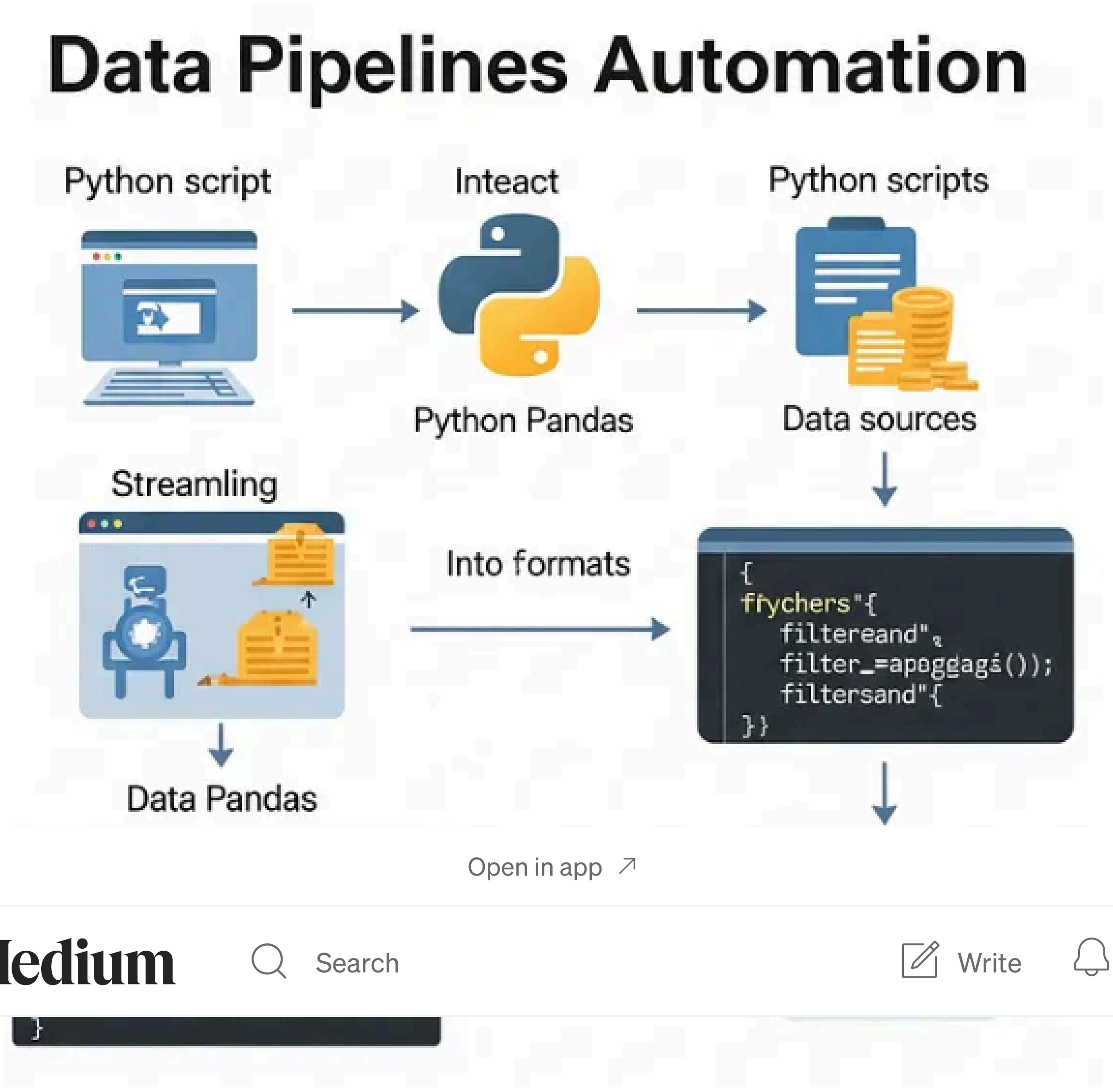

```
{
ffychers"{
   filtereand",
   filter_=apoggagś());
   filtersand"{
}}
```

Automating Data Ingestion Without Parallel Processing

Anurag Sharma 3 min read · Jun 6, 2025

In the fast-paced world of data engineering, automating repetitive tasks can be a game-changer, even when your data is not large enough for heavy-duty parallel processing frameworks. In a recent project, I tackled the challenge of ingesting and validating critical static data from various sources like APIs, CSVs, and databases into our data warehouse. By building a lightweight automation framework with Python and Pandas, we transformed a manual, error-prone process into a streamlined, reliable workflow.

