ending this 1 minute cycle.

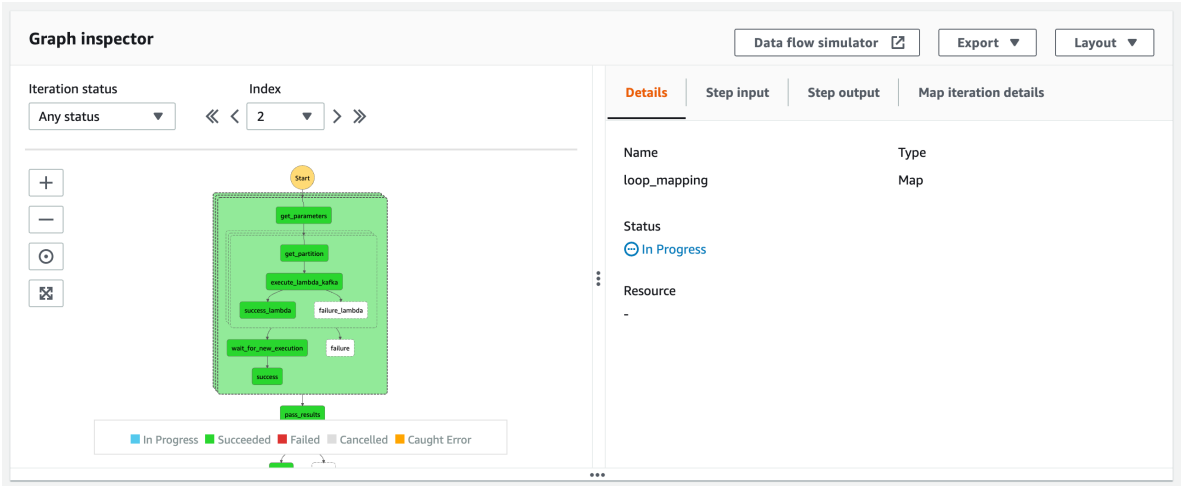
The step functions console allows you to see in real time the execution status of the flow, and the inputs and outputs of each step. If the execution of a complete flow of execution was successful, the flow will be shown in green. If the process fails in any of the steps, the execution will be shown in red, and the remaining flow will not be executed according to the defined logic:



If we want to execute the step functions manually, we can provide this input to the new execution:



At execution time, the step functions web console lets you inspect each iteration behavior, pressing the dotted squares in the flow, and using the “index” section to choose the desired iteration of the map task:



