AMERICAN ACADEMIC PUBLISHER

INTERNATIONAL JOURNAL OF DATA SCIENCE AND MACHINE LEARNING (ISSN: 2692-5141)

Volume 01, Issue 01, 2024,
Published Date: - 07-04-2024
pages 11-36

Event Sourcing for Retail Inventory: Kafka + BigQuery Real-Time Analytics

Sandeep Reddy Gundla
Lead Software Engineer, MACYS Inc, GA, USA

Abstract

The retail business is experiencing difficulties in maintaining accurate inventory levels with low latency because of old batch processing systems. The proposed solution is a real-time application that uses the data stream architecture based on event-sourcing with Apache Kafka and Google BigQuery to provide streaming analytics. The immutability of the events makes it possible to reliably stream to Kafka such events as sales, returns, and stock adjustments. The scalability and fault-tolerant design of Kafka and the fact that BigQuery is serverless ensure a persistent foundation and real-time analytics in a sub-second query resolution. Significant improvements were the elaborate schema design, optimal Kafka topic setup, and introduction of real-time dashboards to gain insights into inventory. The results also show less than one-second update latency up to 100,000 events per second, and cost-efficient benchmarks of cloud infrastructure. The solution improves the real-time monitoring of the stock, thus reducing instances of out-of-stock and facilitating better decision making. The possibilities are immense for retail operations, providing better control of inventory management, pricing, and forecasting of demand. This model represents a significant step away from legacy batch ETL processes towards having the correct data as needed within retail to support a high-velocity, globalised sector. Future efforts will be dedicated to multi-region deployment, automatic schema change, and incorporating edge computing in offline shop scenarios.

Key words*: Event Sourcing, Kafka, BigQuery, Real-Time Analytics, Retail Inventory.*

1. Introduction

In contemporary retailing activities, charting the most up-to-date and precise level of inventory is crucial in addressing customer demand, managing shrinkage, and ensuring effective supply-chain management. Traditional inventory management systems are also oftentimes plagued with data staleness: the inventory levels shown in a centralized database are outdated concerning what is going on in another part of the world, like a point-of-sale, stock movement, or a new shipment inwards. Such latency can cause overselling, out-of-stock, and unfulfilled customers. There is also the issue of concurrency, where multiple services or locations of a store might be trying to update duplicate stock records. Absent a sound means of serializing or reconciling multiple simultaneous writes, conflicting updates may overwrite valid values or necessitate manual intervention, which is expensive and time-consuming. With the diversification of the retail ecosystems into online and physical stores, the problems of retention of coherent, consistent, and real-time inventory state are becoming increasingly painful.

Traditionally, retail analytics has deployed batch Extract, Transform, Load (ETL) to merge the data on a transactional

basis into the centralized data warehouses. A typical example of a workflow is point-of-sale and warehouse systems storing transaction logs in staging tables every few hours. The records are then loaded into a fact table for reporting purposes. Although such a solution makes it possible to research trends on a daily or even hourly basis, it poses a considerable time lag between the actual instances of inventory changes and the corresponding changes in analytics solutions. Delays of several tens of minutes to a few hours imply that essential decisions (such as repricing during flash sales or relocating supplies to areas with high demand) are made based on out-of-date data. Moreover, staged ETL pipelines will need a lot of orchestration overhead, complicated dependency management, and preparation to maintain them to accommodate schema change, data quality problems, and recovery.

Event-sourcing is another form of architecture where each state change is reflected as an immutable event and appended to an event log. Instead of creating updates to a mutable record in-place, the system stores separate data events, including InventoryAdded, SaleRecorded, or StockAdjusted. The present state of any entity can then be determined by replaying or aggregating its event history. The design offers several significant benefits: a complete, auditable account of all inventory-related operations; simpler recovery and temporal querying through replaying events at any point in time; and the ability to support flexible and cross-shard read models by projecting different variations (i.e., per-SKU, per-location, or global inventory) without affecting the source of truth. Event-sourcing also lends itself to concurrency control effortlessly due to the append-only nature of the storage, thereby allowing the resolution of conflicting operations at the time of ingestion.

Apache Kafka is a distributed streaming system designed to make real-time event reads and writes at scale. Kafka supports ingestion of millions of events per second and scalability through horizontal additions of brokers and partitions to seamlessly match the storage needs with the available resources. Its retention model, based on logs, in addition to allowing exactly-once semantics to be observed, fits well with event-sourcing concepts. Google BigQuery is a companion to Kafka and can be used to provide a massively parallel data warehouse as a service, supporting ANSI SQL serverless ad hoc querying and dashboarding. Since 2018, BigQuery has supported streaming data via the BigQuery streaming API or integration with Dataflow (Apache Beam), through which event data can be analysed immediately. Kafka and BigQuery are a powerful combination that can enable an event-sourcing-based pipeline, which provides sub-second information at scale without manual infrastructure provisioning.

The latter part of this paper is structured in various sections. The Literature Review discusses the prior research in the domains of event-driven architectures, change data capture, and streaming analytics, both in academia and industry settings. The methods and techniques section provides implementation details of the system architecture, such as event schema, Kafka topic configuration, partitioning logic, or the use of exactly-once guarantee in such an architecture provided by Apache Beam on Google Dataflow. The study also provides an exploration of data and visualization of plots that highlight the throughput levels, event distributions, and latency profiles of our test environment. It gives information on data transformation techniques and aggregation techniques to maintain a continuous, up-to-date view of the current stock levels and techniques of dealing with late and out-of-order events. The Real-Time Analytics Techniques section discusses the results of the benchmark experiments comparing the end-to-end latency, throughput, resource utilization, and cost, as well as the price per million events of our pipeline to that of the benchmarks. Results are given across the range of event volumes. The study also offers a discussion of our findings, practical implications of our development for the retail operations, trade-offs involved in our design decisions, as well as limitations encountered. It concludes by providing directions on where our future research should move, such as multi-region deployments, more advanced forecasting integrations, and schema evolution mechanisms.

2. Literature Review

2.1. Event-Sourcing and CQRS Patterns in Distributed Systems

Event-sourcing is a method whereby the entire state of an application is made up of a sequence of events that describe the changes made to the application. An event-sourced system captures not just the current state but rather all of the state transitions in chronological order. Such a way of doing has high auditability since the whole history of state changes can be observed after all; it also eases temporal sorts and rollbacks of the system by executing events to replay the past. Event-sourcing is a storage mechanism used in large-scale distributed systems wherein data mutation is decoupled with data storage and services append events into a log without needing to take exclusive locks or coordinate with central instances. The inherently distributed nature of this append-only model makes it a natural fit in any distributed messaging system, with the potential to be horizontally scaled to ingest and process high rates of event stream traffic.

CQRS is used in conjunction with event-sourcing to separate the write path (commands) and read path (queries). In CQRS, commands, requests to change state, are served by a write model that sends out events, and queries, requests to read state, are served by one or more read models that are optimized to read state (12). The separation allows both sides to scale and optimize independently: it is possible to tune the write mechanism to prioritize durability and consistency of writes, and the read mechanisms can optimize towards faster and less reliable querying. In practice, CQRS, as an implementation of event-sourcing, enables retail inventory systems to support concurrent inventory operations (sales, returns, restocking) by publishing internal events to a shared log, and having multiple read models (each devoted to different analytical or operational tasks) read and project the current inventory state.

In an Event Sourcing and CQRS architecture, as shown in Figure 1 below, writes and reads are decoupled, as all state-changing commands are persistently stored as immutable events within an append-only Event Store. Consumers of the Event Store subscribe to these events to update optimized read databases that contain the current state, and independent processors that replay events to rebuild a picture of the state at a particular point in time or to roll back. This pattern allows read/write separation as command handling is decoupled with query projections, which support query operations, giving the benefits of horizontal scale, optimization of write durability and read performance, supports full auditing, and allows temporal navigation of the entire history of system changes.

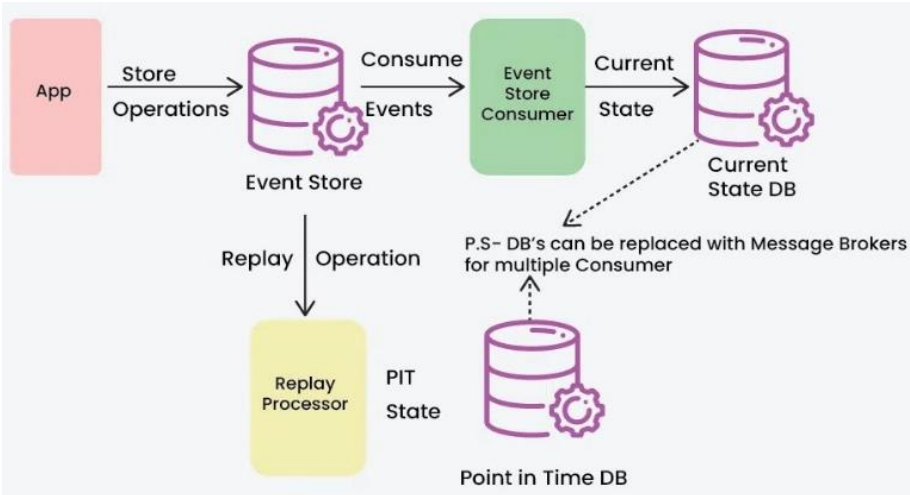

Figure 1: Immutable Event Store with CQRS: commands append events to an append-only log, which consumers replay to project both the current state and point-in-time views

Adopting a CQRS with an event source poses some difficulties when the application runs on distributed platforms. Ordering, idempotency, and cross-partition consistency need to be handled cautiously. Systems commonly approximate such systems by having idempotent consumers and checkpointing so that events are only consumed once (28). Out-of-order events, which may happen when parts of the network are disconnected or when parallel writes are used, necessitate that event processors buffer or otherwise reorder events based on timestamps or sequence numbers. The implementation of a variant schema of events should also be considered: business requirements on events are changing, thus the event definition should change, and schema evolution should be supported, as well as compatibility should be assured. Regardless of these complexities, the paradigm of event sourcing combined with CQRS has effectively been used in the area of fine-grain auditing tools, applications that must be replayed, and those that need to propagate the current state in real time, which is a key requirement in the retail inventory management context.

