Modern Data Platform for ELT Data into Snowflake + AWS

Anurag Sharma 10 min read · Nov 4, 2025

This blog describes a production-ready data platform built for a Data Team that handles data from multiple third-party vendors. The solution leverages AWS S3 for landing, Snowflake (external tables + internal tables) for storage and compute, dbt Core for transformations (orchestrated by Airflow + EMR), and Airflow for orchestration + DQ checks.

The primary goals are ingesting new 3rd-party data streams, restructuring the data framework, and improving the platform for scalability, governance, and downstream analytics (Power BI).

Role of CDM — Ingestion team in Data Migration and Platform Modernization

Migration Context

- **Project Goal:** Migrating from legacy data platform to modern AWS-based architecture for enhanced data control, scalability, and analytics capabilities.
- **Legacy Challenge:** Complex, inconsistent vendor data formats stored in outdated database designs that are difficult to maintain, query, and

integrate with modern tools.

CDM's Core Responsibilities:

- **Data Collection:** Ingests data from multiple investment and wealth management vendors in various legacy formats (XML, fixed-width files, proprietary formats, legacy APIs).
- **Parsing and Standardization:** Transforms disparate vendor formats into consistent, normalized data structures.
- **Simplification:** Removes unnecessary complexity from old database designs, creating streamlined, standardized tables optimized for modern analytics.
- **Aggregation:** Consolidates related data streams from multiple sources into unified domain-specific datasets.

```
Legacy Vendors (XML/Fixed-Width/APIs)
    ↓
CDM Platform (Parse + Normalize + Aggregate)
    ↓
Standardized JSON/Tabular Format
    ↓
AWS SDK → Kinesis Data Firehose
    ↓
Multiple S3 Buckets (Compressed CSV)
```

CDM Processing Steps:

1. **Ingestion:** Collect data from vendors via scheduled pulls, API calls, or file transfers

2. **Parsing**: Convert legacy formats (XML, fixed-width, proprietary) into structured data
3. **Normalization**: Apply business rules, data cleansing, and standardization
4. **Aggregation**: Combine related datasets and create unified views
5. **Publishing:** Push processed data to Kinesis Data Firehose using AWS SDK for reliable delivery

Why Firehose Targets CSV (Compressed)

Operational Benefits

- Universal Readability: CSV format is accessible to audit teams, business users, and compliance teams without specialized tools
- Ad-hoc Analysis: Compatible with S3 Select and Athena for quick exploration without provisioning compute resources
- Legacy Alignment: Matches tabular structure of financial data, making migration validation easier
- Operational Simplicity: Minimizes schema evolution concerns at the RAW tier; downstream transformations handle complexity

Technical Advantages

- Efficient Compression: GZIP compression significantly reduces storage costs for structured financial records
- Streaming Compatibility: Works seamlessly with Firehose's batching and delivery mechanisms
- Audit Trail: Easy to download and inspect individual files for data lineage and compliance verification

CDM-Managed Organization Strategy

Domain-Based Delivery

- Portfolio Data: Position holdings, asset allocations, performance metrics
- Transaction Data: Trade executions, settlements, corporate actions
- Valuation Data: Market prices, NAV calculations, benchmark data
- Reference Data: Security master, client demographics, account hierarchies

S3 Buckets (RAW Landing):

Multiple target buckets by domain to segment access and governance:

