Microservices Log Analysis and Monitoring

Group 6

Riken Shah (rshah9) Daxkumar Amin (dkamin) Khantil Choksi (khchoksi)

CSC 591 - Data Intensive Computing December 10, 2018

Abstract

In this project, we propose building a data-intensive pipeline to process real-time high velocity stream of microservices log data. The system is built upon various technologies and services mainly provided by Amazon Web Services. Primarily, there are three phases in the system, data generation and ingestion using semi-synthetic producers, data streaming and processing, and data visualization and monitoring. The system is designed as per Lambda architecture consisting of speed layer using Amazon Kinesis services, batch layer using DynamoDB and serving layer using Kibana. Use of state-of-the-art technologies like Kinesis Analytics and Elasticsearch makes the system fault-tolerant, highly available and scalable. Each component is completely configurable which makes the system flexible in terms of adapting to needs of one particular solution without compromising on its core features. Also, use of other technologies such as CloudFormation makes our system easily deployable and maintainable.

Keywords: Lambda Architecture, Amazon Web Services, Log Processing, Kinesis, Kibana

1. Introduction

Logs are a fundamental unit of any system. With ever-increasing amount of logs, the need to have robust systems that can effectively process, store, and analyze those logs have also increased. It's not only imperative to manage and store the logs but also to derive actionable insights from those. Manual monitoring of system through logs becomes increasingly difficult as the system complexity and scale increases. Today we are moving from monolithic applications to the microservices, where each service in the system works independently and might produce one or more log streams based on processes that it is running. In this paper, we describe a system that we have developed in order to effectively manage such microservices' logs, store and process them in a reliable way, and provide accurate actionable insights in user-defined configurable dashboards that can be effectively used as one single view of the whole system.

Another major factor to be considered in log processing and monitoring system is latency. We need to make sure that the insights that are required by the end user are delivered in the timely fashion, else those won't be very relevant. The need for such a low latency system can only satisfied with the use of an architecture that supports the handling of streaming data. When lots of such microservices generate such high velocity data, the system should be able to handle huge volumes of such streams in a fault tolerant and reliable manner. Our system exactly fulfills this niche by adopting lambda architecture.

Some of the technical concepts that we talk about are Stream Processing, Lambda Architecture, and Infrastructure as Code. For stream processing, we use Kinesis Streams, Firehose Delivery Stream, and Kinesis Analytics. We provide configurable visualization using Kibana. We have reviewed and taken inspiration from many similar systems such as NewRelic [1]. They provide one-stop solution for variety of features like multi-tenancy, security and variety of plug-ins through SaaS (Software-as-a-Service) model. AWS CloudWatch and CloudWatch Metrics are the in-build AWS services that can process log data and provide options to set up alerts. Our system is more general version of such solutions which can be extended/configured as per user requirements.

The rest of the paper is organized as follows. In section 2, we give an overview of the system, various technical components, and how the data flows inside the system. Section 3 describes our system architecture in detail and information about various modules of the system. We also provide justification on our design decisions and discuss features each module accomplishes. We describe how our infrastructure is built in section 4, primarily as AWS tech stack and in section 5 we talk about visualization interfaces in detail. We conclude in section 6 along with providing high level analysis of system including pros and cons.

2. Overview

There are majorly three components in our system. Semi-synthetic data generators produce microservice logs as per custom configurations, which can be adjusted to simulate behaviors like normal traffic, very high traffic, anomalies and failures, high response times, etc. The second component is our streaming pipeline which handles the high velocity data produced by the generators in fault-tolerant and reliable fashion. The third is our visualization dashboards that are generated using Kibana and are designed to be configurable based on what data the user is currently interested in. The overall system can be viewed in figure 1.

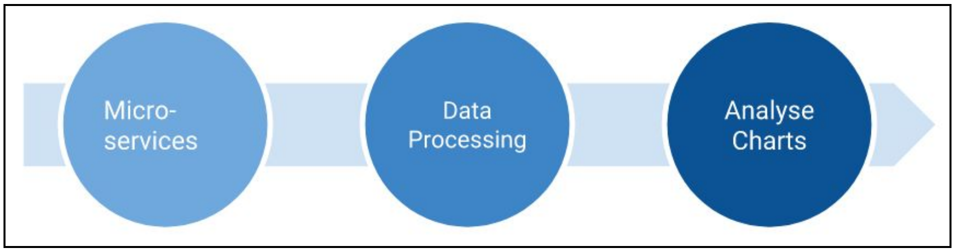

Fig 1: System Overview

The data that is generated by the semi-synthetic producers contains following attributes - Response Time (ms), Response Code (2XX, 3XX, 4XX, 5XX), Endpoints (Login, Home, Profile, Logout), IP and TimeStamps. Response code is very essential in determining the failures in particular endpoints or analyzing overall errors/redirects in the system. Response time provides an accurate understanding of system bottlenecks and current performance of various endpoints in the system. There is a potential for demographic analysis using IPs and mining usage trends.

