

Data Lake Architecture

A Comprehensive Design Document

Medical Data Processing Company

Version 1.1

Tracker

Change Record

Date	Author	Version	Change Reference
November 2025	Christopher O'Hara	1.0	Initial architecture design

Reviewers and Approval

Name	Version Approved	Position	Date
En-te-prise McArchitect	1.0	Udacity Reviewer, Chief Technology Officer	November 2025

Key Contacts

Name	Role	Team	Email
Data d'Architect	Lead Architect	Medical Data Processing	architect@medicaldata.com

1. Purpose

This document presents a comprehensive data lake architecture designed to address the critical scalability, reliability, and operational challenges faced by Medical Data Processing Company. The architecture provides a detailed technical specification for transitioning from the current monolithic SQL Server-based infrastructure to a distributed, fault-tolerant data lake platform capable of processing 77,000 files daily from 8,000 medical facilities while supporting 15-20% year-over-year growth.

The document contains a complete architectural blueprint encompassing four distinct layers: data ingestion, distributed storage, scalable processing, and multi-purpose serving. Each layer is analyzed in detail with technology selections, design rationale, alternatives considered, scaling strategies, and operational considerations. The architecture emphasizes open-source technologies to eliminate vendor lock-in, metadata-driven processing to replace custom ETL scripts, and separation of storage and compute resources to enable independent scaling.

The target audience for this document includes enterprise architects, software engineers, technical directors, and infrastructure specialists who will be responsible for evaluating, implementing, and operating the proposed data lake platform. The document assumes technical familiarity with distributed systems, big data processing frameworks, and enterprise data architecture patterns.

Scope

In Scope:

- Complete data lake architecture design covering ingestion, storage, processing, and serving layers
- Technology stack selection with detailed justification for open-source components
- Scaling strategies to accommodate current data volumes and projected growth
- Fault tolerance mechanisms and backup and recovery strategies
- Security framework including authentication, authorization, and encryption
- Integration patterns for machine learning frameworks and business intelligence tools

Out of Scope:

- Detailed implementation code or deployment automation scripts
- Comprehensive cost analysis and total cost of ownership calculations
- Migration plan and cutover strategy from current SQL Server infrastructure
- Organizational change management and staff training programs
- Compliance documentation for HIPAA, GDPR, or other regulatory frameworks
- Performance benchmarking or proof-of-concept implementation results

2. Requirements

Problem Statement

Medical Data Processing Company faces a critical infrastructure crisis driven by hyper-growth that has exceeded the capabilities of the current monolithic SQL Server architecture. The company currently manages 8 terabytes of data in a single-node SQL Server database, processing 77,000 compressed files daily containing approximately 15 million individual records from 8,000 medical facilities. The system experiences severe performance degradation during nightly ETL operations, with recent database crashes causing hours of production downtime and potential data loss. Hardware optimization efforts, including maximum CPU and RAM configurations, indexing strategies, and stored procedure optimization, have failed to provide sustainable performance improvements.

The architectural limitations create cascading operational problems. The SQL Server represents a single point of failure for critical customer data with no rapid recovery mechanism. The current backup and restoration process requires hours of downtime, creating unacceptable business risk. The system cannot support real-time analytics, machine learning workloads, or concurrent query patterns, forcing the engineering team to export data nightly to separate servers. This data replication creates version control challenges, storage waste, and organizational data silos. The rigid architecture prevents business agility and innovation, as new analytical capabilities require extensive data movement and custom ETL script development.

The fundamental problem is an architectural mismatch between traditional relational database design principles and modern high-velocity, high-volume data processing requirements. The proposed data lake architecture addresses these challenges through distributed storage, horizontal scalability, metadata-driven processing, and separation of storage from compute resources.

Existing Technical Environment

Infrastructure Components:

- Master SQL Database Server: 64-core vCPU, 512 GB RAM, 12 TB disk space at 70% capacity (8.4 TB utilized)
- Stage SQL Database Server: Similar configuration for intermediate data processing
- 70+ ETL jobs managing over 100 database tables with custom SSIS packages
- 3 ingestion servers supporting FTP services and API-based data extraction
- Multiple web and application servers (32 GB RAM, 16-core vCPU each) for customer portal

Current Data Volume and Characteristics

Volume Metrics:

- Data sources: 8,000 medical facilities across multiple care types (urgent care, hospitals, nursing homes, emergency rooms, critical care)
- Daily ingestion: 77,000 compressed files averaging 20 KB to 1.5 MB (99% of files), with edge cases up to 40 MB
- Decompressed records: 15,000,000 individual data files per day in XML, CSV, and TXT formats
- Hourly processing: 3,500 files per hour containing approximately 700,000 records
- Data growth rate: 15-20% year-over-year, requiring architecture to scale from current 8 TB to projected 12-15 TB within two years

Business Requirements

The business requirements establish the strategic objectives that the data lake architecture must achieve to support organizational growth and operational excellence.

