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 |
| 09/16/2024 | Duy Le Khanh | 0.1 | Initial draft |

## Reviewers / Approval

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

## Key Contacts

|  |  |  |  |
| --- | --- | --- | --- |
| Name | Role | Team | email |
| Duy Le Khanh | Data Architect | Medical Data Processing | Duylk.end@email.com |

# Note from Instructor:

# Consider this as a comprehensive design document that you will deliver to the technical audience of the company.

# Provide detailed design and implementation level details

# You are expected to provide at least 6 pages worth of content (Does not include the cover (title) page and tracker page)

# Each section has a set of guiding questions that will help you derive the responses.

# Purpose

The purpose of this document is to provide a detailed technical design proposal for an enterprise data lake system for **Medical Data Processing Company**. It outlines the architectural framework, key technologies, and design choices to handle the company’s data challenges effectively. It aims to demonstrate how the architecture will meet business requirements, ensure scalability, and offer an agile, resilient, and high-performance data management solution.

This document is intended for a technical audience, with detailed insights into architecture decisions. It also identifies assumptions, risks, and implementation considerations while remaining out of scope for data governance and machine learning frameworks.

### ****In-Scope****

* Design a scalable and flexible data lake architecture capable of handling large volumes of data and integrating various data sources.

### ****Out-of-Scope****

* Full implementation of the architecture
* Data governance, machine learning implementation

# Requirements

### ****Business Requirements****

* Improve system uptime and reduce SQL query/report latency.
* Ensure the architecture scales with increasing data volume and velocity.
* System reliability, fault tolerance, and fault recovery.
* Allow ad-hoc analytics and interactive querying using SQL.
* Enable integration with business intelligence (BI) tools and machine learning (ML) frameworks.

### ****Technical Requirements****

* Process incoming data on-the-fly (real-time processing).
* Support change data capture (CDC) and UPSERT operations.
* Separate metadata, compute, and storage layers.
* Store unlimited historical data, ensuring fault tolerance and resilience.
* Create a metadata-driven design for ETL jobs to handle various file formats.
* Enable integration with ML frameworks (e.g., TensorFlow) and BI tools (PowerBI, Tableau, MicroStrategy).

# Data Lake Architecture design principles

### ****Scalability & Flexibility****

Medical Data Systems will process large volumes of data from diverse formats (CSV, XML, TXT). A scalable architecture ensures the system handles data growth efficiently, and flexibility ensures smooth integration of new data formats. Transitioning to standard formats like **Avro** and **Parquet** will optimize performance and allow easy tool integration.

### ****Resilience****

Using a central metadata repository like **AWS Glue** to store metadata will enhance data centralization, reducing IT overhead. Implementing event sourcing, which logs all data-related events in immutable storage (e.g., S3), will provide traceability and facilitate troubleshooting.

### ****Ease of Use****

Automating ETL pipelines using tools like **AWS Glue** and **Lambda** will eliminate bottlenecks. Unlike coding-based ETL frameworks, this approach streamlines data ingestion and transformation, enabling faster onboarding of new datasets.

### ****Performance****

* Storing metadata with each file ensures data is readily accessible.
* Columnar formats like **Parquet** and **ORC** improve performance.
* Adopt a **hot/cold storage** strategy, where frequently accessed data is kept in high-performance storage (hot), while infrequently accessed data is moved to cost-effective storage (cold).

# Assumption

· Immediate transition to a cloud-based data lake architecture.