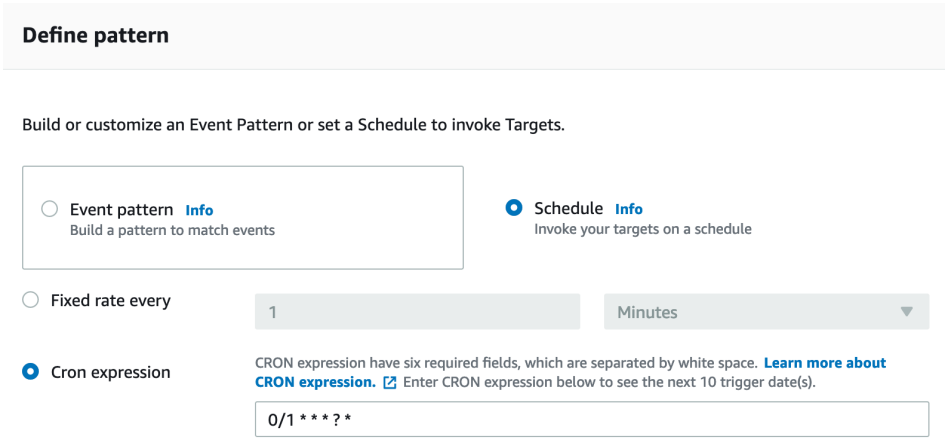
Once you execute the step functions manually, you can inspect the kafka queue information again, to be sure that is consuming messages from the 3 partitions:

```
[ec2-user@ip-10-0-0-142 ~]$ ls
kafka_2.12-2.2.1  kafka_2.12-2.2.1.tgz  python_test
[ec2-user@ip-10-0-0-142 ~]$ cd kafka_2.12-2.2.1/
[ec2-user@ip-10-0-0-142 kafka_2.12-2.2.1]$ bin/kafka-consumer-groups.sh --bootstrap-server "b-1.msktest.qx098q.c21.kafka.us-east-1.amazonaws.com:9094,b-2.msktest.qx098q.c21.kafka.us-east-1.amazonaws.com:9094" --describe --group "ExampleTopic-G1" --command-config client.properties
Consumer group 'ExampleTopic-G1' has no active members.

TOPIC           PARTITION  CURRENT-OFFSET  LOG-END-OFFSET  LAG             CONSUMER-ID     HOST            CLIENT-ID
ExampleTopic    0          2466            14786           12320           -               -               -
ExampleTopic    2          13811           14311           500             -               -               -
ExampleTopic    1          13413           16204           2791            -               -               -
[ec2-user@ip-10-0-0-142 kafka_2.12-2.2.1]$ bin/kafka-consumer-groups.sh --bootstrap-server "b-1.msktest.qx098q.c21.kafka.us-east-1.amazonaws.com:9094,b-2.msktest.qx098q.c21.kafka.us-east-1.amazonaws.com:9094" --describe --group "ExampleTopic-G1" --command-config client.properties
Consumer group 'ExampleTopic-G1' has no active members.

TOPIC           PARTITION  CURRENT-OFFSET  LOG-END-OFFSET  LAG             CONSUMER-ID     HOST            CLIENT-ID
ExampleTopic    0          3666            14786           11120           -               -               -
ExampleTopic    2          14311           14311           0               -               -               -
ExampleTopic    1          14613           16204           1591            -               -               -
[ec2-user@ip-10-0-0-142 kafka_2.12-2.2.1]$ bin/kafka-consumer-groups.sh --bootstrap-server "b-1.msktest.qx098q.c21.kafka.us-east-1.amazonaws.com:9094,b-2.msktest.qx098q.c21.kafka.us-east-1.amazonaws.com:9094" --describe --group "ExampleTopic-G1" --command-config client.properties
Consumer group 'ExampleTopic-G1' has no active members.

TOPIC           PARTITION  CURRENT-OFFSET  LOG-END-OFFSET  LAG             CONSUMER-ID     HOST            CLIENT-ID
ExampleTopic    0          4866            14786           9920            -               -               -
ExampleTopic    2          14311           14311           0               -               -               -
ExampleTopic    1          15813           16204           391             -               -               -
[ec2-user@ip-10-0-0-142 kafka_2.12-2.2.1]$
```

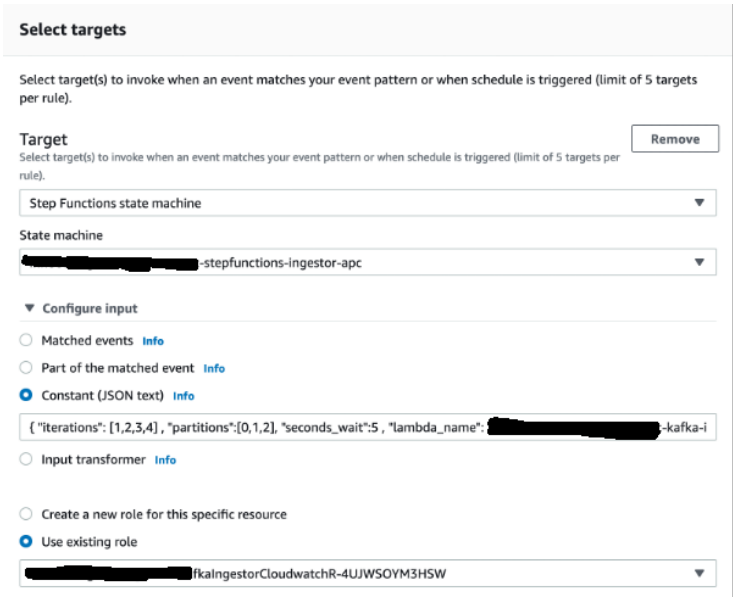
In the example above you can see the execution of the step functions 2 times. For partition 0, first time it lowers the lag value from 12320 messages to 11120 messages, which means it processed 1200 records. It is fine, because it calls the partition 0 four times (1 per iteration) . in the second execution it lowers the lag to 9920 records, processing the same 1200 messages. We can do the same math using the partition 1. As the partition 2 started with a lag value of 500, it processed in the first call.

Cloudwatch Rule to trigger the process

The last component associated with the ingestor Kafka lambda is the cloudwatch rule that launches the previous step functions. In this case, this rule is of type "schedule" which is based on crond Linux syntax. In this case, you have a cron expression configured that executes every 1 minute.



