Real-Time(ish) Fraud Detectin Pipeline (SQS → K3s → RDS/S3 → Athena) [Hands-On Lab]

1. **Project Overview**: In this project, we designed a real-time(ish) fraud detection pipeline that ingests transactions, scores them, and makes results available for live dashboards and historical analytics. The pipeline runs almost entirely in AWS Free Tier, demonstrating some Linux, Kubernetes, and data engineering skills.

2. Objectives

- Build a fraud detection flow end-to-end using AWS services.
- Operate with minimal cost (Free tier + Athena pennies).
- Showcase modern infra skills: containers, event-driven ingest, dual-write storage, and analytics.
- Deliver a resume-ready demo with live dashboards, Athena queries, and blue/green model swaps.

3. Launch your EC2 Instance.

- AMI: Amazon Linux 2023
- Network: default VPC, public subnet, public IP: enabled
- Security group: inbound 22/tcp from your IP only
- IAM role (attach one with admin access, just for this project.)

```
# Installing Tools

sudo dnf update -y

curl -sfL https://get.k3s.io | sh -s - --write-kubeconfig-mode 644

# Python and helpers
```

```
sudo dnf install -y python3-pip git postgresql15
```

4. **Create core AWS resources (SQS, S3, RDS) from your EC2 workstation**

```
# Set env

export AWS_REGION=us-east-1
aws configure set region $AWS_REGION
export BUCKET=fraud-pipe-$RANDOM
```

```
# S3 data lake (raw/analytics)

export BUCKET=fraud-pipe-$RANDOM
aws s3api create-bucket --bucket $BUCKET
aws s3api put-bucket-versioning --bucket $BUCKET --versioning-configuration
Status=Enabled
```

```
# SQS queue (ingest)

export QUEUE_URL=$(aws sqs create-queue \
  --queue-name fraud-tx-queue \
  --attributes ReceiveMessageWaitTimeSeconds=20 \
  --query QueueUrl --output text)


echo "$QUEUE_URL"

aws sqs set-queue-attributes --queue-url $QUEUE_URL \
  --attributes ReceiveMessageWaitTimeSeconds=20
```

```
# RDS PostgreSQL (free tier)
# Why: Postgres gives fast indexed queries for live dashboards.
```

```
export DBID=frauddb
aws rds create-db-instance \
  --db-instance-identifier $DBID \
  --db-instance-class db.t3.micro \
  --engine postgres \
  --allocated-storage 20 \
  --master-username appuser \
  --master-user-password 'StrongPassw0rd!' \
  --publicly-accessible \
  --backup-retention-period 1
aws rds wait db-instance-available --db-instance-identifier $DBID
export DBENDPOINT=$(aws rds describe-db-instances --db-instance-identifier $DBID
\
  --query 'DBInstances[0].Endpoint.Address' -o text)
echo $DBENDPOINT
```

 # Open Security Group (EC2 → RDS:5432)

In this step, we link our EC2 SG to the RDS SG. Without it, our podinside the k3s, cannot reach Postgres.

```
export TOKEN=$(curl -s -X PUT "http://169.254.169.254/latest/api/token" \
  -H "X-aws-ec2-metadata-token-ttl-seconds: 21600")
```

Use the token fetched above to fetch the instance ID and private IP

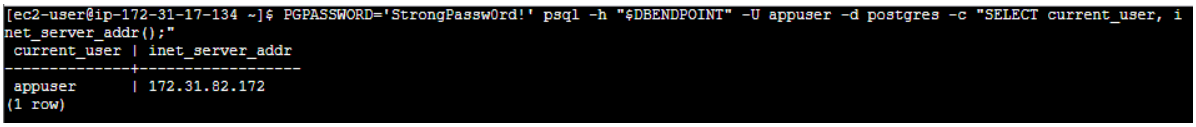
```
export INSTANCE_ID=$(curl -s -H "X-aws-ec2-metadata-token: $TOKEN" \
  http://169.254.169.254/latest/meta-data/instance-id)
export LOCAL_IP=$(curl -s -H "X-aws-ec2-metadata-token: $TOKEN" \
  http://169.254.169.254/latest/meta-data/local-ipv4)

echo "INSTANCE_ID=$INSTANCE_ID  LOCAL_IP=$LOCAL_IP"
```

