CONFLUENT

Stateful Serverless Architectures with ksqlDB and AWS Lambda

Bill Bejeck

Table of Contents

Introduction

While the term "serverless" is not new, its meaning is still not clearly defined across the industry. The term doesn't mean that you can run an application without a server, but that your concerns are focused on the application and the application only: You don't have to be worried with infrastructure, because once you build the application, it gets deployed in an environment suitable to handle the expected load.

This approach can yield tremendous benefits to your business because it lets you focus on the core issues and not on the "ceremony" needed to self-host applications. Additionally, the barrier to entry for any serverless technology is significantly lower than its traditional counterpart due to the reduction in technical knowledge required to make it function properly. Another advantage is serverless's customary pay-as-you-go nature: You are only billed for execution time, not for time spent idling between function calls.

Although serverless technologies reduce the stress associated with infrastructure, they don't all solve the same application problems. By combining them though, you can capitalize on their respective strengths.

Combining Serverless Technologies

A powerful serverless combination is ksqlDB, a database from the Apache Kafka ecosystem that is purpose-built for stream processing applications, matched with AWS Lambda, an event-driven, highly scalable compute service.

ksqlDB is capable of handling both stateful and stateless workloads, but its strength lies in stateful processing to answer complex questions. On the other hand, due to the nature of its transient processing, AWS Lambda is better suited for stateless processing tasks, as well as tasks that require extensive horizontal scalability. Given the respective strengths of the technologies, combining the two is often the best answer to solve a specific problem.

Imagine you have a ksqlDB streaming application that checks for anomalies in an event stream of purchases. When the application determines that it has found a suspicious event, it writes that event out to a topic. In addition, you'd like to notify the customer in question about the suspicious activity. What you'd do is create an instance of a `RequestHandler<I, O>` from the AWS Lambda Java API (or you'd use another language API) to send out the response.

Given that the majority of customers should have activities that fall into expected use patterns, the somewhat infrequent need to create suspicious activity events also plays into the strengths of AWS Lambda because charges only accrue when it's used. And AWS Lambda easily supports spikes and sustained increases in the number of records.

ksqlDB

ksqlDB allows you to develop applications that respond immediately to the events streaming from your Apache Kafka cluster. It provides you with the ability to build applications using SQL, which will be familiar to developers with a wide variety of backgrounds.

Using SQL to Create Streams and Tables

Stateless event stream processing with ksqlDB is straightforwardly accomplished. For example, you could create a stream that reports the distances of trains in a system from their respective destinations:

```
CREATE STREAM LOCATIONS AS
  SELECT rideId, latitude, longitude,
         GEO_DISTANCE(latitude, longitude,
                      dstLatitude, dstLongitude, 'km'
                      ) AS kmToDst
 FROM geoEvents
```

Then if you take the stream from above, you can create a stateful one to track longer train rides over ten kilometers:

```
CREATE TABLE RIDES_OVER_10K AS
    SELECT rideId,
           COUNT(*) AS LONG_RIDES
    FROM LOCATIONS
    WHERE kmToDst > 10
    GROUP BY rideId
```

Maintaining State

ksqlDB uses RocksDB to persist records to local disk on the ksqlDB server. In this way, ksqlDB can maintain a running status of the stream state by constantly updating a materialized view.

This state is made durable through the use of changelog topics. When ksqlDB writes a stateful result to RocksDB, the same record is persisted to a changelog topic backing the store. Should the ksqlDB server experience a failure, thus losing the RocksDB store, data is not lost, it's safely stored in a replicated changelog topic. So when a new ksqlDB instance starts to replace the failed one, it will replay all the

data to populate its RocksDB instance and resume processing queries on the materialized view.

While both stateless and stateful queries in ksqlDB are simple to write, they are shielding you from a lot of processing power: Under the covers, there is a ksqlDB cluster communicating with a Kafka cluster. Both clusters can scale up to handle nearly any volume of incoming events.

AWS Lambda

AWS Lambda, an integral part of the AWS Serverless Application Model (AWS SAM), is a Function as a Service (FaaS) that gives you the ability to make a discrete chunk of code available to run in response to certain events. The code will only execute when it needs to, and will otherwise sit dormant. For example, you may have a function trigger each time a new document is uploaded, checking it for sensitive information such as phone numbers, credit card numbers, addresses, or any other such information.

