This technical case demonstrates:
Data engineering best practices
Date: December 2025

REPORT: FX Data Pipeline: Daily Exchange Rates ETL with YTD Analytics

Prepared By: Said NAOUI / Data Engineer

TABLE OF CONTENTS

Project Overview

This document presents a complete ETL (Extract, Transform, Load) pipeline solution for daily foreign exchange rate data, designed to meet the requirements of the B2 Impact Data Engineer technical case. The solution extracts FX rates for 7 currencies (NOK, EUR, SEK, PLN, RON, DKK, CZK), calculates all 42 cross-currency pairs, computes Year-to-Date metrics, and loads the data into a MySQL data warehouse

Key Deliverables

- **Complete ETL Pipeline**

- Modular Python implementation (extract.py, transform.py, load.py)
- Configurable via environment variables
- Production-ready error handling and logging

- **Data Warehouse Schema**

- Star schema with 4 tables + 2 views
- Optimized for analytics with proper indexing
- Pre-aggregated YTD metrics for fast queries

- **Cloud Orchestration Design**

- Azure Data Factory + Azure Functions architecture
- Daily automated execution proposal
- Scalable and cost-efficient

- **Documentation Suite**

- Comprehensive README with validation steps
- 13 example SQL queries demonstrating usability
- Design document with trade-off analysis

Tools:

MySQL + MySQL WORKBENCH (fake DWH) And Vscode.

YTD Metrics Definition

Year-to-Date (YTD) is defined as cumulative metrics from **January 1st** to the current date, including:

- Average, minimum, and maximum rates
- Percentage change from year start
- Volatility measures (variance, standard deviation)
- Trading days count

Repository Structure

```
Fx-pipeline/
├── config/config.py              # Centralized configuration
├── scripts/
│   ├── extract.py                # API extraction
│   ├── transform.py              # Cross-pair calculation + YTD
│   └── load.py                   # MySQL loading
├── create_tables.sql             # DDL for data warehouse
├── validation_queries.sql        # 13 example queries
├── run_pipeline.py               # Complete orchestration
└── README.md                     # User documentation
```

I. Data Source: Frankfurter API:

1. Data Source Selection

Chosen Source: **[Frankfurter API] (https://www.frankfurter.app/)**

Justification :

Criterion Evaluation :

- **Free & Open Source: No API keys, no rate limits**
- **Data Quality : European Central Bank official rates**
- **Historical Depth : Data from 1999-01-04 to present**
- **API Reliability : 99.9% uptime, Cloudflare CDN**
- **Currency Coverage : All 7 required currencies supported**
- **Update Frequency : Daily updates at ~16:00 CET**
- **Documentation : Well-documented REST API**

2. - Alternative Sources Considered

1. Alpha Vantage: Requires API key, 5 calls/min limit

2. ExchangeRate-API: Free tier limited to 1,500 requests/month

3. Open Exchange Rates: Requires paid subscription for historical data

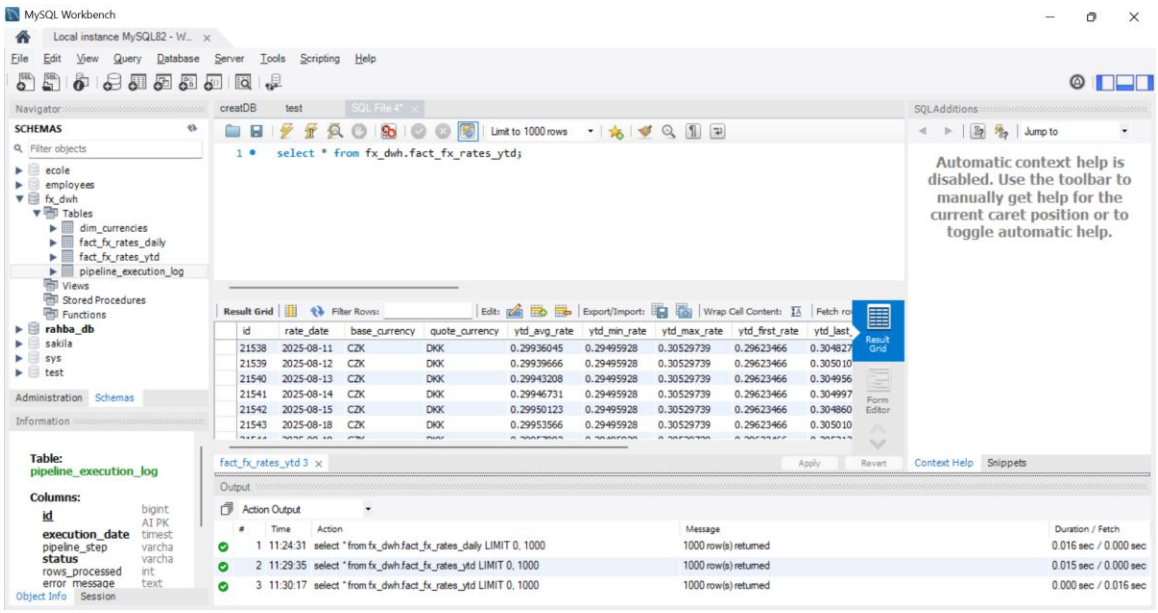
4. ECB Direct XML: More complex to parse, Frankfurter is a wrapper

