

Scalable Data Platform Architecture for Highly Variable e-Commerce Workloads

Gautham Ram Rajendiran

Email: [gautham.rajendiran\[at\]icloud.com](mailto:gautham.rajendiran[at]icloud.com)

Abstract: *The exponential growth in the generation of data related to eCommerce has motivated organizations to seek big data architectures that scale both batch and real-time data processing [1]. This paper discusses an architecture that has fault tolerance, scalability, and low latency for data processing and was implemented in a large-scale eCommerce organization. This architecture ensures integration from varied sources, features data quality, lineage, and provides robust data storage and analytics. It entails design considerations for efficient utilization of cloud-native services, event-driven processing and task orchestration. We explain in detail the various components and brief on technical implementation details so that this can be a standard architecture that can be adopted to build eCommerce data platforms, thereby saving the time and effort in researching and designing such a platform from scratch.*

Keywords: Big Data, Scalability, eCommerce, Data Architecture, Data Processing, Distributed Systems, Cloud Computing, Data Pipelines

1. Introduction

Digital commerce has increased the volume, velocity, and variety of data. Traditionally, relational databases and monolithic architectures have proved to be inadequate in managing such a high volume of data [1]. Scalable and distributed architectures continue to gain ground as eCommerce platforms need batch and stream processing to keep pace with the demands of a personalized customer experience, real-time insight, and large-scale operations.

In this paper, we propose a big data architecture tailored for large-scale eCommerce platforms. This architecture utilizes distributed systems, cloud-native services and adaptive task orchestration to provide a resilient, low-latency and highly scalable data solution. This paper will cover some of the major components of the proposed architecture, including event handling, adaptive workload scaling, lineage tracking, and data warehousing using some of the contemporary design principles in Software Engineering and other open-source tools.

2. Literature Review

The rapid growth of data generation from various sources, including IoT devices, e-commerce platforms, and social media led to the rise of numerous data processing frameworks. Traditional data platforms like Hadoop [2] have been pivotal in processing large-scale batch data. These platforms provide scalable storage and compute capabilities, but they primarily cater to batch processing with high latency, making them less suitable for real-time data needs.

On the other hand, streaming platforms such as Apache Kafka [3] and Apache Flink [4] have gained prominence for handling continuous data streams at low-latency. These platforms excel in real-time analytics, event-driven architectures and supporting microservices, but they often lack efficient support for long-running batch processing or complex aggregations typically required for business intelligence.

To bridge the gap between batch and streaming data, modern frameworks like Apache Spark [5] introduced hybrid architectures through Spark Streaming. Although it offers support for both types of data, Spark's micro-batch processing model introduces latency in real-time processing, preventing truly instantaneous data handling. Similarly, Lambda Architecture [6] was proposed to combine batch and real-time processing pipelines, but its dual pipeline nature leads to complexity and maintenance.

Several studies highlight these limitations. Kiran et al. [6] discuss how maintaining dual pipelines increases system complexity, while Yadranjiaghdam et al. [7] note the difficulties in achieving exactly-once semantics across batch and streaming data in Lambda Architectures. This signifies the need for a unified platform capable of seamlessly handling both batch and streaming data with low latency, scalability and consistency.

