Streaming Data

In this lecture, we will cover AWS Kinesis, a **serverless** streaming service provided by AWS that is hugely useful in managing “always on” data for near real time decision making and analysis.

[**AWS Kinesis**](#_2k4bvjvbqj38) **2**

[Table 1. Batch Processing vs. Streaming](#_4vlrymu8ylqw) 2

[**AWS Kinesis Data Firehose**](#_qai1tbdtuuzv) **3**

[**Setting up our AWS Kinesis Data Firehose**](#_q5m1rl2g7u29) **4**

[**Testing our AWS Kinesis Data Firehose**](#_uowaunmzqck7) **11**

# 

# 

# 

# 

# 

# 

# 

# 

# 

# 

# 

# AWS Kinesis

(A lot of this is from AWS Kinesis homepage)

Amazon Kinesis offers managed services for streaming data called Kinesis Data Streams, Kinesis Video Streams, Kinesis Data Firehose, and Kinesis Data Analytics. Kinesis itself is based on Apache Kafka, a distributed data store for building real-time streaming data pipelines and applications. Amazon paints a thorough description of what constitutes “data streaming”. To paraphrase, when you stream data, thousands of data sources simultaneously and continuously generate records, which you must process [“sequentially and incrementally on a record-by-record basis or over sliding time windows”](https://aws.amazon.com/streaming-data/).

## Table 1. Batch Processing vs. Streaming

|  |  |  |
| --- | --- | --- |
| **Factors** | **Batch Processing** | **Streaming** |
| Data scope | Queries or processing over all or most of the data in the dataset. | Queries or processing over data within a rolling time window, or on just the most recent data record. |
| Data size | Large batches of data. | Individual records or micro batches consisting of a few records. |
| Performance | Latencies in minutes to hours. | Requires latency in the order of seconds or milliseconds. |
| Analyses | Complex analytics. | Simple response functions, aggregates, and rolling metrics. |

Source: <https://aws.amazon.com/streaming-data/>

The benefit of using Amazon Kinesis’s managed services is that you can avoid handling the complex infrastructure provisioning needed for streaming data. As a whole, Amazon Kinesis allows you to capture and instantly process diverse types of data, including “video, audio, application logs, website clickstreams, and IoT telemetry data.” You can use various stream processing frameworks to build applications with custom analysis logic. Most commonly, Kinesis is used for “sharing data between different applications, streaming extract-transform-load, and real-time analytics”. Other use cases include:

* Power real-time dashboards (both internal and consumer-facing)
* Generate alerts
* Generate programmatic reminders
* Implement dynamic pricing and advertising
* Fraud detection
* Live leader-boards
* Clickstream analytics
* Network security monitoring
* Bug monitoring

In a nutshell, if you need alerting on a thing, you might need to use a streaming data service like Kinesis. In companies like Oracle or PlaceExchange, streaming data management was instrumental in ensuring that programmatic ads bought and sold in our ad exchange marketplace were being correctly bid on by the demand side and also correctly downloaded and rendered by the supply side.

These are the types of problems where a multi hour delay in insight could lead to significant losses in revenue. Another good example of this is system health checks. If you are Facebook and for some reason your servers go down, you want to know within seconds or minutes, not hours.

# AWS Kinesis Data Firehose

Amazon Kinesis Data Firehouse is a good first step in learning how to leverage Kinesis services. It is the “easiest way to reliably load streaming data into data lakes, data stores and analytics tools”. Firehose works in NEAR real-time, actually loading the data to the desired location 60 seconds after the data is sent to it. That said, it boasts the feature of serverless data transformation by enabling you to convert the raw streaming data into the format required by its destination.

By “data lakes” or “data stores” for our purposes, we can assume it to simply be S3 files within a bucket.

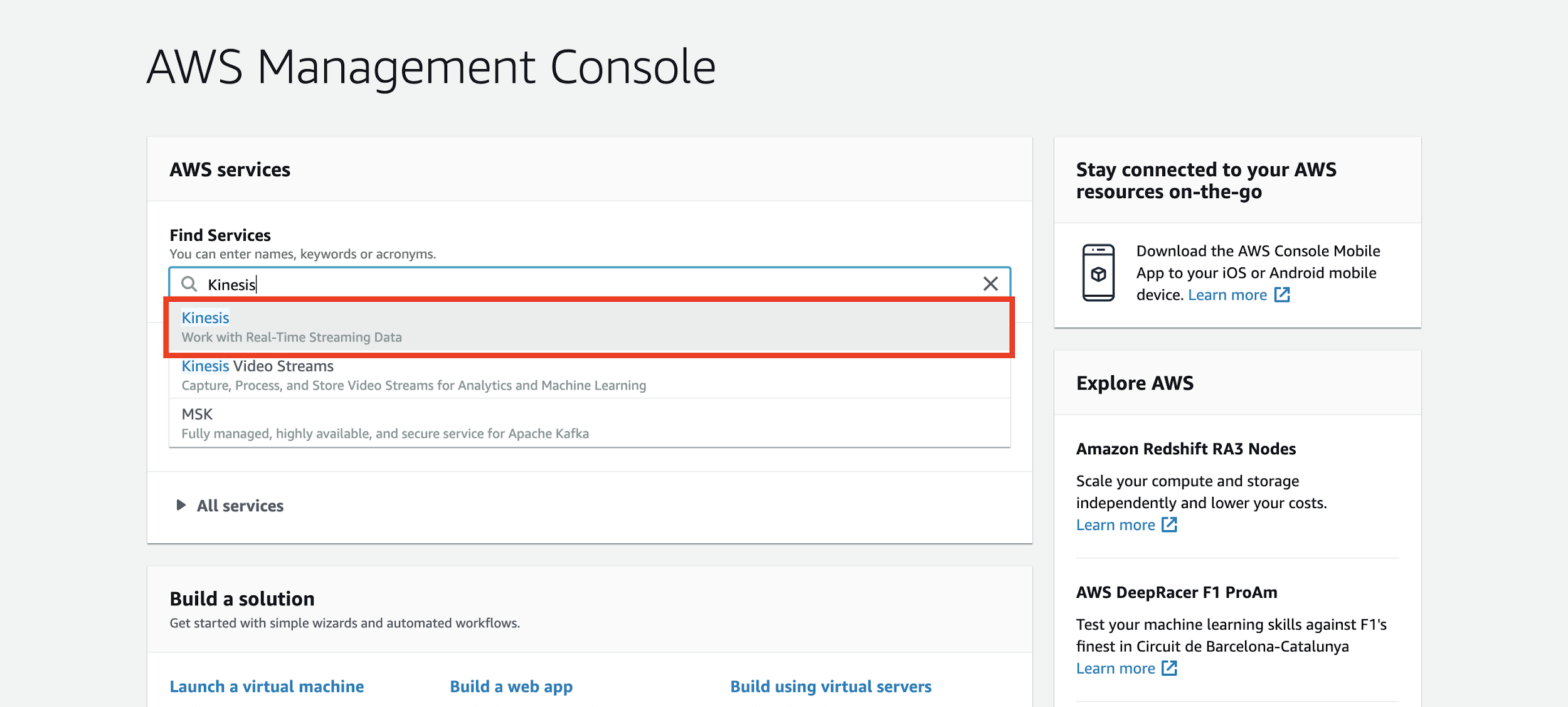
Using Kinesis Firehose Delivery Streams, we have the ability to drop data into S3 buckets which can then be analyzed and read with other services such as AWS EMR (as a spark job) or AWS Athena (as interactive, serverless querying).

Additionally, a lambda function can be leveraged to “transform” the data before it is delivered to S3 which gives us the change to “normalize” information if it is coming from various data sources to ensure that what ends up in our “data lake” is reliable and the same.

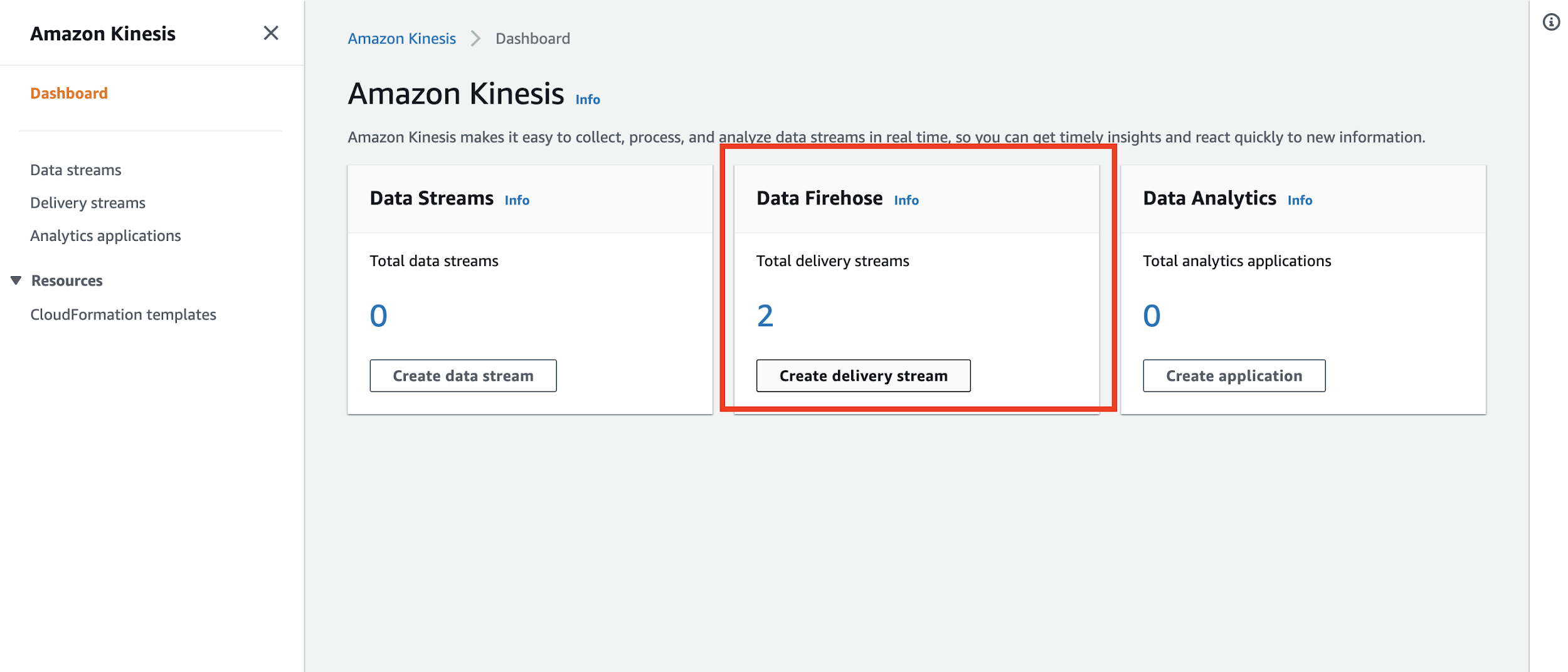
# Setting up our AWS Kinesis Data Firehose

For now, we will set up a “dummy” stream just to get a sense of how they are created. For the upcoming project, we will apply these learnings to a more “real” use case to understand how these technologies can be used to open ourselves up to interesting opportunities for analysis of “real time” data.

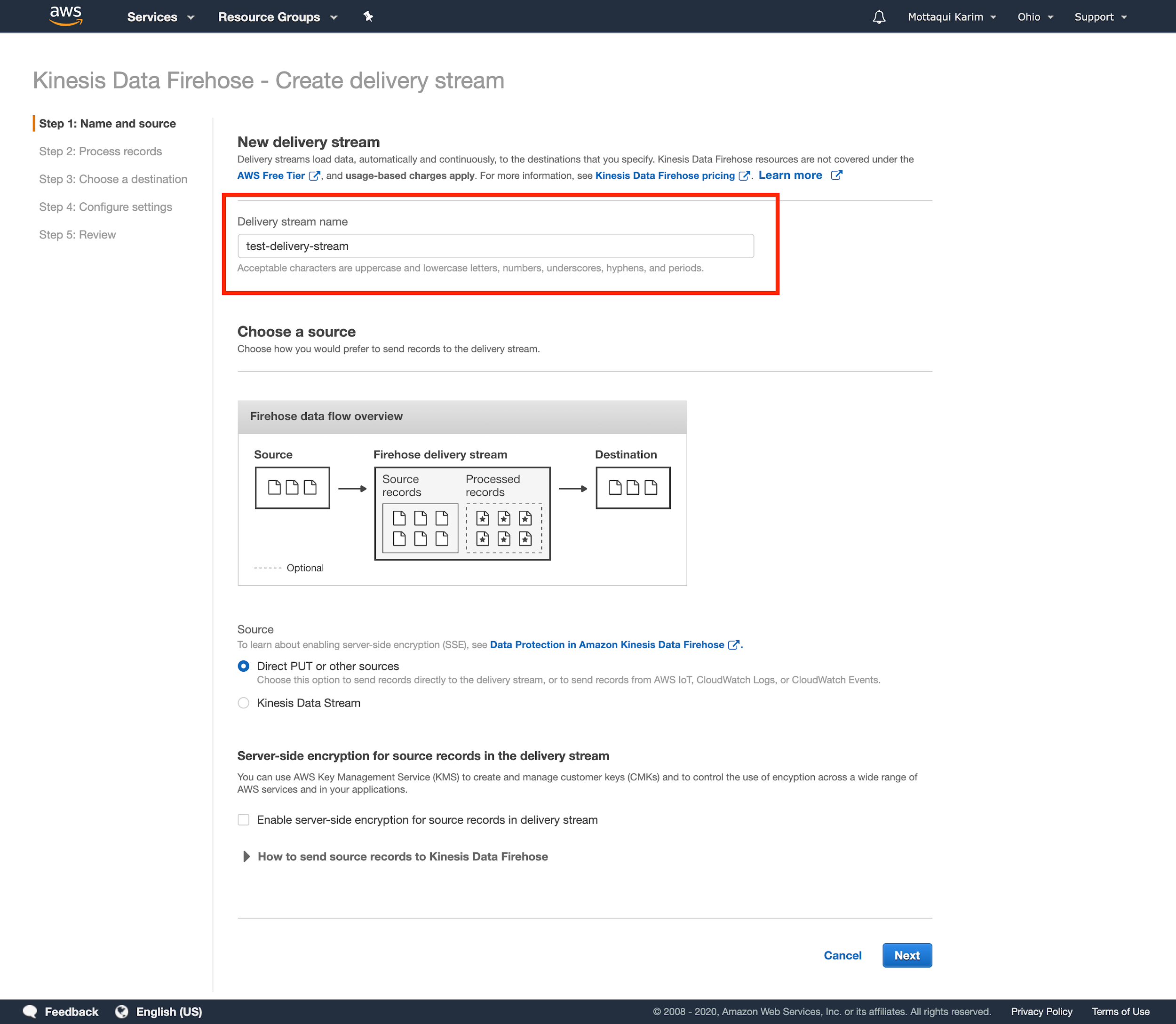
Start by going to the AWS homepage as per usual and type in “Kinesis”.



