Building an end to end Data Analytics using Snowflake, Airflow, dbt, and a BI tool

Yuyao Ding Naga Sai Haritha Gottumukkala

***Abstract*—This project extends the Lab 1 stock price prediction system by implementing a comprehensive ELT pipeline with advanced data transformations and interactive visualization capabilities. Building upon Lab 1's automated ETL workflow using Apache Airflow to fetch stock data from yfinance and load it into Snowflake, Lab 2 extends this by implementing proper data transformations using dbt and creating interactive dashboards in Preset.**

The core contribution of Lab 2 is the implementation of a three-tier dbt architecture comprising staging models for data cleaning, intermediate models for calculating technical indicators (Moving Averages, RSI, MACD, Bollinger Bands), and marts for consolidated analytical views. This modular design ensures data lineage, reusability, and maintainability. The dbt transformation pipeline is fully integrated into the existing Airflow DAG, executing automatically on a daily schedule with comprehensive data quality tests to ensure accuracy.

All transformations are orchestrated through Airflow, where the ETL task first ingests raw data into Snowflake's `fact_stock_price_daily` table using transaction-based MERGE operations for idempotency, followed by sequential dbt tasks (debug, deps, snapshot, run, test) that transform the data through multiple layers. The staging layer (`stg_stock_daily`) performs data quality checks and basic calculations, the intermediate layer computes technical indicators in parallel, and the mart layer (`mart_stock_daily_metrics`) consolidates all metrics into a unified analytical table.

Finally, we utilized Preset to create interactive dashboards that visualize key trading signals, price trends, and volatility indicators, enabling users to analyze stock performance without writing SQL. The system demonstrates modern data engineering best practices by separating concerns across ETL (Airflow), transformation (dbt), storage (Snowflake), and presentation (Preset) layers, resulting in a scalable and maintainable analytics platform.

***Index Terms*—ELT pipeline, dbt, Apache Airflow, Snowflake, Preset, stock analytics, technical indicators**

I. Introduction

A. Problem Statement

After completing Lab 1, we established an automated ETL pipeline that successfully fetched stock prices from yfinance and loaded them into Snowflake. However, the raw data stored in tables containing only basic price fields (open, high, low, close, volume) was insufficient for comprehensive financial analysis.

Traders and analysts require calculated technical indicators such as moving averages to identify trends, RSI to detect overbought or oversold conditions, and volume patterns to confirm price movements. Without these calculations, users would need to export data to Excel and manually calculate indicators—a process that neither scales nor integrates well with automated workflows.

Furthermore, our Lab 1 implementation mixed data extraction and transformation logic within Python code. When adding new calculations, we had to modify the ETL script and redeploy the entire DAG, violating the separation of concerns principle and making the system difficult to maintain.

We also lacked systematic data quality testing. While we checked for null values in Python, there was no mechanism to validate complex business rules or detect calculation errors, leaving potential data quality issues undetected.

The most significant limitation was accessibility. All analytics resided in Snowflake tables, requiring SQL knowledge for every query. Business users could not explore data themselves and had to request custom queries or export data to create charts, severely limiting the system's usability and adoption.

B. Requirements

To address these limitations, Lab 2 introduces the following enhancements:

Keep Lab 1 ETL Working: Our Airflow pipeline from Lab 1 continues to fetch 180 days of stock data from yfinance for AAPL and NVDA daily, loading data into Snowflake using MERGE operations to avoid duplicates.

Add dbt Transformation Layer: We implemented a three-tier dbt architecture with staging models for data cleaning, intermediate models for calculating technical indicators, and marts for consolidated analytical views.

Connect dbt to Airflow: After the ETL task loads data successfully, Airflow automatically executes dbt commands in sequence using BashOperator.

Add Data Quality Tests: We created comprehensive tests in dbt to verify data integrity, including null value checks, indicator range validations, duplicate detection, and date sequencing verification.

Build Preset Dashboards: We connected Preset to Snowflake and created interactive dashboards displaying stock price trends, technical indicators, and trading signals, allowing users to analyze data without writing SQL.

Follow Best Practices: We used Airflow Variables for configuration, Airflow Connections for credentials, transaction-based MERGE operations for idempotency, and comprehensive error handling throughout the pipeline.

C. Specifications

Technology Stack

- **Orchestration**: Apache Airflow 2.x in Docker
- **Data Warehouse**: Snowflake (ASH_DB database, STOCK schema, MINNOW warehouse)

- **Transformation**: dbt Core 1.5+
- **Data Source**: yfinance library
- **Visualization**: Preset

II. System Architecture

A. Overall System Diagram

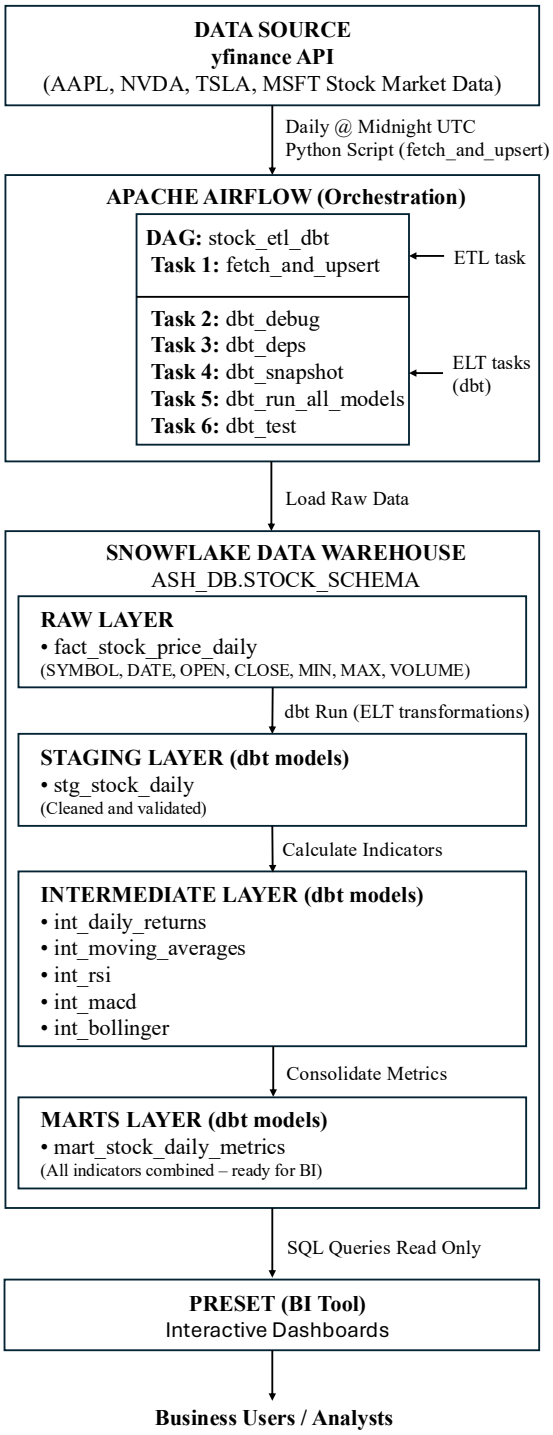


Fig. 1. End-to-end system architecture from data ingestion to visualization.

Figure 1 illustrates the end-to-end architecture of our stock analytics system. The data flows from yfinance API through Apache Airflow orchestration, gets loaded into Snowflake's multi-layered data warehouse, transformed by dbt models, and finally visualized in Preset dashboards for business users.

B. Data Flow

Step 1: Data Extraction (Airflow ETL Task)

Every day at midnight UTC, the Airflow DAG `stock_etl_dbt` initiates the `fetch_and_upsert` task, which retrieves stock symbols from the Airflow Variable `yfinance_symbols` (default: ["AAPL", "NVDA", "TSLA", "MSFT"]).

For each symbol, the task calls the yfinance library to download 180 days of historical data. The yfinance API returns data as a pandas DataFrame with columns: Date, Open, High, Low, Close, and Volume. Column names are standardized to match Snowflake conventions (High $\rightarrow$ MAX, Low $\rightarrow$ MIN), preserving Date, Open, Close, and Volume. A SYMBOL column is added with the ticker name.

The function implements graceful error handling. If yfinance fails for one symbol (e.g., due to network issues), it logs the error and continues with the next symbol, preventing a single failure from breaking the entire pipeline.

Step 2: Data Cleaning (Python)

Before loading to Snowflake, the data undergoes cleaning in pandas. Rows with null OPEN or CLOSE values are removed (essential for technical analysis). VOLUME null values are filled with 0, and the DATE column is converted to proper date format. This cleaning ensures only valid data enters the warehouse.

Step 3: Data Loading (Airflow to Snowflake)

The cleaned data is loaded into Snowflake using SnowflakeHook with credentials stored in Airflow Connections (`snowflake_default`). The connection details include server account, username, password, and warehouse configuration (ASH_DB.STOCK, BEETLE_QUERY_WH warehouse).

A transaction-based approach ensures data consistency:

1) `BEGIN` - Start transaction
2) Create temporary table `tmp_fact_stock_price_daily` with identical structure to the target table
3) Insert new data into the temporary table using `executemany`
4) Execute MERGE statement:
   - `WHEN MATCHED`: Update existing records with new values
   - `WHEN NOT MATCHED`: Insert new records
5) `COMMIT` - Commit all changes

If any error occurs during this process, `ROLLBACK` is executed to undo all changes, ensuring the database remains consistent—either all data is loaded or none of it.

Step 4: dbt Debug and Dependencies (Airflow Tasks 2 and 3)

After ETL succeeds, Airflow executes the dbt tasks. The `dbt_debug` task validates the dbt configuration and

Snowflake connection (`profiles.yaml`), verifying connectivity to ASH_DB.STOCK. If this fails, a connection problem is identified before attempting model execution.

The `dbt_deps` task installs required dbt packages defined in `packages.yml`, ensuring all dependencies are available for subsequent transformations.

Step 5: dbt Snapshot (Airflow Task 4)

The `dbt_snapshot` task captures historical changes in the `fact_stock_price_daily` table using snapshot functionality. This creates `stock_price_snapshot` table that tracks how stock prices change over time with validity timestamps (`dbt_valid_from`, `dbt_valid_to`), enabling historical analysis and time-travel queries.

Step 6: dbt Run - Staging Layer (Airflow Task 5)

The `dbt_run_all_models` task executes all transformation models in dependency order. First, the staging model `stg_stock_daily` runs, which:

- Reads from `fact_stock_price_daily`
- Performs data quality checks (identifying invalid, anomalous, or missing data)
- Adds a `DATA_QUALITY_FLAG` column
- Calculates basic metrics: `DAILY_CHANGE`, `DAILY_CHANGE_PCT`, `INTRADAY_RANGE`, `INTRADAY_RANGE_PCT`
- Materialized as a VIEW for efficient downstream consumption

Only records with `DATA_QUALITY_FLAG = 'VALID'` proceed to subsequent layers.