Core Requirements:

- System uptime improvement: Eliminate database crashes and minimize planned downtime for maintenance
- Query performance: Reduce latency for SQL queries and report generation from hours to seconds
- Reliability and fault tolerance: System must sustain individual node failures without service interruption
- Scalability: Architecture must accommodate data volume and velocity increases without redesign
- Business agility: Enable rapid experimentation with new analytical frameworks and data processing paradigms
- Open-source commitment: Embrace community-supported tools to avoid vendor lock-in and proprietary licensing costs
- Metadata-driven design: Replace custom ETL scripts with configuration-driven processing pipelines
- Centralized data access: Provide unified enterprise data repository eliminating organizational silos

Technical Requirements

Processing and Scalability:

- Real-time file processing capability replacing nightly batch-only operations
- Architectural separation of metadata, data storage, and compute processing layers
- Unlimited historical data retention without forced purging due to capacity constraints
- Linear scaling of processing throughput with data volume increases

Data Management:

- Change Data Capture (CDC) support for tracking incremental updates
- UPSERT operations on designated tables for data synchronization
- Multi-use case support from single dataset without data movement or replication

Integration and Access:

- Integration with machine learning frameworks including TensorFlow
- Dashboard connectivity for PowerBI, Tableau, and Microstrategy
- SQL-based report generation for daily, weekly, and monthly analytics
- Ad-hoc interactive querying capability for data exploration and analysis

3. Data Lake Architecture Design Principles

The architectural design follows three foundational principles that establish baseline criteria for all technology selections, component interactions, and operational patterns. These principles directly address the business and technical requirements while ensuring long-term architectural sustainability.

Principle 1: Open-Source Technology Stack

The architecture exclusively employs open-source technologies and community-supported frameworks, explicitly avoiding proprietary cloud platforms, commercial database systems, and vendor-specific tools. This principle addresses the business requirement to eliminate vendor lock-in and reduce total cost of ownership while maintaining architectural flexibility. Open-source components provide transparent implementation details enabling deep system understanding, community-driven innovation accelerating feature development, and portability across infrastructure environments. The principle directly supports business agility by allowing technology substitution without contractual constraints or licensing renegotiation.

Technology selections prioritize Apache Software Foundation projects with established governance models, active contributor communities, and production deployment track records at enterprise scale. This approach ensures long-term support availability, security patch responsiveness, and compatibility with emerging data processing paradigms. The open-source commitment enables Medical Data Processing Company to contribute improvements back to the community, participate in roadmap development, and maintain architectural control over critical infrastructure components.

Principle 2: Separation of Storage and Compute

The architecture implements strict separation between data storage and computational processing layers, enabling independent scaling of capacity and throughput. Storage resources grow linearly with data volume accumulation, while compute resources scale elastically based on processing workload intensity. This separation eliminates the fundamental limitation of the current SQL Server architecture where storage and compute are tightly coupled within a single server node. The principle directly addresses the technical requirement for unlimited historical data retention while maintaining query performance through computational scaling.

The separation enables multiple processing frameworks to operate concurrently on the same dataset without data replication, supporting batch ETL, real-time streaming, interactive SQL queries, and machine learning training from a unified storage layer. This architectural pattern eliminates the current operational complexity of nightly data exports to separate systems, reduces storage costs through elimination of redundant copies, and enables new analytical use cases without architectural modification. The principle supports the business requirement for centralized data access and the technical requirement for multi-use case support from single datasets.

Principle 3: Metadata-Driven Processing

The architecture replaces custom ETL scripts with configuration-driven processing pipelines that operate based on centralized metadata definitions. Schema information, transformation rules, data quality checks, and processing logic are stored as declarative configurations rather than imperative code. This metadata-driven approach directly addresses the business requirement to eliminate the 70+ custom SSIS packages that currently require individual maintenance and modification for each data source type.

The principle enables adding new data sources through metadata configuration updates rather than custom code development, reducing onboarding time from weeks to hours. Centralized metadata management provides schema evolution tracking, data lineage visualization, and impact analysis for changes. This approach supports the technical requirement for configuration-based data processing while improving operational efficiency through reduction of code maintenance burden. The metadata-driven design enables business users to understand data provenance and transformation logic without examining implementation code, improving data governance and regulatory compliance capabilities.

4. Assumptions

The architectural design relies on several foundational assumptions regarding infrastructure availability, organizational capabilities, and environmental constraints. These assumptions influence technology selections, operational patterns, and implementation approaches. Each assumption carries associated risks that must be monitored and mitigated during deployment.

Assumption 1: Linux-Based Infrastructure

The architecture assumes deployment on Linux operating systems (Ubuntu, CentOS, or Red Hat Enterprise Linux) rather than Windows Server environments. This assumption reflects the ecosystem maturity of open-source big data tools on Linux platforms, superior performance characteristics for distributed systems, and operational expertise availability in the market. The assumption impacts infrastructure provisioning, operational procedures, security hardening, and staff skill requirements.

Impact and Risk: The Linux assumption requires retraining of operations staff currently focused on Windows Server administration. Migration from the current Windows-based SQL Server environment introduces operating system transition complexity alongside application architecture changes. Risk mitigation includes phased staff training programs, contractor engagement for initial deployment, and comprehensive documentation of operational procedures. The assumption reduces licensing costs but increases initial knowledge transfer requirements.

