Data Lake Architecture -

A Comprehensive Design Document

Medical Data Processing Company

Data Architect: Mateus Leao

# Tracker

## Revision, Sign off Sheet and Key Contacts

## Change Record

|  |  |  |  |
| --- | --- | --- | --- |
| Date | Author | Version | Change Reference |
| 06/04/2020 | Mateus Leao | 0.1 | Initial draft |

## Reviewers / Approval

|  |  |  |  |
| --- | --- | --- | --- |
| Name | Version Approved | Position | Date |
| John Speed | 1.0 | Udacity Reviewer  Enterprise Data Lake Architect |  |

## Key Contacts

|  |  |  |  |
| --- | --- | --- | --- |
| Name | Role | Team | email |
| FirstName LastName | Data Architect | Medical Data Processing | student@email.com |

# Purpose

The purpose of this document is deliveryng a technical and business-oriented document that exposes an Enterprise Data Lake Architecture. It exposes the current data architecture that is not performing well anymore, and proposes a way to build a new data architecture solution that can handle the use case in a more robust and scalable way.

This document contains:

* Requirements
* Data Lake Architecture Design Principles
* Assumptions
* The Diagram of the Architecture proposed for Medical Data Processing Company
* Design considerations about Data Lake Layers
  + Ingestion Layer
  + Storage Layer
  + Processing Layer
  + Serving Layer

**Audience:**

The target of this document is anyone into the Medical Data Processing Company that is interested into this project. Overall, we aim to executives to read this document and support and push the proposal to happen.

**In Scope:**

Full Data Lake Architecture containing details about the Layers that were used to design the solution.

Proposed Tools and technologies to accomplish solution and a brief explanation on them.

Data Lake Diagram Architecture (Visual).

**Out of Scope:**

This document has the proposal and the architecture details and requirements, but it doesn’t have considerations about the implementation part of it, which would happen to be in a separate document and effort.

Handling JSONS is not a requirement on this project, the proposed solution would be able to handle it though.

It is out of scope for this project to have a transactional DB but it could be added

It is out of scope for this project to have a data warehouse but it could be added.

# Requirements

The Data Lake should be able to handle and ingest data from more than 8000 thousand machines in an efficient and reliable way. The solution should also be scalable in the sense of being able to handle increases of load.

One of the main requirements and purposes of this architecture is being able to provide real-time insights, metrics, and analytics, to over 8000 hospitals, clinics, and others that use this service.

The **Existing Technical Environment** is a bit complex: The requests from our users are sent through a load balancer that connects with our application monolithic layer. Regarding the connection with the Medical Facility, different mechanisms are being used to ingest Data from the medical facilities into the company environment.

Diagram

Description automatically generated

**Current Data Size:**

The company hosts over 8TB of Data nowadays.

**Business Requirements:**

* Improve uptime of overall system
* Reduce latency of SQL queries and reports
* System should be reliable and fault tolerant
* Architecture should scale as data volume and velocity increases
* Improve business agility and speed of innovation through automation and ability to experiment with new frameworks
* Embrace open source tools, avoid proprietary solutions which can lead to vendor lock-in
* Metadata driven design - a set of common scripts should be used to process different types of incoming data sets rather than building custom scripts to process each type of data source.

Centrally store all of the enterprise data and enable easy access

**Technical Requirements:**

* Ability to process incoming files on the fly (instead of nightly batch loads today)
* Separate the metadata, data and compute/processing layers
* Ability to keep unlimited historical data
* Ability to scale up processing speed with increase in data volume
* System should sustain small number of individual node failures without any downtime
* Ability to perform change data capture (CDC), UPSERT support on a certain number of tables
* Ability to drive multiple use cases from same dataset, without the need to move the data or extract the data
  + Ability to integrate with different ML frameworks such as TensorFlow
  + Ability to create dashboards using tools such as PowerBI, Tableau, or Microstrategy
  + Generate daily, weekly, nightly reports using scripts or SQL
* Ad-hoc data analytics, interactive querying capability using SQL

# Data Lake Architecture design principles

The Data Lake should be able to ingest log data (kafka), handle change capture and updates (apache hudi and spark), and also be able to handle real-time dashboards and data transformations (spark, web app. layer).

Our architecture was designed with a 4-Layer Architecture. We have the Ingestion Layer, where we plan to have a Kafka Cluster that is going to persist the incoming data for 7 days, and that will be sending Data to our storage (S3 or HDFS) with the help of Apache Hudi.

In the Storage Layer we could have HDFS or S3 (S3 was decided for this project), we have a staging area that is going to receive the raw data from Kafka. Storage Layer and Processing Layer work together and is only with the help of Spark that we can write data from our Staging Area to a more consolidated data storage area, in this area we’ll have our Derived Tables, these tables have the Data in a format that works for the Business and for Analytics Serving according to our query patterns and needs.

