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 |
| --- | --- | --- | --- |
| 04/05/2024 | Nhan V. Nguyen | 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 |
| --- | --- | --- | --- |
| FirstName LastName | Data Architect | Medical Data Processing | nvnhan.dev@icloud.com |

# 

# Purpose

This document aims to deliver a comprehensive technical overview of the proposed Data Lake solution for the Medical Data Processing Company. It details the architectural decisions essential for transitioning from the existing processes to the new Data Lake system, highlighting the advantages of this change.

The current setup does not suffice to fulfill the needs and requirements of the Medical Data Processing Company, prompting the need for this new proposal.

The intended readership of this document comprises technical personnel, including engineers, software architects, and others in similar roles.

The project will cover the following aspects:

* Ingestion layer
* Storage layer
* Processing layer
* Serving layer

The project will not address the following aspects:

* Monitoring
* Data quality
* Governance

# Requirements

**Business Requirements**

* Enhance the overall system uptime
* Decrease the latency for SQL queries and reporting
* Ensure system reliability and fault tolerance
* Scale the architecture in response to increasing data volume and velocity
* Foster business agility and innovation speed through enhanced automation and the freedom to test new frameworks
* Prioritize the use of open-source tools to prevent vendor lock-in
* Implement a metadata-driven design approach, utilizing a common set of scripts for processing diverse data sets instead of creating bespoke scripts for each data type
* Centralize storage of all enterprise data to simplify access

**Technical Requirements**

* Process incoming files instantaneously, moving away from the current nightly batch processing
* Architecturally separate metadata, data, and compute/processing layers
* Maintain an unlimited historical data archive
* Enhance processing speeds proportional to data volume increases
* Ensure the system can withstand a few node failures without downtime
* Implement capabilities for change data capture (CDC) and UPSERT operations on selected tables
* Facilitate multiple use cases from the same dataset without relocating or extracting data
  + Integrate with various ML frameworks like TensorFlow
  + Develop dashboards using tools like PowerBI, Tableau, or Cognos
  + Automate the generation of daily, weekly, and nightly reports through scripts or SQL
* Support ad-hoc data analytics and interactive querying with SQL

**Current Data Volume**

* Data sourced from over 8,000 facilities
* 99% of zip files range in size from 20 KB to 1.5 MB
* Edge cases include some zip files as large as 40 MB
* Each unzipped file typically contains CSV, TXT, or XML records
* XML zip files may include 20 to 300 individual XML files, each containing one record
* Average daily receipt of 77,000 zip files
* Approximately 15,000,000 data files processed daily
* Hourly averages include 3,500 zip files and 700,000 data files
* Data volume grows at an annual rate of 15-20%

# Data Lake Architecture design principles

The design decisions for the Data Lake solution are guided by principles essential for constructing a comprehensive end-to-end process.

The Data Lake is introduced as a solution to meet the scaling needs for data ingestion and processing at the Medical Data Processing Company. The existing process is inadequate as it does not utilize big data tools and frameworks, thus limiting its scalability.

Key principles guiding the new Data Lake architecture are:

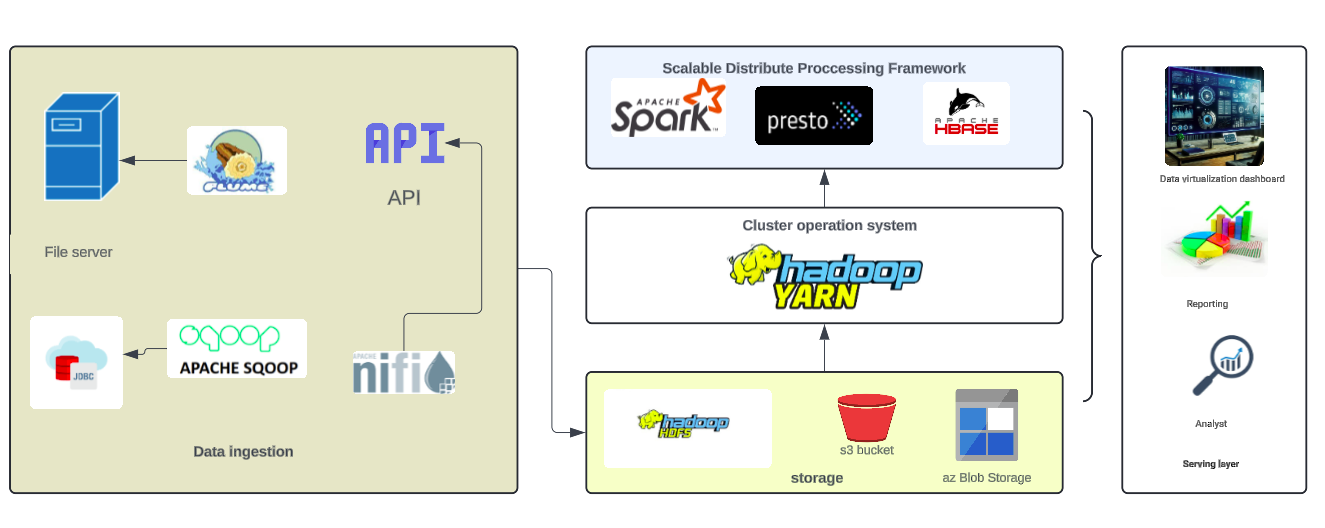
* Utilize open-source technologies to prevent vendor lock-in and enhance system flexibility.
* Segregate functional layers such as ingestion, storage, processing, serving, scheduling, and monitoring to promote independence and specialized efficiency in each area.
* Employ distributed processing where beneficial, to accelerate data processing capabilities.
* Implement streaming processing where appropriate, to enhance real-time data analytics.
* Adopt technologies that support horizontal scaling to improve the system's ability to grow in capacity as needed.
* Integrate monitoring tools extensively to enhance system observability and operational transparency.
* Log activities across all processes to support comprehensive auditing and traceability.
* Perform regular data snapshots to ensure robust data backup measures.
* Establish routines for data validation to maintain high data quality and ensure the delivery of reliable data to end-users.