Assumption 2: Network Bandwidth Sufficiency

The architecture assumes sufficient network bandwidth exists between data ingestion points and the data lake infrastructure to support 77,000 file transfers daily without creating bottlenecks. Current ingestion of 77,000 files averaging 500 KB compressed equates to approximately 38.5 GB daily throughput, requiring sustained network capacity of approximately 3.5 Mbps. The assumption extends to internal cluster networking for data replication, shuffle operations during distributed processing, and result set transfers to serving layers.

Impact and Risk: Insufficient network bandwidth creates ingestion delays, processing bottlenecks, and query latency issues that undermine the performance improvements the architecture aims to achieve. The distributed nature of the data lake amplifies network dependency compared to the current single-server architecture. Risk mitigation includes network capacity assessment during planning phases, quality-of-service configuration for data lake traffic, and monitoring of network utilization metrics. Future growth projections must account for network capacity expansion aligned with data volume increases.

Assumption 3: Staff Training and Adoption

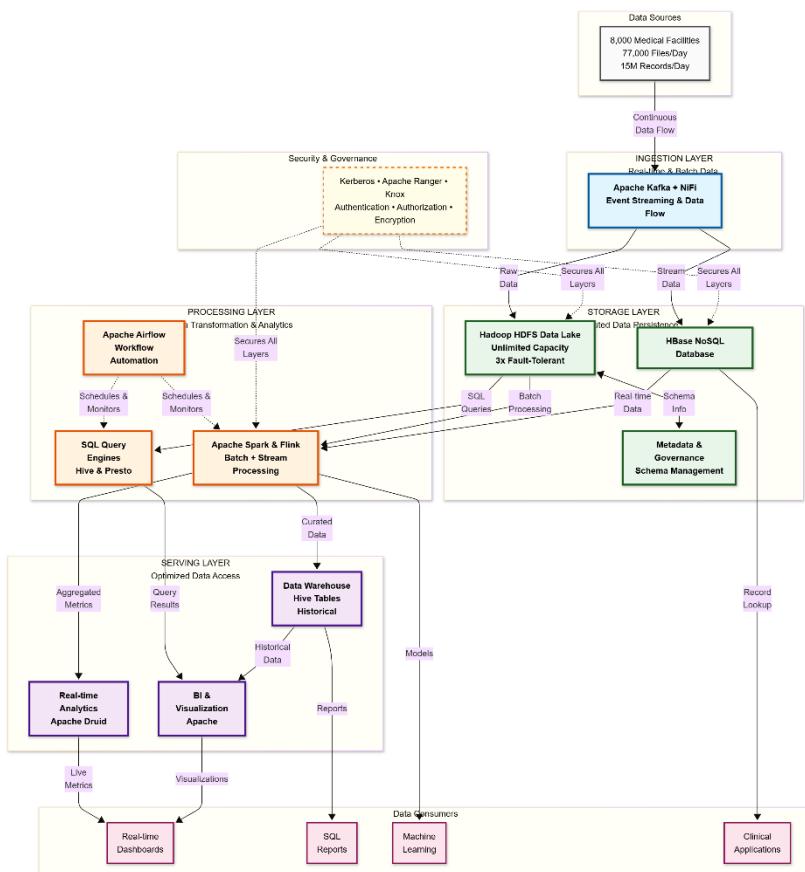
The architecture assumes organizational commitment to staff training for new technology stacks including Hadoop ecosystem tools, distributed processing frameworks, and modern data engineering practices. Current expertise in SQL Server administration, SSIS