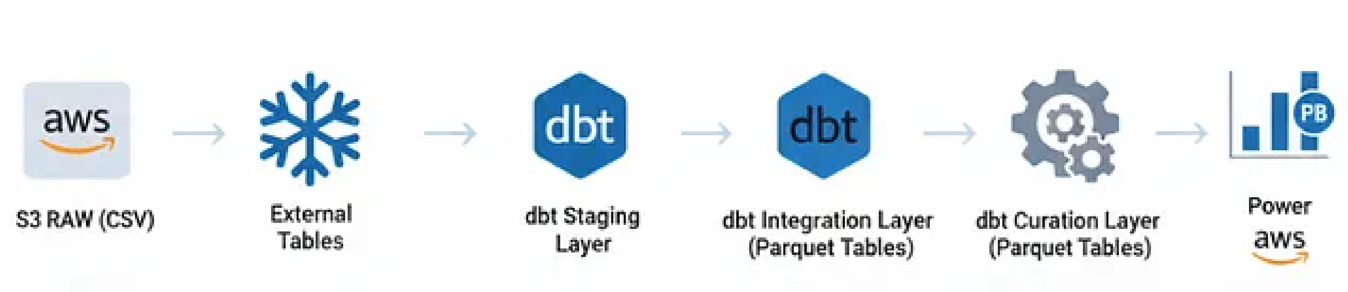
- *company-portfolio-raw*
- *company-transactions-raw*
- *company-valuations-raw*
- *company-reference-raw*

Standardized partitioning:

```
vendor={vendor}/domain={domain}/year=YYYY/month=MM/day=DD/hour=HH/
```

Data Engineering Transformation: dbt + Snowflake + Airflow Orchestration

The Data Engineering team transforms RAW CSV data delivered by the CDM team into curated, analytics-ready datasets using a modern ELT approach. Our technology stack combines dbt for transformations, Snowflake as the analytical data warehouse, and Airflow for orchestration — all working together to deliver reliable, scalable data pipelines.



Component	Purpose	Ownership
dbt Core	Transformation logic & data modeling	Data Engineering
Snowflake	Data storage & compute engine	Data Engineering
Airflow	Pipeline orchestration & scheduling	DevOps/Admin Team

Snowflake in the Modern Data Platform

Role of Snowflake:

- Central analytical warehouse for transformation, storage, and serving curated datasets.
- Bridges RAW data in S3 to analytics-ready models consumed by downstream tools (e.g., Power BI).
- Executes dbt transformations using Snowflake compute for scalability and cost control.

Storage Layers and Objects:

- External Stages: Point to S3 RAW buckets for file access.
- External Tables: Reference CSV (GZIP) files in RAW for staging views.
- Internal Tables: Persist integration and curation layers for performance and governance.
- Transient Tables: Used for TEMP/DEDUP intermediate processing to reduce storage cost.

Layered Modeling Strategy

Staging (Views over External Tables)

- Purpose: Light cleaning, renaming, type casting.
- Benefits: Zero storage, quick reflection of RAW changes.

Integration (Tables, insert_overwrite)

- Purpose: Business logic, cross-vendor joins, harmonization.
- Storage: Internal tables in Parquet-like efficiency via micro-partitions.

TEMP → DEDUP → CORE (Custom SCD2)

- Purpose: Handle back-dated corrections, vendor precedence, and change tracking.
- Artifacts: temp_, *dedup_*, core_* tables with audit columns and history.

Curation (Tables)

- Purpose: Analytics-ready, column-select only, stable contracts.
- Optimizations: Clustering keys, pruning, and query acceleration.*_

Performance and Cost Optimization

- Warehouses: Separate sizes per env/workload (ETL vs Ad-hoc vs BI), auto-suspend/auto-resume.
- Clustering: Define keys on high-cardinality filter columns (e.g., load_date, vendor, portfolio_id).
- Pruning: Leverage partition-style columns in external table locations for efficient scans.
- Caching: Result and data cache to accelerate repetitive queries.
- Resource Monitors: Budgets and alerts for credit usage.

Governance and Security

- RBAC: Schema- and database-level roles (RAW_READ, INT_WRITE, CURATION_READ).
- Least Privilege: DE owns up to curation; Analysts have read-only access to curated schemas.
- Data Contracts: Enforced via dbt tests and Snowflake constraints where applicable.
- Auditability: Query history, access history, and object tagging for lineage and compliance.

