Real-Time Fraud Detection Pipeline Documentation

Table of Contents

1. Objective
2. Abstract
3. Local Setup
4. System Architecture
    - Phase 1: Data Ingestion & Raw Storage
    - Phase 2: Real-Time Processing & Feature Store Update
    - Phase 3: Model Serving & Explainability
    - Phase 4: Observability & Alerting
    - Phase 5: CI/CD, Testing & Maintenance
5. Conclusion
6. Glossary

1.Objective

To design and implement a scalable, fault-tolerant real-time fraud-detection pipeline that seamlessly ingests high-velocity transaction streams, applies machine-learning–based anomaly detection with explainable outputs, and provides comprehensive monitoring and alerting—all within a fully containerized and CI-driven framework.

2. Abstract

This project presents a complete real-time fraud-detection solution built on open-source technologies. Financial transactions are simulated and published to Apache Kafka with Avro serialization, then consumed by a Python service that stores raw logs in PostgreSQL, computes per-user features, and maintains a live feature store in Redis. A FastAPI application exposes a `/predict` endpoint for anomaly scoring using an Isolation Forest model and an `/explain` endpoint powered by a local LLM for human-readable rationales. Prometheus captures service and custom fraud metrics, while Alertmanager routes notifications to console and email upon threshold breaches. The entire stack is orchestrated via Docker Compose, and a GitHub Actions CI pipeline ensures code quality, integration testing against real services (Kafka, Redis, Postgres, Schema Registry), nightly backups, and daily ETL cleanup—delivering a production-grade system ready for deployment and extension.

3. Local Setup

3.1 Clone & Environment Setup

```
git clone https://github.com/Natdan24/real-time-fraud-detection-pipeline.git
cd real-time-fraud-detection-pipeline
python -m venv .venv && source .venv/bin/activate
pip install -r requirements_api.txt -r requirements_dev.txt
```

3.2 Environment Variables

Create a .env file and set:
- KAFKA_BOOTSTRAP_SERVERS=localhost:9092
- SCHEMA_REGISTRY_URL=http://localhost:8081
- POSTGRES_HOST=localhost
- REDIS_HOST=localhost
- SMTP_... for email alerts

3.3 Start Services

```
docker compose up -d zookeeper kafka schema-registry redis postgres api prometheus alertmanager grafana
```

Env-file vs. Compose inline:
You can either set vars in your shell (as above) or place them in a `.env` file and tell Docker Compose to load it via:

```Yaml
services:
  api:
    env_file:
      - .env
    …
```

Persistence Note:
Redis and Postgres are backed by named Docker volumes (`redis-data`, `postgres-data`) in your `docker-compose.yml`.
This ensures your feature store and raw transaction tables survive container restarts.

4. System Architecture

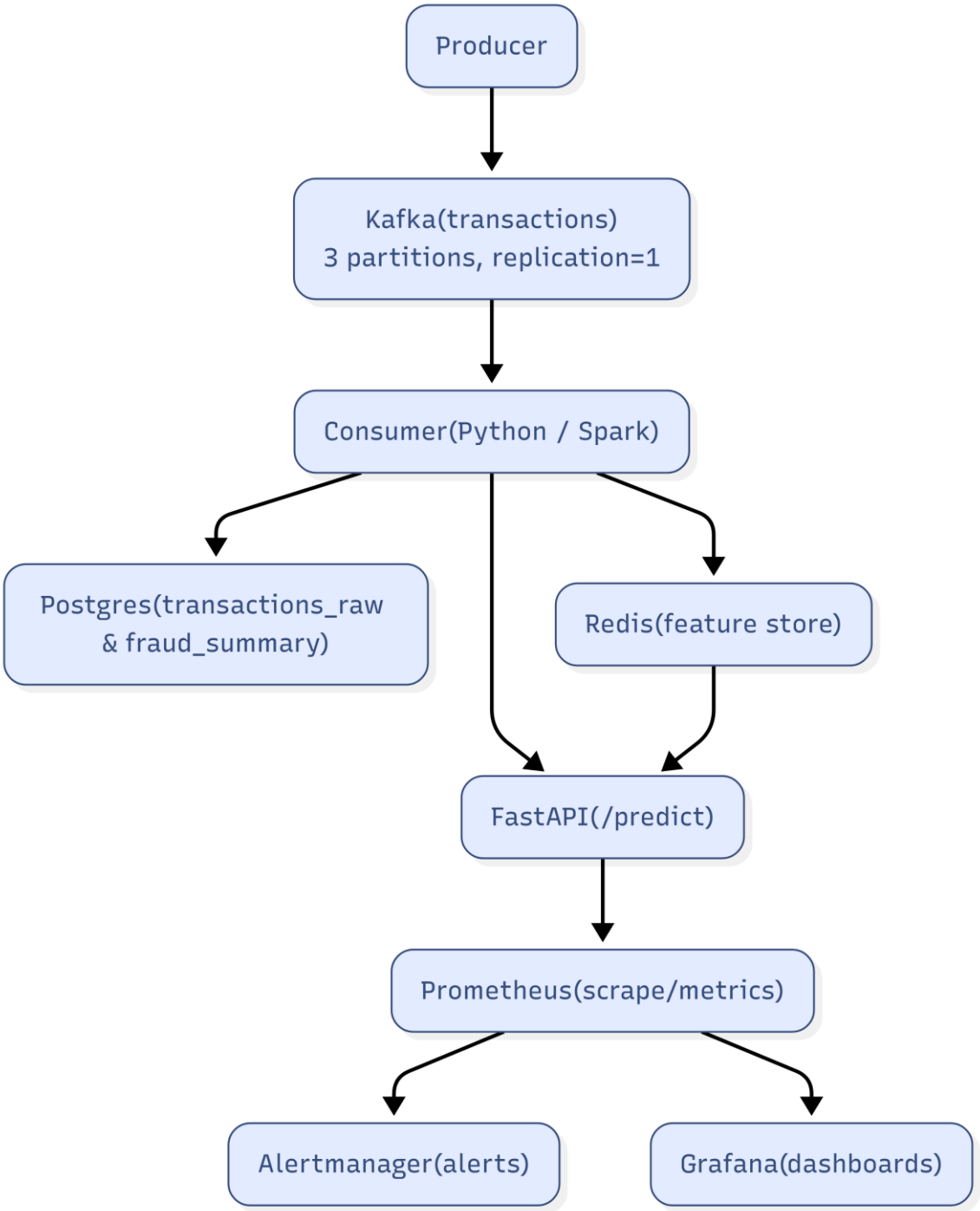


Fig 1: Pipeline Architecture

The Real-Time Fraud Detection Pipeline is composed of several loosely coupled, containerized components that work together to ingest, process, serve, monitor, and alert on transactional data in real time. Below is a detailed breakdown of each piece and how they interact:

1. **Producer (`producer.py`)**
    - **Role:** Simulates a stream of financial transactions and publishes them to the Kafka message bus.
    - **Details:**
        - Generates randomized transaction records (ID, user ID, amount, timestamp, country).

        - Encodes messages in Avro and registers the schema automatically via the Confluent Schema Registry.
        - Sends each record to the `transactions` Kafka topic.
        - Also writes a durable copy of the raw transaction into a PostgreSQL table (`transactions_raw`) for archival and offline analytics.