The Data will be available in .parquet and .avro formats.

# Assumptions

**Assumptions:**

* S3 has higher availability than HDFS, it will also be far less expensive than storing the data in disk. S3 is a viable solution for storage.
* HDFS won’t handle small files well.
* The Apache Cluster will work without a single point of failure.
* Linux O.S is our best option for the Spark/HDFS Cluster.
* The AWS Kafka Cluster is more performant than a self-managed cluster.

**Risks:**

* Since we have decided to use S3 then we have to monitor how s3 is handling the high load of writes and if it’s working as expected without errors.
* If Kafka has a single point of failure then we risk to lose the data that is being sent to Kafka.
* If we use HDFS knowing that it’s block file is 64 mbs and since we know that HDFS doesn’t handle well lots of small files, then we risk having a slow system that can’t handle of loading lots of small files containing data.

# Data Lake Architecture for Medical Data Processing Company

Diagram

Description automatically generated

# Design Considerations and Rationale

Additional Tools that are out of scope (more details below): HDFS, Hive, Pig, MapReduce.

## Ingestion Layer

We will have applications and databases working with the Kafka Connect API. We will use Kafka Sources so we can ingest the data coming from most of our Facilities and equipment into **Kafka Topics.** This is valid for data coming from databases and APIS. For the data coming from the FTP Servers we´ll be sending this data directly to the S3 Storage, with the help of Apache Hudi + Apache Spark.

The main tools in our architecture are:

**Apache Kafka:**

Works as a distributed queuing system capable of handling huges amounts of data for both actions of receiving data (sources) or sending data (sinks)

**Apache Hudi**:

Hudi helps us manage the metadata of our tables. Apache hudi works with Apache Kafka and gives us a way to incrementally add data to our tables.

**HDFS or S3 Storage:**

The core of our data lake, where our data will be stored. HDFS and S3 are scalable distributed storage systems where our data could live in. We decided to use S3.

We will have two data environments, one where data will be stored as it comes (raw), and another where data will be stored after being processed, cleansed, or aggregated with the help of Apache Spark.

**Apache Spark:**

Another core part of our architecture. Spark is responsible for the processing of data within our Data Lake. Scalable, works as a Cluster of Resources, we decided to use AWS EMR in this case.

Spark is a distributed processing tool capable of handling huges amount of data in a very efficient manner. Spark is responsible for sending data from staging to our derived data environment, with Spark it´s possible to do many types of transformations: Joins, Deduplication, Aggregations, Filter data, etc.

**Scalability Considerations:**

Due to the distributed nature of the applications we are going to use, this architecture will be able to easily scale so it can handle larger and larges amount of data. If we are using AWS EMR we can increase our cluster machines (vertically), or increase our cluster horizontally (more machines). The same is valid for the Kafka Cluster.

We should have monitoring and usage metrics in place so we can receive alerts when it´s time to increase the size of our clusters.

**Ingestion tools that won´t be used:**

Apache Hive won´t be necessary for this Architecture. HDFS won´t be necessary either, and Apache Cassandra is out of this project.

## Storage Layer

Here you find considerations about using HDFS and using S3, we opted for using S3.

**Apache Hive** won´t be necessary for this Architecture since we´ll be working with Hudi + Spark in S3 to build tables.

**Considerations about HDFS vs S3:**

There is the possibility in this Architecture of using HDFS (Hadoop File System) or S3 for the storage of our data. S3 should be able to handle small files better than HDFS. S3 is much cheaper than HDFS, additional tests have to be done to make sure that S3 can handle the workload.

**Using HDFS:**

HDFS stands for Hadoop File System. With HDFS Data is stored in file formats (CSV, Parquet, Avro, JSON, etc). HDFS integrates well with other systems from our Architecture.

HDFS will receive data from Apache Kafka, from the FTP, and also from Apache Spark Jobs. All data will be stored in a file system in a Hadoop cluster that can be scaled both in Disk as in Processing or Memory.

**How would HDFS handle 20% YoY Data Growth?**

The disk of the cluster has a fixed size, every working node can have a disk associated to it, the more nodes we have the more disks we can have. These disks form a pool of resources that HDFS can manage. In case the disks are getting filled then it´s possible to simply increase the disk size of every node, or increase the number of nodes and disks.

**OR using S3:**

S3 would work similarly to HDFS, it integrates well with Kafka, Apache Hudi, and Spark.

S3 is much cheaper than HDFS. Besides that, with S3 we can decouple our Spark Processing from our Storage, which is something nice to have.

