**Roster Automation Pipeline**

**Summary**  
This document describes an end-to-end Roster Automation pipeline that exports raw roster JSON events stored in Oracle, writes them to S3, invokes two AWS Lambda transforms (ISF normalization and Dart validation), and finally loads cleaned data into Amazon Redshift. Orchestration, retries, and observability are handled in a single Airflow DAG that invokes Lambdas synchronously and loads the result to Redshift.

**Table of contents**

1. Project overview
2. Objectives
3. High-level architecture & dataflow
4. Technology stack & packaging recommendations
5. Component details
   * Kafka → Oracle consumer
   * Oracle export → S3 Raw (Airflow task)
   * Lambda: ISF transform
   * Lambda: Dart transform & validation
   * Airflow DAG: single orchestrator
   * Redshift load & upsert
6. IAM & Security
7. Deployment & CI/CD
8. Monitoring, observability & error handling
9. Scaling considerations & alternatives
10. Appendix: SQL, Terraform and requirements snippets
11. Full code (Lambdas + Airflow DAG)

**1. Project overview**

The pipeline ingests provider roster events (PLM Portal → Kafka → Oracle). Airflow periodically exports unprocessed rows from Oracle to S3 (Raw), invokes two Lambda functions for transformation/validation, and then loads validated data to Redshift for analytics. Invalid rows are written to an error bucket for replay/inspection.

Key design principle: keep Airflow as the single orchestrator and use Lambda for isolated, versioned transform logic.

**2. Objectives**

* Single orchestrator (Airflow) for scheduling, retrying, lineage, and visibility.
* Offload row/record transforms to Lambda for quick deploys, autoscaling and isolation.
* Preserve traceability: every row/file includes provenance metadata.
* Route invalid rows to an error bucket for replay.
* Use least-privilege IAM roles and store secrets in Secrets Manager.

**3. Architecture & dataflow**

PLM Portal → Kafka → Kafka Consumer → Oracle (ROSTER\_RAW)  
→ Airflow DAG:

1. export\_oracle\_to\_s3 (export JSONL files to S3 raw)
2. invoke\_isf\_lambda (synchronous) → writes Parquet ISF to S3 isf-bucket
3. invoke\_dart\_lambda (synchronous) → writes CSV(s) to S3 dart-bucket and invalid rows to error-bucket
4. load\_to\_redshift → COPY into staging → MERGE/UPSERT to final table

S3 layout example:

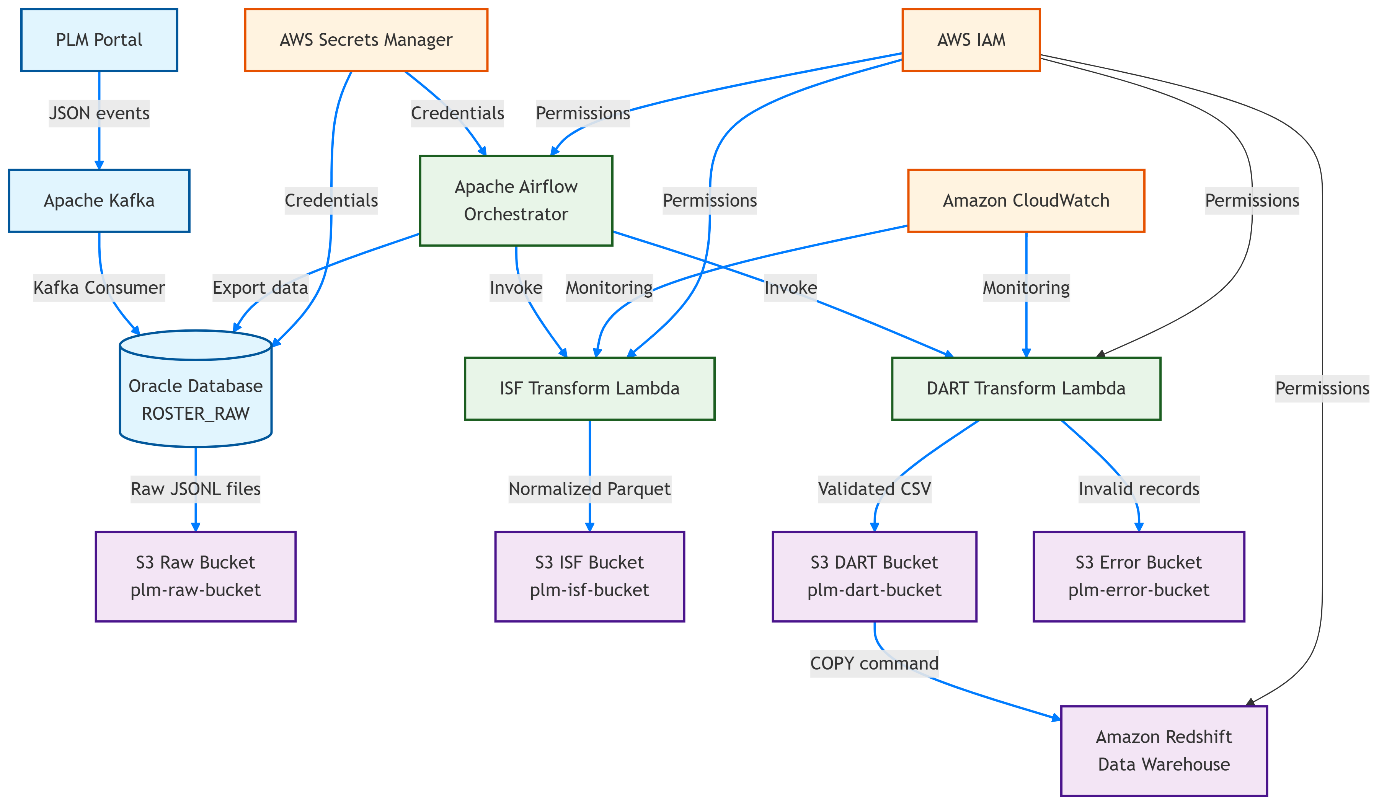
* s3://plm-raw-bucket/year=YYYY/month=MM/day=DD/export\_<runid>.jsonl
* s3://plm-isf-bucket/isf/<job\_id>/... .parquet
* s3://plm-dart-bucket/dart/<job\_id>/... .csv
* s3://plm-error-bucket/errors/<job\_id>/... .parquet