Get our EC2 Security Group ID

```
export EC2SG=$(aws ec2 describe-instances --instance-id "$INSTANCE_ID" \
  --query 'Reservations[0].Instances[0].SecurityGroups[0].GroupId' --output text)
echo "EC2SG=$EC2SG"
```

```
# Get the RDS security group ID

export DBID=frauddb
export RDSSG=$(aws rds describe-db-instances --db-instance-identifier "$DBID" \
  --query 'DBInstances[0].VpcSecurityGroups[0].VpcSecurityGroupId' --output text)
echo "RDSSG=$RDSSG"

# Authorize EC2 to RDS:5432
aws ec2 authorize-security-group-ingress \
  --group-id "$RDSSG" --protocol tcp --port 5432 --source-group "$EC2SG" || true

# Test Postgres
export DBENDPOINT=$(aws rds describe-db-instances --db-instance-identifier
"$DBID" \
  --query 'DBInstances[0].Endpoint.Address' --output text)
export PGPASSWORD='StrongPassw0rd!'
psql -h "$DBENDPOINT" -U appuser -d postgres -c "SELECT current_user,
inet_server_addr();"
```

 # Verify: The output should resemble the one below to show that we are connected to our database.



5. K3s Cluster setup

What:

```
# Get kubeconfig

export KUBECONFIG=/etc/rancher/k3s/k3s.yaml
kubectl get nodes
kubectl create ns fraud
```

Why: k3s is tiny but production-flavored Kubernetes, and a single node keeps the cost at $0

6. App Identity & Secrets (k8s)

What: We will allow pods to use the EC2 instance role (IMDSv2) for AWS (S3/SQS).

Store credentials in a Kubernetes Secret, and general app configuration in a ConfigMap.

```
kubectl -n fraud create secret generic db-conn \
  --from-literal=DB_HOST=$DBENDPOINT \
  --from-literal=DB_NAME=postgres \
  --from-literal=DB_USER=appuser \
  --from-literal=DB_PASS='StrongPassw0rd!'

kubectl -n fraud create configmap app-config \
  --from-literal=S3_BUCKET=$BUCKET \
  --from-literal=SQS_URL=$QUEUE_URL \
  --from-literal=AWS_REGION=$AWS_REGION \
  --from-literal=MODEL_VERSION=2025-09-01

# Why: Secrets/ConfigMaps cleanly separate config from code; MODEL_VERSION allows you to do blue/green swaps y config
```

7. Database Schema

```
psql -h "$DBENDPOINT" -U appuser -d postgres <<'SQL'
CREATE TABLE IF NOT EXISTS tx_raw(
  txn_id text PRIMARY KEY,
  amount numeric,
  merchant_id text,
  ts timestamptz,
  payload jsonb
);

CREATE TABLE IF NOT EXISTS tx_scored(
```

```
  txn_id text PRIMARY KEY,
  ts timestamptz,
  amount numeric,
  merchant_id text,
  score double precision,
  features jsonb
);

CREATE INDEX IF NOT EXISTS idx_scored_ts ON tx_scored(ts DESC);
CREATE INDEX IF NOT EXISTS idx_scored_top ON tx_scored(score DESC, ts DESC);
SQL
```

 # Verify Tables exist

```
[ec2-user@ip-172-31-17-134 ~]$ psql -h "$DBENDPOINT" -U appuser -d postgres -c "\dt"
         List of relations
 Schema |   Name    | Type  |  Owner
--------+-----------+-------+---------
 public | tx_raw    | table | appuser
 public | tx_scored | table | appuser
(2 rows)
```

Why: tx_raw keeps the original envelope; tx_scored powers fast/score queries.

8. Fake Transaction producer

What: We create a Python script that continuously generates fake credit card transactions and pushes them into SQS. This simulates live transaction traffic for your fraud detection system.

```
mkdir -p ~/producer
cd ~/producer
vim producer.py

import os, json, time, uuid, random, boto3, datetime as dt

REGION = os.getenv('AWS_REGION', 'us-east-1')
QUEUE_URL = os.environ['SQS_URL']
sqs = boto3.client('sqs', region_name=REGION)

def fake_tx():
   return {
```

```
    "txn_id": str(uuid.uuid4()),
    "amount": round(random.uniform(1, 500), 2),
    "merchant_id": f"m{random.randint(1,100)}",
    "ts": dt.datetime.utcnow().isoformat()+"Z",
    "card_hash": f"h{random.getrandbits(64)}"
  }

while True:
  batch = [fake_tx() for _ in range(10)]
  entries = [{"Id": str(i), "MessageBody": json.dumps(tx)} for i, tx in enumerate(batch)]
  sqs.send_message_batch(QueueUrl=QUEUE_URL, Entries=entries)
  print("Sent", len(entries))
  time.sleep(1)
```