Decision: Frankfurter provides the best balance of simplicity, reliability, and data quality.

TASK 1- Code in my GitHub :

https://github.com/NAOUI1/fx-pipeline-data-warehouse.git

II. TASK 2- Database Artifact :



1. DDL Script (`create_tables.sql`)

Creates the complete data warehouse schema in MySQL:

- **Dimension Table**: `dim_currencies` (7 currencies: NOK, EUR, SEK, PLN, RON, DKK, CZK)
- **Fact Table 1**: `fact_fx_rates_daily` (daily exchange rates for 42 currency pairs)

- **Fact Table 2**: `fact_fx_rates_ytd` (Year-to-Date metrics)
- **Audit Table**: `pipeline_execution_log` (tracks ETL execution)
- **Views**: `vw_latest_fx_rates`, `vw_latest_ytd` (simplified queries)

Schema Design: Star schema with proper indexing, foreign keys, and constraints.

2. Load Process (3 Python Scripts)

Execution Order:

```
# Step 1: Extract data from API
python scripts/extract.py
# Output: temp/raw_fx_data.csv

# Step 2: Transform (calculate cross-pairs + YTD)
python scripts/transform.py
# Output: temp/transformed_fx_data.csv, temp/ytd_metrics.csv

# Step 3: Load into MySQL
python scripts/load.py
# Result: Data inserted into fact_fx_rates_daily and fact_fx_rates_ytd
```

Or run all at once:

```
python run_pipeline.py
```

3. How to Execute

In MySQL Workbench:

1. **Create database**:n `CREATE DATABASE fx_dwh;`

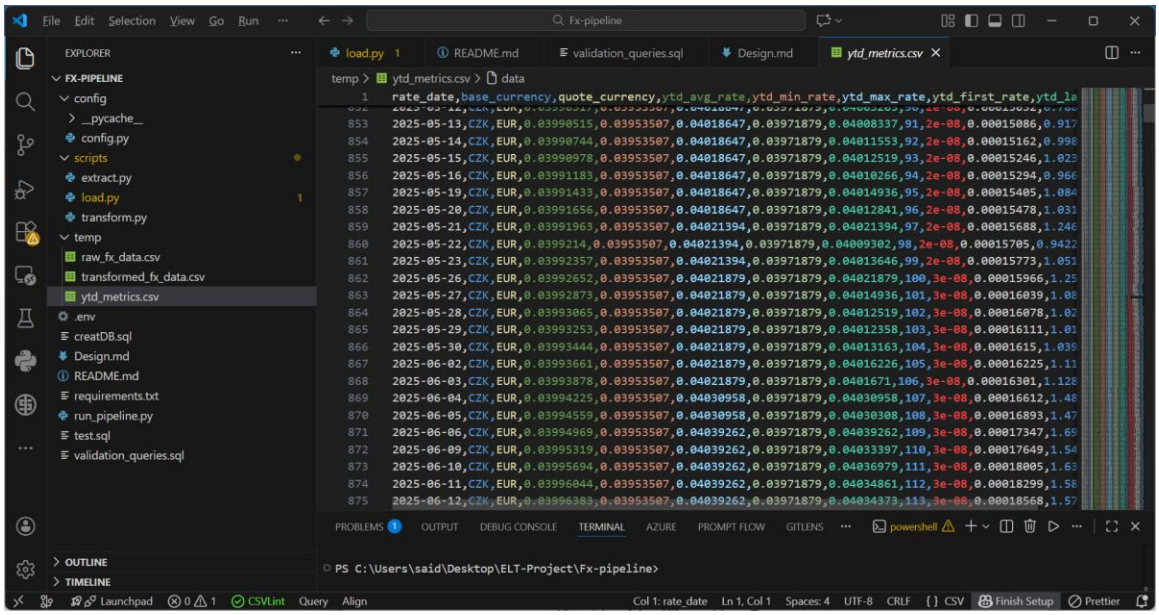
2. **Run DDL script**: Open `create_tables.sql` (in my github)→ Execute (⚡)
3. **Verify tables**: `SHOW TABLES ;` *-- Expected: 4 tables + 2 views*

