**End-to-End Data Pipeline Plan for Telecom XML Processing**

# Overview

We are building a scalable pipeline to process millions of XML files from an S3 bucket in a telecom environment. There are four architectural options:

## ****Scenario 1: Parse XML → Load to Deep Tables → Transform the parsed data -> load to OM Tables****

### ****Pipeline Flow****

Get raw XML in batches(s3)

|

v

[Parser Pods (K8s)]

|

v

[ Flat JSON Output to (Deep Tables)]

|

v

[Data Transformer Pods (K8s)]

|

v

[Final Transformed Tables]

### ****Technologies Used****

**AWS S3**: Raw file storage

**Python (xml.etree.ElementTree)**: XML parsing

**Kubernetes**: Parallel processing

**Docker**: Environment packaging

**DB**: Files metadata tracking

**Grafana** : Metrics

## ****Scenario 2: Parse XML → Directly Transform and Load Final Tables (No Deep Tables)****

### ****Pipeline Flow****

S3 (raw XML)

|

v

[Parser + Transformer Pods]

|

v

[Final Transformed Output (DB / S3)]

### ****Pros****

Lower latency

No intermediate storage

### ****Cons****

Less flexible (no re-processing possible)

All logic must succeed in one run

### ****Implementation Differences****

Single pod handles parsing + transformation

Decoder injected via ConfigMap at startup (Option 2)

Output directly written to final tables

## ****Scenario 3: Parse XML → Store JSON on S3 → Later Transform to Final Tables (Split Stages, No Deep Tables)****

### ****Pipeline Flow****

S3 (raw XML)

|

v

[Parser Pods]

|

v

S3 JSON Output (Raw Flat JSON)

|

v

[Transformer Pods]

|

v

[Final Output]

### ****Advantages****

Separation of concerns

Easier error recovery

### ****Disadvantages****

Slightly more storage

Slightly higher latency

## ****Scenario 4: Parse XML → Transform Using Spark or WASM → Load Final Output (No Deep Tables)****

### ****Pipeline Flow****

S3 (raw XML)

|

v

[Parser Pods → JSON]

|

v

[Spark/WASM Transformer Pods → Final Tables]

### ****Notes****

Use PySpark if data volumes are huge and processing is intensive

Or embed Python into WebAssembly (WASM) modules for edge execution

PySpark jobs can be triggered using Airflow, Argo, or native K8s CronJobs

from pyspark.sql import SparkSession

spark = SparkSession.builder.appName("transform").getOrCreate()

data = spark.read.json("s3a://bucket/json/")

transformed = data.withColumn("network\_key", ...) # transformation logic

transformed.write.mode("append").parquet("s3a://final-bucket/")

### ****Implementation Plan****

**Load Decoder to ConfigMap**kubectl create configmap decoder-config --from-file=decoder.json  
  
and then use the Config map (decoder data) in each K8s pod as a cache memory which will make it super fast.

**Dockerfile**

FROM python:3.10

WORKDIR /app

COPY requirements.txt ./

RUN pip install -r requirements.txt

COPY . .

CMD ["python", "parse.py"]

**Kubernetes Job YAML**

apiVersion: batch/v1

kind: Job

metadata:

name: parse-job

spec:

template:

spec:

containers:

- name: parser

image: your-docker-image

env:

- name: FILE\_LIST

value: "file1.xml,file2.xml"

restartPolicy: Never

backoffLimit: 4

1. **Fetch Files from S3 and Update Metadata Table**

List files from S3 using boto3

Check file names against metadata DB

Mark new files as PENDING

Assign each batch to a new K8s job

Mark file status during lifecycle

**Below Code to fetch file from s3 and updating meta table needs to be added in parsing code.**

import boto3

import psycopg2

# Load decoder

decoder = load\_decoder()

# S3 setup

s3 = boto3.resource(

's3',

endpoint\_url='http://s3-txslmemsql.nss.vzwnet.com:80',

aws\_access\_key\_id='49b1d345d4ca622d4f48',

aws\_secret\_access\_key='BPAIHmNWF4iH/VhDbc6HbjCGCshBFByQwpWQfzEc'

)

bucket\_name = 'slcldn1-xlpt-dev-data'

bucket = s3.Bucket(bucket\_name)

# Connect to metadata DB

conn = psycopg2.connect("dbname=telecom user=etl password=secret")

cur = conn.cursor()

def update\_filedata\_table():

existing\_files = set()

cur.execute("SELECT filename FROM filedata")

for row in cur.fetchall():

existing\_files.add(row[0])

for obj in bucket.objects.all():

filename = obj.key.split('/')[-1]

if filename not in existing\_files:

print(f"New file: {filename}")

cur.execute(

"INSERT INTO filedata (filename, status) VALUES (%s, %s)",

(filename, 'PENDING')

)

conn.commit()

update\_filedata\_table()

## ****Parallel Pod Scaling Strategy (500 Pods for S3 Parallel Processing)****

To run **500 parallel pods**, each fetching one file from S3:

### ****Approach: Kubernetes Job Array****

**Split your S3 file list into 500 file-specific tasks**.

Use a Kubernetes **Job with** .spec.parallelism = 500 and pass each pod a unique file to process via **environment variables or config**.

#### ****Example Job YAML (with Indexed Jobs)****

yaml

CopyEdit

apiVersion: batch/v1kind: Jobmetadata:

name: parse-jobspec:

parallelism: 500

completions: 500

completionMode: Indexed

template:

spec:

containers:

- name: parser

image: your-docker-image

env:

- name: JOB\_INDEX

valueFrom:

fieldRef:

fieldPath: metadata.annotations['batch.kubernetes.io/job-completion-index']

- name: S3\_FILE\_LIST

valueFrom:

configMapKeyRef:

name: file-list-cm

key: files.json

restartPolicy: Never

In your code:

python

CopyEdit

import os, json

index = int(os.environ.get("JOB\_INDEX", 0))with open("/app/files.json") as f:

file\_list = json.load(f)

target\_file = file\_list[index]

process\_file(target\_file)

### ****Steps to Set This Up****

List all S3 files using boto3.

Save the list as files.json and upload it as a ConfigMap (file-list-cm).

Deploy a Kubernetes **Indexed Job** with completions: 500.

Each pod reads the file at its own index using JOB\_INDEX.

### ****Benefits****

Massive parallelism

Isolated retries (each pod only retries its own file)

Easy to monitor and scale

Avoids DB contention if S3 metadata is pre-assigned

**Grafana Metrics**

Metrics exporter inside container (e.g., Prometheus Python client)

Track:

Files parsed

Parse time

Errors per hour

## ****Technology Comparison and Roles****

| **Tech** | **Role** | **Better Alternatives** |
| --- | --- | --- |
| S3 | Scalable file storage | GCS (if on GCP) |
| Python ET | XML Parsing | lxml (faster), Spark-XML (for huge) |
| Kubernetes | Orchestration & auto-scaling | AWS Batch (managed) |
| PostgreSQL | File tracking metadata | DynamoDB (if no joins needed) |
| Docker | Containerization | - |
| Prometheus | Metrics scraping | CloudWatch + Grafana |
| Grafana | Dashboard + alerts | Datadog, Kibana |
| PySpark | Fast parallel transformation | Dask, WASM (alternative) |

## ****Recommendation Summary****

| **Scenario** | **When to Use** |
| --- | --- |
| Scenario 1 | Complex workflows, ad hoc reprocessing |
| Scenario 2 | Speed-focused, less complexity |
| Scenario 3 | Mid-ground: scalable + separable |
| Scenario 4 | High throughput and alternative engines |