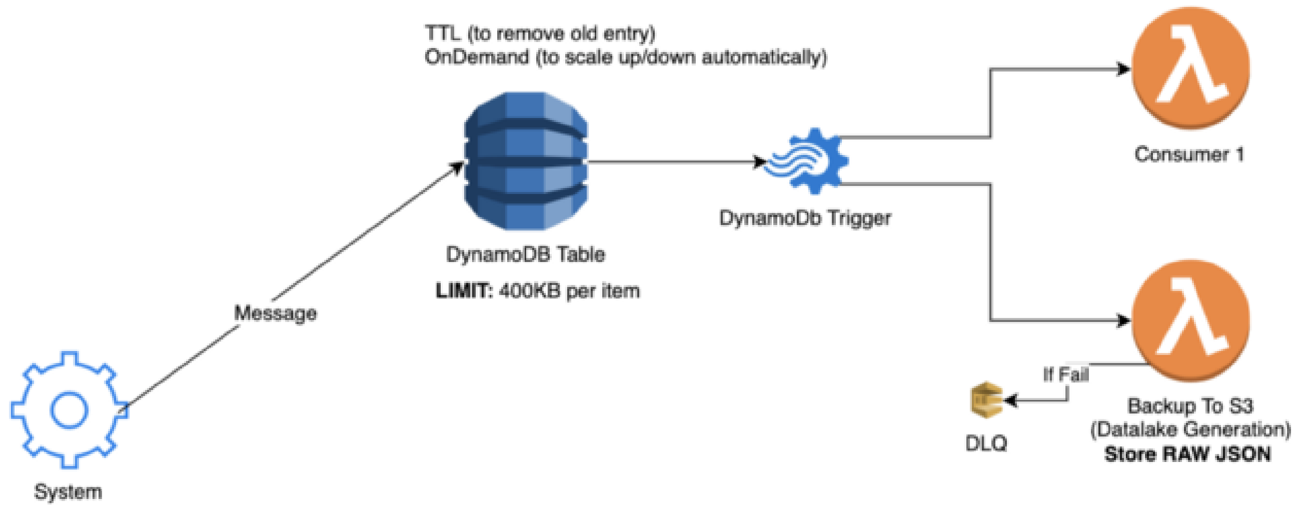
Alberto Cubeddu Follow
Aug 15, 2019 · 5 min read ★

AWS DynamoDB Triggers (Event-Driven Architecture)

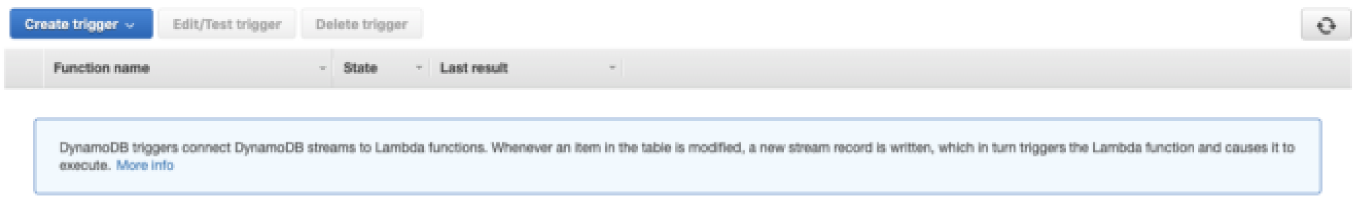
DynamoDB Streams

There are a lot of implementations to achieve **event-driven architecture.** However, today, we will focus on **DynamoDB Streams**. Thanks to this NoSQL database, you can leverage the power of a schemaless database and at the same time, the event-driven architecture pattern.



An example of this architecture is represented in the screenshot above us, where you have your System (“**The Producer**”) creating **JSON** Payload(s) that are sent to **DynamoDB.** After arrival, **DynamoDB Trigger** will execute the **Lambda functions** called **consumers**(*).

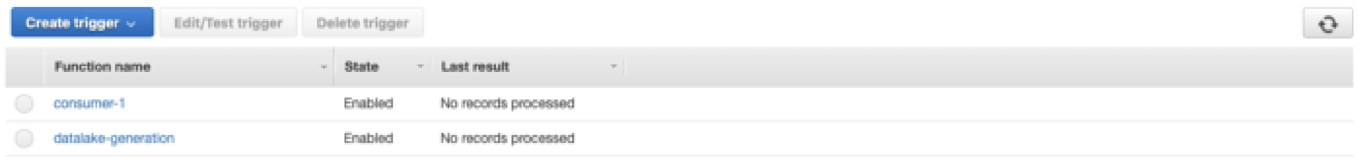
Whenever an item in the table is **created/modified** or **deleted**, a new stream record is written, which in turn triggers the Lambda function and causes it to execute.