The Challenge: Manual Data Ingestion Drains Time and Risks Errors

Our team needed to ingest static data — think configuration files, reference tables, or metadata — from multiple sources into our data warehouse. The data volumes were modest, so parallel processing frameworks like Spark were not needed. However, the process was entirely manual: downloading files, cleaning them, running validation checks, and loading them into the warehouse. This took hours each time, diverted focus from strategic work, and risked human errors that could compromise data quality. We needed a solution that was efficient, accurate, and simple to maintain.

The Solution: A Lightweight Automation Framework with Python and Pandas

I built a Python-based automation framework using Pandas for data manipulation. The goal was to automate data ingestion, cleaning, validation, and loading without overcomplicating the system.

Here’s the step-by-step approach:

Step 1: Automated Data Ingestion with Directory Monitoring

I wrote a Python script to continuously monitor a designated directory for new files. Using the watchdog library, the script detected new files (e.g., CSVs or JSONs from APIs) as soon as they arrived. Once detected, Pandas reads the file into a DataFrame for processing.

Step 2: Data Cleaning with Pandas

Next, I implemented data cleaning routines using Pandas to ensure the data was in the right format.

This included:

- Handling missing values by filling or flagging them (e.g., replacing nulls with defaults or marking them for review).
- Standardizing date formats (e.g., converting “MM-DD-YYYY” to “YYYY-MM-DD” for consistency).
- Removing duplicates based on key columns like IDs to avoid redundancy.
- Trimming and normalizing string data (e.g., removing extra spaces, converting to lowercase for uniformity).
- Converting data types (e.g., ensuring numeric columns like “price” are floats, not strings).
- Filtering out invalid records (e.g., dropping rows with unrealistic values like negative ages).

Step 3: Comprehensive Data Quality Checks

To ensure data reliability, I integrated data quality libraries like pandas and custom validation rules.

The script performed checks such as:

- Verifying data types (e.g., ensuring IDs are integers, not strings).
- Checking key fields for integrity (e.g., no nulls in required columns like "customer_id").
- Validating value ranges (e.g., ensuring a "price" column isn't negative).
- Ensuring referential integrity (e.g., confirming foreign keys like "product_id" exist in the reference table).
- Checking for data consistency (e.g., ensuring "order_date" isn't in the future as of June 04, 2025).
- Validating patterns (e.g., ensuring email addresses match a standard format like "user@domain.com").

Step 4: Data Movement and Alerts

If the data passed all quality checks, the script loaded it into the data warehouse by writing it to the target location (e.g., a specific schema or table).

If any issues were detected, the script:

- Moved the problematic file to a "quarantine" folder.

- Sent an email alert to the team using Python's smtplib, detailing the issue for quick investigation.

The Impact: Efficiency, Accuracy, and Simplicity

This automation framework delivered impressive results:

- **Time Savings**: Reduced ingestion and validation time from hours to minutes, freeing the team for higher-value tasks.
- **Improved Accuracy**: Automated checks eliminated manual errors, ensuring consistently high data quality.
- **Simplicity**: Using Python and Pandas kept the solution lightweight and easy to maintain, avoiding the complexity of parallel processing frameworks.
- **Reliability:** Real-time monitoring and alerts allowed us to catch and resolve issues instantly.

For example, a process that once took 3 hours per dataset is now completed in under 10 minutes, and we reduced data quality issues by 70% through consistent validation.

Lessons Learned: Automation Doesn't Always Need Big Tools

Key lessons for data engineers:

1. **Start Simple**: For small-to-medium datasets, Python and Pandas can be powerful enough for automation without the overhead of frameworks like Spark.

2. **Prioritize Quality:** Automated validation ensures data reliability, reducing downstream errors.
3. **Build for Alerts:** Real-time monitoring and notifications are critical for proactive issue resolution.
4. **Keep It Maintainable:** A lightweight solution is easier to debug and scale as needed.

Python Data Engineering Automation Data Quality Data Engineer

Written by Anurag Sharma

82 followers · 3 following

Edit profile

Data Engineering Specialist with 10+ exp. Passionate about optimizing pipelines, data lineage, and Spark performance and sharing insights to empower data pros!

No responses yet

Anurag Sharma him/he

What are your thoughts?