Kafka-Based Real-Time Data Streaming Pipeline

Shubham Kumar Verma

Introduction

This project implements an end-to-end, production-inspired **real-time data streaming pipeline** using modern data engineering best practices.It demonstrates how event-driven data flows through a **multi-layered lakehouse architecture** consisting of *Bronze*, *Silver*, and *Gold* layers, starting from ingestion in Apache Kafka and culminating in analytics-ready, aggregated datasets.

The pipeline leverages **Apache Spark** for scalable data processing and transformation, while enforcing strict data quality guarantees in the Gold layer using **Great Expectations**.
Workflow orchestration is handled using **Prefect**, enabling reproducible execution, retries, and observability across pipeline stages.

To support operational monitoring, the system exposes custom runtime metrics via **Prometheus**, including pipeline execution counts, failures, and end-to-end latency, providing real-time insight into pipeline health and performance.

The entire platform is containerized using **Docker**, allowing the system to be run locally while closely mirroring real-world production environments.
This repository serves both as a hands-on learning artifact and an interview-ready reference implementation of a modern, observable, streaming data platform.

1 Key Features of the Project

This project implements a production-grade streaming data platform by combining modern data engineering principles with industry-standard tools. The design emphasizes reliability, scalability, data quality, and observability across the entire data lifecycle.

1.1 Multi-Layer Data Lake Architecture (Bronze, Silver, Gold)

The pipeline follows the Medallion Architecture pattern, organizing data into distinct layers:

- **Bronze Layer**: Stores raw, immutable events exactly as received from Kafka. This layer serves as a system of record and enables full reprocessing in case of downstream failures.
- **Silver Layer**: Applies schema enforcement, data quality checks, deduplication, and normalization. This layer provides clean and trustworthy data for analytics.

- **Gold Layer**: Contains business-level aggregations and analytical features optimized for reporting, dashboards, and downstream consumption.

This layered approach improves data governance, debugging, and reproducibility.

1.2 Streaming Data Ingestion with Apache Kafka

Apache Kafka is used as the central event streaming platform:

- High-throughput, fault-tolerant message ingestion
- Decouples producers and consumers
- Supports real-time event processing

Kafka enables scalable and reliable data ingestion for continuous streaming workloads.

1.3 Schema Management with Schema Registry

A centralized Schema Registry enforces data contracts between producers and consumers:

- Avro-based schema definitions
- Compatibility checks for schema evolution
- Prevents malformed or incompatible data from entering the pipeline

This ensures strong schema governance and safe schema evolution.

1.4 Fault Tolerance and Exactly-Once Semantics

The pipeline is designed to be fault-tolerant at every stage:

- Kafka offsets ensure message replayability
- Spark checkpointing prevents duplicate processing
- Idempotent writes to the data lake

Failures can be recovered without data loss or duplication.

1.5 Checkpointing and Stateful Processing

Spark checkpointing is used to maintain state consistency during streaming operations:

- Tracks processing progress across micro-batches
- Enables safe restarts after failures
- Supports stateful transformations and aggregations

Checkpointing ensures reliability in long-running streaming jobs.

1.6 Data Quality Enforcement with Great Expectations

Great Expectations is integrated to enforce data quality rules:

- Null checks on critical business keys
- Schema and data type validation
- Contract enforcement between pipeline layers

Data that violates expectations is detected early, preventing bad data from propagating downstream.

1.7 Deduplication and Data Consistency

The Silver layer performs deduplication using business keys and event timestamps:

- Eliminates duplicate events
- Ensures consistent analytical results
- Handles replays and late-arriving data

This guarantees correctness in aggregated outputs.

1.8 Schema Evolution Support

The pipeline supports controlled schema evolution:

- Backward-compatible schema updates
- Graceful handling of new or optional fields
- No disruption to existing data consumers

Schema evolution is validated end-to-end from Kafka ingestion to Gold aggregations.

1.9 Orchestration with Prefect

Prefect is used for workflow orchestration:

- Defines dependencies between pipeline stages
- Handles retries and failure recovery
- Enables controlled execution of batch and streaming jobs

This improves operational reliability and manageability.

1.10 Monitoring and Observability

The pipeline is fully observable using Prometheus and Grafana:

- Custom application metrics exposed via Prometheus
- Pipeline execution duration tracking
- Failure and success monitoring
- Grafana dashboards for real-time visualization

Monitoring ensures visibility into pipeline health and performance.

1.11 Containerized Deployment with Docker

All core services are containerized using Docker:

- Kafka and Zookeeper
- Schema Registry
- Prometheus and Grafana

This guarantees environment consistency and simplifies deployment.

