Data Lake Architecture -

A Comprehensive Design Document

Medical Data Processing Company

# Tracker

## Revision, Sign off Sheet and Key Contacts

## Change Record

|  |  |  |  |
| --- | --- | --- | --- |
| Date | Author | Version | Change Reference |
| 10/15/2023 | Maleina Bidek | 0.1 | Initial draft |

## Reviewers / Approval

|  |  |  |  |
| --- | --- | --- | --- |
| Name | Version Approved | Position | Date |
| FirstName LastName | 1.0 | Udacity Reviewer  Enterprise Data Lake Architect |  |

## Key Contacts

|  |  |  |  |
| --- | --- | --- | --- |
| Name | Role | Team | email |
| Maleina Bidek | Data Architect | Medical Data Processing | student@email.com |

# Purpose

This document describes the data lake solution architecture for Medical Data Processing Company so that it can serve as a basis for evaluation, as well as more detailed design and development. It contains a summary of the requirements, principles, and assumptions, as well as the actual design with descriptions of the components and rationale. The target audience of this document includes those with a high degree of technical knowledge such as enterprise architects, software engineers, and technical directors. Therefore, it does not include general information concerning data lakes or pipelines, nor does it include the current system design or background information regarding the Medical Data Processing Company.

# Requirements

Medical Data Processing’s current technical environment is having real performance problems due to the company’s rapid growth over the last few years. Its current SQL Server based solution can no longer scale with the large volumes of data that it must process and as a result, its pipeline and query performance are slow and subject to crashes with downtimes. Analytics and reporting processing are complex, making it difficult to consider adding additional services such as real-time dashboards or machine learning. Additionally, the current system lacks critical capabilities such as back-up and recovery. The company has exhausted all options in terms of upgrades and optimizations and therefore must develop a new solution.

The current technical environment consists of a single master SQL Server database and one staging database. Additionally, there are three other data ingestion servers for FTP, API and other data extraction agents, as well as a number of other web and application servers. An average of 77,000 zip files are currently ingested from over 8000 medical facilities per day, with a size range from 20 KB up to 40 MB. These files further consist of CSV, TXL or XML files, resulting in an average of 15 million total files that need to be processed daily (an average of 700,000 per hour). The current data volume growth rate is 15 to 20% year over year (YoY).

The business requirements for the new solution, as specified in the company profile and problem statement document include:

* Improving the uptime of overall system,
* Reducing the latency of SQL queries and reports,
* Ensuring that the system is reliable and fault tolerant,
* An architecture that scales as data volume and velocity increases,
* Improving business agility and speed of innovation through automation and ability to experiment with new frameworks,
* Embracing open-source tools and avoiding proprietary solutions which can lead to vendor lock-in,
* A metadata driven design - a set of common scripts processes different types of incoming data sets rather than building custom scripts to process each type of data source,
* Centrally storing the enterprise data and enabling easy access to it.

The technical requirements, as specified in the company profile and problem statement document include:

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

# Data Lake Architecture design principles

When developing our proposed architecture, we based our decisions upon the following principles:

1. *Build to Change Instead of Building to Last* – This time-tested principle means that it is important to consider how the solution will need to change to address new requirements and challenges, so that it can be built with the appropriate flexibility. Given the problems that resulted from the rapid growth that occurred in the last few years, it is critical that the solution be able to react to changes before they result in slower performance and downtime and further shake customer confidence.
2. *Leverage existing products and platforms to reduce development time* – Because the current system is in distress, we cannot afford to spend years developing a solution. It would be best to choose solution components that are relatively easy to develop and integrate, even if this means sacrificing a bit on our commitment to avoid vendor lock-in.
3. *Scalability and Performance* – The ability of the solution to scale up processing speed with increases in volume is one of the key technical requirements.
4. *Resilience and Fault Tolerance* - Ensuring that the system is reliable and fault tolerant is one of the key business requirements.
5. *Minimize the use of proprietary solutions* - When considering between comparable products, we should choose open-source or source-available products when possible. This is specifically mentioned in the business requirements.