DynamoDB triggers connect DynamoDB streams to Lambda functions. Whenever an item in the table is modified, a new stream record is written, which in turn triggers the Lambda function and causes it to execute. More info

DynamoDB Triggers Interface (from the WEB UI)

(*) The **recommended** number of lambda functions per stream is 2; However, as you can imagine, having just two lambda functions can't be enough. That's the reason why I decided to have one lambda acting as a **data lake** and leave the other one as a **generic consumer for fan-out**.



Function name	State	Last result
consumer-1	Enabled	No records processed
datalake-generation	Enabled	No records processed

The two triggers activated to call the respective lambda functions

DynamoDB Stream — Explanation

The stream is a feature that allows DynamoDB to write any action as a series of event dispatchable to a consumer, whenever is an INSERT, MODIFY or REMOVE.

You can choose between different information to be sent whenever the source table has been modified, such as :

- **Keys only** — only the key attributes of the modified item.
- **New image** — the entire item, as it appears after it was modified.
- **Old image** — the entire item, as it appeared before it was modified.
- **New and old images** — both the original and the old images of the item.

Lambda — Data lake Generation

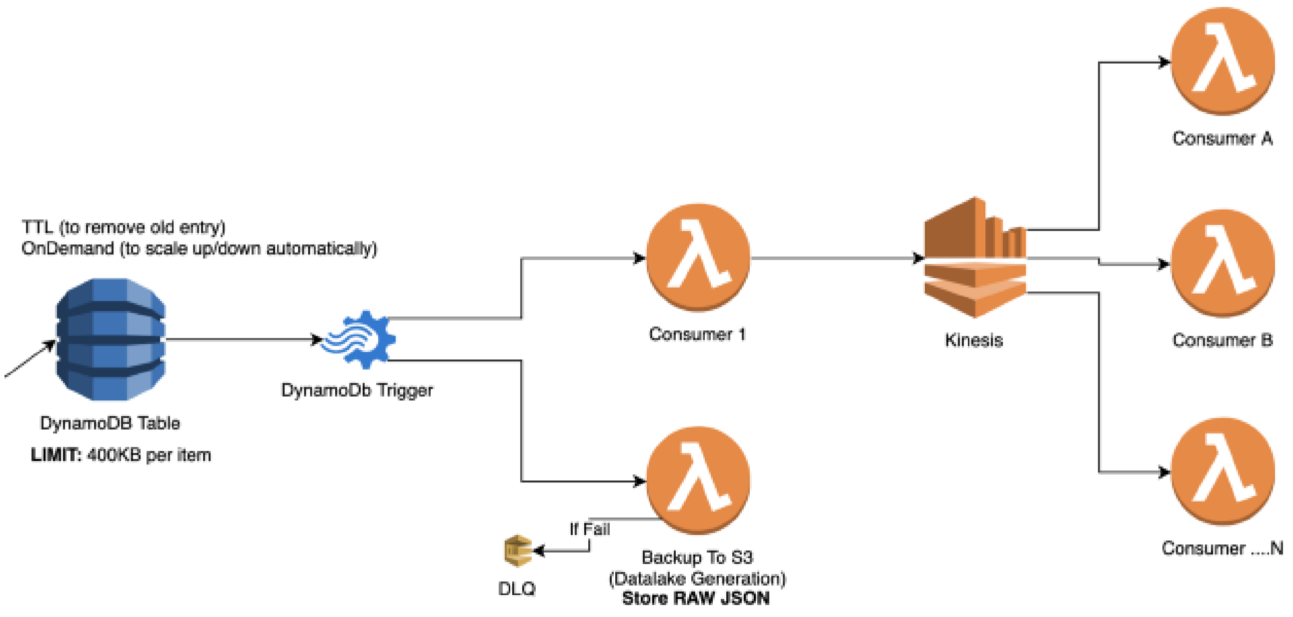
Let's talk about the data lake generation's lambda function. This FaaS is responsible for storing all the events generated by the DynamoDB trigger; this allows us to have all the JSON events for:

- Audit
- Re-run/retrial
- Unit-test (very useful when you found edge-cases, and you can reuse the event data in your tests)

The code behind the lambda function has just one responsibility, receive the JSON event from the DynamoDB Trigger and store it in a bucket.

