## Roster Automation Code with explanation

# File: kafka\_consumer/consumer.py

# Kafka -> Oracle consumer

# Requirements: confluent-kafka, oracledb

* A comment explaining this script consumes messages from **Kafka** and inserts them into **Oracle**.
* It requires two libraries:
  + confluent-kafka → to read Kafka messages.
  + oracledb → to connect and insert into Oracle DB.

import os

import json

import logging

import time

from confluent\_kafka import Consumer, KafkaException

import oracledb

from datetime import datetime

* os → access environment variables (like DB credentials).
* json → parse Kafka messages if they are JSON.
* logging → log messages for monitoring/debugging.
* time → pause between loops if needed.
* Consumer, KafkaException → classes from confluent-kafka to interact with Kafka.
* oracledb → Oracle DB driver.
* datetime → to work with timestamps.

logging.basicConfig(level=logging.INFO)

logger = logging.getLogger("kafka\_consumer")

* Sets logging level to INFO (so we see important messages but not debug noise).
* Creates a logger named "kafka\_consumer".

KAFKA\_BROKERS = os.environ.get("KAFKA\_BROKERS", "broker1:9092")

KAFKA\_TOPIC = os.environ.get("KAFKA\_TOPIC", "plm-roster-events")

GROUP\_ID = os.environ.get("KAFKA\_GROUP", "roster-to-oracle")

ORACLE\_DSN = os.environ.get("ORACLE\_DSN") # host:port/service

ORACLE\_USER = os.environ.get("ORACLE\_USER")

ORACLE\_PASSWORD = os.environ.get("ORACLE\_PASSWORD")

BATCH\_SIZE = int(os.environ.get("BATCH\_SIZE", "100"))

* Reads config values from environment variables (so code works in multiple environments without changes).
* KAFKA\_BROKERS: list of Kafka brokers to connect to (default broker1:9092).
* KAFKA\_TOPIC: topic name where roster messages come.
* GROUP\_ID: Kafka consumer group ID (used to track offsets).
* ORACLE\_DSN: Oracle connection string (like host:port/service).
* ORACLE\_USER, ORACLE\_PASSWORD: Oracle credentials.
* BATCH\_SIZE: how many messages to process at once before flushing to DB.

INSERT\_SQL = """

INSERT INTO roster\_raw (id, payload, source, created\_at, export\_status)

VALUES (:id, :payload, :source, SYSTIMESTAMP, NULL)

"""

* SQL query template to insert a Kafka record into roster\_raw table.
* Uses Oracle **bind variables** (:id, :payload, :source) to prevent SQL injection and speed up queries.
* created\_at is auto-filled with system timestamp.
* export\_status is NULL initially (not exported yet).

def get\_consumer():

conf = {

'bootstrap.servers': KAFKA\_BROKERS,

'group.id': GROUP\_ID,

'auto.offset.reset': 'earliest',

'enable.auto.commit': False

}

return Consumer(conf)

* Creates a Kafka **consumer object** with configuration:
  + bootstrap.servers: broker addresses.
  + group.id: consumer group.
  + auto.offset.reset=earliest: if no offset is saved, start from the beginning.
  + enable.auto.commit=False: don’t auto-mark messages as processed (we’ll commit only after writing to DB).
* Returns a Kafka consumer instance.

def get\_oracle\_conn():

return oracledb.connect(user=ORACLE\_USER, password=ORACLE\_PASSWORD, dsn=ORACLE\_DSN, encoding="UTF-8")

* Opens a connection to Oracle DB with credentials and DSN.
* Uses UTF-8 encoding to handle all characters.

def ensure\_sequence\_and\_table(conn):

# Run once in infra to create table and sequence; this is a no-op if already exists.

with conn.cursor() as cur:

try:

cur.execute("""

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

)

""")

conn.commit()

logger.info("Created roster\_raw table")

except Exception:

conn.rollback()

try:

cur.execute("CREATE SEQUENCE roster\_raw\_seq START WITH 1 INCREMENT BY 1")

conn.commit()

logger.info("Created roster\_raw\_seq")

except Exception:

conn.rollback()

* This function ensures the **table (roster\_raw)** and **sequence (roster\_raw\_seq)** exist.
* roster\_raw stores incoming Kafka messages.
* roster\_raw\_seq generates unique IDs.
* Uses try/except → if table/sequence already exists, just rollback and ignore.
* Usually run **once during setup**, not every time.

def run():

consumer = get\_consumer()

conn = get\_oracle\_conn()

# ensure\_sequence\_and\_table(conn) # call offline once if needed

consumer.subscribe([KAFKA\_TOPIC])

buffer = []

* Main function to run the consumer loop.
* Connects to Kafka and Oracle.
* Subscribes to the Kafka topic.
* Prepares an empty buffer to hold messages before inserting in bulk.