Step 7: dbt Run - Intermediate Layer(Airflow Task 5)

Next, five intermediate models execute in parallel, each calculating specific technical indicators:

- `int_moving_averages`: Computes MA_5, MA_20, MA_50, EMA_12, EMA_26 using window functions, and generates golden cross/death cross signals
- `int_rsi`: Calculates 14-day Relative Strength Index with overbought (>70) and oversold (<30) signals
- `int_macd`: Computes MACD line (12-day EMA - 26-day EMA), signal line (9-day EMA of MACD), histogram, and crossover signals
- `int_bollinger`: Calculates Bollinger Bands with 20-day MA, upper/lower bands (±2 standard deviations), band width, and squeeze detection
- `int_daily_returns`: Computes daily returns, log returns, 20-day volatility, and cumulative returns

All intermediate models are materialized as TABLES for performance optimization.

Step 8: dbt Run - Marts Layer(Airflow Task 5)

Finally, the `mart_stock_daily_metrics` model consolidates all data:

- JOINs staging and all intermediate models on SYMBOL and DATE
- Combines price data with all technical indicators in a single table
- Adds derived signals: MA cross detection, price trend classification, volatility categorization
- Materialized as a TABLE optimized for BI tool consumption

This denormalized structure enables Preset to query all metrics efficiently without complex joins.

Step 9: dbt Test (Airflow Task 6)

The `dbt_test` task executes data quality tests defined in dbt:

- Uniqueness tests on SYMBOL + DATE combinations
- Not-null tests on critical fields (CLOSE, OPEN, DATE)
- Range validation (e.g., RSI between 0-100, prices > 0)
- Referential integrity checks

If any test fails, the task fails and alerts are triggered, preventing bad data from reaching dashboards.

Step 10: Visualization (Preset)

Preset connects to Snowflake with read-only access and queries `mart_stock_daily_metrics`. Users interact with dashboards featuring:

- Time-series charts for price and moving averages
- Technical indicator visualizations (RSI, MACD, Bollinger Bands)
- Volume analysis with trend lines
- Trading signals summary tables
- Interactive filters for date range and stock symbols

All queries execute directly against Snowflake without data movement, ensuring real-time access to the latest transformed data.

C. Technology Stack

Each component in our technology stack was selected based on specific requirements and industry best practices for modern data engineering pipelines.

Apache Airflow for Orchestration

Airflow was retained from Lab 1 due to its strengths in workflow orchestration:

- **Task dependency management**: The » operator ensures dbt runs only after successful ETL completion
- **Monitoring and observability**: Web UI provides real-time visibility into task execution status and logs
- **Containerization**: Running in Docker containers via docker-compose ensures environment consistency and simplified deployment. The configuration in `docker-compose.yaml` includes Airflow webserver, scheduler, postgres database for metadata, and all required Python packages
- **Retry mechanisms**: Automatic retry logic handles transient failures gracefully

Snowflake as Data Warehouse

Snowflake serves as the cloud data warehouse with separation of storage and compute:

- **Storage**: All data resides in `ASH_DB.STOCK` schema
- **Compute**: The `BEETLE_QUERY_WH` warehouse (smallest size) handles dbt transformations and SQL execution efficiently for our dataset scale

- **Multi-user support**: Multiple users can query data concurrently without performance degradation due to Snowflake's automatic concurrency handling
- **Zero-copy cloning**: Enables efficient testing and development workflows
- **Window functions**: Native support for `OVER`, `PARTITION BY`, `ROWS BETWEEN` makes technical indicator calculations straightforward and performant

The architecture eliminates the need to move data between systems—all transformations execute in-place within Snowflake.

dbt for Transformations

dbt (data build tool) was introduced in Lab 2 for SQL-based transformations:

- **SQL-first approach**: All business logic is expressed in SQL files that are version-controlled in GitHub
- **Modularity**: Adding new indicators requires creating a new `.sql` file in the intermediate folder and referencing it in the marts model, without modifying Python code or redeploying Airflow
- **Separation of concerns**: ETL (Python/Airflow) handles data ingestion; ELT (dbt) handles transformations
- **Built-in testing**: Schema tests and data quality tests are defined alongside models
- **Dependency management**: The `ref()` function automatically resolves model dependencies and execution order
- **Documentation**: Models can be documented inline, generating browsable data lineage and field descriptions
- **Incremental models**: Supports incremental processing for efficiency (though we use full-refresh for 180-day windows)

yfinance for Data Ingestion

The yfinance Python library provides free access to Yahoo Finance market data:

- **Ease of use**: Simple API for downloading historical stock prices
- **No authentication**: No API keys or authentication required
- **Pandas integration**: Returns data as DataFrames for easy manipulation
- **Limitations**: Rate limits and occasional outages; error handling is critical

Preset for Visualization

Preset (hosted Apache Superset) serves as the business intelligence layer:

- **Direct Snowflake connection**: Read-only access to `ASH_DB.STOCK` schema
- **No-code interface**: Business users can create charts and dashboards without SQL knowledge
- **Interactive dashboards**: Built-in filters for date ranges and stock symbols
- **Rich visualizations**: Supports line charts, bar charts, KPI cards, tables, and combination charts
- **Real-time queries**: All queries execute directly against Snowflake, ensuring up-to-date data
- **Sharing capabilities**: Dashboards can be shared via links or embedded in other applications

Supporting Technologies

- **Python 3.9+**: Core language for Airflow DAGs and data processing
- **Docker & Docker Compose**: Container orchestration for consistent environments
- **Git/GitHub**: Version control for all code and dbt models

This technology stack follows modern data engineering best practices by separating ingestion, transformation, storage, and presentation layers, enabling independent scaling and maintenance of each component.

III. Data Model

A. Table Structures

1) Raw Layer Tables: **fact_stock_price_daily**

This table stores the raw historical stock prices fetched from yfinance API. It serves as the single source of truth for all downstream transformations.

Column	Data Type	Constraints	Description
SYMBOL	VARCHAR(10)	NOT NULL, PK	Stock ticker symbol
DATE	DATE	NOT NULL, PK	Trading date
OPEN	FLOAT		Opening price for the day
CLOSE	FLOAT		Closing price for the day
MIN	FLOAT		Lowest price during the day
MAX	FLOAT		Highest price during the day
VOLUME	BIGINT		Number of shares traded

TABLE I
FACT_STOCK_PRICE_DAILY TABLE STRUCTURE

Primary Key: (SYMBOL, DATE)

2) Staging Layer Tables: **stg_stock_daily** (dbt View)

A staging view that validates data quality and computes basic daily metrics. Records with invalid data are flagged and filtered out for downstream processing.

Column	Data Type
SYMBOL	VARCHAR
DATE	DATE
OPEN	FLOAT
CLOSE	FLOAT
MIN	FLOAT
MAX	FLOAT
VOLUME	BIGINT
DATA_QUALITY_FLAG	VARCHAR
DAILY_CHANGE	FLOAT
DAILY_CHANGE_PCT	FLOAT
INTRADAY_RANGE	FLOAT
INTRADAY_RANGE_PCT	FLOAT

TABLE II
STG_STOCK_DAILY VIEW STRUCTURE

DATA_QUALITY_FLAG values: 'VALID', 'INVALID', 'ANOMALY', 'MISSING'

3) Intermediate Layer Tables: The intermediate layer computes technical indicators that are later joined in the marts layer.

int_moving_averages (dbt Table)

Computes short-term (5-day), medium-term (20-day), and long-term (50-day) moving averages along with exponential moving averages and crossover signals.

Column	Data Type
SYMBOL	VARCHAR
DATE	DATE
CLOSE	FLOAT
MA_5	FLOAT
MA_20	FLOAT
MA_50	FLOAT
EMA_12	FLOAT
EMA_26	FLOAT
PRICE_VS_MA20_PCT	FLOAT
MA_CROSS_SIGNAL	VARCHAR

TABLE III
INT_MOVING_AVERAGES TABLE STRUCTURE

int_rsi (dbt Table)

Calculates the 14-day Relative Strength Index, a momentum oscillator that measures overbought and oversold conditions.

Column	Data Type
SYMBOL	VARCHAR
DATE	DATE
CLOSE	FLOAT
AVG_GAIN_14	FLOAT
AVG_LOSS_14	FLOAT
RELATIVE_STRENGTH	FLOAT
RSI_14	FLOAT
RSI_SIGNAL	VARCHAR

TABLE IV
INT_RSI TABLE STRUCTURE

int_bollinger (dbt Table)

Calculates Bollinger Bands with 20-day moving average and bands at ±2 standard deviations, useful for volatility analysis.

Column	Data Type
SYMBOL	VARCHAR
DATE	DATE
CLOSE	FLOAT
BB_MIDDLE	FLOAT
BB_UPPER	FLOAT
BB_LOWER	FLOAT
BB_STDDEV	FLOAT
BB_WIDTH_PCT	FLOAT
BB_PERCENT_B	FLOAT
BB_SIGNAL	VARCHAR
BB_SQUEEZE	BOOLEAN

TABLE V
INT_BOLLINGER TABLE STRUCTURE

int_macd (dbt Table)

Calculates the Moving Average Convergence Divergence (MACD), a trend-following momentum indicator showing the relationship between two EMAs.

Column	Data Type
SYMBOL	VARCHAR
DATE	DATE
CLOSE	FLOAT
EMA_12	FLOAT
EMA_26	FLOAT
MACD_LINE	FLOAT
SIGNAL_LINE	FLOAT
MACD_HISTOGRAM	FLOAT
MACD_SIGNAL	VARCHAR

TABLE VI
INT_MACD TABLE STRUCTURE

int_daily_returns (dbt Table)

Computes daily returns, logarithmic returns, rolling volatility, and cumulative returns for portfolio analysis.

Column	Data Type
SYMBOL	VARCHAR
DATE	DATE
CLOSE	FLOAT
PREV_ADJ_CLOSE	FLOAT
ADJ_DAILY_CHANGE	FLOAT
ADJ_DAILY_RETURN_PCT	FLOAT
LOG_RETURN	FLOAT
VOLATILITY_20D	FLOAT
CUMULATIVE_RETURN_PCT	FLOAT

TABLE VII
INT_DAILY_RETURNS TABLE STRUCTURE

4) Mart Layer Tables: **mart_stock_daily_metrics** (dbt Table)

This is the final unified analytics table that joins all intermediate models and adds comprehensive trading signals. This table serves as the primary data source for Preset dashboards.