3. Architecture

3.1 Lambda Architecture

Our system is implemented as Lambda Architecture. It is data-processing architecture which works with batch and stream processing methods to handle massive quantities of data [3]. This architecture can be said to beat CAP theorem, and provide consistency, availability and partition-tolerance at the same time [4]. This approach attempts to balance throughput, latency and fault-tolerance by using batch processing to provide comprehensive and accurate views of batch data, while simultaneously using real-time stream processing to provide views of online data[13]. These two views (batch and speed) output is merged before visualization to provide accurate results. So, by combining the traditional approach of batch processing of logs with stream consumption tools we have achieved reliable near real-time log processing.

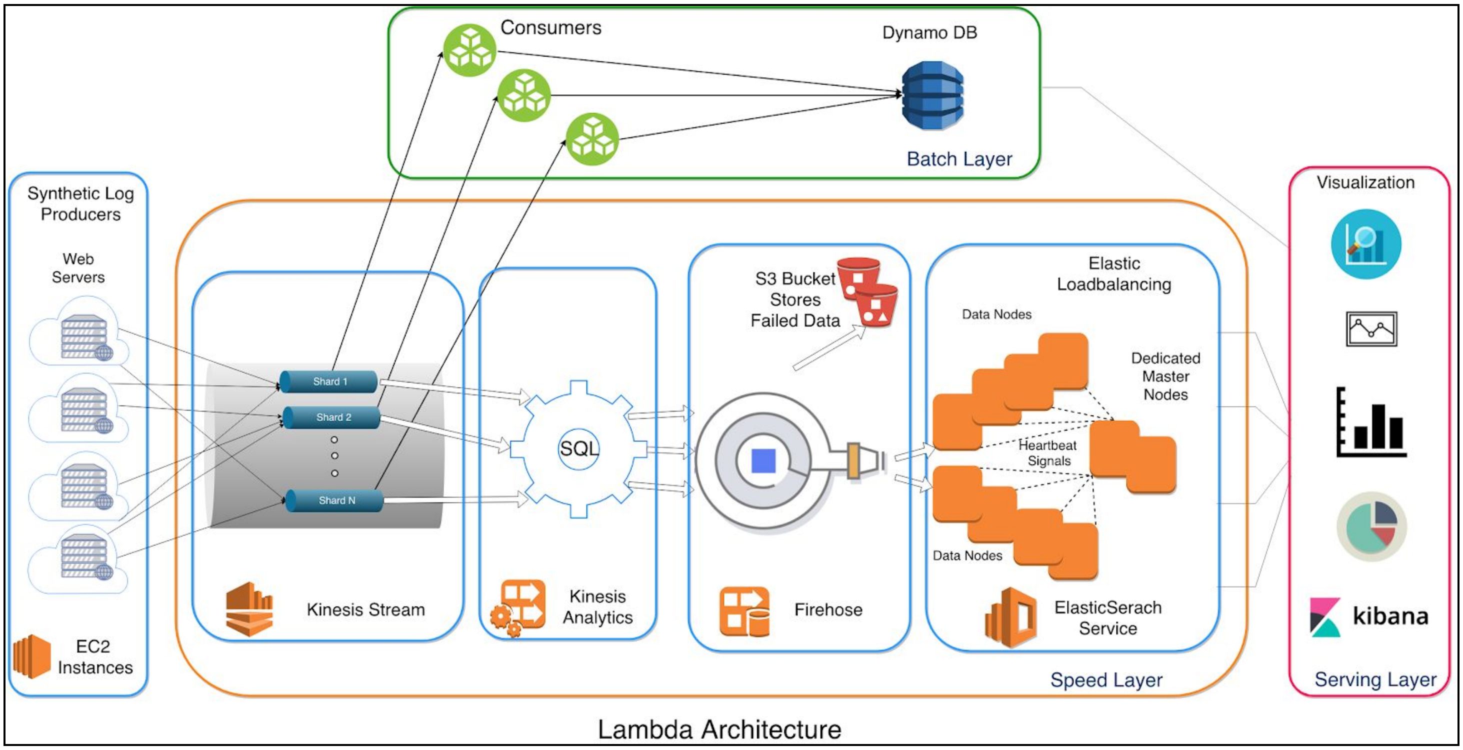

Fig 2: Architecture Diagram

In microservice architecture environment, loosely coupled services are deployed to different servers. The developers put various logs to analyze their run-time microservices and when they need to take an immediate action based on these logs.

