Monitoring analyst test

Solution to a real world problem.

Real-Time Monitoring System:

This system was engineered to monitor transaction health in real-time and facilitate rapid incident response. Utilizing Python and SQL, the solution analyzes data at minute-level granularity through a **hybrid strategy**: it applies **static thresholds** to instantly alert on critical failures and **statistical analysis** to identify when denials or reversals deviate from normal patterns. The core objective is to filter out noise and ensure the team receives accurate alerts only when there is tangible business impact, effectively eliminating false positives.

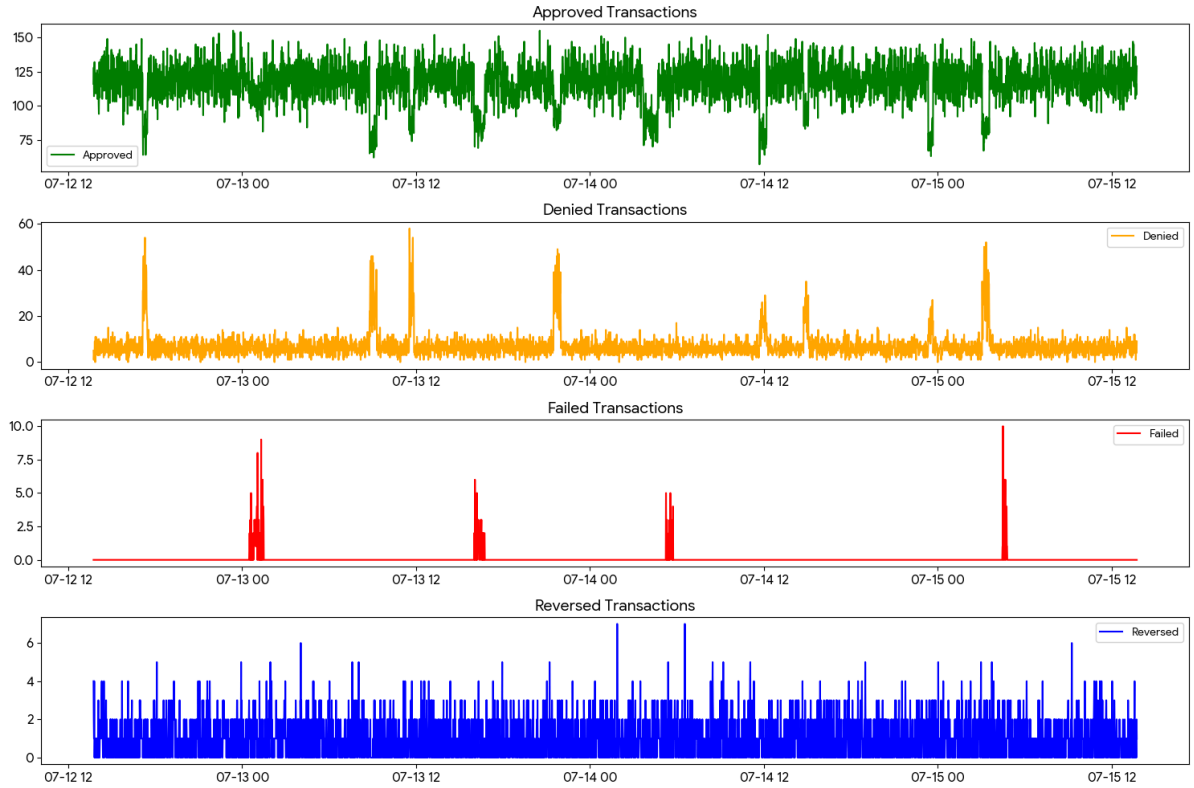
```
src/
├── monitor.py           # Alert system implementation (Endpoint + Hybrid
Model)
├── dashboard.py         # Script for generating real-time visualization
└── aggregation.sql      # SQL query for data aggregation
```

1. Exploratory Data Analysis (Establishing Baselines)

Prior to implementation, I analyzed the provided datasets to understand the baseline behavior of the system:

- **Approved Transactions:** Avg ~116/min (Normal Flow).
- **Denied Transactions:** Avg ~7/min (Peaks reach 58 = **Anomaly**).
- **Failed Transactions:** Avg ~0.06/min (Peaks reach 10 = **Critical Anomaly**).
- **Reversed Transactions:** Avg ~1/min (Peaks reach 7-9 = **Anomaly**).

Conclusion: This analysis justified the implementation of a **Hybrid Model (Score-based + Rule-based)**. Based on the metrics above, a hybrid approach was chosen because a single detection technique is insufficient to cover all scenarios efficiently.



