NEXA ENTERPRISE DATA PLATFORM | TECHNICAL ARCHITECTURE SPECIFICATION

Synapse Real-Time Order & Telemetry ETL Pipeline

Target Warehouse: Meridian Snowflake Data Lakehouse | **Engine:** Apache Spark + dbt Core

Status: PRODUCTION APPROVED
Doc ID: PL-2026-SYN-088 | **Ver:** v2.4.0
SLA Target: P99 < 180s

Author:	Marcus Vance (Staff DE)	**Tech Lead:**	Dr. Aris Thorne	**Reviewer:**	Elena Rostova (Data Arch)
Source Systems:	Kafka CDC, Postgres, S3	**Orchestrator:**	Prefect Enterprise	**Security Level:**	Tier-1 Confidential / PII

1. Executive Overview & System Objectives

The **Synapse ETL Pipeline** ingests high-frequency transactional order streams and device telemetry logs across global regional hubs into the **Meridian Data Warehouse**. It powers operational dashboards, fraud detection models, and financial compliance reporting for Nexa Commerce and Vertex Cloud Services.

DAILY EVENT VOLUME
450 Million
~ 280 GB Compressed Delta / Day

END-TO-END LATENCY
P99 < 180 Sec
CDC Stream to Gold Warehouse

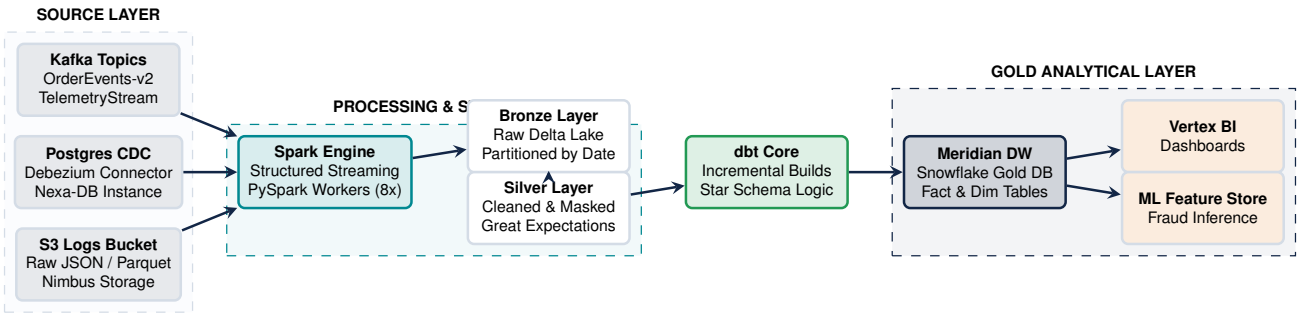
DATA AVAILABILITY
99.99% SLA
Zero Data Loss / DLQ Enabled

1.1 Key Design Principles

- **Medallion Architecture Alignment:** Strict separation between Raw (Bronze), Cleansed/Enriched (Silver), and Business Dimensional (Gold) layers.
- **Idempotent Stream Processing:** All ingestion handlers enforce exact-once delivery semantics via deterministic message key hashing and Snowflake MERGE operations.
- **Automated Schema Evolution:** Schema changes in source Kafka topics are parsed through the Vertex Schema Registry, enforcing backward-compatible migrations.

2. End-to-End Pipeline Architecture

The pipeline topology transitions real-time events from edge data sources through structured streaming layers into analytical gold tables.



3. Data Warehouse Schema & Dimensional Specification

The target analytical layer in **Meridian Snowflake Warehouse** adopts a Kimball Star Schema design optimized for high-throughput aggregation and fast analytical query processing.

3.1 Gold Fact Table: `gold_fact_orders`

This table stores transactional order activity enriched with currency conversion and dimension keys.

Column Name	Data Type	Constraint	Nullable?	Description & Transformation Rules
order_sk	BIGINT	PK	No	Surrogate Key (MD5 Hash of order_id + created_at).
order_id	VARCHAR(64)	BK	No	Natural Order Identifier from Postgres CDC.
customer_sk	INT	FK	No	Foreign Key matching gold_dim_customers.
product_sk	INT	FK	No	Foreign Key matching gold_dim_products.
order_timestamp_utc	TIMESTAMP_NTZ	–	No	Event timestamp converted to UTC. Partition key.
gross_amount_usd	NUMBER(12,2)	–	No	Raw order total converted to USD at event time rate.
tax_amount_usd	NUMBER(10,2)	–	Yes	Calculated tax amount in USD.
order_status	VARCHAR(32)	Check	No	Allowed: PENDING, COMPLETED, REFUNDED, CANCELLED.
is_fraud_flagged	BOOLEAN	–	No	Flag derived from ML inference stream. Default: FALSE.
etl_inserted_at	TIMESTAMP_NTZ	–	No	Pipeline processing timestamp for auditing.

