A Proposal of Integrated Component–Based Big Data Architecture

Oleksii Zhyrenkov[1], Anatolii Doroshenko[1,2]

[1] *Institute of Software Systems of the National Academy of Sciences of Ukraine, Glushkov Ave. 40, build. 5, Kyiv, 03187, Ukraine*
[2] *National Technical University of Ukraine "Igor Sikorsky Kyiv Polytechnic Institute", Peremohy Ave. 37, Kyiv, 03056, Ukraine*

Abstract

This paper introduces a comprehensive component-based architecture for big data systems, designed to overcome the constraints of traditional monolithic structures. By segmenting big data systems into four distinct layers (ingest, process, expose, and storage) this architecture fosters modularity, interchangeability, and technological flexibility. The approach seamlessly integrates established architectural patterns such as Lambda, Kappa, and Medallion architectures, while accommodating both Extract-Transform-Load (ETL) and Extract-Load-Transform (ELT) paradigms. For instance, the Lambda Architecture is exemplified by its dual-path processing, which is effectively utilized in systems requiring both batch and real-time data processing, such as in financial analytics platforms. The Kappa Architecture, on the other hand, is highlighted through its streamlined single-path processing, ideal for applications like real-time monitoring systems using tools like Apache Kafka and Apache Flink.

Key benefits of this architecture include reduced vendor lock-in, independent scaling of components, incremental evolution capabilities, and decreased technical debt. The architecture empowers organizations to select optimal technologies for specific functions, such as using Apache Spark for processing and S3 compatible systems for storage, while maintaining a cohesive framework that can adapt to evolving requirements and emerging technologies. Practical examples include the use of the Medallion Architecture in data lakehouses, where data is refined through progressive layers, enhancing data quality and accessibility.

This paper delves into the principles, patterns, and implementation considerations of this component-based approach, offering a detailed blueprint for designing resilient and adaptable big data systems. By examining real-world applications and tools, such as the integration of ELT processes in cloud-based environments using Snowflake, this paper provides valuable insights into the practical deployment of component-based architectures in diverse organizational contexts.

Keywords

Big Data, ETL, ELT, data pipelines, streaming pipelines, batching pipelines.

1. Introduction

Big data architectures have evolved significantly over the past decade, driven by the need to handle increasing volumes, velocities, and varieties of data. Traditional monolithic data architectures are increasingly being replaced by more flexible, modular approaches that can adapt to diverse requirements and technologies. This paper proposes a component–based architecture for big data systems that emphasizes modularity and interchangeability, allowing organizations to select optimal tools for each architectural layer while maintaining a consistent overall framework. The architecture integrates established patterns such as Lambda, Kappa, and Medallion architectures and accommodates both Extract-Transform-Load (ETL) and Extract-Load-Transform (ELT) paradigms. By decomposing big data systems into four primary layers—ingest, process,

Workshop "Software engineering and semantic technologies" SEST, co-located with the 15th International Scientific and Practical Programming Conference UkrPROG'2025, Kyiv, Ukraine, May 13-14, 2025
ozhyrenkov@gmail.com (O. Zhyrenkov); doroshenkoanatoliy2@gmail.com (A. Doroshenko);
0009-0007-3124-1359 (O. Zhyrenkov); 0000-0002-8435-1451 (A. Doroshenko);

expose, and storage—this approach enables incremental evolution, independent scaling, and technology flexibility while reducing vendor lock–in and technical debt.

2. Overview of big data architecture principles

2.1. Evolution of Big Data Architectures

The evolution of big data architectures reflects the continuous adaptation to growing challenges of data complexity and scale. Early approaches focused on batch processing using technologies like Hadoop MapReduce, which could effectively process massive datasets but with significant latency. As real-time analytics became increasingly important, architectures evolved to combine batch and stream processing capabilities.

This evolution gave rise to several architectural patterns. The Lambda Architecture provides a framework for handling both batch and stream processing through separate paths, while the Kappa Architecture simplifies this by treating all data as streams. The Medallion Architecture offers a structured approach to data refinement through progressive layers (Bronze, Silver, Gold). These patterns have been complemented by the evolution of data processing paradigms, with traditional ETL (Extract, Transform, Load) increasingly being complemented or replaced by ELT (Extract, Load, Transform), particularly in cloud environments.

Having established the evolution of big data architectures and the various patterns that have emerged, we now turn to a detailed examination of the component–based architecture. This approach builds upon the lessons learned from previous architectures while addressing their limitations through modularity and standardization.

2.2. Lambda Architecture

The Lambda Architecture, first proposed by Nathan Marz, addresses the challenge of computing arbitrary functions on massive datasets while providing both comprehensive accuracy and low latency results[1]. This architecture consists of three primary layers:

1. **Batch Layer (Cold Path)**: Stores all incoming data in its raw form and performs batch processing to create comprehensive batch views. This layer prioritizes accuracy over speed, processing the full dataset to produce high–quality results[1][14].
2. **Speed Layer (Hot Path)**: Analyzes data in real–time, providing low–latency results at the expense of some accuracy. This layer handles only the most recent data, compensating for the processing delay in the batch layer[1][5].
3. **Serving Layer**: Indexes the batch views for efficient querying and combines them with real–time views from the speed layer to provide complete results to users[1][14].

