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 |
| 31/07/2021 | Elie Kawerk | 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 |
| Elie Kawerk | Data Architect | Medical Data Processing | eliekawerk@hotmail.com |

# Purpose

This document aims at presenting a comprehensive data lake design architecture proposal for the Medical Data Processing Company (MDPC).

Given that the company has recently faced several issues while processing data with its existing monolithic data architecture, MDPC’s CTO - who is looking for an alternative architecture - has asked us to produce this document.

The document details the following:

* Technical & business requirements
* The underlying principles of the data lake design
* The assumptions made while designing the lake
* An architectural diagram depicting the main elements of the data lake
* The design considerations and rationale
* A conclusion that highlights how a data lake can solve MDPC’s existing data-related inefficiencies

The document is meant to be examined by technical data professionals of the MDPC (database administrators, developers, data engineers/scientists).

It is worth noting the following points:

* We do not go through minute technical requirements or include template code for each technology that we use to develop the data lake. In contrast, we only articulate the high-level decisions made based on the provided requirements.
* We do not discuss thoroughly the details of data lake governance. Topics like metadata management system and data quality assessment and monitoring tools are not addressed.
* We do not discuss how to face extreme events such as disasters (disaster recovery) or data breaches.

# Requirements

In summary, MDPC needs a flexible data architecture that adapts to its growth and reduces the inefficiencies faced with the current monolithic architecture they have on premises.

At a high level, we can summarize the requirements as follows:

* Real-time ingestion and processing of incoming data
* High level of fault tolerance
* High level of processing scalability
* The ability to store all historical data without size limits
* Make data available from a single source without the need of data replication
* An interface to interact with cleansed data through the SQL syntax

MDPC’s existing technical architecture is shown in figure 1. In this monolithic architecture, medical data is ingested using either FTP, a file extract agent, or an API agent into a staging area that uses SQL server. Every night, ETL jobs (proprietary software) to cleanse the data and push it into a Warehouse that also uses SQL server. Users can access data in the Warehouse through either the application layer (using stored procedures and indexes for faster lookups) or through dashboards and reports that read from files that extract data from the Warehouse nightly.

Diagram

Description automatically generated

Figure - Medical Data Processing Company's existing monolithic data architecture

The ingested data lands from ~8K different medical facilities where 99% of the ingested files originate from encrypted zip files (in the CSV, TXT, or XML formats) whose sizes typically range from 20KB to 1.5MB (though some files can be as large as 40MB). XML zip files contain typically 20 to 300 individual XML files which consist of 1 record only. The daily uploaded number of zip files is ~77K (~15M files) which corresponds to ~3.5K zip files/hour (700K files/hour). The data growth rate is projected to be ~15%-20% YoY.

The business requirements can be summarized as follows:

* Continuous uptime (improve on the existing system)
* Reduced latency of SQL queries
* Reliability and fault tolerance
* The system should scale with bigger and faster data
* Automation & capability to use new frameworks
* Usage of open-source tools whenever possible to avoid vendor lock-in
* Centralization of the enterprise data assets with ease of access

The technical requirements can be summarized as follows:

* Real-time ingestion and processing of data (in contrast to the current system)
* Loose coupling of the storage (data, metadata) and the processing layers
* Infinite storage scalability
* Ability to increase compute resources for processing as data volumes grow
* Fault-tolerance: the system should have the ability to survive node failures without downtime
* The ability to read data for modeling or reporting without extracting it
* Interactive querying for ad-hoc analysis using the SQL syntax

All the requirements reported above were inferred from the company profile & problem statement document provided in the project’s resources.

# Data Lake Architecture design principles

To design the data lake, we rely on the following principles:

* Centralized storage and processing: All incoming enterprise data (structured, semi-structured, and unstructured) are stored and processed in a single central location. This is aligned with one of the key business requirements collected from MDPC’s leadership.
* Unlimited scale: The data lake should have the capability to store data at an unlimited scale. This is aligned with:
  + a technical requirement (unlimited storage of historical data) and
  + a business requirement (architecture should scale as data volume and data velocity increase)
* Patterns: A data lake democratizes the access to data and allows performing analytics and machine learning use cases much more conveniently. This is in line with core technical requirements (easy access to interactive SQL queries, integration of ML frameworks, ...)
* Schema on read: As data flows into the lake, no schema is enforced to store the data which is stored as is. The schema of the data is inferred when data is read from the lake. This is aligned with the business requirement stating that the design of the lake should be guided by the principle of metadata driven design.