**4. Technology stack & packaging recommendations**

* **Orchestration:** Apache Airflow (providers: amazon)
* **Transforms:** AWS Lambda (container images if using pandas/pyarrow)
* **Storage:** AWS S3 (raw, isf, dart, error)
* **Warehouse:** Amazon Redshift
* **Source DB:** Oracle (ROSTER\_RAW)
* **Secrets:** AWS Secrets Manager
* **Infra as Code:** Terraform
* **CI/CD:** GitHub Actions / CodeBuild / CodePipeline
* **Languages/Libraries:** Python 3.10+, pandas, pyarrow (or fastparquet), boto3, oracledb (only for exporter)

**Packaging note:** Heavy dependencies (pandas/pyarrow) — prefer Lambda container images (Amazon Linux), or precompiled Lambda layers.

**5. Detailed components**

****

**5.1 Kafka → Oracle consumer**

A durable consumer (ECS/EKS/EC2) listens to PLM Kafka topics and writes each JSON message to ROSTER\_RAW:  
Schema:

id NUMBER PK, payload CLOB, source VARCHAR2(200), created\_at TIMESTAMP, export\_status VARCHAR2(20), export\_at TIMESTAMP

Write lightweight, defer transforms to Lambda. Include metadata: received\_at, topic, partition, offset.

**5.2 Oracle export → S3 Raw (Airflow)**

**Function:** Export EXPORT\_STATUS IS NULL rows to JSON Lines files and upload to S3 (multipart if large). Update EXPORT\_STATUS='EXPORTED' and EXPORT\_AT=UTC\_TIMESTAMP. Push S3 keys to XCom for next steps.

Key practices:

* Stream rows with fetchmany(); avoid loading entire result in memory.
* Partition files by date to simplify retention and downstream wildcard COPY.
* Maintain export ledger table to avoid double processing.

**5.3 AWS Lambda — ISF transform**

**Purpose:** Normalize raw JSON → canonical ISF Parquet.

**Trigger:** Synchronous invocation by Airflow with payload:

{"s3\_keys":["raw/.../export\_a.jsonl"], "job\_id": "airflow-2025-..."}

**Responsibilities:**

* Read JSONL from S3, parse, normalize column names and datatypes, dedupe (provider\_id + roster\_date), add provenance columns (\_src\_bucket, \_src\_key, \_ingested\_at, job\_id), and write Parquet to ISF bucket.
* Return metadata: isf\_keys, processed\_count, errors.

**Implementation notes:**

* Chunk large inputs to avoid OOM.
* If file sizes or processing time exceed Lambda limits, either split files at export, or use Glue/ECS for large jobs.

**5.4 AWS Lambda — Dart transform & validation**

**Purpose:** Validate ISF Parquet → Dart CSV(s) and route invalid rows.

**Trigger:** Synchronous invocation by Airflow with payload:

{"isf\_keys":["isf/..part0.parquet"], "job\_id": "airflow-2025-..."}

**Responsibilities:**

* Read Parquet(s). Run business validations (required columns, date formats, regex checks).
* Write valid rows into CSV(s) in DART bucket (ready for Redshift COPY).
* Write invalid rows (with \_validation\_error) into ERROR bucket.
* Return dart\_keys, error\_keys, total\_valid, total\_invalid.

**Validation examples:**

* provider\_id non-null & regex.
* roster\_date parseable & within allowed date range.
* numeric fields within bounds.

