



Project Overview

This project simulates sensor data collection from cows on a farm and builds a streaming pipeline using:

- A Python Flask API to simulate sensors.
-
- Kafka to transport this data.
-
- Airflow to orchestrate ingestion.
-
- A Kafka consumer to process and store the data in Google Cloud Storage (GCS).
-
- Monitoring is done with Prometheus and Grafana.

Python API to GCS Data storage



Step 1: Python API (Fake Sensor Generator)

Flask app running on port 5000.

- On /cow-data request, generates 30,000 cow telemetry records:
- Temperature, humidity, activity, heart rate, rumination, milk yield, and location.
- Uses a bounded random walk to simulate realistic time-series behavior.
- Data is sent to Kafka sensor-data topic via a Kafka producer.



Why it's useful

- Simulates IoT data in real time.

Helps test the end-to-end pipeline without real hardware.



Scalability

- Horizontal scaling is possible by deploying multiple replicas behind a load balancer.



Cost Efficiency

- Can be containerized and run on-demand in CI/CD pipelines or dev environments.



Layman's Example (Cow Story)

Imagine a farmer has hundreds of cows with smart collars. These collars continuously report health stats like temperature, milk output, and movement. Instead of writing all of this down in a notebook, a digital assistant (the Python API) collects and immediately radios it to a central station (Kafka).

Step 2: Kafka (Transport Layer)

- A message queue that ensures decoupling between producer (API) and consumer.
- sensor-data is the topic used.
- Producer serializes messages as JSON and pushes them to Kafka.
- Kafka acts as a buffer that retains data until it's consumed.

Why it's useful

- Kafka decouples ingestion and processing, enabling asynchronous, scalable architecture.
- Handles bursty traffic reliably.

Scalability

- Partition topics for parallelism.
- Add more brokers to the Kafka cluster.

Cost Efficiency

- Avoids the need for real-time consumers to always be online.

Layman's Example (Cow Story)

Think of Kafka as a big post office. The cows' smart collars send letters with updates. The post office (Kafka) sorts and stores them in bins labeled "sensor-data". Workers (consumers) can come anytime and pick letters to process. Even if no one's reading them right away, the post office keeps them safe.

Step 3: Airflow (Orchestrator)

- Airflow DAG triggers the flow: kicks off tasks to read from the API or Kafka.
- It can be extended to:
- Monitor job success/failure
- Schedule data pulls every hour/day
- Works in tandem with Docker containers (webserver, scheduler, worker).

Why it's useful

- Production-grade orchestration.
- Easily handles retries, dependencies, and time-based triggers.

Scalability

- Distributed task execution with Celery workers.
- Can dynamically scale workers with Kubernetes Executor.

Cost Efficiency

- DAGs are simple Python files; low infra footprint when idle.

Layman's Example (Cow Story)

Imagine a farm supervisor (Airflow) who checks every morning: “Did the cows send their health reports?” If not, he calls the collar manager (API) again. He also tracks how many reports came in, and sets reminders if something fails.

Step 4: Kafka Consumer (Data Processor)

- Listens to sensor-data topic.
- Processes and validates messages.
- Buffers 3000 messages per batch.
- Uploads them to GCS in .ndjson format using google-cloud-storage.

Why it's useful

- Ensures data integrity and handles any needed transformations.
- Stores clean and structured data in cloud storage for analytics or ML.

Scalability

- Can split consumer into multiple threads or replicas by partitioning the topic.
- Uploads can be paralleled.

Cost Efficiency

- Batching before upload reduces GCS API calls.
- No long-running infrastructure needed (runs, finishes, and shuts down).

Layman's Example (Cow Story)

After all the cows have reported their health data, a smart worker (Kafka Consumer) gathers the letters, puts them into boxes of 3000, and drives them to the storage warehouse (GCS). Now a veterinarian or data scientist can read them later.

Output: Data in GCS

- Directory pattern: cow_data/YYYY-MM-DD_HH-MM/batch_X.ndjson
- Designed to be easy to query later using BigQuery or Spark.

Observability (Bonus)

- Prometheus scrapes metrics from Kafka Exporter.
- Grafana dashboards visualize Kafka health and throughput.
- Ensures everything runs smoothly and is debuggable.

|

GCS to Bigquery Ingestion:

This DAG, named gcs_to_bigquery_ingestion, is designed to automate the data ingestion pipeline from Google Cloud Storage (GCS) to BigQuery.