 # Install dependencies

```
pip3 install boto3
```

 # Export environment variables

```
export AWS_REGION=us-east-1
export SQS_URL=$(aws sqs get-queue-url --queue-name fraud-tx-queue --query
QueueUrl --output text)
```

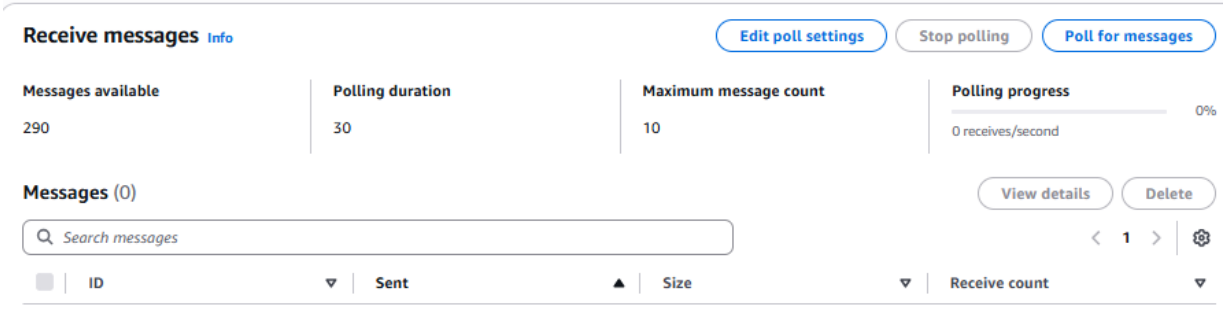
 # Check

```
echo "SQS_URL=$SQS_URL"
```

 # Run the producer.py, and the result should be like the image below

```
[ec2-user@ip-172-31-17-134 producer]$ python3 producer.py
Sent 10
Sent 10
Sent 10
Sent 10
Sent 10
Sent 10
Sent 10
```

 # From the AWS console, in our SQS, we can verify too



 # At this point

- **S3 is ready for historical storage**
- **RDS is ready for scored results**
- **SQS is live and receiving traffic**
- **Producer is stimulating real transaction load.**

9. Scoring Service (FastAPI) on K3s

What:

Mkdir -p ~/fraud-consumer/app
Cd ~/fraud-consumer

 # Create app/main.py

Vim app/main.py
app/main.py

```
import os, json, gzip, boto3, psycopg2, io, time, hashlib
from datetime import datetime
from fastapi import FastAPI

SQS_URL = os.environ["SQS_URL"]
S3_BUCKET = os.environ["S3_BUCKET"]
REGION = os.environ.get("AWS_REGION","us-east-1")
MODEL_VERSION = os.environ.get("MODEL_VERSION","baseline")

sqs = boto3.client("sqs", region_name=REGION)
s3  = boto3.client("s3", region_name=REGION)

conn = psycopg2.connect(
   host=os.environ["DB_HOST"], dbname=os.environ["DB_NAME"],
   user=os.environ["DB_USER"], password=os.environ["DB_PASS"])
conn.autocommit = True

def score(tx):
   h = int(hashlib.sha256(tx["card_hash"].encode()).hexdigest(), 16) % 100
   base = tx["amount"]/500.0
   return min(1.0, 0.3*base + 0.7*(h/100.0))

def write_s3(tx):
   now = datetime.utcnow()
   key =
f"tx/dt={now.strftime('%Y-%m-%d')}/hr={now.strftime('%H')}/part-{int(time.time())}.json.gz
"
   buf = io.BytesIO()
   with gzip.GzipFile(fileobj=buf, mode="w") as gz:
      gz.write((json.dumps(tx)+"\n").encode())
   s3.put_object(Bucket=S3_BUCKET, Key=key, Body=buf.getvalue())

def write_db(tx, s):
   with conn.cursor() as cur:
      cur.execute("""INSERT INTO tx_raw(txn_id, amount, merchant_id, ts, payload)
                     VALUES (%s,%s,%s,%s,%s) ON CONFLICT DO NOTHING""",
                  (tx["txn_id"], tx["amount"], tx["merchant_id"], tx["ts"], json.dumps(tx)))
      cur.execute("""INSERT INTO tx_scored(txn_id, ts, amount, merchant_id, score,
features)
                     VALUES (%s,%s,%s,%s,%s,%s) ON CONFLICT DO NOTHING""",
```

```
                (tx["txn_id"], tx["ts"], tx["amount"], tx["merchant_id"], s, json.dumps({"model":
MODEL_VERSION})))

app = FastAPI()
@app.get("/health")
def health(): return {"ok": True, "model": MODEL_VERSION}

def loop():
   while True:
      msgs = sqs.receive_message(QueueUrl=SQS_URL, MaxNumberOfMessages=10,
WaitTimeSeconds=20).get("Messages",[])
      for m in msgs:
         tx = json.loads(m["Body"])
         s = score(tx)
         write_db(tx, s)
         write_s3({**tx, "score": s})
      if msgs:
         sqs.delete_message_batch(QueueUrl=SQS_URL, Entries=[{"Id":x["MessageId"],
"ReceiptHandle":x["ReceiptHandle"]} for x in msgs])

import threading; threading.Thread(target=loop, daemon=True).start()
```