2. **Consumer (`consumer.py`)**
    - **Role:** Subscribes to the `transactions` topic, enriches and aggregates data, and feeds downstream systems.
    - **Details:**
        - Uses the AvroConsumer API to deserialize Kafka messages.
        - Inserts each transaction into the same `transactions_raw` Postgres table (for fault tolerance).
        - Performs user-level aggregation (transaction count, total and average amount) by querying Postgres.
        - **Updates Redis** with the latest per-user summary (`user:<id>` hash) so the model-serving layer can fetch features in constant time.
        - Invokes the FastAPI `/predict` endpoint to compute an anomaly score and fraud flag.
        - Upserts results into a `fraud_summary` table in Postgres for historical trends and monitoring metrics.
3. **FastAPI Server (`serve_model.py`)**
    - **Role:** Exposes the machine-learning model as an HTTP API, provides explainability, and emits Prometheus-compatible metrics.
    - **Details:**
        - **`GET /predict`**: Looks up precomputed features in Redis, runs an Isolation Forest (or other serialized model) to get an anomaly score and boolean `is_fraud`.
        - **`GET /explain`**: Constructs a natural-language prompt (e.g. "This transaction was flagged because…"), sends it to a local Mistral-7B instance via LMStudio, and returns the LLM's explanation.
        - **Metrics**: Instrumented with `prometheus-fastapi-instrumentator` to automatically export HTTP request counters and latency histograms, plus custom counters for total transactions scored and total frauds detected. Metrics are available at `/metrics`.
4. **Prometheus & Grafana**
    - **Prometheus**
        - Scrapes the FastAPI `/metrics` endpoint and itself to collect both system-level and application-specific metrics.
        - Houses alerting rules (`rules.yml`) that evaluate key conditions:
            - **HighPredictTraffic**: Triggered when request rate to `/predict` exceeds a threshold.
            - **HighFraudRate**: Triggered when the ratio of frauds to total transactions surpasses a threshold.
    - **Grafana**
        - Connects to Prometheus as a data source.
        - Provides dashboards visualizing transaction throughput, fraud-rate trends, latency distributions, and usage of explainability endpoints.

5. **Alertmanager**

- **Role:** Receives fired alerts from Prometheus and routes them to configured destinations.
- **Details:**
  - Uses a declarative `config.yml` to define receivers and routing logic.
  - Sends alerts to the console (for local testing) and to an SMTP e-mail endpoint for production-style notifications.
  - Supports templated notifications with summaries, descriptions, and timestamps for rapid incident awareness.

6. **CI/CD Pipeline (GitHub Actions)**
   - **Role:** Ensures code quality, reliability, and reproducibility.
   - **Details:**
     - **Linting**: Runs Flake8 to enforce style guidelines.
     - **Unit & Integration Tests**: Spins up ephemeral Docker services (Postgres, Redis, Zookeeper, Kafka, Schema Registry) to run pytest against both the API and consumer logic.
     - **Docker Builds**: Builds the FastAPI image (and optionally producer/consumer images) to verify Dockerfiles and dependencies.
     - Can be extended with deployment steps, nightly backup jobs, and more elaborate smoke tests to provide full DevOps coverage.

➢ Phase 1: Data Ingestion & Raw Storage

In **Phase 1**, we focus on generating and persistently storing the incoming transaction stream.

1. **Kafka & ZooKeeper Setup**
   - **ZooKeeper**
     - Coordinates Kafka brokers ("bookkeeping").
     - Runs in Docker on port **2181**.
   - **Kafka Broker**
     - Listens on **9092**, advertised as `localhost:9092`.
     - Automatically creates the `transactions` topic (configured for 3 partitions, replication 1).

```
● localhost:9092 --describe --topic transactions
  >>
  Topic: transactions     TopicId: FhlbQQJSRTe6BqOHha8P2w PartitionCount: 3       ReplicationFactor: 1    Configs:
          Topic: transactions     Partition: 0    Leader: 1       Replicas: 1     Isr: 1
          Topic: transactions     Partition: 1    Leader: 1       Replicas: 1     Isr: 1
          Topic: transactions     Partition: 2    Leader: 1       Replicas: 1     Isr: 1
```

Fig 2: Kafka Transactions , number of Partitions and Replications

2. **Producer (`producer.py`)**
   - **Role:** Simulate a continuous stream of transaction events.
   - **Key steps:**
     1. **Generate** a fake transaction record:

```
python
Copy code
{
  "transaction_id": random.randint(100000,999999),
  "user_id":        random.choice([101,102,103,104]),
  "amount":         round(random.uniform(10,5000),2),
  "timestamp":      int(time.time()),
  "country":        random.choice(["US","NG","GB","IN","DE"])
```

```
}
```

2. **Publish** to Kafka topic `transactions` using **AvroProducer** (registers schema in Schema Registry).
3. **Write** the same record into PostgreSQL's `transactions_raw` table for durable archival and offline analysis.

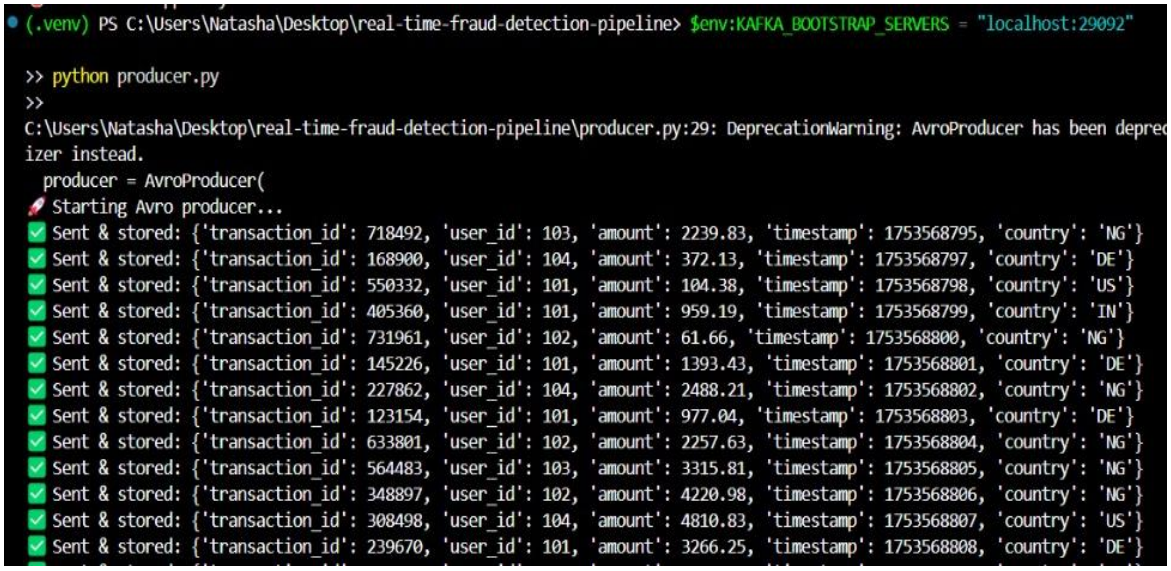


Figure 3: Transactions produced by Avro Producer.py

3. **PostgreSQL Raw Table**
   - **Table:** `transactions_raw`
   - **Columns:**
     - `id` (transaction_id)
     - `user_id`
     - `amount`
     - `timestamp` (converted to `TIMESTAMP`)
     - `country`
   - **Usage:**
     - Acts as a replayable, fault-tolerant ledger of every incoming transaction.
     - Serves as the source of truth for downstream aggregations and audits.

Once Phase 1 is up and running, a steady stream of simulated transactions both in Kafka and in Postgres database. This foundation ensures that no event is lost and provides the raw material for all subsequent processing, feature computation, and monitoring.

➢ Phase 2: Real-Time Processing & Feature Store Update

In **Phase 2**, we process the raw transaction stream, compute user-level features, and make those features available for low-latency model serving.

1. **Consumer (`consumer.py`)**

- **Subscription:** Listens on the `transactions` Kafka topic using `AvroConsumer`.
- **Postgres Write & Aggregate:**
  1. Inserts each incoming record into `transactions_raw` (again), ensuring idempotent durability.
  2. Queries Postgres to compute per-user aggregates:
     - `transaction_count = COUNT(*)`
     - `total_amount = SUM(amount)`
     - `avg_amount = total_amount / transaction_count`
- **Redis Feature Store Update:**
  - Connects to Redis and executes:

```
python
Copy code
r.hset(
  f"user:{user_id}",
  mapping={
    "count":      int(transaction_count),
    "avg_amount": float(avg_amount),
  }
)
```

  - This keeps a live hash per user (`user:<id>`) so the model-serving layer can read features in O(1) time.

