Markdown

# 📘 PROFESSIONAL TEST PLAN

## Parserless XML to DB Automation – XPT Project

### 📌 Objective

To identify and implement the most efficient, scalable, and fault-tolerant method for automated ingestion of large volumes of XML files from an S3-like folder into a relational database, without using the legacy Java parser.

**\*\*Key use case:\*\***

\* XML files (each ~4MB) arrive every 15 minutes in bulk\* Lakhs of historical XMLs already present in storage\* Use of Python-based ET parser is mandatory

### 🧰 Current Setup Overview

| Component | Description |

| :---------------- | :-------------------------------------------------------------------------- |

| 🛠 Project | XPT (Verizon Telecom) |

| 📥 Input | XML files (~4MB) arriving in bulk every 15 min, already lakhs stored |

| 🎯 Output | Structured data inserted into a relational DB |

| ⚙️ Parser Tool | Python ET parser (takes 3s parse + 7s DB insert per file) |

| 💾 Storage | S3-compatible storage / folder |

| 🖥 Infrastructure| Multi-core Linux servers, cron jobs, Docker, optional Kubernetes, S3 |

### 🔬 Evaluation Criteria

| Metric | Description |

| :---------------- | :-------------------------------------------------------------------- |

| ⏱️ Time Efficiency| Speed of processing 100s or 1000s of files in bulk |

| 🔁 Scalability | Support for distributed/multi-node execution |

| ⚙️ Resource Usage | CPU, memory, I/O usage |

| 💥 Fault Tolerance| Retries on failure, resumability, and error handling |

| 🔌 Integration Ease| Fits easily into current cron/Jenkins CI/CD setup |

| 🚀 Automation Friendly| CI/CD, Docker/Kubernetes and S3-aware |

### 🧪 HANDLING LAKHS OF FILES IN S3 AND NEW INCOMING FILES

#### ✅ General Approach for All Methods

\* **\*\*Historical XMLs:\*\*** Read from S3 in paginated batches (1000 keys per call) to avoid memory overload.\* **\*\*New XMLs:\*\*** Use polling or event trigger (cron/CloudWatch/EventBridge) every 15 minutes.

```python

# Example S3 Fetch in Batches

import boto3

def list\_files(bucket\_name, prefix=""):

s3 = boto3.client('s3')

paginator = s3.get\_paginator('list\_objects\_v2')

for page in paginator.paginate(Bucket=bucket\_name, Prefix=prefix):

for obj in page.get('Contents', []):

yield obj['Key']

Use this generator pattern in all methods to fetch files in chunks instead of loading lakhs of filenames at once.

### **📦 METHOD 1: Python Multiprocessing (Local Parallelism)**

* 🎯 ****Best for:**** Single-machine, high-CPU bulk processing
* 🔁 ****Strategy:****
  + Fetch files from S3 in chunks (batch of 1000)
  + Download each file locally
  + Use multiprocessing.Pool to run parser across all CPU cores
* 📂 ****Handles:****
  + ✅ Lakhs of old files via loop over S3
  + ✅ New files via cron job running every 15 minutes
* 🧪 ****Step-by-Step:****
  + Install boto3 if using S3:

Bash

pip install boto3

* + Download and parse in parallel:

Python

from multiprocessing import Poolimport boto3, osfrom your\_et\_parser import parse\_file\_and\_load\_db

s3 = boto3.resource('s3')

bucket\_name = "your-bucket-name"

prefix = "incoming/"

def process\_s3\_key(key):

# Download to /tmp

local\_path = f"/tmp/{os.path.basename(key)}"

s3.Bucket(bucket\_name).download\_file(key, local\_path)

parse\_file\_and\_load\_db(local\_path)

os.remove(local\_path)

def run\_batch():

keys = list(list\_files(bucket\_name, prefix))[:1000] # process in batch

with Pool(os.cpu\_count()) as pool:

pool.map(process\_s3\_key, keys)

run\_batch()

* + Schedule with cron:

Bash

\*/15 \* \* \* \* python3 /home/user/parse\_multiprocessing.py

### **📦 METHOD 2: Python ThreadPool (I/O Optimized)**

* 🎯 ****Best for:**** DB insertion bottlenecks (slow writes)
* 🔁 ****Strategy:****
  + Uses threads instead of processes (lightweight)
  + Good for overlapping I/O (DB or S3 reads)
* ⚠️ ****Limitation:**** Less efficient for parsing, which is CPU-bound.
* 🧪 ****Step-by-Step:****
  + Replace Pool with ThreadPool:

Python

from multiprocessing.pool import ThreadPoolwith ThreadPool(10) as pool: # Tune based on DB load capacity

pool.map(process\_s3\_key, keys)

### **📦 METHOD 3: Kubernetes Jobs (Container-based Scaling)**

* 🎯 ****Best for:**** Cloud-native scalable job processing
* 🔁 ****Strategy:****
  + Dockerize parser
  + Use Kubernetes Job to launch multiple pods
  + Each pod gets S3 file batch, parses, and loads DB
* 📂 ****Handles:****
  + ✅ Lakhs of files via job array or queue
  + ✅ New files via scheduled Kubernetes Jobs or CronJob
* 🧪 ****Step-by-Step:****
  + Dockerfile:

Dockerfile

FROM python:3.10-slimCOPY . /app/WORKDIR /appRUN pip install -r requirements.txtCMD ["python", "run\_job.py"]

* + Batch Logic in run\_job.py:

Python