The Lambda Architecture effectively addresses the tension between accuracy and latency by providing both comprehensive batch processing and real–time analysis. However, it introduces complexity by requiring the implementation and maintenance of two separate processing paths with potentially duplicated logic[1]. This duplication increases development effort and raises the risk of inconsistencies between batch and stream processing results.

2.3. Kappa Architecture

The Kappa Architecture, proposed by Jay Kreps, simplifies the Lambda Architecture by eliminating the batch layer and processing all data through a single streaming path[1][5]. In this architecture, data flows through a unified log (such as Apache Kafka) and is processed by a stream processing system to create real–time views.

According to Kalra, “The Kappa architecture system is like a Lambda architecture system with the batch processing system removed, which avoids duplicating logic”[5]. This simplification

reduces complexity but requires a robust stream processing system capable of handling the entire data workload.

The Kappa Architecture retains some characteristics of Lambda's batch layer, particularly the immutability of event data. When recomputation is necessary (equivalent to what the batch layer does in Lambda), the entire data stream is replayed, typically using parallelism to complete the computation efficiently[1]. This approach provides a more streamlined architecture while maintaining the ability to process historical data when needed.

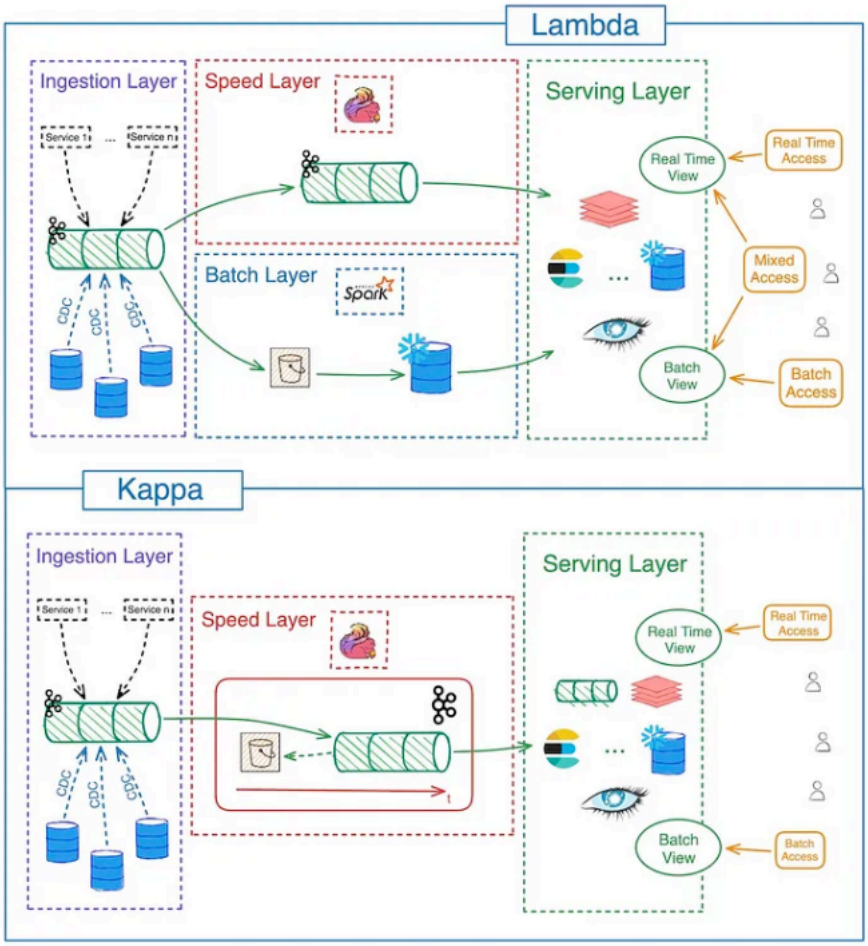

Figure 1: Schematic overview of Lambda and Kappa architecture components.

2.4. ETL vs. ELT Paradigms

The methods of moving and transforming data have evolved significantly, with traditional Extract, Transform, Load (ETL) approaches increasingly being complemented or replaced by Extract, Load, Transform (ELT), particularly in cloud and big data environments.

In ETL, data is extracted from source systems, transformed on a separate processing server, and then loaded into the destination system[7][10]. This approach works well for complex transformations and scenarios requiring data cleansing before storage. ETL is particularly suitable for environments with rigid destination schemas that don't change frequently[16]. It also provides better control over data quality and privacy, as sensitive data can be filtered or masked before reaching the destination system.

ELT, by contrast, extracts data from sources, loads it directly into the destination system, and then performs transformations within that system[7][10]. This approach leverages the processing power of modern data warehouses and lakes, enabling more flexible and scalable data processing.

ELT is particularly advantageous for large datasets requiring speed and efficiency, as it allows for simultaneous loading and transformation of data[10].