2.2. Kafka as a Durable, Distributed Log

Apache Kafka is a low-latency, distributed & scalable stream processing system that guarantees the durability of streams. Kafka is typically used as a robust distributed commit log: a Kafka cluster contains topics, which are divided among cluster nodes, and each topic is an ordered, immutable set of messages (37). The producers attach messages to partitions, and Kafka stores these messages persistently on disk with configurable replication factors. Reading of messages is done by the consumers at varying rates, with the reading tracking of the offsets to provide fault recovery and replay. The architecture allows Kafka to facilitate both pub/sub and queuing semantics, which is the reason why it can be used as the backbone of event-sourced architectures.

The resilience of Kafka is due to its write-ahead log and the retention policies that can be configured. Kafka replicates the data of the partition to many brokers so that data loss cannot occur due to the failure of a broker (32). To provide durability, the producer can configure acknowledgments (acks) on them: when using acks=all, a write is not acknowledged until all in-sync replicas have persisted the message. This extreme durability and high throughput sequential disk writes and zero-copy transfers make Kafka capable of supporting hundreds of thousands of events per second per broker cluster. In addition, Kafka incorporates log compaction, which also enables topics to be backlogged using a message key under the concept of keeping only the last record of a specific key. That is very convenient in stateful event-sourcing, where the reconstructed latest snapshot state of an aggregate is the result of reading the compacted event log.

With distributed inventory, you can more easily scale horizontally with Kafka and its partitioning as well as consumer groups functionalities. They can be parallel producers and consumers partitioning events such as inventory items or by warehouse location, and operate independently on disjoint partitions; high concurrency is achieved. Consumer groups allow multiple instances of a processing application to work together, sharing the load: only a single consumer instance works on any particular partition at a time, and this can achieve both fault-tolerance and parallelism. Kafka has an extended ecosystem to extend its usage: Kafka Connect supplies connectors to read data files and databases, as well as to write to both data warehouses and third-party services (6). Streams and ksqlDB offer in-process SQL-like processing features, respectively. These modules make up event-sourcing pipelines with ease of creation, allowing for the transformation, aggregation, and materialization of inventory events in real-time.

2.3. Streaming Analytics with BigQuery and Similar OLAP Stores

Google BigQuery is a completely managed and serverless data warehouse that is optimized for large-scale analytical queries. In contrast to a row-based OLTP database, BigQuery employs a columnar database with massively parallel processing (MPP) to scan petabytes of data at high speed. It is compatible with the ANSI SQL dialect and has been integrated to work with Google Cloud Storage, Dataflow, and Pub/Sub features; hence, it is mainly used for

streaming and batch analytics (25).To perform real-time analytics, it is possible to load streaming inserts with BigQuery using Streaming API or connectors like Kafka Connect so that the incoming data is written to a special streaming buffer before committing to a base storage table.

As depicted in the figure below, a fully managed, serverless BigQuery architecture consumes data in streaming inserts or free bulk loads to a replicated, distributed columnar storage tier with eleven-nines durability. This storage is mined by a high-availability Dremel compute cluster over a petabit network, via a distributed memory shuffle tier that allows it to run ANSI-SQL:2011-compliant, dauntingly large-scale analytical queries to challenge petabyte scale. Delivered over REST API, CLI, Web UI, or client libraries in seven languages, and with built-in integrations with Cloud Storage, Dataflow, and Pub/Sub, this architecture means real-time and batch analytics can be performed with no infrastructure to manage.

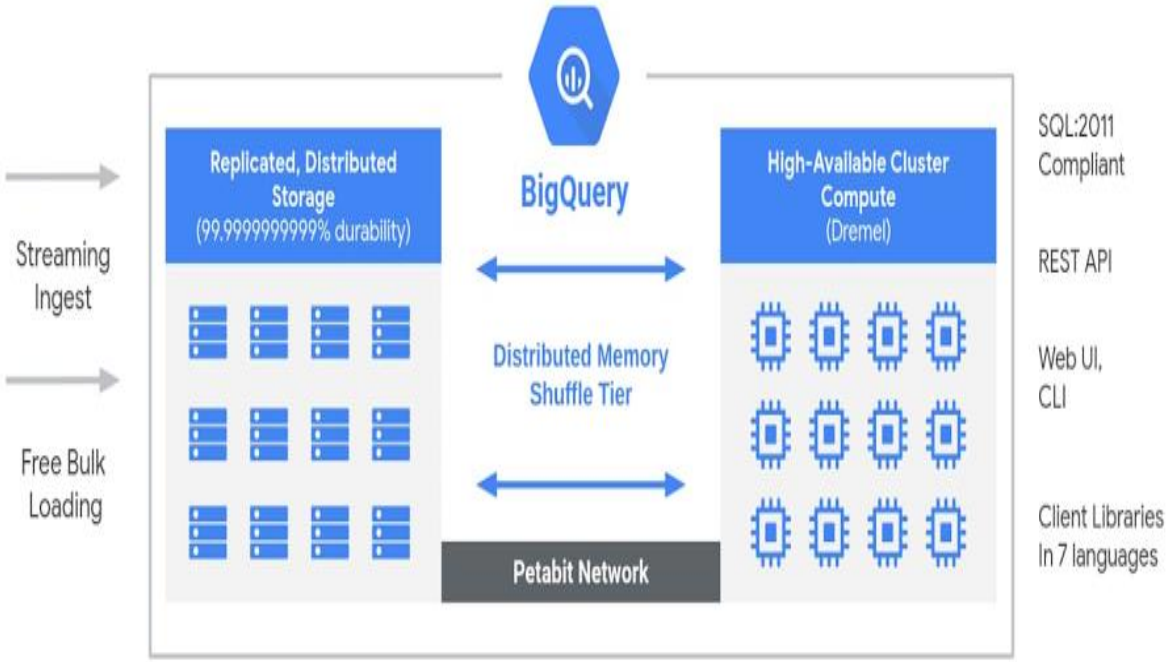

Figure 2: Serverless MPP columnar warehouse with streaming and batch ingest and Dremel compute for ANSI-SQL analytics.

The main point BigQuery meets many of the aspects and needs of real-time event-sourced analytics is that it can have low-latency ingestion with near-interactive query times. The streaming buffer maintains in memory recent inserts and flushes them back to columnar storage periodically; queries can access any row in the streaming buffer and the base table in a transparent way. Such architecture lowers sub-second visibility to new events, allowing operation dashboards to show the latest states of the inventories. Moreover, BigQuery allows time-partitioned and clustered tables that are optimized for efficient storage and query access on time-series and high-cardinality data common to inventory event streams. Materialized views are one way of speeding up repetitive queries by keeping precalculated answers updated on a per-event basis (24).

Streaming OLAP stores, including Amazon Redshift, Snowflake, and Apache Druid, are similar, but provide archival ingest and query bits, in addition to more recent activities as well. Kinesis integration of Redshift will offer real-time data ingestion, whereas continuous data loading via Snowpipe is available using Snowflake (1). Druid, with its real-time batch ingestion, is very effective at low-latency roll-ups and approximations via sketches. There are tradeoffs, however, with each one of these approaches as to their operational complexity, limits to that organization can scale, and the total cost of ownership. BigQuery is a serverless architecture where the cloud provider handles any

infrastructure concern, and automatically scales both upwards and downwards depending on query workload and assigns different workloads into isolated silos using slot reservations. This flexibility eases adoption for organizations that lack in-house infrastructure expertise, and thus, BigQuery is a good option for real-time retail inventory analytics.

2.4. Existing Retail Inventory Solutions (Batch vs. Stream)

Batch ETL processes have been traditionally used with the inventory systems of retailers. In this type of architecture, transactional data is collected from point-of-sale (POS), warehouse management systems (WMS), and e-commerce systems periodically. This data is then transferred to the harmonized schema and business logic and inserted into a central data warehouse. This process is often done on an hourly or daily basis. Batch processes have the advantage of established ETL tools and deterministic performance at the expense of long latency, generally measured in hours, creating potentially costly stale inventory images that can be misused to stock out, overstock, and/or not fulfill the order most optimally. Additionally, batch pipelines typically necessitate human effort to manage schema updates, slow data drift monitoring, and high retry costs in the event of failure.

To get out of the limitations of batch ETL, most retailers are now updating near-real-time via micro-batches (e.g., every five minutes). The architectures of micro-batch will decrease latency relative to daily jobs, but they still cause latency costs and computational overheads. Incremental pipelines need vigilant watermarking and state checkpoints to evade duplicates or gaps. The problem with out-of-order events, which is typical in distributed point-of-sale networks, makes micro-batch logic difficult since late-arriving events may require backfill dataflow or corrective joins in the data warehouse.

An event-sourcing mindset fosters solutions to these problems by thinking of inventory actions (sale, restock, return) as events, emitted and processed in real time. Such open-source initiatives as Apache Hudi, Apache Iceberg, and Delta Lake employ transaction logs to store copies of tables (35). However, they are focused on use cases of large-scale batch processing instead of providing low-latency operational views. There are commercial SaaS products: Salsify / StreamSets Data Collector, which, instead of just giving schema enforcement and pipeline orchestration, rely on underlying batch or micro-batch frameworks. On the other hand, an end-to-end streaming stack of Kafka, stream processor (e.g., Flink, Dataflow), and OLAP store has sub-second end-to-end performance and allows operational dashboards and automatic alerting based on inventory levels.

2.5. Gaps: End-to-End Implementations, Cost/Performance Studies

Although event streaming systems are theoretically advantageous, there is a lack of practical implementation guidance on how to use them, end-to-end, in retail inventory. Current case studies attempt to study specific parts of the pipeline, such as the design of Kafka clusters and testing larger queries using BigQuery, but fail to highlight the challenges of integrating different parts. Some areas that are still lacking are best practices around schema evolution when the event type evolves, strategies for performing data backfills without taking systems down, and techniques to ensure transactional consistency between Kafka and the rest of the world. Furthermore, there is hardly any well-documented reference architecture that illustrates the design of resilient, multi-region inventory pipelines.