**Transformation**

1. **Oracle Export to S3 (Pre-Transformation)**
   * Data exported from ROSTER\_RAW table in Oracle.
   * Streaming fetch (fetchmany()) used to avoid memory overload on large payloads.
   * Rows written as **JSONL** files partitioned by year/month/day in **Raw S3 bucket**.
   * Metadata (export\_status, export\_at, run\_id) updated in Oracle to avoid duplicates.
2. **ISF Lambda Transformation**
   * Reads JSONL files from Raw bucket.
   * **Normalization:**
     + Standardizes column names (CamelCase → snake\_case).
     + Converts date fields into UTC timestamps.
     + Handles missing fields by defaulting to NULL.
   * **Deduplication:**
     + Removes duplicates based on (provider\_id, roster\_date).
     + Keeps the latest record in case of duplicates.
   * **Provenance Tracking:**
     + Adds lineage columns (\_src\_bucket, \_src\_key, \_ingested\_at, job\_id).
   * Writes transformed data as **Parquet** to ISF bucket (optimized for analytics).
3. **Dart Lambda Transformation**
   * Consumes ISF Parquet files.
   * Applies business validation (see below).
   * **Valid records:**
     + Renamed to Redshift schema (providerId, rosterDate, providerName).
     + Written as **CSV** to Dart bucket (for COPY into Redshift).
   * **Invalid records:**
     + Tagged with \_validation\_error.
     + Written as **Parquet** to Error bucket for troubleshooting.
   * Returns counts (total\_valid, total\_invalid) to Airflow for monitoring.

**Validation**

1. **Schema Validation**
   * Ensures required fields exist (provider\_id, roster\_date, name).
   * Enforces data types:
     + provider\_id → string (alphanumeric, regex validated).
     + roster\_date → valid date (no future dates beyond 1 year).
     + numeric fields → within business-defined ranges.
2. **Business Rule Validation**
   * **Provider ID:** Must be present, unique, and match regex ^[A-Z0-9\_-]{5,20}$.
   * **Roster Date:** Must be a valid timestamp within operational window (e.g., last 2 years, not future > 1 year).
   * **Name:** Non-empty, no invalid characters.
   * **Referential Integrity:** Cross-check provider\_id against master provider dataset (if available).
3. **Data Quality Checks**
   * **Null Checks:** No nulls in required fields.
   * **Duplicate Detection:** Deduplicate based on (provider\_id, roster\_date).
   * **Outlier Detection:** Numeric fields like hours\_worked, rate validated against thresholds.
   * **File-Level Checks:**
     + Minimum/maximum record counts.
     + Schema drift detection (extra/missing columns).
4. **Error Handling & Replay**
   * Invalid rows are quarantined in **Error bucket** with descriptive \_validation\_error.
   * All transformations tagged with job\_id → ensures traceability and replay.
   * Airflow retries failed steps automatically (max 2 retries, 5 min delay).
   * Replay strategy:
     + Fix validation rules or input data.
     + Re-run only error files for selected job\_id.

**5.5 Airflow orchestration (single DAG)**

Airflow acts as single control plane:

* export\_oracle\_to\_s3 → pushes raw\_keys to XCom.
* invoke\_isf\_lambda → gets isf\_keys from Lambda response.
* invoke\_dart\_lambda → receives dart\_keys and error\_keys.
* load\_to\_redshift → COPY per dart file and MERGE to final table.

Idempotency: Include job\_id in S3 keys and maintain an ETL runs table. See full DAG in section **11**.

**5.6 Redshift load & upsert**

* COPY CSV(s) into roster\_staging (use IAM role or COPY with credentials).
* MERGE/UPSERT to roster\_final using deterministic keys and job\_id.
* Keep audit columns (\_loaded\_at, \_source\_file, \_job\_id).

**6. Security & IAM**

**Principles:** least privilege, Secrets Manager for credentials, SSE-KMS for S3.

**Roles:**

* **Lambda role:** s3:GetObject, s3:PutObject on raw/isf/dart/error buckets + CloudWatch logs.
* **Airflow role:** lambda:InvokeFunction, s3:ListBucket/GetObject/PutObject as needed, secretsmanager:GetSecretValue.
* **Redshift COPY role:** s3:GetObject on dart bucket.

**Network:** Use VPC endpoints for S3 & Secrets Manager when Airflow runs in a VPC.

**7. Deployment & CI/CD**

**Lambdas:** Build container images with requirements, push to ECR, update Lambda function with new image and alias.  
**Airflow DAGs:** Deploy via git-sync or CI to scheduler & workers. Keep DAG config in Airflow Variables / Connections.  
**Terraform:** Manage buckets, IAM roles, Lambda resources, and Redshift cluster.

**8. Monitoring, observability & error handling**

* **Airflow** for DAG status, retries and alerts.
* **CloudWatch** logs and metrics for Lambdas; set alarms on error rate and duration.
* **DLQs** or error S3 bucket for corrupt/invalid event sets.
* **Lineage:** include \_src\_key, \_ingested\_at, \_job\_id in outputs.
* **Alerts:** SNS/Slack for failure escalations.