As Rivery explains, "ELT processes data faster than ETL. ETL includes a preliminary transformation step before loading data into the target, which becomes difficult to scale and slows performance as data size grows. ELT, in contrast, loads data directly into the target system, transforming it in parallel"[10]. Additionally, ELT preserves raw data in the destination system, enabling more flexible analytics and reducing the need to re-extract data when new transformation requirements emerge.

The choice between ETL and ELT depends on various factors including data volume, transformation complexity, schema flexibility, and security requirements. Many modern data architectures employ a hybrid approach, using different paradigms for different data workflows based on their specific characteristics and requirements[7].

2.5. Medallion Architecture

The Medallion Architecture is a data design pattern used to logically organize data in a lakehouse, with the goal of incrementally and progressively improving the structure and quality of data as it flows through multiple layers[4][13]. This architecture, sometimes referred to as a "multi-hop" architecture, consists of three primary layers:

1. **Bronze Layer (Raw Data)**: The ingestion point for all raw data from external sources. Tables in this layer correspond to source system structures "as-is," along with metadata columns. The focus is on quick ingestion and historical archiving, providing an audit trail and enabling reprocessing if needed[4][13].
2. **Silver Layer (Cleansed and Conformed Data)**: Data from the Bronze layer is cleansed, validated, and conformed to common standards. This layer provides an "Enterprise view" of key business entities and concepts, with improved data quality and structure[4][13].
3. **Gold Layer (Enriched Data)**: Contains highly refined, analysis-ready data optimized for specific business use cases. This layer powers analytics, machine learning, and production applications with high-quality, aggregated, and enriched data[4][8].

According to Databricks, the Medallion Architecture "guarantees atomicity, consistency, isolation, and durability as data passes through multiple layers of validations and transformations before being stored in a layout optimized for efficient analytics"[13]. This progressive refinement ensures that data quality improves at each stage, providing appropriate levels of quality for different use cases.

The Medallion Architecture works particularly well with ELT workflows but can also be adapted for ETL in structured environments[7]. It provides a clear framework for data governance and quality management, making it increasingly popular in modern data platforms.

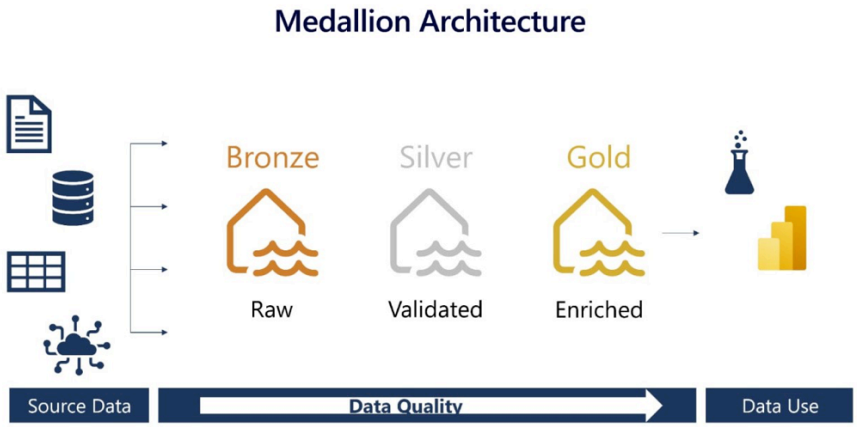

Figure 2: Schematic overview of a Medallion architectire.

3. Proposed Component–Based Architecture

Building upon the architectural patterns discussed above, we now present a comprehensive component–based architecture that integrates the strengths of these patterns while addressing their limitations. This architecture provides a flexible framework that can adapt to diverse requirements and technologies.

3.1. Architectural Principles and Overview

The proposed component–based architecture aims to address the limitations of monolithic big data systems by decomposing the architecture into modular, replaceable components. This approach enables organizations to select the best tools for each part of their data platform while maintaining a consistent overall architecture.

The architecture is guided by several key principles:

1. **Modularity**: Each component has a well–defined responsibility and interface, allowing it to be developed, tested, and replaced independently. This enables teams to focus on specific components without needing to understand the entire system in detail.
2. **Interchangeability**: Components are replaceable with alternative implementations that fulfill the same interface, enabling organizations to select the best tool for each function based on their specific requirements. This reduces vendor lock–in and allows the architecture to evolve over time.
3. **Standardization**: Components communicate through standardized interfaces and data formats, reducing integration complexity and enabling component substitution. This standardization simplifies integration and reduces the risk of incompatibilities.
4. **Separation of Concerns**: Each layer of the architecture focuses on a specific aspect of data processing, with clear boundaries between ingestion, processing, storage, and exposure. This separation simplifies component development and replacement.
5. **Scalability**: Each component is independently scalable based on workload requirements, allowing resources to be allocated efficiently. This ensures that the architecture can handle varying workloads without over–provisioning resources.

These principles enable an architecture that can evolve over time, incorporating new technologies and adapting to changing requirements without requiring a complete system redesign. The architecture provides a flexible framework that can implement various architectural patterns (Lambda, Kappa, Medallion) through appropriate component configuration.