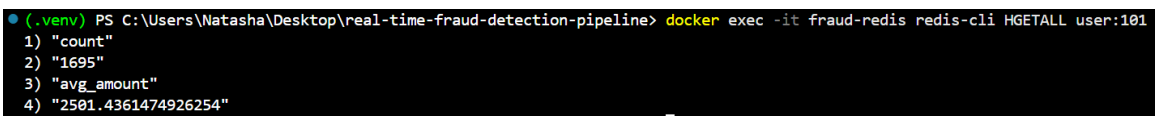


Figure 4 : Feature Store Update

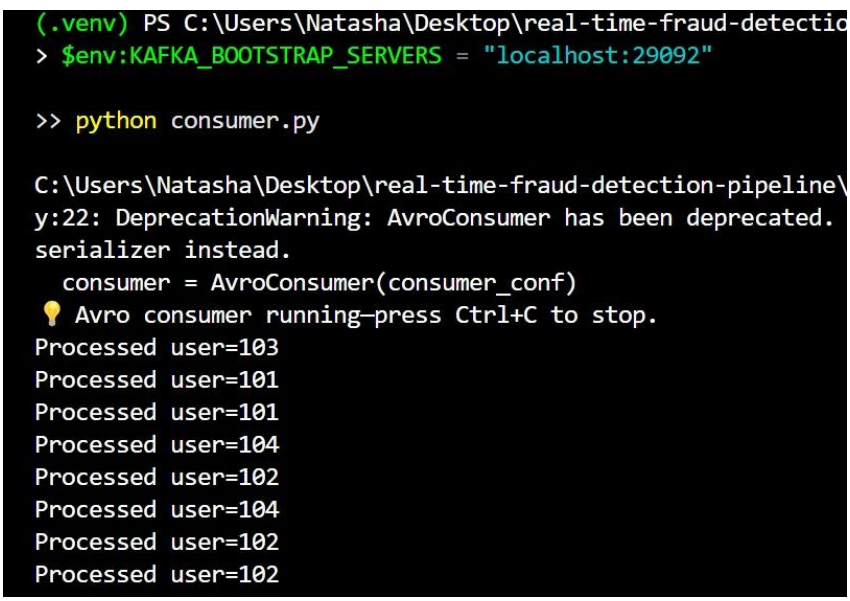


Figure 5: Avro Consumer

- **Error Handling & Transactions:**
  - Wraps Postgres operations in a transaction; on any failure rolls back partial writes.
  - Writes to Redis **before** calling the prediction endpoint to avoid missing feature data.

2. **FastAPI `/predict` Integration**
   - **Endpoint Logic** (`serve_model.py`):

```
python
Copy code
@app.get("/predict")
def predict(user_id: int):
    key = f"user:{user_id}"
    if not r.exists(key):
        raise HTTPException(404, detail=f"No features for user {user_id}")
    data       = r.hgetall(key)
    score      = model.decision_function([[int(data["count"]),
float(data["avg_amount"]) ]])[0]
    is_fraud   = score < 0.0
    return PredictionResponse(user_id, score, is_fraud)
```

- Ensures that every call to `/predict` reflects the very latest aggregates stored in Redis.

3. **Prometheus Metrics**
   - **Automatic Instrumentation:**
     - Uses `prometheus-fastapi-instrumentator` to expose:
       - `http_requests_total{handler="/predict",status="2xx"}`
       - `http_request_latency_seconds_bucket{handler="/predict",…}`
   - **Custom Counters:** Defined in `metrics.py` and incremented on each `/predict` call:

```
python
Copy code
TRANSACTIONS_TOTAL = Counter("fraud_summary_total_transactions", "Total
transactions scored")
FRAUDS_TOTAL       = Counter("fraud_summary_is_fraud_total", "Total
transactions flagged fraud")
```

Phase 2 completes the real-time processing loop: Kafka → Consumer → Postgres + Redis → FastAPI readiness. Now the system is capable of serving fresh anomaly scores at sub-second latency.

➢ Phase 3: Model Serving & Explainability

In **Phase 3**, we expose our anomaly-detection model as an HTTP API, provide natural-language explanations, and integrate feature-driven serving.

1. **FastAPI Server Setup**
   - **File:** `serve_model.py`
   - **Containerization:**

```
dockerfile
CopyEdit
FROM python:3.12-slim
WORKDIR /app
COPY requirements_api.txt .
RUN pip install --no-cache-dir -r requirements_api.txt
COPY serve_model.py metrics.py models/ ./
EXPOSE 8000
CMD ["uvicorn", "serve_model:app", "--host", "0.0.0.0", "--port", "8000", "--
reload"]
```

   - **Startup:**

```
bash
CopyEdit
docker compose up -d api
```

2. **`GET /predict` Endpoint**
   - **Purpose:** Return the latest anomaly score and fraud flag for a given user in real time.
   - **Implementation:**

```
python
CopyEdit
@app.get("/predict", response_model=PredictionResponse)
def predict(user_id: int = Query(...)):
    key = f"user:{user_id}"
    if not r.exists(key):
        raise HTTPException(404, detail=f"No features for user {user_id}")
    data       = r.hgetall(key)
    count      = int(data["count"])
    avg_amount = float(data["avg_amount"])
    score      = model.decision_function([[count, avg_amount]])[0]
    is_fraud   = score < 0.0
    # Increment Prometheus counters
    TRANSACTIONS_TOTAL.inc()
    if is_fraud:
        FRAUDS_TOTAL.inc()
    return PredictionResponse(user_id, score, is_fraud)
```