Column	Data Type
SYMBOL	VARCHAR
DATE	DATE
CLOSE	FLOAT
OPEN	FLOAT
MAX	FLOAT
MIN	FLOAT
VOLUME	BIGINT
DAILY_CHANGE	FLOAT
DAILY_CHANGE_PCT	FLOAT
INTRADAY_RANGE	FLOAT
INTRADAY_RANGE_PCT	FLOAT
MA_5	FLOAT
MA_20	FLOAT
MA_50	FLOAT
EMA_12	FLOAT
EMA_26	FLOAT
MA_CROSS_SIGNAL	VARCHAR
PRICE_TREND	VARCHAR
RSI_14	FLOAT
RSI_SIGNAL	VARCHAR
BB_UPPER	FLOAT

TABLE VIII
MART_STOCK_DAILY_METRICS TABLE STRUCTURE (PART 1)

Column	Data Type
BB_MIDDLE	FLOAT
BB_LOWER	FLOAT
BB_WIDTH_PCT	FLOAT
BB_PERCENT_B	FLOAT
BB_SIGNAL	VARCHAR
BB_SQUEEZE	BOOLEAN
MACD_LINE	FLOAT
SIGNAL_LINE	FLOAT
MACD_HISTOGRAM	FLOAT
MACD_SIGNAL	VARCHAR
VOLATILITY_SIGNAL	VARCHAR

TABLE IX
MART_STOCK_DAILY_METRICS TABLE STRUCTURE (PART 2)

5) Snapshot Tables: **stock_price_snapshot** (dbt Snapshot - SCD Type 2)

Captures historical changes in stock prices for audit and correction tracking using Slowly Changing Dimension Type 2 methodology.

Column	Data Type
snapshot_key	VARCHAR
SYMBOL	VARCHAR
DATE	DATE
OPEN	FLOAT
CLOSE	FLOAT
MIN	FLOAT
MAX	FLOAT
VOLUME	BIGINT
dbt_valid_from	TIMESTAMP
dbt_valid_to	TIMESTAMP
dbt_scd_id	VARCHAR
dbt_updated_at	TIMESTAMP

TABLE X
STOCK_PRICE_SNAPSHOT TABLE STRUCTURE

B. Entity Relationship Diagram

Figure 2 illustrates the data lineage and dependencies across all warehouse layers.

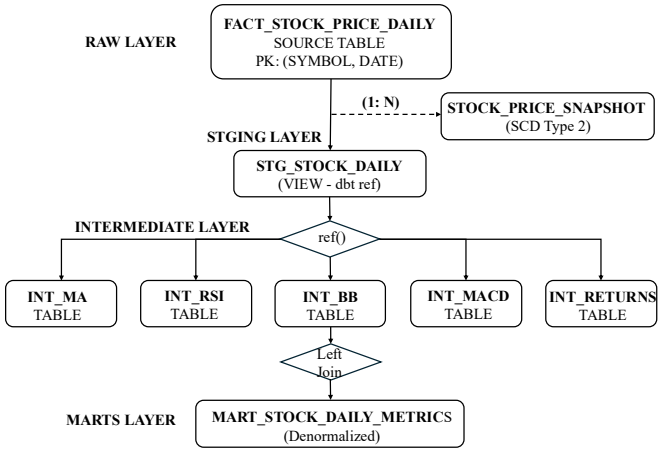


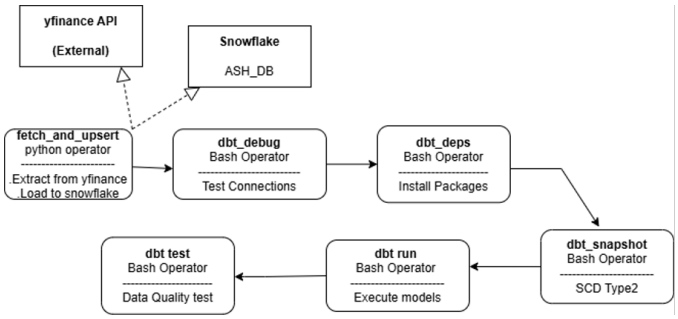
Fig. 2. Entity Relationship Diagram showing data flow from raw layer to marts.

The architecture demonstrates a clear four-layer pattern:

- **Raw Layer**: `FACT_STOCK_PRICE_DAILY` serves as the source table with PK (SYMBOL, DATE), feeding both the transformation pipeline (1:1) and snapshot tracking (1:N)
- **Staging Layer**: `STG_STOCK_DAILY` view validates data quality and computes basic metrics, referencing the raw table via dbt `ref()`
- **Intermediate Layer**: Five parallel models (`INT_MA`, `INT_RSI`, `INT_BB`, `INT_MACD`, `INT_RETURNS`) independently calculate technical indicators, all referencing the staging view
- **Marts Layer**: `MART_STOCK_DAILY_METRICS` consolidates all intermediate models through LEFT JOIN on (SYMBOL, DATE), creating a denormalized table optimized for BI consumption

The `ref()` function establishes automatic dependency resolution, ensuring correct execution order. The snapshot table captures SCD Type 2 changes for historical tracking.

IV. PIPELINE ORCHESTRATION (AIRFLOW)



Execution chain

```
fetch_and_upsert >> dbt_debug >> dbt_deps
                  >> dbt_snapshot >> dbt_run >>
    dbt_test
```

Task summary

Task	Operator	Purpose
`fetch_and_upsert`	PythonOperator	Extract daily prices via `yfinance` and upsert into Snowflake
`dbt_debug`	BashOperator	Validate dbt configuration and Snowflake connection
`dbt_deps`	BashOperator	Install dbt package dependencies (e.g., `dbt_utils`)
`dbt_snapshot`	BashOperator	Capture historical changes using SCD Type 2
`dbt_run`	BashOperator	Execute all dbt models (staging → intermediate → marts)
`dbt_test`	BashOperator	Run data quality tests defined in `schema.yml`

Guarantees: strict sequencing; ELT runs only on fresh data; fail-fast on error.

V. ETL IMPLEMENTATION (AIRFLOW)

This section details the implementation of the ETL component using Apache Airflow, covering DAG configuration, task implementation, connection management, and idempotency mechanisms. The ETL pipeline extracts stock market data

from yfinance, applies necessary transformations, and loads it into Snowflake's raw layer.

A. DAG Configuration

The ETL pipeline is defined as an Airflow DAG with specific configuration parameters that control scheduling, retry behavior, and execution context.

Import Statements

```
from __future__ import annotations
from datetime import datetime, timedelta
import json
import pandas as pd
import yfinance as yf

from airflow import DAG
from airflow.models import Variable
from airflow.operators.python import PythonOperator
from airflow.operators.bash import BashOperator
from airflow.providers.snowflake.hooks.snowflake
    import SnowflakeHook
```

Listing 1. Required imports for ETL DAG

Configuration Constants

```
# Snowflake Configuration
SNOWFLAKE_CONN_ID = 'snowflake_default'
DATABASE = 'ASH_DB'
SCHEMA_ETL = 'STOCK'
WAREHOUSE = 'BEETLE_QUERY_WH'

# dbt Configuration
DBT_PROJECT_DIR = "/opt/airflow/dbt"
```

Listing 2. Pipeline configuration constants

These constants centralize configuration values, enabling easy modification without searching through code. The Snowflake connection ID references the connection configured in Airflow's Admin interface.

DAG Definition

```
default_args = {
    "owner": "data226_lab2",
    "retries": 1,
    "retry_delay": timedelta(minutes=5),
}

with DAG(
    dag_id="stock_etl_dbt",
    start_date=datetime(2025, 1, 1),
    schedule="@daily",
    catchup=False,
    default_args=default_args,
    tags=["lab2", "yfinance", "etl", "elt", "dbt"],
) as dag:
    # Task definitions follow...
```

Listing 3. DAG configuration and instantiation

Parameter	Description
dag_id	Unique identifier for the DAG in Airflow UI
start_date	Beginning date for DAG scheduling (Jan 1, 2025)
schedule	Cron expression; `@daily` runs at midnight UTC daily
catchup	`False` prevents backfilling historical runs on DAG activation
owner	DAG owner for access control and notifications
retries	Number of retry attempts on task failure (1 retry)
retry_delay	Wait time between retries (5 minutes)
tags	Metadata for filtering and organizing DAGs in UI

TABLE XI
DAG CONFIGURATION PARAMETERS

The `catchup=False` setting is critical for this use case. Since we fetch a 180-day rolling window on each run, backfilling would create redundant data fetches and potential inconsistencies.

B. Task Implementation

Helper Function: Symbol Parsing

```
def get_stock_symbols(raw: str | None) -> list[str]:
    if not raw:
        return ["NVDA", "AAPL"]
    try:
        # Try parsing as JSON array
        vals = json.loads(raw)
        return [str(v).strip().upper() for v in vals
    ]
    except Exception:
        # Fall back to comma-separated parsing
        return [s.strip().upper()
                for s in raw.strip("[]").replace('"'
    ,'').split(",")
                if s.strip()]
```

Listing 4. Stock symbol parsing function

This function provides robust symbol parsing with multiple fallback strategies:

- Primary: Parse JSON array format (`["AAPL", "NVDA", "TSLA", "MSFT]`)
- Secondary: Parse comma-separated strings (`AAPL, NVDA, TSLA, MSFT`)
- Fallback: Return default symbols if Variable not set
- Normalization: Strips whitespace and converts to uppercase

ETL Task: fetch_and_upsert

The core ETL logic is implemented in the `fetch_and_upsert` function, which orchestrates data extraction, transformation, cleaning, and loading.

Step 1: Date Range Calculation

```
def fetch_and_upsert(**context):
    # Get execution date from Airflow context
    ds = context["ds"]
    execution_date = datetime.strptime(ds, "%Y-%m-%d
    ").date()

    # Calculate 180-day rolling window
    end_date = execution_date + timedelta(days=1)
    start_date = execution_date - timedelta(days
    =179)

    print(f"Date range: {start_date} to {end_date}")
```

Listing 5. Determining 180-day data window

The `context["ds"]` provides the logical execution date in `YYYY-MM-DD` format. Adding one day to the end date ensures the execution date itself is included in the yfinance query (yfinance uses exclusive end dates).

Step 2: Symbol Retrieval

```
# Get symbols from Airflow Variable
symbols = get_stock_symbols(
    Variable.get("yfinance_symbols", default_var
=None)
)
print(f"Stock symbols: {symbols}")
```

Listing 6. Retrieving configured stock symbols

This approach decouples symbol configuration from code, allowing administrators to modify the symbol list through Airflow's UI without code changes or DAG redeployment.