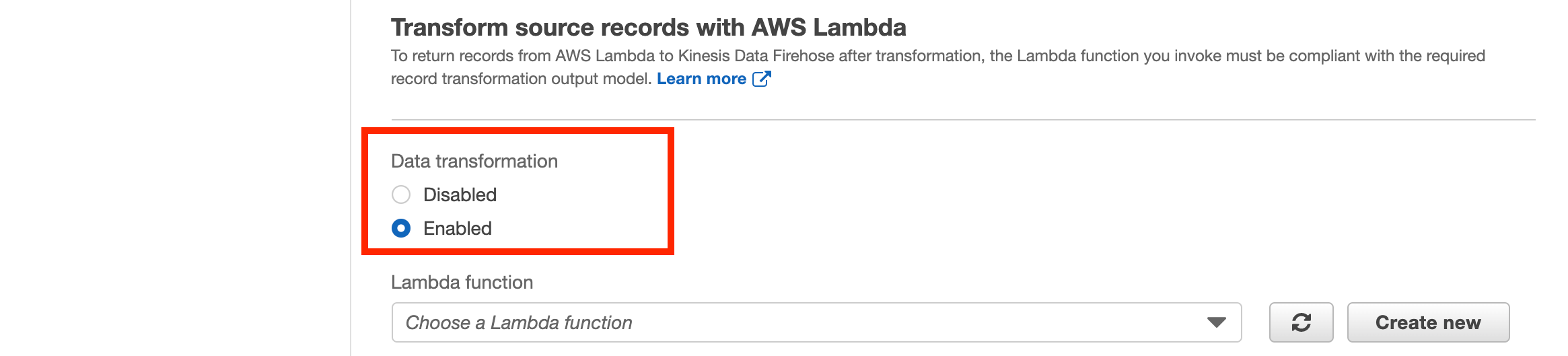
Then, choose the **Data Firehose** option.



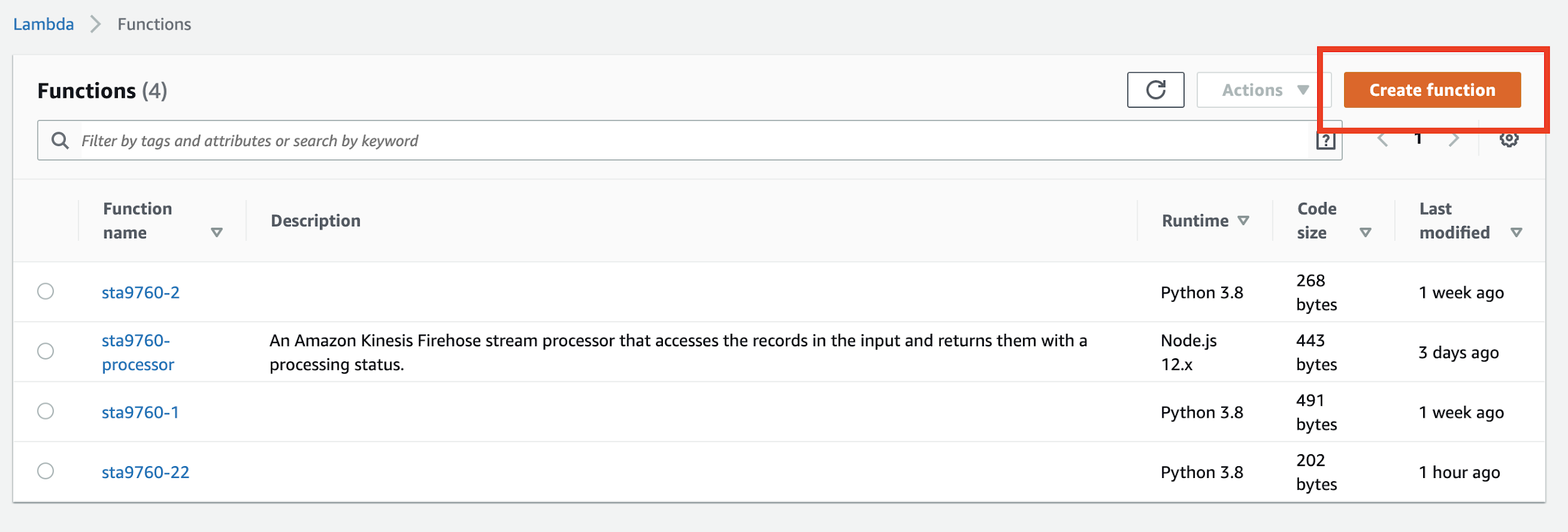
For the next page, you only really need to add a name for your delivery stream.



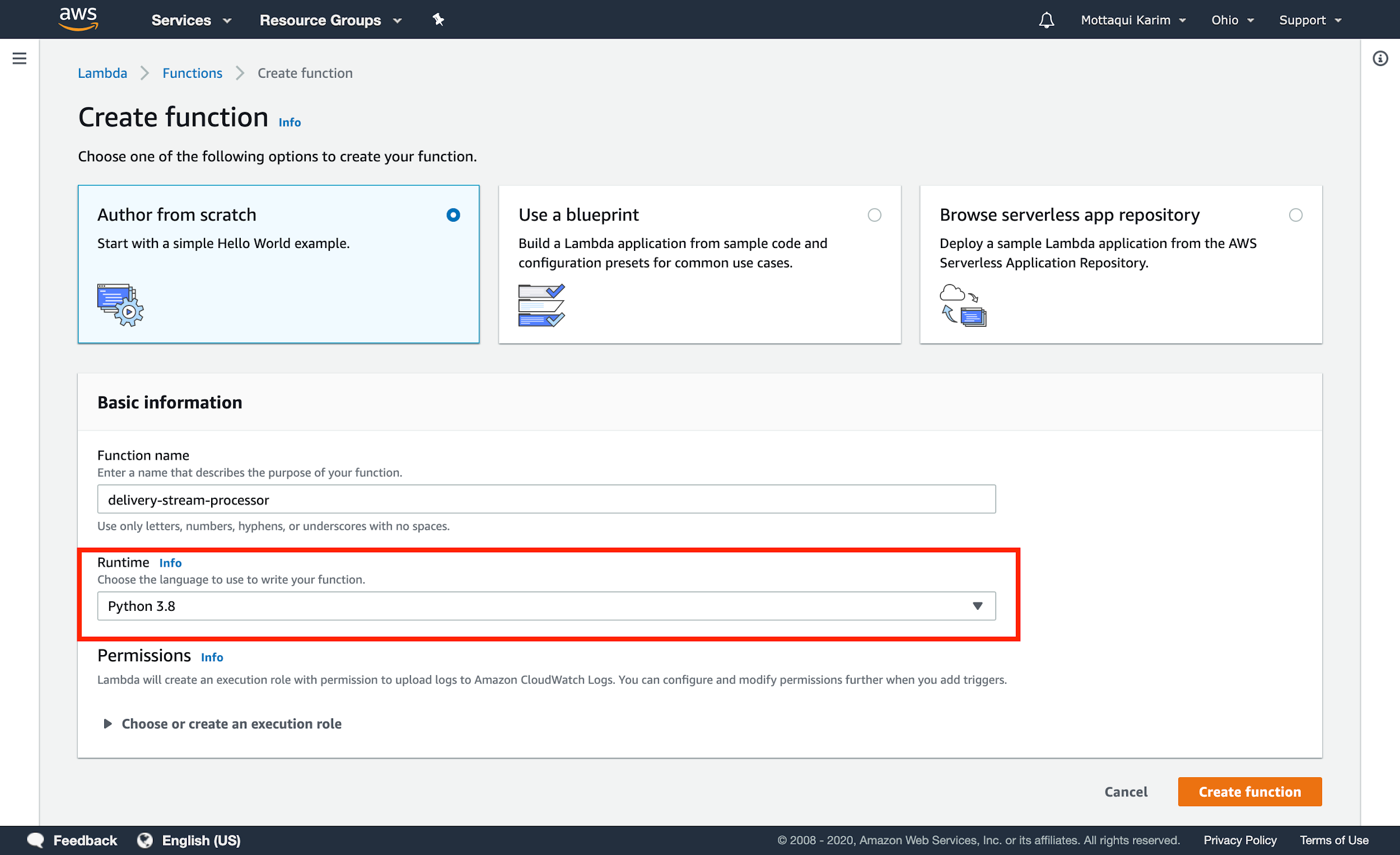
For the next page, you want to choose “Enabled” for the **Transform source records with AWS Lambda** option. Technically we don’t have to go down this step however it is useful to have a lambda listening to and processing these records if we choose to transform them in the future before dropping into the destination S3 buckets. Although you have the option to create a new lambda function from this page, it doesn’t actually give you too much flexibility for creating your own custom function; you can only choose a few blueprints that they have available (none of which seem to have the option that we want + python).



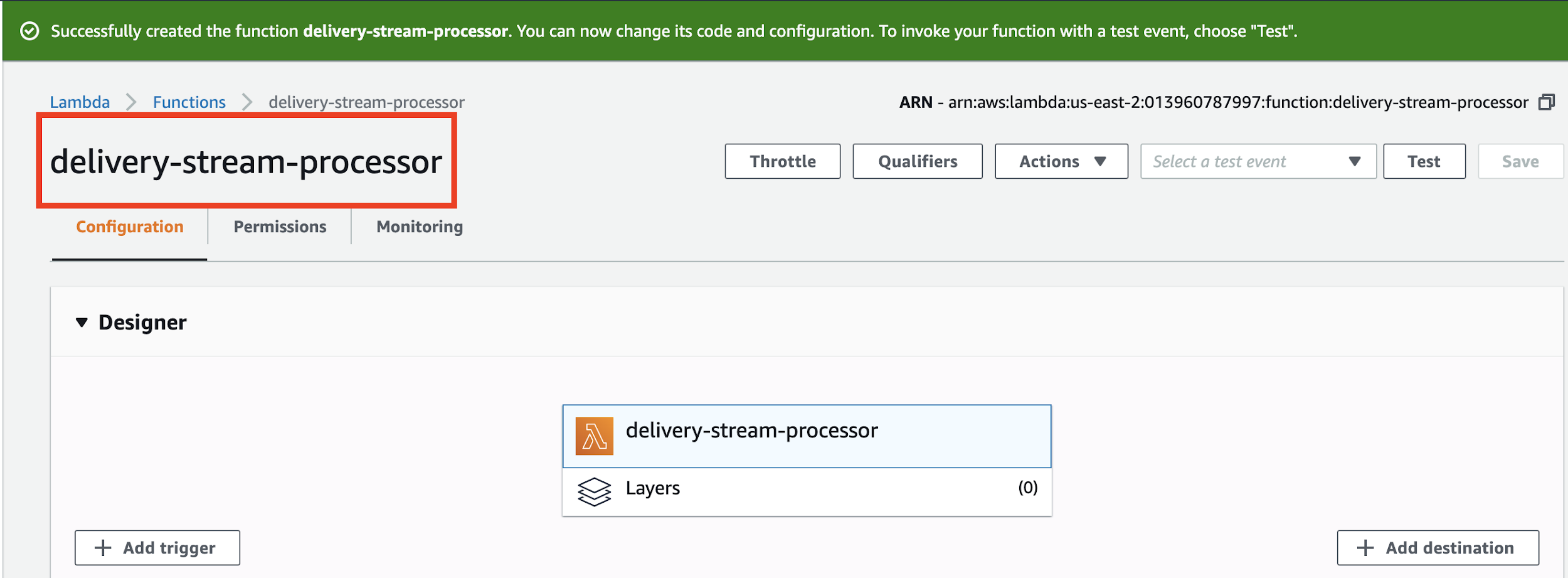
In a new tab, navigate to the **Lambda** landing page and create a new lambda function.