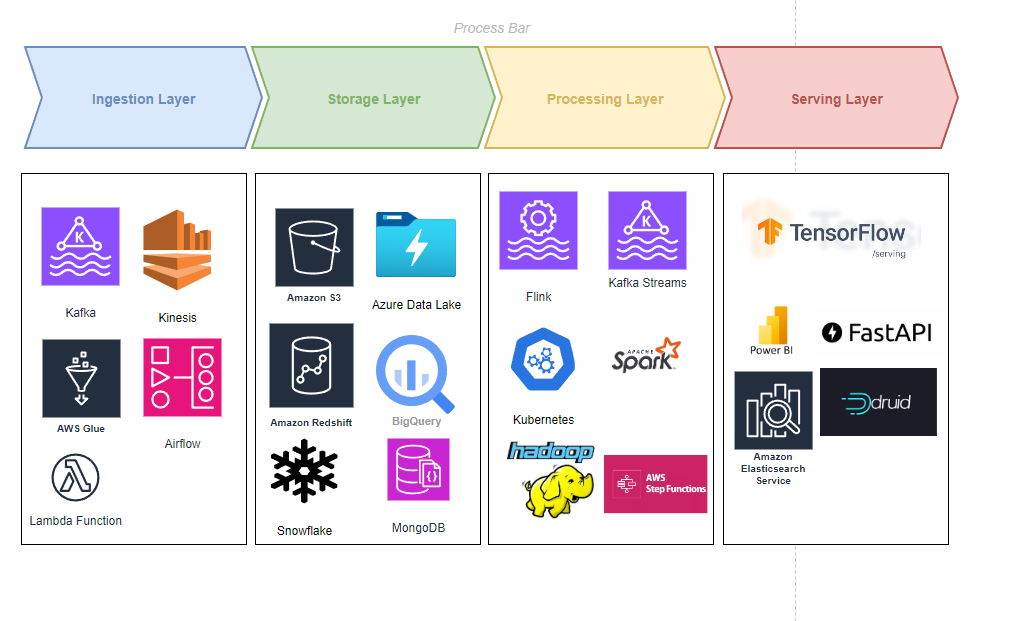
· All data will be migrated to the cloud, prioritizing scalability and reducing operational overhead.

· On-premise solutions are not prioritized due to high upfront costs and manual scaling challenges.

**Potential Current and Future Risks**

* Cloud Vendor Lock-In  
  Transitioning to a cloud-based data lake architecture may result in vendor lock-in, limiting flexibility if future business needs change. Migrating data and services to another provider could be costly and complex, creating long-term dependencies.
* Data Privacy and Regulatory Risks  
  Migrating all medical data to the cloud introduces compliance risks, especially for sensitive patient data. Cloud providers must adhere strictly to healthcare regulations (HIPAA, GDPR). Non-compliance, breaches, or data leaks could result in penalties and reputational damage.
* Downtime and Performance Issues  
  Relying solely on cloud infrastructure may expose the company to cloud provider outages or service disruptions, potentially impacting business operations. Ensuring high availability and disaster recovery plans will be crucial to mitigate this risk.
* Cost Overruns  
  Although on-premise solutions have high upfront costs, cloud scalability may lead to unforeseen expenses over time. Poorly managed cloud resources, such as unused storage or compute services, could cause budget overruns, especially as data volume grows.

# Data Lake Architecture for Medical Data Processing Company



# Design Considerations and Rationale

### 1. ****Ingestion Layer****

The ingestion layer is responsible for collecting data from various sources, which can be real-time streams, batch uploads, or external APIs. Here's how I would handle different types of data:

**Real-time Data**: Use a streaming platform like **Apache Kafka** or **Amazon Kinesis** to handle real-time event data, such as logs, IoT sensor data, or user activity. Kafka provides a distributed, fault-tolerant system for collecting real-time data.

**Batch Data**: For batch data ingestion (e.g., CSV, JSON, or Parquet files), I’d utilize **Apache Nifi** or **AWS Glue**. These can automate the ingestion process, transforming data on the fly and routing it to the appropriate storage system.

**API Data**: For data coming from third-party APIs or webhooks, I would schedule API calls using tools like **Airflow** or **AWS Lambda** (for event-driven ingestion), which could collect and store the data at regular intervals.

**Tools Considered but Not Selected**

Apache NiFi

* Reason Not Selected: Apache NiFi provides an intuitive user interface for designing data flows and is well-suited for real-time streaming ingestion. However, the use case for the Medical Data Processing Company primarily involves batch data ingestion, and Apache NiFi’s capabilities would have been overkill for this specific need. Additionally, its performance overhead can be high for batch-oriented use cases.

Kafka Connect

* Reason Not Selected: Kafka Connect is excellent for real-time streaming data pipelines. However, since the primary requirement here is to ingest large volumes of batch data (e.g., historical medical records), Kafka’s strength in handling streams wasn’t a top priority. Other tools such as AWS Glue or Apache Sqoop were found to be more suitable for the use case of batch ingestion.

### 2. ****Storage Layer****

Once data is ingested, it needs to be stored in a way that facilitates both quick access for processing and long-term storage for analytics. Here’s how I’d design this layer:

**Data Lake for Raw and Semi-Structured Data**: Store raw or semi-structured data in a data lake like **Amazon S3** or **Azure Data Lake**. This is ideal for storing large amounts of data in its native format. It’s cost-effective and scalable.