3.2. Core Components and Interfaces

The proposed architecture consists of four primary layers, each responsible for a specific aspect of data processing. Figure 1 illustrates the high–level architecture and the relationships between layers.

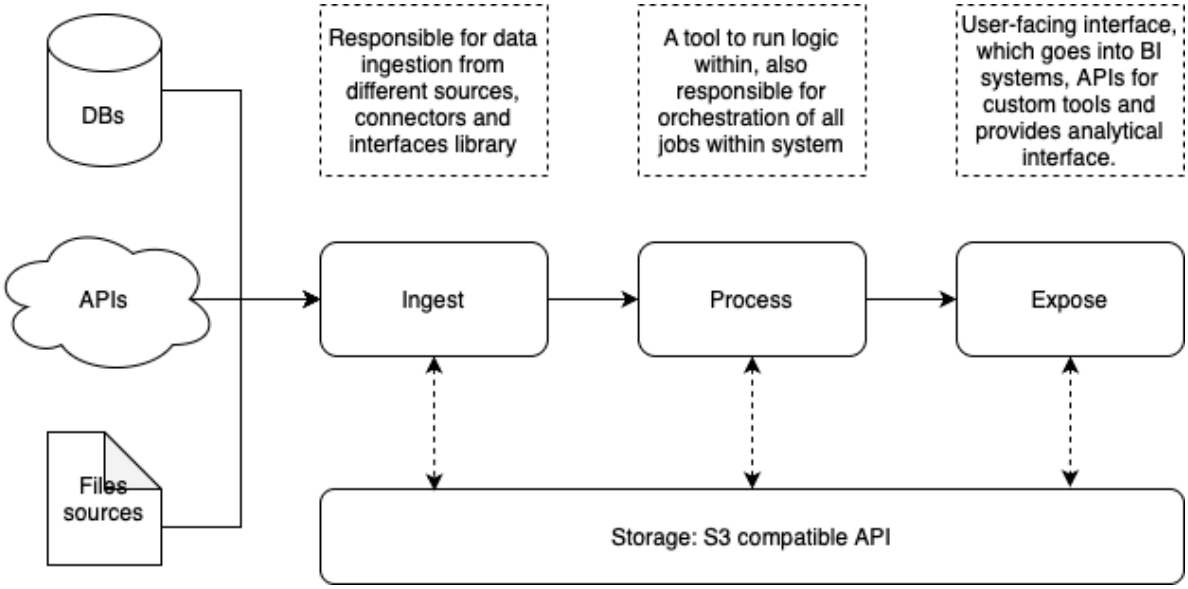

Figure 3: Component–Based Architecture Layers and Example Technologies

The proposed layers would be responsible for different phases of a general system:

1. **Ingest Layer**: Connects to diverse data sources, handles authentication, ensures reliable data transfer, and captures metadata.
2. **Process Layer**: Transforms and enriches data, implements business logic, and orchestrates processing workflows.
3. **Storage Layer**: Provides persistent, scalable storage with support for modern table formats (Iceberg, Delta Lake).
4. **Expose Layer**: Makes processed data available through various interfaces (SQL, REST, GraphQL).

Interface Requirements: – The ingest layer must write data to the storage layer in a standardized format – It must provide metadata about ingested data, including source, timestamp, and schema information – It should support both batch and streaming ingestion patterns

The ingest layer plays a critical role in establishing the foundation for data quality and governance. By capturing comprehensive metadata and ensuring reliable data transfer, it enables downstream processing to operate on well–documented and complete datasets.

3.2.1. Process Layer

The process layer is responsible for transforming, enriching, and orchestrating data processing pipelines. This layer implements the business logic required to convert raw data into valuable insights.

Responsibilities: – Orchestrating data processing workflows – Transforming and enriching data – Implementing data quality rules and validation – Managing dependencies between processing steps – Scheduling and monitoring processing jobs

Example Technologies: – Apache Airflow: A platform for programmatically authoring, scheduling, and monitoring workflows – Dagster: An orchestration tool for data pipelines with a focus on testing and maintainability – dbt (data build tool): A transformation tool that enables

analytics engineers to transform data using SQL – Apache Spark: A unified analytics engine for large–scale data processing

Interface Requirements: – The process layer must be able to read from and write to the storage layer – It must provide monitoring and logging information about processing jobs – It should support both batch and streaming processing paradigms

The process layer is where architectural patterns like Lambda, Kappa, and Medallion are primarily implemented. For example, a Lambda Architecture would involve separate batch and stream processing workflows, while a Medallion Architecture would involve progressive transformations from Bronze to Silver to Gold data.

3.2.2. Storage Layer

The storage layer is responsible for storing and managing data throughout its lifecycle. This layer provides a persistent, scalable, and reliable repository for data at various stages of processing.