Cost performance tradeoffs have not been thoroughly studied in real-life retail inventory in use cases. Although both Kafka and BigQuery provide performance metrics with a specific set of parameters, setting aside end-to-end tests (i.e., the acquisition of events to query results) at the production level (e.g., more than 100k events per second) is not documented in most cases. In addition, the costs of streaming versus batch in terms of infrastructure costs, effort required, and speed-of-access latency advantages have not been well documented (17). Retail organizations need transparent ROI calculations comparing conventional ETL, micro-batch, and streaming systems, considering

the following factors: reserved and on-demand cloud computing costs, data egress costs, and the costs of developer resources.

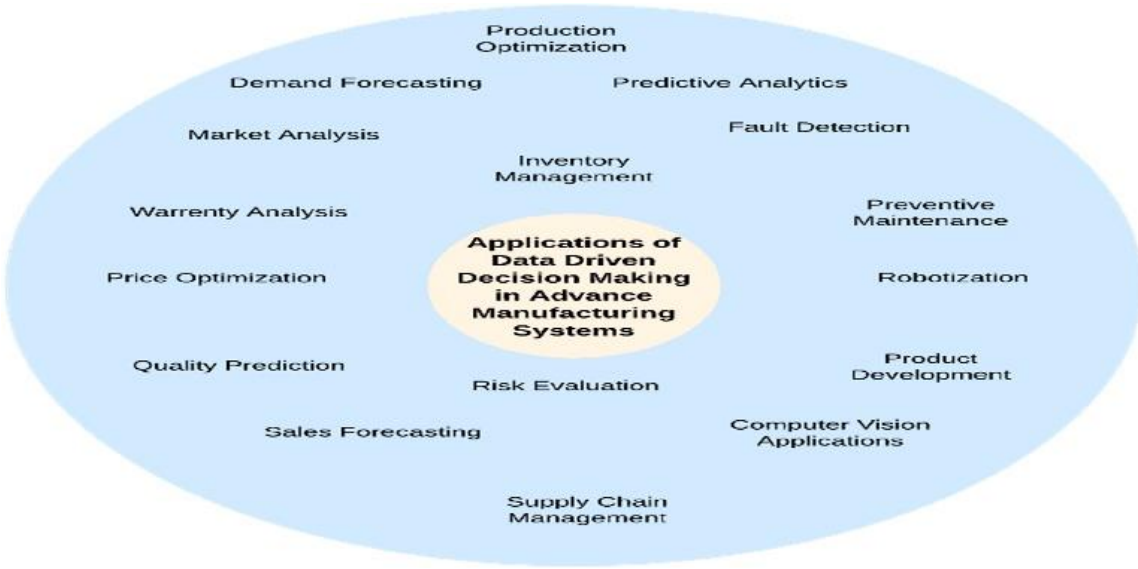

Figure 3: Data-driven decision-making applications in advanced manufacturing systems.

Advanced manufacturing systems, as shown in figure 3, use data-driven decision making throughout the value chain: central inventory management is implemented with demand and sales forecasting, price optimization, market and warranty analysis, and production optimization; risk evaluation and preventive maintenance are facilitated by quality prediction and fault detection; robotization and product development are implemented using predictive analytics and computer vision and supply chain management uses all these insights. The combination of these applications, price optimization, warranty analysis, predictive maintenance, and computer vision, helps illustrate how real-time data and historical data are informed to maintain resilient, efficient, and adaptive manufacturing operations.

There is little discussion about operational maturity models of real-time inventory systems. The most important operational priorities are to track consumer lag, manage Kafka broker failures, roll upgrades without event loss, and manage schema registry governance across dozens of teams. Teams that do not have documented patterns behind these non-functional requirements risk downtimes or data inconsistency when trying to go to scale. Inputs on these blank spaces in the form of empirical research, reference implementations, and standardized benchmarks would substantially hasten the pace of application of event-sourced, real-time analytics to retail inventory (19).

3. Methods and Techniques

3.1. System Architecture

The proposed architecture for event-sourced retail inventory management is designed to combine four major pieces, i.e., point-of-sale (POS) terminals, Apache Kafka, Google Cloud Dataflow (Apache Beam), and BigQuery, into a single low-latency pipeline. Like in table 1 below, the use of POS terminals working in the different points of the store resources broadcasts unchangeable transactional events (sales, returns, restocks, transfers) in real-time. Researchers serialize every event with Avro or JSON Schema and send it to a Kafka topic partitioned by business domain (e.g., inventory-adjustments, sales-transactions). A cross-data-center multi-tenant Kafka cluster guarantees global durability. Google Cloud Dataflow jobs, which in turn are developed using Apache Beam, are subscribed to

these topics continuously, with the application of transformations, business-logic enrichment, and error-handling, followed by loading processed data into BigQuery tables. This end-to-end flow ensures that every inventory update can be captured, processed, and stored in sub-second latency, making the state views and analytics real-time.

Table 1: Methods and techniques for an event-sourced retail inventory pipeline

Technique	Components/Steps	Tools/Tech	Key Configuration	Outcome
System Architecture	POS terminals → Kafka → Dataflow (Beam) → BigQuery	Kafka, Dataflow, BigQuery	Avro/JSON serialization; sub-second latency	End-to-end, low-latency inventory pipeline
Event Modeling & Schema Design	Define immutable InventoryEvent schema; manage versions	Schema Registry	UUID, enum types, backward/forward compatibility	Consistent, evolvable event format
Kafka Cluster Configuration	Topic partitioning by sku_id; replication; retention & compaction; broker tuning; exactly-once writes	Apache Kafka	RF=3; retention=24h + compaction; idempotence	Durable, balanced, high-throughput log
BigQuery Table Design & Partitioning	Raw event tables (partitioned by ingestion date, clustered on keys); materialized views with incremental refresh	BigQuery	_PARTITIONTIME prune; clustering on sku_id	Fast time-range queries; real-time state view
Streaming Pipeline Implementation	Ingestion, schema parse, enrichment, windowing, aggregation, error handling, monitoring	Apache Beam/Dataflow	Fixed windows + lateness; dedupe insertId; Cloud Monitoring metrics	Reliable real-time processing
Consistency & Fault Tolerance	Idempotent producers; Dataflow checkpointing; two-phase BigQuery commits; offset management; replay support	Kafka, Dataflow, BigQuery	Transactions API; Cloud Storage checkpoints	Exactly-once, recoverable end-to-end pipeline

3.2. Event Modeling & Schema Design

Event modeling starts with the definition of an immutable schema, InventoryEvent, that contains:

- **event_id (UUID):** universal unique identifier
- **eventtype (enum):** the legal change of SALE, RETURN, STOCK_ADJUSTMENT, TRANSFER
- **timestamp (ISO 8601 UTC**): time of making the event

- **sku_id (string):** item on sale
- **quantity_delta (integer):** positive - become restock or return, negative - to be sold
- **store_id (string):** site code
- **user_id (optional string):** identity of the operator or system
- **metadata (JSON map):** contextual info as free-form data (promo codes, supply-chain tags)

The compatibility is enforced through a centralized schema registry (Confluent Schema Registry or equivalent). Schema versions are read by producers and consumers using the REST API; schema development obeys backward and forward-compatibility principles, which means that optional fields can easily be added without disrupting the consumers. Deprecated fields are converted with controlled roll-outs: a new schema version is released to run in parallel, consumers upgrade to read the new and old formats, and finally, the old schema is discontinued (36). Such strict administration blocks variables at run-time, together with problems of schema discrepancy and information misappropriation.

3.3. Kafka Cluster Configuration

Meticulous cluster tuning is what gives high performance and resilience to Kafka:

- **The strategy for partitioning:** Topics are split using the hash of sku_id mod N (where N is the number of partitions to create). This is to ensure a balanced usage, as well as to reduce skew on high-volume SKUs.
- **Replication factor:** Each partition is replicated thrice across separate broker nodes and zones so that there is zero data loss in the case of broker failures (3).
- **Retention & compaction:** The time-based retention retains all the raw data for 24 hours to facilitate audit and replay, whereas log compaction retains only the latest record per key (e.g., last adjustment on an inventory per SKU) so that storage costs and storage requirements are balanced.
- **Broker tuning:** default heartbeat intervals, replicated fetch thresholds, and socket buffer sensitivities are configured to achieve high rates of throughput (>200 MB/s per broker) with latencies at the end-to-end below 50 ms.

To ensure exactly-once writes, producers make use of idempotent settings and transactions APIs. The consumers make offsets when they have downstream acknowledgement, which prevents data loss and redundancy.

3.4. BigQuery Table Design & Partitioning

BigQuery is a landing zone of raw events and a real-time data analytical store of aggregated inventory state. There are two types of tables used:

1. **Raw Event Tables**

- Schema mirrors InventoryEvent plus Kafka metadata (_PARTITIONTIME, _OFFSET).
- **Partitioned** by ingestion date (_PARTITIONTIME) to enable time-range pruning.
- **Clustered** on sku_id and store_id to accelerate point-select queries on specific items or stores.

2. **Materialized View Tables**

- Defined via SQL-based aggregation:

```sql
CREATE MATERIALIZED VIEW current_inventory AS
SELECT
  sku_id,
  store_id,
  SUM(quantity_delta) OVER (PARTITION BY sku_id, store_id ORDER BY ingestion_time
    ROWS BETWEEN UNBOUNDED PRECEDING AND CURRENT ROW
  ) AS cumulative_inventory,
  MAX(timestamp) AS last_event_time
FROM raw_inventory_events;
```

Figure 4: SQL for a materialized view that computes per-SKU, per-store cumulative inventory and latest event timestamp.

· **Time-partitioned** on a synthetic date column derived from a timestamp.

· **Clustering** on sku_id and store_id to further optimize lookup performance.

Materialized views have an incremental refresh that guarantees sub-second access to the current state of inventory without a full table scan.

3.5. Streaming Pipeline Implementation

The implementation of the streaming dataflow is in Apache Beam, Google Cloud Dataflow:

1. Source Ingestion

· Subscribe to a set of Kafka topics of interest using UnboundedKafkaIO with a custom name for consumer groups (30).

· Process timestamp extractors to get a hold of the semantics of event times.

2. Schema Verification and Parsing

· Serialize raw bytes into domain objects by the schema registry client.

· Drop or send unusable records to the dead-letter Pub/Sub topic with error data.