# Assumptions

To design the data lake, we made the following assumptions:

* Cloud native data lake: We only use cloud computing (AWS services for ingestion, computing, storage, and serving) to store and process the data, compute for any analytics/ML use case, and serve it through reports, dashboards, or APIs. We assume that we do not have to deal any on-premises storage or computing. We will only include relevant AWS services or open-source tools whenever possible.
* Data migration: we assume that we will be assisted by a professional team to migrate the historical data from the existing monolithic architecture to the new data lake. The design document includes a batch layer that makes such a migration convenient.
* Integration with the application layers and the existing reporting/visualization/analytics channels: we assume that there are dedicated MDPC resources that will replace the integration of the old architecture with new integrations with the new data lake architecture. In our design document, we do not delve into the details on how the integration should take place.
* We use the Linux operating system for configuring any applications or databases on the AWS cloud whenever required.

These assumptions are accompanied with the following risks:

* When using a particular cloud vendor, we assume that MDPC has a team of qualified employees who can manage the data lake and ensure proper operations. In case of an AWS data lake, we assume that MDPC has data professionals who are proficient with AWS services.
* When migrating the data from the old architecture to the cloud data lake, a team must be dedicated for this task. It must consider aspects such as:
  + Ease of migration from SQL Server to the data lake (data formats)
  + Ensure data is secure while on transit and when it lands to the storage layer
  + Ensure that the system stays available (zero downtime) for users during migration
* When the data lake is set-up and the data is migrated into it, MDPC’s developers must make sure that the integrations of MDPC’s apps and reporting layers are properly handled while ensuring a smooth transition from the older system.
* We assume that the MDPC personnel is proficient with debugging, maintaining and configuring Linux machines on the cloud.

# Data Lake Architecture for Medical Data Processing Company

Chart

Description automatically generated with medium confidence

Figure - Diagram of the recommended data lake architecture for MDPC

# Design Considerations and Rationale

The data lake consists of an ingestion layer, a storage layer, a processing layer, and a serving layer. In figure 2, we show these 4 layers along with the components that constitute them.

## Ingestion Layer

As shown in the left side of figure 2, the ingestion layer consists of 2 components:

* A streaming data component that uses AWS Kinesis (Data Streams and Kinesis Firehose), a managed data streaming service that can automatically scale in the AWS cloud. Kinesis Firehose can handle the ingestion of real-time data from structured, unstructured, and semi-structured sources. Such sources include APIs (3rd-party, internal), Enterprise Resource Planning solutions (ERPs), and even FTP servers (a maneuver must be done using lambda functions for this [2]). Kinesis is a serverless service meaning that it scales to match the throughput of the medical data and requires no administration (by managing EC2 instances that would host a similar solution for instance). Kinesis can batch, compress, and secure the data before sending it to the storage layer.
* A data migration layer that uses the AWS Data Migration Service (DMS). DMS is a solution that allows migrating on-premises or cloud databases into the AWS cloud reliably, safely and with minimum downtime. This service can be used to migrate the existing data in the monolithic architecture into the storage layer (either S3 or the Redshift Data Warehouse as we will discuss in the next subsection). According to the documentation [3]:

“With AWS Database Migration Service, you can continuously replicate your data with high availability and consolidate databases into a petabyte-scale data warehouse by streaming data to Amazon Redshift and Amazon S3”.

Instead of using AWS Kinesis and AWS DMS, we could have used tools like Apache Kafka, to handle real-time streaming, Apache Nifi or Apache Sqoop to handle data flow and database migration respectively from the on-premises data warehouse into the Cloud. Although these tools are open-source and vendor-agnostic, we preferred using the managed AWS services as they are fully managed services that scale with the size of the data automatically without the overhead of managing servers. In addition, these AWS ingestion services integrate much better with the AWS Cloud.

