Case Study 4 — Automated ETL Pipeline on GCP Cloud Composer (Infosys)

Role: Data & AI InStep Intern · **Company:** Infosys Ltd, London · **Duration:** June – August 2024

Stack: Google Cloud Composer (managed **Apache Airflow**) · **BigQuery** · **Google Cloud Storage** · Python (pandas) · Google Cloud Shell

Deliverables: 5-DAG automated ETL pipeline · BigQuery curated + summary tables · architecture diagram · pipeline code

> *Built on a **public video-game sales dataset**, not client data. GCP project IDs and bucket names have been redacted in the published code excerpt.*

The Problem

The team needed a **repeatable, scheduled data-warehousing workflow** on GCP — raw files landing in cloud storage, then cleaned, loaded and aggregated into BigQuery **without anyone touching it manually**. Doing this by hand doesn't scale: it's slow, inconsistent, and the moment someone forgets to run a step the downstream reporting is stale or wrong.

What I Built

An **end-to-end automated ETL pipeline** on **Cloud Composer** (Google's managed Apache Airflow), split into **five DAGs** so each stage is independently retryable and observable, running **daily on a schedule**:

#	DAG	What it does
1	`data_ingestion_dag`	Reads the raw `vgsales.csv` from a **GCS** bucket and writes a combined dataset back to storage
2	`data_cleaning_dag`	**Deduplicates** and drops incomplete records, writing a `processed_dataset.csv`
3	`bigquery_dag`	Loads GCS → **BigQuery** via `GCSToBigQueryOperator` with an **explicit 11-field schema** and `WRITE_TRUNCATE`

4	`bigquery_vgsales_aggregation_dag`	Runs a `CREATE OR REPLACE TABLE … AS SELECT` to build a **summary table** aggregating sales by year and genre
5	*(environment DAG)*	Installs/pins the Python dependencies the workers need so tasks run reliably

16,598 raw records flow through ingestion → preprocessing → transformation → BigQuery load on every run. Cleaning drops **307 incomplete records**, so **16,291 rows** land in the curated table, and the aggregation builds a **389-row** summary table grouped by year and genre. (De-duplication removes nothing on this particular source — it's already unique — but the step stays in the DAG so a future dirtier extract can't silently double-count.)

The Techniques

- **Modular DAG design** — one responsibility per DAG, with `retries` and a `retry_delay`, so a transient failure in one stage doesn't force a full re-run.
- **Explicit BigQuery schema definition** — all 11 fields typed and moded (`Rank`, `Name`, `Platform`, `Year`, `Genre`, `Publisher`, and the five regional/global sales measures) rather than relying on autodetect.
- **Idempotent loads** — `WRITE_TRUNCATE` plus `skip_leading_rows` means re-running the pipeline produces the same table, never duplicates.
- **In-warehouse aggregation** — a `BigQueryInsertJobOperator` builds the curated `summary_table` with SQL inside BigQuery, so the heavy grouping runs where the data lives instead of being pulled into Python.
- **Vectorised pandas** for the cleaning step, and batch processing to cut execution time.

Engineering the pipeline to be reliable

Two engineering problems had to be solved for the pipeline to run dependably in a managed Airflow environment — both are the kind of thing that decides whether a pipeline survives production:

1. **Deterministic BigQuery loading.** Relying on schema autodetect makes ingestion fragile — column types drift and loads reject rows. I **defined the schema explicitly** in the operator (all 11 fields typed and moded) and used `WRITE_TRUNCATE`, which made every load **validated and idempotent**. I then added pre-aggregated summary tables so downstream queries read clean, typed columns instead of scanning raw data.
2. **Reproducible worker dependencies.** Cloud Composer workers need their Python libraries provisioned reliably or tasks fail to import. I handled this by **developing installation scripts in**

Google Cloud Shell, uploading them to **Cloud Storage**, and executing them through Airflow — with dedicated `install_pandas` / `install_airflow` tasks in the DAG and custom test scripts to verify module versions.

Throughout, I kept my manager and stakeholders updated on progress and risks **early**, with root causes and a mitigation plan, rather than surfacing problems late.

The Impact

- **Delivered on time**, running end-to-end on schedule.
- **Fully automated, scheduled** pipeline — raw CSV in GCS becomes an analysis-ready BigQuery summary table with **no manual steps**.
- **Reliable and repeatable** — idempotent loads and per-stage retries mean the same run produces the same result.
- **Faster execution** after rewriting the preprocessing with vectorised pandas and batch processing.
- **Query performance improved** by designing typed schemas and pre-aggregated summary tables, so analysts query a small curated table instead of scanning raw data.

What's in this case study

- **Architecture diagram** — GCS (raw) → Cloud Composer / Airflow DAGs (ingest → clean → load) → BigQuery (curated → summary).
- **Pipeline Code Highlights** — a curated, syntax-highlighted excerpt of the actual DAG code (project IDs and buckets redacted).
- **Airflow screenshot** — the data-ingestion DAG running green end-to-end in the real Composer environment, including the `install_pandas` / `install_airflow` dependency-provisioning tasks.
- **Interactive run demo** (`pipeline-run-demo.html`) — replays a full scheduled run: all five DAGs moving through the scheduler, a task **failing and recovering on retry**, and the BigQuery tables it produces. Real task IDs, real retry settings, and the real vgsales row counts; the UI itself is a mockup.

Skills demonstrated: data engineering · **Apache Airflow / Cloud Composer** orchestration · **ETL pipeline design** · **BigQuery** schema design & SQL aggregation · Google Cloud Storage · Python (pandas) · data cleaning & QA (deduplication, type casting, schema validation) ·

debugging & dependency management · query optimisation · risk communication with stakeholders.