This rule also has an associated target, which is pointing to the step functions explained above:



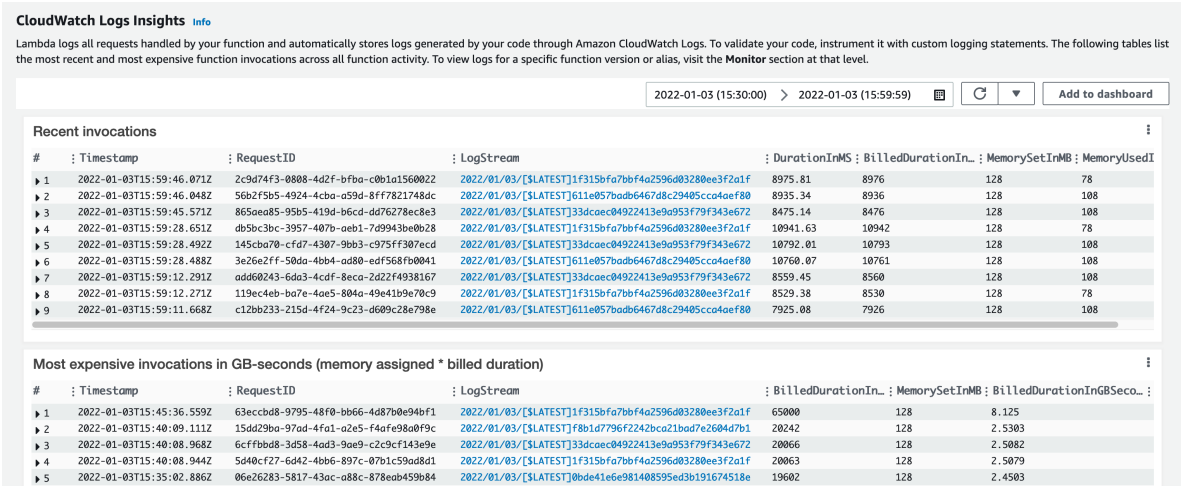
In this case, the input json that is sent to the step functions can be seen in the "Constant" field, an input that has been explained in detail in the previous section of step functions.

Production environment response times and costs

As this solution is running today in a customer's production environment, it's relevant to talk about response times and costs of this solution. One of the customer's concerns was the lambda cost. Comparing the projected lambda cost (USD $21) versus the minimum cost in ec2 or eks (USD $110) using the AWS cost calculator, the lambda appears to be better. In the first month of the solution in production environment, the associated costs are:

Lambda		**$11.52**
US East (N. Virginia)		**$14.05**
AWS Lambda Lambda-GB-Second		$8.03
AWS Lambda - Compute Free Tier - 400,000 GB-Seconds - US East (Northern Virginia)	4,899.013 seconds	$0.00
AWS Lambda - Total Compute - US East (Northern Virginia)	694,981.814 seconds	$11.58
AWS Lambda-Lambda-GB-Second usage covered by Compute Savings Plan	213,243.087 seconds	-$3.55
AWS Lambda Request		$6.02
AWS Lambda - Requests Free Tier - 1,000,000 Requests - US East (Northern Virginia)	11,375.000 Requests	$0.00
AWS Lambda - Total Requests - US East (Northern Virginia)	30,085,379.000 Requests	$6.02
EDP Discounts		**-$2.53**

Which is even less than projected in initial estimation. Regarding the time responses, it varies according the kafka location. If the kafka is deployed in AWS, the response times can vary between 3 and 5 seconds. Against the kafka onpremises, the response time can vary between 8 and 20 seconds depending on the ompremises – AWS bandwith:



CloudWatch Logs Insights Info

Lambda logs all requests handled by your function and automatically stores logs generated by your code through Amazon CloudWatch Logs. To validate your code, instrument it with custom logging statements. The following tables list the most recent and most expensive function invocations across all function activity. To view logs for a specific function version or alias, visit the **Monitor** section at that level.