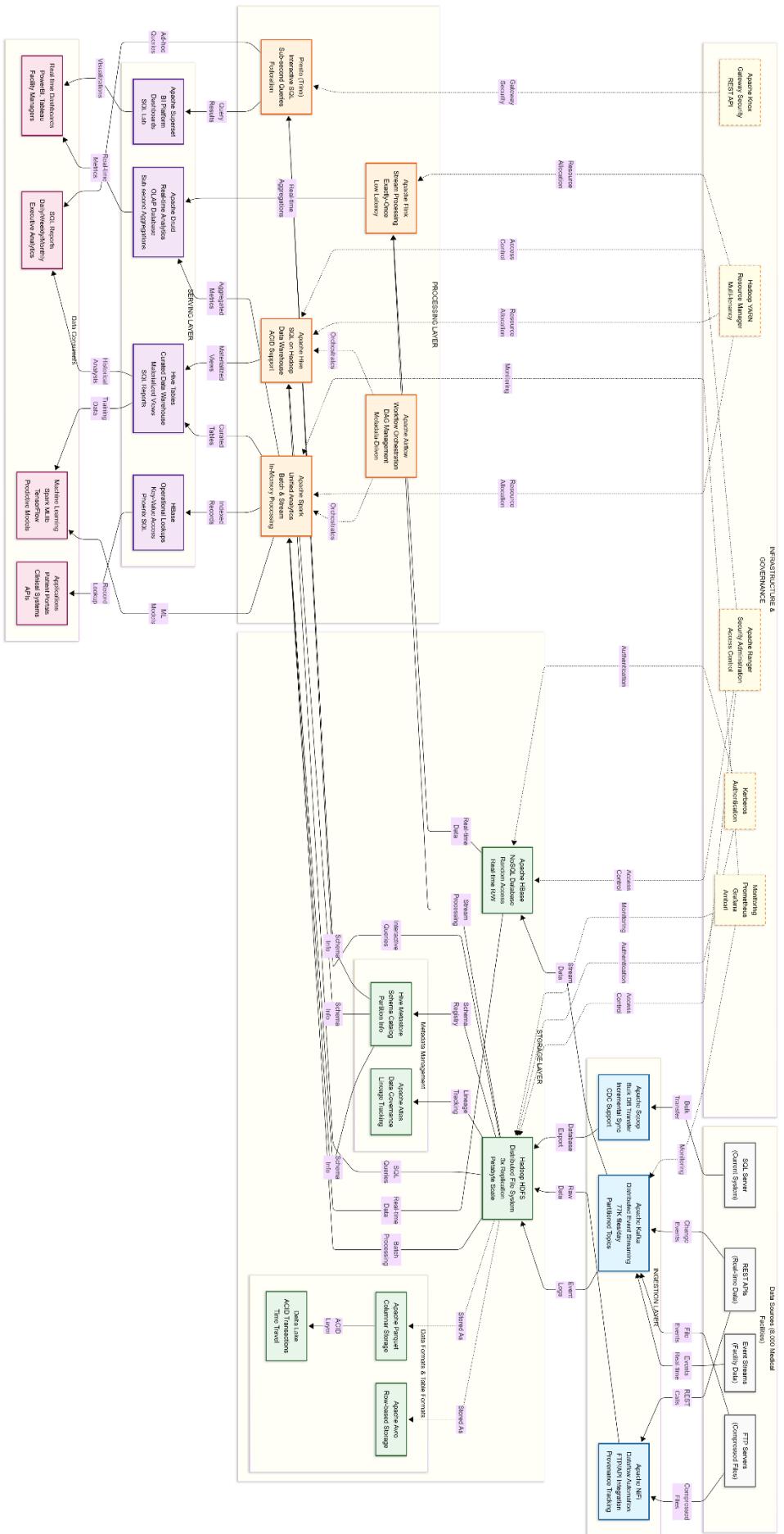
development, and relational database optimization requires supplementation with knowledge of HDFS operations, Spark programming, Kafka administration, and NoSQL database management. The assumption includes budget allocation for training programs, time allocation for learning curves, and acceptance of initial productivity reductions during transition periods.

Impact and Risk: Inadequate training investment results in suboptimal system operation, extended troubleshooting times, and potential data quality issues during the transition period. The paradigm shift from centralized relational databases to distributed data lakes represents significant conceptual complexity requiring sustained learning effort. Risk mitigation includes comprehensive training programs prior to production deployment, engagement of experienced consultants for initial implementation guidance, creation of detailed operational runbooks, and establishment of knowledge-sharing practices within the team. The assumption recognizes that architectural sophistication requires corresponding operational sophistication to realize intended benefits.

5. Data Lake Architecture for Medical Data Processing Company

The architecture diagram below illustrates a high-level version of the critical components. The next page illustrates a complete system architecture (lower-level of abstraction). The diagrams show the four-layer design with ingestion, storage, processing, and serving components. The diagram illustrates data flow from source systems through the ingestion layer into distributed storage, transformation via processing frameworks, and delivery to multiple consumption patterns through the serving layer. Specific technology logos for Apache Kafka, NiFi, HDFS, Spark, Hive, etc., are shown, as well as potential data consumers.





6. Design Considerations and Rationale

6.1 Ingestion Layer

The ingestion layer provides capabilities for collecting data from diverse sources including FTP servers, REST APIs, relational databases, and streaming event producers. The architecture supports both batch and real-time ingestion patterns, enabling the transition from current nightly-only processing to continuous data flow. The layer handles decompression of incoming files, schema validation, data quality checks, and routing to appropriate storage zones based on data type and processing requirements.

Technology Selection

Apache Kafka

Kafka serves as the primary real-time data ingestion platform, providing distributed event streaming capabilities with high throughput and low latency. The system handles 77,000 files daily across multiple topics partitioned by data source type, enabling parallel processing and fault-tolerant message delivery. Kafka's persistent log architecture enables replay of historical events for reprocessing scenarios, supporting the metadata-driven processing principle. The platform provides exactly-once delivery semantics through idempotent producers and transactional consumers, ensuring data integrity during ingestion. Kafka scales horizontally through partition addition and broker expansion, accommodating data volume growth without architectural changes.

Apache NiFi

NiFi provides visual dataflow automation for complex ingestion patterns including FTP polling, HTTP API calls, database change data capture, and file system monitoring. The platform offers 280+ processors supporting diverse protocols and data formats, eliminating custom script development for source system integration. NiFi's provenance tracking maintains complete data lineage from source to destination, supporting regulatory compliance and troubleshooting. The backpressure mechanism prevents system overload during ingestion spikes, while prioritized queuing ensures critical data flows receive processing preference. NiFi clusters scale through node addition and supports geographic distribution for data collection proximity to source systems.

Apache Sqoop

Sqoop enables bulk transfer from the existing SQL Server databases into HDFS, supporting initial data migration and ongoing incremental synchronization. The tool parallelizes data extraction across multiple mapper tasks, maximizing throughput for large table transfers. Sqoop integrates with the Hadoop ecosystem, directly writing data in Parquet or Avro formats optimized for subsequent processing. Incremental import modes support both append and last-modified update patterns, enabling change data capture without custom query development. The tool provides schema evolution handling through automatic type mapping from relational to Hadoop data types.