3. Enrichment & Normalization

Side input enrich events: catalogs of products and store meta tables.

Timestamps and currency columns must be normalized in the process.

4. Windowing & Late Data Handling

· Use fixed-time windows (e.g., one minute) and lateness of up to five minutes.

· Prosper Rose-Welsh user deprived, KB (and pane [triggers]) to issue intermediate and final experiments, late events to rectify the right window.

· Side outputs are used to pick up very late data in the backfills.

5. Transformation & Aggregation

· Calculate per SKU-store combination with each window the incremental deltas of the inventory.

· Insert idempotency by deduplicating insertId when writing results to BigQuery with streaming inserts.

6. Monitoring & Metrics

· Use Cloud Monitoring to monitor essential metrics of Dataflow: event backlog per partition, processing latency percentiles (P50, P95, P99), and Dataflow worker CPU%, memory usage, and auto-scaling events.

3.6. Consistency & Fault Tolerance

Big-Ten end-to-end exactly-once is implemented by a joint Kafka, Dataflow, and BigQuery configuration:

· **Kafka Producers:** Facilitate transactional writes that are idempotent, so that all the InventoryEvents are written atomically if they need to be done across partitions (21).

· **Dataflow Pipelines:** Use checkpointing to Cloud Storage: the state of every processing step (including window buffers, watermarks, and offsets) is saved. In case an employee does not pass, the pipeline can be picked up at the tail end, where no data is lost or duplicated.

· **BigQuery Writes:** Institute two-phase commit; Dataflow inserts into the staging table, and then in one transaction merges, rather than partially writes, into the materialized view.

· **Offset Management:** The Kafka offsets will be retained in a separate BigQuery table of offsets, indexed by job IDs in Dataflow (9). When restarting, the pipeline reads the most recent committed offsets to avoid reprocessing the events.

· **Replay Strategies:** Historical replays are also supported: raw event tables enable users to reprocess events, at any partition range, with patched pipelines to recreate downstream tables, either by specifying Kafka offsets or date ranges of ingestion events.

Such intense idempotent writes, checkpointing, and transactional commits ensure all inventory events are and will always be only processed once, no matter the failures, network partitions, or schema migration.

4. Data Exploration using Visual Analytics

Data exploration is also very vital in analysing the nature of retail inventory systems in real-time. Enterprises can target and analyze with BigQuery using its powerful querying language, SQL, efficiently, and can easily gain insights and performance (31). This section will present how the data on the events will be explored using SQL queries, visualizations, and patterns identified in the data sets.

4.1. Exploratory SQL in BigQuery

The initial part of data exploration will be conducting SQL queries in BigQuery to gain a deeper understanding of

the inventory data. BigQuery also has a distributed architecture that can be used to query large databases very quickly and is suited to event-based applications. The ability to filter, aggregate, and join data across several tables using SQL might include event logs, inventory snapshots, and sales records. A common query could be to retrieve the latest inventory position of each SKU in the system or examine the trend of movements of items over some time. An example of this would be a SQL query that tallies the total amount sold of a specific item over the last 24 hours, or the number of times an item has been restocked.

BigQuery offers an arguably strong capability to execute window functions as well as a time-series analysis. Violating retailers will be able to use the window functions to monitor inventory flow at specific periods and identify spikes in sales as well as out-of-stocks. Grouping the events based on the location, the period, or the SKU through BigQuery using its SQL capabilities, like GROUP BY or PARTITION BY, would provide practical insights (18). BigQuery's exploratory SQL pipeline, as illustrated in the figure below, makes use of materialized views to speed up typical trends. Raw events are stored in base tables, and the aggregate output (e.g., windowed totals or time-series summaries) is cached instead in a materialized view, which is periodically rewritten. Queries against the base table are not modified. However, execution of valid SQL constructs against it is transparently redirected to the materialized view smartly behind the scenes to execute more quickly when appropriate (grouping by SKU or partitioning by time to compute rolling sums, for example). The design can allow analysts to filter, join, and implement window functions on large event logs in real time, reducing query latency.

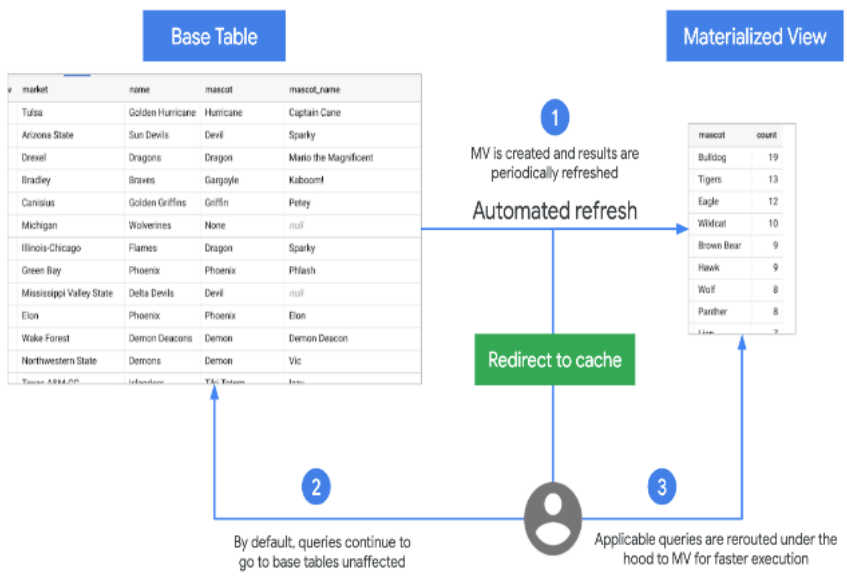

Figure 5: SQL for a materialized view calculating cumulative inventory and latest event timestamp.

4.2. Sample Dashboards (Looker/Data Studio): Event Volume, Item-Level State

After data is queried and processed, it can be visualized with the help of business intelligence tools, such as Looker or Google Data Studio. These tools make dashboards interactive and user-friendly, enabling real-time data viewing through this interface, which can be helpful in decision-making by the stakeholders. For example, a dashboard can show the volume metrics of events, the number of inventory-related events per minute, even per hour, or per day. This is used to track the health of the event-streaming pipeline to determine how quickly the pipeline can respond to events streamed into it. Time serial graphs or heat maps allow monitoring large volumes of inventory changes at peak times, enabling businesses to predict demand spikes.

The other vital visualization would be the item-level state, which provides current inventory on individual SKUs in a

store or a warehouse. This dashboard can be used to indicate whether they have less than, more than, or are at the right level of stock (11). In real-time, a heatmap of the inventory could indicate the products that are at risk of being out of stock, and then measures could be taken promptly to replenish them. The combination of Data Studio and BigQuery allows you to view these metrics easily and to look at the results in a user-friendly format.

4.3. Identifying Hot SKUs and Temporal Patterns

Having the raw event data examined and visualized, the retailers will be able to spot significant trends, like the hot SKUs or the time trends. Hot SKUs are the products that show rapid selling or rapid restocking. Businesses that monitor historical and real-time data will be able to call attention to their products more quickly than others, which may be due to high customer demand or popular offers.

Temporal patterns, conversely, show the time-wise tendencies in inventory. As an example, sales of some products could be at their highest on a particular day of the week, time of the day, or holiday periods. The time-series capability of BigQuery allows tracing and predicting these trends without any problem (16). Retailers can use sophisticated SQL queries to determine the impact of variables on inventories and sales and, in this way, estimate the value of promotions or changes in weather, etc.

These trends can be identified to derive quantifiable decisions concerning supply chain, inventory management, and even demand surges. When the hot SKUs are identified early, faster replenishment can be done, and it is also possible to adopt a dynamic pricing plan. Temporal analysis is beneficial in scheduling promotional operations more effectively. Using such visual analytics methods, retailers are given an overall picture of their stock management system, thus having valuable insights that can have a direct and tangible influence on their operations.

5. Data Transformation & Aggregation

5.1. Incremental Roll-Up: From Raw Events to Current Inventory View

With an event sourcing architecture, the raw data of different activities concerning the inventory (point-of-sale systems, stock adjustments, and stock movements) is stored and recorded as individual events. Such events, which constitute the state changes (such as stock additions and removals), are committed to Kafka topics to enable real-time processing. A timestamp, SKU identifier, change type (such as +5 units, -3 units), along with any metadata, is provided in each event. The challenge with processing this data is to transform these event logs to a near real-time update of the current inventory state, as the event data is often received in an out-of-order fashion, or bursts.

The incremental roll-up refers to the procedure based on which only recent events are taken into consideration in calculating the current level of the inventory, instead of recalculating the whole inventory every time information is updated. (5) The main advantage of having an updated view of inventory state using incremental updates is that only new or changed data is retrieved, minimizing computational requirements and increasing rates of system responsiveness. This is made possible by establishing aggregations that enable efficient calculation of the current inventory count of each SKU, and typically these are time-windowed or event-sequenced aggregations.

5.2. Use of BigQuery Materialized Views for Performance

BigQuery has a flexible way to enhance performance, which is the materialization of query results. Raw data of events is subject to stream processing, and the resulting tables are inserted into BigQuery; however, they are not optimized for high-performance queries. To deal with this, materialized views will be used, which contain pre-computed data, like the total inventory level of each SKU or product.

The creation of materialized views in BigQuery enables fast query of opinions without needing to recompute such

aggregations every time a query is executed. The materialized view is always up to date and reflects the state of the raw data because the BigQuery table is automatically updated when new data arrives (14). By making use of pre-computed values, this reduces the time taken to carry out queries with dashboards and real-time analytics considerably. In the retail inventory scenario, materialized views can be particularly effective when it comes to frequently used measurements like stock levels of popular products, such that real-time measurements can be accessed near-instantly and the data can be used to make decisions in real-time.

The system also minimizes the amount of workload that the data processing pipeline needs to perform since, with the materialized views, the large volume of inventory data can be queried quickly, despite the increase in the volume of the data. The algorithm also saves storage space costs since there is no need to recalculate the results and store them.

