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 |
| 18/12/2025 | Abiodun Dare | 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 | student@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 < approx. ¼ page>

This document explains a new and improved way to collect, store, and use the company’s medical data. The goal is to fix current problems like slow performance, system breakdowns, and limited growth, while also preparing the company to use advanced analytics and artificial intelligence in the future. It describes how data will move step by step—from when it is first received, to where it is safely stored, how it is processed, and how it is finally used for reports, insights, and machine learning. It also explains why certain technologies were chosen, how the system will scale as data grows, how failures will be avoided, and how sensitive medical data will be protected.

Overall, the document serves as a clear roadmap for engineers and technical leaders to build a more reliable and flexible data platform, replacing the old single, fragile database system. It focuses on handling medical records and analytics, but does not cover redesigning user interfaces or detailed infrastructure setup scripts.

# Requirements <approx. 1 page>

The company uses one main database server to handle a very large and fast-growing amount of medical data. As the data grows, the system struggles to keep up—overnight data processing is slow, often breaks, and sometimes causes system downtime. Because everything depends on this one server, when it fails, many reports and services are affected.

On top of that, analysing the data is difficult. Data has to be copied into other systems for reporting and analysis, which creates duplicates, delays, and separate “data silos” that don’t always stay in sync. Overall, the setup is unreliable, hard to scale, and not suited for modern analytics.

**Existing Technical Environment**

* Master SQL Server (single node)
* Stage SQL Server
* 70+ custom SSIS ETL jobs
* FTP servers and API extract agents
* Application and web servers
* 12TB disk (70% utilized)

**Current Data Volume**

* 8,000+ medical facilities
* ~77,000 zip files/day
* ~15 million records/day
* Data formats: CSV, TXT, XML
* XML zip files may contain up to 300 records each
* Growth rate: 15–20% YoY

**Business Requirements**

* Improve system uptime and reliability
* Reduce query and report latency
* Support near-real-time insights
* Enable ML and advanced analytics
* Avoid vendor lock-in (open source preferred)
* Centralized enterprise data access
* Metadata-driven design

**Technical Requirements**

* Stream and batch ingestion
* Separation of storage, compute, and metadata
* Unlimited historical data retention
* Horizontal scalability
* Fault tolerance
* CDC and UPSERT support
* SQL-based ad-hoc analytics
* BI and ML integration

These requirements are derived directly from the problem statement and reflect common large-scale healthcare data platform needs

# Data Lake Architecture design principles <approx. ½ page>

1. **The Decoupling of Storage and Compute** enables independent scaling and eliminates single points of failure.
2. **Leveraging Open-Source Technologies** prevents vendor lock-in and supports long-term flexibility.
3. **Metadata-Driven Design** reduces custom pipelines and improves maintainability.
4. **Schema-on-Read** allows ingestion of diverse data formats without upfront rigid schemas.
5. **Fault Tolerance by Design** to ensure the system survives node failures without downtime.
6. **One Dataset, Multiple Use Cases** should serve BI, reporting, ML, and APIs.

These principles ensure the platform remains scalable, cost-effective, and adaptable over time.

# Assumptions <approx. ⅓ page>

The company already has cloud storage that can hold large amounts of data, and while medical records don’t arrive instantly, they can still be loaded almost in real time. Sensitive health data will be protected using encryption and strict access controls, and the engineering team already has experience with the core technologies needed to run the system. Some data sources can’t send changes automatically, so uploading files will still be the main way data is collected.

There are a few risks to be aware of. If data descriptions and labels aren’t kept accurate, automation won’t work as well. The new system will also be more complex to run than a single traditional database, and the team will need time to get comfortable managing large, distributed systems.