Note that you ought to choose Python 3.8 here.



Click **Create Function** and ensure you are on your new function landing page.



If you take a look at the initial code provided to your function, it looks a little something like this:

|  |
| --- |
| **import** json  **def** **lambda\_handler**(event, context):  # TODO implement  **return** {  'statusCode': 200,  'body': json.dumps('Hello from Lambda!')  } |

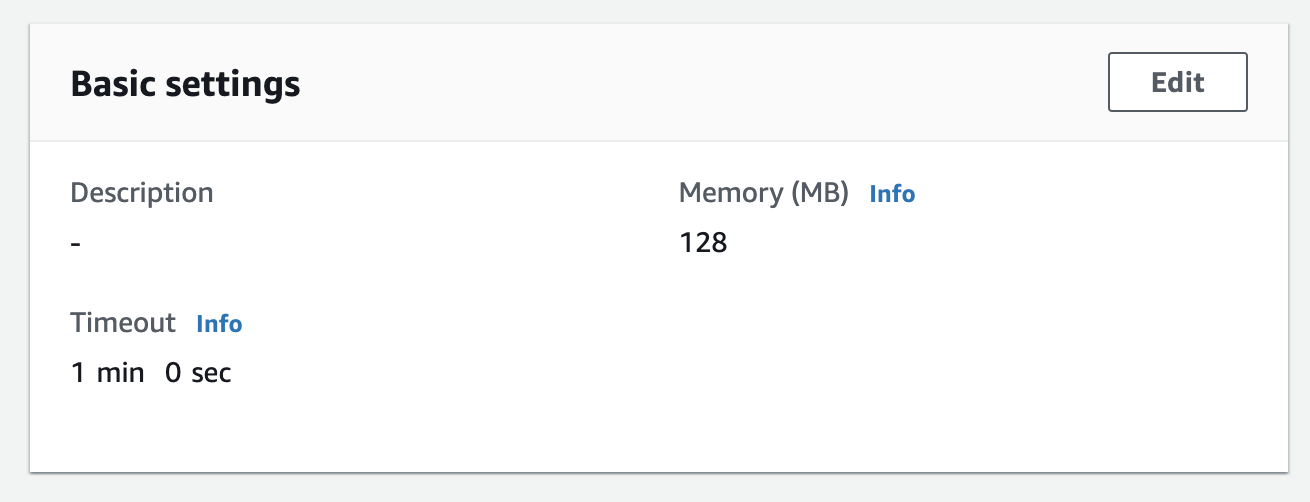
This is assuming that the lambda function will be triggered off of an API call (similar to what we did in the last lecture. However, as we know, we are going to trigger this lambda off of a new record added to our Kinesis Firehose Delivery Stream. As a result, we can actually change our code a bit:

|  |
| --- |
| **def lambda\_handler(event, context):  output\_records = []  for record in event["records"]:  output\_records.append({  "recordId": record['recordId'],  "result": "Ok",  "data": record["data"]  })    print(len(output\_records))  return { "records": output\_records }** |

This implementation is called the “identity” transformation - the simplest transformation we can do is no transformation (basically we just map over the exact fields needed: recordId, result and data and leave the actual data alone). In the future we could choose to take the **record[“data”]** and transform it as needed. For now, let’s leave this alone and return to our delivery stream work.

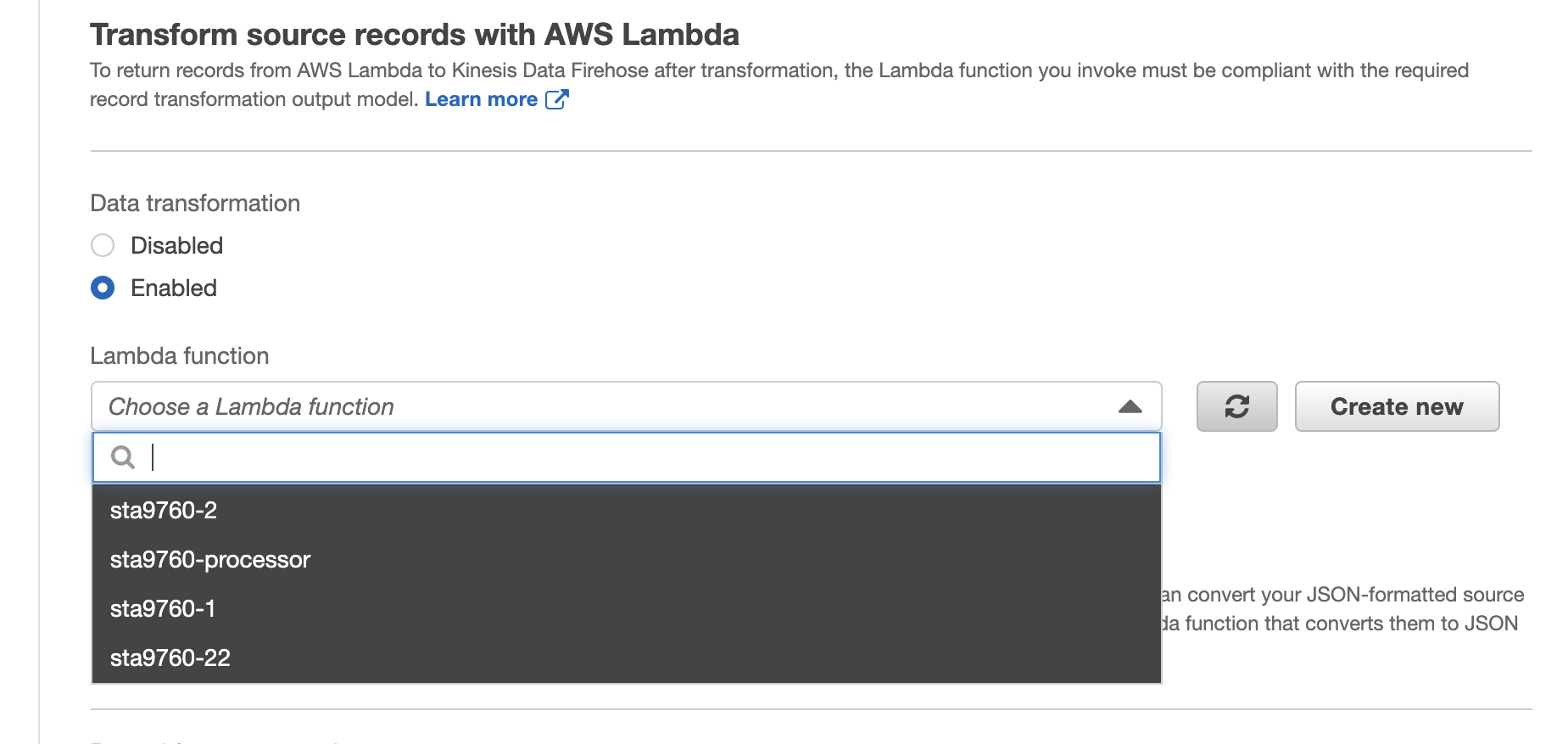
Before we go forward, we must update our **Timeout** for this function, since Kinesis requires at least a timeout of one minute (AWS minimum requirement for lambdas that transform records).

Remember this guy?



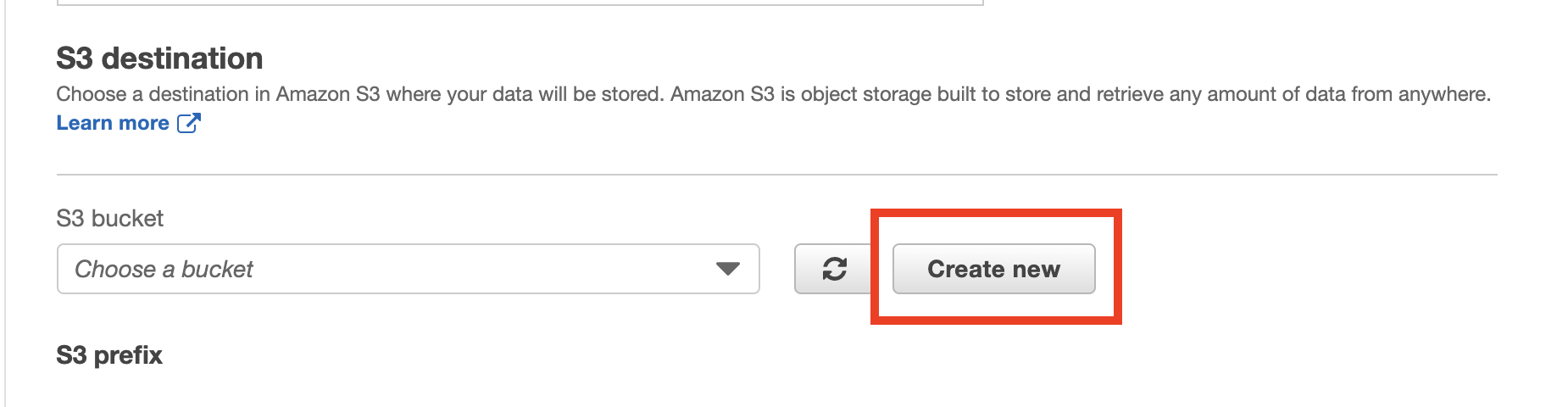
Click on the **Edit** button and increase the timeout to at least 1 min (by default it is super low, like 3s). After this change is made, don’t forget to hit the **Save** button on the top right before heading back to the delivery stream configuration page.

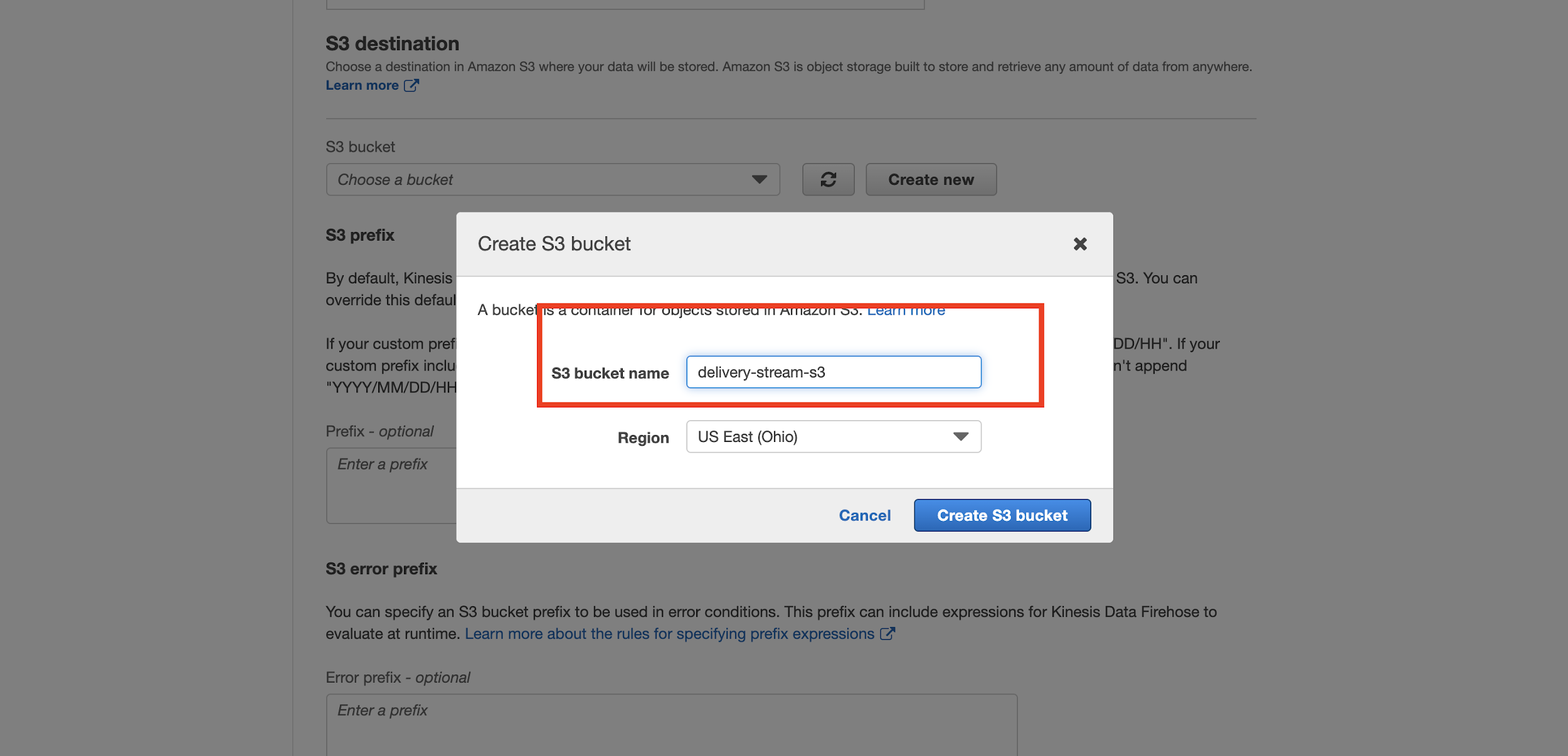
Having created this function, let us go back to where we left off with the delivery stream:



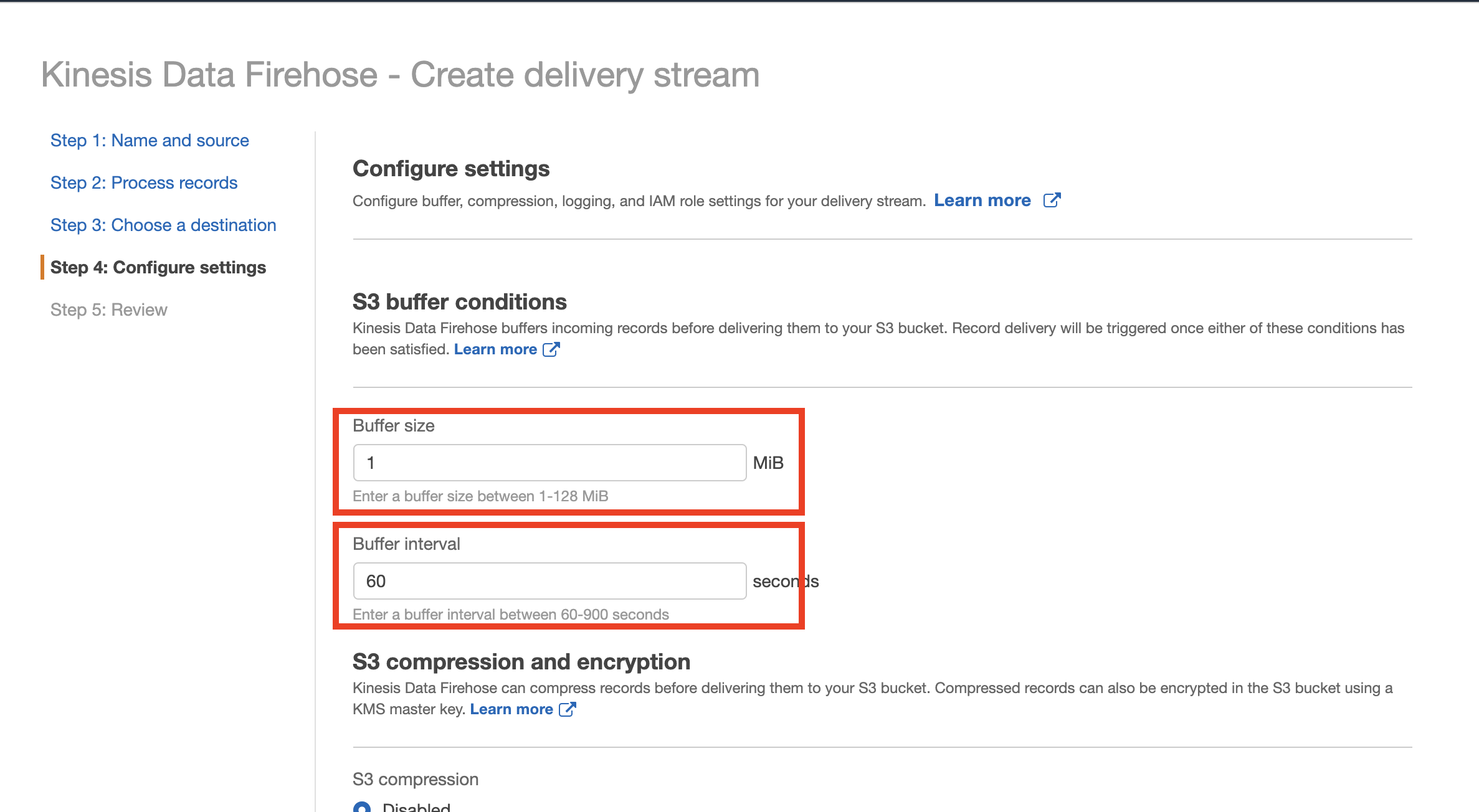
If you don’t see your lambda here, click on the **Refresh** button next to this dropdown and your function should appear. Select it and we are ready to move on to the next page!

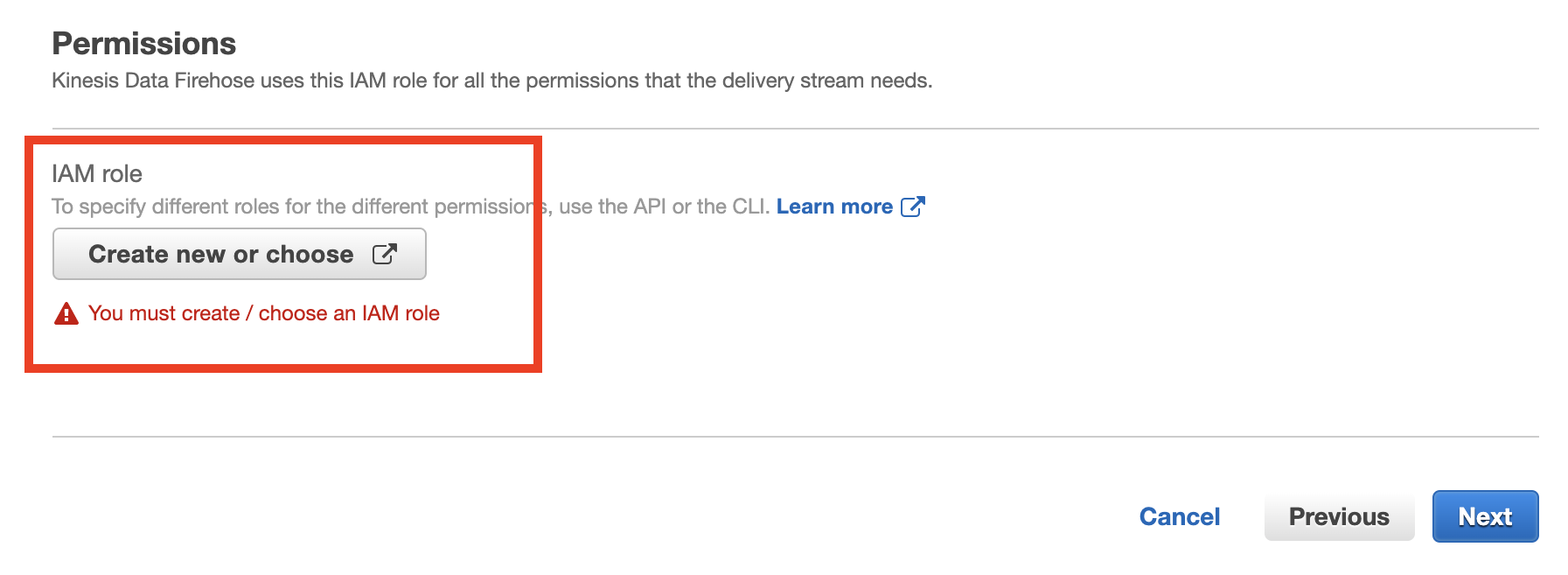
On the next page, we have to create a new s3 bucket to hold this data.



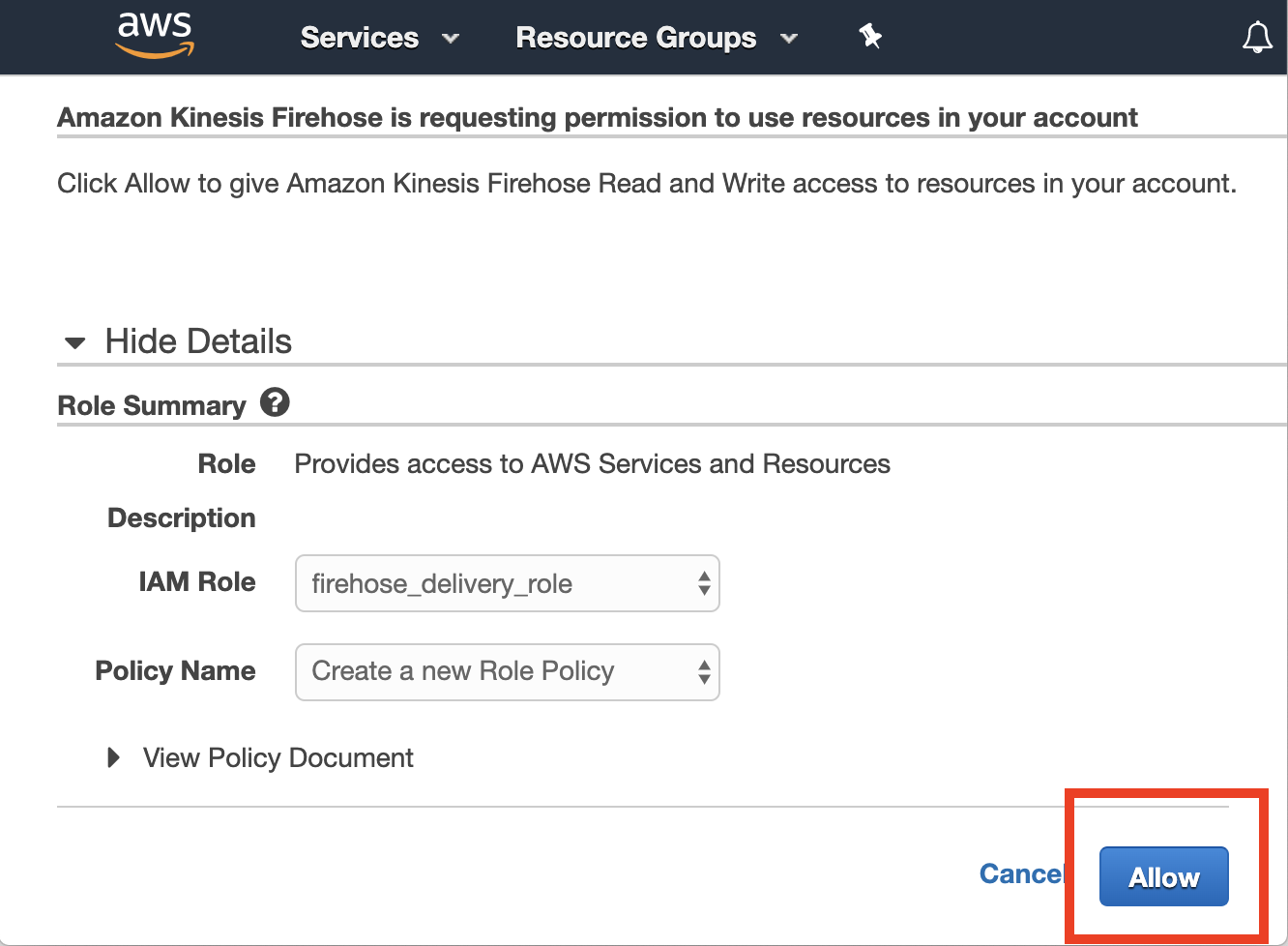


On the next page, update the **Buffer size** and **Buffer interval** to the lowest possible values. Since we are working on a “test” stream, there is no need to keep the defaults since we want to be able to test quickly (a longer buffer interval means you have to wait longer to see your results show up in S3).

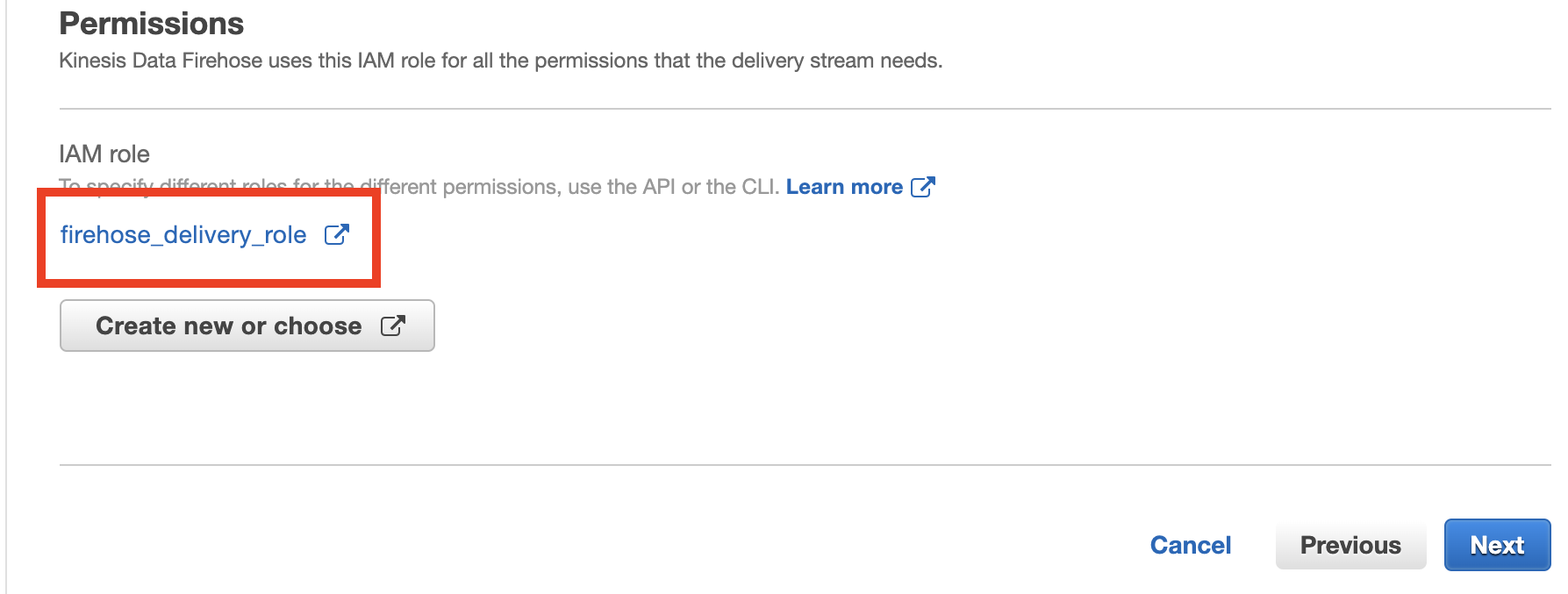


And then finally, before hitting **Next**, click on the AIM role button and pick the default option:  