Orchestration and Operations

- dbt-Snowflake Adapter: Executes models on Snowflake compute from Airflow workers.
- External Table Refresh: Automated metadata refresh for new S3 partitions before staging queries.
- Observability: Query profiling, warehouse utilization, and dbt run artifacts stored and monitored.
- Reliability: Idempotent insert_overwrite patterns; controlled retries via Airflow with small, consistent batches.

dbt Transformation Strategy

Layer-by-Layer Materialization

Staging Layer (`stg_*`)

- Materialization: Views only
- Purpose: Light data cleaning, column renaming, type casting
- Source: Snowflake external tables pointing to S3 RAW files
- Advantage: No storage cost, immediate reflection of source changes

```
-- Example: stg_portfolio.sql
{{ config(materialized='view', schema='staging') }}

SELECT
    event_timestamp,
    portfolio_id,
    asset_class,
    market_value,
    currency,
    source_system,
```

```
    DATE(event_timestamp) AS load_date
FROM {{ source('raw', 'ext_table') }}
```

Integration Layer (`int_*`)

- Materialization: Tables with `insert_overwrite` strategy
- Format: Parquet for optimal query performance
- Purpose: Business logic application, cross-vendor joins, data harmonization

```
-- Example: int_portfolio_positions.sql
{{ config(
    materialized='table',
    schema='integration',
    file_format='parquet',
    incremental_strategy='insert_overwrite',
    partition_by=['load_date']
) }}

SELECT
    {{ dbt_utils.generate_surrogate_key(['vendor', 'portfolio_id', 'load_date'])
    vendor,
    portfolio_id,
    asset_class,
    SUM(market_value) AS total_market_value,
    load_date
FROM {{ ref('stg_portfolio') }}
GROUP BY vendor, portfolio_id, asset_class, load_date
```

Custom SCD2/Processing (`temp_* → dedup*_`) [If Required Sometimes]

- Materialization: Transient tables for intermediate processing
- Purpose: Handle complex Slowly Changing Dimension Type 2 logic
- Pattern:

INT → TEMP (current + new records) → DEDUP (business rules) → CORE

Curation Layer (`core_*`, `curation*_`)

- Materialization: Final physical tables
- Purpose: Business-ready datasets optimized for analytics
- Features: Partitioning, clustering, performance tuning

```
-- Example: portfolio_positions.sql
{{ config(
    materialized='table',
    schema='core',
    alias='portfolio_positions',
    tag='curation',
    file_format='parquet',
    incremental_strategy='insert_overwrite',
    partition_by=['as_of_date'],
    cluster_by=['portfolio_id', 'security_id']
) }}

SELECT
    as_of_date,
    portfolio_id,
    security_id,
    currency,
    quantity,
    market_value,
    cost_basis,
    market_value_usd,
    cost_basis_usd,
    pnl_pct
FROM {{ ref('int_portfolio_positions') }}
```

```
Integration Layer (int_*)
    ↓
TEMP Layer (temp_*) - Combine current + new records
    ↓
DEDUP Layer (dedup_*) - Apply business rules & detect changes
```

```
       ↓
CORE Layer (core_*) - Final SCD2 table with history
```

Why Custom SCD2 vs dbt Built-in

Business Requirements Driving Custom Implementation

Complex Vendor-Specific Logic

- Investment data requires different merge rules per vendor
- Handling of back-dated corrections and restatements
- Vendor precedence and conflict resolution

Enhanced Audit & Traceability

- Detailed change tracking beyond standard SCD2
- Explicit change reasons and business rule application
- Comprehensive reconciliation capabilities

Flexibility in Processing

- Ability to chain multiple intermediate steps
- Manual intervention points for edge cases

- Custom validation gates between stages

YAML-to-Airflow: Conversion Flow, Components, and Generated DAG

Goal: Convert a concise, declarative YAML spec into a production-grade Airflow DAG without engineers writing Python.