It handles:

- Listing newly arrived data folders in a GCS bucket.
- Validating and importing .ndjson batch files into a BigQuery table.
- Moving processed data into a dedicated "processed" subfolder to avoid duplication.

Core Workflow

Folder Detection:

- The DAG starts by identifying folders in the GCS bucket (kafka-airflow-data-bucket) under the prefix cow_data/.
- It ensures it does not pick folders already under processed_folders.
- Batch File Validation (External Script):
- While not embedded directly in the DAG, there's a separate local Python script that validates each .ndjson file:
- Checks for presence of required fields (e.g., temperature, heart_rate).
- Flags missing fields or invalid JSON formats.
- Intended to be used pre-upload or during test runs to maintain data quality.

BigQuery Ingestion:

- Each valid .ndjson file is ingested into a BigQuery table (smaxtec_dataset.smaxtec_data) using GCSToBigQueryOperator.
- The schema is explicitly defined to ensure type integrity and consistency.
- The ingestion is set to WRITE_APPEND, enabling incremental loading without overwriting existing data.

Post-Ingestion Cleanup:

- After successful data loading, the original folder is:
- Moved to a new location (cow_data/processed_folders/) using GCS copy and delete operations.
- This ensures that the same folder isn't reprocessed in the next DAG run.

Scheduling:

- The DAG is scheduled to run every 2 hours (0 */2 * * *), ensuring near real-time ingestion of streaming batch data.
- Key Configurations and Customization

Connection and Auth:

Uses the default GCP connection (google_cloud_default) with a service account key specified (gcs_keyfile.json).

Retries and Fault Tolerance:

- Each task includes retry logic to handle transient GCS or BigQuery errors gracefully.
- `ignore_unknown_values=True` allows ingestion to proceed even if some unexpected fields exist.

Dynamic Task Generation:

- The DAG dynamically generates ingestion and cleanup tasks for each folder detected at runtime.
- Uses a `safe_id` transformation to make task IDs Airflow-compatible (no dashes or colons).

BigQuery to GCS (Next Step)

After data lands in BigQuery, the natural next stage in the pipeline might include exporting curated data from BigQuery to GCS for:

- External reporting
- Integration with other systems
- Data archival


That phase can be managed by another DAG using the `BigQueryToGCSOperator`, which would complement this ingestion pipeline.

Overall Use Case Fit

This DAG structure is highly suitable for:

- Stream-to-batch pipelines where Kafka delivers data into GCS and Airflow orchestrates movement into BigQuery.
- Data lakes with multiple ingestion points and strict schema validation.
- Multi-stage pipelines, where further transformation, aggregation, or export can follow ingestion.

Feature Engineering

 DAG: `smaxtec_feature_engineering`

 **What is Feature Engineering?**

Feature engineering is the process of transforming raw data into meaningful inputs that machine learning models or analytical tools can better understand. It involves summarizing, aggregating, or flagging certain patterns or behaviors from the data.

In simple terms, we're turning messy, raw cow data into smart insights like how active the cow was each hour, whether it showed abnormal heart rate, or if its rumination suddenly dropped.

Purpose of This DAG

This DAG is designed to:

- Check data completeness (ensure no important field is missing),
- Engineer features using SQL,
- Post-process the generated features for quality and stats,
- Export results to .ndjson format for downstream usage.

What Data Are We Using?

We use the `smaxtec_data` table from BigQuery, which contains telemetry data for each cow. Each record includes:

- Temperature
- Humidity
- Timestamp
- Activity Level
- Heart Rate
- Rumination Minutes
- Location
- Milk Yield
- Sensor ID

Why Are We Doing This?

- To understand cow health patterns over time.
- To detect early signs of illness or stress (e.g., sudden drop in heart rate).
- To label cows as healthy, at-risk, or unwell.
- To build a predictive model or dashboard later.

What Will Be the Benefit?

- Improved animal welfare and early disease detection.
- Helps veterinarians or farm managers act early.
- Foundation for automated alerts and data-driven decisions.
- Better milk yield optimization and farm efficiency.

Feature Engineering Breakdown (4 Key Tasks)

- `check_data_completeness` (Python)
- Checks if any important fields (like temperature or `cow_id`) are missing.
- Fails the DAG early if data is incomplete.
- `run_feature_engineering_sql` (BigQuery SQL)
- Transforms raw data into hourly, 6-hour, and daily aggregates.
- Adds flags for abnormal behavior (e.g., high temp, low rumination).
- Labels each cow's health status.
- `post_process_features` (Python)
- Logs how many rows were produced.
- Can add alerts or sanity checks.
- `generate_hourly_features_ndjson` (Python)
- Converts final BigQuery table into `.ndjson` format.
- Makes data usable for external APIs or model ingestion.

