Real-Time Data Streaming with Python + AWS Kinesis — How To... (part 1)

Tom Thornton Follow
May 22, 2019 · 14 min read ★

In this article we will be focusing on the use of AWS Kinesis with Python and Node.js to stream data in near real-time to ElasticSearch.

Kinesis provides the infrastructure for high-throughput data processing and analytics. We can leverage this to push data through to storage/visualisation services for aggregation and historical interrogation.

In this guide we will be using Python 3.6 and AWS' boto3, pandas and inbuilt functions.

In Part 1, we will discuss each of the segments of the Kinesis service, what you can use them for and finally walkthrough a worked example of streaming CSV data with Kinesis to AWS' ElasticSearch service for visualisation in Kibana dashboards. This will cover the necessary code to stream data from source to a Kinesis Data Stream and to transform/push data to ElasticSearch using a Kinesis Firehose.

Every AWS service I will be working with are all within the same region (eu-west-1 for me) so you may have to make adjustments to suit your purposes.

. . .

Kinesis Data Streams

Data streams are like endpoints which accept batches of data to be processed. They provide “shards” which you can think of like processing nodes. The more shards, the more records you can process in parallel and therefore reducing total runtime to process the same number of records. The Data Stream ultimately provides a quick and easy way of bussing your data to a consumer. Too little shards and your data may end up in a dead letter queue, which means you don’t have enough shards for the volume of

data and they are becoming saturated thus rejecting any further data. By default Data Streams will hold data for 24 hours, but this can be modified up to 168 hours. This allows your applications to continue processing items in the Data Stream should the consumer have an issue. You can also provide a timestamp when querying for data from the stream to play back events that have been processed in the past.

. . .

Kinesis Firehoses

Firehoses are the opposite of Data Streams, they distribute records/data to specified endpoints configured in the Firehose from a specified source. In a Firehose, we can also specify any transformations or formatting we'd like to apply to the data before it's pushed out. We will be using Firehose later on to connect our Data Stream to ElasticSearch and provide the transformation to JSON logic.

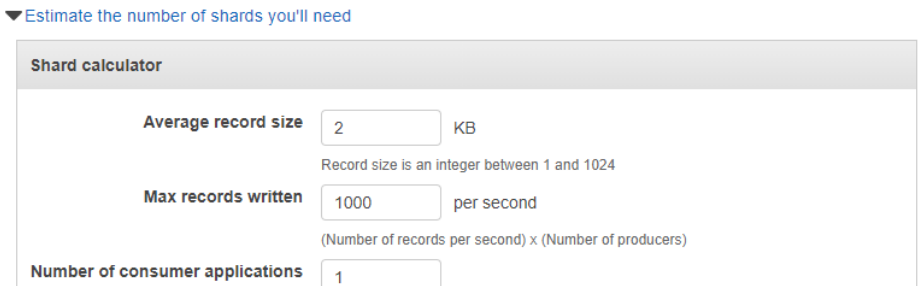
Firehose provides the ElasticSearch Service domain as an output, but you can also target database services or just use a HTTP request. You would be using HTTP requests to submit data to a non-AWS ElasticSearch domain.

. . .

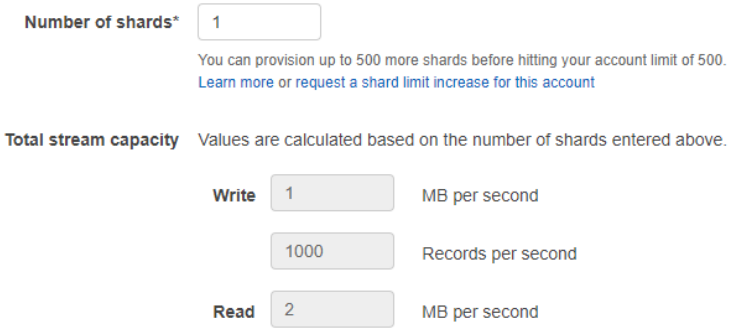
Getting Started...