try:

while True:

msg = consumer.poll(timeout=1.0)

* Infinite loop to keep consuming messages.
* poll(1.0) waits up to 1 second for a new Kafka message.

if msg is None:

if buffer:

flush\_buffer(buffer, conn)

buffer = []

continue

* If no message arrived (msg=None):
  + If buffer has pending messages, flush them to DB.
  + Then continue waiting.

if msg.error():

raise KafkaException(msg.error())

* If message contains an error, raise a KafkaException.

payload = msg.value().decode("utf-8")

meta = {

"payload": payload,

"source": msg.topic()

}

buffer.append(meta)

* Decode message value from bytes → UTF-8 string.
* Create a dict with message payload + topic name (for provenance).
* Add it to buffer.

if len(buffer) >= BATCH\_SIZE:

flush\_buffer(buffer, conn)

consumer.commit()

buffer = []

* If buffer is full (>= batch size):
  + Flush records to Oracle.
  + Commit Kafka offset (mark messages as processed).
  + Reset buffer.

except KeyboardInterrupt:

logger.info("Shutting down consumer")

* If user stops script (Ctrl+C), log shutdown.

finally:

if buffer:

flush\_buffer(buffer, conn)

consumer.close()

conn.close()

* On shutdown, flush any remaining buffer.
* Close Kafka consumer and Oracle connection.

def flush\_buffer(buffer, conn):

with conn.cursor() as cur:

for rec in buffer:

try:

cur.execute("SELECT roster\_raw\_seq.NEXTVAL FROM dual")

seq = cur.fetchone()[0]

cur.execute(INSERT\_SQL, [seq, rec["payload"], rec["source"]])

except Exception as e:

logger.exception("Insert failed for record: %s", e)

conn.commit()

logger.info("Flushed %d records to Oracle", len(buffer))

* Inserts buffered records into Oracle:
  + Fetches next value from roster\_raw\_seq.
  + Executes insert SQL with seq, payload, source.
  + On error, log but continue.
* Commits all inserts as one transaction.
* Logs how many were flushed.

if \_\_name\_\_ == "\_\_main\_\_":

run()

* Standard Python entrypoint.
* When script is run directly, it executes run() (starts consuming Kafka → Oracle).

✅ That’s **file 1 explained fully line by line**.

Would you like me to move on to the next file (**roster\_export/exporter.py**) and do the same detailed breakdown?

##### You said:

yes

###### ChatGPT said:

Great 👍 Let’s go through **roster\_export/exporter.py** line by line.  
This script takes data from Oracle (roster\_raw table), writes it to **S3** as JSONL files, and updates the rows’ export status.

# File: roster\_export/exporter.py

# Oracle export to S3 as JSONL. Used by Airflow DAG.

# Requirements: oracledb, boto3

* A comment: this script exports Oracle data into **S3** in JSONL format (JSON per line).
* It will be used by the Airflow DAG.
* Needs oracledb (to connect Oracle) and boto3 (to talk to AWS S3).

import os

import json

import io

import math

import logging

from datetime import datetime, timezone

import boto3

import oracledb

* os → read env variables.
* json → convert rows into JSON.
* io → handle in-memory file-like buffers (not heavily used here).
* math → math functions (ceil, etc.).
* logging → log messages.
* datetime, timezone → generate export timestamps.
* boto3 → AWS SDK for S3 operations.
* oracledb → Oracle DB client.

logger = logging.getLogger("roster\_export")

logging.basicConfig(level=logging.INFO)

* Creates a logger named "roster\_export".
* Sets logging level to INFO (only important logs).

ORACLE\_DSN = os.environ.get("ORACLE\_DSN")

ORACLE\_USER = os.environ.get("ORACLE\_USER")

ORACLE\_PASSWORD = os.environ.get("ORACLE\_PASSWORD")

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

S3\_PREFIX = os.environ.get("S3\_RAW\_PREFIX", "year=%Y/month=%m/day=%d/")

MAX\_ROWS\_PER\_FILE = int(os.environ.get("MAX\_ROWS\_PER\_FILE", "20000"))

EXPORT\_LEDGER\_TABLE = os.environ.get("EXPORT\_LEDGER\_TABLE", "roster\_export\_runs")

* Reads configs from environment variables:
  + ORACLE\_\* → credentials.
  + S3\_RAW\_BUCKET → where exported files are stored.
  + S3\_PREFIX → folder structure in S3, defaults to year/month/day.
  + MAX\_ROWS\_PER\_FILE → split big exports into multiple files.
  + EXPORT\_LEDGER\_TABLE → optional table to record export history.

s3 = boto3.client("s3")

* Creates an S3 client to read/write objects.

def get\_oracle\_conn():

return oracledb.connect(user=ORACLE\_USER, password=ORACLE\_PASSWORD, dsn=ORACLE\_DSN, encoding="UTF-8")