Step 3: Data Extraction

```
frames = []
for symbol in symbols:
    try:
        print(f"Fetching data for {symbol}...")
        df = yf.download(
            symbol,
            start=start_date.isoformat(),
            end=end_date.isoformat(),
            progress=False
        )

        if df.empty:
            print(f"No data for symbol: {symbol}
")
            continue

        # Handle multi-index columns from
yfinance
        if isinstance(df.columns, pd.MultiIndex)
:
            df.columns = df.columns.
get_level_values(0)

        # Reset index to convert Date to column
        df = df.reset_index()

        # Standardize column names for Snowflake
        df = df.rename(columns={
            "Date": "DATE",
            "Open": "OPEN",
            "Close": "CLOSE",
            "High": "MAX",
            "Low": "MIN",
            "Volume": "VOLUME",
        })

        # Add symbol identifier
        df["SYMBOL"] = symbol

        # Reorder columns to match target schema
        df = df[["SYMBOL", "DATE", "OPEN", "
CLOSE",
                 "MIN", "MAX", "VOLUME"]]

        frames.append(df)

    except Exception as e:
        print(f"Error fetching data for {symbol
}: {e}")
        continue
```

Listing 7. Extracting data from yfinance API

Key implementation details:

- **Per-symbol error handling**: Exceptions for individual symbols don't break the entire pipeline
- **Multi-index handling**: yfinance returns multi-level columns for multiple symbols; we flatten them
- **Column standardization**: Rename columns to match Snowflake schema conventions
- **Symbol attribution**: Add SYMBOL column to identify data source
- **Column ordering**: Ensure consistent order matching target table

Step 4: Data Validation and Cleaning

```
if not frames:
    print("No data fetched for any symbol.")
    return 0

# Concatenate all symbol DataFrames
final_df = pd.concat(frames, ignore_index=True)
final_df["DATE"] = pd.to_datetime(final_df["DATE
"]).dt.date

# Record pre-cleaning row count
original_rows = len(final_df)

# Remove rows with null prices (critical fields)
final_df = final_df.dropna(subset=['OPEN', '
CLOSE'], how='any')

# Fill null volumes with 0 (trading volume can
be missing)
final_df['VOLUME'] = final_df['VOLUME'].fillna
(0)

cleaned_rows = len(final_df)

if original_rows > cleaned_rows:
    print(f"Removed {original_rows -
cleaned_rows} rows with NaN values")

if final_df.empty:
    print("No valid data after cleaning NaN
values.")
    return 0
```

Listing 8. Data quality checks and cleaning

Data quality strategy:

- **Null price handling**: Drop rows missing OPEN or CLOSE (essential for analysis)
- **Null volume handling**: Fill with 0 (volume data sometimes unavailable)
- **Date normalization**: Convert to Python date objects for Snowflake compatibility
- **Empty DataFrame check**: Return early if no valid data remains

Step 5: Database Loading

The loading process is covered in detail in Section 5.4 (Idempotency Implementation).

Task Registration

```
etl_task = PythonOperator(
    task_id="fetch_and_upsert",
    python_callable=fetch_and_upsert,
    provide_context=True,
)
```

Listing 9. Registering ETL task in DAG

The `provide_context=True` parameter enables the function to access Airflow's execution context, including the execution date (`ds`), DAG run information, and other metadata.

C. Connections and Variables

Airflow provides two mechanisms for externalizing configuration: Connections (for credentials) and Variables (for parameters).

Snowflake Connection Configuration

The Snowflake connection is configured through Airflow's web interface: **Admin** → **Connections** → **Create**.

Parameter	Value
Connection Id	snowflake_default
Connection Type	Snowflake
Host	SFEDU02-LVB17920.
snowflakecomputing.com	Snowflake account URL
Schema	STOCK
Login	MALLARD
Password	**********
Account	SFEDU02-LVB17920
Warehouse	BEETLE_QUERY_WH
Database	ASH_DB

TABLE XII
SNOWFLAKE CONNECTION PARAMETERS IN AIRFLOW

Airflow Variables

Variables provide a key-value store for configuration parameters accessible across DAGs.

Configuration: **Admin** → **Variables** → **Create**

Variable Key	Value
yfinance_symbols	["AAPL","NVDA", "TSLA", "MSFT"]

TABLE XIII
AIRFLOW VARIABLES FOR PIPELINE CONFIGURATION

D. Idempotency Implementation

Idempotency ensures that executing the same ETL operation multiple times produces identical results, preventing duplicate records and data inconsistencies. This is critical for reliable data pipelines that may need to reprocess data due to failures or corrections.

Database Connection Setup

```
hook = SnowflakeHook(snowflake_conn_id=
    SNOWFLAKE_CONN_ID)
conn = hook.get_conn()
cs = conn.cursor()

try:
    cs.execute(f"USE WAREHOUSE {WAREHOUSE}")
    cs.execute(f"USE DATABASE {DATABASE}")
    cs.execute(f"USE SCHEMA {SCHEMA_ETL}")

    # Verify connection and context
    cs.execute("""
        SELECT CURRENT_VERSION(), CURRENT_WAREHOUSE
    (),
               CURRENT_ROLE(), CURRENT_DATABASE(),
               CURRENT_SCHEMA()
    """)
    ver, wh, role, db, sch = cs.fetchone()
    print(f"Connected: {db}.{sch} on {wh} (version {
    ver}, role {role})")
```

Listing 10. Establishing Snowflake connection

The `SnowflakeHook` abstracts connection management, automatically handling authentication using credentials from the Airflow Connection.

Table Creation with Constraints

```
# Create fact table if it doesn't exist
cs.execute("""
    CREATE TABLE IF NOT EXISTS
fact_stock_price_daily (
        SYMBOL VARCHAR(10) NOT NULL,
        "DATE" DATE NOT NULL,
        "OPEN" FLOAT,
        "CLOSE" FLOAT,
        "MIN" FLOAT,
        "MAX" FLOAT,
        VOLUME BIGINT,
        PRIMARY KEY (SYMBOL, "DATE")
    )
""")
```

Listing 11. Creating target table with primary key

The composite primary key (`SYMBOL, DATE`) enforces uniqueness at the database level, preventing duplicate records even if application logic fails. The `CREATE TABLE IF NOT EXISTS` pattern makes the script idempotent—safe to run multiple times.

Transaction-Based Loading

```
# Begin transaction
cs.execute("BEGIN")

# Create temporary staging table
cs.execute("""
    CREATE TEMPORARY TABLE
tmp_fact_stock_price_daily AS
    SELECT * FROM fact_stock_price_daily WHERE
1=0
""")

# Prepare rows for insertion
rows = [
    {
        "SYMBOL": row[0],
        "DATE": row[1],
        "OPEN": float(row[2]),
        "CLOSE": float(row[3]),
        "MIN": float(row[4]),
        "MAX": float(row[5]),
        "VOLUME": int(row[6]) if pd.notna(row
[6]) else 0,
    }
    for row in final_df.itertuples(index=False)
]

# Bulk insert into temporary table
if rows:
    cs.executemany("""
        INSERT INTO tmp_fact_stock_price_daily
        (SYMBOL, "DATE", "OPEN", "CLOSE", "MIN",
 "MAX", VOLUME)
        VALUES (%(SYMBOL)s, %(DATE)s, %(OPEN)s,
%(CLOSE)s,
                %(MIN)s, %(MAX)s, %(VOLUME)s)
    """, rows)

# Execute MERGE operation (upsert logic)
cs.execute("""
    MERGE INTO fact_stock_price_daily t
    USING tmp_fact_stock_price_daily s
      ON t.SYMBOL = s.SYMBOL AND t."DATE" = s."
DATE"
    WHEN MATCHED THEN UPDATE SET
```

```
            "OPEN" = s."OPEN",
            "CLOSE" = s."CLOSE",
            "MIN" = s."MIN",
            "MAX" = s."MAX",
            VOLUME = s.VOLUME
        WHEN NOT MATCHED THEN INSERT
            (SYMBOL, "DATE", "OPEN", "CLOSE", "MIN", "
    MAX", VOLUME)
        VALUES
            (s.SYMBOL, s."DATE", s."OPEN", s."CLOSE",
             s."MIN", s."MAX", s.VOLUME)
    """)

    # Commit transaction
    cs.execute("COMMIT")
    print(f"Upserted {len(rows)} rows into
    fact_stock_price_daily.")
    return len(rows)

except Exception as e:
    # Rollback on any error
    cs.execute("ROLLBACK")
    print(f"Error during database operation: {e}")
    raise
finally:
    # Always close connections
    cs.close()
    conn.close()
```

Listing 12. Atomic transaction with MERGE operation

Idempotency Mechanisms

Mechanism	Idempotency Guarantee
BEGIN/COMMIT/ROLLBACK	All operations succeed together or fail together (atomicity); partial updates impossible
MERGE Statement	Updates existing records (MATCHED) or inserts new ones (NOT MATCHED); same data loaded twice produces identical result
Primary Key Constraint	Database enforces uniqueness on (SYMBOL, DATE); prevents duplicate records even if MERGE logic fails
Temporary Table	Isolates staging data from production table; MERGE operates on validated dataset
Try/Catch Block	Any error triggers ROLLBACK; ensures database returns to pre-transaction state

TABLE XIV
IDEMPOTENCY IMPLEMENTATION MECHANISMS

E. Screenshots

DAG Graph View

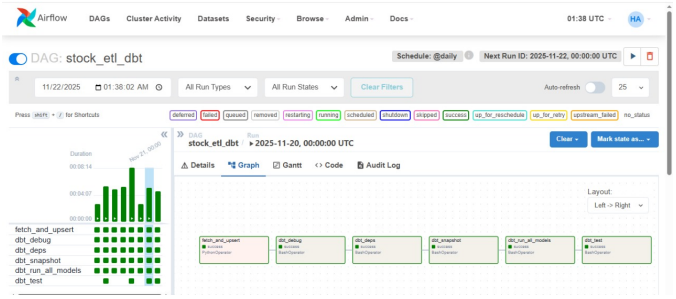


Fig. 3. Airflow DAG graph view showing the linear dependency chain: `fetch_and_upsert` → `dbt_debug` → `dbt_deps` → `dbt_snapshot` → `dbt_run_all_models` → `dbt_test`. Green indicates success (UTC).