Responsibilities: – Storing raw, intermediate, and processed data – Managing data formats and schemas – Providing efficient access patterns for different workloads – Ensuring data durability and reliability – Managing data lifecycle and retention policies

Example Technologies: – MinIO: An S3–compatible object storage server – Apache Hadoop HDFS: A distributed file system for big data – AWS S3: A scalable object storage service – Ceph: A distributed storage system with S3–compatible API

Data Formats: – Apache Iceberg: A table format for large analytics datasets providing ACID transactions, schema evolution, and partition evolution – Delta Lake: An open–source storage layer that provides ACID transactions, scalable metadata handling, and unified batch and streaming – Apache Parquet: A columnar storage format optimized for analytics

Interface Requirements: – The storage layer must provide an S3–compatible API for data access – It must support efficient reading and writing of various data formats – It should ensure data consistency and durability

The storage layer is the foundation of the architecture, providing a reliable and consistent view of data to other components. By supporting modern table formats like Apache Iceberg and Delta Lake, it enables advanced capabilities like time travel, schema evolution, and ACID transactions.

3.2.3. Expose Layer

The expose layer is responsible for making processed data available to downstream consumers. This layer provides fast, efficient access to data insights through various interfaces tailored to different consumption patterns.

Responsibilities: – Providing query interfaces for data access – Optimizing data for specific query patterns – Managing authentication and authorization for data access – Ensuring consistent and reliable data delivery – Supporting various data consumption patterns (ad–hoc queries, dashboards, APIs)

Example Technologies: – Elasticsearch: A distributed search and analytics engine – Cube.dev: An API layer for data analytics – Trino (formerly Presto): A distributed SQL query engine – Apache Superset: A modern data exploration and visualization platform

Interface Requirements: – The expose layer must provide standardized interfaces for data access (SQL, REST, GraphQL) – It must optimize query performance for different consumption patterns – It should ensure consistent data access semantics regardless of the underlying storage

Another example of an expose layer might be Elasticseach database, which provides unique features like simultaneous timeseries and spatial analysis capabilities, while maintaining decent speed of classical aggregation queries and also lives a room for indices tuning by using specific techniques. Another benefit and good example of an Expose functionality might be Kibana, which is de-facto always comes with Elasticsearch and provides user a BI-like experience [18].

The expose layer is where the value of the data platform is realized, providing business users and applications with access to insights derived from the data. By supporting multiple access patterns and interfaces, it enables a wide range of use cases from ad–hoc analysis to embedded analytics.

4. Technical Implementation Examples

Having established the theoretical foundations of the component–based architecture, we now present concrete implementation examples that demonstrate how these concepts can be translated into working systems. These examples showcase the interaction between different components and their integration points, providing practical insights into the architecture's implementation.

4.1. Example 1: Real–time Data Pipeline

This example demonstrates a real–time data pipeline implementation that spans multiple layers of the component–based architecture. The pipeline ingests streaming data through Kafka, processes it using Spark, and stores the results in an Iceberg table, showcasing the interaction between the ingest, process, and storage layers.

The implementation highlights several key aspects of the component–based architecture: – Standardized interfaces between components (Kafka topics, S3 storage) – Independent scaling of processing components – Clear separation of concerns between layers – Integration of streaming and batch processing capabilities

```
# Apache Kafka Producer Configuration
producer_config = {
    'bootstrap.servers': 'kafka:9092',
    'client.id': 'data-ingest-producer',
    'acks': 'all',
    'retries': 3,
    'compression.type': 'snappy'
}

# Apache Spark Processing Pipeline
from pyspark.sql import SparkSession
from pyspark.sql.functions import *

# Setup Spark with Iceberg
spark = SparkSession.builder \
    .appName("RealTimeProcessing") \
    .config("spark.sql.extensions",
"org.apache.iceberg.spark.extensions.IcebergSparkSessionExtensions") \
    .getOrCreate()

# Stream processing pipeline
df = spark.readStream \
    .format("kafka") \
    .option("kafka.bootstrap.servers", "kafka:9092") \
    .option("subscribe", "raw-data") \
    .load()

# Process and write to Iceberg
processed_df = df.select(
    from_json(col("value").cast("string"), schema).alias("data")
).select("data.*")
```

```
processed_df.writeStream \
    .format("iceberg") \
    .outputMode("append") \
    .option("path", "s3://data-lake/processed") \
    .option("checkpointLocation", "/checkpoints") \
    .start()
```

4.2. Example 2: Batch Processing Pipeline

This example illustrates a batch processing workflow using Airflow and dbt, demonstrating how the process layer can orchestrate complex data transformations while maintaining clear separation of concerns. The implementation shows how batch processing can be integrated with the storage layer while providing monitoring and logging capabilities.

Key aspects demonstrated in this example: – Workflow orchestration and scheduling – Data transformation and validation – Monitoring and logging integration – Dependency management between processing steps

```
# Airflow DAG Configuration
from airflow import DAG
from airflow.operators.python import PythonOperator
from datetime import datetime

default_args = {
    'owner': 'data-team',
    'start_date': datetime(2024, 1, 1),
    'retries': 3
}

dag = DAG(
    'batch_processing',
    default_args=default_args,
    schedule_interval='@daily'
)

def process_batch():
    # dbt configuration
    dbt_config = {
        'project_dir': '/dbt/project',
        'profiles_dir': '/dbt/profiles',
        'target': 'prod'
    }

    # Run dbt transformations
    subprocess.run(['dbt', 'run', '--project-dir',
dbt_config['project_dir']])

process_task = PythonOperator(
    task_id='process_batch',
    python_callable=process_batch,
    dag=dag
)
```

4.3. Example 3: Query Interface Implementation

This example shows how the expose layer can be implemented using Trino to provide efficient query access to data stored in the storage layer. It demonstrates the creation of materialized views for optimized query performance and the configuration of the query engine to work with the underlying storage format.