* Opens a connection to Oracle DB using credentials from env.
* Ensures UTF-8 encoding.

def export\_to\_s3(run\_id=None):

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

conn = get\_oracle\_conn()

s3\_keys = []

* Main export function.
* If run\_id is not passed, generate one like airflow-20250913T101234.
* Opens Oracle connection.
* Prepares empty list s3\_keys to collect exported file paths.

try:

cur = conn.cursor()

# Query unexported rows; use FOR UPDATE to avoid double processing if running concurrently

sql = "SELECT id, payload FROM roster\_raw WHERE export\_status IS NULL ORDER BY created\_at"

cur.execute(sql)

batch = cur.fetchmany(MAX\_ROWS\_PER\_FILE)

file\_index = 0

* Creates a cursor.
* SQL selects rows that have **not been exported yet** (export\_status IS NULL).
* Orders them by creation time.
* Fetches rows in chunks of MAX\_ROWS\_PER\_FILE.
* Sets file\_index=0 to name files sequentially.

while batch:

rows = []

ids\_in\_batch = []

for r in batch:

ids\_in\_batch.append(r[0])

payload = r[1]

# if payload is already JSON string, ensure it's valid; else wrap

try:

obj = json.loads(payload)

except Exception:

obj = {"\_raw": payload}

# add provenance columns

obj["\_db\_id"] = r[0]

obj["\_export\_run\_id"] = run\_id

rows.append(obj)

* Loops while rows exist.
* For each row:
  + Save id into ids\_in\_batch.
  + Take payload (the Kafka message).
  + Try to parse it as JSON. If not valid JSON, wrap it as {"\_raw": payload}.
  + Add metadata: \_db\_id (row ID in Oracle) and \_export\_run\_id (this run).
  + Append to rows.

# write JSONL content

jsonl = "\n".join(json.dumps(r, default=str) for r in rows)

now = datetime.utcnow()

prefix = now.strftime(S3\_PREFIX)

filename = f"export\_{run\_id}--part{file\_index}.jsonl"

key = f"{prefix}{filename}"

s3.put\_object(Bucket=S3\_RAW\_BUCKET, Key=key, Body=jsonl.encode("utf-8"))

s3\_keys.append(key)

logger.info("Wrote %d records to s3://%s/%s", len(rows), S3\_RAW\_BUCKET, key)

* Converts rows into **JSONL format** (one JSON per line).
* Builds an S3 key using prefix (year/month/day) + filename (export\_runid--partX.jsonl).
* Uploads JSONL string to S3 (put\_object).
* Saves the key into s3\_keys.
* Logs success.

# update export\_status for ids

update\_sql = "UPDATE roster\_raw SET export\_status='EXPORTED', export\_at = SYSTIMESTAMP WHERE id = :id"

for i in ids\_in\_batch:

cur.execute(update\_sql, [i])

conn.commit()

* After writing to S3, marks those rows as EXPORTED in Oracle.
* Also sets export\_at to current timestamp.
* Commits transaction.

file\_index += 1

batch = cur.fetchmany(MAX\_ROWS\_PER\_FILE)

* Increments file counter.
* Fetches next batch of rows.

finally:

conn.close()

* Ensures Oracle connection is closed (even if errors happen).

# Optionally record run in ledger table (implement if you want)

return {"s3\_keys": s3\_keys, "run\_id": run\_id}

* Returns list of exported S3 keys and run ID.
* (Optionally, this could also insert a record into an export ledger table for tracking).

✅ That’s the **Oracle → S3 Exporter** explained line by line.

Would you like me to go next into the **ISF Lambda (lambdas/isf\_transform/isf\_transform\_lambda.py)** and explain it the same way?

##### You said:

yes

###### ChatGPT said:

Awesome 👍 Let’s now do a **line-by-line breakdown** of the **ISF Lambda (lambdas/isf\_transform/isf\_transform\_lambda.py)**.

This Lambda takes **raw JSONL files from S3** and transforms them into **normalized Parquet files** (ISF format).

# File: isf\_transform\_lambda.py

# isf\_transform\_lambda.py

# Normalizes JSONL -> Parquet

# For heavy deps use Lambda container image or layer (pandas + pyarrow)

* Comment: this script transforms JSONL into **Parquet** (efficient columnar format).
* Since pandas and pyarrow are heavy, it suggests running Lambda as a **container image** or with **Lambda layers**.

import os

import io

import json

import math

import boto3

import pandas as pd

from datetime import datetime, timezone

* os → get environment variables.
* io → create in-memory file buffers for Parquet writing.
* json → parse JSON lines.
* math → calculate chunks for splitting.
* boto3 → AWS SDK to read/write S3 files.
* pandas → data wrangling (normalize + deduplicate).
* datetime, timezone → generate job IDs & timestamps.