 # Create Dockerfile in ~/fraud-consumer

Vim Dockerfile

```
FROM python:3.11-slim
RUN pip install fastapi uvicorn boto3 psycopg2-binary
WORKDIR /app
COPY app/ /app/
CMD ["uvicorn","main:app","--host","0.0.0.0","--port","8080"]
```

 # Create an ECR repo and log in (from your EC2 box)

- **# Install nerdctl**

```
sudo curl -L
https://github.com/containerd/nerdctl/releases/download/v1.7.7/nerdctl-full-1.7.7-linux-amd64.tar.gz -o /tmp/nerdctl.tgz
sudo tar -C /usr/local -xzvf /tmp/nerdctl.tgz
sudo systemctl enable --now buildkit
```

- **# Create ECR repo**

```
export REPO=fraud-consumer

aws ecr create-repository --repository-name $REPO --region $REGION || true
```

- **# login nerdctl to ECR**

```
aws ecr get-login-password --region $REGION \
  | sudo nerdctl login --username AWS --password-stdin
${ACCOUNT_ID}.dkr.ecr.${REGION}.amazonaws.com

**Note** It should give out "login succeeded"
```

 # Build, tag, push the image to ECR

sudo nerdctl --address /run/k3s/containerd/containerd.sock --namespace k8s.io build -t fraud-consumer:latest .

****Note** The result should be like the image below**

```
Loaded image: docker.io/library/fraud-consumer:latest
```

****Note**: Verify the image is visible to k3s**

sudo nerdctl --address /run/k3s/containerd/containerd.sock --namespace k8s.io images | grep fraud-consumer

```
[ec2-user@ip-172-31-17-134 fraud.consumer]$ sudo nerdctl --address /run/k3s/containerd/containerd.sock --namespace k8s.io images
| grep fraud-consumer
fraud-consumer                  latest                  e9eb9540bdc8    3 minutes ago    linux/amd64    219.7 MiB    92.2 MiB
```

export IMG_LOCAL=fraud-consumer:latest
export IMG_ECR=${ACCOUNT_ID}.dkr.ecr.${REGION}.amazonaws.com/${REPO}:latest

 # Tag the image that's in the k3s's containerd namespace.

sudo nerdctl --address /run/k3s/containerd/containerd.sock --namespace k8s.io \
  tag $IMG_LOCAL $IMG_ECR

 # Push it to the ECR
sudo nerdctl --address /run/k3s/containerd/containerd.sock --namespace k8s.io \
  push $IMG_ECR

#Verify it's in the ECR

```
[ec2-user@ip-172-31-17-134 fraud.consumer]$ aws ecr describe-images --repository-name $REPO --query 'imageDetails[].imageTags' --
output table
----------------
|DescribeImages|
+--------------+
|  latest      |
+--------------+
```

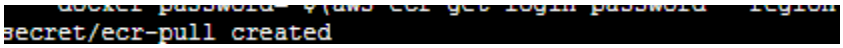
 # Deploy to K3s

- **Create/refresh the ECR pull secret (namespace: fraud)**

```
kubectl -n fraud delete secret ecr-pull 2>/dev/null || true
kubectl -n fraud create secret docker-registry ecr-pull \
```

```
  --docker-server=${ACCOUNT_ID}.dkr.ecr.${REGION}.amazonaws.com \
  --docker-username=AWS \
  --docker-password="$(aws ecr get-login-password --region $REGION)"
```