## Storage Layer

As shown in figure 2, the storage layer consists of 3 components:

* A data lake that has 3 layers:
  + A set of S3 buckets for storing the landing data from the ingestion layer
  + A set of S3 glacier buckets that contain a replicate of the data contained in the set of landing S3 buckets
  + A set of S3 buckets for storing the transformed data coming from the set of landing S3 buckets
* A data Warehouse that uses Amazon Redshift

According to AWS docs [4], S3 (Simple Storage Service) “is an AWS storage service that can scale to meet elastic storage needs with 99.999999999% of data durability without any upfront investments to provision or maintain hardware. S3 protects the data with encryption protocols and can manage data at any scale with robust access control, and flexible replication tools.” These characteristics allow us to store vast amounts of historical data in the data lake without worrying about managing the infrastructure and hardware needed to handle data growth as S3 would elastically provision the necessary resources for storing such data. In order to back up the ingested data to recover it in case of loss, we can replicate the data in the landing S3 buckets into S3 glacier buckets. S3 Glacier [5] is an AWS storage service that can be used for long-term objects storage at an extremely low cost.

For storing metadata, we can define user-level metadata as we ingest the data programmatically as described in [6]. Such metadata may include original file name, ingestion time, file source. In addition, we can run AWS Glue Data Catalog (described in next subsection) to launch crawlers that can periodically pull the schema of the data stored in S3 and infer the data types of the different fields in tables.

To satisfy possible future customer needs, our ingestion layer should have the capability to ingest data formats like png, jpeg (x-ray scans), json, Avro, parquet, ORC, and XML. We decided to use the Apache Hudi format for storing the data in S3 as it allows us to UPSERT records in the data lake whenever a Change in Data Capture (CDC) takes place at the level of the source data.

To secure the data in S3, we can encrypt it using either of the following methods [7]:

* **Server-Side Encryption** – Request Amazon S3 to encrypt your object before saving it on disks in its data centers and then decrypt it when you download the objects.
* **Client-Side Encryption** – Encrypt data client-side and upload the encrypted data to Amazon S3. In this case, you manage the encryption process, the encryption keys, and related tools.

We chose to encrypt the data using Server-Side Encryption as it removes the burden of managing the whole data security aspects of the lake.

It is worth noting that we purposefully ruled out using HDFS for storing the data landing from the ingestion layer. This is mainly because we would have to setup an EMR cluster to store HDFS clusters which would be relatively costly compared to storing the data in S3 as HDFS storage would be coupling storage and compute charges.

One drawback of using S3 for storage is the fact that querying data in S3 using an engine like Presto (or AWS Athena) can be costly unless we allocate resources to design the tables in S3 with proper indexing tailored to the use case of the user.

Finally, we chose using Redshift as a data warehouse to store reporting and analytics data in a columnar format allowing fast data reads. The transformed data in S3 can be easily copied into Redshift using a simple COPY command [8].

## Processing Layer

The processing layer consists of 2 components:

* AWS Glue: a serverless ETL orchestration layer that allows authoring ETL jobs with Apache Spark through either Scala or Python (PySpark). It also has a crawler that can craw through S3 buckets to infer the schema of the stored tables. Using SparkSQL, we can also author jobs using SQL like syntax. Glue can scale horizontally to handle thousands of ETL jobs across multiple data stores. It is worth noting that Glue can also handle processing streaming data coming directly from AWS kinesis through the Spark Streaming framework [9].
* AWS Athena: a serverless compute engine that allows querying several data formats stored in S3 buckets using SQL-like syntax.

To analyze the schema of the data stored in the landing S3 buckets, we can run AWS crawlers to infer the schema of the tables in these buckets. To process the data in the landing S3 bucket, we can author Spark jobs in AWS Glue and run them periodically to cleanse the data, apply business rules and data validation constraints, transform this data, and store it in the S3 buckets storing the transformed data in the Apache Hudi format to allow UPSERTs whenever a CDC occurs.