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"))

* Creates an S3 client.
* RAW\_BUCKET: where input JSONL files live.
* ISF\_BUCKET: where output Parquet files will be written.
* ISF\_PREFIX: folder prefix for ISF outputs (default "isf/").
* MAX\_ROWS\_PER\_CHUNK: maximum number of rows per Parquet file (prevents very large files from exceeding Lambda limits).

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}

* Reads an S3 object (bucket + key).
* Decodes it as UTF-8 text.
* Splits into lines.
* For each non-empty line:
  + Tries to parse it as JSON.
  + If parsing fails, wraps it in a dict {"\_raw": line}.
* yield → returns one record at a time (generator).

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

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

* Normalizes column names:
  + Strip spaces, lowercase, replace spaces with underscores.
  + Example: "Roster Date " → "roster\_date".

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

df["provider\_id"] = None

* Ensures the column "provider\_id" always exists, even if missing in input.

# unify date fields: look for common candidates

for candidate in ("date", "roster\_date", "rosterdate"):

if candidate in df.columns:

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

break

* Checks for common date column names.
* If found, converts it into a **standardized roster\_date column**.
* Uses pd.to\_datetime(..., errors="coerce") → invalid dates become NaT.
* Removes timezone info (tz\_localize(None)).

# if roster\_date missing, leave as NaT

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

if dedupe\_cols:

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

return df

* If we have both provider\_id and roster\_date, use them to **remove duplicates**, keeping only the latest row.
* Returns the cleaned dataframe.

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

buf = io.BytesIO()

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

return buf.getvalue()

* Converts a DataFrame into Parquet format in memory.
* Writes to BytesIO buffer.
* Returns raw bytes (to upload to S3).

def handler(event, context):

"""

event = {"s3\_keys":["raw/.../export\_a.jsonl"], "job\_id":"..."}

"""

* Lambda entry function (handler).
* Input event contains:
  + "s3\_keys": list of JSONL file paths in raw bucket.
  + "job\_id": unique job run ID.

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

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

produced = []

errors = []

processed = 0

* Extracts s3\_keys from event.
* If no job\_id provided, generate one from current timestamp.
* Initialize:
  + produced = list of output files written.
  + errors = list of errors per file.
  + processed = total rows processed.

for key in keys:

try:

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

if not rows:

continue

* Loops through input S3 keys.
* Reads all rows using read\_jsonl\_s3.
* If file is empty, skip it.

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)

* Splits rows into smaller chunks (if file too big).
* Converts chunk into DataFrame.
* Normalizes DataFrame (column names, provider\_id, roster\_date, dedupe).

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

df["\_src\_key"] = key

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

* Adds metadata columns:
  + \_src\_bucket: raw S3 bucket name.
  + \_src\_key: file key in raw bucket.
  + \_ingested\_at: timestamp when processing happened.

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

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

* Extracts input filename (without extension).
* Builds output S3 key in ISF bucket:
  + Includes prefix, job\_id, original filename, and part number.
  + Example: isf/job-123/export\_abc--job-123--part0.parquet.

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

produced.append(out\_key)

processed += len(df)

* Converts DataFrame to Parquet bytes.
* Uploads to ISF bucket with new key.
* Saves output key to produced.
* Adds row count to processed.

except Exception as e:

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

* If processing fails for this key, record the error.

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

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

* If no errors → status = "OK", else "ERROR".
* Returns summary:
  + isf\_keys = output Parquet files.
  + processed\_count = rows processed.
  + errors = any failures.

✅ That’s the **ISF Lambda explained line by line**.

Do you want me to go next with the **DART Lambda (lambdas/dart\_transform/dart\_transform\_lambda.py)**?

##### You said:

yes

###### ChatGPT said:

Perfect 👍 Let’s now go through the **DART Lambda (lambdas/dart\_transform/dart\_transform\_lambda.py)** line by line.

This Lambda takes **ISF Parquet files** (cleaned data) and validates them.

* ✅ Valid rows → written to **DART bucket** as CSV (for Redshift).
* ❌ Invalid rows → written to **Error bucket** as Parquet.

# File: dart\_transform\_lambda.py

# dart\_transform\_lambda.py

# Validate ISF Parquet -> CSV(s) for Redshift + write invalid rows to error bucket

* Comment explains the Lambda’s purpose:
  + Input = ISF Parquet files.
  + Output = CSV files (for Redshift) + Error files (for rejected rows).

import os

import io

import json

import boto3

import pandas as pd

from datetime import datetime, timezone

* os → get env variables.
* io → handle in-memory buffers for Parquet writing.
* json → format errors if needed.
* boto3 → AWS SDK (read/write S3).
* pandas → handle DataFrame validation and CSV writing.
* datetime, timezone → generate job IDs.