Scaling Strategy

The ingestion layer scales through horizontal addition of Kafka brokers and NiFi nodes. Kafka partitioning distributes message load across multiple brokers, with partition count adjusted based on throughput requirements. Each partition enables independent consumer parallelism, allowing processing throughput to scale linearly with partition addition. NiFi clustering provides load balancing across nodes through flow distribution, with automatic rebalancing during node failures. The architecture supports geographically distributed ingestion through Kafka MirrorMaker for cross-datacenter replication and NiFi Site-to-Site protocol for secure remote data transfer.

Alternatives Considered

Apache Flume

Flume provides log aggregation and event streaming capabilities with simpler configuration than Kafka. The tool excels at collecting log files from distributed sources and delivering to centralized storage. However, Flume lacks Kafka's message persistence and replay capabilities, limiting architectural flexibility for stream reprocessing. The single-writer-per-channel model constrains parallelism compared to Kafka's multi-producer architecture. Flume's declining community activity and limited integration with modern stream processing frameworks led to rejection in favor of Kafka's broader ecosystem support.

StreamSets Data Collector

StreamSets provides visual pipeline development with extensive connector library and drift detection capabilities. The platform offers superior user interface compared to NiFi with built-in data preview and validation. However, the smaller community size compared to Apache NiFi creates dependency on commercial support for production issues. Limited availability of third-party tutorials and troubleshooting resources increases operational risk. The commercial licensing model for enterprise features conflicts with the open-source principle, leading to NiFi selection despite StreamSets' interface advantages.

Apache Gobblin

Gobblin provides unified ingestion framework supporting batch and streaming data collection with built-in data quality checking. The platform emphasizes governance through metadata tracking and schema management. However, the steeper learning curve compared to specialized tools creates longer implementation timelines. The framework requires deeper Hadoop ecosystem knowledge for configuration and troubleshooting. Gobblin's unified approach trades simplicity for comprehensiveness, making it better suited for organizations with established big data expertise. The combination of Kafka and NiFi provides equivalent capabilities with gentler adoption curves and larger community support bases.

6.2 Storage Layer

The storage layer provides distributed, fault-tolerant persistence for all enterprise data with unlimited capacity scaling. The architecture implements zone-based organization separating raw ingested data, cleansed standardized data, and curated business-ready datasets. This separation enables data quality improvement through progressive refinement stages while maintaining complete lineage back to original sources. The layer supports schema-on-read flexibility through columnar storage formats, ACID transaction guarantees through table format layers, and multi-paradigm access through diverse storage engines.

Technology Selection

Hadoop HDFS

HDFS serves as the foundational distributed file system providing fault-tolerant storage across commodity hardware nodes. The system stores data in 128 MB blocks replicated across three DataNodes by default, ensuring data availability despite individual node failures. NameNode high availability through standby failover eliminates single points of failure in metadata management. HDFS scales to petabyte capacity through linear DataNode addition, supporting the technical requirement for unlimited historical data retention. The rack-awareness feature distributes replicas across failure domains, protecting against rack-level outages. HDFS integrates natively with all Hadoop ecosystem processing frameworks, providing optimal performance for distributed computation.

Apache Parquet and Avro

Parquet provides columnar storage format optimized for analytical query patterns, enabling predicate pushdown and column projection to minimize I/O. The format achieves superior compression ratios compared to row-based formats through homogeneous column data

types. Parquet supports complex nested data structures matching the XML document complexity in source medical records. Avro complements Parquet for use cases requiring schema evolution and row-based access patterns. Avro's self-describing format embeds schema definitions within data files, simplifying data exchange between systems. The combination supports both analytical and operational access patterns from unified storage.

Delta Lake

Delta Lake provides ACID transaction guarantees on top of Parquet files in HDFS, enabling reliable UPSERT operations and change data capture. The table format maintains transaction logs recording all modifications, supporting time travel queries for historical analysis and data recovery. Delta Lake's optimistic concurrency control allows multiple writers without coordination overhead. The merge operation efficiently handles incremental updates without full table rewrites, addressing the technical requirement for CDC support. Schema enforcement and evolution features prevent data quality issues during ingestion while allowing controlled schema modifications.

Apache Hive Metastore

The Hive Metastore provides centralized catalog for all data lake tables, storing schema definitions, partition information, and storage locations. The metastore enables metadata-driven processing through programmatic schema access, eliminating hardcoded schema definitions in processing code. Support for multiple file formats, compression codecs, and serialization libraries through table properties enables heterogeneous data storage. The metastore integrates with Spark, Hive, Presto, and other query engines, providing consistent schema views across tools. Partition pruning metadata accelerates queries through elimination of irrelevant data scans.

Apache HBase

HBase provides distributed NoSQL database for random read-write access to individual records, complementing HDFS's sequential access patterns. The column-family data model supports sparse, wide tables matching variable medical record structures. Automatic sharding across region servers enables linear scalability with data volume growth. Strong consistency guarantees ensure accurate reads of recently written data, supporting real-time dashboard requirements. HBase integrates with HDFS for underlying storage, leveraging replication for fault tolerance while providing low-latency access through memory caching.