Table 2: Data Transformation and Aggregation Techniques for Real-Time, Event-Sourced Inventory

Process	Description	Mechanism	Implementation	Benefit
Incremental Roll-Up	Aggregate only new events to update current inventory	Time-windowed or sequenced aggregations	Kafka → Dataflow windowed SUM over quantity_delta	Reduces compute by avoiding full re-compute
Materialized Views	Pre-compute and cache inventory totals per SKU	BigQuery materialized views	Define view on raw event tables with window functions	Sub-second query performance on live inventory state
Late-Arriving Corrections	Merge out-of-order or delayed events into existing state	BigQuery MERGE	Run MERGE matching on sku_id and timestamp	Ensures data accuracy without full history replay
Schema Enforcement	Validate and parse incoming events	Avro/JSON Schema + Schema Registry	Schema Registry lookup and compatibility checks	Prevents downstream errors and schema drift
Monitoring & Metrics	Track pipeline health, latency, and backlog	Cloud Monitoring + custom metrics	Monitor Dataflow P50/P95/P99 latencies, backlog	Enables proactive maintenance and SLA adherence

5.3. Late-Arriving Correction via Merge Statements

Event-sourcing systems are not unusual because some events are potentially received out-of-order or because some data is received too late and is not processed at the right time. For example, stock/inventory movements can be updated with a lag in case the system is facing network problems or is down. Whenever these late arrivals are not handled adequately, they may cause discrepancies in the data on the inventory and affect the veracity of reports and real-time analytics. The MERGE statement is a powerful feature of BigQuery that helps to process the late-arriving events (4). It enables the contingent change, insertion, or removal of records in a target table according to matching standards between source data and target data. This is especially helpful in fixing the inventory information in case of an out-of-order event arriving. Similarly, in a situation where a stock adjustment event arrives later than initially forecasted, the MERGE statement will again help to update the existing record in the target table such that the inventory level is always in sync.

The MERGE operation checks for discrepancies with the current inventory state and incoming events, and updates the inventory count. This makes it possible to correct inventory balances even once initial processing has been done by using later arriving events in a manner that does not demand a full reprocessing of all history. With the help of MERGE statements, the system would be able to achieve a high rate of accuracy, so that the current state of inventory could be as accurate as possible at all times, despite the nature of late arrival of events in a distributed system.

6. Real-Time Analytics Techniques

6.1. Parameterized SQL Queries for Alerting (Low-Stock Events)

The factor that is most critical to real-time inventory management is the capability to have early notification in case of low-stock situations. In an environment where there are one hundred things to keep count of, it is essential to ensure that there are no lost sales by maintaining the stock parameters and restocking it in time (27). By using the performance capabilities of BigQuery, which can analyze high-volume data without significant latency, retail systems can configure SQL queries that have parameters so that when their inventory changes, the systems are triggered to alert them.

The process starts with querying of inventory data already stored in BigQuery, where the quantity per item is kept up to date through the transactions. A parameterized SQL statement may be developed to review the stock levels in real time, comparing the current inventory conditions against the set limits (38). To give a pertinent example, a query could be established that would notify inventory managers when the remaining inventory is at a predetermined low level (such as 10 units). This is possible through the creation of the WHERE clause in SQL, and this can be used to list the products that are below a predetermined stock level.

The parameterized queries can also be used to adjust the values of the thresholds without changing the SQL code, which means the solution is flexible and easy to configure. Alerts are triggered when the results of the query are encountered as per the specified rules, which causes notifications or other automation tasks, such as email messages and/or triggers in a service such as Google Cloud Pub/Sub, to be initiated. This approach enables such inventory teams to react promptly to refill products, ensuring they are always available and reducing the risk of product shortages.

6.2. BigQuery ML for Short-Term Sales Forecasting

BigQuery ML (Machine Learning) is a feature that allows users to create machine learning models using BigQuery, providing extensive analytics possibilities that were previously unavailable while using external machine learning systems. Short-term sales forecasting is critical in retail inventory management, where it helps the business identify items that will need a refill based on historical data and trends. Retail businesses can develop their predictive model by implementing machine learning algorithms in BigQuery. This solution can study historical sales trends, seasonalities, effects of promotions, and other significant parameters. BigQuery ML makes it easier to solve since these models are provided as a default: linear regression, logistic regression, and time series forecasting (e.g., ARIMA) (7).

These models can be trained using data that already exists within BigQuery, and such an integration is also possible with the retail inventory pipeline. For example, a time series forecasting model may be used to determine what an item will sell in the future based on its previous pattern of sales and take into account events such as promotions, holidays, and trends. Retailers can run low-latency forecasts in near-real-time, using BigQuery ML to make necessary adjustments to inventory levels and optimize stock replenishment itineraries. This ability to provide a

forecast will have a direct effect on the accuracy of maintaining an inventory. As a result, there will be a better match between sales and inventory levels.

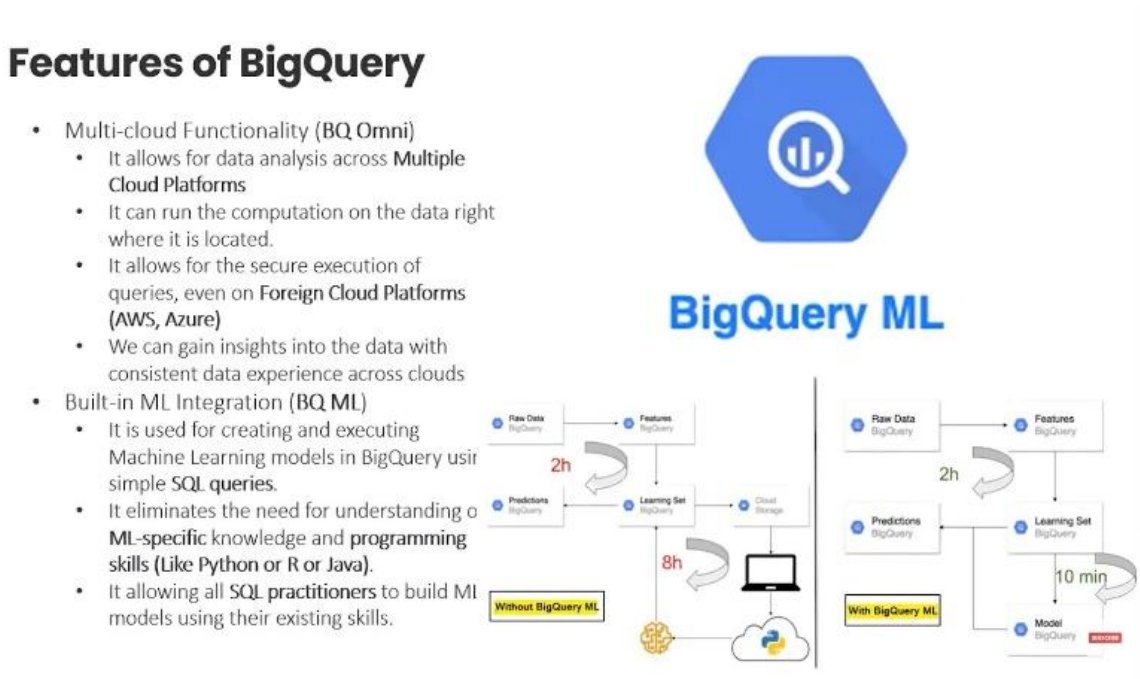

Figure 6: BigQuery ML: native SQL-based machine learning with multi-cloud support

BigQuery's multi-cloud omnichannel architecture and built-in ML integration enable users to build and execute models based on linear regression, logistic regression, ARIMA, and beyond in SQL on already defined tables, as shown in the figure 6 above. This method of predictive analytics without external tooling makes near real-time sales forecasting and predictive analytics easy using data already stored in BigQuery. They could provide low-latency forecasts because models can be trained on past sales, seasonality, and other parameters, and deployed immediately. Automatic model refresh and scalable execution are helpful in retailers. Using SQL and existing data, organisations attain accurate short-term demand forecasting and optimise inventory restocking as well as streamline operational complexity over cost-effective third-party machine learning services.

6.3. Integration with Pub/Sub for Push Notifications

Besides real-time notification, its combination with Google Cloud Pub/Sub ensures a powerful push notification system that enhances the solution's capabilities for real-time inventory control. Pub/Sub is a messaging service that facilitates communication between systems and applications asynchronously. Thus, it is useful when real-time updates are required, especially in high-volume scenarios like retail inventory systems.

The output of low-stock events, sales forecasting predictions, and other vital metrics that are calculated in BigQuery can be published to a specific Pub/Sub topic. This can then be reflected in the subscribed systems, which are typically dashboards on inventory management, mobile applications, or the notification systems. As a specific example, when the stock of an item reaches a certain threshold, a message with necessary information (e.g., product ID, current stock level, and recommendation on how many should be added) is emitted to a topic in Pub/Sub. Subscribed systems can subsequently execute actions like automated purchase orders or apprise store managers through a push notification. The integration provides all the stakeholders with the information in real-time that they need when something happens, be it the low-stock alert, sales spike, or forecasted demand variation (15). By separating the data processing and notification systems, Pub/Sub can also be scaled and be flexible in streaming real-time data to any number of endpoints without overwhelming any one of them.

6.4. Caching with BI Engine for Sub-Second Dashboards

This requires high-performance dashboards that will show real-time analytics as their information is presented. To fulfill this requirement, researchers can use caching tools such as Google BigQuery BI Engine to increase query-delivery performance and achieve sub-second response times for querying on a large dataset. BigQuery BI Engine is an in-memory analytics service that can be used with BigQuery to accelerate the speed of interactive analytics queries. BI Engine uses memory to cache the results of frequently used queries, so that they are executed faster the next time without scanning a large amount of data again. This is what is needed to build interactive real-time dashboards, containing inventory counts, sales patterns, and demand projections with very little latency.

In retail businesses, dashboards are invaluable aids to better decision-making because the displays provide actual stock balances, current sales, and projected demand. When using BI Engine, the dashboards can automatically update based on real-time changes to the inventory, without querying raw data in BigQuery, which can be time-consuming. Such improvements have particular significance to the needs of use cases like tracking the sales of a product at a variety of stores or to view the effects of new sales or a promotion. Besides enhancing the query performance, BI Engine can also be directly integrated with tools such as Google Data Studio or Looker, so creating and managing some real-time dashboards can be done without hassle. BigQuery, alongside BI Engine and the ability of real-time data pipelines, will provide the retail inventory systems with the capacity to generate moment-to-moment insights with an efficient level of scalability.

