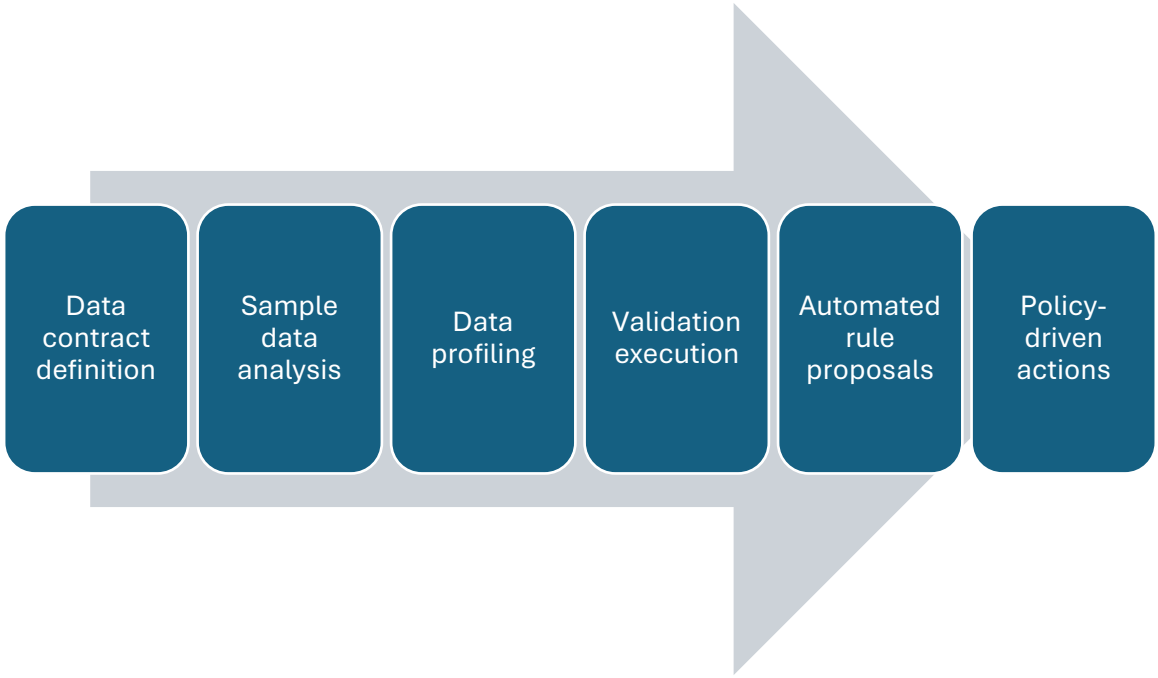
XFlowData | Case Study: Data Quality Validation Pipeline (Technical Overview)

This technical case demonstrates how the data quality validation system processes real-world data through its complete workflow, from contract definition to validation reporting and rule proposals.

A contract is a declarative specification that defines the expected structure, format, and quality rules for a dataset. It serves as a formal agreement between data producers and consumers about what constitutes valid, high-quality data.

Case Overview: E-commerce Order Validation

We'll examine how the system validates e-commerce order data, using actual sample data that contains both valid records and quality violations to showcase the platform's capabilities.



Step 1: Data Contract Definition

The system starts with a **declarative contract** that defines expectations for the orders' dataset:

```yaml
# order.yml
dataset: orders
version: 1.0
columns:
  order_id:
    dtype: integer
    required: true
    unique: true
  order_date:
    dtype: datetime
    required: true
    expectations:
      - name: not_null
      - name: recent_than
        params: { days: 365 }  # within last ~1 years
  amount:
    dtype: float
    required: true
    expectations:
      - name: between
        params: { min: 0, max: 100000 }
      - name: quantile_guard
        params: { p_low: 0.01, p_high: 0.99 }
  currency:
    dtype: string
    required: false
    expectations:
      - name: in_set
        params: { values: ["EUR","USD","GBP"] }
```

The contract establishes **hard business rules**:

- order IDs `order_id` must be unique integers,
- `order_date` must be recent (within 1 year),
- `amounts` must be positive and under €100,000,
- and only specific `currency`(ies) are allowed.