Apache Atlas

Atlas provides data governance framework with metadata management, data lineage, and discovery capabilities. The system tracks data transformations from source to consumption, enabling impact analysis for schema changes. Classification and tagging support data sensitivity marking for security policy enforcement. Atlas integrates with Apache Ranger for policy-driven access control based on metadata tags. The search interface enables business users to discover datasets without technical knowledge of storage locations or file formats.

Capacity and Growth Planning

The storage layer accommodates 20% year-over-year growth through capacity expansion planning. Current 8 TB dataset projects to 9.6 TB in year one, 11.5 TB in year two, and 13.8 TB in year three. Initial cluster deployment with 24 TB raw capacity (8 TB usable after 3x replication) provides two-year headroom. DataNode addition follows linear scaling, with each 4 TB node contributing 1.33 TB usable capacity. HDFS balancer automatically redistributes data across new nodes during off-peak hours. Tiered storage strategy moves cold data exceeding one-year age to higher-density nodes with spinning disks, reserving SSD-backed nodes for hot data requiring low latency.

Backup and Recovery Strategy

The architecture implements multi-layer data protection combining replication, snapshots, and offsite backup. HDFS replication factor of three provides immediate protection against

individual DataNode failures. NameNode metadata receives separate backup to NFS-mounted storage with five-minute intervals, preventing metadata loss from NameNode failures. HDFS snapshots create point-in-time copies for recovery from logical corruption or accidental deletion. Weekly snapshot retention enables recovery of historical versions for compliance requirements. Incremental backup to object storage (MinIO) provides disaster recovery capability, with daily differential backups and weekly full backups maintaining two-month retention. The Kafka event log retention of 30 days enables complete data pipeline replay for catastrophic failures requiring full reprocessing.

Data Security

Security implementation addresses authentication, authorization, encryption, and audit requirements for HIPAA-compliant medical data. Kerberos provides strong authentication for all cluster services, eliminating password-based access. Apache Ranger centralizes authorization policy management with fine-grained access control at database, table, and column levels. Attribute-based access control enables dynamic policies based on user roles and data sensitivity classifications. Encryption at rest through HDFS Transparent Data Encryption protects data files from unauthorized access to storage media. TLS encryption for all network communication prevents eavesdropping on data transfers. Apache Knox gateway provides perimeter security with SSL termination, authentication, and authorization for REST API access. Comprehensive audit logging through Ranger tracks all data access for compliance reporting and security incident investigation.

Alternatives Considered

Ceph Storage

Ceph provides unified storage supporting object, block, and file interfaces from single cluster. The platform offers self-healing capabilities and dynamic data redistribution. However, operational complexity exceeds HDFS requirements for this use case. The additional block and object storage capabilities provide limited value given focus on file-based analytics. Ceph's CRUSH algorithm and erasure coding require deeper storage expertise for optimization. The smaller community compared to HDFS in big data contexts creates risk for troubleshooting and best practices guidance.

Apache Cassandra

Cassandra provides wide-column NoSQL database with linear scalability and multi-datacenter replication. The eventual consistency model supports high write throughput. However, limited support for secondary indexes constrains query flexibility compared to HBase. The lack of cross-row transactions complicates certain data consistency requirements. Cassandra's peer-to-peer architecture eliminates single points of failure but increases operational complexity compared to HBase's master-worker model. The decision favors HBase for stronger consistency guarantees and better Hadoop ecosystem integration, despite Cassandra's superior write performance.

MinIO Object Storage

MinIO provides S3-compatible object storage with high performance and cloud-native architecture. The platform would simplify integration with tools expecting S3 APIs. However, HDFS offers superior integration with Spark, Hive, and other Hadoop ecosystem tools through native file system interfaces. MinIO requires additional connector configuration and may incur performance overhead for data locality optimization. The architecture reserves MinIO for cold storage and disaster recovery backup rather than primary hot storage, leveraging S3 compatibility for archival use cases while maintaining HDFS for operational data.

6.3 Processing Layer

The processing layer transforms raw ingested data into business-ready datasets through batch ETL, real-time stream processing, and interactive SQL queries. The architecture supports multiple processing paradigms from unified storage, eliminating data movement between specialized systems. The layer implements the metadata-driven processing principle through configuration-based pipelines replacing custom SSIS scripts. Horizontal scaling through cluster expansion enables processing throughput to grow with data volumes.

Technology Selection

Apache Spark

Spark serves as the unified analytics engine for batch ETL, stream processing, and machine learning workloads. The in-memory computation model provides 10-100x performance improvement over MapReduce for iterative algorithms. Spark SQL enables schema-on-read processing of Parquet files with predicate pushdown and column pruning. The DataFrame API provides familiar relational operations while leveraging Catalyst optimizer for query planning. Spark Structured Streaming unifies batch and streaming code paths, enabling the same transformation logic for both processing modes. The platform scales through executor addition across cluster nodes, with dynamic resource allocation adjusting compute resources based on workload requirements.