1.12 Production-Ready Design

The project follows production best practices:

- Modular and reusable code structure
- Clear separation of concerns
- Config-driven design
- Scalable and extensible architecture

The result is a robust, end-to-end streaming data platform suitable for real-world use cases.

2 Phase 1 — Data Contracts and Kafka Stream Generation

Phase 1 establishes the foundation of the streaming platform by defining strong data contracts and generating real-time event streams into Apache Kafka. This phase focuses on ensuring schema consistency, producer reliability, and realistic event simulation, which are critical prerequisites for downstream processing and analytics.

Objective

The primary objective of Phase 1 is to create a reliable and contract-driven data ingestion layer. By enforcing schemas at the point of data production, the pipeline guarantees that all downstream consumers receive well-structured and predictable data, reducing runtime failures and schema drift.

Schema Definition and Data Contracts

A formal data contract is defined using an Avro schema located at:

- `schemas/input_event.avsc`

This schema specifies the structure, data types, and required fields for incoming events, such as unique identifiers, user attributes, campaign metadata, and event timestamps. Using Avro enables compact binary serialization and strong schema validation, ensuring backward and forward compatibility as the data model evolves.

Kafka Infrastructure Setup

Kafka and its supporting services are provisioned using Docker Compose:

- `docker/docker-compose.yml`

This configuration initializes Kafka brokers and required dependencies in a containerized environment, closely simulating a production-grade messaging system. Containerization ensures reproducibility, isolation, and ease of deployment across different environments.

Event Generation and Production

A Kafka producer implemented in Python generates synthetic but realistic streaming events:

- `src/producers/faker_avro_producer.py`

The producer uses a data generation library to simulate user behavior and event activity, serializes records according to the Avro schema, and publishes them to Kafka topics in real time. Each message is validated against the schema before being sent, enforcing strict adherence to the defined data contract at ingestion time.

Outcome

At the end of Phase 1, the system produces a continuous, schema-validated stream of events into Kafka. This stream serves as the authoritative source of truth for all subsequent pipeline stages, enabling scalable consumption, transformation, and analytical processing in later phases.

3 Phase 2 — Bronze Layer: Raw Ingestion to Delta Lake

Phase 2 implements the Bronze layer of the data platform, responsible for ingesting raw streaming data from Kafka and persisting it in a durable, append-only storage format. This layer acts as the immutable system of record, preserving original event data while enabling fault tolerance, replayability, and auditability.

Objective

The primary objective of the Bronze layer is to reliably consume Kafka streams and store them with minimal transformation. By maintaining raw event fidelity, the Bronze layer ensures that downstream processing can be re-run or corrected without data loss, which is a core principle of modern data lakehouse architectures.

Kafka Consumption Utilities

Kafka connectivity and consumption logic are abstracted into reusable utility modules:

- `src/utils/kafka_utils.py`

This module encapsulates Kafka consumer configuration, topic subscription logic, and message deserialization. By centralizing Kafka-related functionality, the pipeline promotes code reuse, consistency, and easier maintenance across multiple consumers.

Schema Handling and Validation

Schema management and validation are handled through a dedicated utility layer:

- `src/utils/schema_utils.py`

This component ensures that incoming Kafka messages conform to the expected Avro schema before ingestion. Schema validation at this stage prevents corrupt or incompatible records from entering the data lake, thereby protecting downstream processing layers from unexpected failures.

Bronze Ingestion Consumer

The main ingestion logic is implemented in the Bronze consumer:

- `src/consumers/bronze_ingest.py`

This consumer continuously reads events from Kafka, applies schema validation, and writes the raw data into the Bronze layer using Delta Lake format. The write process is append-only, preserving the original structure and content of each event while leveraging Delta Lake features such as ACID transactions, schema enforcement, and time travel.

Storage Characteristics

The Bronze layer stores data with the following characteristics:

- Raw, minimally transformed event data
- Append-only writes for immutability
- Partitioning based on ingestion or event time
- Delta Lake format for transactional guarantees

These characteristics enable efficient downstream processing, replay capabilities, and strong data lineage tracking.

Outcome

Upon completion of Phase 2, all Kafka events are durably stored in the Bronze Delta Lake. This layer serves as the trusted raw data foundation for subsequent Silver and Gold transformations, enabling scalable analytics, data quality enforcement, and business-level aggregation in later phases.

4 Phase 3 — Silver Layer: Data Quality & Deduplication

Phase 3 introduces the Silver layer, where raw Bronze data is refined into clean, standardized, and analytics-ready datasets. This phase focuses on enforcing data quality rules, removing duplicates, and applying controlled transformations while maintaining traceability to the raw source data.