****Note** It should give out the outcome like in the image below.**



- **Use the image in your Deployment**

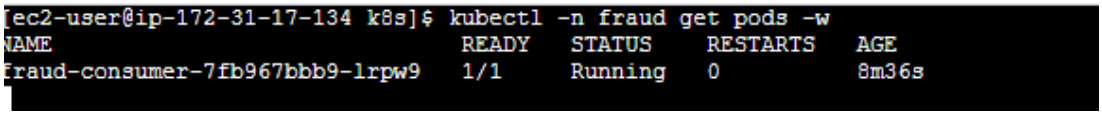
```
Mkdir -p ~/fraud-consumer/k8s

cat > ~/fraud-consumer/k8s/k8s-consumer.yaml <<EOF
apiVersion: apps/v1
kind: Deployment
metadata:
  name: fraud-consumer
  namespace: fraud
spec:
  replicas: 1
  selector:
    matchLabels:
      app: fraud-consumer
  template:
    metadata:
      labels:
        app: fraud-consumer
    spec:
      imagePullSecrets:
      - name: ecr-pull
      containers:
      - name: app
        image: ${IMG}
        imagePullPolicy: Always
        ports:
        - containerPort: 8080
        envFrom:
        - secretRef:
            name: db-conn
        - configMapRef:
            name: app-config
```

```
---
apiVersion: v1
kind: Service
metadata:
  name: fraud-api
  namespace: fraud
spec:
  type: NodePort
  selector:
    app: fraud-consumer
  ports:
  - port: 80
    targetPort: 8080
EOF
```

- **# Apply and watch**

  kubectl -n fraud apply -f ~/fraud-consumer/k8s/k8s-consumer.yaml
  kubectl -n fraud rollout status deploy/fraud-consumer
  kubectl -n fraud get pods -w

  ****Note** If everything is working well, you should show your pods like in the image below.**



- **# Verify**
  export PGPASSWORD='StrongPassw0rd!'
  psql -h "$DBENDPOINT" -U appuser -d postgres -c "SELECT count(*) FROM tx_scored;"
  aws s3 ls s3://$BUCKET/tx/ --recursive | tail -n 5

  ****Note** If everything is running well, you should see the output like in the image below.**

```
[ec2-user@ip-172-31-17-134 k8s]$ export PGPASSWORD='StrongPassw0rd!'
psql -h "$DBENDPOINT" -U appuser -d postgres -c "SELECT count(*) FROM tx_scored;"
aws s3 ls s3://$BUCKET/tx/ --recursive | tail -n 5
 count
-------
   490
(1 row)

2025-09-02 02:47:22 do-not-delete-ssm-diagnosis-782781395980-us-east-1-qpmrx
2025-09-05 04:02:53 fraud-pipe-22591
```

-

10. Athena with Partitioon Projecton

What: We enable Athena to query the fraud-scored transactions stored in S3 without Glue crawlers. We use partition projection so Athena knows how to "pretend" partition exist (date/hour), even if we don't explicitly register them. This keeps cost at near ero and avoids constant *MSCK REPAIR* or crawler runs.

Why: We built a fraud detection pipeline that streams into RDS for live dashboards and into S3 for audit/BI. Athena queries now JSON on S3 with partition projection at almost no cost.

- **Setup: Got to AWS Console → Athena - editor preference.**

```
s3://fraud-pipe-22591/athena/
```

- **Query Editor**

```
# Create database

CREATE DATABASE IF NOT EXISTS fraud;

 Click run and on the left panel choose Database = fraud
```

-

```
# Create the external table

CREATE EXTERNAL TABLE IF NOT EXISTS fraud.fraud_tx (
  txn_id string,
```

```
  amount double,
  merchant_id string,
  ts string,
  card_hash string,
  score double
)
PARTITIONED BY (dt string, hr string)
ROW FORMAT SERDE 'org.openx.data.jsonserde.JsonSerDe'
LOCATION 's3://fraud-pipe-22591/tx/'
TBLPROPERTIES (
  'projection.enabled'='true',
  'projection.dt.type'='date',
  'projection.dt.format'='yyyy-MM-dd',
  'projection.dt.range'='2025-01-01,NOW',
  'projection.hr.type'='integer',
  'projection.hr.range'='0,23',
  'storage.location.template'='s3://fraud-pipe-22591/tx/dt=${dt}/hr=${hr}/'
);

# Click run again, and you will see fraud_tx  appear under the table
```

