Project Report

Monitoring Intelligence Analyst Test Case

Author
David Londres

February 16, 2026

Contents

Introduction

This report details a thorough solution proposed for the CloudWalk Intelligence Analyst Test Case. The intended purpose is for this report to be **AI free**, so it may contain some misspellings and grammatical errors.

AI was heavily used to generate the code. System design decisions were made by me, although I accepted some advice from the AI. I will highlight AI interventions throughout the report, but in summary, *concept awareness* is now way more valuable than writing down code and I leveraged the tool to speed up the coding process. I believe this is the future of software engineering.

This report will follow the same logical steps I took when I read the case. The next chapter is dedicated to understanding the domain of the issue at hand. The last chapter will be dedicated to explaining the solution proposed and the engineering hurdles and decisions I had to make.

Chapter 1

Domain Understanding

1.1 Getting my hands dirty

The technical case encompasses a domain in which I have little to no experience: anomaly detection in transactional data. My first step was to try to understand the domain in which the data comes from. Terms like POS and Auth Codes were unknown to me, but anomaly detection, which is fundamentally a pattern recognition problem, and time series analysis were not.

1.1.1 EDA

The EDA was done using Jupyter Notebook. The two POS datasets were loaded into a pandas DataFrame and the AI generated the code to perform some data cleaning and sanity checks. I reviewed the code and saw that the data was already clean and formatted in a way that was easy to work with, gladly no data engineering was needed.

Checkout data showed POS aggregated by hour, with the number of transactions and the total amount. By my understanding, that imples seasonability and my first instinct was to plot the volumetry along the time.

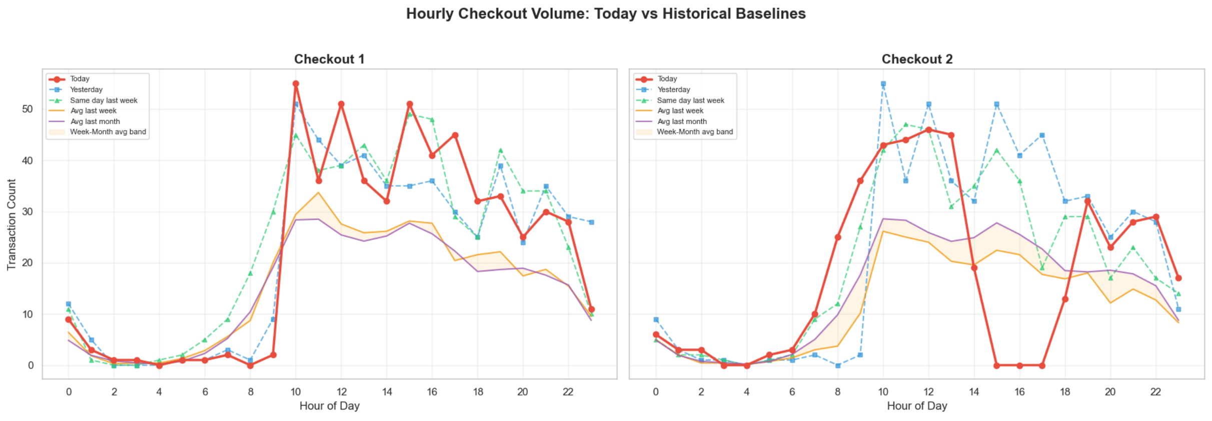

Figure 1.1: Checkout Volumetry by Hour

Aha. I see seasonality and I can see a system outage, in fact, there are two distinct outages.

- The first one is a brief outage that happens in checkout_1.csv from 07:45 to approximately 08:15.
- The second one is a critical system outage that lasted for a long time and happens in checkout_2.csv from 15:00 to approximately 17:00.

Based on that I asked the AI to generate the sql query to detect anomalies, and this was the result:

```sql
SELECT
    time,
    hour,
    today,
    avg_last_week,
    avg_last_month,

    -- Simple percentage deviation
    CASE
        WHEN avg_last_week = 0 THEN NULL
        ELSE ((today - avg_last_week) * 1.0 / avg_last_week) * 100
    END AS pct_deviation_week,

    -- Naive anomaly flag
    CASE
        WHEN avg_last_week = 0 THEN 0
        WHEN ABS((today - avg_last_week) * 1.0 / avg_last_week) > 0.5 THEN 1
        ELSE 0
    END AS is_naive_anomaly,

    -- Naive severity
    CASE
        WHEN avg_last_week = 0 THEN 'UNKNOWN'
        WHEN ABS((today - avg_last_week) * 1.0 / avg_last_week) > 1.0 THEN '
    CRITICAL'
        WHEN ABS((today - avg_last_week) * 1.0 / avg_last_week) > 0.5 THEN '
    WARNING'
        ELSE 'NORMAL'
    END AS naive_severity

FROM checkout_data
ORDER BY hour
```

Listing 1.1: Naive Anomaly Detection Query

This implementation is naive because it only takes standard deviation into consideration and variance is completely disregarded. The result is that many hours are flagged as anomalies, when they in fact are not. If this was a real life scenario, this would lead to *alert fatigue*.

```
=== checkout_1 ===

   Flagged: 14 / 24 hours
   Severity counts:
naive_severity
WARNING     11
NORMAL      10
CRITICAL     3


=== checkout_2 ===

   Flagged: 20 / 24 hours
   Severity counts:
naive_severity
WARNING     11
CRITICAL     9
NORMAL       4
```

Listing 1.2: Naive Anomaly Detection Results

I then explained this context to the AI and asked for a better implementation that took into consideration variance. And it then proposed a multi-layered approach, which I found really interesting.