A combination of real-time analytics methods, such as parameterized SQL queries, BigQuery ML forecasting queries, push notifications using Pub/Sub, and caching using BI Engine, offers a robust and scalable means of managing inventories in the retail market. Such methods can help these organizations to act promptly by responding to any variations in inventory levels, optimizing restocks, and keeping the involved stakeholders in the know to make informed decisions and prompt actions (13). Machine learning forecasting and pairing it with the capabilities to access real-time data and efficient messaging systems can provide retailers with more control over their inventory management, which results in a more efficient operation and customer satisfaction.

7. Results and Experiments

In this section, the author shares the outcomes of a series of experiments that were carried out to assess the behavior and scalability of the event-sourcing solution on retail inventory use of Kafka and BigQuery. The experiments were to be carried out to measure latency, throughput, scalability, and fault-tolerance in a realistic setting, and a practical case study on anomaly detection.

7.1 Benchmark Setup

The software or system used to perform the testing was a cloud-based infrastructure that can be used to replicate the real-life retail inventory system. The cluster was set up on Google Cloud Platform (GCP), where Kafka (an event-streaming platform), Google Cloud Dataflow (an Apache Beam-based platform), and BigQuery were used for processing, and BigQuery was used as the analytical platform.

The benchmark tests were carried out in two steps, namely, synthetic event generation and simulation of real retail event streams. The modelling of the data was carried out through synthetic event generation to provide a controlled set of data that would permit the prediction of behavioral patterns. Conversely, operational inventory systems were used to provide real retail event streams, as a stock was updated, a product restocked, and a sales transaction processed, these were streamed into Kafka topics in real-time. This two-pronged methodology meant that the performance of the system was tested in the idealized and realistic cases. The Kafka cluster is configured with three

brokers to provide high availability and fault tolerance. All brokers had four vCPUs and 16 GB RAM. The Kafka streams were ingested into the Datasource using Dataflow, transformations were applied as needed, and the result was loaded into BigQuery (2). Tables in BigQuery were organized into partitions by timestamp, where each event was timestamped as it was inserted.

Two different sets of retail events configuration were taken into consideration as part of the experiments:

· **Synthetic data:** This type of event is created according to specific parameters, such as random sales transactions, product arrivals, and inventory updates. These permitted manipulations of the occasion frequency and magnitude.

· **Real-world data:** Logs of an event triggered by a real retail inventory system, which can include a combination of stock replenishment data, graphical data on point-of-sale, and inventory movements.

These synthetic and physical data sets provided valuable insights into how the system performed under varied conditions of operation.

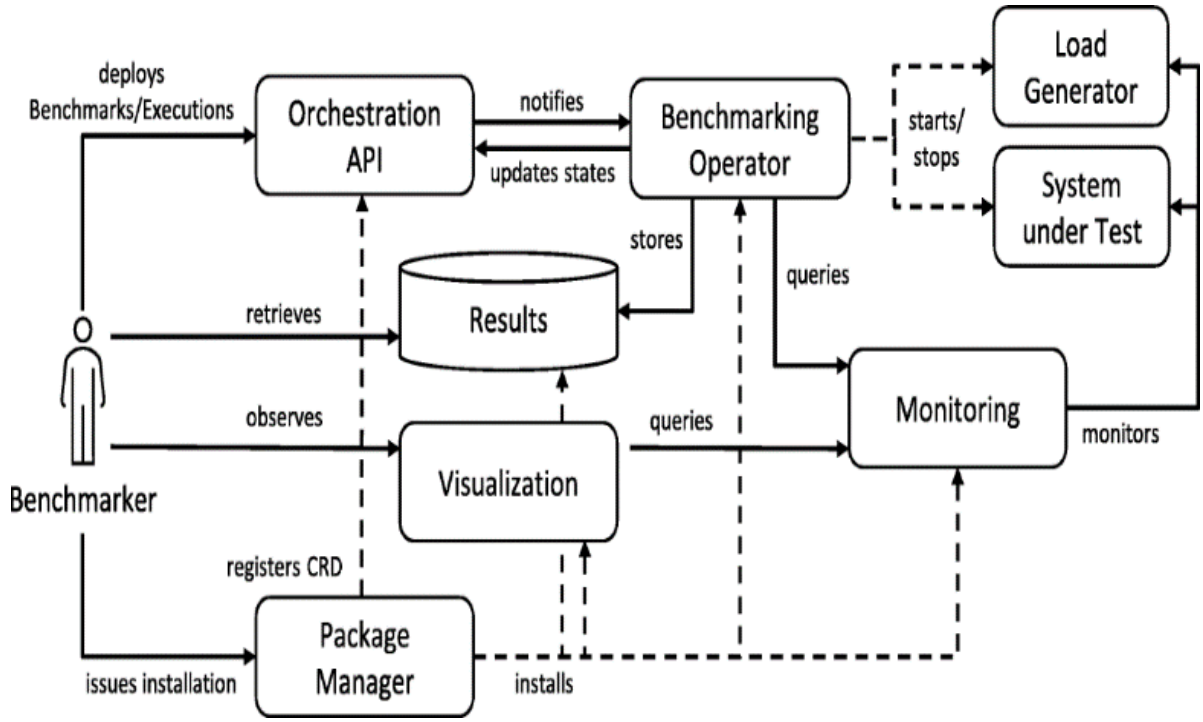

Figure 7: Orchestration-driven benchmarking pipeline: deploys synthetic & real-stream tests, generates load, monitors performance, stores results, and drives visualizations

The benchmarking framework, as shown in the figure above, instruments both synthetic and real retail event streams using a declarative API to run benchmarks and executions in a GCP-based cluster running Kafka, Dataflow, and BigQuery. The Orchestration API provides tests and monitors the state. The Benchmarking Operator launches and shuts down the load generator against the system under test, and the Results store persists performance data. Monitoring services constantly scrape metrics of brokers, pipelines, and queries, and a Visualization layer queries the Results database to display dashboards. The Package Manager supports installation and custom resource definition and allows end-to-end performance measurement to be replicated.

7.2 Latency & Throughput Measurements

Latency and throughput are essential performance measurements for real-time event-sourcing systems and are common in retail applications where the inventory updates must be seen instantaneously by all systems (8).The

latency was calculated between the time an event was published to Kafka and when the same query would give a result in BigQuery. During the synthetic data tests, a sustained flow of events was sent into Kafka at 10,000 and 100,000 events per second (EPS). The average write-to-query latency at 10000 EPS was 300 milliseconds, and the latency at 100000 EPS slightly increased to 350 milliseconds. This slight decrement in latency is still acceptable in the realm of real-time analytics, and this shows that the system can process larger amounts of data with much less penalty in the response time.

The latency tests also monitored the resource consumption. Kafka brokers were allowed to sustain both 10k and 100k EPS without facing much strain on resources. Nonetheless, as the event rates increased, a few scaling problems in Dataflow jobs that were processing Kafka streams were observed (34). In particular, the default settings in Dataflow needed further optimization in parallelism at 100,000 EPS to maintain the same latency levels. The cost metrics were very encouraging in this phase of testing as the system delivered on the increased load at an incremental cost of about 0.12 per 10,000 events ingested, which is a good value in large-scale retailing environments. Evaluation of throughput with the system was recorded, and it was found that the system was able to deliver a constant stream of 100,000 events per second without a loss of delivery and minimal impact on stage processing times. Such capacity is in line with the anticipated throughput in large-scale retail settings where high-volume transactions must occur in real-time environments.

7.3 Scalability Tests

It is critical to have scalability to ensure the system can handle the growth of the events in the future, including their volume and complexity of data. They performed a set of scalability tests by manipulating the number of Kafka partitions and the degree of parallelism in the Dataflow work. The first test was carried out by setting the partitions in Kafka to 100 as compared to the 10 partitions and noting how the throughput and latency of the systems were affected by this difference. Its findings were that with the increase in terms of partitions, throughput and latency had improved, and a linear connection existed between the number of partitions and the system (10). By that 100 partitions, the system could maintain up to 150 K EPS with little latency difference (350400 milliseconds). This scaling behavior is an essential feature of Kafka because it ensures that it is easy to spread the load among many partitions so that the system can expand without reaching any performance limits.

The other influencing factor that led to the scalable performance in Dataflow was parallelism. During the early tests, the Dataflow defaulted to a factor of parallelism. Nonetheless, the parallelism settings had to be optimized when the event volume was greater. At event subsystem speed 100,000 events per second, there was already a significant decrease in event processing time with increasing parallelism of the worker pool; with eight workers, the average processing time was 500 milliseconds per event, but when the parallelism was increased to 32 workers, the average processing time decreased to 300 milliseconds per event. Such changes enabled the system to process events in real-time, even when under considerable load. Partitioning strategies also influenced performance, a move that was also tried in scalability tests. The partitioning strategy based on a hash also helped mitigate load skew across the Kafka brokers and increased the consistency of throughput (26). The fact that the Dataflow pipeline was scaled to 32 workers and also employed the utilization of multiple Kafka partitions made it possible to observe the efficiency requirement of the respective system to meet the scale, both in terms of the volume of events and the complexity thereof, without compromising the performance thereof.

Table 3: Benchmark and experiment outcomes for the event-sourced inventory pipeline

Phase	Configuration	Workload	Key Findings	Impact

Benchmark Setup	3-broker Kafka (4 vCPU, 16 GB RAM), Dataflow, BigQuery	Synthetic & real retail event streams	Infrastructure mirrored real use cases; enabled controlled vs. live testing	Validated testbed for subsequent experiments
Latency & Throughput	10 k EPS & 100 k EPS sustained	10 000 EPS → 300 ms; 100 000 EPS → 350 ms latency	Throughput held at 100 k EPS with minimal write-to-query delay; incremental cost ~$0.12/10 k events	Meets real-time SLA
Scalability	Kafka partitions: 10 → 100; Dataflow workers: 8 → 32	Up to 150 k EPS	Higher partitions and parallelism improved throughput linearly; latency ~350–400 ms	Demonstrated seamless horizontal scale
Anomaly Detection	Streaming SQL in BigQuery + real-time dashboards	Stuck-inventory detection over 24 h windows	Identified unupdated stock in near real-time; triggered alerts	Enabled rapid error remediation
Trade-off Analysis	Operational tuning vs. cost	High EPS scenarios	Complexity and cost rose with partitions/workers; ultra-low latency demands increased overhead	Informed ROI and resource planning

7.4 Case Study: Anomaly Detection

An actual simulation was given to test the feasibility of the system to identify inventory anomalies in real time. Its emphasis was on the detection of stuck inventory occurrences, where the stock level of an inventory was not updated even when it was sold or restocked, which could be the result of data quality problems, process failures, or problems with the system. An Anomaly was established using past inventory records. Goods whose stock levels remained unchanged over some time (e.g., 24 hours) despite a massive number of transactions were identified and said to be stuck. BigQuery allowed the system to aggregate the inventory events by product ID and location and filter them to identify the discrepancies between restocking events and sales transactions.