Objective

The objective of the Silver layer is to transform raw ingested data into a reliable and consistent representation of business events. By applying data quality validations and deduplication logic at this stage, the pipeline ensures that downstream analytical workloads operate on trusted data.

Expectation-Based Data Quality Framework

Data quality rules are defined declaratively using expectation configuration files:

- `configs/expectations/bronze_expectations.json`
- `configs/expectations/silver_expectations.json`

These configuration files specify validation rules such as non-null constraints, data type checks, uniqueness requirements, and value range validations. Separating expectations from execution logic enables flexible rule evolution without modifying core pipeline code.

Reusable Data Quality Utilities

Quality validation logic is centralized in a dedicated utility module:

- `src/utils/quality_utils.py`

This module applies expectation rules to incoming datasets and produces validation results that determine whether records are accepted, quarantined, or rejected. Centralizing quality logic promotes consistency and ensures that data contracts are enforced uniformly across pipeline stages.

Delta Lake Utilities

Delta Lake operations are abstracted into reusable helpers:

- `src/utils/delta_utils.py`

These utilities manage optimized read and write operations, schema enforcement, merge semantics, and transactional consistency. They enable idempotent processing and support incremental updates through merge and upsert strategies.

Silver Transformation Consumer

The Silver processing logic is implemented in:

- `src/consumers/silver_transform.py`

This consumer reads data from the Bronze Delta Lake, applies data quality validations using the defined expectations, removes duplicate records based on business keys and event timestamps, and standardizes data formats and field naming conventions. Only validated and deduplicated records are promoted to the Silver layer.

Deduplication Strategy

Deduplication is performed using deterministic business keys and event-time ordering. This ensures that only the most recent or most accurate version of a record is retained while preserving deterministic behavior during reprocessing.

Outcome

At the end of Phase 3, the Silver layer contains clean, validated, and deduplicated datasets that conform to defined data contracts. This layer forms the foundation for analytical transformations and business aggregations performed in the Gold layer, ensuring accuracy, consistency, and trust in downstream metrics.

5 Phase 4 — Gold Layer: Feature Aggregation

Phase 4 represents the Gold layer of the data pipeline, where cleaned and validated Silver data is transformed into business-level features and aggregated metrics. This layer is designed specifically for analytics, reporting, and downstream consumption by dashboards, machine learning models, and decision-support systems.

Objective

The primary objective of the Gold layer is to convert event-level data into meaningful, query-efficient aggregates that directly reflect business KPIs. All transformations in this phase are deterministic, reproducible, and derived exclusively from the trusted Silver layer.

Gold Aggregation Consumer

The Gold layer processing logic is implemented in the following module:

- `src/consumers/gold_aggregate.py`

This consumer reads validated Silver datasets and applies aggregation logic aligned with analytical requirements. Unlike earlier stages, no raw data cleansing occurs in this phase; instead, the focus is on summarization, enrichment, and feature engineering.

Feature Engineering and Aggregation Logic

The aggregation process groups events along meaningful business dimensions such as campaign identifiers and event dates. Typical aggregations include event counts, engagement metrics, and time-based summaries.

The Gold layer introduces derived fields such as daily aggregates, enabling efficient time-series analysis and reducing the computational cost of downstream queries. All aggregations are computed using event-time semantics to ensure temporal correctness.

Data Model Characteristics

Gold datasets are designed to be:

- Denormalized for fast analytical access
- Optimized for read-heavy workloads
- Stable and schema-consistent across executions

These characteristics make the Gold layer suitable for direct consumption by BI tools and monitoring dashboards.

Outcome

At the conclusion of Phase 4, the pipeline produces a curated Gold dataset containing business-ready features and aggregated metrics. This layer serves as the single source of truth for reporting and analytics, abstracting away the complexity of upstream ingestion, validation, and transformation processes.

6 Phase 5 — Orchestration & Monitoring

Phase 5 focuses on operationalizing the data pipeline by introducing workflow orchestration and observability. This phase ensures that the end-to-end pipeline can be executed reliably, monitored continuously, and debugged efficiently in production-like environments.

Objective

The objective of this phase is to provide automated coordination of pipeline stages and real-time visibility into pipeline health, performance, and failures. By separating orchestration from business logic, the system becomes easier to scale, maintain, and operate.