**9. Scaling considerations & alternatives**

* If files frequently exceed Lambda memory/time → use Glue (Spark), EMR or run transforms in ECS/EKS.
* Frequent small Lambda invocations cost vs. fewer Glue runs — compare based on volume & latency SLAs.
* Use schema versioning and compatibility checks to handle schema drift.

## Challenges Faced & Solutions

### 1. Large Payloads in Oracle Export

**Challenge:**  
Exporting data from ROSTER\_RAW in Oracle often involved very large JSON payloads. Loading the entire dataset in memory caused slow performance and occasional out-of-memory issues.

**Solution:**  
Implemented streaming using fetchmany() and partitioned the output into JSONL files. This ensured memory efficiency and allowed for parallel downstream processing.

### 2. AWS Lambda Limitations

**Challenge:**  
Transformations using pandas/pyarrow frequently hit AWS Lambda’s memory (up to 10 GB) and execution time (15 min) limits. Packaging heavy dependencies was also difficult.

**Solution:**  
Used AWS Lambda container images to package large dependencies and introduced file-chunking logic during the export phase. For larger datasets, the design allows switching to AWS Glue or Spark for distributed processing.

### 3. Idempotency & Duplicate Data

**Challenge:**  
Retries in Airflow or Lambda re-runs risked duplicate inserts in Redshift.

**Solution:**  
Introduced **job\_id tagging** for all S3 outputs and Redshift audit columns (\_job\_id, \_loaded\_at). The MERGE/UPSERT process ensures only the latest data is retained, making the pipeline idempotent.

### 4. Schema Drift & Data Validation

**Challenge:**  
Source roster JSON had inconsistent field names and evolving schemas. This caused downstream validation errors and potential data corruption.

**Solution:**  
The ISF Lambda normalized schema and standardized datatypes. The Dart Lambda enforced strict business validations (e.g., provider\_id non-null, valid date ranges). Invalid records were routed to an error bucket for later inspection and replay.

### 5. Redshift MERGE Performance

**Challenge:**  
Redshift MERGE operations slowed down with increasing data volumes in staging tables.

**Solution:**  
Optimized by batching COPY operations, applying sort/dist keys on Redshift, and cleaning up old staging partitions before each insert. This improved query performance and reduced table bloat.

### 6. Error Handling & Observability

**Challenge:**  
Failures in transformation or Airflow orchestration sometimes left partial or corrupted data in S3.

**Solution:**  
Implemented CloudWatch alarms and Airflow retries for fault tolerance. Invalid records were redirected to a dedicated **error bucket**, ensuring traceability and allowing reprocessing without affecting valid data.

### 7. Security & IAM Complexity

**Challenge:**  
Managing IAM policies across Airflow, Lambda, Redshift, and S3 introduced complexity and risk of over-privileged access.

**Solution:**  
Applied least-privilege IAM roles, used AWS Secrets Manager for credentials, enabled SSE-KMS encryption for S3, and restricted data access using VPC endpoints.

### 8. Scaling & Cost Tradeoffs

**Challenge:**  
Frequent small file exports increased Lambda invocation costs, while very large files risked Lambda timeouts.

**Solution:**  
Balanced export logic by splitting large files into manageable chunks. Monitored data growth and designed the system to switch to Glue/Spark-based transformations when data volumes exceed Lambda’s efficiency.

**10. Appendix — snippets & operational queries**

**Oracle staging table skeleton**

CREATE TABLE roster\_raw (

id NUMBER PRIMARY KEY,

payload CLOB,

source VARCHAR2(200),

created\_at TIMESTAMP DEFAULT SYSTIMESTAMP,

export\_status VARCHAR2(20),

export\_at TIMESTAMP

);

**Redshift staging / final**

CREATE TABLE public.roster\_staging (

providerid varchar(256),

rosterdate timestamp,

providername varchar(512),

\_source\_file varchar(1024),

\_job\_id varchar(128),

\_loaded\_at timestamp default current\_timestamp

);

CREATE TABLE public.roster\_final (

providerid varchar(256) PRIMARY KEY,

rosterdate timestamp,

providername varchar(512),

\_last\_updated timestamp

);

**Upsert pattern (pseudo)**

BEGIN;

DELETE FROM public.roster\_final f

USING public.roster\_staging s

WHERE f.providerid = s.providerid

AND s.\_job\_id = 'airflow-20250910-...';

INSERT INTO public.roster\_final (providerid, rosterdate, providername, \_last\_updated)

SELECT providerid, rosterdate, providername, CURRENT\_TIMESTAMP

FROM public.roster\_staging

WHERE \_job\_id = 'airflow-20250910-...';

COMMIT;

**Backlog query**

SELECT TRUNC(created\_at) day, COUNT(\*)

FROM roster\_raw

WHERE export\_status IS NULL

GROUP BY TRUNC(created\_at)

ORDER BY day DESC;

**11. Lambdas & Airflow DAG**

**Note:** adjust environment variables, bucket names, Lambda names and dependency packaging for your environment. For heavy libs, prefer Lambda container images. The code below is ready to be copied; adapt small env-specific details.