3. Objective

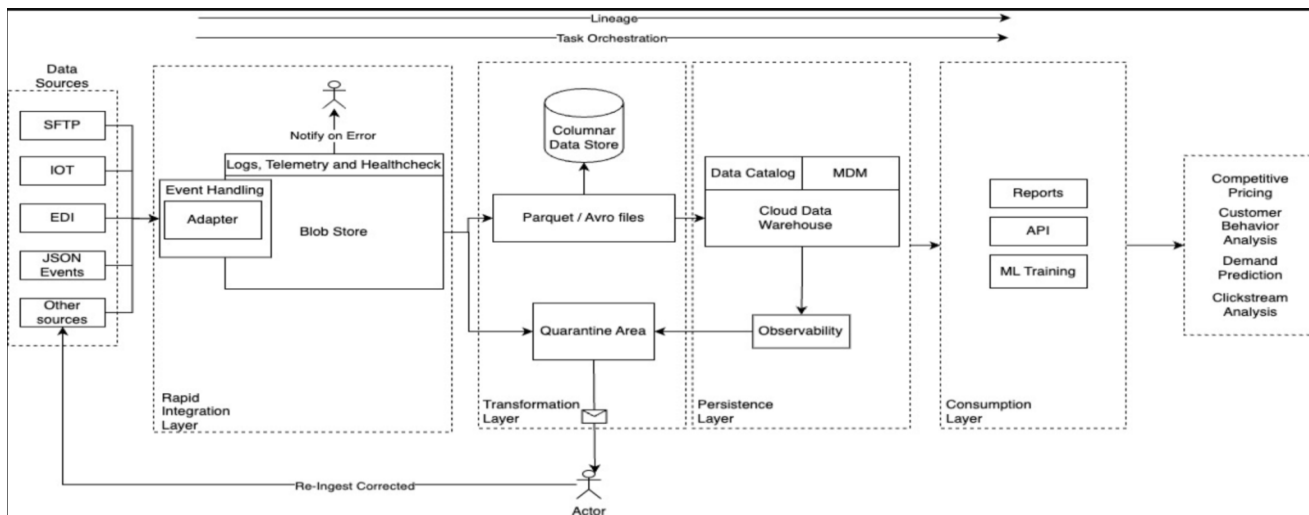
The following objectives are derived from challenges that were encountered in the production environment of an eCommerce system where data volume and velocity can have a wide range, reaction time to certain events need to be almost immediate, data quality issues and data audits are common.

Design for Massive Scalability: Ensure the architecture can seamlessly scale from minimal data loads to extremely large data volumes, adapting to the dynamic demands of eCommerce systems.

Real-Time Analysis: Enable mechanisms to provide both real-time and near real-time data insights, supporting business-critical decisions in high-velocity environments.

High Auditability and Data Quality: Establish robust mechanisms for ongoing data quality monitoring and automated correction to ensure high accuracy in data handling across various processes.