The implementation showcases: – Query optimization techniques – Materialized view creation and management – Integration with the storage layer – Performance tuning considerations

```
-- Trino Configuration
CREATE CATALOG iceberg WITH (
    type = 'iceberg',
    warehouse = 's3://data-lake/warehouse'
);

-- Create materialized view for optimized queries
CREATE MATERIALIZED VIEW iceberg.analytics.daily_metrics AS
SELECT
    date_trunc('day', event_time) as day,
    count(*) as event_count,
    sum(amount) as total_amount
FROM iceberg.raw.events
GROUP BY 1;

-- Query optimization example
EXPLAIN ANALYZE
SELECT * FROM iceberg.analytics.daily_metrics
WHERE day >= current_date - interval '7' day;
```

5. Comparative Analysis

The component-based architecture offers significant advantages over traditional monolithic approaches while addressing some inherent challenges. When compared to traditional architectures, it provides greater flexibility by enabling independent evolution of each layer. Organizations can replace individual components as requirements change without disrupting the entire system, unlike tightly coupled traditional implementations where changing one component (such as a batch processing engine) might require significant rework across multiple layers.

This architecture can implement various patterns (Lambda, Kappa, Medallion) through appropriate component configuration. For Lambda, it uses separate batch and stream processing pipelines in the process layer with results merged at the expose layer. Kappa implementation focuses on stream processing with storage formats supporting both streaming and batch operations. Medallion patterns emerge through progressive transformations in the process layer with different refinement stages in the storage layer.

The emphasis on standardized interfaces (like S3 API) and data formats (Apache Iceberg, Delta Lake) reduces integration complexity and enables component substitution. This standardization simplifies testing as components can be validated against their interfaces rather than within the entire system context.

Key advantages include technology flexibility (selecting optimal tools for specific functions), future-proofing (adapting to new technologies by replacing individual components), independent scalability of components, specialized expertise development, and progressive adoption possibilities. However, challenges exist: integration overhead between components, potential performance impacts from component communication, increased operational complexity in

managing distributed components, consistency challenges across implementations, and the need for diverse technical skills.

Performance considerations include managing inter-component communication latency (mitigated through co-location and efficient protocols), component-specific optimizations, data format efficiency impacts, and independent scaling strategies. Modern data formats provide performance advantages through statistics, indexing, and partition pruning, while independent component scaling enables efficient resource allocation directed at bottleneck components without over-provisioning the entire system.

5.1. Implementation Example: NGODS

Having established the theoretical foundations and comparative analysis of component–based architectures, we now turn to a practical implementation example. The New Generation Open Source Data Stack (NGODS) represents a concrete realization of the component–based architecture principles discussed earlier. This implementation demonstrates how the theoretical concepts can be translated into a working system that addresses real–world big data challenges.

The New Generation Open Source Data Stack (NGODS) represents a practical implementation of the component–based architecture described in this paper. As described by Svoboda, NGODS is a proof–of–concept open–source data stack composed of Apache Iceberg, Apache Spark, and Trino[3]. This implementation demonstrates how a modular, component–based approach can create a data platform that is both fast and feature–rich.

NGODS was initially motivated by the desire to experiment with Apache Iceberg features like git–like data snapshots, schema evolution, and partitioning. However, it evolved into a more comprehensive data stack that integrates multiple components to create a cohesive yet flexible data platform.