**Data Warehouse for Structured Data**: For structured data that needs to be queried often, I’d use a columnar data warehouse like **Amazon Redshift**, **Google BigQuery**, or **Snowflake**. These databases are optimized for analytical queries and can handle large-scale data storage.

**NoSQL Databases**: NoSQL databases such as **MongoDB** or **DynamoDB** would be leveraged for highly scalable, flexible-schema storage. They are especially suitable for scenarios like storing unstructured data (e.g., JSON or logs), rapidly changing data, and data that needs to support high read/write throughput.

**Cold Storage**: For older, infrequently accessed data, I would utilize **Amazon S3 Glacier** or **Azure Archive Blob Storage** for cost-effective long-term archival.

**Tools Considered but Not Selected**

Amazon Redshift

* Reason Not Selected: While Redshift is a powerful data warehouse solution that offers scalability and high performance for structured data analytics, it lacks the flexibility of a Data Lake when dealing with unstructured and semi-structured data. Redshift imposes a structured schema on data, whereas the use case involves various types of medical data, including raw, unstructured data (e.g., medical images, sensor data), making a more flexible storage solution like AWS S3 or HDFS preferable.

Google BigQuery

* Reason Not Selected: BigQuery is a managed data warehouse service with great query capabilities for structured data, but the Medical Data Processing system also needs to handle large amounts of unstructured data, including logs, images, and text. BigQuery’s structured data focus and cost model for queries were not ideal for our flexible data requirements. We selected AWS S3 due to its cost-effectiveness and better support for varied data formats.

### 3. ****Processing Layer****

Processing large amounts of data at scale requires a combination of batch and stream processing techniques. Here's how I'd handle it:

**Batch Processing**: For large-scale batch processing, I would use **Apache Spark** on a distributed platform like **Hadoop** (HDFS) or **Databricks**. Spark's in-memory processing speeds up data transformations and its distributed nature ensures scalability for large datasets.

**Stream Processing**: For real-time data processing, tools like **Apache Flink** or **Apache Kafka Streams** would be employed. These platforms allow for the processing of data in near real-time, which is crucial for applications like fraud detection or real-time analytics.

**ETL Workflows**: Orchestration tools like **Apache Airflow** or **AWS Step Functions** would be used to manage data workflows. These tools ensure that data processing happens in a structured way and can handle dependencies between tasks.

**Distributed Computing**: For heavy computations or ML model training at scale, I’d use **Kubernetes** or **AWS EMR** (Elastic MapReduce) to scale Spark, TensorFlow, or PyTorch jobs across a cluster.

### 4. ****Serving Layer****

Once the data has been processed, it needs to be served in a format that can be consumed by different applications, such as machine learning models, reports, or dashboards. Here's how I’d approach the serving layer:

**For ML Models**: Processed data would be fed into machine learning pipelines using frameworks like **TensorFlow Serving**, **MLflow**, or **SageMaker** for model inference and deployment. These tools allow for automatic scaling of ML models based on traffic.

**For Analytics and Reports**: I’d set up data marts in the data warehouse (Redshift, BigQuery, or Snowflake) and use BI tools like **Tableau**, **Power BI**, or **Looker** to visualize and serve reports. These tools can query data from the warehouse and display real-time dashboards.

**For APIs**: To serve processed data to external applications, I would use **Flask** or **FastAPI** to build RESTful APIs. These APIs could query processed data from a fast-access storage like **Redis** or directly from a relational database if real-time queries are required.

**For Streaming Analytics**: If real-time dashboards are needed, I’d use **Apache Druid** or **ElasticSearch** for low-latency data querying and visualization. These databases are optimized for time-series and log data, making them ideal for real-time analytics.

# 8. Conclusion

The proposed data lake architecture for **Medical Data Processing Company** provides a scalable, resilient, and high-performance solution to manage and process vast amounts of data. By leveraging **AWS services**, the architecture addresses current limitations and provides future-proofing for increasing data volume and velocity. The system supports diverse data formats, enhances fault tolerance, and enables seamless integration with BI tools and ML frameworks

# 9. References <If any>

<Provide links of any external documentation, wiki, blogs that you used to complete your research to put this solution together>