System Architecture

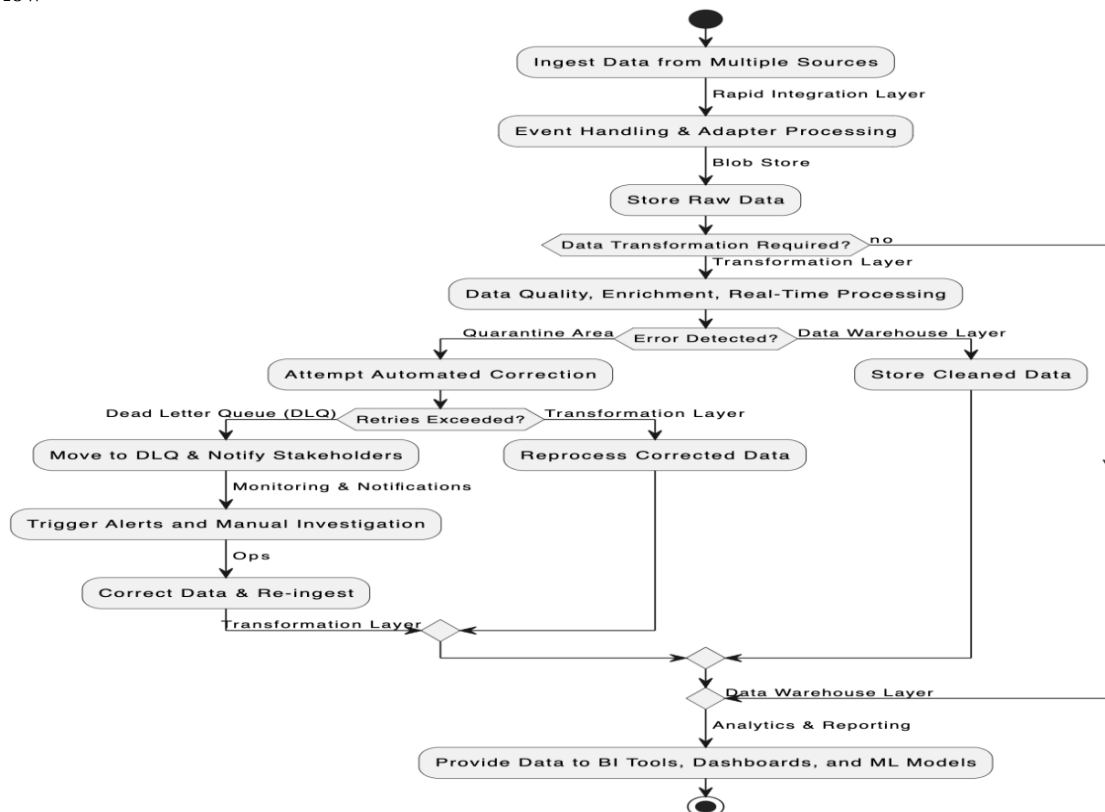


The System architecture depicted above covers all the components, processes and actors that comprise a data processing pipeline that can adapt to ingest various sources of data, process what is classified as big data [8] at scale and explain mechanisms that can be used for auditing, quality checking, storing and analyzing the same. Individual components and their interactions with actors are explained in detail in the sections below.

We will cover details on how some tools like Apache Airflow, Apache Hudi and Apache Spark come into play when working with data at scale. We will also discuss using the Publisher-Consumer architecture to deal with high velocity

data and also re-purpose the same to deal with smaller batches of data as such workloads are very common in e-Commerce systems. One of the important requirements of e-Commerce systems is tracking the lineage, as audits and compliance requirements are common in this domain, and in this architecture, we explain mechanisms that can be used to track data at a very granular level, for example a single data point or a row can be traced back to the ingestion source, transformations and to the data warehouse. Some of the mechanisms used to ensure data quality, on-demand data analysis and long-term storage mechanisms like data warehousing are also explored in the context of e-Commerce.

Process Flow



Components

Rapid Integration Layer

This layer is intended to ingest data from multiple sources that are classified as either streams or batch. It will guarantee ingestion of the data at speed and with high efficiency, regardless of the source type. The layer itself will deal with the preprocessing of data, ensuring its idempotency as well as taking care of metadata enrichment.

Adapter

The adapter component transforms and enriches the raw payload that the event handler receives. This logic is decoupled from the event handler, so it keeps the system flexible and adaptable to different data sources. The idempotency key logic is owned by the transformer since e-

Commerce sources can uniquely be identified only based on the content of the event. This is different from the usual definition of idempotency. Idempotency is an important feature for Lineage Tracking explained in other sections of this paper.

Technical Implementation

Common functionalities, such as telemetry and generating idempotent keys, are abstracted into the BaseAdapter class. To enable source-specific transformations and prepare per-source unique idempotency keys, adapters extending this class can do so.

The method `emit_metrics` allows the passing of telemetry data to monitoring and alerting systems.

```

from abc import ABC, abstractmethod

class BaseAdapter(ABC):
    def __init__(self):
        self._telemetry_service = TelemetryService()

    @abstractmethod
    def generate_idempotency_key(self, payload: Payload) -> str:
        """
        The transformer owns the idempotency key logic since a
        lot of cases depend on the underlying data in the payload
        to check idempotency. This is a crucial component
        which is used in the Lineage Tracking System explained in other sections.
        """
        pass

    @abstractmethod
    def transform(self, payload: Payload) -> TransformedPayload:
        pass

    def emit_metrics(self, data: TransformedPayload, metric_name: str) -> None:
        """
        A sample implementation that emits telemetry metrics for
        monitoring and alerting purposes.
        """
        self._telemetry_service.emit_metrics({
            "data_source": data.payload.source,
            "id": data.idempotency_key,
            "timestamp": data.payload.timestamp,
            "metric_name": metric_name,
            "metric": "count",
            "value": 1
        })