Creating a Data Stream is easy, all you have to do is provide a name for the stream and a number of shards to use behind the stream. You can use the handy estimator to figure out how many you will need.

To do this simply, take the average size of the source data you'll be posting and divide this by the number of records to get your average record size. Once you have this you can feed it into the estimator to get a guide for the number of shards you'll need. I'd suggest slightly over-estimating to avoid any potential hiccups:



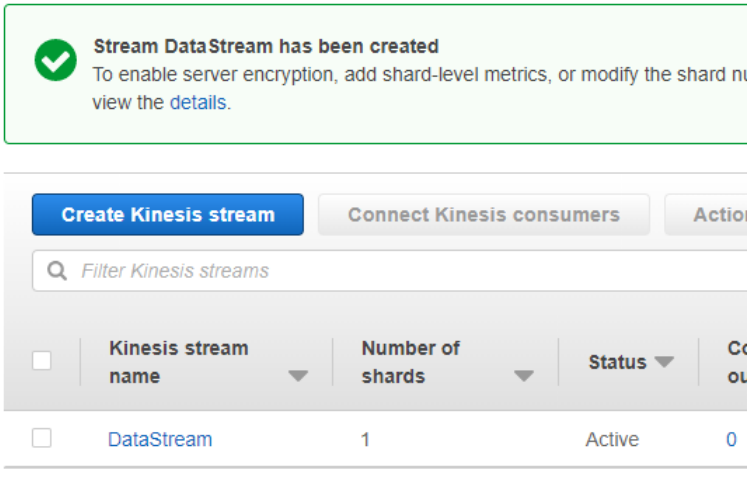
Feed the estimated shards value into the Number of shards, you'll see a single shard can provide 1MB of writes/second, and 2MB of reads/second. This scales linearly as you increase shards.



Using this approach, we can clearly see that Kinesis Data Streams are much more efficient when working with smaller amounts of data at lower throughput. We will be looking to use Kinesis for high-throughput processing, so we must try to reduce our data payload as much as possible to keep the costs low. More shards = More Cost.

Full details on Kinesis pricing can be found here — https://aws.amazon.com/kinesis/data-streams/pricing/

After hitting create, it'll take a few moments to create the Data Stream, but once it's done you should see a status of 'Active':



We're now ready to start pushing data to the Data Stream!

. . .

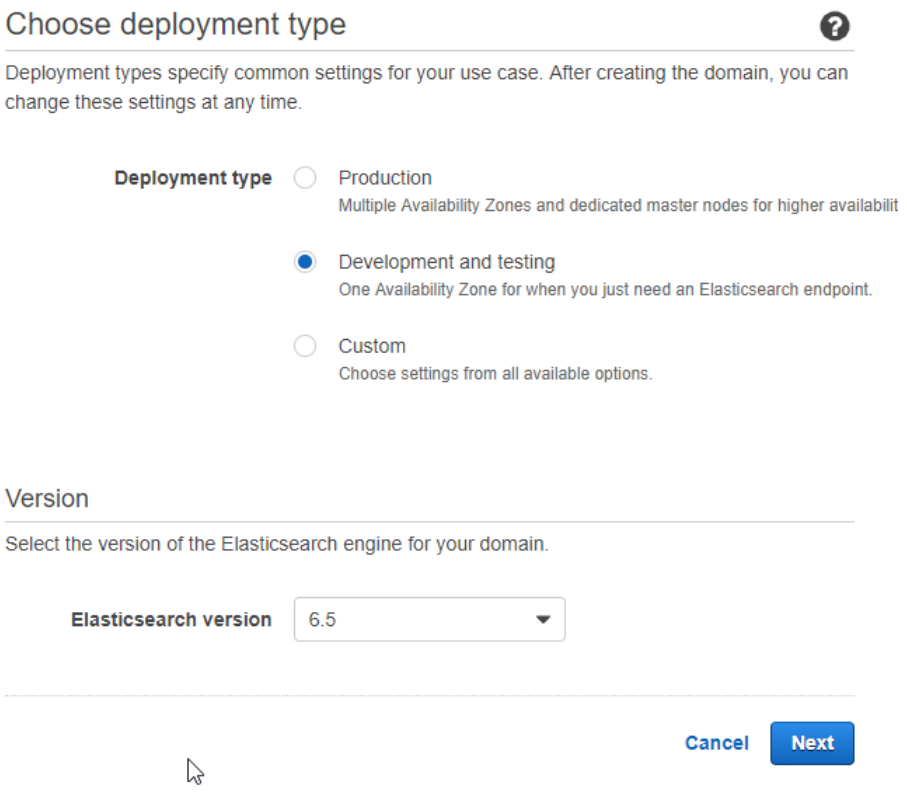
Creating an ElasticSearch Service Domain (optional)