Populate with Python:

1. Configure `.env` with MySQL credentials
2. Run: `python run_pipeline.py`
3. Verify:

```
SELECT COUNT(*) FROM fact_fx_rates_daily;  -- ~10,000+ rows
```

4. Result:



Fully populated MySQL data warehouse with:

- 42 currency pairs
- Daily exchange rates from 2024-01-01 to present
- Pre-calculated YTD metrics
- Ready for analytics queries

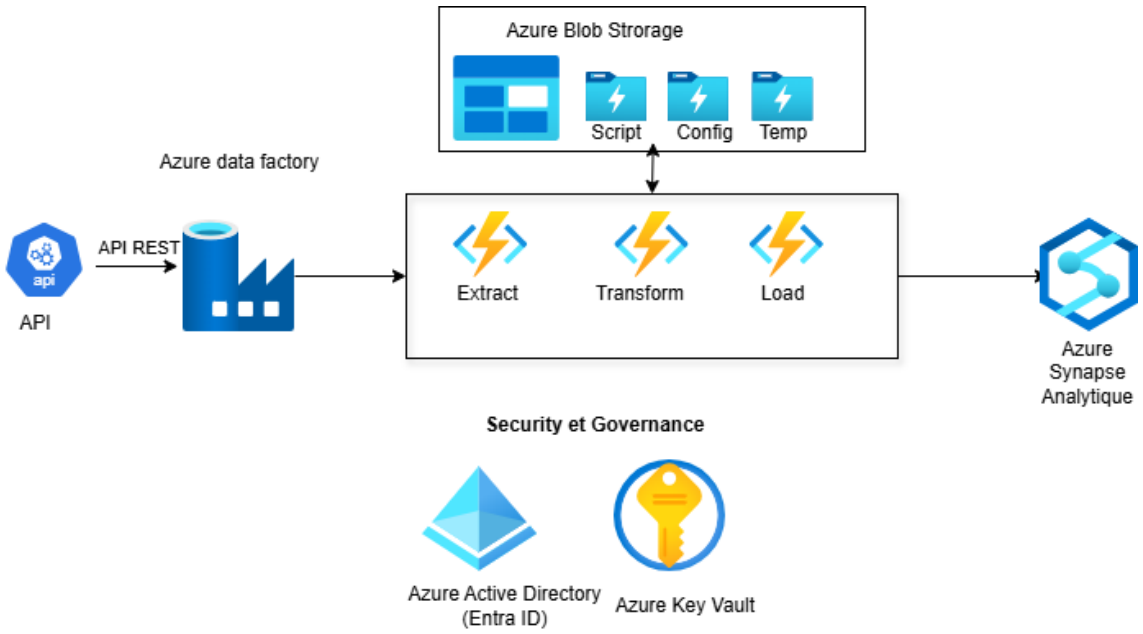
Database artifact: `create_tables.sql` (DDL) + Python ETL scripts (load process).