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/")

* Creates S3 client.
* ISF\_BUCKET: where input files (Parquet) live.
* DART\_BUCKET: where valid CSVs will go.
* ERROR\_BUCKET: where invalid rows will be written.
* DART\_PREFIX: folder prefix inside DART bucket (default dart/).
* ERROR\_PREFIX: folder prefix inside error bucket (default errors/).

REQUIRED = ["provider\_id", "roster\_date", "name"]

* Defines **mandatory columns** every row must have.
* If missing/invalid, the row goes to **errors**.

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

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

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

* Reads a Parquet file from S3.
* Downloads as bytes, wraps with BytesIO, then loads into a pandas DataFrame.

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

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

* Uploads CSV **string** (text) to S3 as an object.

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())

* Converts a DataFrame to Parquet in-memory.
* Uploads it to S3.
* Used for writing **invalid rows**.

def validate(df: pd.DataFrame):

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

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

* Adds a new column \_validation\_error to track validation failures.
* Creates valid\_mask (boolean series) initially marking all rows as valid (True).

for col in REQUIRED:

if col not in df.columns:

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

valid\_mask &= False

* For each required column:
  + If missing in DataFrame → mark error missing:<col> for all rows.
  + Mark those rows as invalid (valid\_mask=False).

else:

if col == "roster\_date":

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

bad = parsed.isna()

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

valid\_mask[bad] = False

* If roster\_date exists:
  + Try to parse as datetime.
  + If parsing fails → mark "bad\_date".
  + Those rows become invalid.

if col == "provider\_id":

mask = df["provider\_id"].astype(str).str.match(r"^[A-Z0-9\_-]{5,20}$", na=False)

bad = ~mask

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

valid\_mask[bad] = False

* If provider\_id exists:
  + Convert to string, check regex pattern:
    - Must be 5–20 chars, only A-Z, 0-9, \_, or -.
  + If it doesn’t match → mark "bad\_provider\_id".
  + Mark row invalid.

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

* Splits DataFrame into two:
  + ✅ Valid rows (passed validation).
  + ❌ Invalid rows (failed validation).

def handler(event, context):

"""

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

"""

* Lambda entry function.
* Event contains:
  + "isf\_keys": list of input ISF Parquet files.
  + "job\_id": unique run 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

* Extracts isf\_keys and job\_id.
* Prepares:
  + dart\_keys = output valid CSVs.
  + error\_keys = error files.
  + Counters for total valid and invalid rows.

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', '')

* Loops through each ISF file.
* Reads it from S3 into DataFrame.
* Runs validation → splits into valid and invalid sets.
* Extracts a base\_name (filename without .parquet).

if not valid\_df.empty:

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

"provider\_id": "providerId",

"roster\_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 there are valid rows:
  + Rename columns into **camelCase** for Redshift.
  + Convert to CSV string.
  + Save to DART bucket with key format:
    - dart/<job\_id>/<filename>--<job\_id>.csv.
  + Record output key in dart\_keys.
  + Increment total\_valid counter.

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)

* If invalid rows exist:
  + Save them into error bucket as Parquet file.
  + Key format: errors/<job\_id>/<filename>--<job\_id>.parquet.
  + Add to error\_keys.
  + Increment total\_invalid.

except Exception as e:

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

* If the whole processing of a file fails, record the error.

return {

"status": "OK",

"dart\_keys": dart\_keys,

"error\_keys": error\_keys,

"total\_valid": total\_valid,

"total\_invalid": total\_invalid

}

* Returns a summary:
  + status: "OK" even if there were invalid rows (they’re handled).
  + dart\_keys: list of output CSV files.
  + error\_keys: list of error files.
  + total\_valid, total\_invalid: counts.

✅ That’s the **DART Lambda fully explained**.

Next up, do you want me to explain the **Airflow DAG (dags/full\_roster\_pipeline\_lambda.py)** line by line? That’s where all these pieces get orchestrated together.

##### You said:

yes

###### ChatGPT said:

Perfect 👍 Let’s now do the **Airflow DAG (dags/full\_roster\_pipeline\_lambda.py)** line by line.

This DAG orchestrates the entire pipeline:

1. Export raw rows from Oracle → S3 (JSONL).
2. Invoke **ISF Lambda** → normalize + Parquet.
3. Invoke **DART Lambda** → validate + CSV + errors.
4. Load CSVs into Redshift.

# File: full\_roster\_pipeline\_lambda.py

# Airflow DAG orchestrating the pipeline

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

* Imports:
  + datetime, timedelta: to schedule DAG runs.
  + DAG: Airflow DAG object.
  + PythonOperator: lets us run Python functions as Airflow tasks.
  + AwsLambdaHook: to invoke AWS Lambda functions from Airflow.
  + os: read environment variables.
  + json: format payloads for Lambda.