- **Run the demo query**

```
SELECT *
FROM fraud.fraud_tx
WHERE dt = date_format(current_date, '%Y-%m-%d')
  AND hr = date_format(current_timestamp, '%H')
  AND from_iso8601_timestamp(ts) > current_timestamp - INTERVAL '15' MINUTE
  AND score > 0.9
ORDER BY ts DESC
LIMIT 50;

# After this, Athena will return rows directly from your s3 data lake. Thats your "audit and BI" proof point for the project.
```

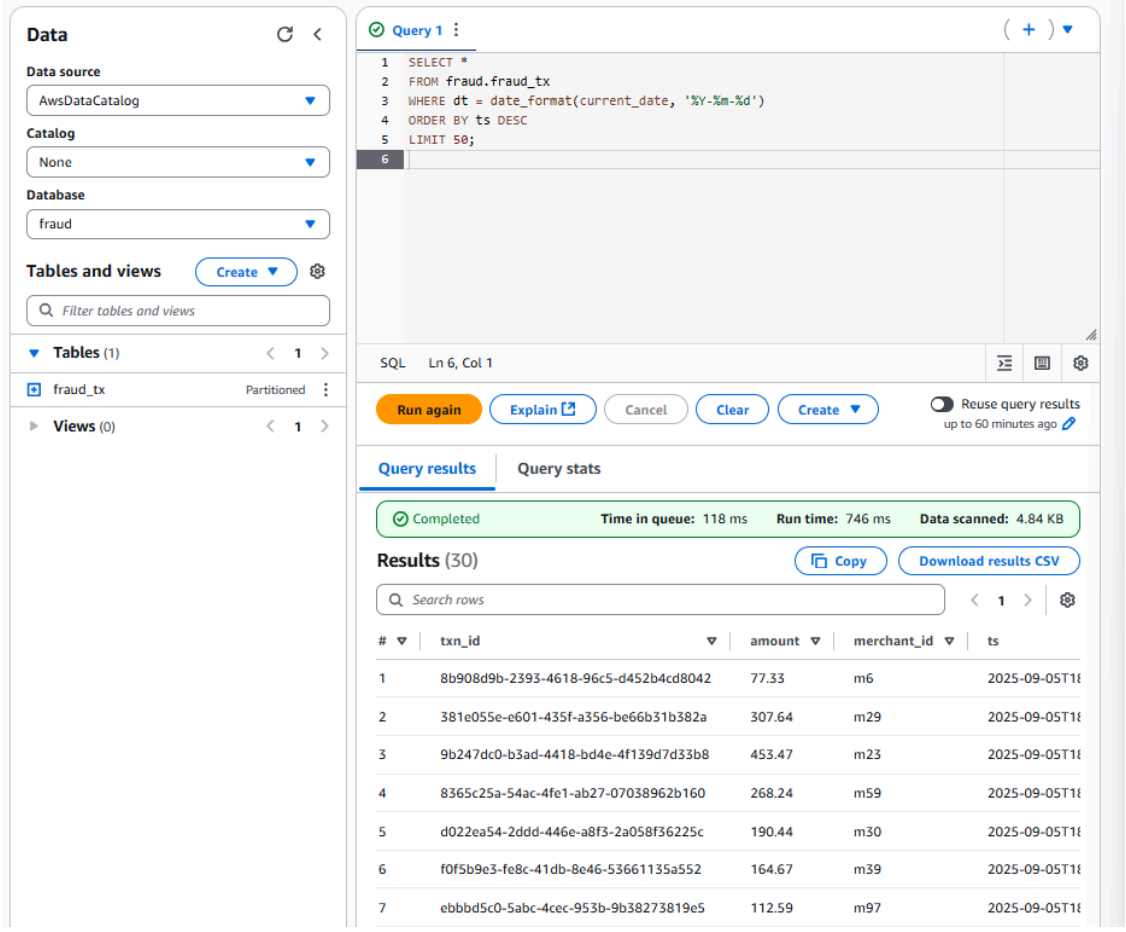
- **Verify - Broaden the filter (test mode). Run this query to check all rows in today's partition.**

```
SELECT *
FROM fraud.fraud_tx
WHERE dt = date_format(current_date, '%Y-%m-%d')
```

ORDER BY ts DESC
LIMIT 50;

****Note** The output should resemble the one in the image below. The query just pulled 30 fresh rows from the JSON files in S3.**

- Producer is pushing transactions into SQS
- Consumer pod is scoring and dual writing (RDS +S3).
- Athena is projecting partitions and reading the S3 lake in real-time.
- The query results show structured fields (txn_id, amount, merchant_id,..)