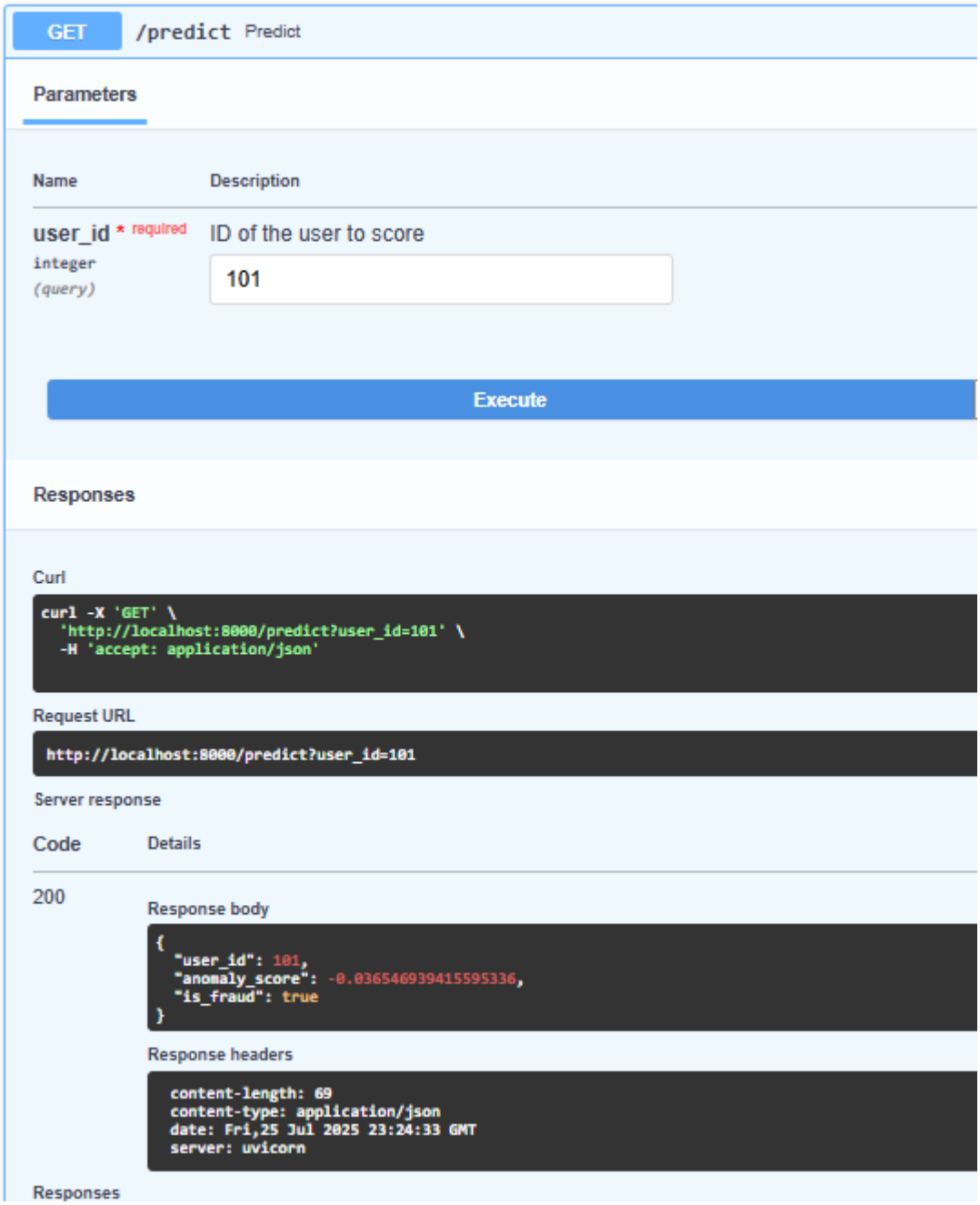


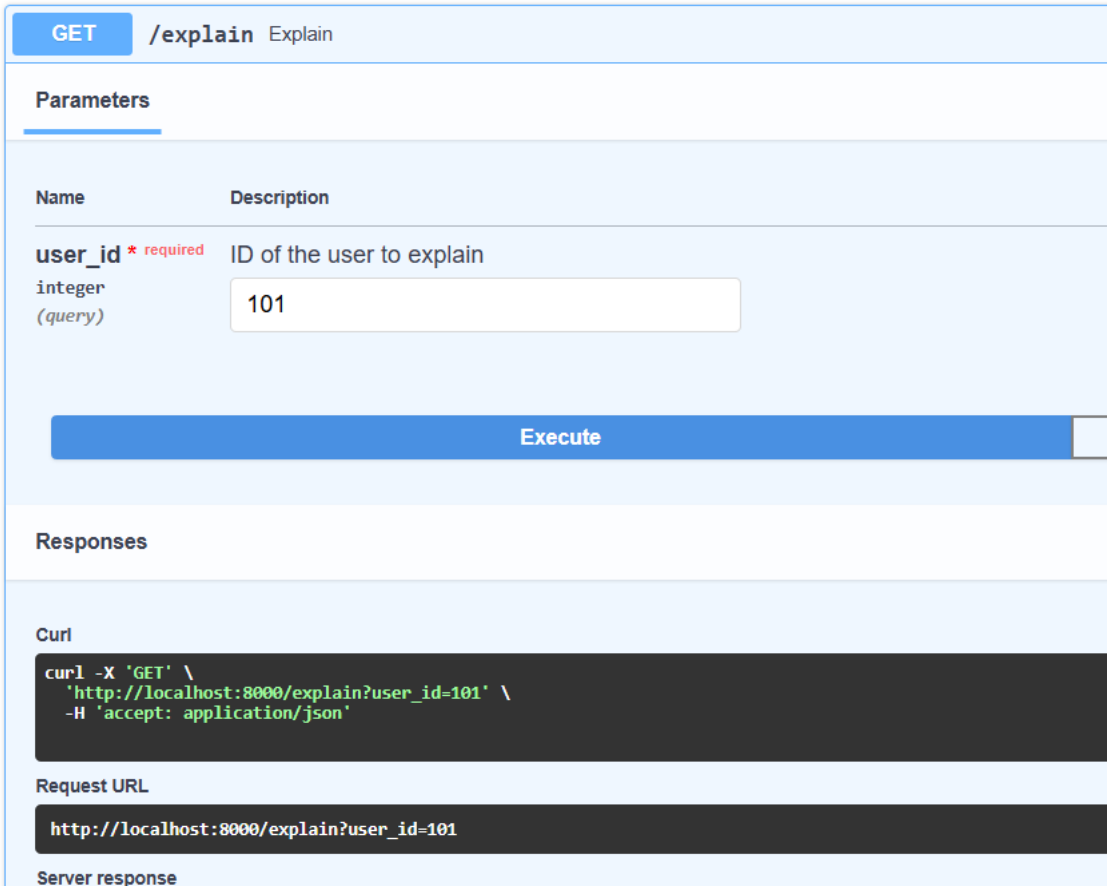
Figure 6 : Get / Predict Endpoint and Response Body

3. **`GET /explain` Endpoint**
   - **Purpose:** Provide a human-readable rationale for each fraud decision using an LLM.
   - **Flow:**
     1. Fetch features from Redis and prediction from the model.
     2. Construct a prompt JSON snippet (e.g., `{"count":10,"avg_amount":200.5,"score":-1.2}`).
     3. Send to a local Mistral-7B endpoint via LMStudio:

```
python
CopyEdit
resp = requests.post(
  f"{LMSTUDIO_URL}/v1/generate",
  json={"model": LMSTUDIO_MODEL, "prompt": prompt},
  timeout=5
```

```
)
explanation = resp.json()["text"]
```

4. Return `{"user_id":…, "explanation": explanation}`.



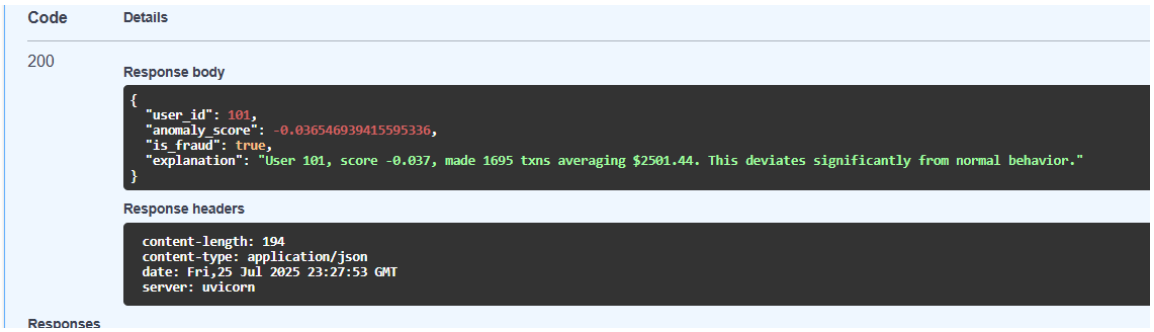


Figure 7 : Get / Explain Endpoint and Response Body

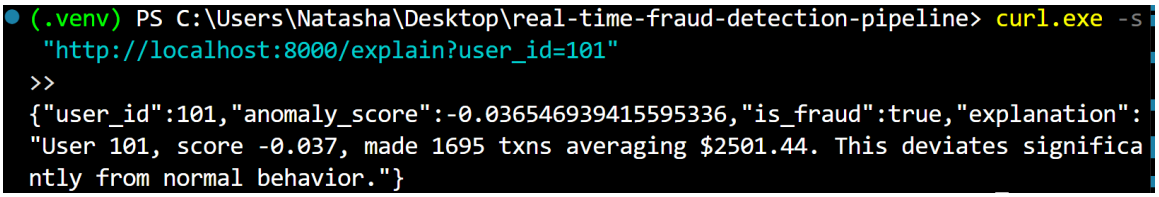


Figure 8 : Get / Explain Endpoint and Response Body by LLM

4. **Model Artifact Management**
   - **Storage:** Serialized as `models/isolation_forest.pkl` (or ONNX/hugging-face format).
   - **Reload on Startup:** The FastAPI app loads the model into memory when Uvicorn starts, ensuring sub-100 ms inference.
5. **Versioning & Rollouts**
   - Tag Docker images with semantic versions (`v1.0.0`, `v1.1.0`, …).
   - Maintain backward compatibility of `/predict` and `/explain` schemas.

With Phase 3 complete, you have a live API that serves both numerical anomaly scores and LLM-powered explanations, fully instrumented for Prometheus scraping and ready for integration into any client or dashboard.

➢ Phase 4: Observability & Alerting

In **Phase 4**, we instrument every layer with metrics, define alerting rules, and configure notification routing so that anomalies and system issues surface immediately.