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")

* Environment configs:
  + S3\_RAW\_BUCKET: where raw JSONL exports go.
  + ISF\_LAMBDA\_NAME: name of ISF Lambda function.
  + DART\_LAMBDA\_NAME: name of DART Lambda function.
  + DART\_BUCKET: S3 bucket for valid CSV outputs.
  + REDSHIFT\_CONN\_ID: Airflow connection ID for Redshift.
  + AWS\_CONN\_ID: Airflow connection ID for AWS credentials.

default\_args = {

"owner": "data-eng",

"depends\_on\_past": False,

"email\_on\_failure": True,

"retries": 2,

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

}

* Default task settings:
  + owner: DAG owner (data engineering team).
  + depends\_on\_past=False: don’t block runs if a previous run failed.
  + email\_on\_failure=True: send email when a task fails.
  + retries=2: retry twice if a task fails.
  + retry\_delay=5 min: wait 5 minutes before retrying.

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:

* Defines the DAG:
  + dag\_id: unique ID for the DAG.
  + start\_date: DAG starts from Jan 1, 2025.
  + schedule\_interval="0 \* \* \* \*": run every hour on the hour.
  + catchup=False: don’t backfill past runs.
  + max\_active\_runs=1: only one run at a time.

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

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

from roster\_export.exporter import export\_to\_s3

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

* First task: export raw rows from Oracle to S3.
* Generates a unique run\_id (timestamped).
* Imports and calls export\_to\_s3.
* Pushes results into **XCom**:
  + raw\_keys: list of S3 files created.
  + run\_id: job identifier.
* Returns result (for logging).

export\_task = PythonOperator(

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

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

provide\_context=True,

)

* Defines an Airflow task called export\_oracle\_to\_s3.
* Runs the Python function above.
* provide\_context=True: passes task instance (ti) and context to the function.

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

* Utility function to invoke a Lambda synchronously.
* AwsLambdaHook handles the AWS Lambda call.
* Payload is JSON-encoded.
* If Lambda returns bytes, decode to string.
* Try to parse JSON response, else return raw string.

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

* Second task: call ISF Lambda.
* Pulls raw\_keys and run\_id from XCom (set by export task).
* If no files → return "NO\_FILES".
* Otherwise, build payload (s3\_keys, job\_id) and invoke ISF Lambda.
* Push ISF response to XCom (isf\_response).

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

* Defines Airflow task for invoking ISF Lambda.

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

* Third task: call DART Lambda.
* Pulls ISF Lambda response from XCom.
* Extracts isf\_keys.
* If no ISF files → return "NO\_ISF".
* Otherwise, call DART Lambda with payload.
* Push DART response into XCom (dart\_response).

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

* Defines Airflow task for invoking DART Lambda.

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"}

* Fourth task: load validated CSV files into Redshift.
* Pulls DART response from XCom.
* Extracts dart\_keys.
* If no files, return "NO\_DART".

import psycopg2

from airflow.hooks.base import BaseHook

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()

* Imports Postgres driver psycopg2.
* Gets Redshift connection from Airflow.
* Builds connection string.
* Opens connection to Redshift.
* Creates a cursor.

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")

* Reads environment variables for Redshift schema/table.
* IAM role is needed for Redshift COPY from S3.

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()

* For each CSV file in DART:
  + Build S3 path.
  + Construct Redshift COPY command.
    - CSV: since DART files are CSV.
    - IGNOREHEADER 1: skip header row.
    - TIMEFORMAT 'auto': auto-parse dates.
  + Execute COPY → load data into Redshift staging table.
  + Commit.

cur.close()

pg.close()

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

* Close Redshift cursor and connection.
* Return success status with list of loaded files.

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

* Defines Airflow task to load Redshift.

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

* Defines task dependencies:
  1. export\_task →
  2. isf\_invoke →
  3. dart\_invoke →
  4. load\_task.

This enforces the full pipeline order. ✅

That’s the **Airflow DAG fully explained line by line** 🎯.

Do you want me to continue with the **Redshift SQL merge/upsert (redshift/merge\_upsert.sql)** explanation next?

##### You said:

yes

###### ChatGPT said:

Perfect 👍 Let’s now break down the **Redshift SQL merge script (redshift/merge\_upsert.sql)** line by line.

This SQL handles **upserting**:

* Removes old versions of rows from **final table**.
* Inserts fresh data from the **staging table**.

# File: merge\_upsert.sql

-- Example MERGE/upsert in Redshift

-- Run after staging table has been COPY'ed with \_job\_id and \_source\_file

* Comment: This is a sample upsert script for Redshift.
* Assumes you’ve already loaded raw CSVs into a **staging table** (roster\_staging) via COPY.
* Staging has metadata like \_job\_id and \_source\_file.

BEGIN;

* Starts a **transaction**.
* Ensures delete + insert happen together.
* If something fails, you can rollback.