The improved query is too big for this document so I will *link* to the file and include it as an additional artifact. The following graph is the result of the query.

The function combines these signals into a final severity score:

Layer	What it catches	Strength
Statistical Z-Score	Values far outside expected range	Adapts to natural variance
Volume-Aware Rules	Context-dependent thresholds	Handles low-traffic hours
Drop-to-Zero / Spike	Outages and attacks	Catches catastrophic events

Table 1.1: Proposed Anomaly Detection Layers

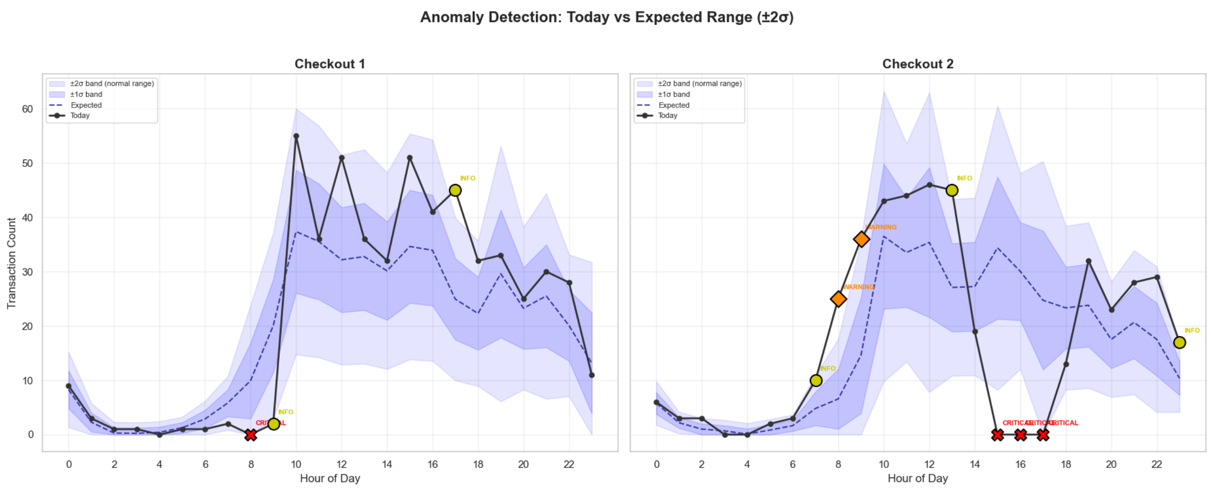

Figure 1.2: Layered Query Result

Severity	Condition	Meaning
CRITICAL ●	Confirmed Outage (0 transactions)	System down
WARNING ●	Massive Spike or Multi-layer anomaly	Major incident
INFO ●	Statistical anomaly only	Investigate
NORMAL	Everything looks fine	All clear

Table 1.2: Anomaly Severity Levels

The labels are now way more accurate given the granularity and matched my initial hypothesis. One might notice, however, that the query does not catch the sudden drop in transactions that happens in checkout_2.csv from 14:00 until system failure at 15:00. I still tested a hypothesis about adding a 4th layer to the query, that being the slope component (derivative of the time series), calculated as the difference between the current hour and the previous hour. This, however, adds risk of bad generalization since it's added complexity to a tiny dataset and will generate a lot of INFO labels. One way to be more sure is to get the raw hourly volumetry instead of the pre aggregated data. This way we can compute the proper standard deveations per hour and know the true distribution shape.

Chapter 2

The fun part

2.0.1 First considerations

Alert incident in transactions: Implement the concept of a simple monitoring with real time alert with notifications to teams. (citation)

The problem is pretty much open ended except with the requirement of 1 recommending endpoint, an alerting policy, a graph and a notification channel. Non-functional requirements are not specified, so I took some liberty and ran the extra mile to mimetize a production enviromment, with an API required to handle high throughput and messages in real time comming from a streaming service (the real time aspect of the design). A purely asynchronous design would not fit the intricacies of the problem domain.

The key features of the solution are:

- A pub/sub architecture to handle high throughput and real time messages.
- Profiling in both Producer and Consumer with Prometheus
- Distributed tracing in both Producer and Consumer with Jaeger
- A alerting policy with a simple threshold
- A notification channel with a simple webhook
- A Grafana dashboard to visualize the metrics, traces, logs and the anomaly rate

2.0.2 Design decisions

2.0.3 Implementation

2.0.4 Results