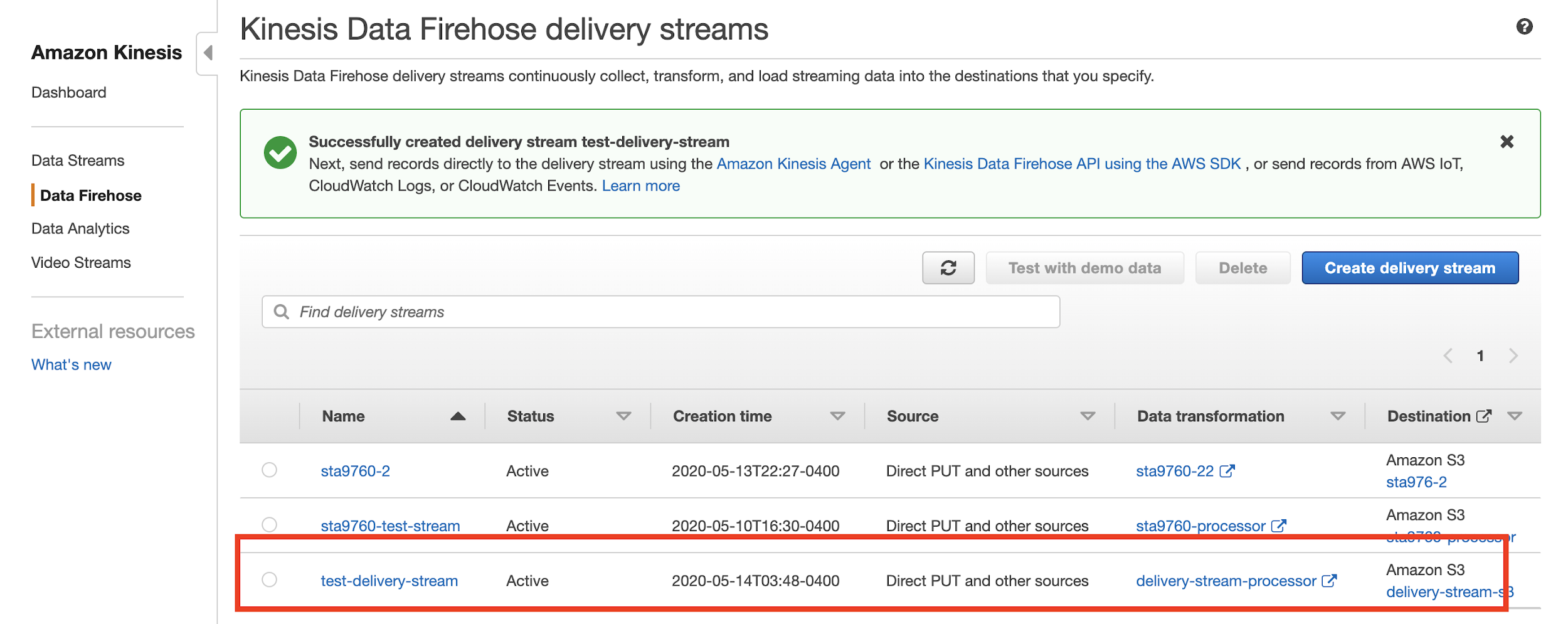
You don’t have to change anything, just click **Allow** and you should be redirected back.



Back to the original page, you should see something like:

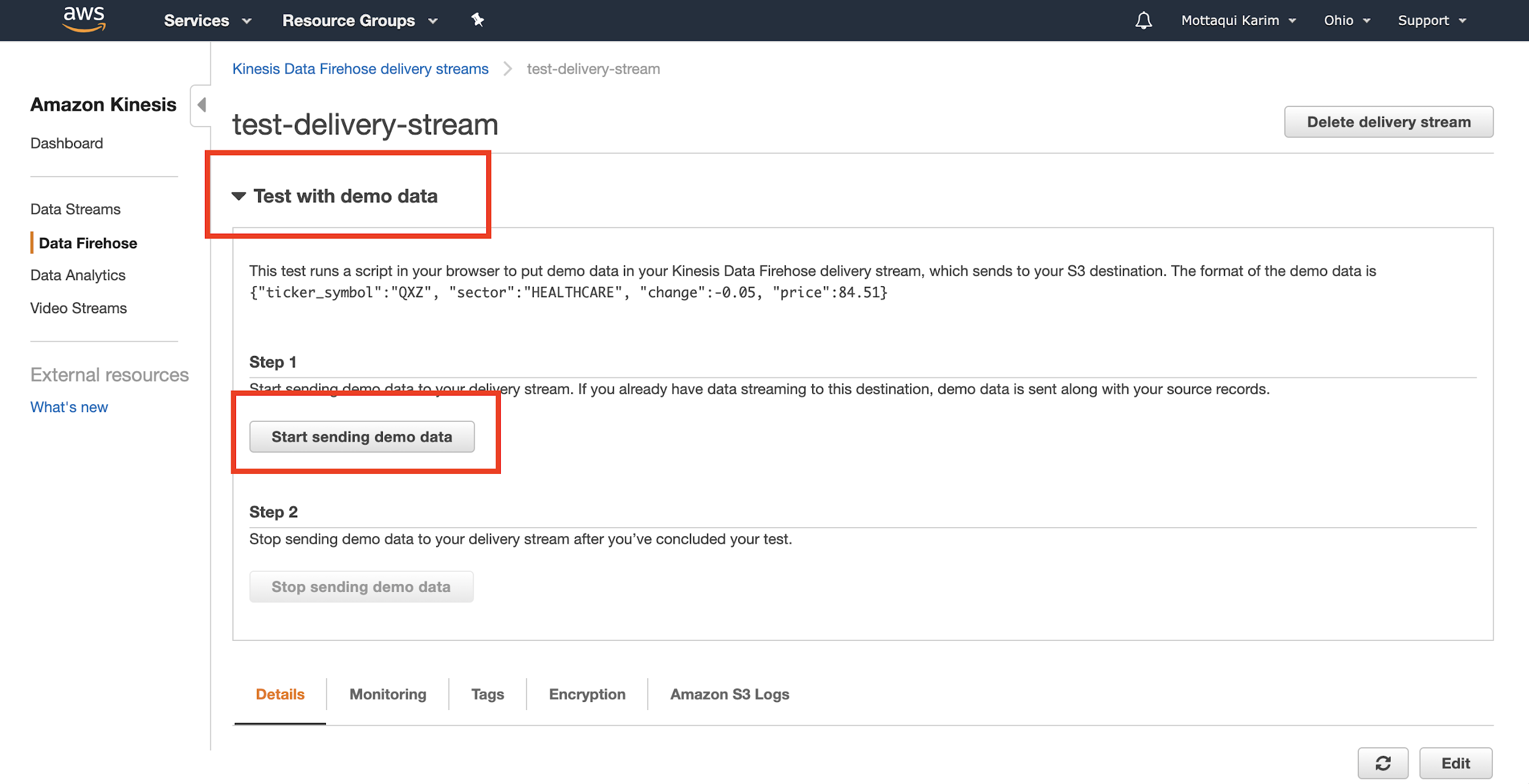


At this point you are good to go! Hit **Next**. On the review page, ensure your options are entered as intended and hit **Create**.

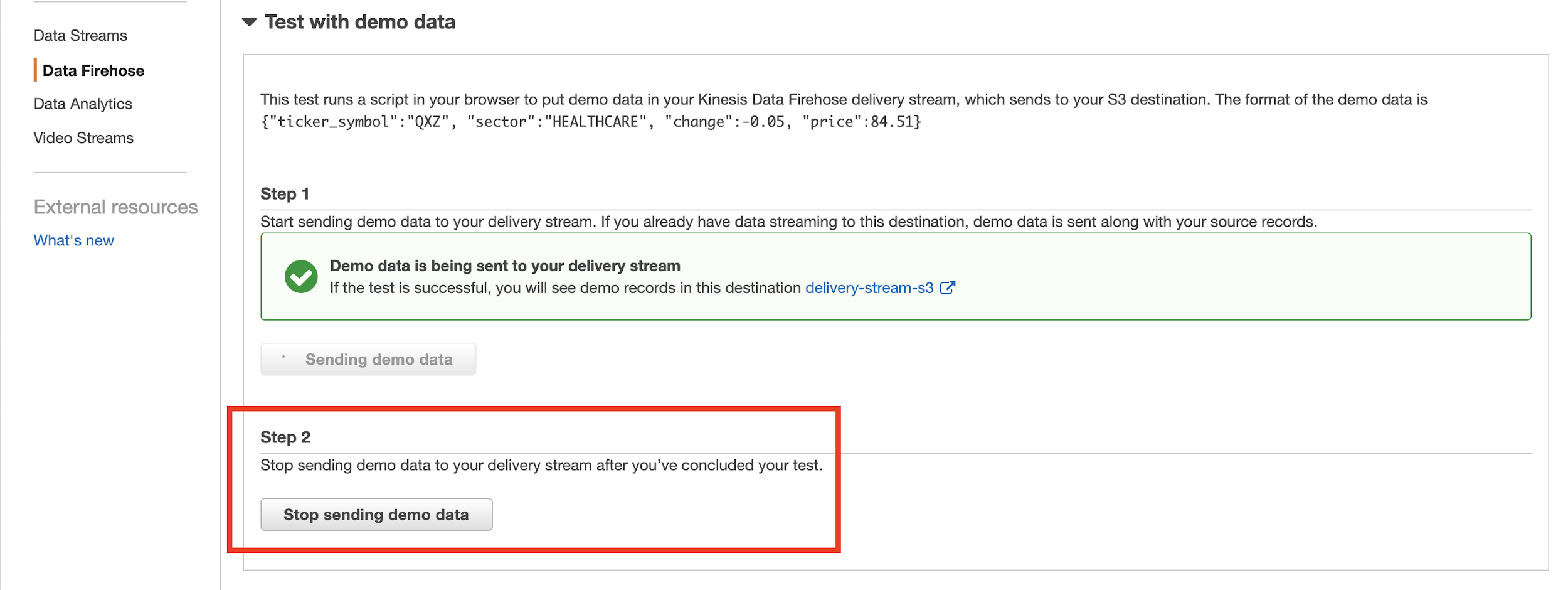


# Testing our AWS Kinesis Data Firehose

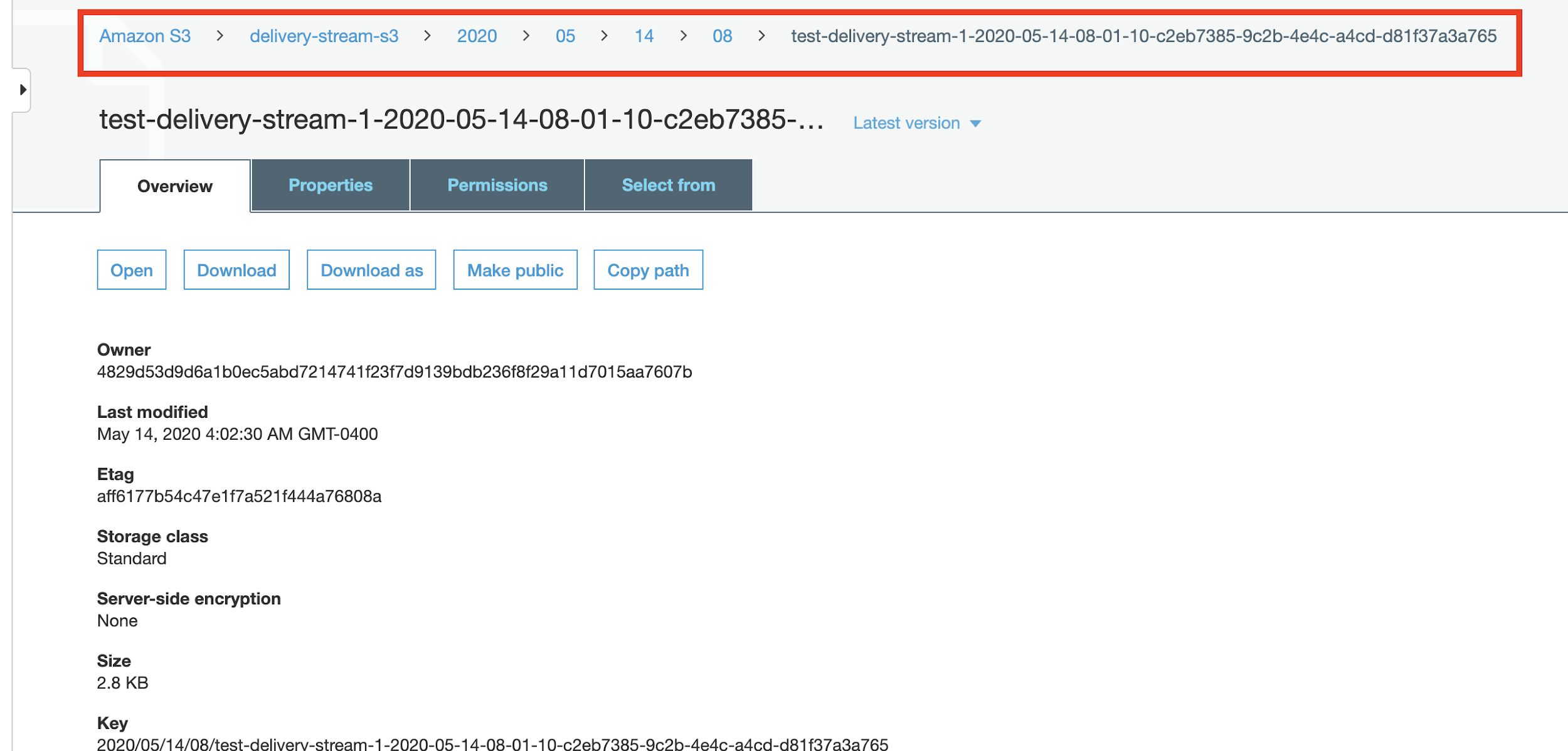
Now, let’s take this thing for a spin! Click into your delivery stream (from the image above, click on the name).