1. **Prometheus Configuration**
   - **Scrape Targets** (`prometheus/prometheus.yml`):

```
yaml
CopyEdit
scrape_configs:
  - job_name: 'fastapi'
    static_configs:
      - targets: ['api:8000']
  - job_name: 'prometheus'
    static_configs:
      - targets: ['localhost:9090']
```

   - **Alerting Rules** (`prometheus/rules.yml`):

```
yaml
CopyEdit
groups:
  - name: FraudAlerts
    rules:
      # High traffic to /predict
      - alert: HighPredictTraffic
        expr:
sum(rate(http_requests_total{handler="/predict",status="2xx"}[5m])) > 10
        for: 2m
        labels:
          severity: warning
        annotations:
          summary: "High predict traffic ({{ $value }} req/s)"
          description: "The /predict endpoint is receiving an unusually high
request rate."

      # Elevated fraud-rate
      - alert: HighFraudRate
        expr: sum(increase(fraud_summary_is_fraud_total[5m]))
              / sum(increase(fraud_summary_total_transactions_total[5m])) >
0.05
        for: 2m
        labels:
```

```
        severity: critical
      annotations:
        summary: "High fraud rate (>{{ printf \"%.2f\" (100 * $value) }}%)"
        description: "More than 5% of transactions were flagged as fraud in
the last 5 minutes."
```

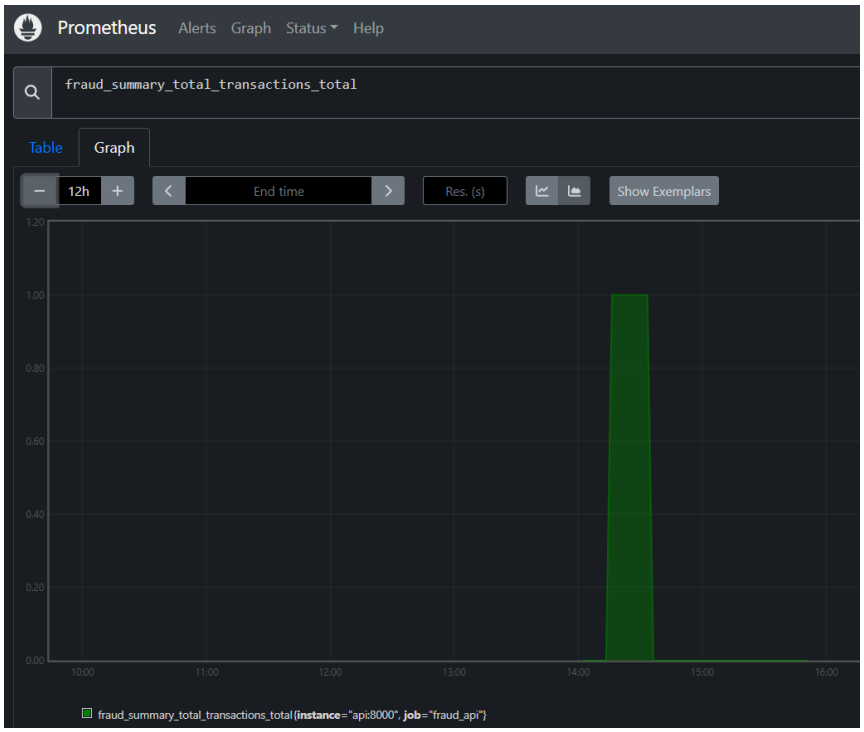


Figure 9 : Prometheus Graph

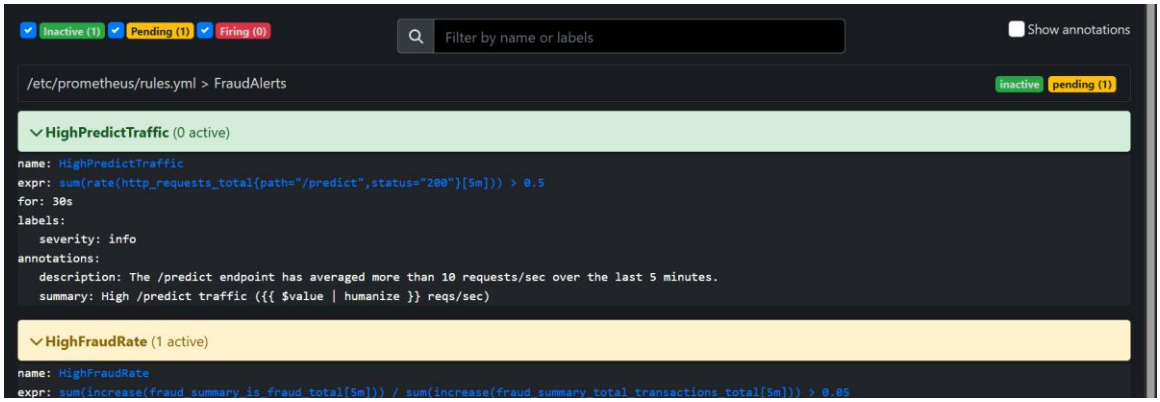


Figure 10 : Prometheus Alerts showing Inactive and Pending

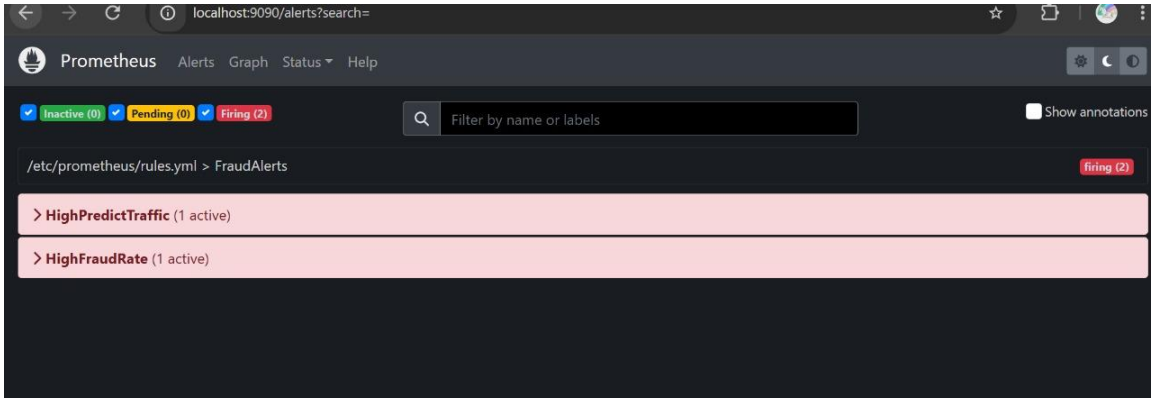


Figure 11 : Prometheus Alerts Firing High Predict Traffic and High Fraud Rate

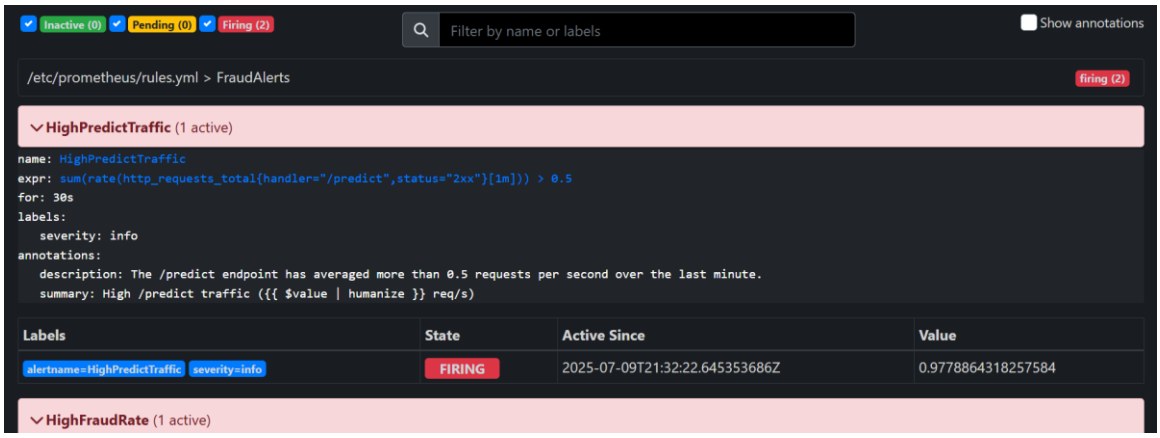


Figure 12 : Prometheus Alerts Firing High Predict Traffic showing 0.5 requests per second