3.2 Gold Dimension Table: gold_dim_customers (SCD Type 2)

Customer dimension preserving historical state updates (e.g., tier changes, address updates).

Column Name	Data Type	Constraint	Nullable?	Description & Business Rules
customer_sk	INT	PK	No	Auto-incrementing Surrogate Key.
customer_id	VARCHAR(64)	BK	No	Unique Customer Natural Identifier.
customer_name_hash	VARCHAR(64)	PII	No	SHA-256 Hashed Customer Name for anonymity.
email_domain	VARCHAR(128)	–	Yes	Extracted domain portion of email (e.g., nexa.io).
account_tier	VARCHAR(32)	–	No	Tiers: STANDARD, PREMIUM, ENTERPRISE.
country_code	CHAR(2)	–	No	ISO 3166-1 alpha-2 country code.
effective_start_date	DATE	–	No	Validity start date for SCD Type 2 record.
effective_end_date	DATE	–	Yes	Validity end date (NULL if current active record).
is_current	BOOLEAN	–	No	TRUE for current record state, FALSE for historical.

3.3 Partitioning, Clustering & Indexing Strategy

Warehouse Optimization Settings

- **Snowflake Clustering Keys:** gold_fact_orders is clustered by (order_timestamp_utc::DATE, customer_sk). Re-clustering runs automatically on 50 GB threshold deltas.
- **Delta Lake Partitioning:** Bronze/Silver Delta tables are partitioned by date=YYYY-MM-DD/hr=HH to accelerate backfill execution and vacuum cycles.
- **Micro-Partition Pruning:** All dbt incremental models enforce a 3-day lookback filter (order_timestamp_utc >= CURRENT_DATE - INTERVAL '3 DAYS') to eliminate unnecessary scan overhead.

4. ETL Logic & Code Specifications

4.1 Incremental Data Transformation (dbt Model)

The following production SQL model executes within dbt to incrementally merge Silver staging records into the gold_fact_orders analytical target table.

```sql
{{ config(
    materialized='incremental',
    unique_key='order_sk',
    cluster_by=['order_timestamp_utc::DATE', 'customer_sk'],
    incremental_strategy='merge'
) }}

WITH silver_staged AS (
    SELECT
        MD5(CONCAT(order_id, ':', created_at)) AS order_sk,
        order_id,
        customer_id,
        product_id,
        DATE_TRUNC('second', created_at) AS order_timestamp_utc,
        ROUND(gross_amount * fx_rate_to_usd, 2) AS gross_amount_usd,
        ROUND(tax_amount * fx_rate_to_usd, 2) AS tax_amount_usd,
        UPPER(order_status) AS order_status,
        COALESCE(fraud_score > 0.85, FALSE) AS is_fraud_flagged,
        CURRENT_TIMESTAMP() AS etl_inserted_at
    FROM {{ ref('silver_orders_cleansed') }}
    {% if is_incremental() %}
        -- Incremental processing lookback window to handle late-arriving data
        WHERE created_at >= (SELECT MAX(order_timestamp_utc) - INTERVAL '3 DAYS' FROM {{ this }})
    {% endif %}
),

dim_customers AS (
    SELECT customer_sk, customer_id
    FROM {{ ref('gold_dim_customers') }}
    WHERE is_current = TRUE
)

SELECT
    s.order_sk,
    s.order_id,
    COALESCE(c.customer_sk, -1) AS customer_sk, -- -1 handles unmapped dimensions
    s.product_id AS product_sk,
    s.order_timestamp_utc,
    s.gross_amount_usd,
    s.tax_amount_usd,
    s.order_status,
    s.is_fraud_flagged,
    s.etl_inserted_at
FROM silver_staged s
```

```sql
LEFT JOIN dim_customers c ON s.customer_id = c.customer_id;
```

Listing 1: dbt Model: `models/gold/gold_fact_orders.sql`

4.2 PySpark Ingestion & PII Sanitization Handler

The raw streaming engine executes PII hashing and schema validation prior to writing Bronze records.

