Portfolio Project 2

Data Validation and Continuous Improvement in Action

Executive Summary

While working on our company's data pipelines, I encountered a critical issue that risked our financial integrity and our professional relationship with a key client. While generating financial reports, I observed inconsistencies in the data—figures weren't aligning as expected, totals were off, and there were noticeable gaps in coverage. These discrepancies weren't just anomalies; they were red flags. After a deeper investigation, I discovered missing data in our internal tables and even inconsistencies in the source files provided by the client. This realisation sparked a complex, technically challenging journey into data validation, error tracing, and system improvement.

To resolve this, I designed and implemented a Python-based application that validates and compares data from both our internal systems and external sources. The app identifies missing rows, empty mandatory fields, mismatched date ranges, and column discrepancies. Once the root issues were identified, I collaborated with the relevant teams to refine ingestion processes, establish more effective data practices, and ultimately introduce automated alerts to prevent future data corruption. What began as a minor reporting inconsistency evolved into a complete improvement of a critical part of our data pipeline, ultimately saving time, money, and reputation.

Introduction

In data engineering, problems rarely present themselves loudly—they whisper. That whisper came in the form of slightly off numbers in a routine report. At first glance, it seemed minor, maybe a formatting issue or a late data sync. But as I dug deeper, that whisper became a warning siren. This wasn't just a formatting glitch. The discrepancies pointed to a more concerning issue: data loss and inconsistency between what the client sent us and what we stored on our UK servers.

The client in question, Senpiper, is responsible for submitting periodic datasets, which we then integrate into our reporting systems. These reports feed directly into financial summaries,

performance metrics, and client-facing insights. When these numbers are off—even slightly—decisions based on them could become flawed, risking financial loss and client trust.

This project reflects the process I followed to identify the data integrity issue, trace it back to its origins, resolve it using technical solutions, and implement safeguards to prevent its recurrence. It embodies a clear case of technical problem solving, risk management, continuous improvement, and stakeholder collaboration.

Context and Discovery

The issue first surfaced when I was reviewing a monthly performance report. Some key figures didn't align with historical trends. Initially, I suspected that the client had submitted the wrong file. I contacted the relevant team, who assured me the file matched their records. To be sure, I cross-referenced the submitted data with the tables stored on our internal UK server.

What I found was alarming. The number of rows in our internal data version was significantly lower than in the client's file. A dataset expected to have 45,000 entries had only 36,000. This wasn't a small deviation—it was a significant loss. If our reports were based on the internal data, we could underreport key metrics by 20%.

But the surprises didn't end there. As I manually explored more tables, I found that some mandatory fields—critical columns like `Location ID`, `Building`, `Site`, `Team`, and `Depot`—contained empty or null values. These were fields we relied on to allocate performance metrics and attribute costs. Missing entries in such columns could mean entire teams or locations were excluded from reporting.

The final blow came when I checked the date ranges. Using the `Submitted-On` column, I calculated the minimum and maximum dates in both datasets. The client file covered a full month, as expected. Our internal tables only spanned about 22 days. We were missing over a week of data, confirming that the problem wasn't isolated—it was systemic.

This moment was critical. I realised we were sitting on a potentially severe failure in our data pipeline.

Root Cause Analysis and Technical Response

With the issue confirmed, my next goal was to understand its origin. Was the client at fault for sending corrupted files? Or was our internal pipeline dropping data during ingestion?

I began by checking the ingestion logs from our ETL pipeline. These logs are usually verbose, but they can be lifesavers. Unfortunately, I discovered several warnings—not errors, so they hadn't raised any alarms. These warnings pointed to data schema mismatches. The ingestion script was built to expect a strict set of column names. However, the client occasionally renamed or slightly reformatted columns, for example, using `'location_id'` instead of `'Location ID'`.

Because these columns didn't match the expected schema exactly, our ingestion job quietly skipped the affected rows rather than halting or raising an alert. This was the key flaw: our pipeline lacked validation and fault tolerance.

To verify my hypothesis, I ran manual `pandas` comparisons using both the client and server data. I loaded both Excel files into separate DataFrames, printed their column headers, and used `set()` operations to compare them. Sure enough, even minor differences in naming caused the ingestion to fail silently.

I also wrote a quick script to count missing values across key columns:

```
mandatory_fields = ['Location ID', 'Team', 'Building', 'Site', 'Depot']
for col in mandatory_fields:
    print(f"{col}: {df[col].isnull().sum()} missing entries")
```

The results confirmed that thousands of entries were being discarded or stored with missing critical fields.