Logs generated from semi-synthetic generators, which represent microservices, are pushed into various shards within the Kinesis streams. Consumer subscribes to a single shard and fetch the data to dump into DynamoDB, which further can be used for batch processing. For speed layer, the Kinesis analytics will perform SQL queries on the streaming data from shards and pass the processed streams to Firehose System. Firehose will reliably load this streaming data into data stores such as Elasticsearch and analytics tools [5]. Backup of logs is taken into S3 bucket, for which delivery fails. ElasticSearch domain will store the data in Elastic Block Storage with high availability, low-latency and replication policy from where we visualize data using Kibana dashboard.

3.2 Producers

As discussed above, the architecture consists of Kinesis Data Stream to stream the logs. It is based on pub-sub model. The publishers in Kinesis Streams are known as *producers*. Producers put records into the shards within data streams.

Producers in our system are semi-synthetic log generators, that simulate logging of a microservice. We use the term semi-synthetic since the records are generated from a base dataset obtained from Kaggle [6], but the generation process is configurable for the type and amount of logs. The generators are scripted in python3 and run on an AWS EC2 instance. To increase the velocity of log generations we run multiple processes on several EC2 instances. In one of our experiments, we generated 16k logs per second, running 10 processes on 8 EC2 instances windowing for 3 minutes, aggregating upto 3.2 million records.

The logs generated by these scripts consists of attributes mentioned in the overview, however, data stream is not strictly restricted to any data format and the visualizations can be tailored easily. From a user perspective, each microservice acts like a producer and have to dump the data in Kinesis Stream using AWS SDK (boto3 for python).

3.3 Batch Layer

Batch Layer is used to store historical data and perform batch processing over it to guarantee consistency. In our implementation, it consists of two main components, bunch of consumers and DynamoDB.

3.3.1 Consumers

Similar to producers, consumers are scripted in python3 and run on EC2 instances. Consumers, in general, get the data records from Kinesis stream and can perform any action on them. As per pub-sub model, consumers can get data from a particular shard within the stream. Hence, it is ideal to have as many consumers as number of shards to guarantee no data loss.

As seen in the architecture, each consumer loads the data it gets from a shard into DynamoDB, which acts as a data lake. A typical consumer can get upto 2 MB per sec. If there is latency in Kinesis stream, consumer will get all the buffered data from the stream guaranteeing consistency.

3.3.2 DynamoDB

Amazon DynamoDB is a NoSQL database service that provides fast and predictable performance with seamless scalability [15]. It provides high availability and durability. A table is a collection of items, and each item is a collection of attributes. In our architecture, DynamoDB acts like a data lake where the log data is store in batch layer. It can be very helpful to analyse the historical data and perform batch processing to gain deeper insights. We have made a single table in the database. Data from multiple consumers is loaded into that table as key-value pair. As DynamoDB serve high level of traffic, without hindering the properties mentioned above, it was ideal for us to use in our system. That allowed multiple consumers to load the data in the database table at same time.

3.4 Speed Layer

3.4.1 Kinesis Stream

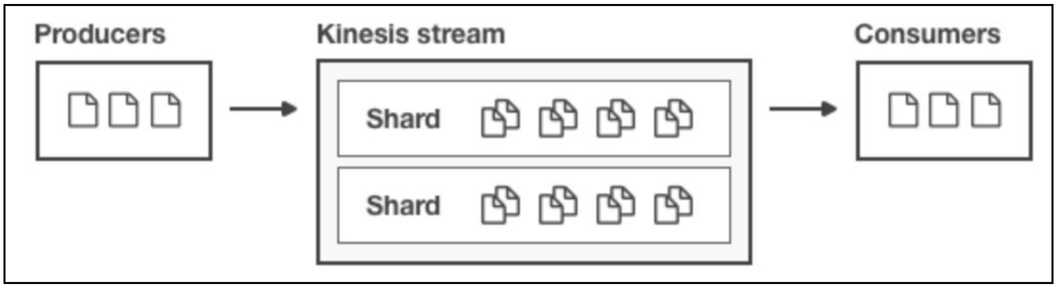

Fig 3: Kinesis Data Stream Overview [7]

As seen in figure 3, Kinesis data stream is a set of shards. Each shard has a sequence of data records. Data records are composed of a sequence number, a partition key, and a data blob, which is an immutable sequence of bytes [7]. Data blob can be upto 1 MB, which satisfies our requirement, as in this case each blob is a log, which is much smaller in size.