For this guide, we will be using an ElasticSearch Service domain provided by AWS. You can of course use an existing ElasticSearch instance you have, but for simplicity we will be sticking to the services provided by AWS. I'll guide through setting up a simple ElasticSearch domain, as we will be using this later with a Kinesis Firehose.

Go to the ElasticSearch Service, and hit 'Create a new domain'…

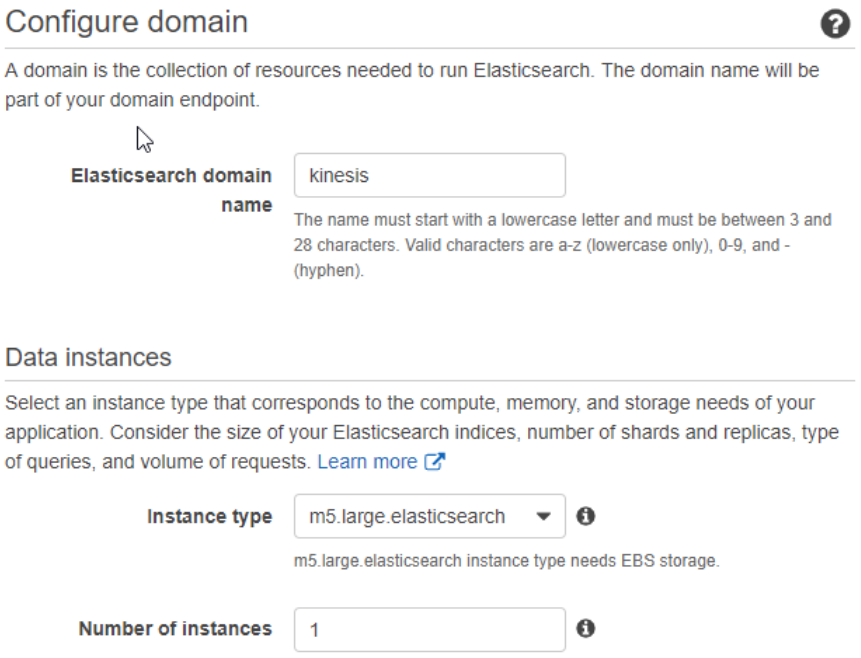
. . .

Here I would just select 'Development and testing' and for the version, select the highest available and hit 'Next'



. . .

Here we provide a name for the domain, and the instance size we would like to use. For our testing purposes, we are going to use a single m5.large. For production purposes, you should always use more than a single node in case there is an issue. You can end up losing data!



. . .

Next up, set the storage you'd like to provide to ElasticSearch. We will only be loading small amounts of data for testing so you can keep this at 10Gb.

. . .

Next up, set the encryption settings for your domain. I'd enable encrypt data at rest just to be on the safe side, I also just use the default key. We only have a single node, so no point enabling node encryption. Hit 'Next' after this.

. . .

Next, we have to provide the network settings for our ElasticSearch domain. For testing purposes, we will be using a Public instance restricted to a specific IP address. YOU SHOULD NOT BE DOING THIS IN PRODUCTION, VPC'S SHOULD BE USED INSTEAD!!!

. . .

Next we have to configure the form of authentication, I don't like using

Cognito so I won't be using it here.

. . .