11. Nightly Retrain (CronJob)

 # create k8s/k8s-retrain.yaml

Cd fraud-consumer/k8s
Vim k8s-retrain.yaml

```
apiVersion: batch/v1
kind: CronJob
metadata:
  name: retrain
  namespace: fraud
spec:
  schedule: "0 3 * * *"          # nightly at 03:00 UTC
  concurrencyPolicy: Forbid
  successfulJobsHistoryLimit: 2
  failedJobsHistoryLimit: 2
  jobTemplate:
    spec:
      backoffLimit: 1
      template:
        spec:
          restartPolicy: OnFailure
          containers:
          - name: retrain
            image: python:3.11-slim
            envFrom:
            - configMapRef:
                name: app-config
            command:
            - bash
            - -c
            - |
              set -euo pipefail
              pip install --no-cache-dir boto3 pandas scikit-learn psycopg2-binary pyarrow
              python /job/retrain.py
            volumeMounts:
            - name: job-script
              mountPath: /job
          volumes:
          - name: job-script
            configMap:
              name: retrain-script
```

 # ConfigMAp for Python script

```
kubectl -n fraud create configmap retrain-script \
  --from-literal=retrain.py='
import os, boto3, datetime as dt
s3 = boto3.client("s3", region_name=os.getenv("AWS_REGION","us-east-1"))
key = f"models/model-{dt.datetime.utcnow():%Y%m%d}.pkl"
s3.put_object(Bucket=os.environ["S3_BUCKET"], Key=key, Body=b"MODEL_BYTES")
print("Uploaded", key)
```

```
kubectl -n fraud apply -f ~/fraud-consumer/k8s/k8s-retrain.yaml
kubectl -n fraud get cronjobs
```

 # Test run immediately.

```
kubectl -n fraud create job --from=cronjob/retrain retrain-now
kubectl -n fraud get pods -w
```

****Note** This means our CronJob definition is correct. It ran immediately we created a one-off job. It uploaded a versioned model artifact to S3. At 03:00 utc every night, the cronjob will repeat automatically.**

```
[ec2-user@ip-172-31-17-134 fraud-consumer]$ kubectl -n fraud create job --from=cronjob/retrain retrain-now
kubectl -n fraud get pods -w
job.batch/retrain-now created
NAME                              READY   STATUS              RESTARTS   AGE
fraud-consumer-7fb967bbb9-lrpw9   1/1     Running             0          98m
retrain-now-p6tn4                 0/1     ContainerCreating   0          0s
retrain-now-p6tn4                 1/1     Running             0          1s
retrain-now-p6tn4                 0/1     Completed           0          29s
retrain-now-p6tn4                 0/1     Completed           0          30s
retrain-now-p6tn4                 0/1     Completed           0          31s
```

```
[ec2-user@ip-172-31-17-134 fraud-consumer]$ aws s3 ls s3://$BUCKET/models/
2025-09-02 02:47:22 do-not-delete-ssm-diagnosis-782781395980-us-east-1-qpmrx
2025-09-05 04:02:53 fraud-pipe-22591
```

12. Live Dashboard (Grafana → Postgres)

 # What we did

- **Deploy Grafana in the** *fraud* **namespace.**

```
# Create Grafana deployment
```

```
kubectl -n fraud create deployment grafana --image=grafana/grafana:latest

# Expose the created Grafana pod and expose it via Kubernetes Service on a random NodePort
kubectl -n fraud expose pod grafana –type=NodePort –port=3000
```

- **Find the NodePort + EC2 public IP**

```
kubectl -n fraud get svc grafana -o wide
curl -s http://169.254.169.254/latest/meta-data/public-ipv4
# Example: http://<EC2_PUBLIC_IP>:32571
```

- **Update Security Group**

In the AWS console, add an Inbound Rule to the EC2's security group.