The stream's retention period is set to 24 hours, which means data can be accessed upto that period after being created in the stream, and it is configurable upto 7 days. So even if our consumers die or SQL app within Kinesis Analytics dies, developers have fair amount of time to resolve the issue, while the data will be retained in the stream. This make our architecture fault-tolerant.

A partition key is used to group data by shard within a stream. We have given endpoint name as our partition key. It means that logs for login page will go through the same shard and never from any other shard.

3.4.2 Kinesis Analytics

Kinesis Data Analytics allows users to process and analyze streaming data using standard SQL or Flink [8]. Since we needed basic aggregations and time-based filtering, we have used a familiar SQL based analytics. Kinesis analytics provide a wonderful feature of being a serverless service, eliminating the overhead of managing additional infrastructure. In the speed layer, Kinesis Analytics gets data from Kinesis Data Stream. It divides the incoming data into multiple streams as per SQL queries, needed for different analysis. These streams are then loaded into Kinesis Firehose Delivery Stream. It provides a timestamp column in each application stream, which we have used for time-based windowed queries. The application is a series of SQL statements that process input and produce output.

3.4.3 Firehose Delivery Stream

Firehose is used for delivering streaming log data to destination i.e Elasticsearch cluster. We have created a Firehose delivery stream that takes data from SQL analytics application and send it to Elasticsearch cluster, as seen in figure 4. Like Kinesis Data Stream, each data record in the firehose delivery stream can be upto 1 MB size, more than sufficient for current system needs of managing text based log records [9].

Firehose also buffers incoming streaming data to a certain size or for a certain period of time, whichever occurs first, before delivering it to destinations [9]. We have set Buffer Size to 1 MB and Buffer Interval to 60 seconds. In our experiments, the buffer interval was reached first and hence data was delivered every 60 seconds. This can be configured as per need and helps make the system near real-time. It can be easily observed that as the velocity of data increases, the volume increases and the buffer size would be the driving factor for delivery mechanism maintaining the system's near real-time behaviour.

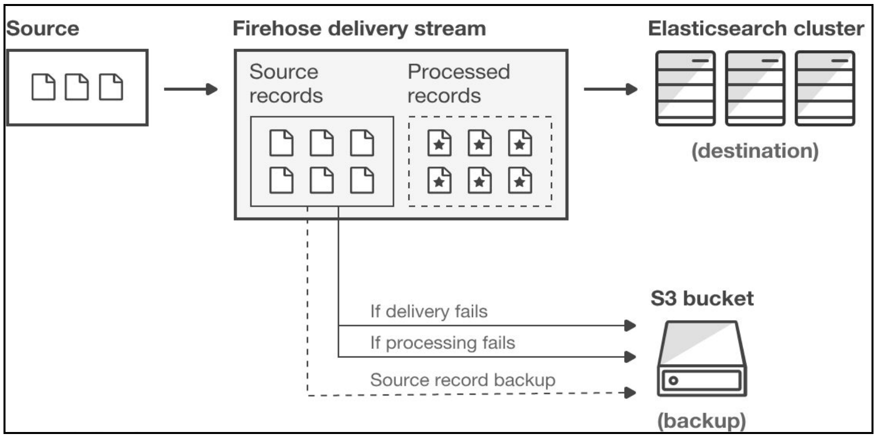

Fig 4: Firehose Delivery Stream [9]

We also backup the data for which delivery has failed to S3 bucket. This make our system fault-tolerant and no data is loss even if the destination cluster dies. We can configure the backup as per need to store every data record or only failed delivery record.

3.4.4 Elasticsearch Service

Elasticsearch is open-source search and analytics engine for use cases such as log analytics and real-time application monitoring and analysis. Amazon Elasticsearch Service is managed service which makes it easy to deploy, operate and scale Elasticsearch clusters [10][14].