# Assumptions and Risks

When designing the solution, we made the following assumptions:

* + Given that our data sources are currently document based, it would be preferable to choose solution components, such as NoSQL databases and cloud object storage, that are well suited for these formats.
  + We will want to retrieve most of our data via real-time streaming in the future and thus we will encourage our business partners to start moving away from sending documents via FTP.
  + It will not take much more effort to ingest data retrieved from data streaming (from databases or APIs) directly into a NoSQL database. This effort we be more beneficial in the long run than placing the data in object storage and it will facilitate change data capture.
  + Documents retrieved (or uploaded) via FTP can be placed in cloud object storage and stay their indefinitely.
  + To increase performance and reduce costs, we may at some point want to archive older database data into cloud object storage.
  + We need a solution that allows us to retain and use the data that we already have. The ability to convert or import data from our existing SQL Server database is important.
  + The solution needs to run in the cloud to maximize availability and scalability.
  + There is less risk to our development timeline in using components from a single vendor, such as MongoDB, in both the storage and processing layers rather than trying to integrate components from different vendors or foundations.
  + While not explicitly mentioned in the following sections, we can use Apache Airflow to provide scheduling capabilities across the various layers of our solution.

Some potential risks of our design include:

* + The team may need extra time to learn and adapt to working in a cloud environment and with NoSQL databases.
  + We propose the use of MongoDB’s Atlas platform and its date lake. These are relatively recent product offerings compared to, for example, AWS’s solutions. This means that it may be a little more difficult to find resources who have experience with these technologies and thus, it may be more difficult to troubleshoot issues in-house.

# Data Lake Architecture for Medical Data Processing Company

# A screenshot of a diagram Description automatically generated

# Design Considerations and Rationale

As noted earlier in our assumptions, we have decided to host the application in the cloud. As we will see in subsequent sections, we chose the MongoDB Atlas platform as the basis for our solution. Atlas has the option of running on either Amazon Web Services (AWS), Microsoft Azure, or Google Cloud Platform (GCP). Due to its familiarity and its support of third-party and open-source tools, we chose AWS. However, as you will see, our solution components are mainly open-source and are not specific to any cloud vendor, so that we still have the option of deploying our solution in total or in part to another cloud provider in the future, thus avoiding vendor lock-in and maintaining our principle of flexibility.

## Ingestion Layer

We will use Kafka for real-time continuous data streaming from medical facility databases and application APIs. We chose Kafka because it is a widely used, open-source and proven platform that has ready-made connectors that can be used to integrate various data sources. It can be more easily and reliably scaled as more and more data sources are added. Kafka also has a connector for MongoDB, where we will store the data that enters via streaming. The source data will be published into Kafka ‘topics’ that queue and store the data for a limited time until it is read and ingested into Mongo DB. Note that we will need to develop a standard JSON data format and encourage (read: require) its use by medical facilities so that we can easily import the streamed data into MongoDB.

As mentioned above, we would like to encourage facilities to use continuous data streaming. However, we will still accommodate those that still need to use FTP to deliver files. We will use Amazon S3 to house the documents in object storage, which can accommodate the high volume of data that we are ingesting and scale as needed. AWS offers a secure FTP capability to can be used to load documents directly into S3.

Because we will be implementing a data lake, these files can stay indefinitely in their raw format. However, note that in this case we will not be able to offer change data capture. That is the key advantage of storing data in a database such as MongoDB (which does have an upsert capability).

Other ingestion tools considered for streaming include Apache Flume, which was ruled out because it is primarily used for log data which we are not ingesting, Sqoop for RDMS data, which was ruled out since it has subsequently been retired, and Amazon Kinesis, which we ruled out because it is proprietary and can only be run on AWS.

## Storage Layer

As mentioned above we will use both Amazon S3 (object storage for uploaded files) and MongoDB (for data that arrives via streaming) as the first level for our storage layer. In some models this could be considered as part of the ingestion layer, but because it involves actual storage, we include these components here as part of the storage layer.

We chose MongoDB because it is a source-available, document-based database which is well suited to our data, which has traditionally been document-based. It can store the vast amount of data that we require to handle our 20% YoY data growth rate.

Another reason for choosing MongoDB is that it has the tools needed to convert and load data from our existing SQL Server relational database. As mentioned above, a key advantage of using a NoSQL database to store the data that enters from streaming is that we can perform change data capture on that data, which is one of our technical requirements. Finally, as a NoSQL solution, MongoDB is extremely scalable. However, if we do ever find that performance starts to lessen or if we want to reduce the costs of querying vast amounts of data, we can develop an archiving process to move older data into S3 object storage.

We did consider Apache Cassandra, which is another widely used and highly scalable NoSQL database, however we felt that MongoDB’s Atlas platform provided a wider range of tools, such as a data lake and data federation that could be leveraged to develop our system more quickly.