Benefits: Standardization, security (no secrets in YAML), faster onboarding, and consistent observability.

Ownership: Data Engineering authors YAML; DevOps/Admin maintains the converter and Airflow deployment.

```
pipeline:
  name: "curation_daily_portfolio_etl"
  description: "Daily curation layer processing"

  schedule:
    start_from: "2025-01-01"
    end_at: None
    schedule_interval: "0 6 * * *"

  ownership:
    job_owner: "data-eng@company.com"
    data_owner: "investment-ops@company.com"

  tasks:
    dependency_check:
      type: "s3_sensor"
      config:
        bucket: "company-raw"
        prefix: "year={{ ds_nodash[:4] }}/month={{ ds_nodash[4:6] }}/day={{ ds_n
        timeout: 3600
      dependencies: []

    dbt_curation:
      type: "dbt_command"
      config:
```

```
        command: "run"
        select: "portfolio_positions.sql"
        target: "prod"
      dependencies: ["dependency_check"]

    dbt_tests:
      type: "dbt_command"
      config:
        command: "test"
        select: "tag:curation"
        target: "prod"
      dependencies: ["dbt_curation"]

    success_notification:
      type: "slack_notification"
      config:
        channel: "#etl-alerts"
        message: "Curation ETL completed for {{ ds }}"
      dependencies: ["dbt_tests"]

    failure_notification:
      type: "email_notification"
      config:
        recipients: ["{{ job_owner }}", "{{ data_owner }}"]
        subject: "FAILED: Curation ETL {{ ds }}"
      trigger_rule: "one_failed"
```

End-to-End Flow

Author YAML

- Data Engineer commits pipeline YAML to repo: pipelines/curation_daily_portfolio_etl.yaml

DAG Generation

- Converter parses YAML, maps tasks to Airflow operators, injects defaults, and assembles dependencies.

Secret & Config Binding

- Replace logical names with Airflow Connections/Variables (Snowflake, Slack, email lists).

DAG Registration

- CI publishes generated Python DAG to airflow/dags/ with versioned filename (e.g., curation_daily_portfolio_etl_v1.py)

Airflow Scheduler

- Detects new DAG file, parses it, and schedules runs per cron.