If you click on the **Test with demo data**, you will be provided with a button to start sending dummy data to your stream via the browser. Click on that button to start the test. **Wait about a minute or so**. And then click the next button to **stop** the test.



At this point, you **will have to wait one minute** as per the **Buffer interval** we set. Once that one minute is up, navigate to your S3 bucket, you should see a bunch of subfolders and at the bottom of it all - a file! (These subfolders are called partitions and basically AWS will stream data that was sent to the firehose per year/month/day/hour etc into individual folders).



On top the folder structure goes: 2020 > 05 > 14 > 08 > filename - this corresponds to: YEAR > MONTH > DAY > HOUR > FILES.

Downloading that file and opening, we notice something -

|  |
| --- |
| {"ticker\_symbol":"AZL","sector":"HEALTHCARE","change":0.85,"price":17.66}{"ticker\_symbol":"JYB","sector":"HEALTHCARE","change":1.94,"price":43.89}{"ticker\_symbol":"DFT","sector":"RETAIL","change":-1.44,"price":92.23} |

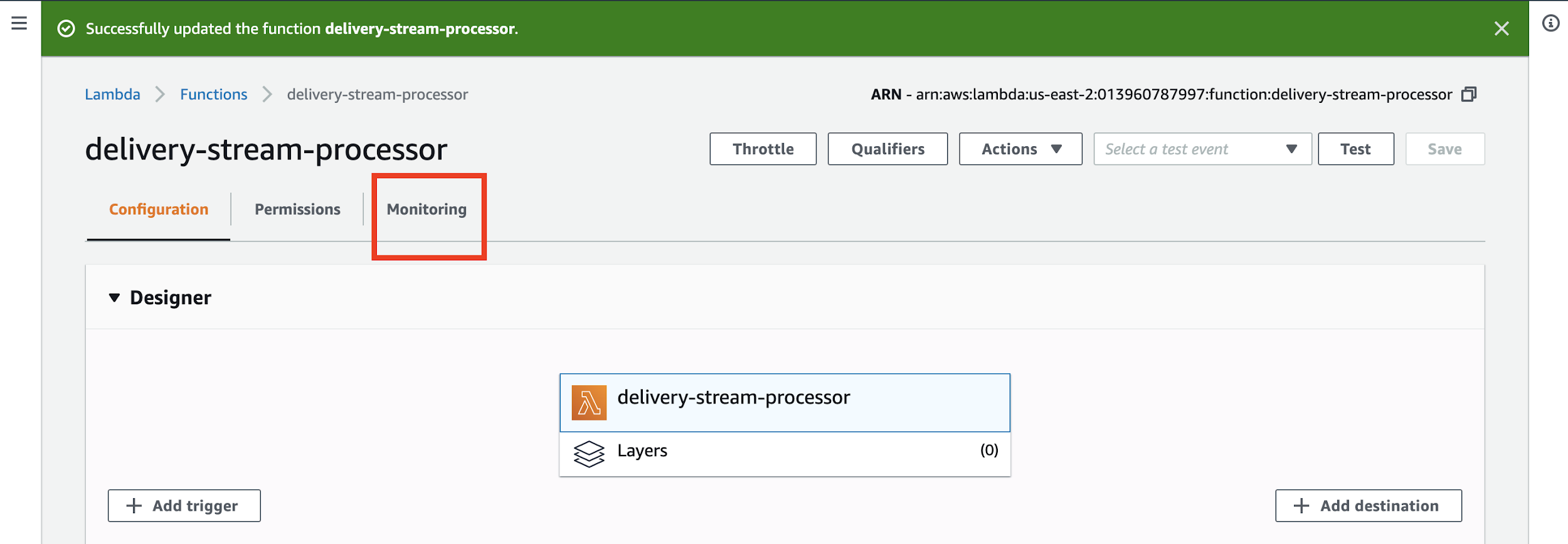
Notice how the JSON records are **not** on difference lines!! This is a problem because Glue will not recognize them as individual rows in a table. (If you looked at our yelp json dataset, you’ll note that it also consisted of “lines” of JSON, one per line. This is also why I insisted that our project 1 data as set up this way).

To fix, let us revisit our Lambda function and update slightly:

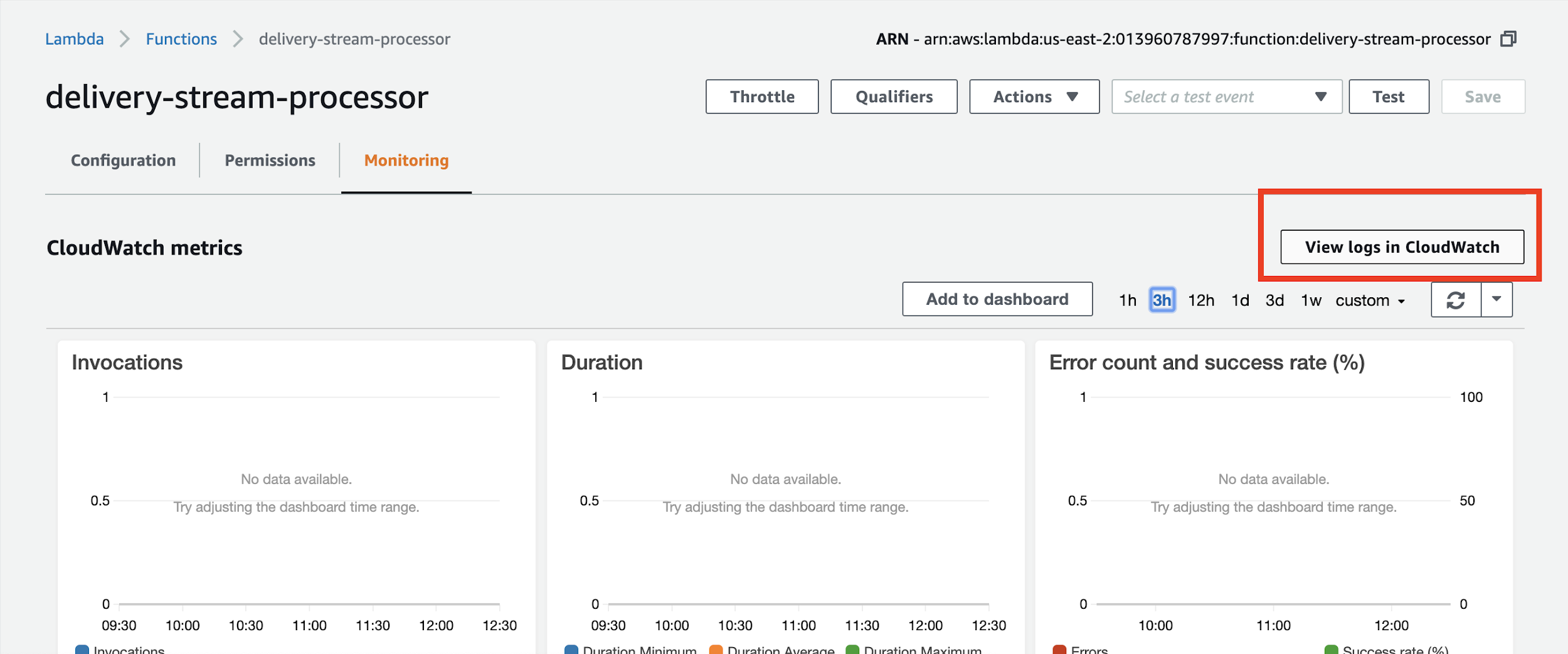
|  |
| --- |
| **def** **lambda\_handler**(event, context):  output\_records = []  **for** record **in** event["records"]:  print(type(record["data"]))  print((record["data"]))  output\_records.append({  "recordId": record['recordId'],  "result": "Ok",  "data": record["data"]  })    **return** { "records": output\_records } |

So notice that we added a few **print** statements - let’s use some of the built in debugging capabilities of Lambda to figure out why our JSON records are not showing up in their own lines. Because we are not running the Lambdas in our machines, we can’t just open up a terminal in debug. To that end, we will instead use an AWS service called **Cloudwatch Logs**.

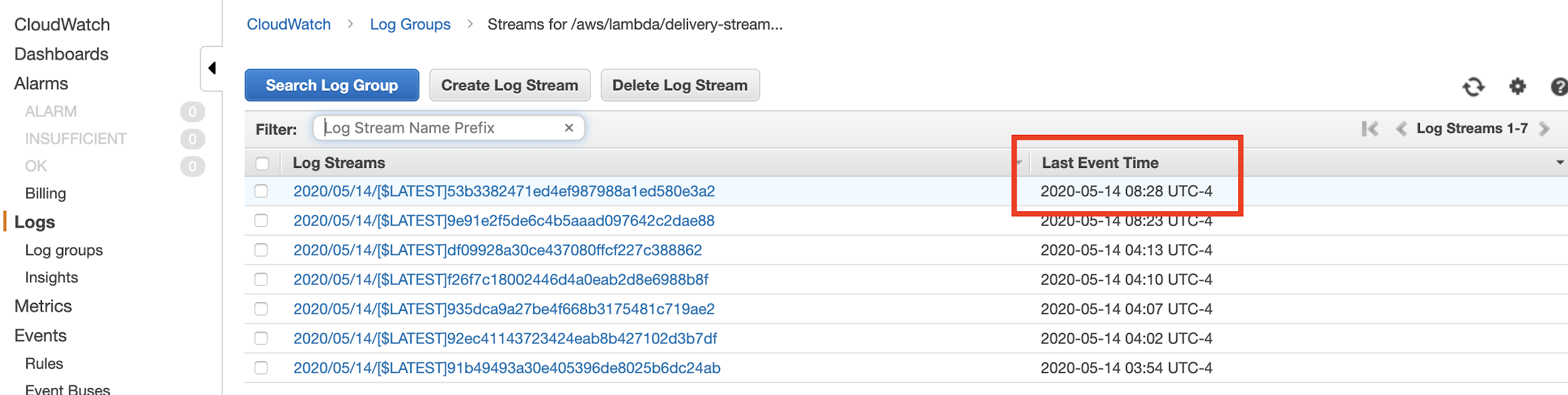
First, go to your Lambda homepage and click on the **Monitoring** tab:



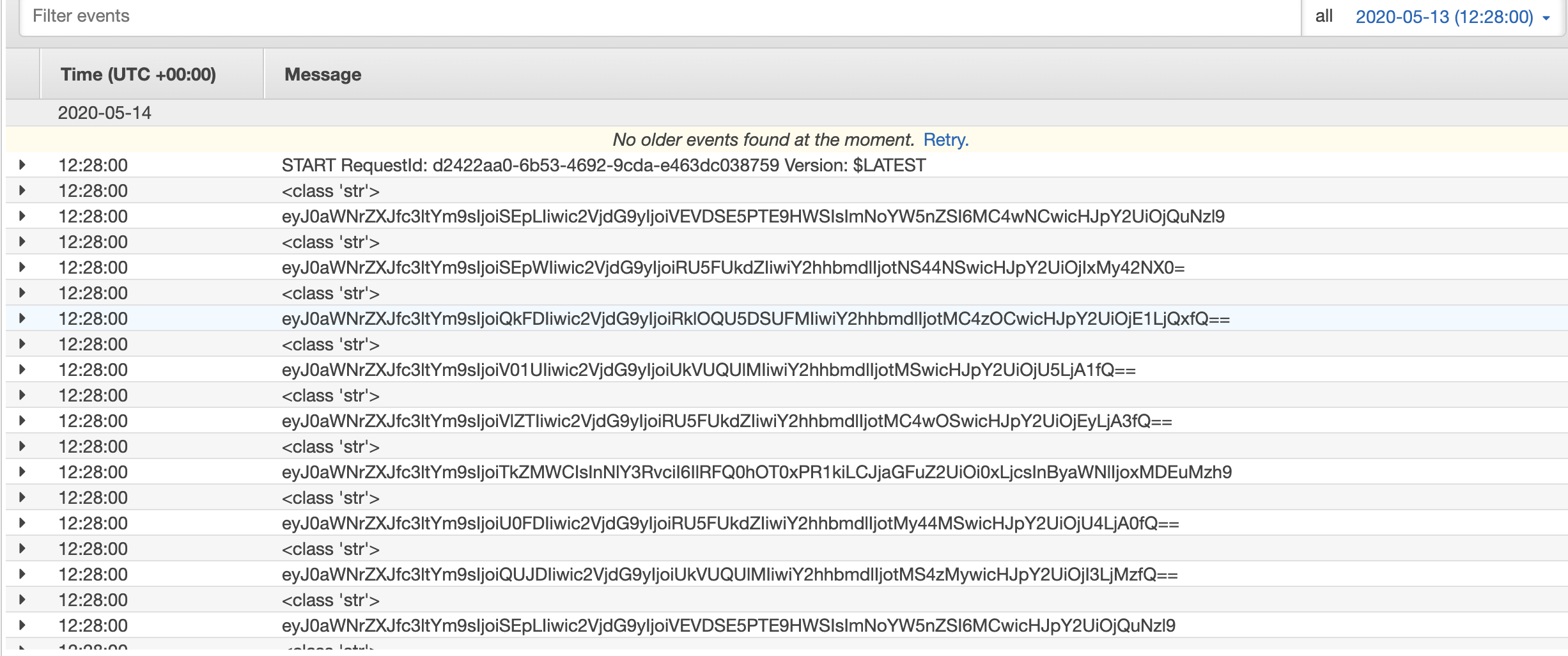
Then, click on the **View Logs in CloudWatch** button. Also, note that all of our Lambda usage metrics are empty because so far we have only been running this via a *test* from our delivery stream.