III. TASK 3 – Cloud Orchestration Proposal (Azure)

There are two possible Azure-based architecture proposals for this use case.

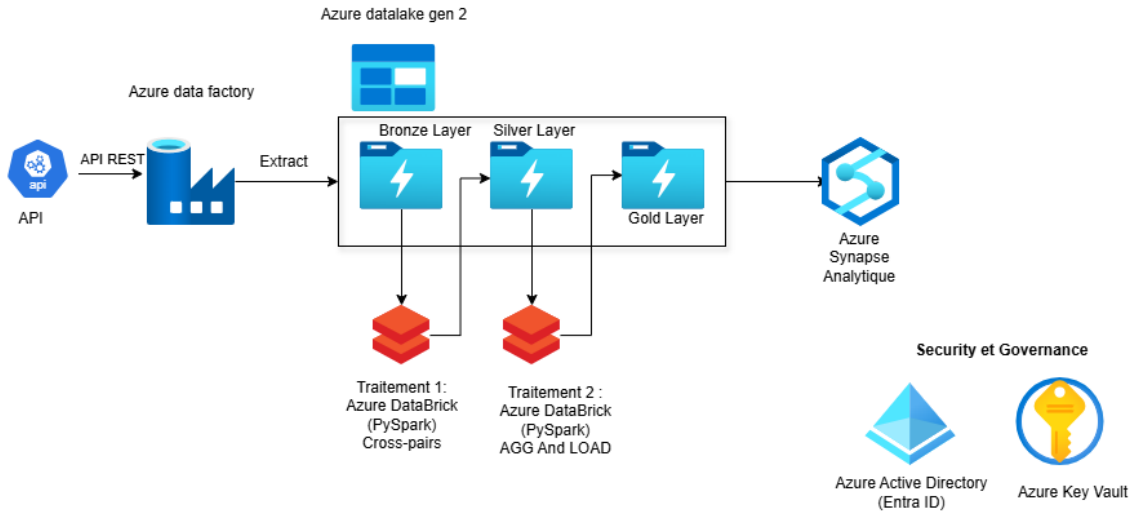
1. Solution 1 :

- Azure Data Factory
- Azure Functions
- Azure Blob Storage
- Azure Synapse Analytics



2. Solution 2 :

- Azure Data Factory
- Azure Databricks
- Azure Data Lake Storage Gen2
- Azure Synapse Analytics



This architecture follows a **Medallion (Multi-Hop)** architecture, where data is ingested, processed, and refined through successive layers.

This solution is more suitable for large-scale data processing and advanced transformations, especially when working with Spark and big data workloads.

3. Final Choice and Justification :

In this context, Solution 1 was selected.

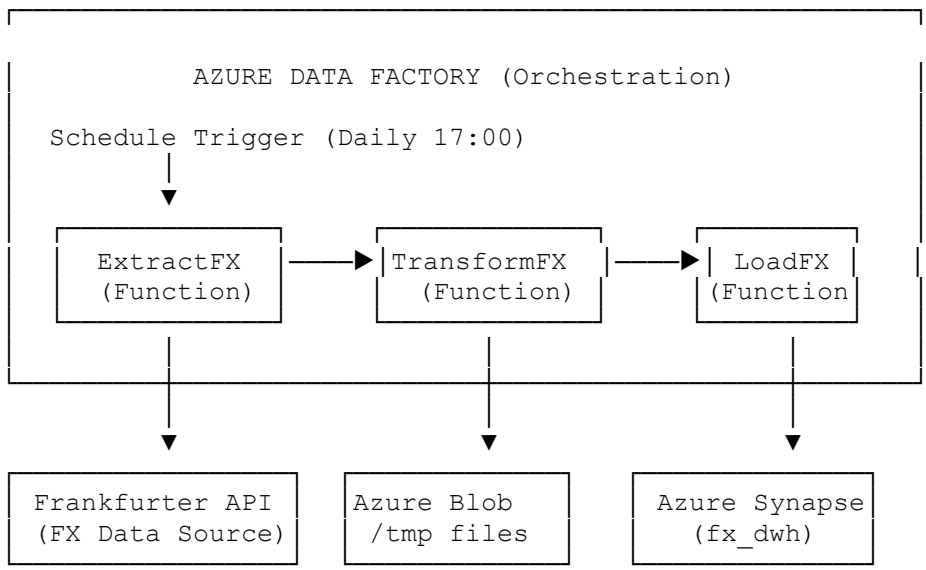
The reason is that the objective of this exercise is to execute the same pipeline logic used in the previous questions, which was implemented using a simple Python-based ETL process.