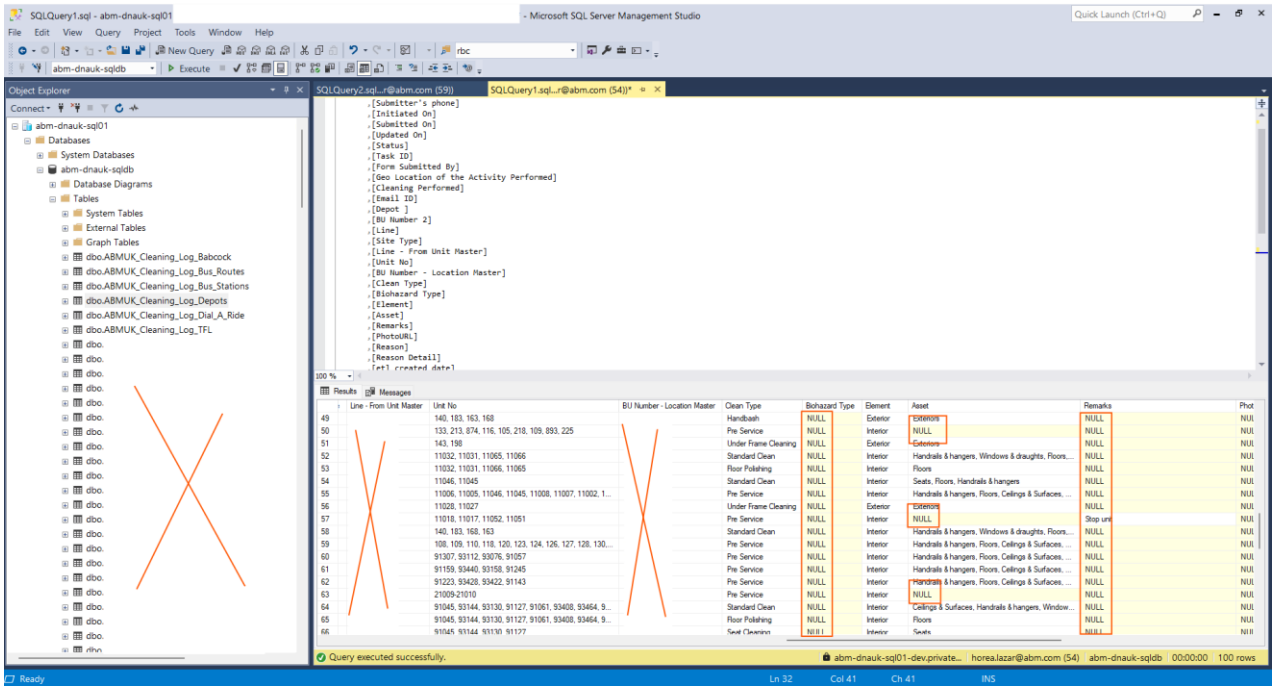
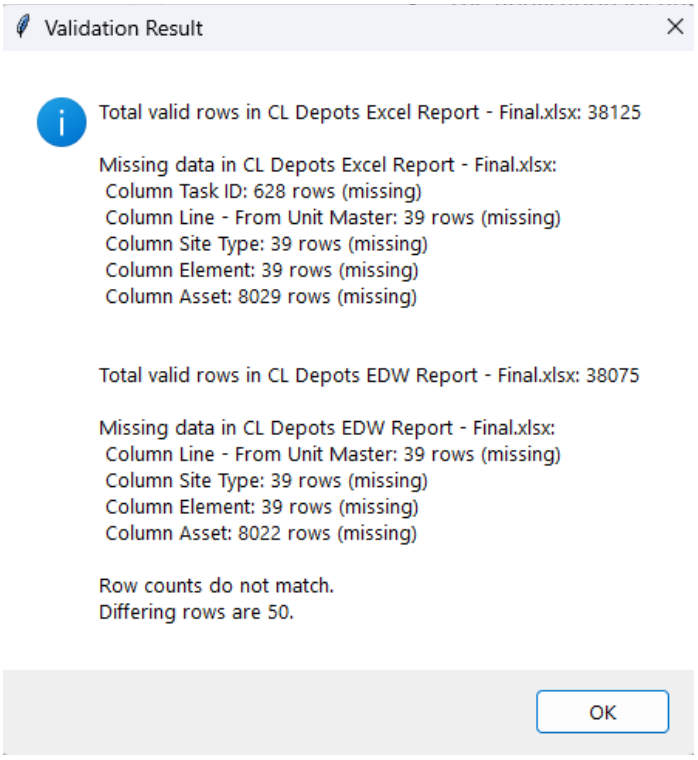


Image 1. Missing data from our UK server ('X' covers sensitive data)

At this point, it became clear that we needed a solution that could:

- Validate incoming files against a predefined schema
- Compare the data against what was already stored
- Alert the team if something went wrong—before we built reports on flawed data



- **Image 2. Count of rows from both files and a list of columns with missing data**