From the logs page, select the log group that is the latest:



These are your lambda function logs:



Usually, whenever you update your lambda function, a new “log group” will be created. Otherwise you can just hit the refresh icon on the top left of your log screen (not the browser refresh) to view new logs produced by your function.

For our purposes, note that the data that is being printed looks like gibberish but in fact it is actually a **base64 encoded** string, which takes up less space and therefore saves you money as AWS charges Firehose usage on data transfer (per 500TB).

So, in conclusion, in order for us to add our data 1/line, we must simply add the **base64 encoded** equivalent of a new line:

|  |
| --- |
| **import** base64 base64.b64encode(b'\n') |

The output of this is:

|  |
| --- |
| b'Cg==' |

The **“b”** is python’s way of telling you this is a **bytes** data type and not a **string** datatype. Not relevant to us for now. (PS: you can run this code above either in a REPL.itonline or by running:

|  |
| --- |
| docker run -it python:3.7 |

And then just pasting the two lines above).

So! Going back to our lambda function now, we can update it like so:

|  |
| --- |
| **def** **lambda\_handler**(event, context):  output\_records = []  **for** record **in** event["records"]:  output\_records.append({  "recordId": record['recordId'],  "result": "Ok",  "data": record["data"] + "Cg==" # this is the key here  })    **return** { "records": output\_records } |

If we went back and re-ran a test, navigated to our **S3** bucket and downloaded the latest file, we would see something like so:

|  |
| --- |
| {"ticker\_symbol":"HJK","sector":"TECHNOLOGY","change":0.04,"price":4.79} {"ticker\_symbol":"HJV","sector":"ENERGY","change":-5.85,"price":213.65} {"ticker\_symbol":"BAC","sector":"FINANCIAL","change":-0.38,"price":15.41} {"ticker\_symbol":"WMT","sector":"RETAIL","change":-1,"price":59.05} {"ticker\_symbol":"VVS","sector":"ENERGY","change":-0.09,"price":12.07} |

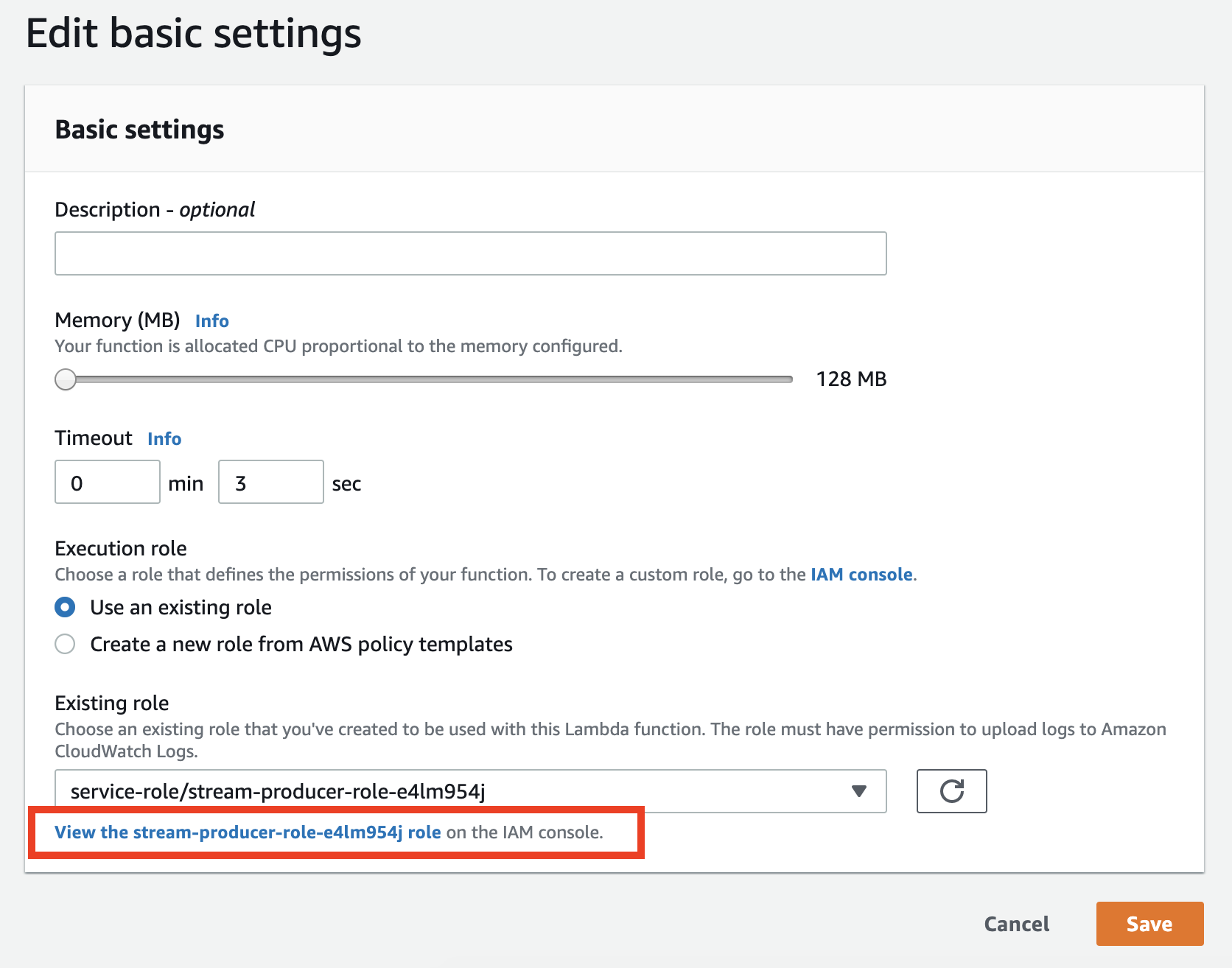
Tada! Our pipeline is now set up and ready to receive high throughput real time data for adhoc, interactive querying (with say Athena) or historical analysis (with say Spark)

# Streaming data to AWS Kinesis Data Firehose from Lambda

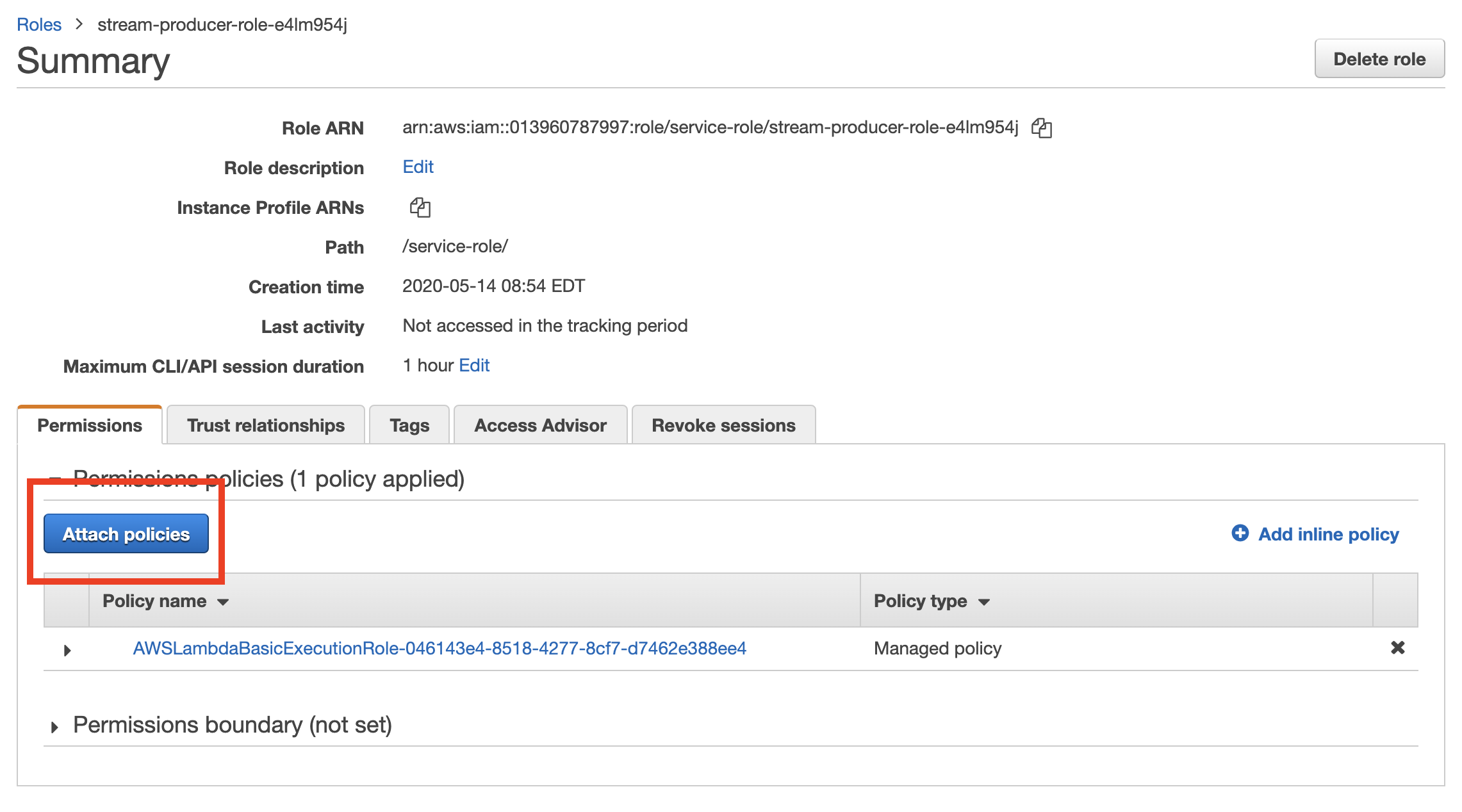
It behooves us to stream data *into* our firehose form Lambda functions, because as we established, lambda functions can be triggered on various occasions which is perfect for populating our data stream.

For now, I want to showcase a very simple use case and for our project 3, we will consider a more sophisticated use case.