Apache Flink

Flink provides stateful stream processing with exactly-once semantics and low latency characteristics. The checkpoint mechanism ensures fault tolerance through periodic state snapshots to reliable storage. Event time processing handles out-of-order events common in distributed medical record collection. Flink's window operations support tumbling, sliding, and session windows for time-based aggregations. The platform integrates with Kafka for source and sink connectors, enabling end-to-end stream processing pipelines. Flink scales through task parallelism across TaskManager nodes, with savepoints enabling version upgrades without data loss.

Apache Hive

Hive provides SQL interface to data lake storage, enabling analysts familiar with SQL to query distributed datasets without programming. The cost-based optimizer generates efficient execution plans leveraging data statistics and partition pruning. Hive supports complex data types including arrays, maps, and structs matching medical record structures. ACID transaction support through Hive 3.0 enables UPDATE and DELETE operations on existing tables. Materialized views accelerate repeated queries through pre-computed aggregations. Hive-on-Spark execution engine leverages Spark's performance advantages while maintaining Hive's SQL dialect compatibility.

Presto (Trino)

Presto provides distributed SQL query engine optimized for interactive analytics with sub-second latency. The MPP architecture parallelizes queries across worker nodes with pipelined execution. Memory-based processing eliminates disk I/O for intermediate results, accelerating complex joins and aggregations. Presto's connector architecture enables federation across HDFS, HBase, and external systems in single queries. The platform supports standard SQL including window functions, common table expressions, and correlated subqueries. Dynamic filtering pushes predicates from fact tables to dimension tables, reducing data scanned during star schema queries.

Apache Airflow

Airflow orchestrates complex data pipelines through directed acyclic graphs defined in Python code. The scheduler manages task dependencies, retries, and parallel execution across distributed workers. Dynamic pipeline generation enables metadata-driven workflows reading configuration from databases or files. Extensive operator library supports Spark, Hive, HDFS, and external system integrations. The web interface provides pipeline

monitoring, execution history, and manual trigger capabilities. Airflow replaces the 70+ SSIS packages with configuration-driven DAGs, reducing maintenance burden and improving observability.

Processing Paradigm Support

The architecture supports three processing paradigms addressing different use case requirements. Batch processing through Spark handles bulk ETL transformations of historical data with optimal resource utilization. Stream processing through Flink enables real-time data quality checks and continuous aggregations as data arrives. Interactive queries through Presto support ad-hoc analysis and dashboard refresh with sub-second latency. The separation of batch and stream processing from interactive querying prevents resource contention, with YARN resource queues ensuring fair allocation. The architecture supports Lambda architecture patterns through Spark for both batch and speed layers, or Kappa architecture through stream-only processing with replayable Kafka topics.

Scaling Strategy

Processing layer scaling occurs through horizontal addition of worker nodes to Spark, Flink, and Presto clusters. YARN dynamic resource allocation adjusts executor count based on queue depth and resource availability. Spark application parallelism increases through partition count adjustment matching cluster capacity. Presto worker addition immediately increases query parallelism without configuration changes. Flink scales through task slot addition enabling higher parallelism for stream processing. Airflow worker expansion increases concurrent task execution for pipeline orchestration. Auto-scaling policies trigger node addition when CPU utilization exceeds 70% for sustained periods, with scale-down during low-utilization periods reducing operational costs.

Alternatives Considered

Apache Beam

Beam provides unified programming model for batch and stream processing with portability across execution engines. The abstraction enables code written once to run on Spark, Flink, or other runners. However, the additional abstraction layer introduces complexity and potential performance overhead. Beam's dependency on runner-specific features creates portability limitations in practice. The smaller operator ecosystem compared to Spark and Flink requires more custom development. Direct use of Spark and Flink provides better performance optimization opportunities and simpler troubleshooting through elimination of abstraction layers.

Apache Storm

Storm provides mature stream processing with guaranteed message processing and low latency. The platform pioneered distributed stream processing in the Hadoop ecosystem. However, declining community activity and limited feature development favor Flink adoption. Storm's tuple-based API requires more boilerplate code compared to Flink's DataStream API. Lack of exactly-once semantics without additional complexity creates data quality risks. Flink's superior state management, event time processing, and exactly-once guarantees represent the modern evolution of Storm's original concepts.

Apache Impala

Impala provides distributed SQL engine with excellent performance for Hadoop data. The daemon architecture enables sub-second query latency through always-on processes. However, Impala's memory requirements constrain concurrent user support compared to Presto's more efficient resource utilization. Limited support for complex nested types affects usability with Parquet data containing deeply nested structures. Impala's integration primarily targets Cloudera ecosystem rather than general Hadoop distributions. Presto's federation capabilities, active development community, and lower resource footprint provide better fit for multi-tenant interactive analytics.

6.4 Serving Layer

The serving layer provides optimized data access patterns for diverse consumption use cases including SQL reporting, real-time dashboards, machine learning training, and business intelligence visualization. The layer materializes aggregations, denormalizes relationships, and indexes data for low-latency access patterns that would be inefficient against raw data lake storage. Multiple specialized storage engines coexist in the serving layer, each optimized for specific query characteristics and access patterns.

Purpose and Scope

The serving layer bridges the impedance mismatch between data lake storage optimized for bulk processing and application requirements for interactive queries and real-time updates. While the data lake stores comprehensive historical data in normalized form, the serving layer provides denormalized views, pre-aggregated metrics, and indexed lookups tailored to specific consumption patterns. The layer implements the technical requirement for supporting multiple use cases from single datasets without requiring data movement from the data lake. Materialization from data lake to serving layer occurs through scheduled batch jobs or continuous streaming pipelines based on latency requirements.