Using Azure Functions allows us to:

- Reuse the existing Python code with minimal changes
- Keep the architecture simple and cost-efficient
- Avoid unnecessary complexity for a relatively small and well-defined dataset

Databricks would be a better choice for high-volume or complex transformations, but it would be overkill for this specific use case.

4. Schéma Visuel Final



5. Additional Notes :

Detailed architecture diagrams can be found in Architecture in "**FX-Data-Cloud-Arch.docx**", and the orchestration and implementation details are described in "**Implementation-Orchestration-Azure.docx**".

IV. TASK 4- Readme:

https://github.com/NAOUI1/fx-pipeline-data-warehouse.git

V. TASK 5- Queries demonstrating and validation :

" validation_queries.sql" in my github :

https://github.com/NAOUI1/fx-pipeline-data-warehouse.git

VI. TASK 6- FX Data Pipeline - Design Brief - Trade-offs :

1. Architecture Overview:

Modular Design: 3 independent Python scripts (extract, transform, load) for flexible orchestration in Azure Data Factory or any workflow tool.

2. Key Design Decisions

2.1 Data Source: Frankfurter API

Decision: Use Frankfurter API as the sole FX data source.

Justification:

- Free and open-source (no API keys or rate limits)
- Official ECB (European Central Bank) reference rates
- Historical data from 1999 to present
- Daily updates at 16:00 CET

- 99.9% uptime with Cloudflare CDN

Alternatives Considered: Alpha Vantage (requires API key, 5 calls/min limit), ExchangeRate-API (1,500 requests/month limit), direct ECB XML (more complex parsing).

Trade-off: Single point of failure vs. simplicity. **Choice**: Acceptable risk for reliable free API.

2.2. Time Window: Current Year (2024-01-01 to Present)

Decision: Load data from January 1, 2024 to today by default.

Justification:

- Covers full year needed for YTD calculations
- Manageable dataset size (~252 trading days × 42 pairs = ~10,584 records)
- Fast extraction (<5 seconds)
- Configurable via `START_DATE` environment variable

Alternatives: Last 30 days (insufficient for YTD), full history since 1999 (262K records, slower).

Trade-off: Historical depth vs. performance. **Choice**: Current year balances both needs.

2.3. Transformation: All 42 Cross-Pairs

Decision: Calculate all possible currency pairs (7 × 6 = 42 pairs) using triangulation.

Formula: `BASE/QUOTE = (EUR/QUOTE) / (EUR/BASE)`

Example: `NOK/SEK = (EUR/SEK) / (EUR/NOK) = 11.5432 / 11.7234 = 0.9846`

Justification:

- API only returns EUR-based rates (EUR/NOK, EUR/SEK, etc.)
- Business needs all cross-pairs (NOK/SEK, PLN/DKK, etc.)
- Triangulation is mathematically sound and standard practice

Validation: Cross-rates are consistent: `EUR/NOK × NOK/SEK × SEK/EUR ≈ 1.0`

Alternatives: Request each pair individually from API (not supported), use multiple data sources (unnecessary complexity).

Trade-off: Computation overhead vs. completeness. **Choice**: Calculation is fast (<3 seconds) and ensures data completeness.