# Assumptions

Data ingestion into the Data Lake will utilize FTP servers and APIs, with connectors facilitating data transfer either directly from the FTP servers or via Kafka brokers. Key elements and processes of the Data Lake architecture include:

1. **Data Ingestion**: Utilizes connectors to import data from external sources such as FTP servers and APIs, integrating this data into the Data Lake through direct transfers or Kafka brokers.
2. **Task Orchestration**: Airflow manages dependencies and orchestrates tasks across the entire data flow to ensure seamless data processing.
3. **Observability**: Grafana is employed across the platform to monitor and observe all components, ensuring high visibility into system performance and health.
4. **Data Backup**: Regular snapshots are scheduled as specified in the technical requirements, providing reliable data recovery options.
5. **Scalability Risk**: While the solution is designed to scale horizontally more efficiently than the current system, rapid data increases pose a potential scalability challenge.
6. **Query Performance Risk**: There may be a slight decrease in the speed of SQL query responses due to the integration with Athena and Iceberg, which could affect performance compared to previous solutions.

# Data Lake Architecture for Medical Data Processing Company



# Design Considerations and Rationale

## Ingestion Layer

To maintain compatibility with external source patterns, it's essential to minimize significant alterations to the interfaces within the ingestion layer.

The platform is designed to seamlessly ingest data in various formats such as XML, CSV, and TXT through APIs, FTP servers, and JDBC connectors.

For streaming data, the API serves as the interface, enabling requests that are subsequently published in Kafka topics for real-time processing.

Data can also be ingested via FTP servers, where external sources upload files like XML, CSV, or TXT, subsequently pulled by the Data Lake.

When dealing with data from relational databases, utilizing the JDBC connector is the preferred method for retrieval.

Scalability is addressed through the implementation of a proxy with a load balancer on the API side, alongside the ability to scale the Kafka cluster as needed.

While evaluating options, other open-source tools like Apache Sqoop, Apache Flume, and Apache Nifi were considered.

Apache Sqoop facilitates data transfer between Apache Hadoop and relational databases like MySQL, Oracle, and PostgreSQL via a command-line interface.

Apache Flume specializes in collecting and moving large volumes of log data to centralized storage systems like HDFS or Apache HBase.

Apache Nifi automates data flow across systems, offering features for ingestion, routing, and monitoring.

## Storage Layer

In designing the storage layer for our system, several critical considerations are paramount to efficiently manage vast amounts of data and accommodate a 20% Year-over-Year (YoY) data growth rate. Firstly, we opt for a scalable, distributed storage solution such as a cloud-based object storage service or a distributed file system. This allows us to dynamically expand storage capacity as needed to accommodate increasing data volumes without compromising performance.

To handle backup and recovery effectively, we implement a multi-tiered strategy. This includes regular automated backups of data to redundant storage systems, ensuring data integrity and availability. Additionally, we employ snapshot-based backups to capture point-in-time copies of data for rapid recovery in case of failures or data corruption.

In terms of securing data at a high-level, encryption plays a pivotal role. We implement both data-at-rest and data-in-transit encryption to protect sensitive information from unauthorized access or interception. Access control mechanisms are also crucial, including role-based access control (RBAC) and robust authentication protocols such as OAuth or LDAP integration.

Furthermore, we plan to store custom metadata information alongside the data itself. This metadata will include a variety of information such as timestamps, data sources, versioning details, and user-defined tags. Storing metadata alongside the data enables efficient indexing, searching, and retrieval operations, enhancing overall system usability and performance.

Regarding data format, we opt for standardized, interoperable formats such as JSON or Parquet. These formats facilitate seamless data exchange and interoperability with other systems and tools while also offering efficient storage and processing capabilities.

While considering tools for our architecture, we evaluated several third-party and open-source options. Among them were Apache Hadoop, Apache Cassandra, and MongoDB. However, we ultimately chose to build our architecture using a combination of cloud-native services like Amazon S3 for storage and AWS Glue for data cataloging and ETL processes. These selections align closely with our scalability, performance, and cost-efficiency requirements.