Consumer 1 — Fan-Out

Due to the limitation of having just two consumers, we can use fan-out using Kinesis to be able to solve the deficiency.



Consumer 1 Fan-out to a Kinesis Stream

Using this approach, we can send as many events as we want to as many consumers as we need!

DynamoDB — Configuration (TTL)

The Time To Live is a beneficial configuration if you want to use DynamoDB **only to produce events** and **not for storing other information**.
The TTL will help you to reduce the amounts of items in your table, moreover, will help you **cut costs**.
Just remember that if you want to use this configuration, you need to have an **attribute** containing the **time in epoch format.**

If you enable the DynamoDB Streams, you can even create some workflow for backing-up the deleted documents. (This is not part of this tutorial).

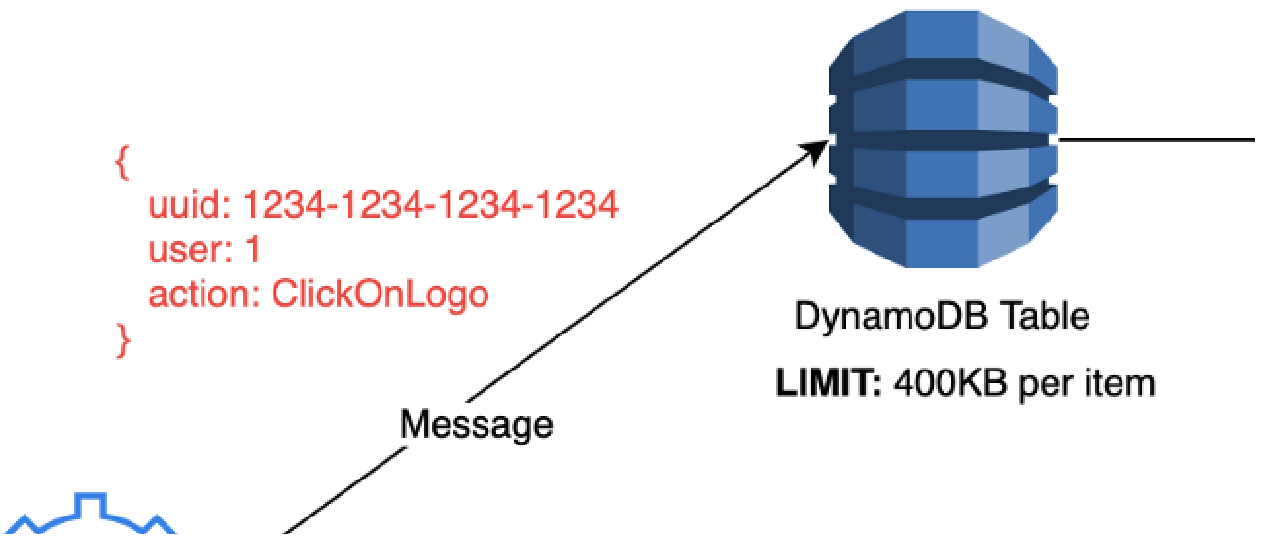
DynamoDB — Configuration (On-Demand)

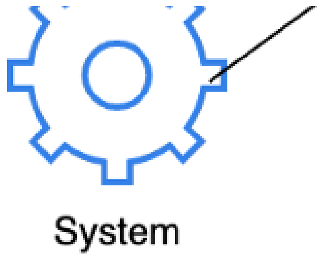
DynamoDB **On-Demand** allows you to scale up/down according to the serverless philosophy to scale horizontally almost to the infinite. I will suggest you use on-demand capacity **only if**:

- Unknown workload or unpredictable application's traffic
- Oscillation in your workload with unpredictable peaks