**11.1 isf\_transform\_lambda.py (Lambda handler — normalize JSONL → Parquet)**

# isf\_transform\_lambda.py

import os

import io

import json

import math

import boto3

import pandas as pd

from datetime import datetime, timezone

S3 = boto3.client("s3")

RAW\_BUCKET = os.environ["S3\_RAW\_BUCKET"]

ISF\_BUCKET = os.environ["ISF\_BUCKET"]

ISF\_PREFIX = os.environ.get("ISF\_PREFIX", "isf/")

MAX\_ROWS\_PER\_CHUNK = int(os.environ.get("MAX\_ROWS\_PER\_CHUNK", "100000"))

def read\_jsonl\_s3(bucket: str, key: str):

obj = S3.get\_object(Bucket=bucket, Key=key)

text = obj["Body"].read().decode("utf-8")

for line in text.splitlines():

if line.strip():

try:

yield json.loads(line)

except Exception:

yield {"\_raw": line}

def normalize\_df(df: pd.DataFrame) -> pd.DataFrame:

# normalize column names

df.columns = [c.strip().lower().replace(" ", "\_") for c in df.columns]

# ensure required columns exist

if "provider\_id" not in df.columns:

df["provider\_id"] = None

# handle common date column names

if "date" in df.columns:

df["date"] = pd.to\_datetime(df["date"], errors="coerce").dt.tz\_localize(None)

# dedupe by provider\_id + date if present

dedupe\_cols = [c for c in ("provider\_id", "date") if c in df.columns]

if dedupe\_cols:

df = df.drop\_duplicates(subset=dedupe\_cols, keep="last")

return df

def write\_parquet\_bytes(df: pd.DataFrame) -> bytes:

buf = io.BytesIO()

df.to\_parquet(buf, index=False)

return buf.getvalue()

def handler(event, context):

"""

event = {"s3\_keys":[...], "job\_id":"..."}

"""

keys = event.get("s3\_keys", [])

job\_id = event.get("job\_id", f"job-{int(datetime.now(timezone.utc).timestamp())}")

produced = []

errors = []

processed = 0

for key in keys:

try:

rows = list(read\_jsonl\_s3(RAW\_BUCKET, key))

if not rows:

continue

n\_chunks = math.ceil(len(rows) / MAX\_ROWS\_PER\_CHUNK)

for idx in range(n\_chunks):

chunk = rows[idx \* MAX\_ROWS\_PER\_CHUNK:(idx + 1) \* MAX\_ROWS\_PER\_CHUNK]

df = pd.DataFrame(chunk)

df = normalize\_df(df)

df["\_src\_bucket"] = RAW\_BUCKET

df["\_src\_key"] = key

df["\_ingested\_at"] = pd.Timestamp.utcnow()

filename = key.split("/")[-1].rsplit(".", 1)[0]

out\_key = f"{ISF\_PREFIX}{job\_id}/{filename}--{job\_id}--part{idx}.parquet"

S3.put\_object(Bucket=ISF\_BUCKET, Key=out\_key, Body=write\_parquet\_bytes(df))

produced.append(out\_key)

processed += len(df)

except Exception as e:

errors.append({"key": key, "error": str(e)})

status = "OK" if not errors else "ERROR"

return {"status": status, "isf\_keys": produced, "processed\_count": processed, "errors": errors}

**Packaging tip:** test with a local Docker image matching Lambda runtime. Ensure pandas/pyarrow are available.

**11.2 dart\_transform\_lambda.py (Lambda handler — validate & produce CSV + errors)**

# dart\_transform\_lambda.py

import os

import io

import json

import boto3

import pandas as pd

from datetime import datetime, timezone

S3 = boto3.client("s3")

ISF\_BUCKET = os.environ["ISF\_BUCKET"]

DART\_BUCKET = os.environ["DART\_BUCKET"]

ERROR\_BUCKET = os.environ["ERROR\_BUCKET"]

DART\_PREFIX = os.environ.get("DART\_PREFIX", "dart/")

ERROR\_PREFIX = os.environ.get("ERROR\_PREFIX", "errors/")

# business required fields

REQUIRED = ["provider\_id", "date", "name"]

def read\_parquet\_s3(bucket, key):

obj = S3.get\_object(Bucket=bucket, Key=key)

return pd.read\_parquet(io.BytesIO(obj["Body"].read()))

def write\_csv\_s3(text, bucket, key):

S3.put\_object(Bucket=bucket, Key=key, Body=text.encode("utf-8"))

def write\_parquet\_s3\_bytes(df, bucket, key):

buf = io.BytesIO()

df.to\_parquet(buf, index=False)