Understanding the SQL Logic:

The SQL script runs several transformations step-by-step:

1. Base Layer

- Pulls only clean (non-null) records and converts timestamps properly.

2. Hourly Aggregates

- Groups data by cow and hour.
- Calculates average metrics like temperature, activity, heart rate.

3. 6-Hour Aggregates

- Smooths short-term trends over rolling 6-hour windows.

4. Daily Aggregates

- Captures long-term health signals and behavior.

5. Herd Benchmarks

- Calculates average values for the whole herd.
- Helps compare individual cows vs. group behavior.

6. Health Flags

Adds boolean flags for abnormal signs:

- High temperature
- Low rumination
- Extreme heart rate
- Sudden drop in vitals

7. Health Labels

Labels cows into:

- healthy: no flags
- at-risk: temp or heart rate flags
- unwell: rumination issues

Machine Learning Algorithms

ML Task 1:

Cow Health Monitoring – ML Task Breakdown

Imagine you're a farmer managing a dairy farm with hundreds of cows. You want to keep your cows healthy, but it's tough to monitor each one 24/7. What if a smart system could analyze behavior and body signals to tell you when a cow is healthy, at risk, or unwell?

That's exactly what this ML task does: It builds and runs a model to predict a cow's health using past behavior and body data.

What is this ML task?

This ML task is designed to predict the health condition of each cow—whether it's healthy, at-risk, or unwell—based on features like temperature, heart rate, activity, etc. The prediction helps automate early alerts for farm managers and vets.

Which data are we using?

We're using processed and aggregated data from this table:

 **smaxtec_features (in BigQuery)**

This data includes:

- Hourly averages of cow vitals and activity
- Health-related flags (e.g. low heart rate)

- Final health labels (used for training the model)

Why are we doing this task?

- To detect health issues early before symptoms worsen.
- To save vet costs and prevent large-scale outbreaks.
- To provide daily automated predictions using machine learning.
- To enable data-driven decisions on large farms.

What is the goal or purpose?

- Train a machine learning model (Random Forest) using historical data and health labels.
- Predict daily health status for each cow using new hourly data.
- Store the predictions in BigQuery so they can be used in dashboards or alerts.

What model is used and why?

- We are using a Random Forest Classifier, which:
- Handles noisy and missing data well
- Works well with mixed features like temperature and rumination
- Is interpretable and fast for predictions

What features are used for prediction?

We are using six core health indicators:

- avg_temperature
- avg_humidity
- avg_activity_level
- avg_heart_rate
- avg_rumination_minutes
- avg_milk_yield

These features are reliable signals of physical health and stress levels.

How often does it run?

This pipeline runs daily using Airflow. It:

- Retrains the model (can be optimized later to do weekly)
- Predicts today's cow health status using the most recent data

What is the label / target column?

 health_status: It's a classification label with values like:

- healthy
- at-risk
- unwell

This label was generated during feature engineering using flags like abnormal heart rate, temperature, etc.

 **Where do we read the prediction data from?**

We use a JSON file:

 /opt/airflow/dags/scripts/features/hourly_features.ndjson

This is the latest hourly features data generated earlier during feature engineering.

 **Where do predictions go?**

BigQuery table:

 smaxtec_dataset.cow_health_monitoring_table


This allows dashboards and alerting systems to access the predicted labels.

 **Why not use deep learning or neural networks?**

Because:

- The dataset size is relatively small
- Random Forest gives great results with less tuning
- It's easier to explain to stakeholders (interpretability)
- Later, we can test more advanced models if needed.

 **Where is the model stored?**

 /opt/airflow/dags/scripts/models/cow_health_model.joblib

It's saved as a joblib file for reusability.

ML task 2:

 **Milk Yield Forecasting – ML Task Breakdown**

Picture this: you're a farm manager trying to predict how much milk each cow will produce tomorrow. You need to plan tank storage, schedule pickups, and detect unusual drops in yield before they impact revenue.

That's where machine learning steps in—quietly monitoring patterns in cow behavior and environment to forecast milk output, automatically and daily.

✅ What is this ML task?

This ML task predicts how much milk a cow is expected to yield using sensor data from her recent activity, such as rumination and temperature.

It's designed to:

- Detect production drops early
- Improve farm resource planning
- Spot potential health or nutrition issues through yield change

📦 Which data are we using?