An Example is worth 1000's of words

The System will create a **JSON payload** to be sent over **DynamoDB**



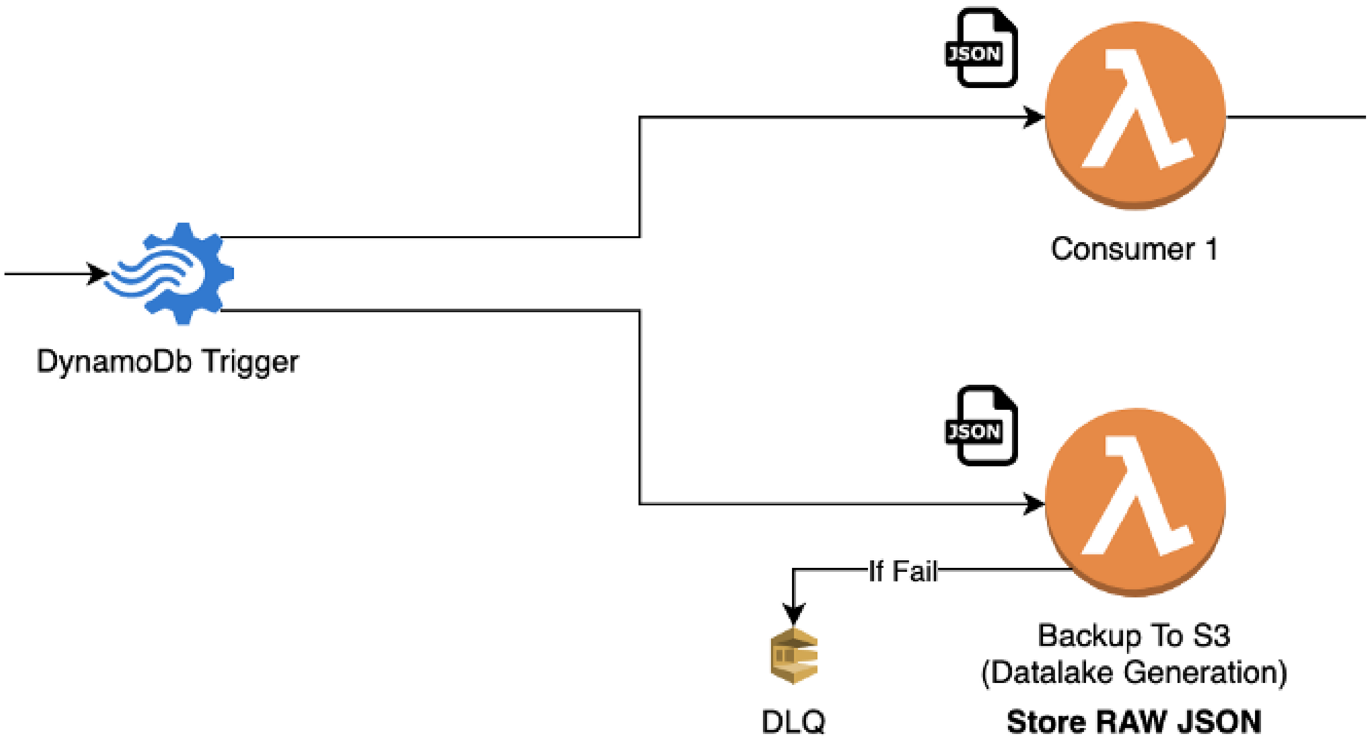


The System is "The Producer" you can use lots of different programming languages to ship an item to DynamoDB