S3.put\_object(Bucket=bucket, Key=key, Body=buf.getvalue())

def validate(df: pd.DataFrame):

df["\_validation\_error"] = ""

valid\_mask = pd.Series(True, index=df.index)

for col in REQUIRED:

if col not in df.columns:

df["\_validation\_error"] += f"missing:{col};"

valid\_mask &= False

else:

if col == "date":

parsed = pd.to\_datetime(df["date"], errors="coerce")

bad = parsed.isna()

df.loc[bad, "\_validation\_error"] += "bad\_date;"

valid\_mask[bad] = False

return df[valid\_mask].copy(), df[~valid\_mask].copy()

def handler(event, context):

"""

event = {"isf\_keys":[...], "job\_id":"..."}

"""

keys = event.get("isf\_keys", [])

job\_id = event.get("job\_id", f"job-{int(datetime.now(timezone.utc).timestamp())}")

dart\_keys = []

error\_keys = []

total\_valid = 0

total\_invalid = 0

for key in keys:

try:

df = read\_parquet\_s3(ISF\_BUCKET, key)

valid\_df, invalid\_df = validate(df)

base\_name = key.split('/')[-1].replace('.parquet', '')

if not valid\_df.empty:

# rename columns to target column names expected by Redshift/COPY

out\_df = valid\_df.rename(columns={

"provider\_id": "providerId",

"date": "rosterDate",

"name": "providerName"

})

csv\_text = out\_df.to\_csv(index=False)

out\_key = f"{DART\_PREFIX}{job\_id}/{base\_name}--{job\_id}.csv"

write\_csv\_s3(csv\_text, DART\_BUCKET, out\_key)

dart\_keys.append(out\_key)

total\_valid += len(out\_df)

if not invalid\_df.empty:

err\_key = f"{ERROR\_PREFIX}{job\_id}/{base\_name}--{job\_id}.parquet"

write\_parquet\_s3\_bytes(invalid\_df, ERROR\_BUCKET, err\_key)

error\_keys.append(err\_key)

total\_invalid += len(invalid\_df)

except Exception as e:

error\_keys.append({"isf\_key": key, "error": str(e)})

return {

"status": "OK",

"dart\_keys": dart\_keys,

"error\_keys": error\_keys,

"total\_valid": total\_valid,

"total\_invalid": total\_invalid

}

**11.3 dags/full\_roster\_pipeline\_lambda.py (Airflow DAG — full orchestration)**

# dags/full\_roster\_pipeline\_lambda.py

from datetime import datetime, timedelta

from airflow import DAG

from airflow.operators.python import PythonOperator

from airflow.providers.amazon.aws.hooks.lambda\_function import AwsLambdaHook

import os

import json

# env-config / defaults (override via Airflow Variables or env vars)

S3\_RAW\_BUCKET = os.environ.get("S3\_RAW\_BUCKET", "plm-raw-bucket")

ISF\_LAMBDA\_NAME = os.environ.get("ISF\_LAMBDA\_NAME", "isf-transform-lambda")

DART\_LAMBDA\_NAME = os.environ.get("DART\_LAMBDA\_NAME", "dart-transform-lambda")

DART\_BUCKET = os.environ.get("DART\_BUCKET", "plm-dart-bucket")

REDSHIFT\_CONN\_ID = os.environ.get("REDSHIFT\_CONN\_ID", "redshift\_conn")

AWS\_CONN\_ID = os.environ.get("AWS\_CONN\_ID", "aws\_default")

default\_args = {

"owner": "data-eng",

"depends\_on\_past": False,

"email\_on\_failure": True,

"retries": 2,

"retry\_delay": timedelta(minutes=5),

}

with DAG(

dag\_id="full\_roster\_pipeline\_lambda",

default\_args=default\_args,

start\_date=datetime(2025, 1, 1),

schedule\_interval="0 \* \* \* \*",

catchup=False,

max\_active\_runs=1,

) as dag:

def export\_oracle\_to\_s3(\*\*kwargs):

"""

Implement export logic in a module `roster\_export.export\_to\_s3`.

This function must return a dict: {"s3\_keys": [...], "run\_id": "..."}.

"""

run\_id = f"airflow-{datetime.utcnow().strftime('%Y%m%dT%H%M%S')}"

# Example: import your real exporter module

from roster\_export.exporter import export\_to\_s3 # implement this separately

result = export\_to\_s3(run\_id=run\_id)

kwargs["ti"].xcom\_push(key="raw\_keys", value=result.get("s3\_keys", []))

kwargs["ti"].xcom\_push(key="run\_id", value=run\_id)

return result

export\_task = PythonOperator(

task\_id="export\_oracle\_to\_s3",

python\_callable=export\_oracle\_to\_s3,

provide\_context=True,

)

def invoke\_lambda(lambda\_name, payload):

hook = AwsLambdaHook(function\_name=lambda\_name, aws\_conn\_id=AWS\_CONN\_ID)

resp = hook.invoke\_lambda(payload=json.dumps(payload), invocation\_type="RequestResponse")

if isinstance(resp, (bytes, bytearray)):

resp = resp.decode("utf-8")

try:

return json.loads(resp)

except Exception:

return resp

def invoke\_isf\_lambda(\*\*kwargs):