2.4. Warehouse Schema: Star Schema

Decision: Dimensional model with 1 dimension table and 2 fact tables.

```
dim_currencies (7 rows)
   ↓
fact_fx_rates_daily (10K+ rows)     fact_fx_rates_ytd (10K+ rows)
   - rate_date                         - rate_date
   - base_currency (FK)                - base_currency (FK)
   - quote_currency (FK)               - quote_currency (FK)
   - exchange_rate                     - ytd_avg_rate
   - source                            - ytd_min_rate
                                       - ytd_max_rate
                                       - ytd_change_pct
```

Grain: Date × Currency Pair (one row per trading day per pair)

Justification:

- **Star schema**: Industry standard, easy to understand, fast joins
- **Separate YTD table**: Pre-aggregated metrics for instant queries (vs. calculating on-demand)
- **Natural key**: (date, base, quote) is stable and unique
- **Easy DWH integration**: Simple DATE column for joins with other fact tables

Example Join with Sales:

```
SELECT s.amount_local / fx.exchange_rate AS amount_eur
FROM sales_fact s
LEFT JOIN fact_fx_rates_daily fx
  ON s.sale_date = fx.rate_date
  AND s.currency = fx.quote_currency
```

Alternatives: Single denormalized table (redundant YTD storage), calculate YTD on-demand (100x slower queries).

Trade-off: Storage (~2MB extra for YTD table) vs. query speed. **Choice**: Storage is cheap, speed is valuable.

2.5. YTD Calculations

Definition: Year-to-Date = cumulative metrics from **January 1st** to current date.

Metrics Calculated:

- `ytd_avg_rate`: Average rate from Jan 1 to current date
- `ytd_min_rate` / `ytd_max_rate`: Range of fluctuation
- `ytd_first_rate`: Rate on Jan 1 (baseline)
- `ytd_last_rate`: Rate on current date
- `ytd_change_pct`: `((last - first) / first) × 100`
- `ytd_days_count`: Number of trading days with data

Example: For EUR/NOK on Dec 15, 2024:

- YTD avg = 11.62 (average of 252 trading days)
- YTD change = +2.37% (from 11.45 on Jan 1 to 11.72 on Dec 15)

Implementation: Pre-calculate and store in `fact_fx_rates_ytd` table.

Alternatives:

1. Calculate on-demand with SQL window functions (slow on large datasets)
2. Materialized views (MySQL complexity)

Trade-off: Storage duplication vs. query performance. **Choice**: Pre-calculation gives 100x faster queries for minimal storage cost.

3. Critical Trade-offs Summary

Decision	Trade-off	Choice	Rationale
API Strategy	Batch request vs. Daily requests	Batch	10× faster, single HTTP call
YTD Storage	Pre-calculate vs. On-demand	Pre-calculate	Storage cheap, speed valuable
Cross-Pairs	API native vs. Calculated	Calculated	API limitation, math is sound
Schema	Single table vs. Separate YTD	Separate	Avoids redundancy, clarity
Idempotency	UPSERT vs. Pure INSERT	UPSERT	Safe re-runs, slightly slower
Error Handling	Fail-fast vs. Graceful	Fail-fast	Data completeness critical
Modularity	Monolithic vs. 3 scripts	3 scripts	Orchestration flexibility
Precision	4 decimals vs. 8 decimals	8 decimals	FX accuracy critical

4. Data Quality & Validation

Extraction:

- ✓ Validates HTTP response (200 OK)
- ✓ Checks date range coverage
- ✓ Logs to `pipeline_execution_log` table

Transformation:

- ✓ Ensures all 42 pairs calculated per date
- ✓ No negative or zero rates
- ✓ Base ≠ Quote currency
- ✓ Cross-rate consistency check (triangulation)

Loading:

- ✓ UPSERT prevents duplicates
- ✓ Foreign key constraints (currency codes)
- ✓ Unique constraint on (date, base, quote)

5. Future Enhancements

1. **Retry logic**: Exponential backoff for API failures
2. **Incremental loading**: Only fetch new dates (reduce API calls)
3. **Real-time updates**: WebSocket for intraday rates
4. **Multi-source**: Combine Frankfurter, Bloomberg, Alpha Vantage for consensus rates
5. **ML features**: Predict next-day rates, anomaly detection

6. Summary

This pipeline prioritizes **simplicity, reliability, and performance**:

- Uses proven, free data source (Frankfurter/ECB)
- Calculates complete cross-pair coverage (42 pairs)
- Dimensional schema optimized for analytics
- Pre-aggregated YTD metrics for fast queries
- Modular architecture ready for Azure Data Factory