Listing 2: PySpark Processor: `jobs/ingest_orders_stream.py`

```python
from pyspark.sql import SparkSession
from pyspark.sql.functions import col, sha2, concat_ws, current_timestamp, from_json
from pyspark.sql.types import StructType, StructField, StringType, DoubleType, TimestampType

def process_orders_stream(spark: SparkSession, kafka_bootstrap: str):
    schema = StructType([
        StructField("order_id", StringType(), False),
        StructField("customer_id", StringType(), False),
        StructField("customer_name", StringType(), True),
        StructField("gross_amount", DoubleType(), False),
        StructField("timestamp", StringType(), False)
    ])

    raw_stream = spark.readStream \
        .format("kafka") \
        .option("kafka.bootstrap.servers", kafka_bootstrap) \
        .option("subscribe", "nexa.synapse.orders.v2") \
        .option("startingOffsets", "latest") \
        .load()

    parsed_df = raw_stream \
        .select(from_json(col("value").cast("string"), schema).alias("data")) \
        .select("data.*")

    # Enforce PII Masking: Hash customer_name using SHA-256 with static platform salt
    sanitized_df = parsed_df \
        .withColumn("customer_name_hash", sha2(concat_ws("::", col("customer_name"), col("customer_id")), 256)) \
        .drop("customer_name") \
        .withColumn("ingested_at", current_timestamp())

    query = sanitized_df.writeStream \
        .format("delta") \
        .outputMode("append") \
        .option("checkpointLocation", "s3a://nexa-synapse-checkpoints/orders_v2/") \
        .partitionBy("ingested_at") \
        .start("s3a://nexa-synapse-lake/bronze/orders/")

    return query
```

5. Data Quality Controls, Validation Rules & SLAs

Data quality is verified at each pipeline boundary using automated assertions powered by Great Expectations and Soda Core.

5.1 Automated Validation Matrix

Rule ID	Target Field	Assertion Check	Threshold / SLA	Action on Failure
`DQ-ORD-01`	`order_id`	Expect column values to never be NULL	100% Compliance	Quarantine to DLQ
`DQ-ORD-02`	`gross_amount`	Expect value to be strictly positive (> 0.00)	99.99% Row Pass	Quarantine to DLQ
`DQ-ORD-03`	`created_at`	Timestamps within 24h of system time	Max Drift < 15 min	Trigger Slack Alert
`DQ-CUST-04`	`customer_sk`	Foreign Key integrity matching `dim_customers`	$> 99.50\%$ Match	Assign Default `-1`
`DQ-LAT-05`	End-to-End Stream	Stream ingestion latency from Kafka source	P99 < 180 Seconds	Page On-Call DE

5.2 Dead Letter Queue (DLQ) & Error Recovery Protocol

Dead Letter Queue (DLQ) Execution Workflow:

When a record violates critical assertions (`DQ-ORD-01` or `DQ-ORD-02`), it is routed to the S3 quarantine location `s3://nexa-synapse-dlq/orders/`.

1. **Alert Notification:** Prefect fires an automated incident payload to PagerDuty (`#alerts-data-ops`).
2. **Inspection & Patching:** Engineers inspect quarantine payloads using the Nexa DataOps CLI Tool:

```
nexa-dataops dlq inspect -pipeline PL-2026-SYN-088 -date 2026-07-24
```

3. **Replay Trigger:** Fixed records are re-published to the replay Kafka topic `nexa.synapse.orders.replay` for re-processing.

6. Security, Governance & Operational Runbook

6.1 Access Control Matrix (RBAC)

Data access in **Meridian Warehouse** adheres to the principle of least privilege using Snowflake role hierarchies.

Role Name	Bronze Layer	Silver Layer	Gold Layer Access
NEXA_DATAOPS_ADMIN	FULL ACCESS	FULL ACCESS	FULL ACCESS (DDL + DML)
SYNAPSE_TRANSFORMER	READ ONLY	READ + WRITE	READ + WRITE (dbt Executor)
VERTEX_BI_ANALYST	NO ACCESS	NO ACCESS	READ ONLY (PII Masked Columns)
MERIDIAN_AUDITOR	NO ACCESS	READ (Audit Logs)	READ ONLY (Compliance Views)

6.2 Operational Runbook & Incident Recovery Commands

The following standard operating procedure (SOP) applies when recovering from upstream Kafka ingestion halts or performing manual data backfills.

DataOps Operational Backfill Procedure

Scenario: Source system outage required a 48-hour data backfill for period `2026-07-20` to `2026-07-22`.

1. **Pause Real-Time Orchestrator:** Disable the Prefect schedule to avoid concurrent writing conflicts:

```
prefect deployment pause "synapse-etl-pipeline/production-stream"
```

2. **Execute Backfill PySpark Job:** Submit the backfill Spark job targeting the specific historical date partition range:

```
spark-submit --master k8s://https://k8s.nexa.internal:6443 \
  --deploy-mode cluster --num-executors 16 --executor-memory 8G \
jobs/backfill_orders.py --start-date 2026-07-20 --end-date 2026-07-22
```

3. **Re-run dbt Incremental Models with Full Refresh Filter:**

```
dbt run --select path:models/gold/gold_fact_orders.sql --vars '{"backfill": true, "start_date": "2026-07-20"}'
```

4. **Run Data Reconciliation Suite & Unpause:** Verify row counts match source Postgres CDC offsets, then resume real-time deployment:

```
dbt test --select test_type:singular && prefect deployment resume "synapse-etl-pipeline/production-stream"
```

— **END OF TECHNICAL SPECIFICATION** | Nexa Data Platform Architecture Board —