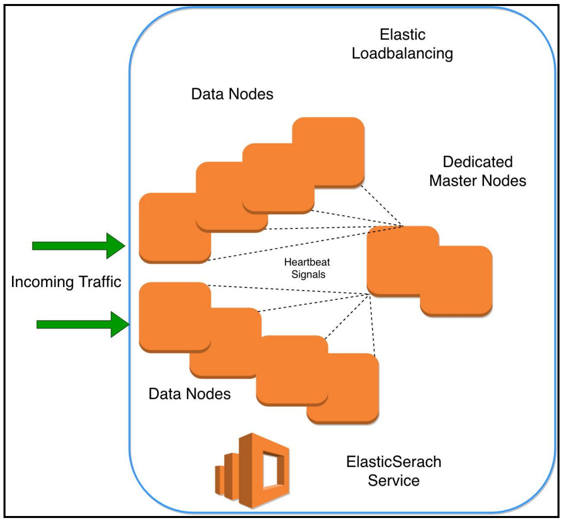

Fig 5: Elasticsearch Service

In an Elasticsearch cluster, we have configured distributed nodes of type Elasticsearch. The auto-scaling policy is set to scale out the traffic across nodes during high velocity of streaming log data. It also automatically detects and replaces failed Elasticsearch nodes. It has dedicated master nodes to offload cluster management tasks as seen in figure 5. It also provides automatic backup and restores Elasticsearch domains[10]. The cluster node allocation is done across two availability zones in the same region.

3.5 Serving Layer

Our serving layer is designed based on Kibana, where we perform analysis of various parameters primarily to observe response time and errors occurred, as a function of time and endpoints. Kibana is an open-source data visualization and exploration tool. It offers powerful and easy-to-use features such as histograms, line graphs, pie charts, heat maps, and built-in geospatial support [11]. We have integrated Kibana with Elasticsearch cluster. We have elaborated more on the types of visualization currently provided in Section 6.

5. Implementation

Using AWS Cloud infrastructure solutions, we have setup the above mentioned architecture. We are using AWS CloudFormation Templates in YAML to automate resource creation, mapping these resources for secured data flow between resources and provide auto-scaling policy.

First, we set up EC2 instances as producers of microservices' logs with configurable ingestion flow. Then, we spin the resources for Kinesis stream with configuring the optimal number of shards to put records from the producers based on key_id. After that, we spin up the resources for Kinesis Analytics, which includes SQL application, to analyze, transform and process streaming data. The output of the Kinesis Stream is configured as the input of *RawStreamData* for Kinesis analytics. Using Firehose, we build a reliable delivery stream by allocating resources. We setup the Elasticsearch cluster having 6 data nodes and 2 master nodes for dedicated cluster management tasks.

5. Visualization

We have provided four primary visualizations on the Kibana dashboard as evident in Figure [6]. However, Kibana is totally configurable and based on user requirements can be modified to present hundreds of different types of visualization based on incoming data from ElasticSearch service.

a. *Current Average Response Time*: We show a meter indicating current average response time of all endpoints. It is useful because it provides a single number metric of the system health. It is on bottom right of Figure 6.