ti = kwargs["ti"]

raw\_keys = ti.xcom\_pull(key="raw\_keys", task\_ids="export\_oracle\_to\_s3") or []

run\_id = ti.xcom\_pull(key="run\_id", task\_ids="export\_oracle\_to\_s3") or f"airflow-{datetime.utcnow().strftime('%Y%m%dT%H%M%S')}"

if not raw\_keys:

return {"status": "NO\_FILES", "isf\_keys": []}

payload = {"s3\_keys": raw\_keys, "job\_id": run\_id}

resp = invoke\_lambda(ISF\_LAMBDA\_NAME, payload)

ti.xcom\_push(key="isf\_response", value=resp)

return resp

isf\_invoke = PythonOperator(task\_id="invoke\_isf\_lambda", python\_callable=invoke\_isf\_lambda, provide\_context=True)

def invoke\_dart\_lambda(\*\*kwargs):

ti = kwargs["ti"]

isf\_resp = ti.xcom\_pull(key="isf\_response", task\_ids="invoke\_isf\_lambda") or {}

isf\_keys = isf\_resp.get("isf\_keys", [])

run\_id = ti.xcom\_pull(key="run\_id", task\_ids="export\_oracle\_to\_s3")

if not isf\_keys:

return {"status": "NO\_ISF", "dart\_keys": []}

payload = {"isf\_keys": isf\_keys, "job\_id": run\_id}

resp = invoke\_lambda(DART\_LAMBDA\_NAME, payload)

ti.xcom\_push(key="dart\_response", value=resp)

return resp

dart\_invoke = PythonOperator(task\_id="invoke\_dart\_lambda", python\_callable=invoke\_dart\_lambda, provide\_context=True)

def load\_to\_redshift(\*\*kwargs):

ti = kwargs["ti"]

dart\_resp = ti.xcom\_pull(key="dart\_response", task\_ids="invoke\_dart\_lambda") or {}

dart\_keys = dart\_resp.get("dart\_keys", [])

run\_id = ti.xcom\_pull(key="run\_id", task\_ids="export\_oracle\_to\_s3")

if not dart\_keys:

return {"status": "NO\_DART"}

# perform COPY per file (or implement single COPY with prefix)

# Here we show a simple psycopg2 approach using Redshift connection.

import boto3

from airflow.hooks.base import BaseHook

import psycopg2

# fetch redshift connection details (implement as you prefer)

conn = BaseHook.get\_connection(REDSHIFT\_CONN\_ID)

conn\_string = f"host={conn.host} port={conn.port} dbname={conn.schema} user={conn.login} password={conn.password}"

pg = psycopg2.connect(conn\_string)

cur = pg.cursor()

# Use an IAM role ARN or credentials for COPY depending on your setup.

redshift\_schema = os.environ.get("REDSHIFT\_SCHEMA", "public")

redshift\_table = os.environ.get("REDSHIFT\_TABLE", "roster\_staging")

iam\_role = os.environ.get("REDSHIFT\_COPY\_ROLE\_ARN") # recommended

for key in dart\_keys:

s3\_path = f"s3://{DART\_BUCKET}/{key}"

copy\_sql = f"""

COPY {redshift\_schema}.{redshift\_table}

FROM '{s3\_path}'

IAM\_ROLE '{iam\_role}'

CSV

IGNOREHEADER 1

TIMEFORMAT 'auto';

"""

cur.execute(copy\_sql)

pg.commit()

# After COPYs, call stored procedure or run MERGE logic here (optional)

cur.close()

pg.close()

return {"status": "COPIED", "files": dart\_keys}

load\_task = PythonOperator(task\_id="load\_to\_redshift", python\_callable=load\_to\_redshift, provide\_context=True)

export\_task >> isf\_invoke >> dart\_invoke >> load\_task

**Notes:**

* The DAG uses AwsLambdaHook.invoke\_lambda. Ensure apache-airflow-providers-amazon is installed and your Airflow environment has network access to AWS.
* The export\_to\_s3 module should be implemented by you — it should stream Oracle rows to JSONL files and upload them to S3. It must return the uploaded S3 keys and run\_id.

**12. Operational checklist & recommendations**

* Maintain etl\_runs table with job\_id, start/end, counts, and status.
* Keep run-specific S3 prefixes for easier atomic COPY operations.
* Set CloudWatch alarms on Lambda error rates and durations.
* Implement unit tests for Lambda transform logic and integration tests for DAG.
* For high-volume data, evaluate Glue or Spark-based transforms.

**📌 Roster Automation Project – Interview Q&A**

**Project Understanding**

**Q1. Can you explain your Roster Automation project end-to-end?**  
**A1.**  
This project automates ingestion, transformation, validation, and loading of provider roster data from a PLM portal into Amazon Redshift for analytics.