# Data Lake Architecture for Medical Data Processing Company

# Design Considerations and Rationale <at least 3 pages>

## Ingestion Layer

**Data sources**

* 8,000+ medical facilities
* Data formats: **CSV, TXT, XML**
* Delivery methods: **Customer APIs, FTP pulls, customer push to hosted FTP**
* Volume: ~77,000 zip files/day (~15M records/day)

**Ingestion approach**

* **Landing zone on Object Storage** (e.g., Apache NIFi, Apache Sqoop, etc)
  + All incoming zip files land as-is (immutable, compressed, encrypted)
  + Folder structure by source → facility → date/hour
* **Streaming & event-driven ingestion**
  + Use **Kafka / cloud-native equivalents (Kinesis / Event Hubs / Pub/Sub)** to capture file arrival events
  + Enables near-real-time processing instead of nightly batches
* **Metadata-driven ingestion framework**
  + One generic ingestion service reads schema, format, and rules from metadata tables
  + Eliminates hundreds of custom SSIS scripts
* **CDC ingestion**
  + For systems that support it, use **Debezium / CDC connectors** to stream updates

**Tools (Open Source Preferred)**

* **Apache NiFi**: Flow-based ingestion, backpressure, visual monitoring
* **Apache Sqoop**: Bulk ingestion from relational databases
* **Apache Kafka** (optional): Near-real-time ingestion where feasible

**Why this works**

* Decouples ingestion from processing
* Handles spikes safely
* Improves uptime and fault tolerance
* Supports on-the-fly processing (business requirement)

## Storage Layer

### < Core Storage: ****Data Lake (Object Storage)****

Organized into zones:

**Bronze (Raw): Apache HDFS**

* Raw zipped and unzipped files
* Immutable, append-only
* Retained forever (compliance & audit)

**Silver (Cleaned / Standardized)**

* Parsed XML/CSV/TXT into columnar formats (Parquet/ORC)
* Schema normalization, deduplication, basic validation

**Gold (Curated / Business-ready)**

* Patient metrics, admissions, discharge summaries
* Aggregated and analytics-ready datasets

### Where NoSQL fits

* **Metadata Store (NoSQL: DynamoDB / Cosmos DB / Cassandra)**
  + Dataset definitions
  + Schema versions
  + Processing rules
* **Operational lookups**
  + Facility metadata
  + Reference data with high read/write needs
* **Low-latency APIs**
  + Serving near-real-time patient dashboards

**Why NoSQL**

* Horizontally scalable
* Handles semi-structured metadata
* Low-latency access without stressing analytical systems

## Processing Layer

**Core processing engine**

* **Apache Spark (batch + structured streaming)**
  + Handles millions of records per hour
  + Schema evolution support
  + Built-in fault tolerance

**Streaming**

* Spark Structured Streaming or Apache Flink
* Processes files/events as they arrive

**Orchestration**

* **Apache Airflow**
  + Manages dependencies
  + Retry logic
  + SLA monitoring

**Transformations**

* Metadata-driven Spark jobs
* CDC-based **UPSERTS / MERGE** using **Delta Lake / Apache Hudi / Iceberg**
* No nightly ETL bottlenecks

**Why these tools**

* Open source (avoids vendor lock-in)
* Horizontally scalable
* Proven at petabyte scale
* Node failures do not cause downtime

## Serving Layer

### SQL & Analytics

* **Distributed SQL Engine** (Trino / Presto / Spark SQL)
* Directly queries Data Lake (no data movement)
* Fast interactive analytics for analysts and engineers

### BI & Reporting

* PowerBI, Tableau, MicroStrategy
* Connect via SQL engines to Gold datasets
* Near-real-time dashboards (bed availability, admissions)

### Machine Learning

* Direct access to Silver/Gold data
* Integration with **TensorFlow, PyTorch**
* Feature extraction from historical + streaming data
* No data duplication

### APIs & Applications

* Application services query:
  + NoSQL stores (low latency)
  + SQL engines (aggregations)
* Removes SQL Server as single point of failure

# 8. Conclusion <approx 2-5 lines>

The proposed Data Lake architecture replaces the fragile monolithic SQL system with a scalable, fault-tolerant, and open-source platform. It enables real-time ingestion, advanced analytics, and machine learning while eliminating data silos and single points of failure. Recommended next steps include a pilot implementation and phased migration.

# 9. References <If any>

 Apache Spark Documentation

 Apache Kafka Documentation

 Delta Lake / Apache Hudi / Iceberg

 Apache NiFi User Guide