- Type: Custom IP
- Port: 32571
- Source: 0.0.0.0/0 (or your own IP for more security)

Now Grafana is reachable at http://<EC2_PUBLIC_IP>:32571

- **Log in to Grafana**

Default Credentials:

- Username: admin
- Password: Admin (forces reset on first login)

- **Add Postgres as a Data Source**

Go to configuration → Data Sources → Add new → PostgreSQL

- Host: <RDS_ENDPOINT>:5432 (from our earlier RDS setup)
- Database: postgres
- User: appuser
- Password: <your DB password>
- TLS/SSL
- Version: set to 15 (since our RDS is PostgreSQL 17, use the highest available option in Grafana).
- Click Save and Test → should say "Database Connection OK".

- **Add Panels: Import as JSON**

```
{
  "dashboard": {
    "id": null,
    "title": "Fraud Detection Pipeline",
    "tags": ["fraud", "postgres", "pipeline"],
    "timezone": "browser",
    "schemaVersion": 36,
    "version": 1,
    "panels": [
      {
        "id": 1,
        "title": "High-Risk Transactions (Last 15m)",
        "type": "timeseries",
        "datasource": "Fraud Pipeline",
        "targets": [
          {
            "format": "time_series",
            "rawSql": "SELECT date_trunc('minute', ts) AS minute, count(*) AS
high_risk_count FROM tx_scored WHERE ts > now() - interval '15 minutes' AND
score > 0.9 GROUP BY 1 ORDER BY 1;",
            "refId": "A"
          }
        ],
        "fieldConfig": {
          "defaults": {
            "unit": "short"
          }
        }
      },
      {
        "id": 2,
        "title": "High-Risk in Last 15m",
        "type": "stat",
        "datasource": "Fraud Pipeline",
        "targets": [
          {
            "format": "table",
            "rawSql": "SELECT count(*) FROM tx_scored WHERE ts > now() - interval '15
minutes' AND score > 0.9;",
            "refId": "A"
```

```
        }
      ],
      "fieldConfig": {
        "defaults": {
          "unit": "none",
          "color": {
            "mode": "thresholds"
          },
          "thresholds": {
            "mode": "absolute",
            "steps": [
              { "color": "green", "value": null },
              { "color": "red", "value": 1 }
            ]
          }
        }
      },
      "options": {
        "reduceOptions": {
          "calcs": ["lastNotNull"],
          "fields": "",
          "values": false
        }
      }
    },
    {
      "id": 3,
      "title": "Model Output Comparison (Last 1h)",
      "type": "timeseries",
      "datasource": "Fraud Pipeline",
      "targets": [
        {
          "format": "time_series",
          "rawSql": "SELECT date_trunc('minute', ts) AS minute, features->>'model'
AS model_version, count(*) AS cnt FROM tx_scored WHERE ts > now() - interval
'1 hour' GROUP BY 1,2 ORDER BY 1;",
          "refId": "A"
        }
      ],
      "fieldConfig": {
```

```
        "defaults": {
          "unit": "short"
        }
      }
    },
    {
      "id": 4,
      "title": "Top Merchants (High-Risk, Last 15m)",
      "type": "table",
      "datasource": "Fraud Pipeline",
      "targets": [
        {
          "format": "table",
          "rawSql": "SELECT merchant_id, count(*) AS risky_count FROM tx_scored
WHERE ts > now() - interval '15 minutes' AND score > 0.9 GROUP BY merchant_id
ORDER BY risky_count DESC LIMIT 5;",
          "refId": "A"
        }
      ],
      "fieldConfig": {
        "defaults": {
          "unit": "short"
        }
      },
      "options": {
        "showHeader": true
      }
    }
  ]
 },
 "overwrite": true
}
```

12. Conclusion

This project demonstrated how to design and deploy a real-time(ish) fraud detection pipeline entirely on AWS Free Tier resources. By combining SQS for event ingest, RDS for fast dashboards, S3 for cost-effective data lake storage, Athena for queryable analytics, and Grafana for live visualization, we delivered a modern, end-to-end data pipeline without leaving the free tier.

Along the way, we also:

- Practiced Linux and Kubernetes operations by running k3s on a single EC2 instance.

- Showcased containerization skills by packaging our FastAPI scoring service and deploying it via Kubernetes manifests.

- Implemented blue/green model swaps using ConfigMaps and CronJobs for nightly retraining—highlighting real-world ML lifecycle practices.

- Balanced cost and security trade-offs (e.g., public IP only for EC2, SG-to-SG linking, no NAT Gateway)