* **Ingestion:** Data flows PLM Portal → Kafka → Oracle (ROSTER\_RAW).
* **Export:** Airflow exports new/unprocessed rows from Oracle to S3 as JSONL files.
* **Transformations:**
  + **ISF Lambda** → Normalizes JSON to canonical ISF Parquet format.
  + **Dart Lambda** → Validates ISF data, writes valid rows to S3 as CSV (Dart bucket) and invalid rows to an error bucket.
* **Loading:** Airflow then loads validated CSV files to Redshift (staging), followed by a MERGE/UPSERT into final tables.
* **Orchestration:** Entire pipeline is controlled by a single Airflow DAG.

**Architecture & Design**

**Q2. Why did you choose Airflow as the orchestrator instead of Step Functions or Glue?**  
**A2.**

* Airflow provides visibility, retries, and scheduling in one place.
* We wanted a **single orchestrator** to handle lineage and observability.
* Step Functions would tightly couple logic, while Airflow keeps transforms modular.
* Glue was considered, but Lambda was better for lightweight, versioned transformations.

**Q3. What role does Kafka play here?**  
**A3.**  
Kafka streams provider roster events from the PLM portal. A consumer writes them to Oracle (ROSTER\_RAW). Kafka ensures durability and decoupling between event source and downstream database.

**Q4. Why did you offload transformations to Lambda instead of Airflow tasks?**  
**A4.**

* Lambdas provide **isolation and quick deployment** of transformation logic.
* Easier to scale horizontally — multiple concurrent Lambdas can process files.
* Packaging heavy dependencies (like pandas/pyarrow) is cleaner using Lambda container images.
* Keeps Airflow lightweight and focused on orchestration.

**Data Transformation & Validation**

**Q5. How does the ISF Lambda work?**  
**A5.**

* Reads JSONL from S3.
* Normalizes column names & datatypes.
* Deduplicates by (provider\_id, roster\_date).
* Adds provenance metadata (\_src\_bucket, \_src\_key, job\_id).
* Writes output in Parquet format to ISF bucket.

**Q6. How does the Dart Lambda validate records?**  
**A6.**

* Reads ISF Parquet.
* Runs business validations:
  + provider\_id non-null + regex check
  + roster\_date parseable and within range
  + Required fields present (provider\_id, date, name)
* Writes **valid records** as CSV (Dart bucket).
* Writes **invalid records** to Error bucket with \_validation\_error column.

**Redshift Load**

**Q7. How do you load data into Redshift?**  
**A7.**

* Airflow task uses Redshift COPY command from S3 → staging table.
* Then performs MERGE/UPSERT into final table using job\_id.
* This ensures idempotency and prevents duplicates.

**Q8. How do you handle schema evolution or drift?**  
**A8.**

* Schema versioning is enforced at ISF layer.
* Any new columns are validated; incompatible changes are flagged.
* For larger schema drift, we would migrate via Glue/ECS instead of Lambdas.

**Security & Reliability**

**Q9. What security measures are in place?**  
**A9.**

* Least-privilege IAM roles per component.
* Secrets (Oracle, Redshift) stored in AWS Secrets Manager.
* All S3 buckets use SSE-KMS encryption.
* VPC endpoints for S3/Secrets Manager to avoid public internet.

**Q10. How do you ensure observability and error handling?**  
**A10.**

* Airflow provides DAG-level retries, lineage, and monitoring.
* CloudWatch logs/metrics for Lambdas with alarms on error rate/duration.
* Invalid records always stored in error bucket for replay.
* Alerts via SNS/Slack for pipeline failures.

**Scaling & Challenges**

**Q11. What are the scaling considerations for this pipeline?**  
**A11.**

* If input file size exceeds Lambda’s memory/time → split files upstream or switch to Glue/Spark/EMR.
* Batch vs. streaming tradeoff: frequent small files may increase Lambda cost.
* Maintain partitioned S3 layout (year/month/day) for scalable storage and Redshift COPY.

**Q12. What challenges did you face?**  
**A12.**

1. **Large payloads in Oracle** → solved by streaming with fetchmany() during export.
2. **Lambda dependency limits** → used container images for pandas/pyarrow.
3. **Idempotency** → solved by tagging all outputs with job\_id.
4. **Schema drift** → mitigated via ISF normalization and validation checks.
5. **Redshift MERGE performance** → optimized by batching COPY operations and indexing join keys.

**Scenario-Based**

**Q13. What if Lambdas frequently fail due to large datasets?**  
**A13.**

* Introduce file-splitting logic in Oracle export.
* Or migrate transformation from Lambda → Glue/Spark for distributed processing.

**Q14. How would you replay failed runs?**  
**A14.**

* Since outputs are job\_id partitioned, rerunning the DAG with same job\_id won’t duplicate data.
* Error bucket allows selective reprocessing of invalid rows after fixing issues.

**Q15. How would you extend this pipeline for real-time needs?**  
**A15.**

* Replace Oracle batch export with direct Kafka → Lambda → S3 streaming sink.
* Use Kinesis Firehose or Kafka Connect for near real-time S3 writes.
* Airflow could be replaced with Step Functions for event-driven orchestration.