Runtime

- Sensors gate execution.
- dbt tasks run using Snowflake compute (via dbt-snowflake adapter on Airflow workers).
- Tests and notifications execute.
- XComs capture metrics and states.

```
# airflow/dags/curation_portfolio_daily_etl_v1.py
from datetime import datetime, timedelta
from airflow import DAG
from airflow.models.baseoperator import chain
from airflow.operators.empty import EmptyOperator
from airflow.providers.amazon.aws.sensors.s3 import S3KeySensor
from airflow.providers.slack.operators.slack_webhook import SlackWebhookOperator
from airflow.operators.email import EmailOperator
from airflow.operators.bash import BashOperator
```

```
from airflow.operators.python import PythonOperator
from airflow.utils.trigger_rule import TriggerRule
from airflow.utils.state import State
from airflow.hooks.base import BaseHook
import os

DAG_ID = "curation_portfolio_daily_etl_v1"

default_args = {
    "owner": "data-eng@company.com",
    "email": ["data-eng@company.com", "investment-ops@company.com"],
    "email_on_failure": True,
    "retries": 2,
    "retry_delay": timedelta(minutes=10),
    "sla": timedelta(hours=2),
}

with DAG(
    dag_id=DAG_ID,
    description="Daily curation layer processing",
    start_date=datetime(2025, 31, 12),
    schedule_interval="0 6 * * *",
    catchup=True,
    default_args=default_args,
    tags=["curation", "dbt", "snowflake"],
    max_active_runs=1,
) as dag:

    start = EmptyOperator(task_id="start")

    dependency_check = S3KeySensor(
        task_id="dependency_check",
        bucket_key="year={{ ds_nodash[:4] }}/month={{ ds_nodash[4:6] }}/day={{ d
        wildcard_match=True,
        bucket_name="company-raw",
        aws_conn_id="aws_default",
        poke_interval=60,
        timeout=3600,
        mode="reschedule",
        soft_fail=False,
    )

    # Helper to build dbt command (Snowflake compute via dbt-snowflake)
    DBT_PROJECT_DIR = "/opt/airflow/dags/dbt_project"
    DBT_PROFILES_DIR = "/opt/airflow/dags/.dbt"
    DBT_PROFILE_TARGET = "prod"

    dbt_curation = BashOperator(
        task_id="dbt_curation",
        bash_command=(
```

```
                "cd {{ params.project_dir }} && "
                "dbt deps --profiles-dir {{ params.profiles_dir }} && "
                "dbt run --profiles-dir {{ params.profiles_dir }} "
                "--target {{ params.target }} --select 'portfolio_positions.sql'"
            ),
            params={
                "project_dir": DBT_PROJECT_DIR,
                "profiles_dir": DBT_PROFILES_DIR,
                "target": DBT_PROFILE_TARGET,
            },
            env={
                # Airflow Snowflake connection envs can be mapped in profiles.yml us
                "DBT_ENV_SECRET_SNOWFLAKE_ACCOUNT": "{{ conn.snowflake_default.extra
            },
        )

        dbt_tests = BashOperator(
            task_id="dbt_tests",
            bash_command=(
                "cd {{ params.project_dir }} && "
                "dbt test --profiles-dir {{ params.profiles_dir }} "
                "--target {{ params.target }} --select 'portfolio_positions.sql'"
            ),
            params={
                "project_dir": DBT_PROJECT_DIR,
                "profiles_dir": DBT_PROFILES_DIR,
                "target": DBT_PROFILE_TARGET,
            },
        )

        success_notification = SlackWebhookOperator(
            task_id="success_notification",
            http_conn_id="slack_webhook",
            message="Curation ETL completed for {{ ds }}",
            channel="#etl-alerts",
            username="airflow",
            trigger_rule=TriggerRule.ALL_SUCCESS,
        )

        failure_notification = EmailOperator(
            task_id="failure_notification",
            to=["data-eng@company.com", "investment-ops@company.com"],
            subject="FAILED: Curation ETL {{ ds }}",
            html_content="Airflow DAG {{ dag.dag_id }} failed for {{ ds }}. Please i
            trigger_rule=TriggerRule.ONE_FAILED,
        )

        end = EmptyOperator(task_id="end")

        # Graph
```

```
    chain(
        start,
        dependency_check,
        dbt_curation,
        dbt_tests,
        [success_notification, failure_notification],
        end,
    )
```

Reporting to Power BI and Deployment Lifecycle

- The Analyst team owns all Power BI datasets, dataflows, and reports.
- The Data Engineering (DE) team's responsibility ends at the Snowflake curation layer.
- DE has no access to Power BI (workspaces, datasets, or gateways) for security and separation of duties.

Role	Access Scope	Tools	Notes
Data Engineering	Snowflake curation schemas only	Snowflake, dbt, Airflow	Can create/modify curated tables; cannot publish to Power BI
Data Analysts	Read to Snowflake curation + Power BI workspaces	Power BI Desktop/Service, Gateways	Build datasets, define relationships, measures, reports
Security/Admin	Connections and credentials	Azure AD/Entra, Snowflake, Gateway	Manages service principals, secrets, and audit logs

Environment Promotion: Dev → UAT → Prod

Lifecycle Overview:

Stage	Who	What Moves	Quality Gates	Outputs
Development (Dev)	Developer	Table-wise dbt models, tests, docs	Code review, unit tests, style checks	Merge-ready branch
User Acceptance Testing (UAT)	Peer Developer (assigned story)	Same artifacts promoted to UAT target	Peer UAT checklist, data validation against acceptance criteria	UAT sign-off comment
Production (Prod)	Developer + Scrum Master + Product Owner	Tagged release	CAB/bi-weekly approval, change ticket, rollback plan	Production deployment

Development (Sprint/Agile, Table-Wise)

Working Pattern: Plan sprints with granular stories: one table/model per story, wherever possible.