Technology Selection

Apache Druid

Druid provides real-time analytics database optimized for event data and time-series aggregations. The columnar storage with bitmap indexes enables sub-second aggregation queries over billions of rows. Druid's ingestion supports both batch loading from HDFS and streaming ingestion from Kafka, enabling hybrid batch-streaming architecture. The roll-up functionality pre-aggregates data during ingestion, reducing storage requirements and query execution time. Tiered storage automatically moves older segments to deep storage while maintaining hot data in memory and SSD. Druid serves the technical requirement for real-time dashboards with guaranteed query latency through query result caching and approximate algorithms.

Apache Hive Tables

Hive serves as the data warehouse layer for curated business datasets accessed through SQL reporting tools. Materialized views pre-compute expensive joins and aggregations referenced by multiple reports, amortizing computation cost across queries. Partitioning by date enables efficient time-range queries common in medical analytics. Bucketing on facility ID distributes data evenly across files, enabling map-side joins. ACID transaction support allows incremental updates to dimension tables without full rewrites. Hive's JDBC interface provides compatibility with existing SQL reporting tools, minimizing changes to current analytical workflows.

Apache HBase

HBase in the serving layer provides key-value lookup for application integration requiring individual record access. The sparse column-family model supports variable medical record schemas efficiently. Secondary indexes through Phoenix enable queries on non-key columns with acceptable performance. HBase snapshots provide point-in-time exports for analytical workloads without impacting operational read-write performance. The integration with Spark enables hybrid workloads combining large-scale analytics with real-time updates.

Apache Superset

Superset provides open-source business intelligence and visualization platform integrated with the data lake. The SQL Lab interface enables interactive query development against Hive, Presto, and Druid data sources. Drag-and-drop dashboard creation democratizes data visualization without requiring technical skills. The semantic layer defines metrics and dimensions once, ensuring consistent calculations across all visualizations. Superset's caching layer improves dashboard load times through query result storage. The platform

satisfies the business requirement for data access democratization while maintaining the open-source commitment.

Machine Learning Integration: Spark MLlib provides distributed machine learning algorithms operating on data lake storage. The library supports common algorithms including classification, regression, clustering, and collaborative filtering. Integration with TensorFlow through TensorFlowOnSpark enables deep learning training on distributed datasets. The ML pipeline API enables model training workflows with feature engineering, model selection, and hyperparameter tuning. Trained models serve predictions through batch scoring on Spark or real-time scoring through model serving frameworks.

Data Types and Usage Patterns

The serving layer stores three primary data types addressing different consumption patterns. Aggregated metrics in Druid provide pre-computed KPIs for real-time dashboards with guaranteed sub-second latency. Curated dimensional models in Hive support complex analytical queries joining multiple entities with historical context. Indexed operational data in HBase enables application integration requiring individual record lookup and updates. The data types represent different points on the latency-completeness tradeoff spectrum, with Druid favoring speed over completeness, Hive providing comprehensive historical analysis, and HBase balancing real-time access with data freshness.

Usage patterns align with organizational roles and responsibilities. Real-time operational dashboards consumed by facility managers query Druid for current patient counts, bed availability, and admission trends. Weekly and monthly executive reports query Hive for longitudinal analysis across facilities and care types. Application APIs serving patient portals and clinical systems query HBase for individual record details. Data scientists access raw data lake storage through Spark for exploratory analysis and model training, bypassing serving layer optimization when comprehensive data access is required.

7. Conclusion

The proposed data lake architecture addresses Medical Data Processing Company's critical scalability, reliability, and operational agility challenges through distributed storage, horizontal scaling, and separation of storage from compute resources. The open-source technology stack eliminates vendor lock-in while providing enterprise-grade capabilities for processing 77,000 files daily and accommodating 15-20% year-over-year growth. Implementation of the architecture enables real-time analytics, machine learning integration, and unified data access eliminating current organizational silos. Recommended next steps include detailed implementation planning, proof-of-concept deployment with representative workloads, and phased migration strategy minimizing disruption to current operations.

8. References

- Apache Software Foundation. "Apache Hadoop Documentation." <https://hadoop.apache.org/docs/>
- Apache Software Foundation. "Apache Spark Documentation." <https://spark.apache.org/docs/latest/>
- Apache Software Foundation. "Apache Kafka Documentation." <https://kafka.apache.org/documentation/>
- Apache Software Foundation. "Apache Flink Documentation." <https://flink.apache.org/documentation.html>
- Databricks. "Delta Lake Documentation." <https://docs.delta.io/latest/index.html>
- Presto Foundation. "Trino Documentation." <https://trino.io/docs/current/>
- Apache Software Foundation. "Apache NiFi Documentation." <https://nifi.apache.org/docs.html>

- Apache Software Foundation. "Apache Druid Documentation." <https://druid.apache.org/docs/latest/design/>
- Kleppmann, Martin. "Designing Data-Intensive Applications." O'Reilly Media, 2017.
- Narkhede, Neha, et al. "Kafka: The Definitive Guide." O'Reilly Media, 2017.