-- delete existing rows for this run\_id (idempotency)

DELETE FROM public.roster\_final f

USING public.roster\_staging s

WHERE f.providerid = s.providerid

AND s.\_job\_id = :job\_id;

* First, **delete old rows** in the final table that match new staging rows.
* public.roster\_final = permanent target table.
* public.roster\_staging = temporary table with new data.
* USING allows joining staging to final.
* Condition: same providerid and same \_job\_id.
* This ensures:
  + If a provider’s roster row already exists, it’s removed before inserting the new one.
  + If job is re-run, old rows from that job\_id are cleaned up (idempotency).

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

SELECT providerid, rosterdate, providername, CURRENT\_TIMESTAMP

FROM public.roster\_staging

WHERE \_job\_id = :job\_id;

* Next, **insert fresh rows** from staging into final.
* Copies only rows with the current job\_id.
* Adds \_last\_updated = CURRENT\_TIMESTAMP to mark load time.

COMMIT;

* Ends transaction and saves changes.
* Now the final table has clean, up-to-date records.

⚡ Key points about this SQL:

1. It’s not a true MERGE (since Amazon Redshift doesn’t support ANSI MERGE natively) → implemented with DELETE + INSERT.
2. The :job\_id parameter must be replaced at runtime (Airflow or a script).
3. Staging table is usually truncated after the merge.
4. This ensures **deduplication** and **idempotency** (safe re-runs).

✅ That’s the **Redshift merge script fully explained**.

Would you like me to also break down the **Terraform snippet (infra/s3\_and\_iam.tf)** line by line next?

##### You said:

yes

###### ChatGPT said:

Awesome 👍 Let’s now go line by line through the **Terraform snippet (infra/s3\_and\_iam.tf)**.

This code provisions **S3 buckets** (for raw data, etc.) and an **IAM role** (for Lambda functions).

# File: infra/s3\_and\_iam.tf

# minimal Terraform snippet (not complete) - create S3 buckets & IAM role for Lambdas

* A comment: this file is a **partial example**, not the full infra.
* It sets up S3 buckets and IAM permissions for Lambdas.

resource "aws\_s3\_bucket" "plm\_raw" {

bucket = "plm-raw-bucket"

* Creates an S3 bucket resource named plm\_raw.
* The bucket name is "plm-raw-bucket".
* This will hold **raw exported JSONL files**.

server\_side\_encryption\_configuration {

rule {

apply\_server\_side\_encryption\_by\_default {

sse\_algorithm = "aws:kms"

}

}

}

}

* Enables **default encryption** for all objects in the bucket.
* Uses AWS KMS-managed keys (sse\_algorithm = "aws:kms").
* Ensures compliance/security by encrypting data at rest.

resource "aws\_iam\_role" "lambda\_role" {

name = "roster\_lambda\_role"

assume\_role\_policy = data.aws\_iam\_policy\_document.lambda\_assume.json

}

* Creates an **IAM role** for Lambda functions.
* Role name: roster\_lambda\_role.
* The assume\_role\_policy defines who can assume this role (Lambdas).

data "aws\_iam\_policy\_document" "lambda\_assume" {

statement {

actions = ["sts:AssumeRole"]

principals {

type = "Service"

identifiers = ["lambda.amazonaws.com"]

}

}

}

* Defines a policy document (lambda\_assume) that allows **Lambda service** to assume the role.
* Action: "sts:AssumeRole".
* Principal: "lambda.amazonaws.com" → AWS Lambda runtime.
* This is the standard **trust policy** for Lambda roles.

resource "aws\_iam\_role\_policy" "lambda\_policy" {

name = "roster\_lambda\_policy"

role = aws\_iam\_role.lambda\_role.id

* Creates an **inline policy** attached to the Lambda role.
* Policy name: roster\_lambda\_policy.
* Attached to role lambda\_role.

policy = jsonencode({

Version = "2012-10-17",

Statement = [

{

Action = [

"s3:GetObject",

"s3:PutObject",

"s3:ListBucket"

],

Effect = "Allow",

Resource = [

aws\_s3\_bucket.plm\_raw.arn,

"${aws\_s3\_bucket.plm\_raw.arn}/\*",

# add other buckets...

]

},

* Policy statement:
  + Allows Lambda to GetObject, PutObject, ListBucket on S3.
  + Resources:
    - plm\_raw bucket itself.
    - All objects inside that bucket (.../\*).
  + Comment reminds you to add other buckets (ISF, DART, Error).

{

Action = ["logs:CreateLogGroup","logs:CreateLogStream","logs:PutLogEvents"],

Effect = "Allow",

Resource = "\*"

}

]

})

}

* Second policy statement:
  + Grants permissions to **CloudWatch Logs**.
  + Required for Lambda to create log groups, streams, and push logs.
  + Resource="\*" → all logs allowed (normal for Lambda).

✅ That’s the **Terraform snippet fully explained**.