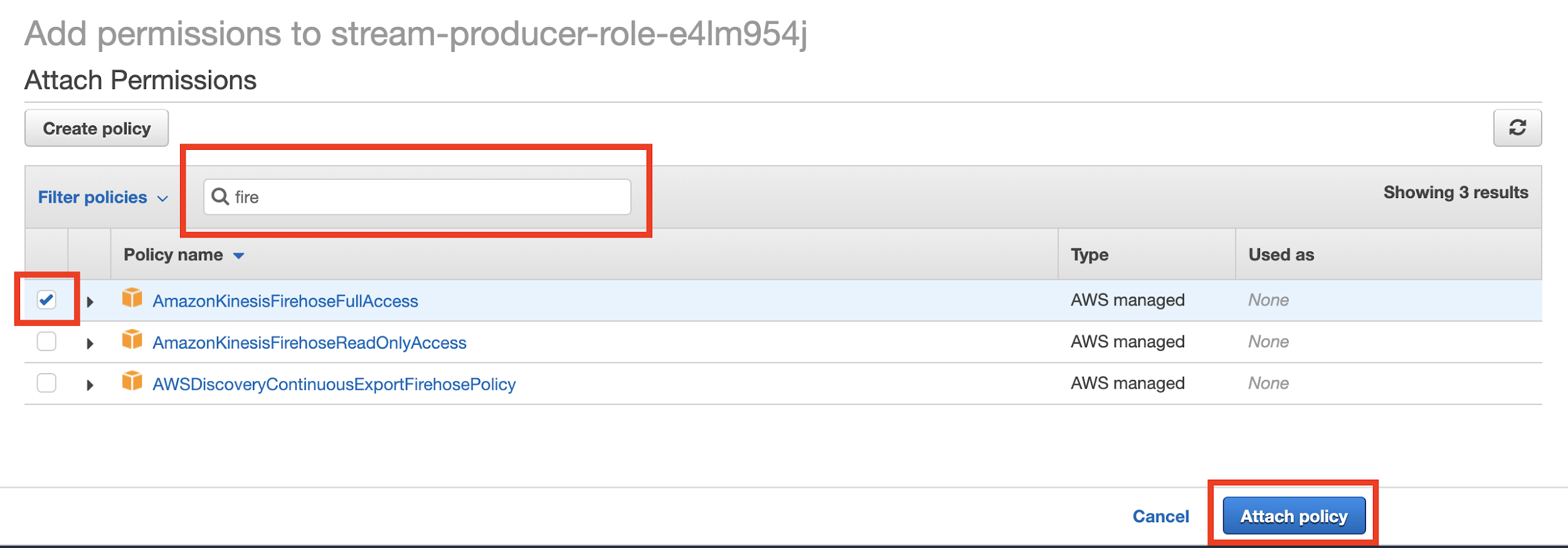
**First**, create a Lambda function (a **simple** one) as per the guidelines in Lecture 11. Then, go down to the **Basic Settings** block and hit **Edit**. From the subsequent page, click on your Lambda **policy**:



This will open a new tab as follows:



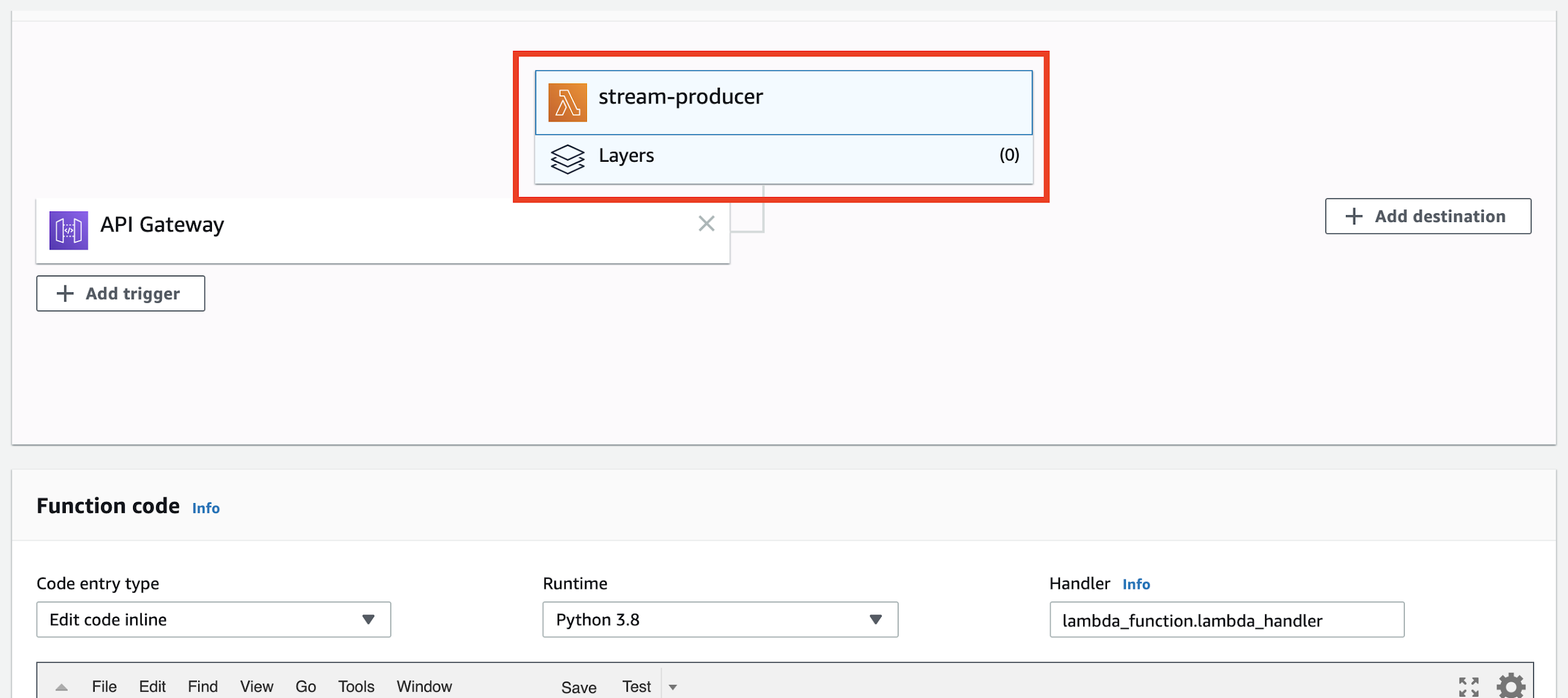
In the policy page, look for **firehose** and attach that policy.



A few notes:

* A **Policy** is a construct in AWS that allows various services to talk to one another. If you want Lambda to be able to produce to a firehose, you must explicitly give it permissions to do so via attachment of a **policy**.
* We are taking a very very *lax* approach here for the sake of simplicity. When there are many services and many teams, etc, we usually take advantage of the configuration features of a policy and specify **specific** firehose streams (or other specific instances of services available).

Once you’ve attached your policy, you are good to go! **Remember:** to open up the code viewer, **click** on your function name in the function landing page:



As you can see, clicking on the block containing your function name will manifest the code editor for your function below it.

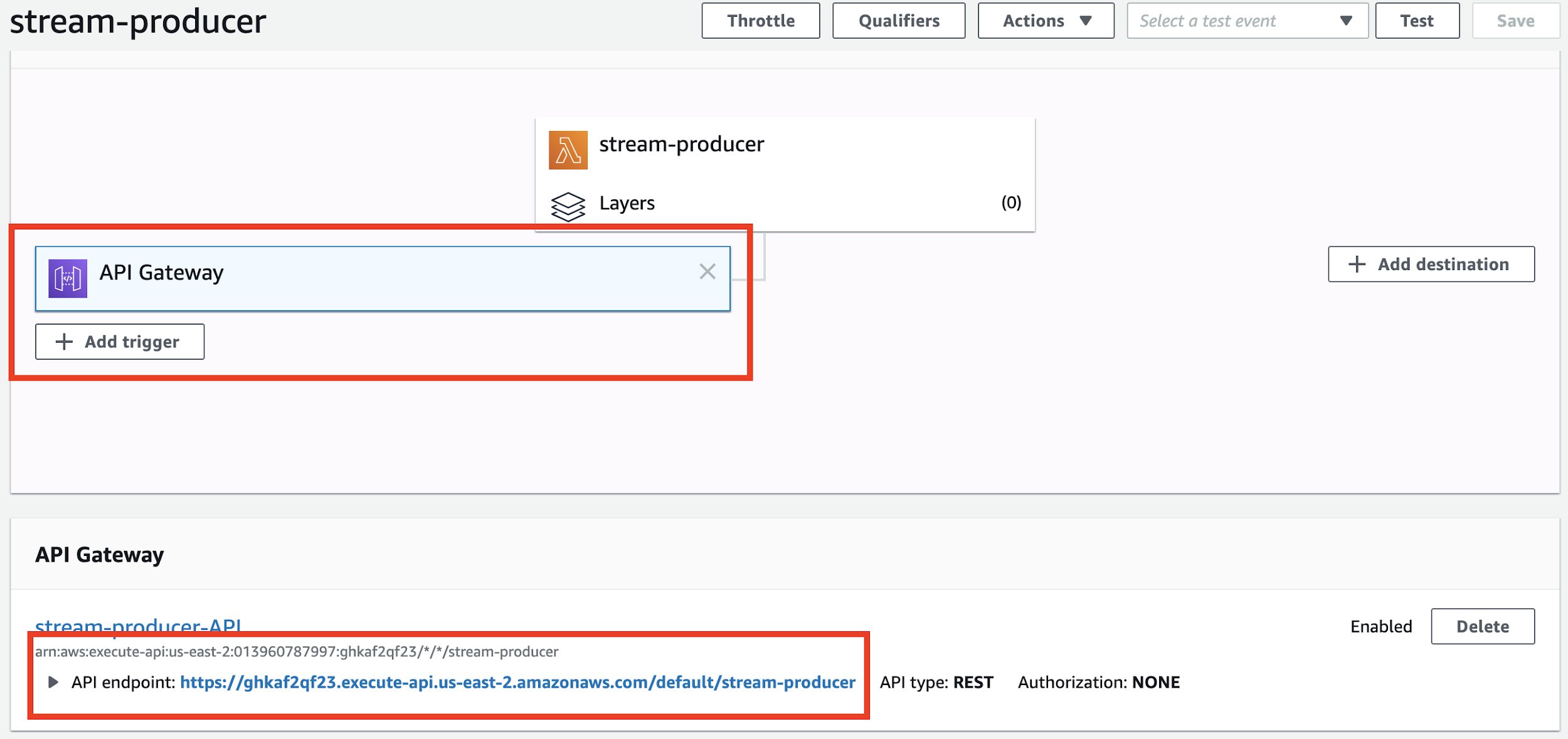
In the code block, replace the existing code base with the following code:

|  |
| --- |
| **import** json **import** boto3 **import** random  CHOICES = ["TECHNOLOGY", "ENERGY", "FINANCIAL"]  **def** **lambda\_handler**(event, context):  # just generate some random data  data = {"ticker\_symbol":"HJK","sector":random.choice(CHOICES),"change":0.04,"price":4.79}    # convert it to JSON -- IMPORTANT!!  as\_jsonstr = json.dumps(data)    # initialize boto3 client  fh = boto3.client("firehose", "us-east-2")    # this actually pushed to our firehose datastream  # we must "encode" in order to convert it into the  # bytes datatype as all of AWS libs operate over  # bytes not strings  fh.put\_record(  DeliveryStreamName="test-delivery-stream",   Record={"Data": as\_jsonstr.encode('utf-8')})   # return  **return** {  'statusCode': 200,  'body': json.dumps(f'Done! Recorded: {as\_jsonstr}')  } |

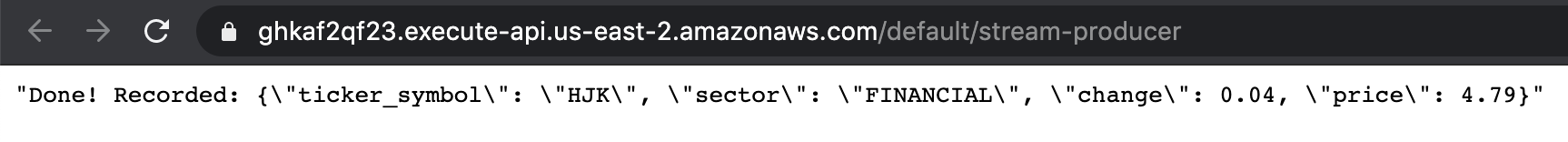
Basically what this function does is the **same thing** our delivery stream **test** option did, but now via our own codebase in python.

We generate one of three random **sector** values, create a **dict** that stores it, convert it to JSON and then **manually produce a data entry** for our delivery stream. This delivery stream then pushes it to S3 making it available for usage by other services like Athena or Spark. Moreover, because we can now achieve this with python code, we can of course produce *any type of data* we want from any data source!

Because our Lambda is triggered by a browser URL call, you can “trigger” this lambda function by opening the URL in your browser:



(Click on the **API Gateway** trigger to view the URL that will run your lambda). When you click into it, you should see something like so (two examples provided):



Our final validation here would be to check out **S3** and ensure that these records made it there. A few parting notes:

* **Boto3** is AWS’s official python library for interacting with their services. Always use boto3 if producing to firehoses or reading from s3 or any other programmatic interaction with your AWS services
* We are using the firehose.**put\_record**, which allows us to add records **one at a time**. For more real use cases, it is much better to use [put\_record\_batch](https://boto3.amazonaws.com/v1/documentation/api/latest/reference/services/firehose.html#Firehose.Client.put_record_batch) instead.

# Final thoughts

So first - let us recap what the heck we just did here:

* We created a **Delivery Stream**, a very powerful construct that allows us to collect real time data.
* Our **Delivery Stream** supports running an AWS Lambda function to transform records before they are dropped into our distributed file system (in this case S3). This is powerful because we can “cast a wide net” but normalize all the various values that we accumulate from various sources.

Essentially streaming allows us to satisfy the **3 Vs** of big data from lecture one:

* Volume (Firehose gives us the capability to basically scale to 100s of TBs of data)
* Variety (The lambda processor we attach to the *stream* (not the producer we just wrote above) gives us the ability to normalize incoming data)
* Velocity (With Firehose producing to S3, we can leverage Athena to run analysis on this status *as it is coming in*, in near real time),

As a side note, the S3 data isn’t going anywhere meaning we can also run historical analysis on all the data we accumulated with MapReduce technologies (such as Spark on EMR).

Of course, we might want to have multiple ways to produce content into our datastream, this is where AWS Lambda functions come in as the allow us to provision multiple functions off of multiple triggers that can produce to the same stream.