**How would S3 handle 20% YoY Data Growth?**

S3 is a service fully managed by AWS that increases infinitely (almost). No configurations have to be done, getting more points to S3. No one would have to worry about scaling the size of the S3 buckets because they’re unlimited.

**Backup and Recovery Considerations and Strategy:**

For recoveries we can count with the data that Kafka sends to MongoDB, so if something fails when sending data from Kafka to our staging area, then we could just reprocess and get the data that got errored out. But if something more serious happen and we lose the data from our staging area in S3, then we could go to Mongo, that would be persisting our Data, and get the missing data from there.

If, let´s say, a Spark Job fails when sending data from Staging to our Derived tables, there are two things we could do, one is allowing our pipeline to retry the job, so if the problem is intermittent or just a failure in the S3/Kafka Cluster, retrying could solve or problem.

Data is being persisted in Mongo as a Backup Source. The Mongo Cluster won’t be accessible externally, and will be a single source of truth for our Data, allowing data recoveries to be made through the API Gateway that will connect to Mongo.

**Considerations about Metadata:**

Apache Hudi handles the metadata for us. We should have metadata on the staging raw area with all of the different datasets we have, and also in the derived area, where we should have metadata about each of our datasets and about our processed data model. The metadata would have information on the tables, the fields of the tables, the quantity of data per table, the relation between tables, and others.

**Data Formats:**

The data in our Warehouse will be in the Parquet and in the Avro Format. For historical querying the data will be in Parquet format only, and for incremental querying the data will be available in the Avro format. Apache Spark can work with both file types.

**Considerations about data security:**

Our mongodb won’t be accessible from the web.

Data will be encrypted at rest and in transit, AWS helps with that.

Users need to authenticate with a token to access the data.

## Processing Layer

We have choosed to work with Apache Spark instead of working with Pig or MapReduce. We decided that because spark is much faster.

The processing Layer is responsible for working with the data coming from our staging environment. Spark is our main player here; it works as a central processing engine capable of processing and doing multiple types of transformations with our Data. After doing the needed transformations, merges, joins, and others, the Data is stored in our Derived Tables, the Derived tables can have aggregated data, historical data, or simply data that has been processed and is in a more analytics-ready format.

The processing layer applies business rules to the Data, together with Quality operations that give more value to our Data.

**Batch vs Realtime vs CDC:**

Spark and Airflow are responsible for the Batch processing of our Data, with Airflow we can have CRON trigger that run spark jobs to process and generate data according to our needs.

Realtime Data can be queried from our raw and derived tables, when data gets into staging we have streaming spark applications that transform the data and store it in our Derived environment.

CDC: Apache Hudi is responsible for managing the changes of Data, we are able to upsert (updates) data and handle incremental querying and data processing with Spark and Hudi.

**Enabling Ad-hoc queries:**

One of the easiest solutions for Ad-Hoc queries is using AWS Athena to query the tables available in our Data Lake.

Another one is using apache spark (notebooks for example) – the spark applications can read the files in both of our environments and do queries using SQL or using the DataFrame API.

**Scalability Considerations:**

We can use auto-scaling configurations within our EMP AWS Cluster, allowing us to easily handle jobs that require more processing power. After configuring auto-scaling in the Spark Cluster there´s nothing more to be done, AWS handles everything. We can scale our cluster both vertically and horizontally.

We should have a metric that shows our Memory usage since Apache Spark uses memory to cache our Data.

**Different tools involved in the Processing Layer:**

The main Processing Engine we use is Apache Spark.

## Serving Layer

**Overview:**

The serving layer is the interface between the Data Lake and the users, it is through this layer that users can have access to real-time dashboards & insights, and through this layer that they can query the database and retrieve information about the data in the Data Lake.

**Types of Data Available in the serving layer:**

Both the raw and the consolidated (derived tables) data are available in the serving layer. Raw data can be accessed through the API Gateway that gets data from mongo.

**Uses of the Serving Layer in respect to Data:**

It is thanks to the serving layer that the users can access the data in the lake and the dashboards and insights.

AWS Athena allows S3 data to be queried.

**Load Balancer and Real Time Dashboards**

The company will have real time dashboards deployed and available to their users to access it’s information. The dashboards read directly from the Data Lake.

# 8. Conclusion

This project was made to provide a robust architecture that is able to ingest and process and make available huge amounts of data in real time (PB Scale) with a very high throughput, additionally to that we have proposed systems that are able to scale up to petabytes of data, making this architecture sustainable in the long run.

# 9. References

<https://www.xenonstack.com/insights/what-is-hudi/>

<https://hadoop.apache.org/docs/r1.2.1/hdfs_design.html>