Branching: Feature branch per table/model (e.g., feature/<Story_ID>).

Build:

- Implement dbt model + schema.yml tests + documentation.
- Materializations aligned to layer standards (staging=view, integration=table, core/curation=table).

Validate locally:

- dbt build — select <model_name>+
- Ensure references resolve, tests pass, and performance is acceptable.

Pull Request:

- Automated checks: dbt parse/compile, unit tests, style, column contract validation.
- Reviewer confirms no business logic in CORE (column-select only).

UAT Testing (Peer-Driven with Assigned Story)

Data Correctness:

- Row counts and aggregates match expected ranges.
- Join cardinalities as designed (no unexpected fan-outs).

- Spot-check sample rows back to RAW/source when feasible.

Contract and Schema:

- Column names, data types, and nullability match the agreed contract.
- Surrogate keys, primary keys, and unique constraints validated via dbt tests.

Incremental Behavior:

- Validate insert_overwrite or incremental logic across two or more days' loads.
- Ensure late-arriving/backfilled data behaves per spec.

Performance:

- Query timings under target thresholds on curation tables.
- Proper partitioning/clustering verified on Snowflake.

Documentation and Lineage:

- dbt docs updated (descriptions, sources, tests).
- Lineage shows correct upstream dependencies via ref().

Backward Compatibility:

- No breaking changes to existing curation schemas without versioned rollout or clear migration notes.

UAT Execution Steps:

Deploy feature branch to UAT target profile —

Execute UAT checklist:

- Counts, uniqueness, not_null, and accepted_values tests pass.
- Business rule validations pass (custom tests/queries).
- Performance sampling on representative queries.

Peer Sign-off:

- Peer developer records results in the story, attaches SQL queries and screenshots, or saves results.
- If issues are found, loop back to Dev; otherwise, mark the story "UAT Passed".

Production Deployment (Bi-Weekly, Approval Required)

Change Ticket/CRQ:

- Create a change request with scope: models impacted, schemas, and expected data volume.
- Validation queries.

Release Tag:

- Tag the repo (e.g., release-YYYYMMDD).
- Freeze changes for the release train.

Approvals:

- Scrum Master: Confirms sprint scope, readiness, and CAB schedule.
- Product Owner: Confirms business acceptance and priorities for release.
- Optional CAB/Change Manager: Confirms window and risk level.

Deployment Steps:

Promote to Prod:

- Merge release tag to main; CI deploys dbt artifacts to Prod.
- Airflow variables/connections already configured; no YAML secrets.

Post-Deployment Validation:

- Run predefined validation queries:
- Row counts for key tables for the current partition.
- Critical metrics (e.g., sums of market_value_usd) within tolerance bands.
- Monitor Snowflake credits and warehouse queues.

Communication:

- Share release notes with the Analyst team (schema changes, new tables, deprecations).
- Confirm Power BI datasets are unaffected or provide migration guidance.

Key Takeaways from Modern Data Engineering Pipeline Architecture

1. Layered Architecture

Staging (views), Integration (business logic), Core/Curation (column-select only) with distinct materialization strategies.

2. dbt Dependency Management

Use ref() functions; run only curation models in Airflow, dbt auto-resolves upstream dependencies.

3. YAML-Driven DAGs

Convert declarative YAML to production Python DAGs automatically for standardization and security.

4. Snowflake-Centric Execution

dbt runs on Snowflake compute; DE owns the curation layer, Analysts own Power BI with role-based access.

5. Structured Deployment

Dev (table-wise sprints) → UAT (peer validation) → Prod (bi-weekly approvals) with quality gates.

Dbt AWS Snowflake Airflow Data Engineering