DAG Runs History

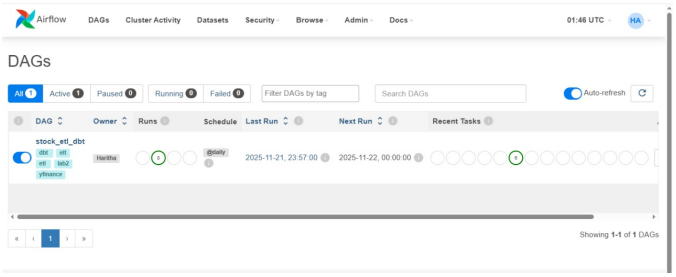


Fig. 4. DAG runs overview showing the @daily schedule and recent successful executions (UTC).

Task Logs

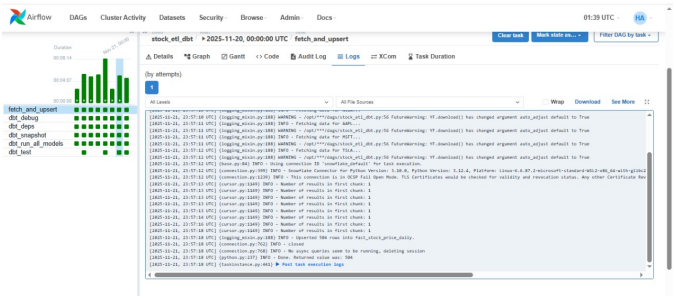


Fig. 5. Task logs for `fetch_and_upsert`, including date range calculation, symbols fetched, data cleaning, Snowflake connection checks, and MERGE upsert into `fact_stock_price_daily`.

VI. ELT Implementation (dbt)

This section details the implementation of the ELT (Extract-Load-Transform) component using dbt (data build tool). Unlike traditional ETL where transformations occur before loading, ELT leverages Snowflake's compute power to transform data in-place using SQL. The dbt project implements a three-tier architecture (staging, intermediate, marts) with built-in testing, documentation, and version control.

A. dbt Project Structure

The dbt project follows a modular, layered architecture that separates data cleaning, business logic, and analytics-ready outputs.

Project Organization

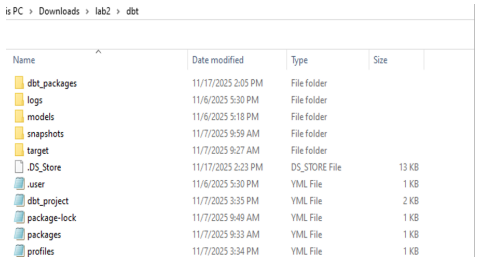


Fig. 6. dbt project directory structure showing modular organization.

Configuration Files

dbt_project.yml: Project-level configuration

Key configuration decisions:

- **Staging as views**: Lightweight, always fresh data without storage costs
- **Intermediate as tables**: Materialized for performance (parallel reads by marts)
- **Marts as tables**: Optimized for BI tool queries, denormalized structure
- **Schema separation**: Logical organization within Snowflake database

profiles.yml: Snowflake connection configuration

```
snowflake_stock:
  target: dev
  outputs:
    dev:
      type: snowflake
      account: SFEDU02-LVB17920
      user: MALLARD
      password: "{{ env_var('SNOWFLAKE_PASSWORD') }}
    "
      role: ACCOUNTADMIN
      database: ASH_DB
      warehouse: BEETLE_QUERY_WH
      schema: STOCK
      threads: 4
      client_session_keep_alive: False
```

Listing 13. profiles.yml Snowflake connection

Configuration highlights:

- **Environment variable**: Password sourced from `SNOWFLAKE_PASSWORD` env var for security
- **Threads**: 4 parallel model executions for performance
- **Schema**: Default schema (STOCK); models can override with `+schema`
- **Warehouse**: Uses BEETLE_QUERY_WH for all transformations

packages.yml: External dependencies

```
packages:
  - package: dbt-labs/dbt_utils
    version: 1.1.1
```

Listing 14. packages.yml dependency specification

The `dbt_utils` package provides reusable macros for common operations like generating surrogate keys, pivot operations, and advanced testing utilities.

B. Models Implementation

dbt models are SQL SELECT statements that define transformations. Each model represents a table or view in Snowflake. Models reference other models using the `ref()` function, which dbt uses to build a dependency graph and determine execution order.

1) Staging Models: Staging models form the foundation layer, connecting to raw source data and applying initial cleaning and validation.

Source Configuration

```
version: 2

sources:
  - name: raw
    database: ASH_DB
    schema: STOCK
    tables:
      - name: fact_stock_price_daily
        description: "Raw stock prices from yfinance
    "
        columns:
          - name: SYMBOL
            description: "Stock ticker symbol"
            tests:
              - not_null
          - name: DATE
            description: "Trading date"
            tests:
              - not_null
```

Listing 15. models/staging/sources.yml

The `source()` function provides:

- **Documentation**: Centralized metadata about source tables
- **Lineage tracking**: dbt tracks dependencies on source tables
- **Freshness checks**: Can monitor data freshness (last updated time)
- **Testing**: Apply schema tests to raw data

Staging Model: stg_stock_daily

```
{{
    config(
        materialized='view'
    )
}}

WITH source_data AS (
    SELECT
        SYMBOL,
        "DATE",
        "OPEN",
        "CLOSE",
        "MIN",
        "MAX",
        VOLUME
    FROM {{ source('raw', 'fact_stock_price_daily')
    }}
),

data_quality_check AS (
    SELECT
```

```
        *,
        CASE
            WHEN "CLOSE" = 0 OR "OPEN" = 0 THEN '
    INVALID'
            WHEN "MAX" < "MIN" THEN 'ANOMALY'
            WHEN "CLOSE" IS NULL OR "OPEN" IS NULL
    THEN 'MISSING'
            ELSE 'VALID'
        END AS DATA_QUALITY_FLAG
    FROM source_data
)

SELECT
    -- Basic fields
    SYMBOL,
    "DATE",
    "OPEN",
    "CLOSE",
    "MIN",
    "MAX",
    VOLUME,
    DATA_QUALITY_FLAG,

    -- Derived metrics
    "CLOSE" - "OPEN" AS DAILY_CHANGE,
    ROUND(("CLOSE" - "OPEN") / NULLIF("OPEN", 0) *
    100, 2)
        AS DAILY_CHANGE_PCT,
    "MAX" - "MIN" AS INTRADAY_RANGE,
    ROUND(("MAX" - "MIN") / NULLIF("MIN", 0) * 100,
    2)
        AS INTRADAY_RANGE_PCT

FROM data_quality_check
WHERE DATA_QUALITY_FLAG = 'VALID'
```

Listing 16. models/staging/stg_stock_daily.sql

Design rationale:

- **CTE pattern**: Uses Common Table Expressions for readability and modularity
- **Data quality flags**: Categorizes records without discarding (visibility)
- **Filter valid only**: Downstream models only see clean data
- **Basic calculations**: Adds simple derived metrics used across models
- **NULLIF protection**: Prevents division by zero errors
- **View materialization**: No storage cost, always reflects source changes

2) Intermediate Models: Intermediate models contain focused business logic, with each model calculating a specific set of related metrics. This modular approach improves maintainability and enables parallel execution.

int_moving_averages: Moving Average Calculations

```
{{
    config(
        materialized='table'
    )
}}

SELECT
    SYMBOL,
    "DATE",
    "CLOSE",

    -- Short-term: 5-day MA
    ROUND(
        AVG("CLOSE") OVER (
            PARTITION BY SYMBOL
            ORDER BY "DATE"
            ROWS BETWEEN 4 PRECEDING AND CURRENT ROW
        ), 4
    ) AS MA_5,

    -- Medium-term: 20-day MA
    ROUND(
        AVG("CLOSE") OVER (
            PARTITION BY SYMBOL
            ORDER BY "DATE"
            ROWS BETWEEN 19 PRECEDING AND CURRENT
    ROW
        ), 4
    ) AS MA_20,

    -- Long-term: 50-day MA
    ROUND(
        AVG("CLOSE") OVER (
            PARTITION BY SYMBOL
            ORDER BY "DATE"
            ROWS BETWEEN 49 PRECEDING AND CURRENT
    ROW
        ), 4
    ) AS MA_50,

    -- Golden Cross / Death Cross detection
    CASE
        WHEN AVG("CLOSE") OVER (...MA_5...) >
             AVG("CLOSE") OVER (...MA_50...)
        THEN 'GOLDEN_CROSS'
        WHEN AVG("CLOSE") OVER (...MA_5...) <
             AVG("CLOSE") OVER (...MA_50...)
        THEN 'DEATH_CROSS'
        ELSE 'NEUTRAL'
    END AS MA_CROSS_SIGNAL

FROM {{ ref('stg_stock_daily') }}
WHERE DATA_QUALITY_FLAG = 'VALID'
```

Listing 17. models/intermediate/int_moving_averages.sql

Technical implementation notes:

- **Window functions**: Snowflake's windowing efficiently calculates rolling averages
- **PARTITION BY SYMBOL**: Ensures calculations stay within each stock
- **ORDER BY DATE**: Maintains chronological order for time-series
- **ROWS BETWEEN**: Defines rolling window size (4 preceding = 5 days total)
- **ref() function**: References staging model, establishing dependency

int_rsi: Relative Strength Index

```
WITH price_changes AS (
    SELECT
        SYMBOL,
        "DATE",
        "CLOSE",
        "CLOSE" - LAG("CLOSE") OVER (
            PARTITION BY SYMBOL ORDER BY "DATE"
        ) AS PRICE_CHANGE
    FROM {{ ref('stg_stock_daily') }}
),

gains_losses AS (
    SELECT
        SYMBOL,
        "DATE",
```

```
        "CLOSE",
        CASE WHEN PRICE_CHANGE > 0 THEN PRICE_CHANGE
     ELSE 0
            END AS GAIN,
        CASE WHEN PRICE_CHANGE < 0 THEN ABS(
    PRICE_CHANGE) ELSE 0
            END AS LOSS
    FROM price_changes
),

avg_gains_losses AS (
    SELECT
        SYMBOL,
        "DATE",
        "CLOSE",
        AVG(GAIN) OVER (
            PARTITION BY SYMBOL
            ORDER BY "DATE"
            ROWS BETWEEN 13 PRECEDING AND CURRENT
    ROW
        ) AS AVG_GAIN_14,
        AVG(LOSS) OVER (
            PARTITION BY SYMBOL
            ORDER BY "DATE"
            ROWS BETWEEN 13 PRECEDING AND CURRENT
    ROW
        ) AS AVG_LOSS_14
    FROM gains_losses
)

SELECT
    SYMBOL,
    "DATE",
    "CLOSE",
    ROUND(100 - (100 / (1 + (AVG_GAIN_14 / NULLIF(
    AVG_LOSS_14, 0)))), 4)
        AS RSI_14,
    CASE
        WHEN RSI_14 > 70 THEN 'OVERBOUGHT'
        WHEN RSI_14 < 30 THEN 'OVERSOLD'
        ELSE 'NEUTRAL'
    END AS RSI_SIGNAL
FROM avg_gains_losses
```