Next we will restrict our domain to just our IP address. To do this, access this link to verify your public IP address — http://checkip.amazonaws.com/

Note it down, and then select from the 'Select a template' dropdown 'Allow access to the domain from specific IP(s)', enter in your IP from the link above and hit OK. You'll see the access policy has been populated with a definition to allow only your IP. This can be modified to a suit a range using the traditional slash notation:

i.e. 10.0.0.0/24 or 10.0.0.0/16 etc.

Mine looks like this (replace XXX.. with your account number). Notice I have also permissioned anything from my account to access the domain, we will need this later for Lambda to access the domain also:

```
{
  "Version": "2012-10-17",
  "Statement": [
    {
      "Effect": "Allow",
      "Principal": {
        "AWS": "arn:aws:iam::XXXXXXXXXXX:root"
      },
      "Action": "es:*",
      "Resource": "arn:aws:es:eu-west-1:XXXXXXXXXXX:domain/kinesis/*"
    },
    {
      "Effect": "Allow",
      "Principal": {
        "AWS": "*"
      },
      "Action": "es:*",
      "Resource": "arn:aws:es:eu-west-1:XXXXXXXXXXX:domain/kinesis/*",
      "Condition": {
        "IpAddress": {
          "aws:SourceIp": "1.1.1.1"
        }
```

```
        }
      }
    ]
}
```

es-domain-access-policy.json hosted with ❤ by GitHub view raw

Once that's all done with, hit Next.

. . .

Finally, you'll get a chance to go through all of your chosen settings for the domain. Once you're happy with everything (don't forget things can be changed later) just hit Confirm.

Whilst the domain is being created we'll make a start on the code.

Building the Data Pipeline in Python

First of all, we need to source some data to load. In this guide, I'll be using a sample CSV of Sacramento Crime Records. It's got a few interesting fields you might want to play around with in Kibana, namely location data. You can grab a copy of this data here.

Understanding the different types of data you need to push and how ElasticSearch understands them is very important. If you don't perform the proper data typing, ElasticSearch will only ever understand it as a string. This will become more apparent later when we build the transformation function.

We can see the CSV contains a datetime field we can use in Kibana as our @timefield. We will have to do some work on it first so ElasticSearch can understand it as a datetime.

First of all, I'll always look to build a lightweight script that can quickly collect the required data, perform any transformations required and batch the data up to send to Kinesis. This is where you will discover the largest