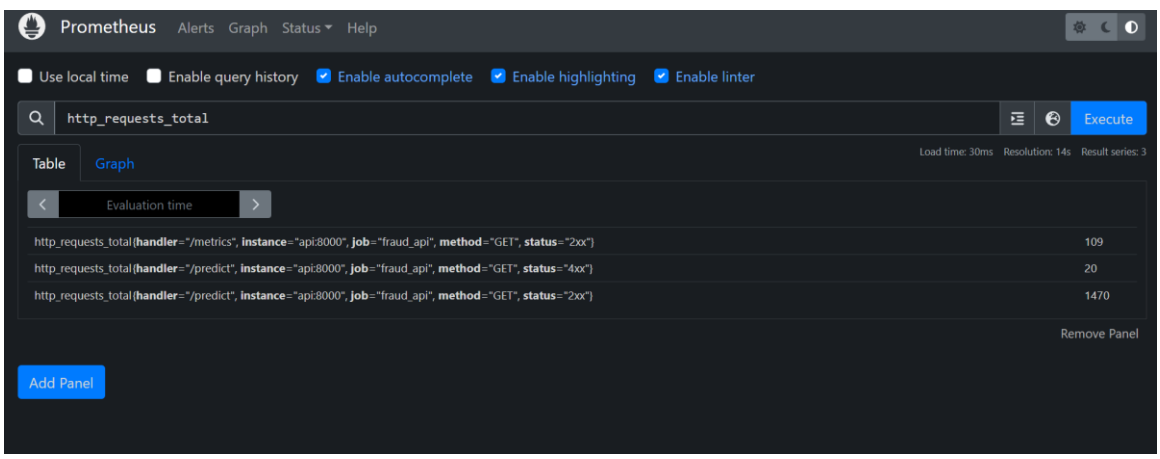


Figure 13 : Prometheus Alerts Https_Requests_Total

2. **Grafana Dashboards**
   - **Data Source:** Prometheus at `http://prometheus:9090`.
   - **Panels:**
     - **Request Throughput**:

```promql

sum(rate(http_requests_total{handler="/predict"}[1m]))

```

- **Fraud Rate**:

  ```promql


  sum(increase(fraud_summary_is_fraud_total[5m]))
  / sum(increase(fraud_summary_total_transactions_total[5m]))
  ```

- **Latency Histogram**: 95th-Percentile Inference Latency

  ```promql

  histogram_quantile(

       0.95,

       sum by (le) (

         rate(http_request_duration_seconds_bucket[5m])

         )

        )

       ```

> **Note:** We dropped the `{handler="/predict"}` filter because our buckets aren’t labeled with `handler`. If you do have that label in your setup, you can narrow to only prediction calls:

```promql

   histogram_quantile(0.95,sum by (le) (

     rate(http_request_duration_seconds_bucket{handler="/predict"}[5m])

                       )

                     )

  ```

Figure 14 : Inference Latency, Fraud Rate(5m), Total Transactions

3. Visualization Dashboard

A Streamlit app (`app.py`) connects directly to the Postgres `transactions_raw` and `fraud_summary` tables to render:

- Fraud-by-country bar charts

- Top-risk-user tables

- Fraud-over-time line charts

- LLM explanations alongside scores

Run locally with:

```
Bash
streamlit run app.py
```

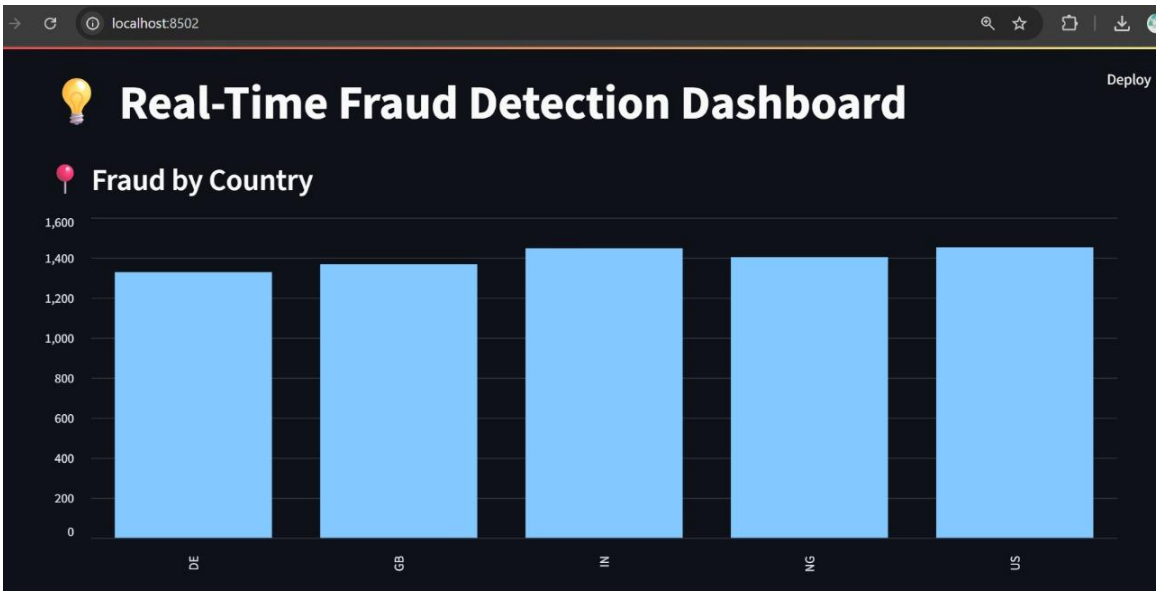


Figure 15 : Streamlit Dashboard Fraud by Country

Top-Risk Users

	user_id	anomaly_score	transaction_count
0	101	-0.0422	1731
1	102	-0.0422	1808
2	103	-0.0217	1849
3	104	-0.0163	1611

Figure 16 : Streamlit Dashboard Top-risk Users

Figure 16: Streamlit Dashboard Fraud Over Time

4. **Alertmanager Configuration**
   - **Routes & Receivers** (`alertmanager/config.yml`):

```
yaml
CopyEdit
global:
  resolve_timeout: 5m
  smtp_smarthost:     {{ env "SMTP_SMARTHOST" }}
  smtp_from:          {{ env "SMTP_FROM" }}
  smtp_auth_username: {{ env "SMTP_AUTH_USERNAME" }}
  smtp_auth_password: {{ env "SMTP_AUTH_PASSWORD" }}

route:
  receiver: console
  routes:
    - receiver: email-main
      matchers:
        - 'severity=~"warning|critical"'

receivers:
  - name: console
    webhook_configs:
      - url: "http://localhost:5001/"
  - name: email-main
    email_configs:
      - to: team-alerts@yourdomain.com
        send_resolved: true
        headers:
          Subject: '[Fraud-Pipeline] {{ .CommonLabels.alertname }} - {{
.Status }}'
        html: |
          <p><b>{{ .CommonAnnotations.summary }}</b></p>
          <p>{{ .CommonAnnotations.description }}</p>
          <p>Occurred at: {{ .StartsAt }}</p>
```

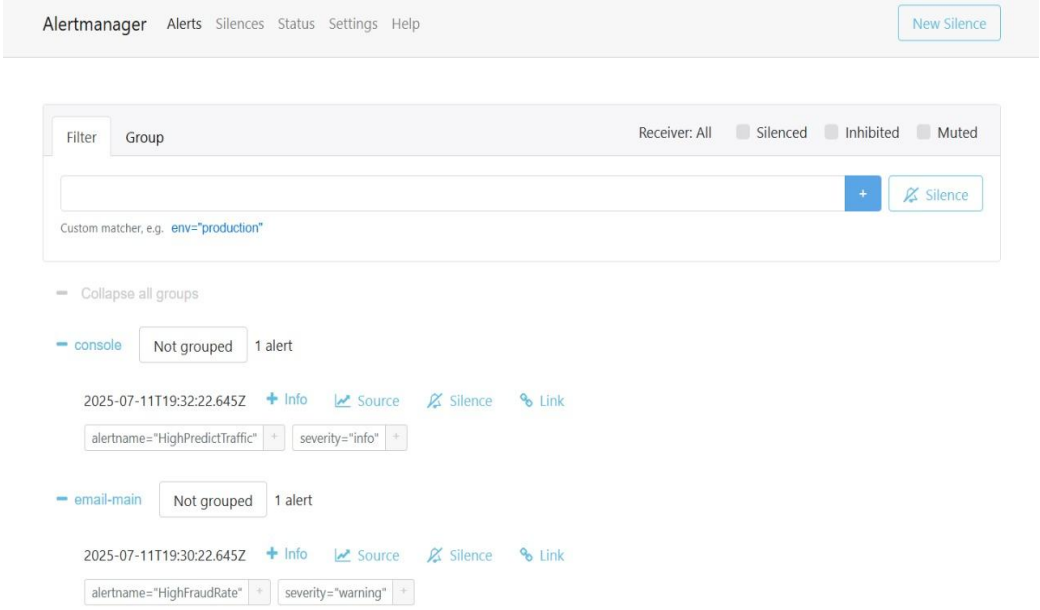


Figure 17 : Alert Manager showing alert notification

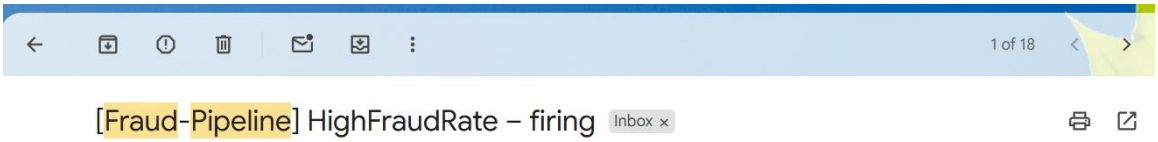


Figure 18 : Receiving Email alerts

5. **End-to-End Alert Test**
   - **Generate Load:**