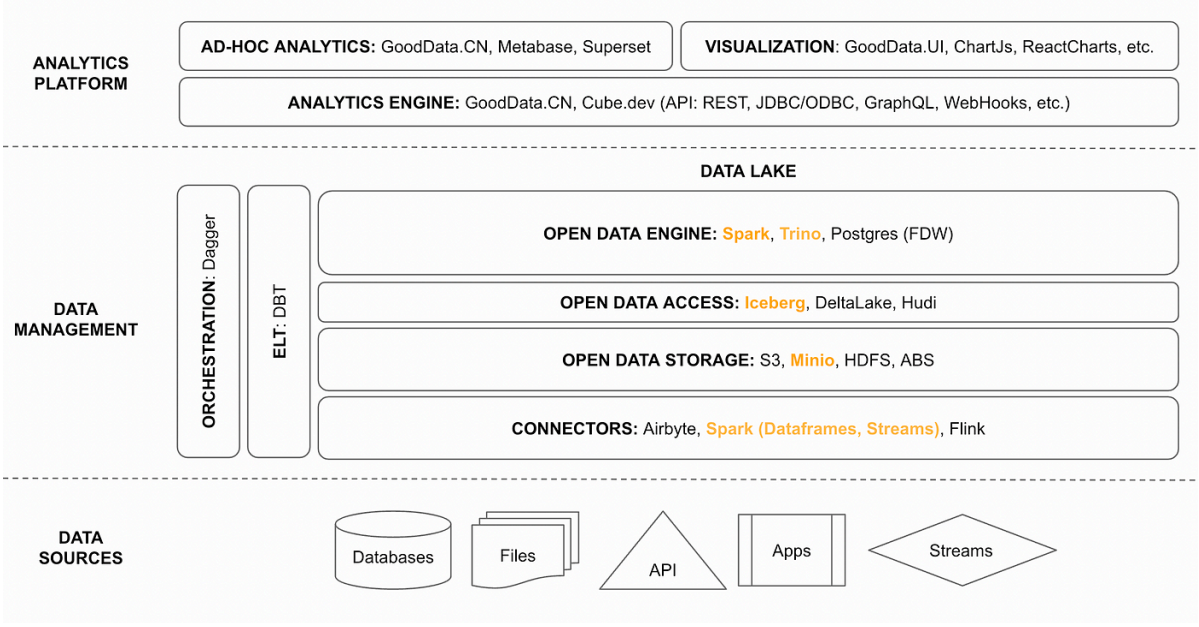

Figure 4: Components proposed in NGODS – New Generation Open Data Stack

5.2. NGODS Component Implementation

NGODS implements the component–based architecture with the following technologies:

Ingest Layer: While the initial NGODS implementation does not specify a dedicated ingestion tool, it can be integrated with tools like Airbyte for data ingestion. Airbyte provides a wide range of pre–built connectors for databases, APIs, files, and other data sources, making it a suitable choice for the ingest layer.

Process Layer: NGODS uses Apache Spark as the primary processing engine. Spark provides a unified analytics engine for large–scale data processing, supporting both batch and streaming workflows. It offers high–level APIs in Java, Scala, Python, and R, making it accessible to a wide range of data engineers and scientists.

Storage Layer: NGODS uses Apache Iceberg as its table format, providing features like git–like data snapshots for versioning and time travel, schema evolution for adapting to changing data structures, and flexible partitioning for optimizing query performance. These features enable a robust and flexible storage layer that can adapt to evolving data requirements while maintaining data integrity and consistency.

Expose Layer: NGODS uses Trino (formerly PrestoSQL) as its query engine, providing a fast and scalable way to expose data to analysts and applications. Trino is a distributed SQL query engine designed for analyzing large datasets, offering high–performance queries across diverse data sources, ANSI SQL compatibility, federated queries across multiple data stores, and a REST API for programmatic access.

Svoboda mentions plans to extend the stack with additional components including DBT for transformation management, Dagger for workflow orchestration, Flink for stream processing, and Postgres for relational data storage[3]. These additions would enhance the capabilities of the stack while maintaining its modular, component–based nature.

5.3. Integration and Data Flow in NGODS

In the NGODS implementation, components are integrated through standardized interfaces and data formats:

5. Data is ingested from various sources and stored in Apache Iceberg tables.
6. Apache Spark processes the data, performing transformations and enrichments.
7. Trino provides a SQL interface for querying the processed data.

This architecture enables a flexible, scalable data platform that can handle diverse workloads while maintaining component independence. Each component can be replaced or upgraded individually without disrupting the entire system.

The use of Apache Iceberg as the table format provides several advantages, including: – Version control for data, enabling time travel and rollback – Schema evolution, allowing the data model to adapt over time – Transaction support, ensuring data consistency – Partition evolution, optimizing query performance as data grows

These capabilities make NGODS a robust foundation for building data–intensive applications, from business intelligence to machine learning.

5.4. Future Directions and Emerging Trends

Building upon the implementation examples and comparative analysis, we now explore emerging trends and future directions in component–based big data architectures. These developments promise to further enhance the flexibility, performance, and maintainability of data platforms. Several emerging technologies hold promise for enhancing component–based data architectures:

Serverless Data Processing: Technologies like AWS Lambda, Azure Functions, and Google Cloud Functions enable fine–grained, event–driven processing without managing infrastructure. This approach can reduce operational complexity and improve scalability.

Unified Compute and Storage: Projects like Delta Lake and Apache Iceberg blur the line between storage and compute, providing table formats with rich processing semantics. This unification can simplify architecture while improving performance and data consistency.

Streaming SQL Engines: Tools like Materialize and Decodable enable SQL queries over streaming data, simplifying real–time analytics. These engines make streaming data more accessible to a wider range of users.

Data Contracts and Schemas: Schema registries and data contract frameworks formalize agreements between components, improving interoperability. These tools enable more robust integration between components.

Data Quality Frameworks: Tools like Great Expectations and Deequ help ensure data quality throughout the processing pipeline. These frameworks enable automated testing and validation of data at each stage of processing.

6. Conclusion

This paper has proposed a component-based architecture for big data systems that emphasizes modularity, interchangeability, and standardization. The architecture consists of four primary layers—ingest, process, expose, and storage—each with well-defined responsibilities and interfaces.

