# Cisco Meraki: Simple Data Stream Analytics

**Task:** In an incoming data stream, create a service to calculate the minimum, maximum and average value for each device in each minute and store it to database.

**Approach:**

Input – In real world, the streamed data is mostly accessed through an API or through some port which is listening to it. To keep things simple for this console application, a .txt file is utilized to access the streaming data. The input data will be stored in the .txt file in the form of a string in the following format – ‘device\_id: 2, value: 3, timestamp: 1611741659’.

Base Architecture – To handle such a data stream in the service, there is a need for a distributed stream processing framework. For this scenario, I chose Apache Kafka distributed publish-subscribe messaging system. Apache Kafka is chosen due to its performance and scalability characteristics. It is a fault-tolerant stream processing system which can handle large amounts of data with ease. Kafka’s ecosystem consists of producers, topic and consumers. Producers are used to produce messages to a topic. Here topics are the logs where the streaming data is stored across multiple partitions depending on the size in the Kafka cluster. Consumer then consumes(read) the messages and perform the business logic required on the streaming data.

In the service, the producer reads every line of .txt file and send it to the Topic (i.e. CiscoMeraki). Kafka Producer and Kafka Consumer’s APIs are utilized to communicate with the data.

Processing Data – An event processing framework is required to perform requisite processing on the incoming real-time data stream. In this application, Faust is utilized as the stream processing library. Faust is a python library to build high performance distributed applications and real-time data pipelines to process billions of incoming events every day. This library enables distributing the workload among multiple agents present in the same cluster. It also supports asynchronous/await jobs which can be split among different agents. Another reason to choose Faust is that it is strongly compatible with Apache Kafka and uses Kafka as its source of data.

In the service, Faust will consume the streaming data from Kafka Topic and divide the processing into multiple asynchronous jobs.

Storage Database – After the processing the data, Mongo DB is used to store the output of the service. Since Mongo DB is a NoSQL database which provides scalability and flexibility when it comes to querying and indexing data, it is chosen for the service.

In the application, Mongo DB’s asynchronous driver, Motor, is used to support Faust’s async jobs. Motor’s AsyncIOMotorClient is used to connect with MongoDB.

**Workflow:**

1. First the data is stored in the .txt file in the given format. In the main.py, a Kafka producer is initiated which will read the input from .txt file and validate it with the given format. After validating, it will send to the Kafka topic defined synchronously.
2. In the process\_faust.py, a Faust App with a consumer group is defined to consume the data from defined Kafka Topic. A Faust Table is created to store the intermediate processed streaming information. Faust Table is a distributed in-memory dictionary which provides scalability and reliability and handles thousands of records with ease.
3. A Faust streaming agent i.e., receiver will consume the incoming data stream. The agent receiver will segregate data on the basis of each min i.e. timestamp. The timestamp is converted to nearest starting minute. If the incoming line of stream belong to the same minute, it is sent to async method eval\_stats. Here, the data is segregated on the basis of device id. In async eval\_stats, the minimum value, maximum value, total count and sum of each device id is computed and stored into Faust Table where the key is the combination of device id and timestamp.
4. If the incoming line of stream belongs to the next minute, then the computed data present in the table for the current minute is sent to async method avg\_stats where the average value for each device id in each minute is calculated. After calculating the average value for all devices in the given minute, the data is sent as json strings to async method store\_to\_db.
5. The application supports a delay of 60 seconds i.e., 1 minute in the streaming data. If the incoming line of stream belongs to the previous minute, then the data is sent to an async method handle\_delay. Here the intermediate values are computed in the same fashion as it is computed in async eval\_stats and stored in dictionary delayed\_devices.
6. Once the average value is calculated for all devices in the given minute, the values are inserted in the database with the help of insert\_one() and insert\_many() await calls depending on the number of incoming data for each minute.
7. For delayed stream of data, the flag is set to 1 indicating that it is delay. When the delayed stream of data is sent to async store\_to\_db, it first checks whether an existing entry for particular device id in particular minute exists or not. If it exists, update it with new information and if not then simply insert it into the db.
8. For the incoming stream of data, all the event processing is performed in an asynchronous and distributed way. To handle delayed information a timer function per\_min is defined which will run after every 60 seconds. It will process the delayed stream of data received in the given minute.
9. The async store\_to\_db utilizes mongo db’s async insert\_one(), insert\_many(), find\_one() and replace\_one() calls to insert and query data with mongoDB.
10. When the application reaches the end of the data stream, the timer function will process the last minute i.e., last timestamp when no other information is coming for that particular minute anymore.
11. The current limitation of this service is that delayed stream of data can’t be stored in Faust Table as the table can only be modified while event streaming not through timer functions.

**Future Scope:**

Though the scale of this application is small, there are many things which can be done if given more time.

1. Perform thorough error handling to check all the async jobs are running perfectly
2. Perform Garbage collection of Faust Tables in an effective manner while event streaming
3. Try to test and improve the scalability of the application.