Listing 18. models/intermediate/int_rsi.sql

RSI calculation methodology:

1) Calculate price changes day-over-day using `LAG()`
2) Separate gains (positive changes) from losses (negative changes)
3) Compute 14-day average gains and losses using rolling windows
4) Apply RSI formula: $RSI = 100 - \frac{100}{1+RS}$ where $RS = \frac{AvgGain}{AvgLoss}$
5) Classify as overbought (> 70), oversold (< 30), or neutral

Similar patterns in other intermediate models:

- **int_macd**: MACD = EMA(12) - EMA(26), signal line = EMA(9) of MACD
- **int_bollinger**: Middle band = 20-day MA, bands = middle ± 2 std deviations
- **int_daily_returns**: Daily return %, log returns, 20-day volatility

All intermediate models:

- Materialized as **tables** for performance (marts join multiple intermediates)
- Execute in **parallel** (no dependencies on each other, only on staging)
- Follow **single responsibility**: Each calculates one family of indicators
- Include **business context**: Not just numbers, but interpreted signals

3) Marts Models: The marts layer consolidates all intermediate calculations into denormalized, analytics-ready tables optimized for BI tool consumption.

mart_stock_daily_metrics: Unified Analytics Table

```
{{
    config(
        materialized='table'
    )
}}

WITH base AS (
    SELECT * FROM {{ ref('stg_stock_daily') }}
    WHERE DATA_QUALITY_FLAG = 'VALID'
),
ma AS (SELECT * FROM {{ ref('int_moving_averages')
    }}),
rsi AS (SELECT * FROM {{ ref('int_rsi') }}),
bollinger AS (SELECT * FROM {{ ref('int_bollinger')
    }}),
macd AS (SELECT * FROM {{ ref('int_macd') }})

SELECT
    -- Primary keys and basic data
    base.SYMBOL,
    base."DATE",
    base."CLOSE",
    base."OPEN",
    base."MAX",
    base."MIN",
    base.VOLUME,

    -- Staging metrics
    base.DAILY_CHANGE,
    base.DAILY_CHANGE_PCT,

    -- Moving averages
    ma.MA_5,
    ma.MA_20,
    ma.MA_50,
    ma.EMA_12,
    ma.EMA_26,

    -- RSI indicators
    rsi.RSI_14,
    rsi.RSI_SIGNAL,

    -- Bollinger Bands
    bollinger.BB_UPPER,
    bollinger.BB_MIDDLE,
    bollinger.BB_LOWER,
    bollinger.BB_SIGNAL,

    -- MACD indicators
    macd.MACD_LINE,
    macd.SIGNAL_LINE,
    macd.MACD_HISTOGRAM,
    macd.MACD_SIGNAL,

    -- Derived trading signals
    CASE
        WHEN base."CLOSE" > ma.MA_5
             AND base."CLOSE" > ma.MA_20
             AND base."CLOSE" > ma.MA_50
        THEN 'STRONG_UPTREND'
```

```
        WHEN base."CLOSE" < ma.MA_5
             AND base."CLOSE" < ma.MA_20
             AND base."CLOSE" < ma.MA_50
        THEN 'STRONG_DOWNTREND'
        ELSE 'SIDEWAYS'
    END AS PRICE_TREND

FROM base
LEFT JOIN ma ON base.SYMBOL = ma.SYMBOL AND base."
    DATE" = ma."DATE"
LEFT JOIN rsi ON base.SYMBOL = rsi.SYMBOL AND base."
    DATE" = rsi."DATE"
LEFT JOIN bollinger ON base.SYMBOL = bollinger.
    SYMBOL
                    AND base."DATE" = bollinger."
    DATE"
LEFT JOIN macd ON base.SYMBOL = macd.SYMBOL AND base
    ."DATE" = macd."DATE"
```

Listing 19. models/marts/mart_stock_daily_metrics.sql

Design principles:

- **LEFT JOINs**: Preserve all dates even if some indicators unavailable (early dates)
- **Denormalization**: All metrics in single table eliminates BI-side joins
- **Composite signals**: Combines multiple indicators (e.g., PRICE_TREND from MAs)
- **Single source of truth**: Preset queries only this table, not intermediates
- **Performance optimized**: Materialized as table, indexed on (SYMBOL, DATE)

C. Tests and Data Quality

dbt provides two test types: schema tests (generic, reusable) and data tests (custom SQL assertions).

```
version: 2

models:
  - name: stg_stock_daily
    description: "Staging table with cleaned stock
    data"
    columns:
      - name: SYMBOL
        description: "Stock ticker symbol"
        tests:
          - not_null
          - accepted_values:
              values: ['AAPL', 'NVDA']

      - name: DATE
        description: "Trading date"
        tests:
          - not_null
          - unique:
              config:
                where: "SYMBOL = 'AAPL'"

      - name: CLOSE
        description: "Closing price"
        tests:
          - not_null
          - dbt_utils.expression_is_true:
              expression: "> 0"
```

Listing 20. models/staging/schema.yml

Test execution: Running `dbt test` executes all defined tests. Failed tests mark the dbt run as failed, preventing bad data from reaching dashboards.

```
-- RSI must be between 0 and 100
SELECT *
FROM {{ ref('int_rsi') }}
WHERE RSI_14 < 0 OR RSI_14 > 100
```

Listing 21. tests/assert_rsi_range.sql

Custom tests return rows that violate constraints. Zero rows = test passes.

Test categories implemented:

- **Uniqueness**: (SYMBOL, DATE) combinations unique in each layer
- **Null checks**: Critical fields (CLOSE, OPEN, DATE) never null
- **Value ranges**: RSI [0,100], prices > 0, Bollinger % [0,100]
- **Referential integrity**: All mart records have corresponding intermediate records
- **Freshness**: Source data updated within expected timeframe

D. Snapshots

Snapshots implement Slowly Changing Dimension Type 2 (SCD2) logic, capturing historical changes in source data.

```
{% snapshot stock_price_snapshot %}

{{
    config(
      target_schema='snapshots',
      unique_key='SYMBOL || DATE',
      strategy='check',
      check_cols=['OPEN', 'CLOSE', 'MIN', 'MAX', '
    VOLUME'],
    )
}}

SELECT * FROM {{ source('raw', '
    fact_stock_price_daily') }}

{% endsnapshot %}
```

Listing 22. snapshots/stock_price_snapshot.sql

Configuration explanation:

- **target_schema**: Snapshots saved to separate schema for organization
- **unique_key**: Composite key identifying unique records
- **strategy='check'**: Monitors specified columns for changes
- **check_cols**: List of columns to track for changes

Generated columns:

- `dbt_valid_from`: Timestamp when this version became valid
- `dbt_valid_to`: Timestamp when superseded (NULL = current version)
- `dbt_scd_id`: Unique identifier for this snapshot record
- `dbt_updated_at`: Last update timestamp

Use cases:

- Audit trail: Track price corrections or data updates

- Time-travel queries: Analyze data as it appeared on specific date
- Change detection: Identify when/how frequently data changes
- Regulatory compliance: Maintain immutable historical records

E. Airflow Integration

The dbt transformations are orchestrated through Airflow using BashOperator tasks, ensuring models execute after successful ETL completion.

dbt Command Execution

All dbt commands execute from the project directory with the `--profiles-dir` flag:

```
DBT_PROJECT_DIR = "/opt/airflow/dbt"

dbt_run = BashOperator(
    task_id="dbt_run_all_models",
    bash_command=f"cd {DBT_PROJECT_DIR} && dbt run
    --profiles-dir .",
)

dbt_test = BashOperator(
    task_id="dbt_test",
    bash_command=f"cd {DBT_PROJECT_DIR} && dbt test
    --profiles-dir .",
)
```

Listing 23. dbt task definitions in Airflow

The `--profiles-dir .` flag tells dbt to use `profiles.yml` in the project directory instead of the default `~/.dbt/` location, enabling containerized deployments.

Execution Order and Dependencies

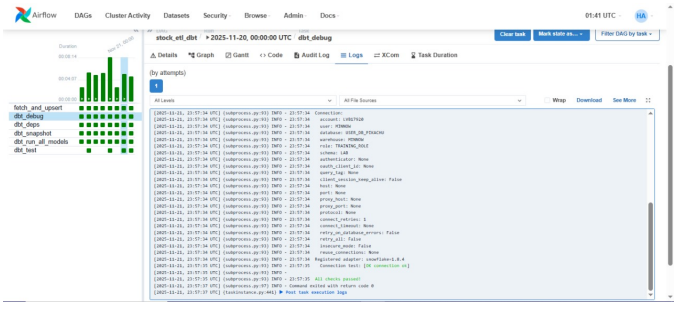
```
etl_task >> dbt_debug >> dbt_deps >> dbt_snapshot >>
    dbt_run >> dbt_test
```

Dependency rationale:

- **dbt_debug after ETL**: Validates configuration after fresh data loaded
- **dbt_deps before run**: Ensures packages installed before models use them
- **dbt_snapshot before run**: Captures raw data state before transformations
- **dbt_test after run**: Validates transformed data quality

F. Screenshots

dbt Debug



dbt Dependencies

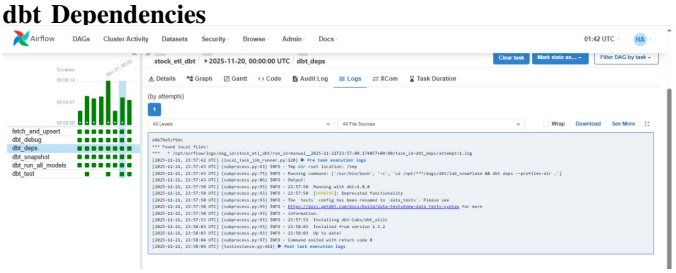


Fig. 8. `dbt_deps` installing packages (e.g., `dbt_utils`); run completed successfully.