One potential shortcoming of our chosen tools is their reliance on cloud provider-specific technologies, which could introduce vendor lock-in. However, the benefits of scalability, reliability, and managed services outweigh this concern. Additionally, our architecture is designed with modularity and abstraction layers, allowing for future migration or integration with alternative tools if needed, mitigating the risks associated with vendor lock-in.

## Processing Layer

Data Processing Strategy: The first step is to devise a robust data processing strategy. This involves determining how you plan to process the data—whether it's through batch processing, real-time streaming, or Change Data Capture (CDC) mechanisms. Each approach has its advantages and is suited to different use cases. Batch processing is ideal for handling large volumes of data at scheduled intervals, while real-time processing caters to scenarios where immediate insights are crucial. CDC ensures that data changes are captured and processed in near-real-time, enabling timely updates to downstream systems.

Satisfying Different Processing Needs: Your architecture should be designed to accommodate diverse processing needs. This entails implementing mechanisms that can handle batch, real-time, and CDC processing seamlessly. For instance, you might employ Apache Spark for batch processing, Apache Kafka for real-time streaming, and Debezium for CDC.

Enabling Ad-hoc Querying Capabilities: Ad-hoc querying capabilities are essential for enabling users to explore and analyze data flexibly. This can be achieved by integrating a data warehouse or a data lake with querying tools like Apache Hive, Apache Impala, or Presto. These tools provide SQL interfaces for querying large datasets stored in distributed environments.

Tools for Processing: The processing layer involves a suite of tools tailored to different processing requirements. This may include Apache Spark, Apache Flink, Apache Kafka, Apache NiFi, Apache Beam, and Debezium, among others. Each tool serves a specific purpose within the processing pipeline, such as data ingestion, transformation, stream processing, and CDC.

Considered and Excluded Tools: During the selection process, various third-party and open-source tools were evaluated. Among these, alternatives like Apache Storm for stream processing, Confluent Platform for Kafka-based solutions, and Apache Flume for data ingestion were considered. However, they didn't make it to the final architecture due to factors like complexity, scalability limitations, or lack of community support.

Scalability: The proposed architecture is designed to scale horizontally to handle growing data volumes and processing demands. Horizontal scalability ensures that additional resources can be added dynamically to the processing layer, allowing it to scale out as needed. Tools like Apache Spark and Apache Flink inherently support distributed computing, making them well-suited for scaling processing workloads across clusters of machines.

## Serving Layer

In the context of a data lake proposal for the Medical Data Processing Company, the serving layer plays a crucial role in facilitating efficient data access and utilization. The serving layer typically refers to the component of the architecture responsible for serving processed or analyzed data to end-users or downstream applications in real-time or near-real-time.

In this proposal, the serving layer will likely contain curated, processed, and possibly aggregated data that has undergone transformations and optimizations to meet specific business needs and requirements. This could include structured data, such as reports, dashboards, or aggregated statistics derived from the raw data stored in the data lake.

The type of data stored in the serving layer could vary depending on the company's objectives and the needs of its stakeholders. It may encompass a wide range of data types, including but not limited to:

1. Analytical insights: Pre-calculated metrics, key performance indicators (KPIs), and analytical models derived from the raw data.
2. Aggregated data: Summarized data sets that provide a higher-level view of the underlying information, suitable for decision-making and reporting.
3. Processed data: Data that has been cleansed, transformed, and enriched to enhance its usability for specific business purposes.
4. Real-time data: Streaming data or data with low latency requirements that require immediate processing and delivery to end-users or applications.

The data in the serving layer serves various purposes, primarily aimed at enabling efficient decision-making, data-driven insights, and operational effectiveness. Some key uses of the data in the serving layer include:

1. Business intelligence and reporting: Providing stakeholders with access to pre-aggregated metrics, reports, and dashboards for monitoring performance, identifying trends, and making informed decisions.
2. Operational analytics: Supporting operational processes by delivering real-time or near-real-time insights into key operational metrics, enabling timely interventions and optimizations.
3. Application integration: Serving as a data source for downstream applications, analytics platforms, or machine learning models that require access to curated and processed data for their operations.
4. Data exploration and ad-hoc analysis: Enabling data scientists, analysts, and other users to interactively explore and analyze the data to uncover insights, discover patterns, and validate hypotheses.

# 8. Conclusion

The system outlined in this document offers a remedy for the prevailing issues faced by the Medical Data Processing Company. The current solution falls short of meeting both the technical and business demands of the organization. To address this, a redesigned architecture has been proposed to accommodate the escalating volume of data and the imperative for swift data accessibility demanded by platform users.

Should the board endorse the proposal, the next phase of the project involves initiating a Proof of Concept (PoC). This PoC will bring to life a scaled-down version of the project, enabling users to evaluate and test the efficacy of the proposed architecture.

# 9. References

Apache Nifi -<https://nifi.apache.org/>

Apache Spark -<https://spark.apache.org/>

Apache Sqoop -<https://sqoop.apache.org/>

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