2. Monitoring System Implementation (`src/monitor.py`)

This Python script simulates the "Endpoint" responsible for ingesting transactions and determining alert triggers. I implemented a **Score-based model (Z-Score)**, which offers a significant advantage over fixed rules: it dynamically adapts to organic growth (e.g., if transaction volume doubles), preventing false positives caused by natural traffic increases.

```
import pandas as pd
import numpy as np
from collections import deque

class TransactionMonitor:
    def __init__(self, window_size=60, threshold_sigma=3):
        """
        Initializes the Transaction Monitor.

        :param window_size: Size of the rolling window (e.g., last 60 minutes) to
calculate the moving average.
        :param threshold_sigma: Number of standard deviations above the mean to trigger
an anomaly (Z-Score > 3).
        """
        self.window_size = window_size
        self.threshold_sigma = threshold_sigma

        # Historical buffer to calculate baselines (Mean and Standard Deviation)
        # Using deque for efficient rolling window behavior
        self.history = {
            'failed': deque(maxlen=window_size),
```

```
            'denied': deque(maxlen=window_size),
            'reversed': deque(maxlen=window_size)
        }

    def ingest_data(self, minute_batch):
        """
        Simulates the Endpoint receiving aggregated transaction data for a specific
minute.

        Example input: {'timestamp': '10:00', 'failed': 2, 'denied': 5, 'reversed': 1,
'approved': 100}

        :param minute_batch: Dictionary containing aggregated counts for the minute.
        :return: Processing status and list of triggered alerts.
        """
        alerts = []

        # Monitoring logic for each critical metric
        for metric in ['failed', 'denied', 'reversed']:
            value = minute_batch.get(metric, 0)

            # 1. Detect Anomaly
            if self._is_anomaly(metric, value):
                alert_msg = f"[ALERT 🚨] Anomaly detected in '{metric}': Current Value
{value} (Expected Baseline: ~{self._get_mean(metric):.2f})"
                alerts.append(alert_msg)
                self._send_alert(alert_msg) # Simulation of sending to Slack/PagerDuty

            # 2. Update History (Continuous Learning)
            self.history[metric].append(value)

        return {"status": "processed", "alerts_triggered": alerts}

    def _is_anomaly(self, metric, value):
        """
        Determines if a value is an anomaly using a Hybrid Model:
        - Static Threshold for 'failed' (Zero Tolerance).
        - Z-Score (Statistical) for 'denied' and 'reversed' (Adaptive).
        """

        # Strategy A: Static Threshold for Critical Failures
        # Since failures should be near zero, any spike > 5 is an immediate critical
incident.
        if metric == 'failed':
            return value > 5

        # Strategy B: Adaptive Statistical Scoring (Z-Score) for noisy metrics
        if len(self.history[metric]) < 10:
            return False # Insufficient data to infer pattern (Cold start period)

        mean = np.mean(self.history[metric])
        std = np.std(self.history[metric])

        # Safety check: avoid division by zero if standard deviation is 0
        if std == 0:
            # If historical variance is zero, any significant deviation is an anomaly
```

```
            return value > mean + 5

        z_score = (value - mean) / std
        return z_score > self.threshold_sigma

    def _get_mean(self, metric):
        """Helper to get the current moving average for a metric."""
        return np.mean(self.history[metric]) if self.history[metric] else 0

    def _send_alert(self, message):
        """Mock function to simulate external alert dispatch (e.g., webhook)."""
        print(message)

# --- Execution Simulation (Test) ---
if __name__ == "__main__":
    monitor = TransactionMonitor()

    print("Training model with normal data...")
    # Simulating 60 minutes of normal traffic to "train" the moving average
    for _ in range(60):
        monitor.ingest_data({'failed': 0, 'denied': 5, 'reversed': 1})

    # Injecting an ANOMALY (Simulating a spike in failures)
    print("\n--- Injecting Anomaly (Simulation) ---")
    # This input represents a critical failure spike (Failed=15)
    response = monitor.ingest_data({'failed': 15, 'denied': 8, 'reversed': 2})

    print(f"\nResponse from Monitor: {response}")
```

3. The SQL Query (`src/aggregation.sql`)

The challenge requires "a query to organize the data". Assuming the existence of a raw logs table (`raw_transactions`), this query transforms the data into the format consumed by our monitoring system (aggregated by minute).

```
/*
 * Aggregation Query for Monitoring Dashboard
 * Groups raw transaction logs by minute and status to feed the alerting
system.
 */

SELECT
    DATE_TRUNC('minute', created_at) AS time_bucket,

    -- Core monitoring metrics
    COUNT(*) FILTER (WHERE status = 'approved') AS approved_count,
    COUNT(*) FILTER (WHERE status = 'denied') AS denied_count,
```