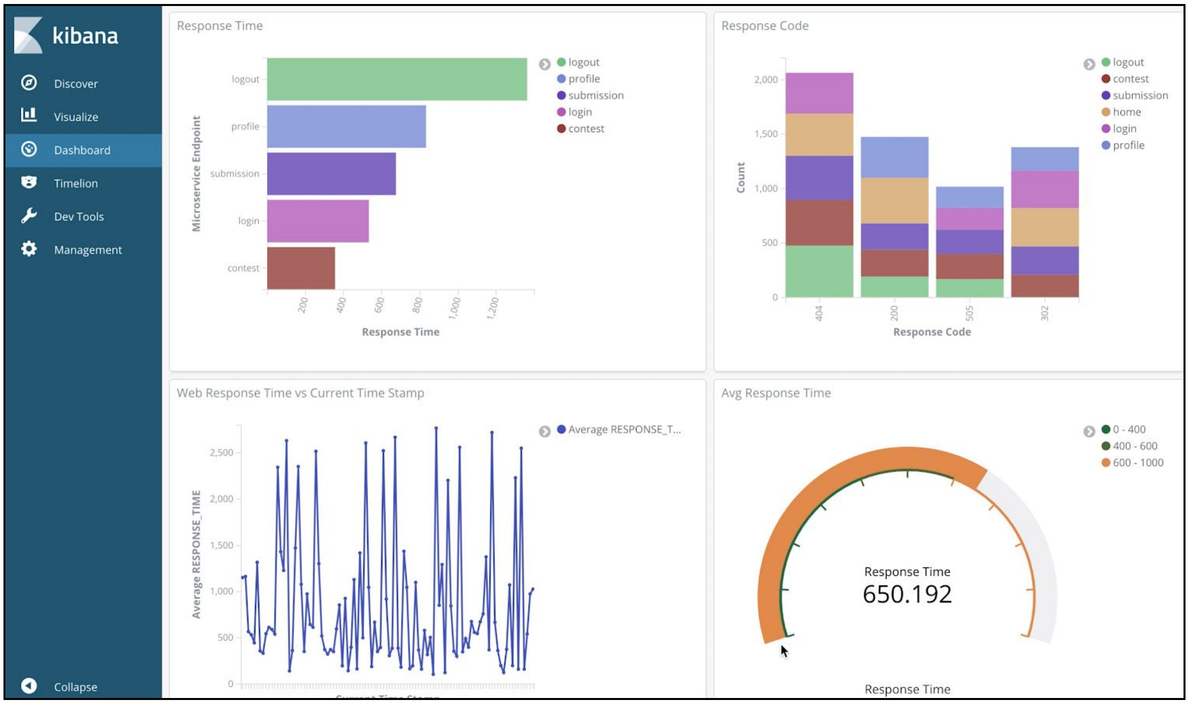

Fig 6: Visualization Dashboard

b. *Response Time per endpoint*: The horizontal bar graph is useful to analyze which endpoints are facing maximum traffic and can help detect possible bottlenecks of the system. It is on the top left of Figure 6.

c. *Number of particular error codes per endpoint*: Stacked bar chart is useful to know which endpoints are resulting in maximum errors and can be useful to debug those. It is on the top right of Figure 6.

d. *Historical Average Response Time:* The line graph can help monitor the system health as a function of time and can help identify events of high traffic or deployments. It is on bottom left of Figure 6.

6. Conclusion

In this paper we described a scalable architecture to handle near real-time log data processing and reasoning behind various design choices we made. The resulting system has many advantages. It provides high availability, scalability and fault tolerance. It is configurable at each level, from the semi-synthetic producers to flexible visualizations options on Kibana. We have tried to make the system as low cost as possible using optimal configurations of each AWS services. One can only imagine how costly it would be to build such a system using bare metal servers. Using CloudFormation, we have provided one-click deployment of the whole infrastructure and can be automated by integrating in existing CI/CD pipelines.

However, there are some limitations of the system as well. Using services in AWS Ecosystem comes with a minor disadvantage that, AWS Services play well with one another but not so much with other external systems. Hence, integrating with external third party services would not be as easy. There is also some replication involved since we store data both in batch layer (DynamoDB) and in Speed Layer (ElasticSearch). However it can be configured to store data in a particular time window. Also, with firehose delivery stream we are limited by the latency that it supports.

Overall, we can say that the system is able to ingest, process and analyze the log data at scale and in an efficient manner, fulfilling its primary aim, while also providing with plethora of additional features and configurable options.

References

[1] https://newrelic.com/
[2] https://aws.amazon.com/cloudwatch/
[3] https://en.wikipedia.org/wiki/Lambda_architecture
[4] http://nathanmarz.com/blog/how-to-beat-the-cap-theorem.html
[5] https://aws.amazon.com/kinesis/data-firehose/
[6] https://www.kaggle.com/shawon10/web-log-dataset
[7] https://docs.aws.amazon.com/streams/latest/dev/introduction.html
[8] https://docs.aws.amazon.com/kinesisanalytics/latest/dev/how-it-works.html
[9] https://docs.aws.amazon.com/firehose/latest/dev/what-is-this-service.html
[10] https://aws.amazon.com/elasticsearch-service/
[11] https://aws.amazon.com/elasticsearch-service/kibana/
[12] https://docs.aws.amazon.com/amazondynamodb/latest/developerguide/Introduction.html
[13] http://jameskinley.tumblr.com/post/37398560534/the-lambda-architecture-principles-for
[14] https://docs.aws.amazon.com/elasticsearch-service/latest/developerguide/what-is-amazon-elasticsearch-service.html
[15] https://docs.aws.amazon.com/amazondynamodb/latest/developerguide/Introduction.html

Appendix

A. Code hosted at - https://github.ncsu.edu/rshah9/LiveLogAnalytics.git