Attributes

Lambdas can be triggered directly using a tool such as a CLI, by another AWS resource, or by an event source mapping, which is a resource within a Lambda that reads items from a stream and calls a function in response. Event source mappings are the logical way to interface with Apache Kafka.

When using an event source mapping, the Lambda polls for new messages from the source, then calls the target function synchronously, providing as an event payload the messages read in as a batch. The maximum batch size is configurable, with a default of 100 (see "Performance Implications — Experimenting with 1000 Lambdas," below).

To preserve ordering of event processing, only one Lambda function instance may process events from a Kafka topic partition at a time. Functions need to complete processing of events within 15 minutes.

Credentials can be accessed by Lambdas using AWS Secrets Manager and there are provisions for longer running components accessible across Lambdas. See more about these aspects in "Security Configuration" and "When to Create an Apache® Kafka Producer Instance."

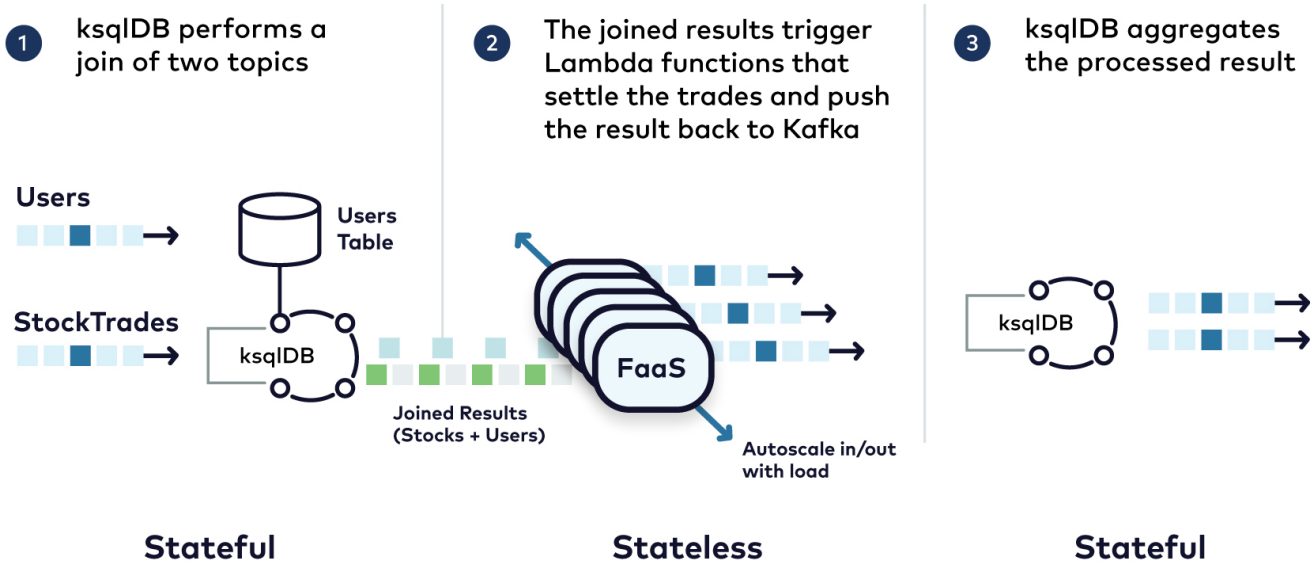
Procedure

To run a Lambda, you upload your code packaged as a ZIP file, or a container image, and you create a Lambda function definition, including the code to run and the triggering mechanism. The Lambda service takes care of everything else such as scaling out the number of active instances when the load is high and determining the optimum amount of computing power needed for your code to get the job done.

AWS Lambda and ksqlDB Combined

The advantages and strategies of the AWS Lambda/ksqlDB combination are best illustrated with a concrete example. You can clone the corresponding GitHub repo and run it with the `ccloud-build-app.sh` script.

The scenario represented is a full end-to-end example of integrating a ksqlDB application with an AWS Lambda for two-way communication: ksqlDB performs some work and writes the result to a Kafka topic. The Lambda function does some processing on the result and writes a new result back to a topic on Confluent Cloud. ksqlDB has additional long-running queries to analyze the results of the Lambda output.