```

Event Handler

This component is in charge of routing and managing incoming events. It handles retries, failover mechanisms, and guarantees at-least-once delivery. The next code snippet can be used as an example to extend this event handler to deal with other types of sources. The base Payload model may go on to further extend to suit various source formats like Json, EDI, or even SFTP. This flexibility, where any real-world event can be modeled as a Payload Object, will ensure that the event handler has specialized implementations catering to any kind of data event, much like what is observed with Event Sourcing and Domain-Driven Design [10].

Technical Implementation

The EventHandler class is designed as an abstract class with customizable event handling logic.

The getConfig method can be extended to pull configuration details from external services like AWS AppConfig or DynamoDB. This can be useful to implement business logic based on properties of the payload. The decoupled configuration can also be used to determine the type of nodes that may be required to handle the event.

The processEvent method manages the core event processing logic, including transforming the payload and handling any errors that may occur.

```

from dataclasses import dataclass
from abc import ABC, abstractmethod
from datetime import datetime
from typing import List, Dict, Any, Optional

@dataclass
class Payload:
    schema_version: str = "v1"
    source: str
    timestamp: datetime
    data: List[Dict[Any, Any]]

@dataclass
class TransformedPayload:
    schema_version: str = "v1"
    idempotency_key: str
    payload: Payload

class EventHandler(ABC):
    def getConfig(self, payload: Payload) -> Dict:
        """
        Retrieves configuration for processing the payload,
        such as scaling properties,
        memory, instance types, and other runtime configurations.
        """
        pass

    @abstractmethod
    def handleEvent(self, payload: TransformedPayload) -> Any:
        """
        Main event handling logic that routes the transformed event downstream.
        """
        pass

    def processEvent(self, payload: Payload):
        try:
            config = self.getConfig(payload)
            transformed_event = self._adapter.transform(payload)
            self.handleEvent(transformed_event)
        except Exception as e:
            # Error handling logic
            print(f"Error processing event: {e}")
        else:
            # Success logic
            print(
                f"Successfully processed event:
                {transformed_event.idempotency_key}")

```

Kappa Architecture

The Kappa architecture [9] considers systems where all data processing is treated as a real-time stream; batch processing is just a particular case of stream processing. The Event Handler adapts the Kappa architecture principles to deal with data of varying volume and speed.

Reasons to apply principles from Kappa architecture to the Event Handler

Real-Time and Batch Coexistence: While this architecture will favor stream-first design, it does allow handling batch workloads in processing, if required. Such data, when it arrives in bulk, can be ingested and processed in a manner consistent with real-time events—to preserve data lineage and integrity.

Event-Driven Adaptability: It can be seen that, based on the event types, the event handler would change dynamically. For example, schema validation and metadata enrichment can be applied differently to an incoming event from an SFTP source than to a JSON event coming from an API stream. This adaptiveness of the architecture makes it robust for different use cases.

Source-Agnostic Ingestion: The event handler will execute the same logic for transformation, agnostic of multiple input

sources: SFTP, IoT, or API events. Each source will be enriched, standardized, and tagged with unique identifiers like source name, filename, and timestamp.

Idempotency and Deduplication: Since every event gets a unique id at the time of ingestion, idempotency is taken care of by the event handler. This is very critical where data might arrive multiple times due to retries, network issues, or source errors. The transformation step will make sure to check for consistency; hence, no duplicate record passes downstream.

Scalability: The Event handler could be deployed using Kubernetes, a container orchestration *solution*, to support the varying volume of data usually experienced in e-Commerce. This will enable the event handler to scale up or down, and allow flexibility to choose between mechanisms to prioritize high volume or throughput.