Pipeline Orchestration

Workflow orchestration is implemented using a directed execution model that coordinates the Bronze, Silver, and Gold layers as a single logical pipeline. The orchestration logic is defined in the following module:

- `docker/airflow/dags/pipeline_dag.py`

This orchestration layer defines task-level dependencies and execution order, ensuring that downstream stages only run after upstream stages complete successfully. It also provides retry mechanisms, centralized logging, and failure propagation across the pipeline.

Execution Semantics

Each pipeline stage is executed as an independent task, enabling modular execution and isolated failure handling. This design allows individual layers to be re-run without recomputing the entire pipeline, improving development velocity and operational flexibility.

Monitoring and Observability

To achieve runtime observability, the pipeline exposes operational metrics that are scraped and stored by a monitoring system. Monitoring configuration is defined in:

- `docker/monitoring/prometheus.yml`

Prometheus is configured to collect metrics from both the Spark execution environment and custom pipeline instrumentation. These metrics include execution duration, pipeline run counts, and failure indicators, enabling proactive monitoring of system health.

Benefits

The integration of orchestration and monitoring provides:

- Automated and repeatable pipeline execution
- Centralized visibility into pipeline performance
- Faster detection and diagnosis of failures
- Improved reliability for production-scale data workflows

Outcome

At the end of Phase 5, the pipeline is fully production-ready, with automated execution, monitoring, and observability in place. This phase transforms the pipeline from a collection of scripts into an operational data platform capable of supporting continuous data ingestion and analytics.

7 Phase 6 — Schema Evolution Demonstration

Phase 6 demonstrates how the data pipeline handles schema evolution in a controlled and backward-compatible manner. Schema evolution is a critical requirement in real-world streaming systems, where data models change over time as business requirements evolve.

Objective

The primary objective of this phase is to validate that the pipeline can safely ingest, process, and transform data when the input schema changes, without breaking downstream consumers or corrupting existing datasets.

Schema Versioning

A new version of the input schema is introduced to simulate schema evolution:

- `schemas/evolution_v2.avsc`

This updated schema represents a realistic evolution scenario, such as adding optional fields, extending event metadata, or refining data types. The evolution is designed to remain backward-compatible with previously ingested data.

Producer Adaptation

The Kafka producer is updated to emit events conforming to the new schema version:

- `src/producers/faker_avro_producer.py`

The producer dynamically switches to the evolved schema while continuing to publish events to the same Kafka topic. This simulates real-world scenarios where producers are upgraded independently of consumers.

Silver Layer Compatibility Handling

The Silver layer transformation logic is enhanced to handle multiple schema versions gracefully:

- `src/consumers/silver_transform.py`

The Silver layer applies schema-aware transformations, ensuring that:

- New fields are handled safely with defaults or nullability
- Existing transformations continue to function correctly
- Data quality checks remain valid across schema versions

This approach prevents pipeline failures caused by unexpected schema changes and preserves data consistency.

Data Integrity and Backward Compatibility

By enforcing schema evolution rules at the Silver layer, the pipeline guarantees that downstream Gold aggregations and analytics remain unaffected. Older records and newly evolved records coexist seamlessly in the same dataset.

Outcome

Phase 6 validates that the pipeline supports schema evolution end-to-end, from producer to analytical outputs. This capability is essential for maintaining long-lived streaming data platforms where schemas inevitably change over time.

8 Conclusion

This project demonstrates the design and implementation of a robust, end-to-end real-time data streaming platform using modern data engineering best practices. By integrating Apache Kafka, Spark-based processing, Delta Lake storage, and a layered data lake architecture, the pipeline ensures scalable, reliable, and high-quality data ingestion and transformation.

The adoption of the Bronze, Silver, and Gold layers enables clear separation of concerns, allowing raw data preservation, systematic data cleansing, and business-focused feature aggregation. Strong schema governance through Schema Registry, combined with automated data quality validation using Great Expectations, ensures data correctness and contract enforcement across all stages of the pipeline.

Fault tolerance is achieved through Kafka offset management, Spark checkpointing, and idempotent writes, allowing the system to recover gracefully from failures without data loss or duplication. Workflow orchestration using Prefect further enhances operational reliability by managing dependencies, retries, and execution order.

Comprehensive monitoring with Prometheus and Grafana provides real-time observability into pipeline health, performance, and execution metrics, making the system production-ready and operationally transparent. Containerization with Docker ensures environment consistency and simplifies deployment across different setups.

Overall, this project serves as a complete reference implementation of a modern streaming data platform, showcasing how real-time ingestion, data quality enforcement, schema evolution, orchestration, and observability can be combined into a scalable and maintainable architecture suitable for real-world data engineering use cases.