Description of the Application Flow

From a high-level perspective, the flow of the application is as follows:

The ksqlDB application runs on Confluent Cloud and leverages the managed Datagen connector to create a stream and a table. The SQL for creating the stream looks like this:

```
CREATE STREAM STOCKTRADE (side varchar, quantity int,
symbol varchar, price int, account varchar, userid varchar)
with (kafka_topic = 'stocktrade', value_format = 'json');
```

The `STOCKTRADE` stream represents simulated stock trades. The SQL for creating the `USERS` table containing customer information takes the form of:

```
CREATE TABLE USERS (userid varchar primary key,
                    registertime BIGINT, regionid varchar )
with ( kafka_topic = 'users', value_format = 'json');
```

ksqlDB then performs a stream-table join:

```
CREATE STREAM USER_TRADES WITH (kafka_topic = 'user_trades')
AS SELECT s.userid as USERID, u.regionid,
quantity, symbol, price, account, side
FROM STOCKTRADE s LEFT JOIN USERS u on s.USERID = u.userid;
```

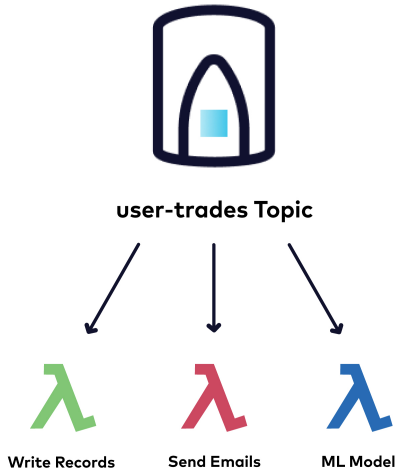
The join enriches the stock trade data with information about the user executing the trade.

We use a left outer join so that we'll always have the trade information, regardless of whether or not the user information exists yet at that point in time.

ksqlDB writes the results of the join to a topic named `user-trades`. This topic serves as the event source mapping for an AWS Lambda. In this case, we're using the Lambda as a proxy for an external process—settling the trade. The Lambda code creates a Protobuf object, `TradeSettlement`, containing one of four disposition types based on the information contained in the original stock trade transaction: `Rejected`, `Pending`, `Flagged`, and `Completed`. Then the Lambda produces the completed `TradeSettlement` object back to a topic in Confluent Cloud named `trade-settlements`.

What we've described here just scratches the surface of what you can do from inside a Lambda. You could also opt to:

- write records to different topics, based on the disposition code
- send emails directly to customers when the status of the trade is negative
- contact an ML model to check the trade for fraud or suspicious activity



The point is, the Lambda presents an opportunity for you to perform per-trade business logic without having to manage any of the infrastructure.

For the last leg of the serverless processing, the ksqlDB application creates a stream from the `trade-settlements` topic. This stream serves as the source for four tables that calculate the total number of results for each status over the last minute using a tumbling window. For example, here's the SQL for determining the number of fully settled trades:

```
CREATE TABLE COMPLETED_PER_MINUTE AS
SELECT symbol, count(*) AS num_completed
FROM TRADE_SETTLEMENT WINDOW TUMBLING (size 60 second)
WHERE disposition like '%Completed%'
GROUP BY symbol
EMIT CHANGES;
```