To allow ad-hoc querying, we store the data in a well-thought-out partitioning scheme using AWS Athena compatible formatting. Athena can handle the processing of large-scale datasets without the need of complex ETL jobs [10]. It can be used to query the data in the landing and processed data S3 buckets.

Redshift, being a data warehouse, can also be considered a processing engine that copies the data from the data in the processed S3 buckets into the warehouse. Redshift can handle petabyte-scale data.

Finally, it is worth noting that we discarded similar open-source solutions such as Presto, Apache Spark deployed on an EMR cluster, and Apache Airflow which is an ETL workflow orchestration tool that allows authoring and managing ETL workflows using Direct Acyclic Graphs (DAG). We avoided using Presto, Spark and Airflow because the AWS-Glue can replace Spark + Airflow and is managed in a serverless solution. This reduces the burden of managing and scaling these solutions. One shortcoming of using AWS Glue is the fact that, unlike Apache Airflow, there is no interactive UI that shows the tasks composing ETL workflows.

## Serving Layer

The serving layer consists of 3 components:

* Quicksight: an AWS serverless dashboarding solution that allows reporting metrics and visualizing analytics using a backend engine such as Redshift or Athena.
* SQL interface for querying the data in the AWS Redshift data warehouse
* SQL interface for querying the data lake in S3 buckets through AWS Athena

This layer is responsible of serving the data from our platform to data consumers. These consumers may have the following archetypes:

* Business analysts and company executives: they consume visualizations of KPIs and metrics through dashboards which allows them to make business decisions and monitor the health of the business. For these consumers, dashboards displayed on AWS Quicksight can be used to monitor business KPIs.
* Data/Product Analysts: they consume mostly aggregated and sometimes raw data to conduct analyses and deep data dives that allow them to understand several areas of the business. These consumers may use mostly Redshift to query aggregated data - or Athena to query granular data in the lake - and perform their analyses
* Data Scientists: they consume raw and aggregated data to build sophisticated statistical and machine learning models that automate mundane business processes. Unlike data analysts, these consumers would use mostly Athena to query granular data which they might wrangle to create suitable features to train their models.

In this layer, it would make sense to collect consumers’ usage data. Being able to understand which columns and which tables are consumed the most would be interesting to monitor the health of the business and to understand which clients are the most active.

# Conclusion

In this document, we presented a scalable data lake architecture that accommodates to MDPC’s growing data management needs. The lake consists of 4 decoupled layers: ingestion, storage, processing, and serving and uses serverless compute components (AWS Glue, Athena, Redshift) whenever possible. The storage layer in the lake uses S3 which allows scaling flexible data storage (no need for defining rigid schemas) automatically without the need of CAPEX and at a reasonable cost. The processing layer is not only efficient and scalable but is also affordable as cost is incurred only when compute power is consumed. Finally, the serving layer democratizes data access through dashboards (Quicksight) and SQL interfaces to query raw data in S3 through Athena or aggregated and curated data through Redshift.

# 8. References

1. <https://aws.amazon.com/blogs/big-data/stream-transform-and-analyze-xml-data-in-real-time-with-amazon-kinesis-aws-lambda-and-amazon-redshift/>
2. <https://www.reddit.com/r/aws/comments/ci3249/best_way_to_replicate_files_from_an_external_ftp/>
3. <https://aws.amazon.com/dms/>
4. <https://aws.amazon.com/s3/>
5. <https://aws.amazon.com/s3/glacier/>
6. <https://docs.aws.amazon.com/AmazonS3/latest/userguide/UsingMetadata.html>
7. <https://docs.aws.amazon.com/AmazonS3/latest/userguide/UsingEncryption.html>
8. <https://docs.aws.amazon.com/redshift/latest/dg/tutorial-loading-run-copy.html>
9. <https://docs.aws.amazon.com/glue/latest/dg/add-job-streaming.html>
10. <https://aws.amazon.com/athena/?whats-new-cards.sort-by=item.additionalFields.postDateTime&whats-new-cards.sort-order=desc>