Building the Data Validation Application

To address the issue, I created a Python application using `pandas` and `tkinter`. The app allows users to upload both the client file and our internal version. It then performs multiple layers of validation:

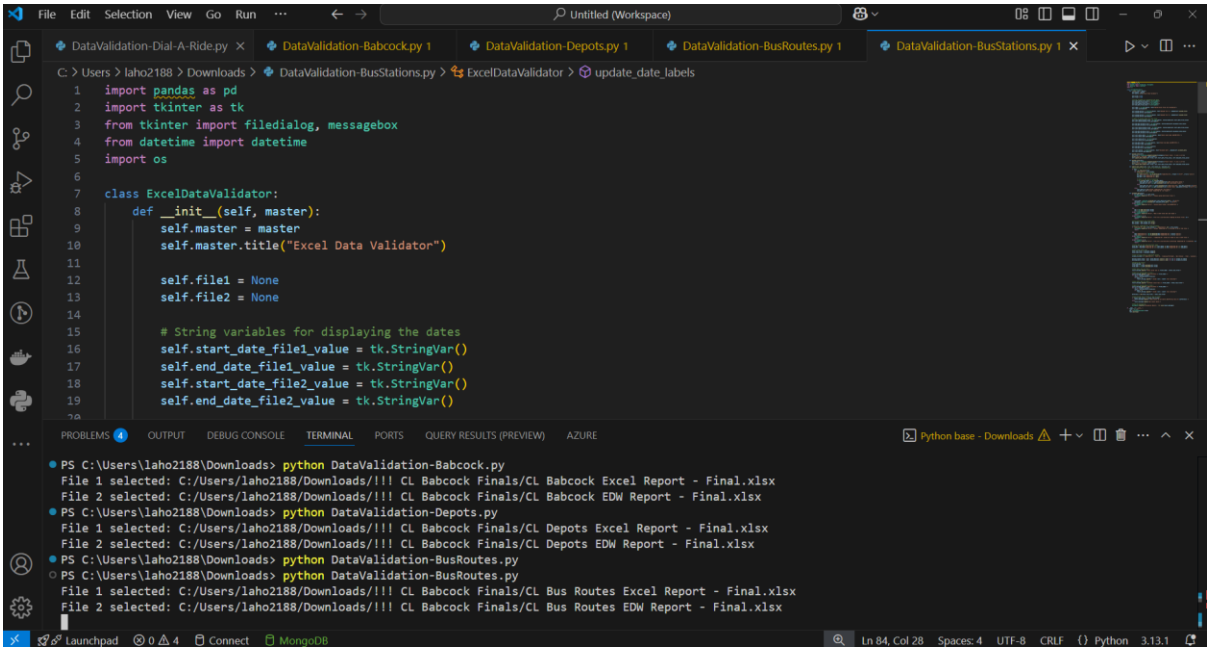


Image 3. Using Visual Studio Code to create the Python application

The app calculates the row count for both files and highlights any differences. A mismatch here indicates missing or extra data entries. It compares column names and flags mismatches, lists missing columns on either side, and checks date ranges using the `Submitted On` field to ensure coverage is consistent.

A particularly valuable feature is its mandatory field validation. The app checks each of the mandatory fields and counts the number of missing values in both datasets. This is crucial to ensuring our analytics are complete.

The GUI was built using `tkinter`, so even non-technical users can use the tool. They can select files through a dialogue box and see results in real time via a message box summary. The final output is a comprehensive report showing row count differences, missing columns, date range mismatches, and count of missing values per column.

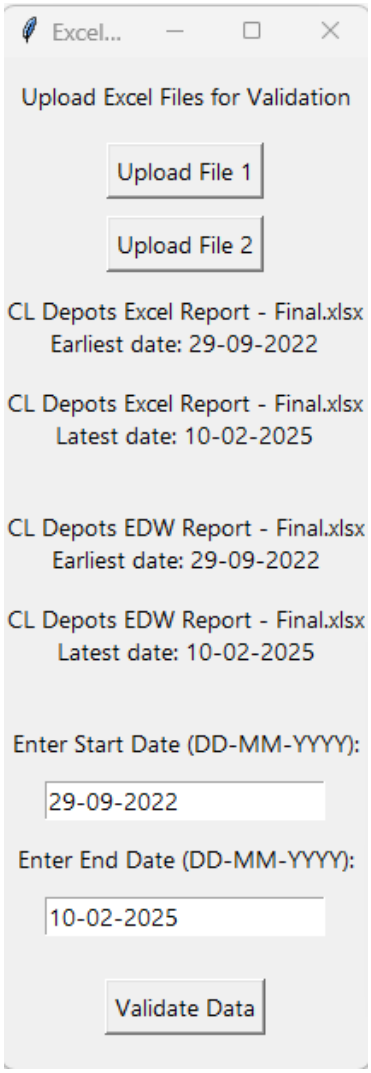


Image 4. Earliest/latest date from both files and input fields

Stakeholder Engagement and Communication

One crucial aspect of this project that evolved alongside the technical work was the involvement of stakeholders. Our finance team depends directly on this data. When I communicated the discrepancies, the concern was immediate: any inaccurate report could misrepresent our client's activity or our service performance.