Lineage Tracking System

The Lineage Tracking System is an important feature for data integrity and full transparency across complex data pipelines. In an e-commerce environment, data flows from multiple sources; therefore, there should be clear visibility into how every data point evolves from its ingestion to its final consumption.

During ingestion, the system assigns a unique identifier to each record. Most often, it is a concatenation of source name, file name, and timestamp, which will then make the record uniquely traceable across all pipeline stages. More complex systems could use dedicated Metadata management, to add more attributes to the unique request. However, the same unique identifier stays along with the data across all the transformations—from an initial cleaning and enrichment to storing columnar formats like Parquet and further into data warehouses.

Technical Implementation

The lineage tracking metadata is stored with data itself at every point. It can be integrated into systems like Apache Atlas, AWS Glue Data Catalog, or even a custom metadata store. This system can be visualized as a graph where nodes represent steps of data transformation, edges of the flow of data along with its unique ID. This type of graphical system makes it easy to visualize and understand the complexity of data flows in real time for teams.

The lineage tracking system provides great support not only in operational efficiency but also in strategic confidence via data-driven decisions. Whether for compliance, debugging, or performance optimization, a state-of-the-art lineage tracking system lends an eCommerce platform the auditability and accountability that evolving data challenges demand.

Transformation Layer

The transformation layer is the backbone of the data pipeline and takes raw, unstructured, or semi-structured data, transforming it into clean, structured data ready for storage in a data warehouse or direct consumption by analytics. Here, in the context of eCommerce, where data comes from multiple heterogeneous sources such as sales transactions, clickstream logs, customer interactions, and supplier data, the transformation layer has an important place to guarantee data consistency, accuracy, and usability.

Why is the Transformation Layer Necessary?

Data Quality Assurance: High-quality data is critical to eCommerce, wherein the wrong decisions are made, and customers either endure poor experiences or the operations become inefficient. The transformation layer will conduct a variety of data quality checks, including deduplication, filling missing values, and applying business rules. It will ensure, for example, that every order event contains the fields for a customer ID and product SKU, discarding or quarantining any that do not fulfill such checks. This degree of rigor at the time of transformation greatly reduces the potential for errors to propagate downstream.

Data Enrichment and Improvement: Data enrichment adds context and value to raw data by combining the data with additional information. This can be metadata attached in eCommerce, like geographic locations based on IP addresses or product categorization based on machine learning models. For instance, clickstream data that is raw can be enriched with session identifiers, user demographic data, and product attributes, setting the data up for further and meaningful analysis later on. The transformation layer is where all this enrichment of the raw data happens to present actionable information.

Distributed processing frameworks like Apache Spark and orchestration tools such as Apache Airflow running on Kubernetes are used by the Transformation Layer. These technologies enable the dynamic distribution of workloads, making it possible for velocity streams—even of the highest order—to be processed without bottlenecks in near real-time. For instance, an eCommerce platform may receive millions of click events per hour during a Black Friday sale. The transformation layer processes these events in real time, filtering, aggregating, and making data ready for real-time dashboards that track user behavior.

Technical Implementation:

Airflow and Task Orchestration

Airflow is typically used for the orchestration of the transformation pipelines. DAGs represent a series of transformation tasks that are to be executed in sequence. Every task is independent, which makes scaling and retries easy. The tasks are distributed across clusters using Kubernetes Pod Operators, scaling dynamically as workload demands require. This is where the blob store artifacts produced by the event handler can be consumed for batch transformations.