Anomaly detection is done in real-time using the event streams to keep monitoring the flow and trigger an alert every time the inventory number of an item does not change, even though something happens on related Kafka topics. The algorithm used to detect it was incorporated into the streaming SQL features of BigQuery, enabling real-time analytics to identify the ailing inventory promptly (22).

A dashboard snapshot was created to display inventory variations and highlights on items that were stuck in progress, marked in red to be urgently addressed. In the case study, the system was able to identify inventory anomalies in near real-time, which was very valuable to an inventory management team. This automation of such detection allowed the retailer to react to problems more promptly and minimize the effect of data errors.

7.5 Trade-off Discussion

The system yielded great results in the various test conditions; nonetheless, several trade-offs need to be taken

into account when deciding to adopt an event-sourcing solution to retail inventory on a large scale. A trade-off is the complexity that is associated with the management of real-time data streams. The system needed to be thoroughly tuned in terms of Kafka partitioning and Dataflow parallelism to sustain the event rates at a reasonable performance level. Although Kafka enjoys significant horizontal scaling capabilities, the operational overhead of dealing with numerous partitions and topics will naturally add overhead once large numbers start to come into play. The other trade-off was the cost of streaming data processing. The running cost of the Dataflow jobs and BigQuery queries was very high at high event rates when the real-time inventory data to be processed was high. Although the cost performance of the system was not high enough to be accepted by large retailers, it can be among the necessities requiring the evaluation of the budget and performance needs by smaller organizations to take such a solution.

The system also performed well in processing high-throughput event streams. Still, it also experienced some latency as the system continued to scale, especially when it came to late-arriving events or simply the number of dataflow workers beyond a given scale. Although such increases are pretty small, it is a concern when the cost is high in the case of ultra-low-latency or precision timing. Although there are inevitable trade-offs in terms of complexity, cost, and latency of such a system, the solution that combines the usage of Kafka and BigQuery can be defined as quite potent and scalable in terms of real-time management of retailing inventory (33). Its performance capability of handling millions of events per second and low-latency querying make it a good candidate to run in modern retail settings, where it is helpful to have up-to-date inventory information to make the best decisions.

6. Future Work

6.1 Extending to Multi-Region Clusters and Global Inventory

Scalability between regions is one of the first aspects that need further development of the System of Event Sourcing as a retail-inventory system. Currently, the system operates well in one of the regions and uses cloud-based services such as Google Cloud to run Kafka and BigQuery. But as retail operations become more globalized, the need to distribute inventory data to various geographical locations increases to enable low-latency access and fault tolerance. It is possible to deploy multi-region Kafka clusters where event streams can replicate across regions efficiently. This will help avoid service interruptions due to regional outages, enhance disaster recovery, and result in the real-time synchronicity of store inventory data worldwide. This arrangement will also positively affect global inventory management, since the consistent flow of real-time stocks, prices, and forecasts of demand will be real-time, irrespective of the geographic location of a particular store.

6.2 Automated Schema Evolution and Compliance Auditing

The inventory management system will likely evolve in terms of data structure complexity as the retail industry continues to expand. The event schemas might require modification over time in the event of new product attributes, inventory processes, or business regulations. A significant problem in an event-sourcing system is how to ensure backward compatibility with schema evolution (20). To tackle this, researchers can work on having automated schema evolution tools that can monitor changes in the event data and perform the required transformation to ensure compatibility at different levels of the event processing pipeline. These instruments would also decrease the manual intervention, thus giving less downtime and fewer errors. Moreover, compliance auditing is automatable, allowing for the checking of variables in the events and ascertaining their alignment with the regulatory laws and the internal regulations. With auditing systems in place that monitor schema changes, lineages, and processing actions, organizations can ensure that their event sourcing solutions can remain oriented towards industry standards, as well as regulatory mandates.

6.3 Integration with Edge Computing for Offline Stores

The other area that offers improvement is the incorporation of edge computing services for less-connected retail shops. This would allow stores to operate locally on local computing resources rather than being connected to the central cloud infrastructure in areas with unstable internet connections, enabling them to process inventory data. Events could be in real time, such as stock updates, sales of products, and returns made by customers in edge devices. Each of these local systems would then be able to publish aggregated events to the central Kafka stream once connectivity is made available again, and keep inventory data consistent in all stores (23). This would enhance the resilience of the retail inventory system by keeping it going despite the hostile nature of the environment where network connectivity is sparse, intermittent, or nil. Whereby, edge learned in real-time may support dynamic pricing and localized demand forecasting so that inventory management may be outcome-adjusted per store, but still consistent with the overall scheme.

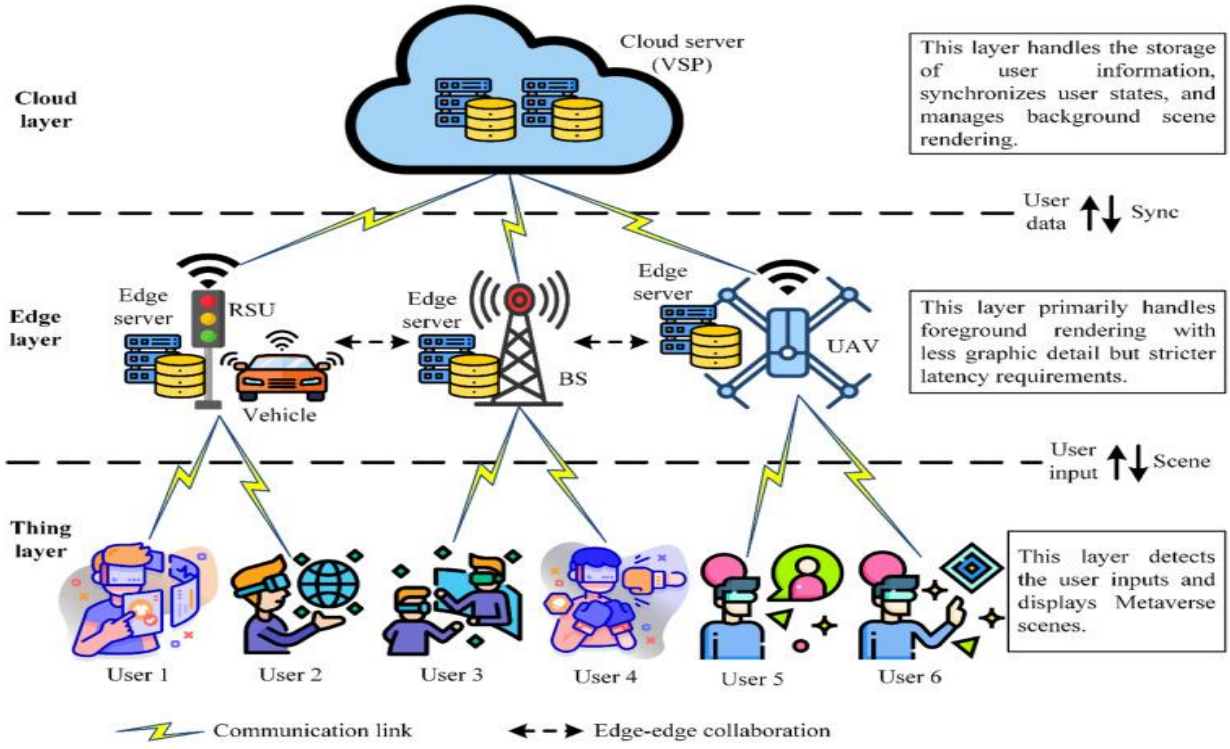

Figure 8: Three-tier cloud–edge–device architecture for resilient offline retail event processing and sync

A three-layer architecture can support resilient offline retail operations by computing inventory events at the edge, even when connectivity is sparse, as shown in figure 8 above. Sales, restocks, and returns are recorded as immutable events by in-store devices at the level of the "Thing". The localized demand forecasting, dynamic pricing, and synchronization of user data consist of the aforementioned local servers forming what is known as the Edge Layer. After the network connection is restored, the border nodes post-batched and schema-verified events to the main Kafka stream. The "Cloud" layer consumes and stores all the updates from the stores in Kafka, Dataflow, and BigQuery in a constantly consistent approach and with no-failure connectivity of intermittent networks.

6.4 Cost-Optimization Strategies (Reservation vs. On-Demand)

The cost of cloud resources may be huge as the number of events in the retail inventory system rises. It will also be imperative to optimize costs to make the system sustainable. One of them is to consider the trade-off between on-demand and reserved resources. Flexibility and scalability are also made possible through on-demand resources, which allow them to scale capacity up or down to meet fluctuating event loads (29). Nevertheless, in the case of

foreseeable loads, it may lead to a significant cost reduction when instances or resources are reserved. In the future, experts will apply the concepts of dynamic resource allocation, utilizing intelligent methods to switch reservation resources to on-demand resources, based on events and system loads. For example, at peak retail times (e.g., Black Friday), the system might automatically compute and increase the necessary resources with on-demand infrastructure.

During off-peak times, the consistent flow of events could be served by a smaller number of reserved instances, but at a slower pace. The inclusion of cost-tracking tools also allows for tracking spending and provides real-time advice on how to manage your time better. Through these challenges, the system will become more scalable, resilient, and cost-effective. In this way, the event sourcing solution will not break down once the retail inventory landscape shifts.