# Use environment variable or job args to pick batch (like batch\_1):import os

batch\_id = int(os.getenv("BATCH\_ID", 0))# fetch keys from S3 for this batch range# ... parsing and loading logic ...

* + K8s Job (job.yaml):

YAML

apiVersion: batch/v1kind: Jobmetadata:

name: xpt-parser-batch-{{BATCH\_ID}}spec:

parallelism: 5

template:

spec:

containers:

- name: parser

image: registry/xpt-parser:latest

env:

- name: BATCH\_ID

value: "{{BATCH\_ID}}"

restartPolicy: Never

backoffLimit: 3

* + Run job per batch of 1000:

Bash

for i in {0..100}; do

kubectl create -f job.yaml --set BATCH\_ID=$idone

* + For new files:

YAML

apiVersion: batch/v1kind: CronJobmetadata:

name: parser-every-15minspec:

schedule: "\*/15 \* \* \* \*"

jobTemplate:

spec:

template:

spec:

containers:

- name: parser

image: registry/xpt-parser:latest

restartPolicy: OnFailure

### **📦 METHOD 4: Apache Spark + spark-xml (Big Data Scale)**

* 🎯 ****Best for:**** Processing 10K+ files at once in a distributed fashion
* 🔁 ****Strategy:****
  + Use spark.read.format("xml") to load entire S3 prefix
  + Converts XML into DataFrame
  + Writes directly to DB
* 🧪 ****Step-by-Step:****
  + Set up Spark locally or on EMR/Databricks
  + Install XML package:

Bash

spark-shell --packages com.databricks:spark-xml\_2.12:0.10.0

* + Sample PySpark Script:

Python

from pyspark.sql import SparkSession

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

df = spark.read.format("xml") \

.option("rowTag", "TowerData") \

.load("s3a://your-bucket/incoming/")

df.write.jdbc("jdbc:postgresql://db:5432/xpt", "tower\_table", "append")

* + For continuous loading, schedule the script via cron or Airflow DAG.

### **📦 METHOD 5: Kafka + Streaming Consumer**

* 🎯 ****Best for:**** Near real-time stream processing
* 🔁 ****Strategy:****
  + Watch S3 or local folder for new files every 15 min
  + Producer sends file content as Kafka message
  + Consumer parses + writes to DB
* 📂 ****Handles:****
  + ✅ Lakhs of files: produce from historical list (can resume using offset)
  + ✅ New files: consumer listens indefinitely
* 🧪 ****Step-by-Step:****
  + Start Kafka/Zookeeper using Docker
  + Create topic:

Bash

kafka-topics.sh --create --topic xpt-xml --bootstrap-server localhost:9092

* + Producer script:

Python

# Assuming list\_files function is available and s3 client is configuredfor key in list\_files(bucket):

content = s3.get\_object(Bucket=bucket, Key=key)['Body'].read()

producer.send("xpt-xml", content)

* + Consumer:

Python

# Assuming consumer and parse\_file\_and\_load\_db are configuredimport uuid # Import uuid for unique filenamesfor message in consumer:

file\_path = f"/tmp/{uuid.uuid4()}.xml"

with open(file\_path, "wb") as f:

f.write(message.value)

parse\_file\_and\_load\_db(file\_path)

os.remove(file\_path)

* + Schedule or run as daemon for real-time processing

### **📈 Metrics to Track**

|  |  |
| --- | --- |
| ****Metric**** | ****Tool**** |
| Parse Time | Python time.time() blocks |
| DB Insert Time | Same |
| Files Processed | stdout or logs |
| CPU/Memory Usage | htop, Prometheus |
| Retry Rate / Errors | Logs / Grafana / K8s Job Status |
| Throughput per Batch | Custom Prometheus counter |

#### **📊 Grafana + Prometheus Setup (Recommended)**

Use prometheus\_client in Python:

Python

from prometheus\_client import start\_http\_server, Summary

REQUEST\_TIME = Summary('parse\_time\_seconds', 'Time taken to parse')

@REQUEST\_TIME.time()def parse\_file\_and\_load\_db(file):

...

Start a metrics server:

Python

start\_http\_server(8000)

Setup Prometheus to scrape metrics and connect Grafana:

****Dashboards:****

* Parse rate
* Failure rate
* p50/p95 insert time
* CPU/Memory by pod

### **✅ Final Recommendation Matrix**

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| ****Method**** | ****Historical Files**** | ****New Files**** | ****Scaling**** | ****Fault Tolerance**** | ****Best Use Case**** |
| Multiprocessing | ✅ Batchwise | ✅ Cron | Single server | ❌ No retries | Simple local setup |
| ThreadPool | ✅ Yes | ✅ Cron | Medium I/O | ❌ Minimal retry | When DB insert is slow |
| Kubernetes Jobs | ✅ Parallel Pods | ✅ CronJob | Horizontal | ✅ Retries | Enterprise cloud setup |
| Spark XML | ✅ Bulk | ✅ Schedule | Massive scale | ✅ Built-in retry | Very large batch processing |
| Kafka Stream | ✅ Re-processable | ✅ Realtime | High | ✅ Replayable | Event-driven stream ingestion |

Please copy the entire text block above (from `# 📘 PROFESSIONAL TEST PLAN` to the end of the last table) and paste it into your document editor. The Markdown formatting should translate well into headings, lists, code blocks, and the tables when you paste it.

As mentioned before, I cannot generate a downloadable `.docx` file directly. You will need to perform the copy-paste action and save it from your document editor.