```
bash
CopyEdit
for i in {1..600}; do
  curl -s http://localhost:8000/predict?user_id=101 > /dev/null
done
```

   - **Verify in Prometheus UI:** Alerts → *HighPredictTraffic* → Pending → Firing
   - **Check Alertmanager UI:** http://localhost:9093 → shows both alerts
   - **Receive Email:** team-alerts@yourdomain.com inbox

6. 🛠️ **Troubleshooting**

- **`/metrics` target is DOWN**

1. In Prometheus UI, go to **Status → Targets** and check the scrape error.

2. Verify your FastAPI container is running:

```bash
docker-compose ps api
```

3. Test directly:

```bash
curl http://localhost:8000/metrics
```

4. Fix any binding or network issues, then restart Prometheus:

```bash
docker-compose restart prometheus
```

- **Producer can’t resolve `kafka:9092`**
- **Symptom:** Errors like “Failed to resolve ‘kafka:9092’” in your producer logs.
- **Fix:** When running `producer.py` on your host, point it at the host-exposed broker port:

```bash
export KAFKA_BOOTSTRAP_SERVERS=localhost:29092
python producer.py
```

- Or run the producer inside Docker so it can resolve the service name:

```bash
docker-compose exec api python producer.py
```

- **Consumer group lag builds up**

1. Check offsets:

```bash
docker exec -it kafka \
kafka-consumer-groups --describe --group fraud-consumer-group --bootstrap-
server kafka:9092
```

2. Ensure your consumer is running and not failing on startup.

Phase 4 ensures that not only is your pipeline performing real-time fraud detection, but you also have full visibility into its performance and immediate notification channels when things go awry.

➢ Phase 5: CI/CD, Automated Testing & ETL Cleanup

In **Phase 5**, we close the loop on delivery and operations by automating code validation, backups, and log maintenance.

5.1 GitHub Actions CI (`.github/workflows/ci.yml`)

1. **Triggers**

```yaml
yaml
CopyEdit
on:
  push:
    branches: [ main ]
  pull_request:
    branches: [ main ]
```

2. **Services**

```yaml
yaml
CopyEdit
services:
  postgres:
    image: postgres:15-alpine
    ...
  redis:
    image: redis:6.2
    ...
  zookeeper:
    image: confluentinc/cp-zookeeper:6.2.1
    ...
  kafka:
    image: confluentinc/cp-kafka:6.2.1
    ...
  schema-registry:
```

```
    image: confluentinc/cp-schema-registry:7.5.0
    ...
```

3. **Steps**

```
yaml
CopyEdit
steps:
  - uses: actions/checkout@v3

  - name: Set up Python
    uses: actions/setup-python@v4
    with:
      python-version: "3.11"

  - name: Install dependencies
    run: |
      pip install --upgrade pip
      pip install -r requirements_api.txt
      pip install -r requirements_dev.txt

  - name: Wait for services to be healthy
    run: sleep 20

  - name: Lint with flake8
    run: flake8 .

  - name: Run pytest
    env:
      DATABASE_URL:
postgresql://postgres:password@localhost:5432/fraud_db
      REDIS_HOST:               localhost
      REDIS_PORT:               6379
      KAFKA_BOOTSTRAP_SERVERS:  localhost:9092
      SCHEMA_REGISTRY_URL:      http://localhost:8081
    run: pytest --maxfail=1 --disable-warnings -q

  - name: Build API Docker image
    run: docker compose build api
```

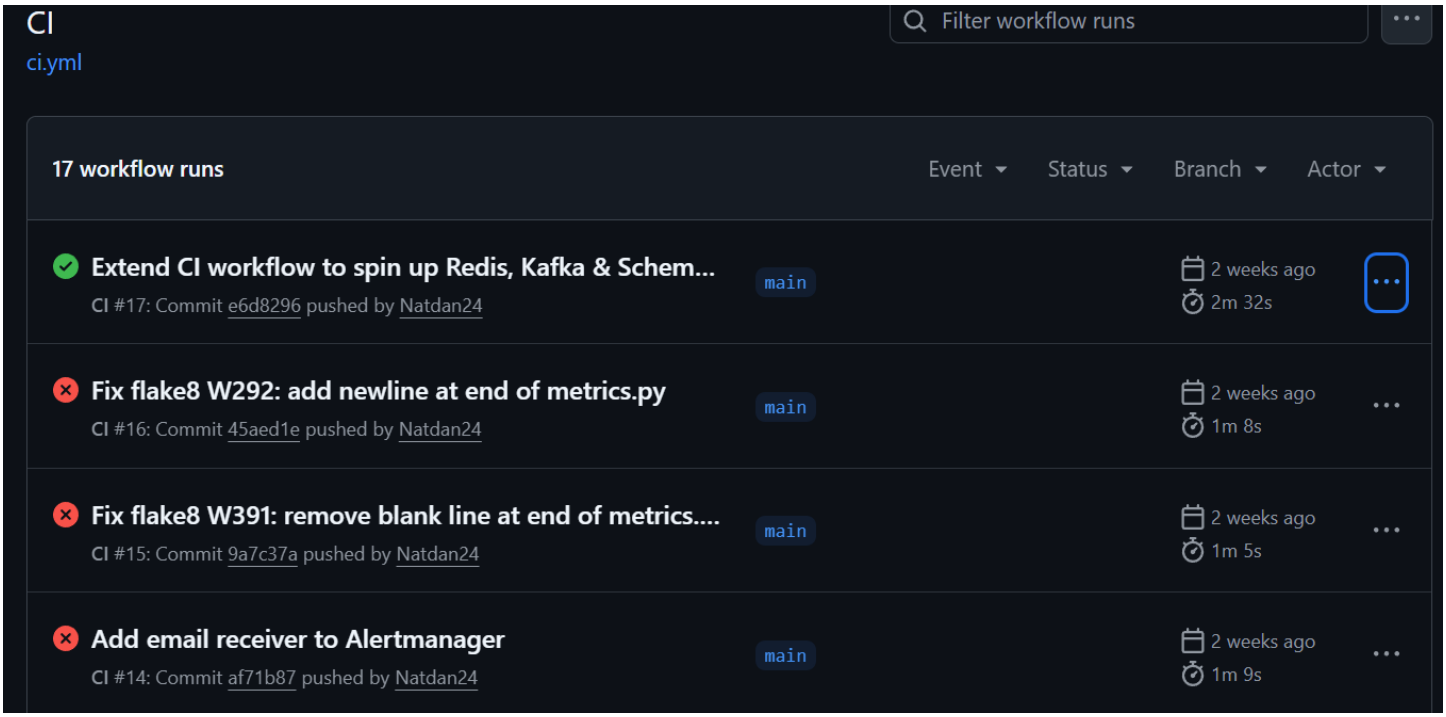


Figure 19 : CI.YML Workflow run

5.2 Nightly Backup (`.github/workflows/nightly-backup.yml`)

1. **Trigger**

```
yaml
CopyEdit
on:
  schedule:
    - cron: "0 2 * * *"  # daily at 02:00 UTC
  workflow_dispatch:     # manual trigger
```

2. **Backup Steps**

```
yaml
CopyEdit
jobs:
  backup:
    runs-on: ubuntu-latest
    steps:
      - uses: actions/checkout@v3

      - name: Install PostgreSQL client
        run: sudo apt-get update && sudo apt-get install -y postgresql-client