7. Conclusion

The combination of event sourcing, Kafka, and BigQuery is a big step forward in terms of real-time retail inventory management. This site supports a highly scalable and fault-tolerant distributed network, and accurate and up-to-date inventory states throughout distributed networks. The event-sourcing pattern with immutable events has the benefit that it keeps a comprehensive audit trail, which can be easily restored, time-traveled, and gracefully supports concurrency control. A combination of Kafka and BigQuery enhances the system's capacity to handle a high volume of events and provides low-latency performance, which is suitable for contemporary retail operations. The design of the system focuses on scalability, fault tolerance, and optimisation of performance. The fact that Kafka can support millions of events every second and that BigQuery can analyze data in real-time means they can form an imposing tandem to manage massive volumes of inventory-related data. Streaming architecture will help in ensuring the processing of the inventory and its analysis in real-time only, and provide exact insights to drive operational decisions such as dynamic pricing, demand forecast, and inventory resupply. All these are essential capabilities to a business that aims to optimize its supply chain and customer satisfaction.

Moreover, the solution gives significant advantages in comparison to the classical approaches on batch-based ETL pipelines. Event-sourcing can guarantee that decisions are made on the latest data possible, businesses can respond promptly to demand changes and stock fluctuations due to the removal of the latency that batch processing would have. This instant insight into the inventory status has the potential to enhance operational efficiency significantly and, of course, to avoid overselling or even stockouts. The study also points out several trade-offs and difficulties resulting from the adoption of this system. Managing distributed streams of events and optimizing resource use while working with the overhead of schema evolution and fault tolerance is substantial. These are the aspects that should be considered a priori, as in the case of large-scale retailing, where such events can flood unoptimized systems.

Even though the event-sourcing architecture has successfully passed the preliminary testing phases, it remains evident that it will still require future optimization to support the volume of events it will have to process and address the changing business preferences. Fine-tuning of multi-region clustering, auto-schema evolution, and edge integration will be key competencies to take the system to the next level of scale. Also, the implementation of cost-optimization parameters, including the use of dynamic switching between the reserved and on-demand resources, will be crucial for making the system cost-efficient as it scales. Event-sourcing used along with Kafka and BigQuery to manage real-time retail inventory is a paradigm-shifting proposal capable of providing scale-out event-driven real-time analytics on a low-latency and high-throughput scale. Although there are issues to address and integrate, especially regarding system complexity and cost, the advantages of not only increased inventory accuracy and

operational efficiency but also a scalable approach make it a potentially beneficial solution for the modern-day retailer. This architecture will no doubt become prevalent in the evolution of retail inventory management systems with further refinement and optimization.

References

1. Adelusi, B. S., Ojika, F. U., & Uzoka, A. C. (2022). Systematic Review of Cloud-Native Data Modeling Techniques Using dbt, Snowflake, and Redshift Platforms. *International Journal of Scientific Research in Civil Engineering*, *6*(6), 177-204.
2. Akanbi, A., & Masinde, M. (2020). A distributed stream processing middleware framework for real-time analysis of heterogeneous data on big data platform: Case of environmental monitoring. *Sensors*, *20*(11), 3166.
3. Alfatafta, M. (2019). *An analysis of partial network partitioning failures in modern distributed systems* (Doctoral dissertation, University of Waterloo).
4. Armbrust, M., Das, T., Sun, L., Yavuz, B., Zhu, S., Murthy, M., ... & Zaharia, M. (2020). Delta lake: high-performance ACID table storage over cloud object stores. *Proceedings of the VLDB Endowment*, *13*(12), 3411-3424.
5. Chavan, A. (2021). Exploring event-driven architecture in microservices: Patterns, pitfalls, and best practices. International Journal of Software and Research Analysis. https://ijsra.net/content/exploring-event-driven-architecture-microservices-patterns-pitfalls-and-best-practices
6. Chavan, A. (2022). Importance of identifying and establishing context boundaries while migrating from monolith to microservices. Journal of Engineering and Applied Sciences Technology, 4, E168. http://doi.org/10.47363/JEAST/2022(4)E168
7. CHITNIS, A. (2022, February). *Machine Learning for Fraud Detection Leveraging Sap Finance Data: A Case Study of BigQuery ML Application*.
8. Emily, H., & Oliver, B. (2020). Event-driven architectures in modern systems: designing scalable, resilient, and real-time solutions. *International Journal of Trend in Scientific Research and Development*, *4*(6), 1958-1976.
9. Emma, O. T., & Peace, P. (2023). Building an Automated Data Ingestion System: Leveraging Kafka Connect for Predictive Analytics.
10. Feng, M., Krunz, M., & Zhang, W. (2021). Joint task partitioning and user association for latency minimization in mobile edge computing networks. *IEEE Transactions on Vehicular Technology*, *70*(8), 8108-8121.
11. Karwa, K. (2023). AI-powered career coaching: Evaluating feedback tools for design students. Indian Journal of Economics & Business. https://www.ashwinanokha.com/ijeb-v22-4-2023.php
12. Kindson, M., & Péter, M. (2023). A simplified approach to distributed message handling in a CQRS architecture. *Acta Polytechnica Hungarica*, *20*(4).
13. Kumar, A. (2019). The convergence of predictive analytics in driving business intelligence and enhancing DevOps efficiency. International Journal of Computational Engineering and Management, 6(6), 118-142. Retrieved from https://ijcem.in/wp-content/uploads/THE-CONVERGENCE-OF-PREDICTIVE-ANALYTICS-IN-DRIVING-BUSINESS-INTELLIGENCE-AND-ENHANCING-DEVOPS-EFFICIENCY.pdf
14. Lakshmanan, V., & Tigani, J. (2019). *Google Bigquery: the definitive guide: data warehousing, analytics, and machine learning at scale*. O'Reilly Media.

15. Lindberg, S. (2022). Real-time Balance Management in Omnichannel Retail.

16. Marrandino, A. (2021). *Machine Learning with BigQuery ML: Create, execute, and improve machine learning models in BigQuery using standard SQL queries*. Packt Publishing Ltd.

17. Mehmood, E., Anees, T., Al-Shamayleh, A. S., Al-Ghushami, A. H., Khalil, W., & Akhunzada, A. (2023). DHSDJArch: An Efficient Design of Distributed Heterogeneous Stream-Disk Join Architecture. *IEEE Access*, *11*, 63565-63578.

18. Mucchetti, M. (2020). *BigQuery for Data Warehousing*. Springer.

19. Nyati, S. (2018). Revolutionizing LTL carrier operations: A comprehensive analysis of an algorithm-driven pickup and delivery dispatching solution. International Journal of Science and Research (IJSR), 7(2), 1659-1666. Retrieved from https://www.ijsr.net/getabstract.php?paperid=SR24203183637

20. Nyati, S. (2018). Transforming telematics in fleet management: Innovations in asset tracking, efficiency, and communication. International Journal of Science and Research (IJSR), 7(10), 1804-1810. Retrieved from https://www.ijsr.net/getabstract.php?paperid=SR24203184230

21. Oliveira Rocha, H. F. (2021). How to Manage Eventual Consistency. In *Practical Event-Driven Microservices Architecture: Building Sustainable and Highly Scalable Event-Driven Microservices* (pp. 187-225). Berkeley, CA: Apress.

22. Pamisetty, A. (2019). Big Data Engineering for Real-Time Inventory Optimization in Wholesale Distribution Networks. *Available at SSRN 5267328*.

23. Pandey, P. K. (2019). *Kafka Streams-Real-Time Stream Processing*. Learning Journal.

24. Raju, R. K. (2017). Dynamic memory inference network for natural language inference. International Journal of Science and Research (IJSR), 6(2). https://www.ijsr.net/archive/v6i2/SR24926091431.pdf

25. Ramuka, M. (2019). *Data analytics with Google Cloud platform*. BPB Publications.

26. Raptis, T. P., & Passarella, A. (2023). A survey on networked data streaming with apache kafka. *IEEE access*, *11*, 85333-85350.

27. Sardana, J. (2022). Scalable systems for healthcare communication: A design perspective. *International Journal of Science and Research Archive*. https://doi.org/10.30574/ijsra.2022.7.2.0253

28. Sardana, J. (2022). The role of notification scheduling in improving patient outcomes. *International Journal of Science and Research Archive*. Retrieved from https://ijsra.net/content/role-notification-scheduling-improving-patient

29. Sarvari, P. A., Khadraoui, D., Martin, S., & Baskurt, G. (2023, August). Next-Generation Infrastructure and Application Scaling: Enhancing Resilience and Optimizing Resource Consumption. In *Global Joint Conference on Industrial Engineering and Its Application Areas* (pp. 63-76). Cham: Springer Nature Switzerland.

30. Shapira, G., Palino, T., Sivaram, R., & Petty, K. (2021). *Kafka: the definitive guide*. " O'Reilly Media, Inc.".

31. Singh, V. (2022). Advanced generative models for 3D multi-object scene generation: Exploring the use of cutting-edge generative models like diffusion models to synthesize complex 3D environments. https://doi.org/10.47363/JAICC/2022(1)E224

32. Sulkava, A. (2023). Building scalable and fault-tolerant software systems with Kafka.

33. Tranquillin, M., Lakshmanan, V., & Tekiner, F. (2023). *Architecting data and machine learning platforms: enable analytics and AI-driven innovation in the cloud*. " O'Reilly Media, Inc.".

34. Van Dongen, G., & Van den Poel, D. (2020). Evaluation of stream processing frameworks. *IEEE Transactions on Parallel and Distributed Systems*, *31*(8), 1845-1858.

35. Vargas, R. F. L. (2022). A performance comparison of data lake table formats in cloud object storages.

36. Wang, G., Chen, L., Dikshit, A., Gustafson, J., Chen, B., Sax, M. J., ... & Rao, J. (2021, June). Consistency and completeness: Rethinking distributed stream processing in apache kafka. In *Proceedings of the 2021 international conference on management of data* (pp. 2602-2613).

37. Wang, Y. (2022). *Evolution of microservice-based applications: Modelling and safe dynamic updating* (Doctoral dissertation, Institut Polytechnique de Paris).

38. Zargar, S., Yao, Y., & Tu, Q. (2022). A review of inventory modeling methods for missing data in life cycle assessment. *Journal of Industrial Ecology*, *26*(5), 1676-1689.