Because we are storing both semi-structured and raw data in different sources, we will need to use a data lake to facilitate the use of our data in queries and by other applications. We have chosen MongoDB’s Atlas Data Lake as our solution. We did evaluate the AWS data lake products (AWS Glue, AWS Lake Formation), which are compatible with MongoDB’s NoSQL database, however we gave preference to Atlas because it should be easier to integrate as a single solution. Additionally, as mentioned above, Atlas can be run on any of the big 3 cloud providers (AWS, Azure, GCP), thus further reducing vendor lock-in. If Atlas’ tools prove not to be robust enough, we still have the flexibility to swap out these components in the future.

When data is ingested into the Atlas data lake, it is enriched with metadata to optimize lake performance. This metadata, which includes information about the name, type, and size of the various data fields, is then stored in its own MongoDB database cluster, separate from the other data, thus meeting the requirement to separate metadata, data, and processing. The data itself is stored in a flat, non-hierarchical format as objects with metadata that users can access later. For now, for simplicity’s sake, we will use MongoDB’s native analytical format to store the data. However, we do have the option to revisit this decision in the future if necessary.

Atlas offers both backups and replication. Snapshots can be scheduled and taken at various intervals and used subsequently for recovery. Replication will ensure that we meet our requirement to ensure that if a small number of nodes fail, that our solution will still be available. Additionally, MongoDB guarantees distributed fault tolerance, ensuring that we meet our goal of our system being both reliable and fault tolerant.

To ensure that our data is secure, we will be implementing role-based access control (RBAC) and issue credentials per user (as opposed to account sharing by multiple users). Additionally, Atlas offers both encryption at rest and encryption in transit, which we can enable given the sensitivity of medical data. Finally, we take advantage of Atlas’s ability to whitelist certain IP addresses to further limit access to the data.

## Processing Layer

We will use MongoDB’s Data Federation, as well as Spark (in a limited context) to implement the processing layer.

MongoDB’s Data Federation provides the ability to create federated queries across data in the both the MongoDB cluster and the data in the S3 buckets. Under the hood, Data Federation customizes MapReduce and deploys multiple compute nodes to process queries, working in parallel in regions as close to the data as possible, thus minimizing data transfer. As we scale, this will be very important to ensure that our performance is not degraded.

Another choice we considered was solely using Apache Spark for the entire processing layer. However, we chose to use MongoDB’s Data Federation given the ease of its set-up on top of the Atlas Data Lake, and the fact that it provides a plug-in for ad-hoc SQL query capabilities, as well as for easier integration of Tableau and Power BI.

However, there is also a role for Spark in enabling AI capabilities. We will use MongoDB’s Spark Connector to connect to Spark so that we can enable machine learning or AI application development using the most popular libraries (Tensor Flow, Pytorch, SciKitLearn). Spark can use the aggregation and indexes already created by MongoDB so that it does not extract the data again and it can minimize data movement and processes only the data that is needed, reducing latency.

Lastly, we did consider using MapReduce to implement the processing layer, but due to its complexity, and the fact that both Spark and MongoDB data federation can more easily enable the services required in the serving layer, it was quickly ruled out.

## Serving Layer

The serving layer is the part of our solution that enables the presentation of the data to end users via the use of tools such as dashboards and analytics software or via ad-hoc queries. It does not store data, per se, other than what may be cached by a specific application.

As mentioned above, the Atlas SQL Interface enables the ability to perform ad-hoc queries, as well as generate daily, weekly, or nightly reports using SQL scripts. Real-time dashboards will be developed in Tableau and/or Power BI, which can also use the Atlas SQL Interface to retrieve data. Finally, we are able to integrate with different ML frameworks such as TensorFlow via Spark and present the results in reports or applications.

# 8. Conclusion

This document presents a data lake solution architecture that will meet the needs of Medical Data Processing Company to scale to the vast amounts of data that it needs to process daily. We recommend as next steps that it be the basis for a more detailed design or in a proof of concept before its actual implementation.

# 9. References

<https://www.mongodb.com/>

<https://aws.amazon.com/big-data/datalakes-and-analytics>

<https://www.tutorialspoint.com/software_architecture_design/key_principles.htm>

<https://thisiszone.medium.com/principle-three-picking-the-right-tool-for-the-job-d45ede555d42>