Real-Time “Big” data processing

In the case of high-velocity data, this transformation layer could leverage Apache Spark Streaming or Apache Flink for real-time processing, where time-critical events are involved. This could be user activity at times like flash sales. In this architecture, the event handler acts as an in-house message queue adhering to the Publisher-Subscriber pattern. Instances of an event handler play the role of publishers, while some kind of consumer service subscribes to these events. It makes use of one of the well-known patterns, such as Long Polling, which allows the Consumer to process events in real time.

```

from airflow import DAG
from airflow.providers.cncf.kubernetes.operators.kubernetes_pod import KubernetesPodOperator
from datetime import datetime

dag = DAG(
    dag_id='ecommerce_transformation_pipeline',
    start_date=datetime(2024, 8, 16),
    schedule_interval='@hourly'
)

transform_task = KubernetesPodOperator(
    namespace='airflow',
    image='transform-image:latest',
    cmds=["python", "transform.py", "--blob-location", ""],
    name='transform-task',
    task_id='transform_task',
    get_logs=True,
    dag=dag,
)

```

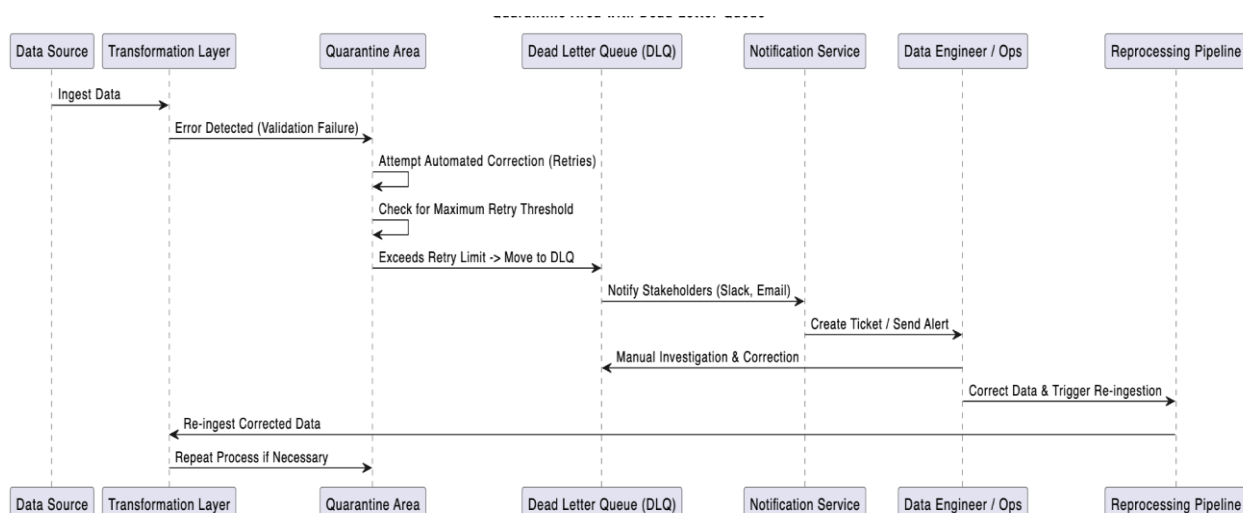
Operational Data Analysis

The result of the transformation layer will be stored in special file formats like *Apache Parquet* or *ORC*, which are designed with an aim to reduce as much latency connected with data access as possible. All of these formats compressed column by column making a query faster for large-scale in-memory analytics. This kind of data can be analyzed using big data processing frameworks such as Apache Druid, or Apache Hudi. The main aim of this component is to enable fast data summary by efficient partitioning strategies. This is mainly useful for operational data analysis where large amounts of data needs to be analyzed in near real-time to enable faster

responses from Marketing and Sales teams in reaction to changes in customer behavior and inventory sales.

Quarantine Area

The quarantine area is primarily responsible for data integrity by catching and isolating all records that fail transformation—for example, validation errors, missing fields, or format inconsistencies. To make this fault-tolerant—to guarantee that no data will be lost while it is being processed—the quarantine area can be combined with a Dead Letter Queue, which is a standard pattern in dealing with records that continue to fail after reprocessing attempts.