We use historical data that includes:

- avg_rumination_minutes
- avg_temperature
- Timestamps like feature_hour
- Optionally: cow_id (used for traceability)

🎯 What is the goal or purpose?

- Train a regression model to predict milk yield (liters).
- Use it daily to forecast the yield for each cow using fresh data.
- Save predictions to BigQuery for dashboards, planning, and alerts.

🧠 What kind of model is used?

We're using a regression model (likely a tree-based one like RandomForestRegressor, though it's a placeholder for now).

Why regression?

- Milk yield is a numeric prediction
- Regression models are good at predicting quantities
- They can capture non-linear relationships (e.g., temperature vs. milk)

🔍 What does a prediction record look like?

Each record contains:

- cow_id – ID of the cow (or "unknown_cow" as fallback)
- feature_hour – Timestamp of the input features
- predicted_yield – Forecasted milk quantity
- prediction_time – When the prediction was made

17 **How often does this run?**

 This pipeline runs daily, scheduled via Airflow. The DAG does two things:


- Trains or updates the forecasting model (currently a placeholder).
- Runs predictions using the latest sensor data.

Error Handling & Data Checks

 The prediction script:

- Ensures feature_hour is valid (no 9999-12-31)
- Defaults missing cow_id to "unknown_cow"
- Saves results in NDJSON format for easy ingestion into BigQuery

Where is the model stored?

-  /opt/airflow/dags/ml_models/yield_model.pkl
- Stored as a joblib-serialized file for fast reuse.

ML task 3:

Fertility Prediction – ML Task Breakdown

Imagine a dairy farm trying to optimize breeding cycles. Knowing which cows are fertile and ready for insemination can boost herd productivity and reduce costly guesswork. This ML task predicts fertility status from sensor data collected daily.

What is this ML task?

This task predicts whether a cow is fertile or not using sensor readings like activity level, rumination time, and temperature. It classifies each cow into categories such as:

- fertile / not fertile

- or ready / not ready for insemination (there are similar but slightly different models)
- The purpose is to automate fertility monitoring to improve breeding decisions.

Which data are we using?

- Data is sourced from BigQuery and includes features:
- avg_activity_level — how active the cow is
- avg_rumination_minutes — minutes spent chewing cud
- avg_temperature — average body temperature
- cow_id and feature_hour — identifiers and timestamps
- Labels are derived heuristically from these features using simple rules (e.g., activity > 7, rumination < 60, temperature > 38 means fertile).

What is the goal or purpose?

- Train a classification model (Gradient Boosting Classifier) to predict fertility labels daily.
- Use predictions to decide when cows should be inseminated.
- Save and upload predictions to BigQuery for reporting and decision-making.

What kind of model is used?

- A Gradient Boosting Classifier is used because:
- It handles classification problems well
- Works robustly with tabular data
- Handles non-linear feature interactions effectively

What does a prediction record look like?

Each prediction includes:

- cow_id — unique cow identifier
- feature_hour — timestamp of feature data
- predicted_label — predicted fertility class (fertile or not_fertile)
- prediction_time — when prediction was made

How often does this run?

The pipeline runs daily on Airflow, retraining the model and then generating fresh predictions.

Data Cleaning & Validation

- Ensures feature_hour timestamps are valid and within a reasonable range.
- Filters out rows with missing essential features.
- Converts all data types as needed for consistent processing.

Business Impact

- Automates fertility tracking, reducing manual monitoring.
- Improves insemination timing, leading to better herd productivity.
- Integrates easily with farm management dashboards and alerts.

Conclusion

This project successfully implements an end-to-end data pipeline for intelligent dairy farm monitoring, leveraging modern data engineering and machine learning tools. Using Kafka for real-time data ingestion, the system simulates and streams sensor telemetry for multiple cows. These data streams are processed and stored efficiently, enabling downstream ML tasks.

Three key machine learning pipelines were developed and deployed using Airflow: health status classification, milk yield forecasting, and fertility prediction. Each model was designed using appropriate algorithms (Random Forest, Gradient Boosting, Regression), trained on engineered features, and produces daily predictions for each cow. The results are stored in BigQuery for visualization, alerts, and decision-making.

The pipeline supports scalability, error handling, idempotent processing, and containerized reproducibility via Docker. Monitoring is integrated using Prometheus and Grafana, ensuring visibility into system health and data quality.

Overall, this architecture demonstrates how ML-driven telemetry analytics can optimize livestock health, productivity, and operational efficiency in modern agriculture.