I initiated discussions with the data analytics team, the ingestion team, and client support. In these meetings, I presented the findings—missing rows, schema mismatches, and uncovered date gaps. To support my explanation, I brought forward clear visual summaries generated by the application, making it easy for non-technical stakeholders to understand the scale of the issue.

The client account manager revealed that they had noticed issues but couldn't identify the cause. Thanks to the tool, we could now detect and correct these issues internally before they reached the client.

I documented the issues in a shared Confluence page and created an internal guide to using the app. This transparency helped build trust within the team and reassured stakeholders that the problem was being actively addressed—not just patched.

CI/CD Integration and Technical Evolution

To future-proof the solution, I integrated the validation process into our CI/CD pipeline using GitLab. I modified our existing `gitlab-ci.yml` configuration to include a validation job before data was committed to production tables.

This automated step acts as a gatekeeper. It prevents faulty data from even reaching our reporting tables. I also created a metrics dashboard in Grafana using Prometheus exporters. This dashboard tracks the frequency of validation failures, the number of missing values encountered each month, and which columns most frequently contain incorrect data.

This evolution from manual checks to automated validation embedded in our pipeline represents a shift in our approach—from reactive to preventive. It reflects a continuous improvement mindset, where one problem leads to a better system, not just a temporary patch.

Scalability, Monitoring, and Future Development

While the initial solution successfully addressed the core discrepancies in static Excel files, the next challenge lies in adapting this validation framework for larger-scale and real-time environments. Currently, our ingestion pipelines mostly process batch files, but our data strategy is increasingly shifting toward streaming data and near-real-time reporting.

To prepare for this, I've begun designing a modular validation engine that can be containerised and deployed as a microservice. The idea is to decouple the validation logic from the user interface and expose it as an API endpoint. This allows other teams—or even external partners—to send data payloads and receive structured feedback on quality, completeness, and integrity. By using FastAPI and asynchronous processing, the system will be scalable and suitable for integration with our Kafka-based pipelines.

Another future step is to integrate data contract enforcement through a schema registry like Confluent or using Great Expectations. These tools enable us to define expectations not only in code but also as living documentation that evolves alongside our data products. For example, we can set rules such as "95% of rows must have a valid Location ID" or "No nulls are allowed in 'Depot' for

any weekday batch." Violations of these rules can automatically trigger pipeline rejections or alerting mechanisms.

Monitoring is another area of focus. While our current Grafana dashboards provide visibility into validation failures, we still rely on reactive approaches when something breaks. In the future, I plan to introduce anomaly detection using time series forecasting. If a sudden drop in row count or spike in null values occurs, it can be flagged even if the validation script technically passes.

Finally, there's a human component to continuous improvement: documentation and knowledge sharing. I've begun working on a playbook—a living document that covers common ingestion pitfalls, how to use the validator tool, best practices for client file formatting, and how to handle common failures. This ensures that as our data engineering team grows, we don't repeat mistakes or lose valuable insights. We move forward with collective knowledge, not just individual fixes.

Lessons Learned: Data Governance and Sustainability

Working on this project taught me that technical issues are rarely isolated. A seemingly minor bug—such as a few missing rows—often reveals a much more significant weakness in the system.

We lacked a formal definition of what constitutes "valid data". After resolving the technical issues, I collaborated with our data team to draft a formal schema document outlining the expected column names, data types, mandatory fields, acceptable ranges, and null handling policy. We now review this schema quarterly and store it in Git.

Performance issues were also addressed. For large files, I optimised the validation script by replacing row-by-row checks with vectorised operations. I also migrated file loading from `read_excel()` to `read_csv()` where possible.

We're now saving analyst time previously spent debugging incorrect reports. Early estimates suggested savings of 5–10 hours per week, freeing up time for deeper analysis.

Final Reflection

What started as a minor discrepancy evolved into a comprehensive project that impacted multiple teams, tools, and processes. It was a deep dive into the realities of data pipeline fragility

and the importance of validation. I learned the value of curiosity, persistence, and building tools that empower teams to be proactive rather than reactive.

I now better understand how a data engineer can not only solve problems but also shape systems and influence culture. I plan to apply this same approach to other fragile points in our pipeline, particularly real-time Kafka streams, which currently lack schema enforcement.

Beyond the technical, this experience reinforced that quality data underpins everything—from decisions and insights to trust and business continuity. And that trust begins with us, the engineers who build the foundations that others stand on.