dbt Snapshot

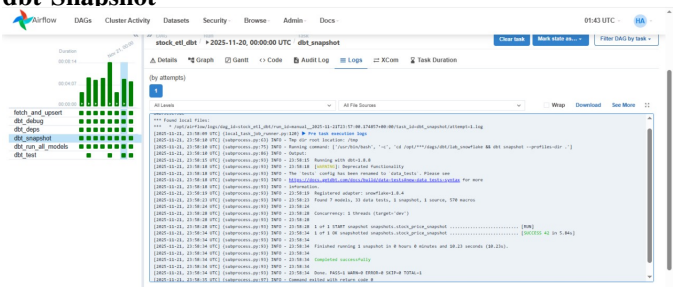


Fig. 9. `dbt_snapshot` execution showing SCD Type 2 snapshot `stock_price_snapshot` completed successfully.

dbt Run

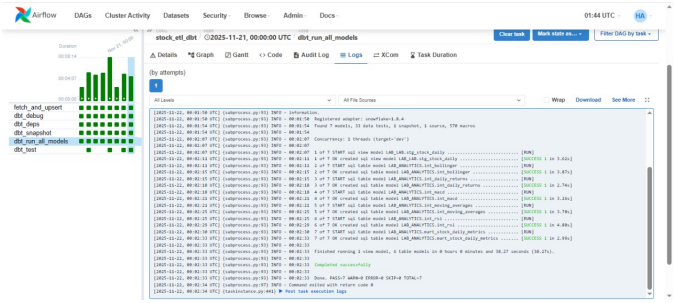


Fig. 10. `dbt run` summary: models built in dependency order (staging → intermediates → marts) with per-model timings and success status.

dbt Tests

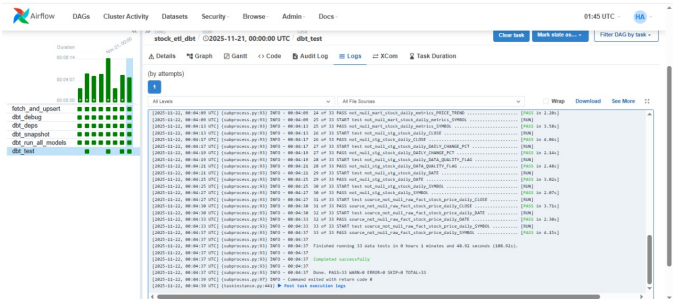


Fig. 11. `dbt test` results: all schema/data tests passed; total tests and durations shown (UTC).

VII. Data Visualization (Preset)

This section presents the business intelligence layer implemented using Preset (hosted Apache Superset), which provides interactive dashboards for visualizing stock analytics without requiring SQL knowledge. The dashboards connect directly to Snowflake's `mart_stock_daily_metrics` table, enabling real-time analysis of technical indicators and trading signals.

A. Dashboard Overview

The Stock Analysis dashboard consolidates all technical indicators and trading signals into a single interactive interface, designed for traders and analysts to make informed decisions.

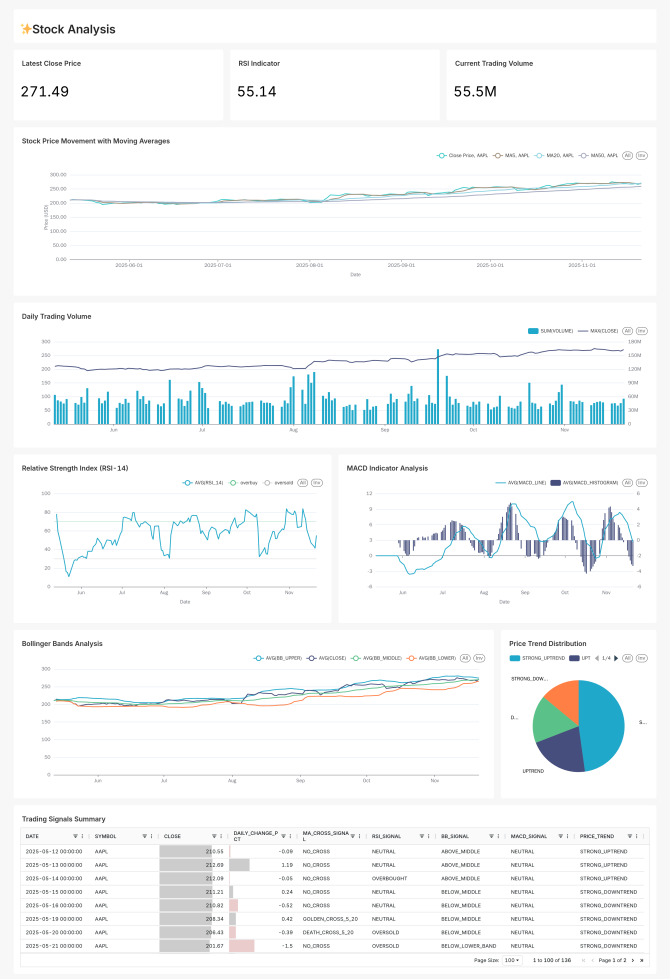


Fig. 12. Complete Stock Analysis dashboard showing comprehensive technical indicators for AAPL.

Dashboard Specifications:

- **Data Source**: Snowflake `ASH_DB.STOCK.mart_stock_daily_metrics`
- **Refresh Strategy**: Real-time queries (no caching for latest data)
- **Date Range**: Configurable via dashboard filters
- **Stock Selection**: Filter by symbol (AAPL, NVDA, TSLA, MSFT)
- **Components**: 8 visualizations + 1 data table

The dashboard is organized into three functional areas:

1) **Key Metrics (Top)**: Three KPI cards showing instant market snapshot
2) **Trend Analysis (Middle)**: Four time-series charts for technical indicators
3) **Signal Summary (Bottom)**: Trading signals table with comprehensive metrics

B. Chart Descriptions

1) KPI Cards

Three metric cards at the top provide immediate visibility into current market conditions.

Latest Close Price

Metric	Value (AAPL)
Latest Close Price	$271.49
Date	2025-05-21

Displays the most recent closing price, serving as the primary reference point for all other indicators.

RSI Indicator

Metric	Value (AAPL)
Current RSI (14-day)	55.14
Signal	NEUTRAL
Interpretation	Neither overbought nor oversold

RSI of 55.14 indicates neutral momentum:

- $>$ **70**: Overbought (potential pullback)
- $<$ **30**: Oversold (potential bounce)
- **30-70**: Neutral range (current state)

Current Trading Volume

Metric	Value (AAPL)
Latest Volume	55.5M shares

Trading volume indicates market interest and liquidity. Higher volume during price moves confirms trend strength.

2) Stock Price Movement with Moving Averages

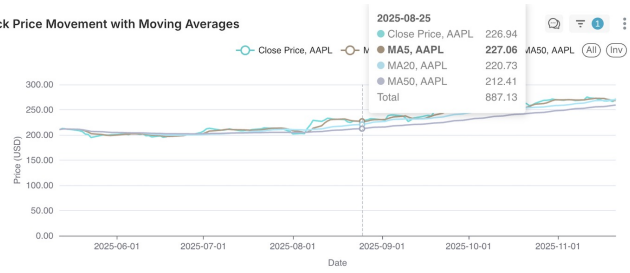


Fig. 13. Stock price with 5-day, 20-day, 50-day moving averages.

Chart Configuration:

- **Chart Type**: Multi-line time series
- **X-axis**: Date (June 2025 - November 2025)
- **Y-axis (Left)**: Price in USD ($0 - $300)
- **Lines Displayed**:
  - Close Price (teal, primary trend line)
  - MA 5 (light blue, short-term trend)
  - MA 20 (green, medium-term trend)
  - MA 50 (orange, long-term trend)

3) Daily Trading Volume

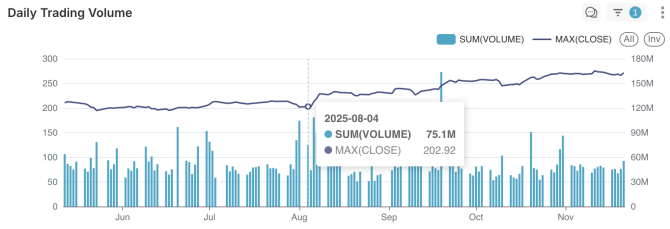


Fig. 14. Daily trading volume (bars).

Chart Configuration:

- **Chart Type**: Combination (bar + line)
- **Primary (Bars)**: Daily volume in millions
- **Secondary (Line)**: Close Price
- **Y-axis (Left)**: Price in USD ($0 - $300)
- **Y-axis (Right)**: Volume (0-180M shares)

4) Relative Strength Index (RSI-14)

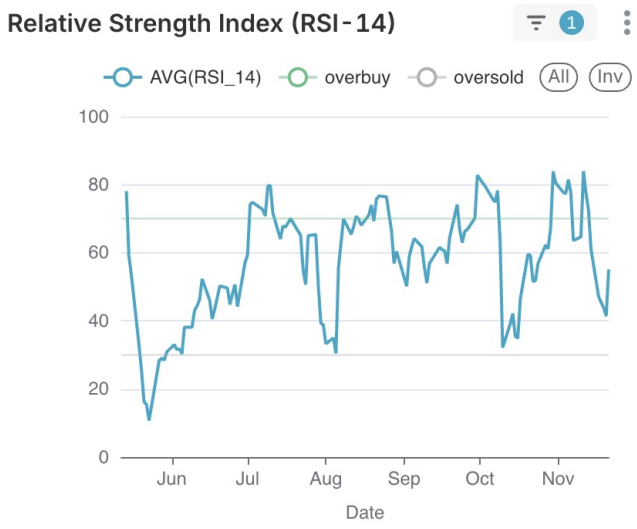


Fig. 15. 14-day RSI with overbought (70) and oversold (30) threshold lines.

Chart Configuration:

- **Chart Type**: Line chart with reference lines
- **Y-axis**: RSI value (0-100)
- **Reference Lines**:
  - Red dashed (70): Overbought threshold
  - Blue dashed (30): Oversold threshold
- **Color Zones**:
  - Above 70: Overbought
  - Below 30: Oversold
  - 30-70: Neutral

5) MACD Indicator Analysis

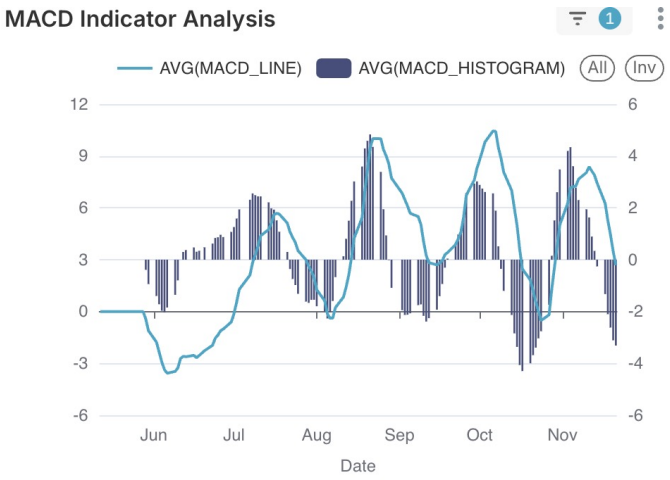


Fig. 16. MACD line, signal line, and histogram showing momentum changes.