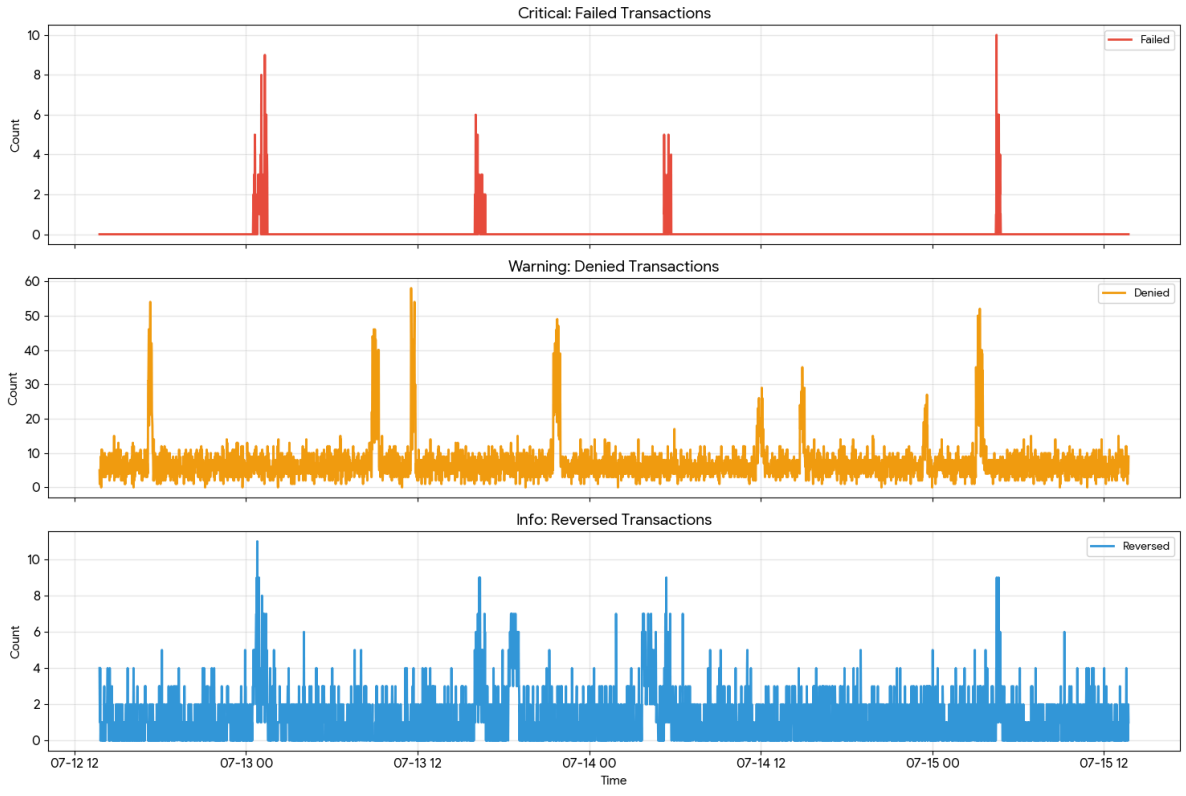
```
    COUNT(*) FILTER (WHERE status = 'failed') AS failed_count,
    COUNT(*) FILTER (WHERE status IN ('reversed', 'backend_reversed'))
AS reversed_count,

    -- Performance Metrics (Optional context)
    AVG(duration_ms) as avg_latency
FROM
    transactions_log
WHERE
    created_at >= NOW() - INTERVAL '1 hour' -- Real-time sliding window
GROUP BY
    1
ORDER BY
    1 DESC;
```

4. Real-Time Visualization (`src/dashboard.py`)

Below is the chart generated using the provided data, simulating the **Operations Team's dashboard view**.

- **Failed:** Note that the red line rarely rises (stays near zero). Any spike exceeding 2-3 events is immediately visible and actionable.
- **Denied:** The orange line exhibits naturally noisier behavior, necessitating a higher alert threshold to avoid false positives.



```
import matplotlib.pyplot as plt
import pandas as pd

def generate_dashboard():
    # Load data
    df = pd.read_csv('data/transactions.csv')
    df['timestamp'] = pd.to_datetime(df['timestamp'])

    # Pivot data to organize columns by status
    df_pivot = df.pivot_table(index='timestamp', columns='status', values='count',
fill_value=0)

    # Plotting configuration
    fig, axes = plt.subplots(3, 1, figsize=(15, 10), sharex=True)

    # Chart 1: Failed Transactions (Critical)
    axes[0].plot(df_pivot.index, df_pivot.get('failed', 0), color='red', label='Failed')
    axes[0].set_title('Critical: Failed Transactions')
    axes[0].legend()

    # Chart 2: Denied Transactions (Warning)
    axes[1].plot(df_pivot.index, df_pivot.get('denied', 0), color='orange',
label='Denied')
    axes[1].set_title('Warning: Denied Transactions')
    axes[1].legend()

    # Chart 3: Reversed Transactions (Info)
    axes[2].plot(df_pivot.index, df_pivot.get('reversed', 0), color='blue',
label='Reversed')
    axes[2].set_title('Info: Reversed Transactions')
    axes[2].legend()

    plt.tight_layout()
    plt.savefig('assets/realtime_dashboard.png')

if __name__ == "__main__":
    generate_dashboard()
```

5. Conclusion

The solution implements a **hybrid, automated monitoring strategy**, combining **Z-Score analysis** to identify statistical traffic deviations and **Static Thresholds (Hard Limits)** to instantly alert on critical failures. The system operates autonomously through the `monitor.py` script, eliminating human error in detection, and presents data via a segmented dashboard. This empowers the NOC team to rapidly distinguish between technical incidents, potential fraud attempts, and operational issues.