Technical Implementation

Error Detection and Initial Quarantine

All records failing transformation are sent to a quarantine area for inspection and editing. If the error is a common problem with known resolutions to that problem, then the record is edited to correct the problem and re-ingested into the pipeline.

However, if an error continues to manifest or is one that requires complex handling, then, after a certain number of reprocessing attempts, it will be moved to a dead-letter queue. Example: An order record missing the mandatory field `customer_id` would first be quarantined. If the attempts at automated correction fail more than once, then the record is sent to the DLQ for further investigation.

Dead Letter Queue (DLQ) Implementation

The DLQ works in a manner similar to a persistent store for all those records that cannot be processed after several retries. Removing such problematic records into the DLQ ensures you do not clog the pipeline with them; at the same time, you will have them saved for further analysis or manual correction. Traditionally, DLQs are combined with message brokers like Amazon SQS, Kafka, and Google Pub/Sub, which are in charge of retry logic and message persistence. In this architecture, the retry and failover mechanisms would be contained within the EventHandler implementation. Example: A batch of records from an unreliable data source fails validation repeatedly. Rather than trying and failing to process these records again and again, they go into the DLQ, where they can be inspected and repaired manually or resolved by correcting upstream data ingestion processes.

Error Categorization and Routing Logic

Before any record is sent to the DLQ, quarantine categorizes the errors by type, such as missing data, invalid format, and out-of-bounds values. This grouping will help in diagnosis of the root cause and help decide whether reprocessing should be automatic or to route directly to the DLQ. Example: In case of format inconsistencies in a high percentage of records from one source, the system will automatically mark this source as inappropriate and suspend further ingestion from it; then, quarantined records will be redirected to the DLQ.

Manual Review and Escalation from the DLQ

Records in the DLQ require manual intervention, typically from data engineers, operations teams, or business stakeholders. The DLQ maintains an exhaustive log and metadata of each record that has been problematic, and root cause analysis is hence very simple to do. Furthermore, integration with workflow management tools like Jira or ServiceNow allows the tracking, escalation, and resolution of issues in a very organized way. Example: An EDI file with several errors could be escalated through a ticketing system to an operation team, who can correct the data manually and trigger the re-ingestion of corrected records back into the pipeline.

Automated Alerts and Notifications

The DLQ is integrated into monitoring and alerting systems that convey to stakeholders, through alerts and notifications, whenever records are added to the queue. Alerts can then be set on thresholds, such as when there is a sudden spike in entries into the DLQ, thus pointing toward a possible systemic issue that warrants immediate attention. Example: If the count of records in the DLQ exceeds the threshold, Slack will send an alert to the data engineering team to investigate immediately.

Data Warehouse Layer

This layer serves as the backbone of analytics and reporting for eCommerce platforms. A data warehouse holds transactional and analytical data models that are particularly optimized to query and report. Normally, they contain dimensional models and star schemas which help in running OLAP workloads efficiently. The warehouse becomes a single, focused location for business intelligence activities to pull data from multiple sources, including sales transactions, customer interaction, and logistics data. A modern warehouse

is set on a cloud-native platform like Snowflake and Amazon Redshift that offers scalable, distributed storage and processing. This scalability is critical to e-commerce because of the high transaction rates and large data sets that may result in rapid growth in volume. It also hosts ETL processes that clean, enrich, and transform data before consumption by downstream applications. In addition, the data warehouse provides advanced analytics, real-time reporting, and machine learning applications with reliable and high-quality data.