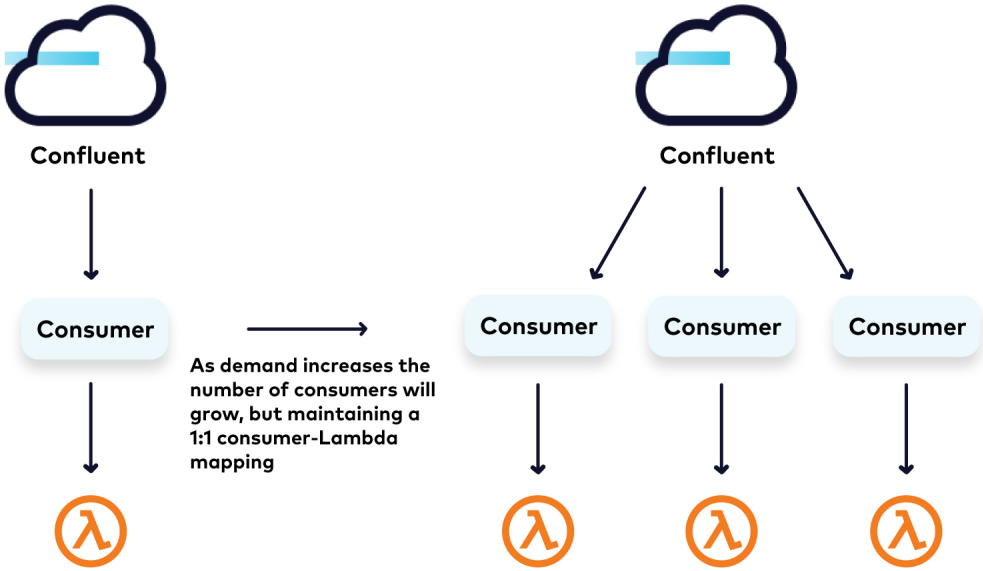
This `COMPLETED_PER_MINUTE` table can then be queried interactively. For example, you could use a pull query to find trades for the symbol CFLT:

```
SELECT * FROM COMPLETED_PER_MINUTE WHERE SYMBOL='CFLT'
```

Performance Implications — Experimenting with 1000 Lambdas

In order to seamlessly scale in the face of increasing demand, AWS monitors the progress of your underlying Kafka consumer. Should the consumer begin to lag, AWS will create a new consumer and

Lambda instance to help handle the load. Note that the maximum number of consumers is equal to the number of partitions. There is also only one Lambda instance per consumer, to ensure that all events in a partition are processed in sequential order.



Currently, there is a soft limit of 100 consumers. If you need maximum parallelization above 100 instances, you will need to file a service ticket with AWS to increase the limit for you.

To that end, in our working example we wanted to see how the architecture would respond to a topic with 1,000 partitions, containing a large amount of records to process.

First, we coordinated with the AWS service team to have them set the concurrency limit to 1,000 Lambda instances. To force increasing horizontal scaling, we made changes to the Lambda by first setting the batch size to one (definitely ***not a production*** value!) and adding an artificial wait of one second to the method handler code. These values were selected to simulate a high instance workload, without having to produce (and consume) billions of records. For our event data, we produced 50M records to a topic of 1K partitions, ensuring an even distribution of 50K records per partition.

With the test setup completed, we were now ready to start the AWS Lambda instances and begin the test. Initially, it started with one consumer and Lambda instance responsible for all 1,000 partitions. But at a rate of one record every five seconds, that was a level of progress that wouldn't do. After the first rebalance (approximately 15 minutes into the test), 500 consumer/Lambda pairs sprang into action, yielding a 500x processing improvement to 500 records every five seconds. Every few minutes (five to ten) the Lambda service continually added Lambda instances, ultimately reaching the target number.

That's 1,000 consumer/Lambda pairs working in concert. Now that's some serious parallel processing! What started at one record per five seconds ended up with 1,000 records per five seconds, significantly increasing the progress made through the topic backlog. While this is a contrived example and doesn't

reflect a realistic production setting, that's not the main point here. The main point to consider is that under a considerable increase of required processing power, this architecture has the elasticity to respond and meet those demands and then reduce capacity ***automatically*** after completing the surge in workload.

When to Create an Apache® Kafka Producer Instance

We mentioned during the high-level application description that the Lambda will produce records back to a topic on Confluent Cloud. Using a Kafka Producer from within an AWS Lambda is straightforward, but there is an important detail to consider (a detail not unique to Kafka, but rather related to any long-living connection, such as a database).

When building a Lambda instance, you can initialize any long-lived resources in the constructor, inline at the field level, or in a static initializer block. Any objects created this way will remain in memory and the Lambda will reuse them, potentially across thousands of invocations. In the reference example the producer is declared at the class level and initialized in the constructor as shown below (some details left out for clarity):

```
public class CCloudStockRecordHandler implements RequestHandler<Map<String, Object>, Void> {

//Once initialized, producer is reusable for future invocations
    private final Producer<String, TradeSettlementProto.TradeSettlement> producer;

    private final StringDeserializer stringDeserializer = new StringDeserializer();

    public CCloudStockRecordHandler() {
        producer = new KafkaProducer<>(configs);
        stringDeserializer.configure(configs, false);
    }
}
```

By creating the producer instance this way, it's used across executions for the life of the Lambda instance. It's important that you never create a producer instance in the handler method itself as this will end up creating potentially thousands of producer clients and putting an unnecessary strain on the brokers.