bottleneck in time to process data. If you are sourcing data from the results of a SQL query, it's good idea to consider the latency you introduce when connecting to certain data sources and the time it takes for that query to return, as that will affect the time it takes for Kinesis to actually begin processing the data. Here we are using a lightweight example designed for local use, in part 2 we'll look at building a different streaming function so our pipeline is completely serverless with Lambda.

```
# necessary imports
import boto3
import datetime as dt
import pandas as pd
import time


# function to create a client with aws for a specific service and region
def create_client(service, region):
    return boto3.client(service, region_name=region)

# function to load data from CSV
def load_data(filename):
    df = pd.read_csv(filename)
    return df

# function to correctly display numbers in 2 value format (i.e. 06 instead of 6)
def lengthen(value):
    if len(value) == 1:
        value = "0" + value
    return value

# function for generating new runtime to be used for timefield in ES
def get_date():

    today = str(dt.datetime.today()) # get today as a string
    year = today[:4]
    month = today[5:7]
    day = today[8:10]

    hour = today[11:13]
    minutes = today[14:16]
    seconds = today[17:19]

    # return a date string in the correct format for ES
    return "%s/%s/%s %s:%s:%s" % (year, month, day, hour, minutes, seconds)

# function to modify the date time to be correctly formatted
def modify_date(data):

    dates = data['cdatetime'] # get the datetime field

    new_dates = [] # create empty lists
    load_times = []

    load_time = get_date() # get current time

    # loop over all records
    for date in dates:

        date = date.replace('/','-') # replace the slash with dash
```

```
        date = date + ":00" # add seconds to the datetime

        split = date.split(" ") # split the datetime

        date = split[0] # get just date

        months = date.split('-')[0] # get months
        days = date.split('-')[1] # days
        years = "20" + date.split('-')[2] # years

        time = split[1] # get just time

        hours = time.split(':')[0] # get hours
        minutes = time.split(':')[1] # get minutes
        seconds = time.split(':')[2] # get seconds

        # build up a string in the right format
        new_datetime = years + "/" + lengthen(months) + "/" + lengthen(days) + " " + le

        # add it the list
        new_dates.append(new_datetime)
        load_times.append(load_time)

    # update the datetime with our transformed version
    data['cdatetime'] = new_dates
    data['loadtime'] = load_times

    # return the dataframe
    return data

# function for sending data to Kinesis at the absolute maximum throughput
def send_kinesis(kinesis_client, kinesis_stream_name, kinesis_shard_count, data):

    kinesisRecords = [] # empty list to store data

    (rows, columns) = data.shape # get rows and columns off provided data

    currentBytes = 0 # counter for bytes

    rowCount = 0 # as we start with the first

    totalRowCount = rows # using our rows variable we got earlier

    sendKinesis = False # flag to update when it's time to send data

    shardCount = 1 # shard counter

    # loop over each of the data rows received
    for _, row in data.iterrows():

        values = '|'.join(str(value) for value in row) # join the values together by a

        encodedValues = bytes(values, 'utf-8') # encode the string to bytes

        # create a dict object of the row
        kinesisRecord = {
            "Data": encodedValues, # data byte-encoded
            "PartitionKey": str(shardCount) # some key used to tell Kinesis which shard
        }


        kinesisRecords.append(kinesisRecord) # add the object to the list
        stringBytes = len(values.encode('utf-8')) # get the number of bytes from the st
        currentBytes = currentBytes + stringBytes # keep a running total
```

```
        currentBytes = currentBytes + stringBytes # keep a running total

        # check conditional whether ready to send
        if len(kinesisRecords) == 500: # if we have 500 records packed up, then proceed
            sendKinesis = True # set the flag

        if currentBytes > 50000: # if the byte size is over 50000, proceed
            sendKinesis = True # set the flag

        if rowCount == totalRowCount - 1: # if we've reached the last record in the res
            sendKinesis = True # set the flag

        # if the flag is set
        if sendKinesis == True:

            # put the records to kinesis
            response = kinesis_client.put_records(
                Records=kinesisRecords,
                StreamName = kinesis_stream_name
            )

            # resetting values ready for next loop
            kinesisRecords = [] # empty array
            sendKinesis = False # reset flag
            currentBytes = 0 # reset bytecount

            # increment shard count after each put
            shardCount = shardCount + 1

            # if it's hit the max, reset
            if shardCount > kinesis_shard_count:
                shardCount = 1

        # regardless, make sure to incrememnt the counter for rows.
        rowCount = rowCount + 1


    # log out how many records were pushed
    print('Total Records sent to Kinesis: {0}'.format(totalRowCount))

# main function
def main():

    # start timer
    start = time. time()

    # create a client with kinesis
    kinesis = create_client('kinesis','eu-west-1')

    # load in data from the csv
    data = load_data('crime.csv')

    # modify the date and add loadtime field
    data = modify_date(data)

    # send it to kinesis data stream
    stream_name = "DataStream"
    stream_shard_count = 1

    send_kinesis(kinesis, stream_name, stream_shard_count, data) # send it!

    # end timer
    end = time. time()

```

```
    # log time
    print("Runtime: " + str(end - start))

if __name__ == "__main__":

    # run main
    main()
```

stream-data-kinesis-local.py hosted with ❤ by GitHub view raw

This script does the following:

- First, we start a timer to capture the script runtime.
- We create a client with Kinesis in the Ireland region (eu-west-1), returns a client
- Loads data in from the CSV renamed to 'crimes.csv', returns a pandas DataFrame
- Modifies the DataFrame to correct the datetime field and appends a new datetime field of current time. Returns a DataFrame
- Each record's fields are joined together using a '|' (pipe) character which we leverage later. We batch up all the data and post it Kinesis to be processed using the client we created, providing the Stream name and shard count. Shards are rotated through to allow for proper fanning out of the workload, however we are only using 1 here.
- Finally, we log the the number of records posted and time taken.