2022-01-03 (15:30:00) > 2022-01-03 (15:59:59) | Add to dashboard

Recent invocations

#	Timestamp	RequestID	LogStream	DurationInMS	BilledDurationIn…	MemorySetInMB	MemoryUsedI
1	2022-01-03T15:59:46.071Z	2c9d74f3-0808-4d2f-bfba-c0b1a1560022	2022/01/03/[$LATEST]1f315bfa7bbf4a2596d03280ee3f2a1f	8975.81	8976	128	78
2	2022-01-03T15:59:46.048Z	56b2f5b5-4924-4cba-a59d-8ff7821748dc	2022/01/03/[$LATEST]611e057badb6467d8c29405cca4aef80	8935.34	8936	128	108
3	2022-01-03T15:59:45.571Z	865aea85-95b5-419d-b6cd-dd76278ec8e3	2022/01/03/[$LATEST]33dcaec04922413e9a953f79f343e672	8475.14	8476	128	108
4	2022-01-03T15:59:28.651Z	db5bc3bc-3957-407b-aeb1-7d9943be0b28	2022/01/03/[$LATEST]1f315bfa7bbf4a2596d03280ee3f2a1f	10941.63	10942	128	78
5	2022-01-03T15:59:28.492Z	145cba70-cfd7-4307-9bb3-c975ff307ecd	2022/01/03/[$LATEST]33dcaec04922413e9a953f79f343e672	10792.01	10793	128	108
6	2022-01-03T15:59:28.488Z	3e26e2ff-50da-4bb4-ad80-edf568fb0041	2022/01/03/[$LATEST]611e057badb6467d8c29405cca4aef80	10760.07	10761	128	108
7	2022-01-03T15:59:12.291Z	add60243-6da3-4cdf-8eca-2d22f4938167	2022/01/03/[$LATEST]33dcaec04922413e9a953f79f343e672	8559.45	8560	128	108
8	2022-01-03T15:59:12.271Z	119ec4eb-ba7e-4ae5-804a-49e41b9e70c9	2022/01/03/[$LATEST]1f315bfa7bbf4a2596d03280ee3f2a1f	8529.38	8530	128	78
9	2022-01-03T15:59:11.668Z	c12bb233-215d-4f24-9c23-d609c28e798e	2022/01/03/[$LATEST]611e057badb6467d8c29405cca4aef80	7925.08	7926	128	108

Most expensive invocations in GB-seconds (memory assigned * billed duration)

#	Timestamp	RequestID	LogStream	BilledDurationIn…	MemorySetInMB	BilledDurationInGBSeco…
1	2022-01-03T15:45:36.559Z	63eccbd8-9795-48f0-bb66-4d87b0e94bf1	2022/01/03/[$LATEST]1f315bfa7bbf4a2596d03280ee3f2a1f	65000	128	8.125
2	2022-01-03T15:40:09.111Z	15dd29ba-97ad-4fa1-a2e5-f4afe98a0f9c	2022/01/03/[$LATEST]f8b1d7796f2242bca21bad7e2604d7b1	20242	128	2.5303
3	2022-01-03T15:40:08.968Z	6cffbbd8-3d58-4ad3-9ae9-c2c9cf143e9e	2022/01/03/[$LATEST]33dcaec04922413e9a953f79f343e672	20066	128	2.5082
4	2022-01-03T15:40:08.944Z	5d40cf27-6d42-4bb6-897c-07b1c59ad8d1	2022/01/03/[$LATEST]1f315bfa7bbf4a2596d03280ee3f2a1f	20063	128	2.5079
5	2022-01-03T15:35:02.886Z	06e26283-5817-43ac-a88c-878eab459b84	2022/01/03/[$LATEST]0bde41e6e981408595ed3b191674518e	19602	128	2.4503

Note that sometimes (most expensive invocations) the channel is very busy, and the lambda fails with timeout. Other times it can take 20 seconds to execute. This is because network behavior between on-premises and AWS customer network. In this case the lambda will use the dynamodb control table to avoid duplicates.

How customers can use this solution? Advantages / Disadvantages

In any situation you need to ingest data from a Kafka server to a datalake, regardless if it's on premises or over any cloud, you can use this artifact. However, keep in mind the following advantages / disadvantages when using it:

Advantages

- If the key chain of the TLS certificate that enables the communication between Kafka and lambda is too big, you can use this artifact instead to use the generic kafka trigger, because has the option to use the truststore from a secured s3 bucket instead using secrets manager which has a size limit.

- You can use it when you need highest flexibility in configure the confluent kafka options described in : https://docs.confluent.io/platform/current/clients/confluent-kafka-python/html/index.html

- Whit this option you can control the number of times the lambda kafka ingestor can be executed.

- It reduces the execution costs over 80% in comparison with EC2 or ECS/EKS solutions with the same functionality.

Disadvantages

- It requires 5 services to maintain the solution: Cloudwatch, step functions, lambda, dynamo and s3, compared with the lambda trigger that only needs the lambda service.

- The code sample is provided as is, without any warranty , and is not an official AWS service.

Conclusion

It is totally possible to mount a lambda based kafka ingestion pipeline with the presented architecture, at a very low cost. For the project success, is very important to estimate the data ingestion volume vs the network capacity, especially if the communication is against onpremises kafka.