The architecture enables organizations to: - Select the best technologies for each layer based on specific requirements - Replace individual components as requirements change or technologies evolve - Implement various architectural patterns (Lambda, Kappa, Medallion) through appropriate component configuration - Scale components independently based on workload demands - Evolve their data platform incrementally without disruptive rewrites

The paper has demonstrated the practical application of this architecture through the NGODS example implementation, which combines Apache Iceberg, Apache Spark, and Trino to create a flexible, high-performance data platform[3]. This example illustrates how the component-based approach can be applied in practice, providing a foundation for organizations looking to implement similar architectures.

The component-based architecture presented in this paper represents an evolution in big data system design, moving from monolithic implementations toward modular, flexible architectures that can adapt to changing requirements and technologies. By emphasizing standardized interfaces, clear separation of concerns, and modular design, this architecture helps organizations navigate the complexity of modern data ecosystems while building systems that can grow and evolve with their needs.

As the big data landscape continues to evolve, with new tools and techniques emerging regularly, the ability to incorporate new capabilities without disrupting existing systems becomes increasingly valuable. The component-based approach provides a framework for this evolution, enabling organizations to build data platforms that are both robust and adaptable. This flexibility, combined with the performance and scalability benefits of modern big data technologies, positions organizations to derive maximum value from their data assets both today and in the future.

Declaration on Generative AI

The author(s) have not employed any Generative AI tools.

References

[1] Microsoft Learn, “Big data architectures - Azure Architecture Center,” 2025. [Online]. Available: https://learn.microsoft.com/en-us/azure/architecture/databases/guide/big-data-architectures

[2] Chen Cuello, Rivery, “Data Ingestion Architecture: A Comprehensive Guide for 2025,” 2025. [Online]. Available: https://rivery.io/data-learning-center/data-ingestion-architecture-guide/

[3] Z. Svoboda, “ngods: new generation open-source data stack,” 2022. [Online]. Available: https://zsvoboda.medium.com/ngods-new-generation-open-source-data-stack-48094aea2ba1

[4] Databricks, “What is a Medallion Architecture?,” 2025. [Online]. Available: https://www.databricks.com/glossary/medallion-architecture
[5] N. Kalra, “Big Data Architectural patterns - Lambda (λ), Kappa (κ) and Zeta (ζ),” 2022. [Online]. Available: https://www.linkedin.com/pulse/big-data-architectural-patterns-lambda-%CE%BB-kappa-%CE%BA-zeta-kalra
[6] Datavid, “Data ingestion architecture: The complete guide,” 2022. [Online]. Available: https://datavid.com/blog/data-ingestion-architecture
[7] M. Angelo, “ETL vs. ELT: Tools, Synergies, Advantages, and the Medallion Architecture,” 2025. [Online]. Available: https://www.linkedin.com/pulse/etl-vs-elt-tools-synergies-advantages-medallion-miguel-angelo-zjovf
[8] LinkedIn, “Data Platform Architectures & Design Patterns: A Comparative Analysis,” 2025. [Online]. Available: https://www.linkedin.com/pulse/data-platform-architectures-design-patterns-comparative-tfwoc
[9] Microsoft Learn, “Big data architecture style - Azure Architecture Center,” 2024. [Online]. Available: https://learn.microsoft.com/en-us/azure/architecture/guide/architecture-styles/big-data
[10] Kevin Bartley, Rivery, “ETL vs ELT: Key Differences, Comparisons, & Use Cases,” 2024. [Online]. Available: https://rivery.io/blog/etl-vs-elt/
[11] Software Architecture Academy, “Big Data Architecture Patterns | Lambda vs Kappa,” 2022. [Online]. Available: https://www.youtube.com/watch?v=waDJcSCXz_Y
[12] LumenData, “ETL Data Architectures - Part 1: Medallion, Lambda & Kappa,” 2025. [Online]. Available: https://lumendata.com/blogs/etl-data-architectures-part-1/
[13] Azure Databricks, “What is the medallion lakehouse architecture?,” 2024. [Online]. Available: https://learn.microsoft.com/en-us/azure/databricks/lakehouse/medallion
[14] Databricks, “Lambda Architecture Basics,” 2025. [Online]. Available: https://www.databricks.com/glossary/lambda-architecture
[15] Confiz, “What Is Data Ingestion in Big Data? Key Tools and Techniques,” 2025. [Online]. Available: https://www.confiz.com/blog/what-is-data-ingestion-in-big-data/
[16] DataForge, “ETL vs ELT: Key Differences,” 2013. [Online]. Available: https://www.dataforgelabs.com/data-transformation-tools/etl-vs-elt
[17] DZone, “Are Your ELT Tools Ready for Medallion Data Architecture?,” 2024. [Online]. Available: https://dzone.com/articles/are-your-elt-tools-ready-for-medallion-data-archit
[18] O. Zhyrenkov, A. Doroshenko “Elasticsearch for big geotemporal data”, 2025 Problems in Programming Number 1 (in print), ISSN: 1727-4907