Feel free to modify the variable in main() to suit you.

Running this script loads 7584 records in just under 6 seconds with only one shard.

Obviously, this is a bit of an extreme test and you should be looking to push data more regularly to keep the volume down, this way Kinesis doesn't get overwhelmed with massive amounts of data all coming in at once resulting in rejected records.

Sourcing the data is by far the most complex of all the steps in this process.

Setting up a Kinesis Firehose with ElasticSearch

In Firehose, hit ‘Create delivery stream’ and provide a name, I will call mine ‘firehose’ for simplicity.

. . .

Next, we set the source of our data to be the Kinesis stream we created earlier, then hit Next.

. . .

Next we are presented with options for processing records. An important point to note here is that for ElasticSearch to understand the data we have sent up, we must first unpack it into a format it will understand. ElasticSearch understands JSON data so that’s what we will need to build.

Currently our data is in the stream waiting to be processed, each record is a pipe delimited string, but we need it as a JSON object. We can use record transformations with a Lambda function to easily do this for us.

Select enabled for record transformation, and let's open a new tab in Lambda.

Building the Transformation Lambda

Here we are going to create a Lambda function that receives data in the form of event JSON containing the records to be transformed. The record's fields are unpacked and then built into a JSON object that ElasticSearch will understand. Here we are using NodeJS 10.x but you may use any language you see fit.

The key to this is understanding the underlying data being passed through the DataStream. It's at this point you will need to perform any data typing ready for the JSON object as you cannot change this once it has been ingested by ES. The keys of your JSON document will also become the fields in ES, so consider the names of them carefully. Another consideration to take into account, is any further processing done here could slow your pipeline down, so I'd recommend doing this before Kinesis receives the data.

You want to get this right as it will affect the uses of the field in Kibana dashboards also.

As everything is a string, we will need to parse certain fields as numbers in order to use them correctly later. Go to Lambda and hit Create Function. Select author from scratch, provide a name (I called mine 'transform-firehose-data') and select Node.js 10.x. Let Lambda create a new role as we don't really need any permissions other than logging. Hit create function.

Once loaded up, we are going to use the following code:

```
'use strict';
console.log('Loading function');


exports.handler = (event, context, callback) => {
    let success = 0; // Number of valid entries found
    let failure = 0; // Number of invalid entries found

    /* Process the list of records and transform them */
    const output = event.records.map((record) => {
```

```
        // Kinesis data is base64 encoded so decode here
        console.log(record.recordId);
        const payload = (Buffer.from(record.data, 'base64')).toString('ascii');
        console.log('Decoded payload:', payload);

        // Split the data into it's fields so we can refer to them by index
        const match = payload.split('|');

        if (match) {
            /* Prepare JSON version from Syslog log data */
            const result = {

                // build all fields from array
                crime_time: match[0],
                address: match[1],
                district: parseInt(match[2]),
                beat: match[3],
                grid: parseInt(match[4]),
                description: match[5],
                crime_id: parseInt(match[6]),
                latitude: parseFloat(match[7]),
                longitude: parseFloat(match[8]),
                load_time: match[9],
                location:{
                     lat: parseFloat(match[7]),
                     lon: parseFloat(match[8])
                }
            };
            success++;
            return {
                recordId: record.recordId,
                result: 'Ok',
                data: (Buffer.from(JSON.stringify(result))).toString('base64'),
            };
        } else {
            /* Failed event, notify the error and leave the record intact */
            failure++;
            return {
                recordId: record.recordId,
                result: 'ProcessingFailed',
                data: record.data,
            };
        }
    });
    console.log(`Processing completed.  Successful records ${success}, Failed records ${
    callback(null, { records: output });
};
```

transform-data-kinesis.js hosted with ❤ by **GitHub** **view raw**

As you can see we've built the documents fields using the JSON keys, this will be more apparent later on. We perform proper parsing of values into their correct data types. It's important to note that any failures to convert a value will result in the record not arriving in ES. This is more likely due to a field not being completely consistent, you may find some fields that contain a string of 'null' / 'N/A' / 'NaN' to represent a null value when other values are floats or integers.