Chart Configuration:

- **Chart Type**: Line + bar combination
- **MACD Line** (teal): 12-day EMA - 26-day EMA
- **Signal Line** (gray): 9-day EMA of MACD
- **Histogram** (bars): MACD - Signal (momentum strength)
- **Y-axis**: MACD value (-6 to +12)
- **Zero Line**: Separates bullish (above) from bearish (below)

6) Bollinger Bands Analysis

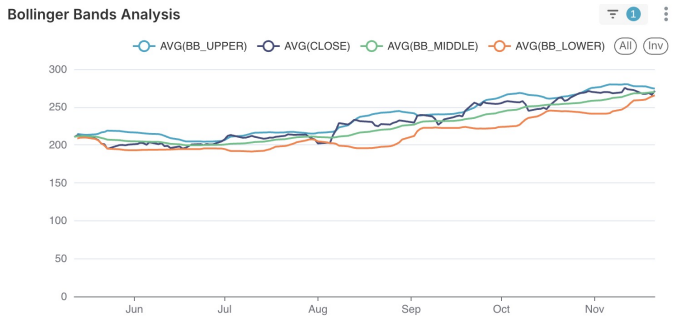


Fig. 17. Bollinger Bands (upper, middle, lower) with closing price showing volatility patterns.

Chart Configuration:

- **Chart Type**: Multi-line area chart
- **Upper Band** (teal): 20-day MA + 2 standard deviations
- **Middle Band** (black): 20-day moving average

- **Lower Band** (orange): 20-day MA - 2 standard deviations
- **Close Price** (green): Actual closing price
- **Shaded Area**: Visual representation of volatility channel

7) Price Trend Distribution

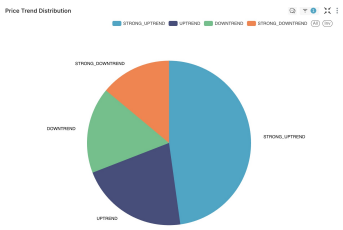


Fig. 18. Pie chart showing distribution of trading days by price trend classification.

Chart Configuration:

- **Chart Type**: Pie chart
- **Metric**: Count of trading days by `PRICE_TREND` classification
- **Categories**:
  - STRONG_UPTREND (teal): Close $>$ MA 5, MA 20, MA 50
  - UPTREND (dark blue): Close $>$ MA 20
  - DOWNTREND (green): Close $<$ MA 20
  - STRONG_DOWNTREND (orange): Close $<$ all MAs

8) Trading Signals Summary Table

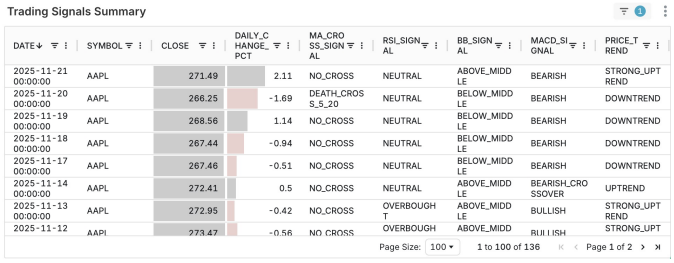


Fig. 19. Comprehensive trading signals table showing all indicators for recent dates.

Table Configuration:

- **Columns**: DATE, SYMBOL, CLOSE, DAILY_CHANGE_PCT, MA_CROSS_SIGNAL, RSI_SIGNAL, BB_SIGNAL, MACD_SIGNAL, PRICE_TREND
- **Sorting**: Descending by date (most recent first)
- **Pagination**: 100 rows per page
- **Row Count**: 1-10 of 138 total records
- **Filtering**: All columns searchable

C. User Guide

1) Dashboard Interactivity

Users can interact with the dashboard through several mechanisms:

Date Range Filter:

Users can filter data by selecting specific time periods to focus analysis on relevant timeframes.

1) Click date filter in dashboard header
2) Select preset ranges (Last 7 days, Last 30 days, Last 90 days)
3) Or specify custom date range
4) Click "Apply" to refresh all charts

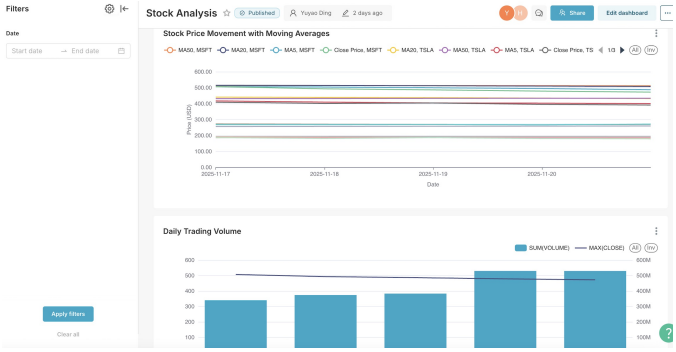


Fig. 20. Full 180-day data view showing all available historical data.

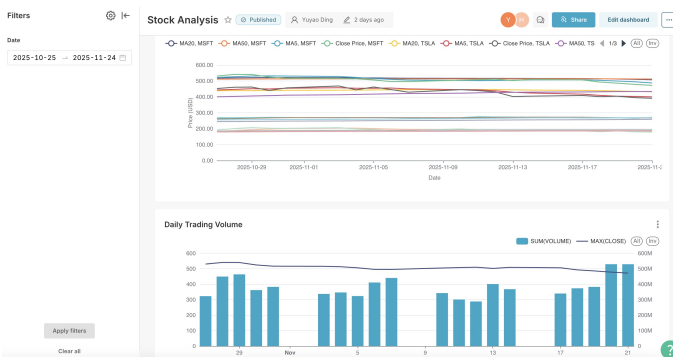


Fig. 21. Filtered view (2025-10-25 to 2025-11-24) showing focused time period with adjusted trends.

Stock Symbol Filter:

Users can select individual stocks to view detailed metrics and technical indicators for specific symbols.

1) Click symbol dropdown in header
2) Select AAPL, NVDA, TSLA, or MSFT
3) Dashboard automatically refreshes with selected stock's data

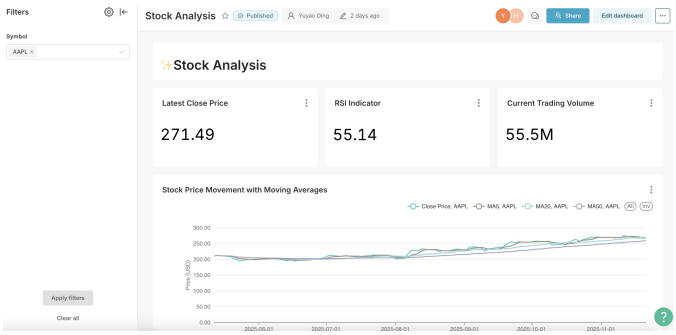


Fig. 22. AAPL dashboard showing close price of $271.49 and RSI of 55.14.

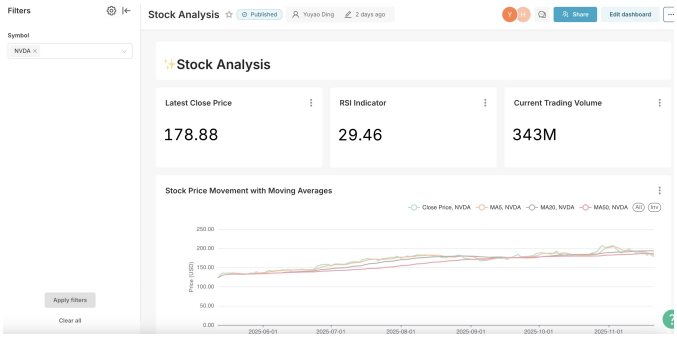


Fig. 23. NVDA dashboard showing close price of $178.88 and RSI of 29.46, demonstrating different market behavior.

Table Interactions:

- **Sort**: Click column headers to sort ascending/descending
- **Search**: Use search box to filter by any column value
- **Pagination**: Navigate through pages with controls at bottom

2) Typical User Workflows

Daily Market Check:

1) Open dashboard
2) Check KPI cards for latest price, RSI, volume
3) Scan price chart for trend continuation
4) Review signals table for today's indicators
5) Decision: Hold, Buy, or Sell based on signal consensus

VIII. Conclusion

This project successfully built an end-to-end stock analytics system integrating Apache Airflow, dbt, Snowflake, and Preset. The system automates the extraction of stock market data from yfinance, transforms it through a modular three-tier architecture (staging-intermediate-marts), and presents actionable insights through interactive dashboards.

A. What We Built

ETL Component: An Airflow pipeline that fetches 180 days of stock prices daily for AAPL and NVDA, cleans the data, and loads it into Snowflake using transaction-based MERGE operations for idempotency.

ELT Component: A dbt project with 7 models calculating technical indicators including moving averages (5, 20, 50-day), RSI, MACD, Bollinger Bands, and daily returns. All transformations are SQL-based, version-controlled, and tested automatically.

Visualization Layer: Preset dashboards displaying real-time technical analysis with 8 interactive charts showing price trends, momentum indicators, volatility patterns, and comprehensive trading signals.

B. Final Thoughts

This implementation demonstrates modern data engineering best practices: separation of ETL and ELT concerns, modular SQL transformations, comprehensive data quality testing, and self-service analytics. The architecture is maintainable, scalable, and provides measurable business value through automated technical analysis.

IX. GitHub Repository

Project code and documentation available at:

Repository URL: https://github.com/Ashley96yy/DATA226_LAB2

References

[1] Apache Software Foundation, "Airflow Documentation." [Online]. Available: https://airflow.apache.org/
[2] Snowflake Inc., "Snowflake Documentation: Forecasting." [Online]. Available: https://docs.snowflake.com/en/user-guide/ml-powered-forecasting
[3] Yahoo Finance, "yfinance Python Library." [Online]. Available: https://pypi.org/project/yfinance/
[4] Docker Inc., "Docker Desktop Installation Guide." [Online]. Available: https://docs.docker.com/get-dock/
[5] Keeyong Han, "SJSU Data226 GitHub Repository." [Online]. Available: https://github.com/keeyong/sjsu-data226-FA25/blob/main/docker-compose.yaml