      - name: Run backup script
        env:
          DATABASE_URL: postgresql://postgres:password@localhost:5432/fraud_db
        run: |
          chmod +x backups/backup_fraud_summary.sh
          backups/backup_fraud_summary.sh

      - name: Upload CSV artifact
        uses: actions/upload-artifact@v4
        with:
```

```
          name: fraud_summary_${{ github.run_number }}
          path: backups/fraud_summary.csv
```

5.3 Daily ETL Cleanup (`.github/workflows/etl-cleanup-workflow.yml`)

1. **Trigger**

```
yaml
CopyEdit
on:
  schedule:
    - cron: '0 3 * * *'       # daily at 03:00 UTC
  workflow_dispatch:       # manual run
```

2. **Service Startup**

```
yaml
CopyEdit
- name: Start minimal services
  run: |
    docker compose up -d redis kafka zookeeper
```

3. **Health Checks**

```
bash
CopyEdit
# inside a run step
for i in {1..10}; do
  docker exec redis redis-cli ping && break
  sleep 3
done

for i in {1..10}; do
  docker exec kafka kafka-topics --bootstrap-server kafka:9092 --list && break
  sleep 3
done
```

4. **Run Cleanup Script**

```
yaml
CopyEdit
- name: Execute ETL cleanup
  run: |
    chmod +x etl_cleanup.sh
    ./etl_cleanup.sh
```

5. **Archive Logs**

```
yaml
CopyEdit
- name: Upload compressed logs
  uses: actions/upload-artifact@v4
  with:
    name: etl-logs-${{ github.run_number }}
    path: |
      producer.log.gz
      consumer.log.gz
      backup.log.gz
```

6. **Tear Down**

```
yaml
CopyEdit
- name: Tear down Docker services
  run: docker compose down
```

With these workflows in place, every commit is validated, data snapshots are taken nightly, and logs are cleaned and archived each morning—ensuring both development velocity and operational health.

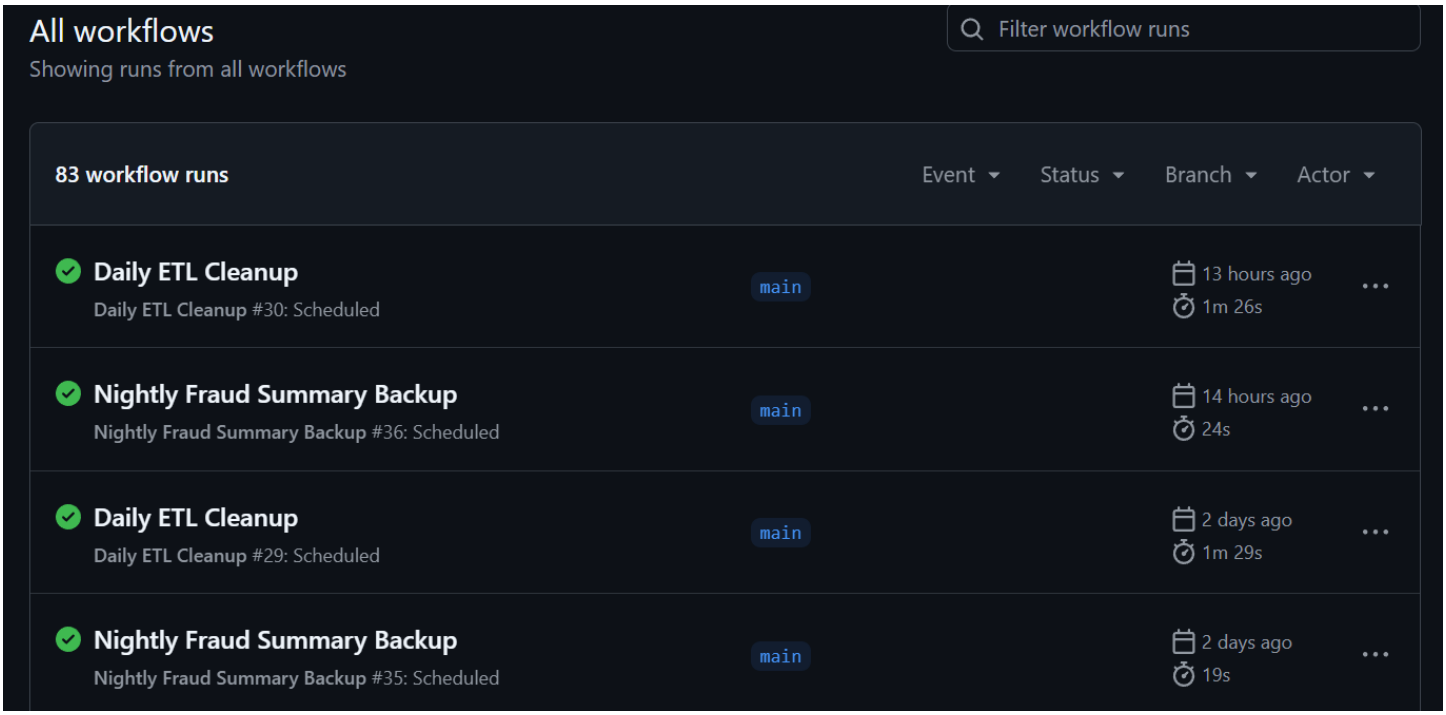


Figure 20 : Daily ETL Cleanup and Nightly Fraud Summary Backup Workflow run

5.Conclusion

In this project, we've built a robust, end-to-end real-time fraud-detection pipeline that:

- **Ingests** high-velocity transaction streams via Kafka with Avro serialization.
- **Processes** and **aggregates** data in real time, storing raw events in PostgreSQL and maintaining live feature hashes in Redis.
- **Serves** anomaly scores through a low-latency FastAPI `/predict` endpoint, complemented by an LLM-powered `/explain` for human-readable rationales.
- **Monitors** service health, request throughput, and fraud rates with Prometheus, and **alerts** on anomalies through Alertmanager to both console and SMTP e-mail.
- **Automates** quality checks and integration tests in GitHub Actions, as well as nightly data backups and daily ETL cleanup routines.

This architecture demonstrates best practices in data engineering, machine learning operations, and DevOps. It provides a solid foundation for further enhancements—whether that's adding advanced models, richer dashboards in Grafana, multi-channel notification support, or deploying the entire stack to Kubernetes.

By following the phases outlined above, you can reproduce, extend, and customize this pipeline to fit real-world fraud-detection use cases, ensuring both accuracy in anomaly detection and reliability in production environments.

Thank you for exploring this pipeline—here's to catching fraud faster and smarter!

6.Glossary

Term	Definition
Avro	A binary serialization format that uses schemas (JSON-based) for efficient, language-agnostic data encoding and schema evolution.
Kafka	A distributed event streaming platform used for high-throughput, fault-tolerant message ingestion and processing.
ZooKeeper	A coordination service used by Kafka brokers for leader election and cluster metadata management.
Confluent Schema Registry	A service for managing Avro schemas centrally, ensuring producers and consumers use compatible schema versions.
PostgreSQL	An open-source relational database used here for raw transaction storage, aggregates, and historical fraud summaries.
Redis	An in-memory key-value store used as a feature store for low-latency retrieval of per-user aggregates.
FastAPI	A modern, high-performance Python web framework for building APIs, used here to serve the `/predict` and `/explain` endpoints.
Isolation Forest	An unsupervised machine-learning algorithm for anomaly detection, isolating outliers based on random feature splits.
Mistral-7B	A large language model used for generating natural-language explanations of fraud decisions via LMStudio.
Prometheus	An open-source monitoring and alerting toolkit that scrapes metrics from services and evaluates alerting rules.
Grafana	A visualization platform that builds dashboards and graphs from Prometheus (and other) data sources.

Term	Definition
Alertmanager	A component of the Prometheus ecosystem that routes and manages alerts, supporting multiple notification channels (email, Slack, webhooks).
Flake8	A Python tool for enforcing coding style and checking for common errors (PEP 8 compliance).
GitHub Actions	A CI/CD platform integrated with GitHub that automates workflows like linting, testing, and Docker builds on code changes.
Docker Compose	A tool for defining and running multi-container Docker applications, orchestrating services like Kafka, Redis, Postgres, and the API in one config.
CI/CD	Continuous Integration and Continuous Deployment/Delivery—a practice of automatically building, testing, and deploying code changes.