When this kind of scenario arises, it's good to go back to your original data sourcing/posting script and make the necessary alterations in code so that these issues are corrected. Sometimes this means literally looking through all the rows to fully understand the content of the fields to identify the problematic values. For cases where you do want to have nulls represented correctly in ES, you will have to make changes to the transformation script to check if the field value is equal to your 'null' / 'N/A' / 'NaN' string and then exchange it for an actual null.

Make sure to also increase the timeout of the functiom to something long like 5 minutes.

Finish creating the Firehose

Hop back to the Firehose creation and select the Lambda function we just created and select the latest version.

We don't need to use record format conversion so we can just leave that disabled. Hit Next when ready.

. . .

Next, we have to select the destination place for our data. We will be using ElasticSearch but you can use anything you wish. You can see the flow this data will take with this setup below:

S3 will take care of anything that fails, great!

. . .

Next we are going to configure the ElasticSearch domain and index settings we want to use.

An index can be considered a bit like a schema or a specific table structure. You don't really want to mix different structured data within the same index, generally this leads to a bit of headache when it comes to visualisation as it's not always clear which field are related. I'd suggest as a rule of thumb you use an index per feed/firehose to keep it simple.

A type is yet another way of further classifying data but in the latest version of ElasticSearch posting multiple types to the same indexes is not permitted, so yet again I would go with the same rule as above here.

. . .

Next we configure the S3 backup settings and location for any failed records ElasticSearch rejects. This is a good way to monitor that your pipeline is healthy and that records are not failing on their way in. Here we've configured them to go to a specific folder in the bucket.

Hit next when you are ready.

. . .

Next, we configure the buffer conditions. This controls how often data is pushed out to the destination. We want this to be as low as possible, set this to 1MB and 60 seconds to decrease the amount of time it takes for data to be output.

. . .

Leave everything else default and scroll down to the IAM role section. Here we are going to click ‘Create new or choose’, then allow Firehose to create a new IAM role for this Firehose, by just hitting Allow. Your role should have been created and then you can just click Next.

. . .

We get a chance to review all the settings for the Firehose before we create them. When you are ready just hit ‘Create delivery stream’.

Once created, you should see a status of Active shortly after, once you see this your Firehose should be pushing data to ElasticSearch when the DataStream receives data.

Checking ElasticSearch for Data with Kibana

We can check if ElasticSearch is receiving our data from the Kibana Dashboard.

To access this:

- Go to My Domains, and select your domain.
- Click the Kibana link shown on the Overview tab.

This will launch you into the Kibana UI. If this is your first time accessing Kibana, you'll see something like this:

If you see this, just click 'Explore on my own'.

- Then go to Management > Index Patterns
- Under Define index pattern, enter 'crime-data*'. It should match only one index, and then you can hit next.

On the next screen, we have to select which timefield we want to view the data on.

- If you want to see crimes over time select ‘crime_time’
- If you want every load of data over time select ‘load_time’

I am going for crime_time as I want to see the value used in the data.

Await it’s creation and then you should see all the fields we captured from the CSV.

If you want, you can also set this to be the default index for easy access from other screens. If you want to do this hit the star in the top right:

. . .

Going to the Discover tab and setting our time in the top right to view all the way back to 2006 we can see our data has arrived (one batch for each run) in the dashboard. If you used load_time as the time field, you can search for last 1 hour.

You can now leverage all the Kibana tools available to build visualisations and then dashboards etc.

You can learn more about that below:

- Visualisations (start here) — link
- Dashboards — link

That's all for Part 1

You should now have data flowing into Kibana whenever you run the collection script. It is just simulation data to show how each of the services interact and how data can end up in the dashboard.

In the next installment, we will be looking at collecting live data from a web service using Lambda as a serverless approach to data streaming and visualisation.

Follow me to know when it's ready! 😃