Technical Implementation

Star and snowflake schemas optimize the designs for OLAP workloads that support the analytical query and reporting needs. These models structurally place the data into fact and dimension tables and assist in today's data aggregation and filtering of huge datasets. Star schemas are preferred most of the time as they are simple and faster, whereas snowflake schemas further normalize the dimension tables for better data integrity and working. All such models are optimized techniques of business intelligence tools and data marts where querying is to be fast and ad hoc. In the field of e-commerce, dimensional models apply to analysis on the metrics that reflect the trend of sales, customer behavior, and the effectiveness of marketing campaigns through the segmentation of data across several dimensions, such as time, geography, and the set product categorization of sales in a given period of time. These techniques become highly relevant in streaming data systems because of the likelihood that events will arrive out of order or with variable latencies. Late-arriving data in an e-commerce setting might include delayed transaction logs, inventory updates that are out of sync, or backlogged shipping records.

4. Result

The adopted architecture is designed with one fundamental principle - "What works for one should work for one million" - and hence adopts best practices from designing microservice architectures, horizontal scaling as opposed to vertical scaling and best practices for data partitioning. With this as the operating principle, this architecture is designed to scale from handling kilobytes to petabytes of data, as such variance is very common in eCommerce systems. It has also been observed to provide accurate real-time and near real-time insights, mechanisms to handle data audits in highly sensitive eCommerce environments such as hazardous / classified inventory sales, mechanisms to scale-out as much as infrastructure permits and robust data quality monitoring and correction strategies.

5. Conclusion

The proposed architecture stems from various experiences through trial and error while working with data in large scale eCommerce environments where the volume and velocity of data has wide lower and upper bounds. Any version of a data platform that builds on top of this architecture will resolve the issues with respect to scalability in modern eCommerce data engineering. It incorporates features like fault tolerance, data quality monitoring and auditability, which are highly desirable features of a data pipeline in an eCommerce environment. It also sets up the infrastructure to derive

insights from data at scale through distributed systems and cloud-native services. This allows engineers to focus on implementation and saves much time due to research and design of such a platform. Future works should focus on fine-tune cost optimization strategies and integrate AI-driven automation for better data-processing capabilities.

References

- [1] A. Gandomi and M. Haider, "Beyond the hype: Big data concepts, methods, and analytics," *International Journal of Information Management*, vol. 35, no. 2, pp. 137-144, 2015. doi: 10.1016/j.ijinfomgt.2014.10.007.
- [2] Borthakur, D.: HDFS architecture guide. Hadoop Apache Project, 1–13 (2008). https://hadoop.apache.org/docs/r1.2.1/hdfs_design.pdf
- [3] Apache Kafka. <http://kafka.apache.org>.
- [4] Apache Flink. <https://flink.apache.org>
- [5] Apache Spark. <https://spark.apache.org/>
- [6] M. Kiran, P. Murphy, I. Monga, J. Dugan and S. S. Baveja, "Lambda architecture for cost-effective batch and speed big data processing," 2015 IEEE International Conference on Big Data (Big Data), Santa Clara, CA, USA, 2015, pp. 2785-2792, doi: 10.1109/BigData.2015.7364082.
- [7] Yadranjiaghdam, Babak & Pool, Nathan & Tabrizi, Nasseh. (2016). A Survey on Real-Time Big Data Analytics: Applications and Tools. 404-409. 10.1109/CSCI.2016.0083.
- [8] S. Sagiroglu and D. Sinanc, "Big data: A review," 2013 International Conference on Collaboration Technologies and Systems (CTS), San Diego, CA, USA, 2013, pp. 42-47, doi: 10.1109/CTS.2013.6567202.
- [9] Lin, Jimmy. "The lambda and the kappa." *IEEE Internet Computing* 21, no. 05 (2017): 60-66.
- [10] Uludağ, Ö., Hauder, M., Kleehaus, M., Schimpfle, C., Matthes, F. (2018). Supporting Large-Scale Agile Development with Domain-Driven Design. In: Garbajosa, J., Wang, X., Aguiar, A. (eds) *Agile Processes in Software Engineering and Extreme Programming. XP 2018. Lecture Notes in Business Information Processing*, vol 314. Springer, Cham. https://doi.org/10.1007/978-3-319-91602-6_16