Security Configuration

When using Kafka as an event source for an AWS Lambda, you'll need to provide the user name and secret for the Confluent Cloud Kafka cluster via an AWS Secrets Manager. Part of setting up a

Confluent Cloud cluster is generating the key and secret to enable client access to the cluster (note that the working example generates these for you).

But using the AWS Secrets Manager also presents an opportunity to store all of the sensitive information needed to connect to various components in the Kafka cluster such as endpoints and authentication for ksqlDB and Schema Registry. By placing all these settings in the Secrets Manager, it makes it seamless to configure the producer and any Schema Registry (de)serializers inside the Lambda instance.

The AWS SDK provides the SecretsManagerClient, which makes it easy to programmatically retrieve the data stored in the Secrets Manager. Let's take another look at the constructor for your Lambda instance:

```
public class CCloudStockRecordHandler implements RequestHandler<Map<String, Object>, Void> {
    private final Producer<String, TradeSettlementProto.TradeSettlement> producer;

    private final Map<String, Object> configs = new HashMap<>();

    private final StringDeserializer stringDeserializer = new StringDeserializer();

    public CCloudStockRecordHandler() {
        configs.putAll(getSecretsConfigs());

        producer = new KafkaProducer<>(configs);
        stringDeserializer.configure(configs, false);
    }
}
```

At the class level, you've defined a `HashMap` named `configs`. Then in the constructor, you're using the method `getSecretsConfigs` which leverages the `SecretsManagerClient` to retrieve all the necessary connection information and store it in the `configs` object. Now you can easily provide the required connection credentials to any object that requires them.

Kafka Records Payload

As mentioned above, the Lambda service delivers records in batches (the default batch size is 100) to your `RequestHandler` instance via the `handleRequest` method. The signature of `handleRequest` consists of two parameters: a `Map<String, Object>` and a `Context` object. The `Map` contains a mix of object types for the values, hence the `Object` generic for the value type of the map. The `Context` object provides access inside the Lambda execution environment, such as a logger you can use to send information to AWS CloudWatch.

For our purposes, we're most interested in the `records` key, which points to a value of `Map<String, List<Map<String, String>>>`. The keys of the records map are the topic-partition names and the

values are a list of map instances, where each map in the list contains a key-value pair from the topic.

You'll notice the types on the `Map` are `String` for both the key and the value. But this doesn't represent the actual types of records from the topic. The Lambda service converts the key and value byte arrays into base64 encoded strings. So to work with the expected key and value types, you'll need to first base64 decode them back into byte arrays and then use the appropriate deserializer. In the case of this example, we're expecting JSON, so it uses the `StringDeserializer`.

Ensuring Message Delivery

When executing the `KafkaProducer#send` method, the producer does not immediately forward the record to the broker. Instead, it puts the record in a buffer and forwards a batch of records when either the batch is full or when it determines it's time to send them. When using a Kafka Producer from within an AWS Lambda, it's important that you execute `KafkaProducer#flush` as the last action the Lambda takes before exiting. Not doing so risks that records don't get sent. But it's important that you call flush ***only*** once at the end of the handler method (when you've fully processed the entire record batch) and not after each call to `KafkaProducer#send`. And it's always a best practice to set `acks=all` to ensure record durability.

Conclusion

Now that you've learned how combining ksqlDB and AWS Lambda gives you a powerful, serverless one-two punch, it's time to learn more about building your own serverless applications. Head over to Confluent Developer and check out the ksqlDB introduction course and the Inside ksqlDB course for a deeper understanding of the fundamentals. Then go over to Confluent Cloud and start building a serverless event streaming application. You can use the promo code `CLOUD50` to get $50 of free Confluent Cloud usage *.

Also, check out building an AWS Lambda in Java and the AWS Lambda documentation for a full understanding of the Lambda landscape.