Next, do you want me to explain the **Lambda Dockerfile (lambdas/Dockerfile)** line by line?

##### You said:

yes

###### ChatGPT said:

Great 👍 Let’s now go through the **Lambda Dockerfile (lambdas/Dockerfile)** line by line.

This Dockerfile builds a container image for AWS Lambda functions (like ISF and DART), which need heavy Python libraries (pandas, pyarrow).

# File: lambdas/Dockerfile

FROM public.ecr.aws/lambda/python:3.10

* Uses AWS’s **official Lambda base image** for Python 3.10.
* This image has the Lambda Runtime API preinstalled, so AWS can run it as a Lambda container.

# Install system deps (if needed)

RUN yum -y install gcc openssl-devel libffi-devel

* Installs extra **system libraries** that Python packages might need:
  + gcc → C compiler (needed for compiling some Python libs).
  + openssl-devel → cryptography support.
  + libffi-devel → foreign function interface library (used by Pandas, cryptography, etc.).

# Copy requirements and install

COPY requirements.txt /tmp/requirements.txt

RUN pip install -r /tmp/requirements.txt

* Copies requirements.txt into the image.
* Installs all dependencies listed in it (e.g. pandas, pyarrow, boto3, oracledb).
* These will be available for the Lambda functions.

# Copy function code (assumes handler is isf\_transform\_lambda.handler or dart\_transform\_lambda.handler)

COPY isf\_transform\_lambda.py ${LAMBDA\_TASK\_ROOT}

COPY dart\_transform\_lambda.py ${LAMBDA\_TASK\_ROOT}

* Copies the Lambda function Python files into the container.
* ${LAMBDA\_TASK\_ROOT} is the default working directory where Lambda expects handler code.
* Both ISF and DART functions are included here, so you can switch entrypoint at deploy time.

# Set entrypoint for AWS Lambda

CMD ["isf\_transform\_lambda.handler"]

* Defines the **default Lambda handler**.
* Format: ["<filename>.<function>"].
* Here: isf\_transform\_lambda.py → function handler.
* If you want to deploy the **DART Lambda**, you would change this to:
* CMD ["dart\_transform\_lambda.handler"]

✅ That’s the **Dockerfile fully explained**.

Would you like me to also explain the **GitHub Actions CI file (.github/workflows/ci.yml)** line by line next?

##### You said:

yes

###### ChatGPT said:

Perfect 👍 Let’s now break down the **GitHub Actions CI workflow (.github/workflows/ci.yml)** line by line.

This workflow builds and pushes the **Lambda container image** to Amazon ECR whenever code is pushed to the main branch.

# File: .github/workflows/ci.yml

name: CI

* Names the workflow **CI** (Continuous Integration).
* This label will show up in the GitHub Actions UI.

on:

push:

branches: [ main ]

* Defines the trigger event.
* Run this workflow **on push to the main branch**.
* So whenever you merge or push directly to main, the CI pipeline runs.

jobs:

build-and-push:

runs-on: ubuntu-latest

* Defines a single job called build-and-push.
* It runs on a GitHub-hosted **Ubuntu runner** (ubuntu-latest).

steps:

- uses: actions/checkout@v4

* First step: **checkout** the repo code so it’s available inside the runner.
* Uses GitHub’s official actions/checkout action, version v4.

- name: Login to ECR

uses: aws-actions/amazon-ecr-login@v1

* Step 2: **login to Amazon ECR** (Elastic Container Registry).
* Uses AWS’s official GitHub Action.
* This lets Docker push images to your AWS ECR repository.
* Needs AWS credentials (usually provided via GitHub secrets).

- name: Build Docker image

run: |

docker build -t roster-lambda:latest lambdas/

* Step 3: **build the Docker image**.
* docker build → builds from lambdas/Dockerfile.
* Tags the image as roster-lambda:latest.
* Context = lambdas/ directory.

- name: Tag & push

run: |

IMAGE\_URI=${{ secrets.ECR\_REGISTRY }}/roster-lambda:latest

docker tag roster-lambda:latest $IMAGE\_URI

docker push $IMAGE\_URI

* Step 4: **tag and push the image to ECR**.
* IMAGE\_URI is built from a secret ECR\_REGISTRY (e.g., 123456789012.dkr.ecr.us-east-1.amazonaws.com).
* docker tag → re-tags the local image with full ECR URI.
* docker push → pushes it to AWS ECR so Lambda can use it.

✅ That’s the **GitHub Actions CI workflow fully explained**.

So far we’ve covered every main code file:

* Kafka → Oracle consumer ✅
* Oracle exporter ✅
* ISF Lambda ✅
* DART Lambda ✅
* Airflow DAG ✅
* Redshift merge SQL ✅
* Terraform IAM + S3 ✅
* Lambda Dockerfile ✅
* GitHub Actions CI ✅

Bottom of Form