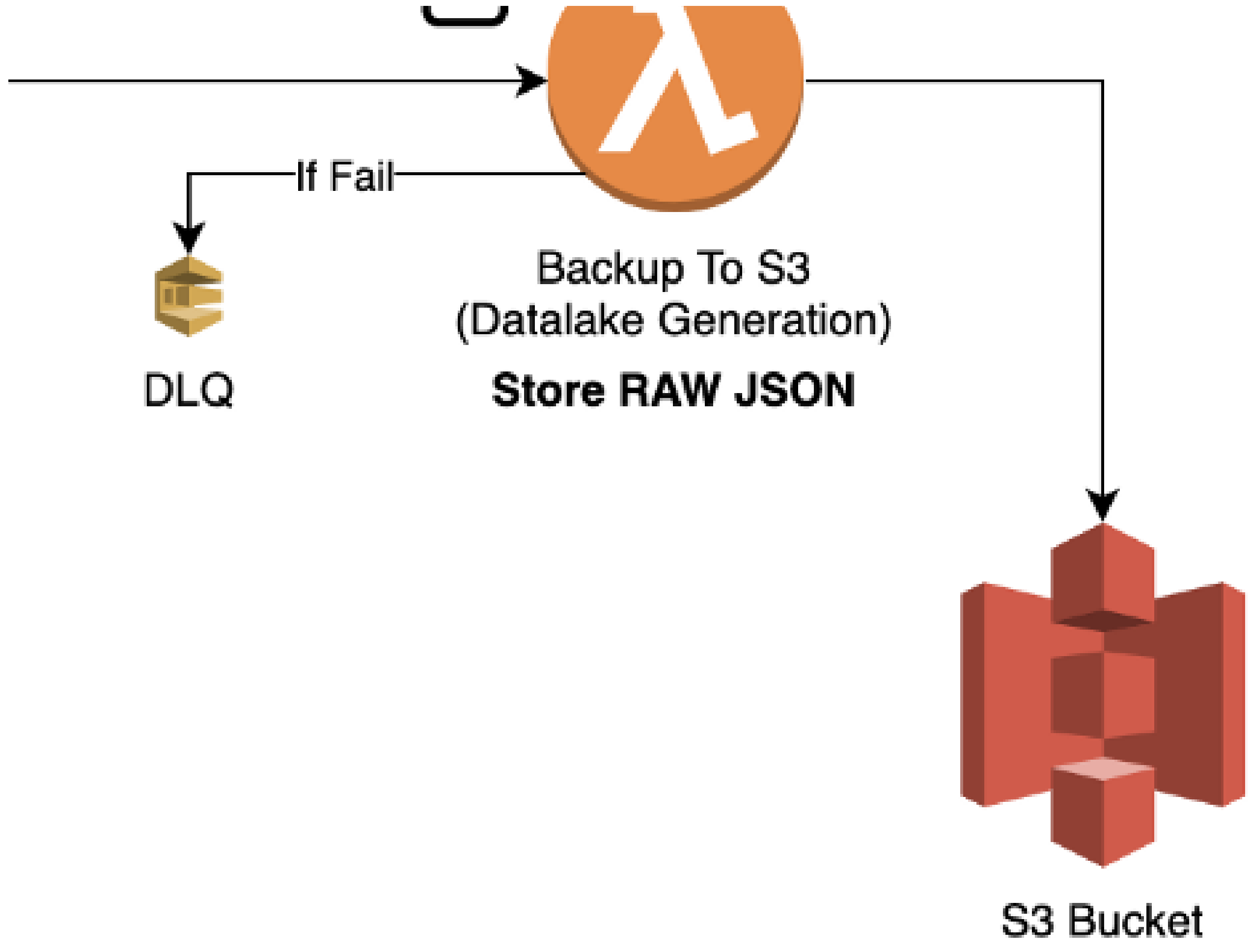
After the event has been sent to the DynamoDB Table, the Triggers will take place, and it will generate the **JSON.** Depending on the configuration (e.g. In this case, the stream configured on **NEW_IMAGE**) you will have a different structure.

```
{
  "Records": [
    {
      "eventID": "2a7961c7e109606c28e0d820f0b4a28f",
      "eventName": "INSERT",
      "eventVersion": "1.1",
      "eventSource": "aws:dynamodb",
      "awsRegion": "ap-southeast-2",
      "dynamodb": {
        "ApproximateCreationDateTime": 1562819720,
        "Keys": {
          "uuid": {
            "S": "1234-1234-1234-1234"
          },
          "user": {
            "S": "1"
          },
          "action": {
           "S": "ClickOnLogo"
          }
        },
        "NewImage": {
          "uuid": {
            "S": "1234-1234-1234-1234"
          },
          "user": {
            "S": "1"
          },
          "action": {
           "S": "ClickOnLogo"
          }
        },
        "SequenceNumber": "100000000007063771489",
        "SizeBytes": 44,
        "StreamViewType": "NEW_IMAGE"
      },
      "eventSourceARN": "arn:aws:dynamodb:ap-southeast-
2:xxxxxxxxx:table/testProducerTable/stream/2019-07-11T04:32:45.949"
    }
  ]
}
```

DynamoDB Trigger will send the above JSON to the two consumers:



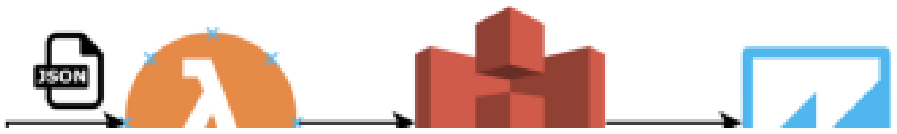
The Backup To S3 is used to store all the events in an S3 bucket:

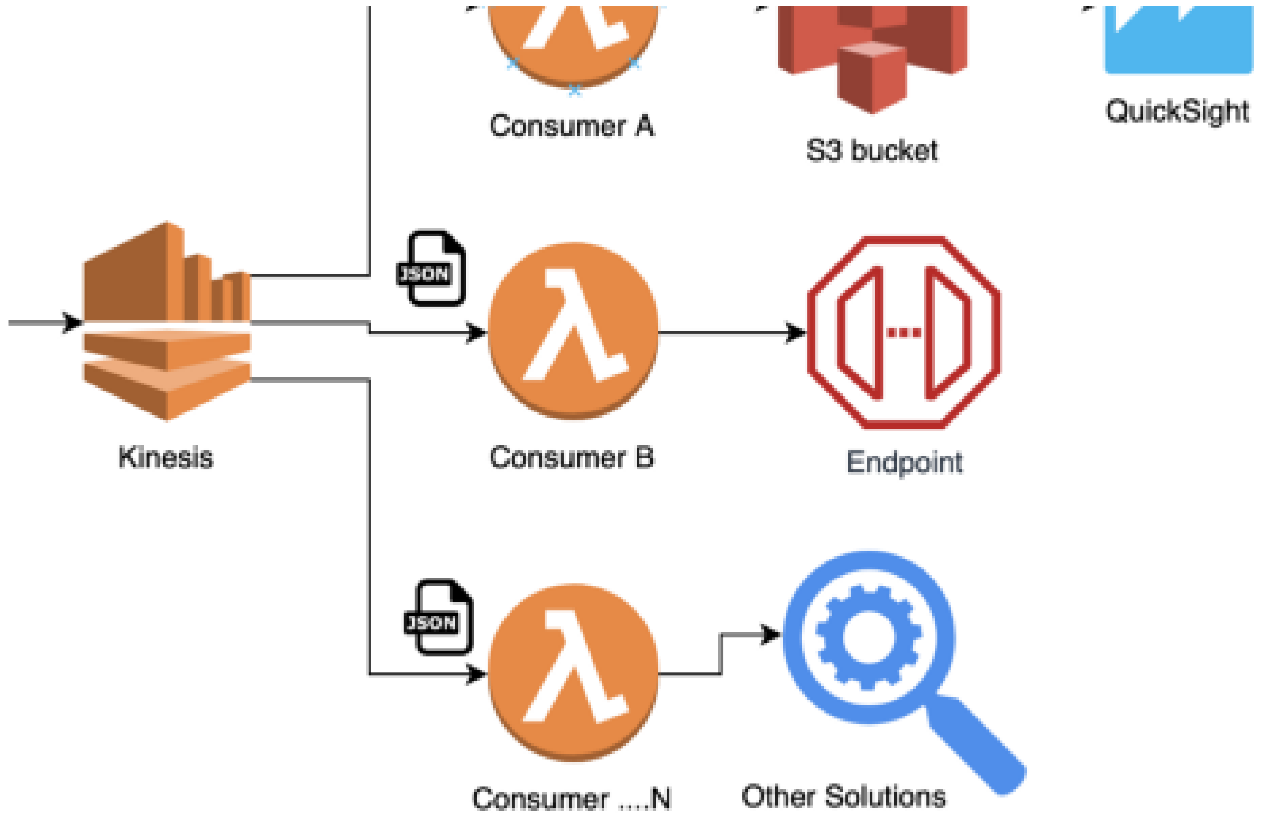


Storing all the events in a JSON format into an S3 Bucket

This can be useful in the future, for example, if you want to use the S3 Bucket as a Data Warehouse to run **AWS Athena.** And also as we said before, you can get all the events stored and re-run it on other lambdas.

Talking about **fanning-out** and **Kinesis Streams**:





Where the **Consumer A** is writing JSON object to an S3 Bucket that then can be read by QuickSight to visualise your data

Consumer B will call another endpoint (Lambda, HTTP/https, etc.etc.)

Consumer N all the other possible solutions in an event-driven architecture

Conclusion

Thanks for reading the article, hope it will help you on your journey with the EDA (Event-Driven Architecture). If you have any questions:

View my LinkedIn® Profile

Follow me on Linkedin

Best regards, Alberto Cubeddu (You can follow me on medium!) Remember to **clap the story** to help other people find it easily on medium!

AWS Event Driven Architecture Event Driven Systems Dynamodb Stream Dynamodb Trigger