Step 2: Sample Data Analysis

The system processes incoming order data containing various quality issues:

order_sample.csv

order_id	order_date	amount	currency	**extra_note**
1	2024-10-01T10:00:00	12.50	EUR	ok
5	2025-09-01T08:00:00	**-7**	EUR	bad_amount
7	2025-09-04T00:00:00	42000	**MAD**	future_date

This dataset intentionally contains multiple quality violations:

- **Negative amount** (-7 EUR) violating the positive range expectation.
- **Unsupported currency** (MAD) not in the allowed set
- **New column** (extra_note) not defined in the contract.
- **Data type mismatch** (dates stored as strings instead of datetime objects)

Step 3: Data Profiling

The profiler analyzes the actual data characteristics and generates statistical summaries: *profile.json*


```
{
  "columns": {
    "order_date": {
      "non_null_ratio": 1.0,
      "dtype": "string",
      "max_length": 19,
      "n_unique": 7,
      "sample_values": [
        "2024-10-01T10:00:00",
        "2025-09-01T08:00:00",
        ...
      ]
    },
    "amount": {
      "non_null_ratio": 1.0,
      "dtype": "float",
      "min": -7.0,
      "max": 42000.0
    },
    "currency": {
      "non_null_ratio": 1.0,
      "dtype": "string",
      "max_length": 3,
      "n_unique": 4,
      "sample_values": [
        "EUR", "USD", "GBP", "MAD"
      ]
```

```
    }
  },
  "extra_note": {
    "non_null_ratio": 1.0,
    "dtype": "string",
    "max_length": 11,
    "n_unique": 5,
    "sample_values": [
      "ok",
      "note",
        ...
    ]
  },
  "row_count": 7
}
```


The profiler **discovers data drift**: it detects that:

- `order_date` is stored as strings (not datetime objects),
- identifies `amount` ranges that exceed contract boundaries,
- and finds `currency` values outside the expected set.

Step 4: Validation Execution

The validator compares **actual data** against **contract expectations** and produces a detailed violation report:

report.json


```
{
  "dataset": "orders",
  "version": "1.0",
  "hard_errors": [
    {
      "column": "order_date",
      "error": "dtype_change"
    },
    {
      "column": "amount",
      "error": "expectation_failed:between[0,100000]"
    },
    {
      "column": "currency",
      "error": "expectation_failed:in_set[3]"
    }
  ],
  "soft_warnings": [],
  "new_columns": [
    "extra_note"
  ],
  ...
}
```

Hard Errors Detected

- **Data Type Change**: `order_date` expected as datetime but found as string.
- **Range Violation**: `amount` value -7 fails the between [0, 100.000] expectation.
- **Set Violation**: `currency` value "MAD" fails the in_set["EUR", "USD", "GBP"] expectation.

Schema Changes

- **New Column**: `extra_note` column discovered but not defined in contract.

Detailed Column Results

report.json


```
{
  ...
  "column_results": {
    "order_id": [
      { "expectation": "required", "ok": true },
      { "expectation": "unique", "ok": true }
    ],
    "order_date": [
      { "expectation": "required","ok": true },
      { "expectation": "not_null", "ok": true },
      { "expectation": "recent_than[3650]", "ok": true }
    ],
    "amount": [
      { "expectation": "required", "ok": true },
      { "expectation": "between[0,100000]", "ok": false },
      { "expectation": "quantile_guard", "ok": true }
    ],
    "currency": [
      { "expectation": "in_set[3]", "ok": false }
    ]
  },
  "row_count": 7
}
```


The system tracks **expectation-level outcomes**:

- `order_id`: ✅ required and unique expectations passed.
- `order_date`: ✅ required, not empty and order registered within last year.
- `amount`: ❌ between expectation failed, but quantile_guard passed.
- `currency`: ❌ in_set expectation failed.

Step 5: Automated Rule Proposals

Based on the profiling data, the system generates **adaptive rule proposals**:
proposal.yml

```yaml
dataset: orders
proposals:
  amount:
    - name: between
      params:
        min: -7.0
        max: 42000.0
  currency:
    - name: in_set
      params:
        values: [EUR, USD, GBP, MAD]
  extra_note:
    - name: dtype_is
      params:
        type: string
    - name: max_length
      params:
        n: 11
```

The “Expert system” / “Machine learning” **learns from actual data patterns** and suggests:

- Expanding the `amount` range to accommodate the observed -7 to 42,000 range
- Adding “MAD” to the allowed `currency` set
- Creating rules for the new `extra_note` column based on observed characteristics.

Step 6: Policy-Driven Actions

The contract’s policy section defines **automated responses** to different violation types: *orders.yml*

```yaml
...
policy:
  on_new_column: "propose_soft_rule"   # "allow" | "quarantine"
  on_dtype_change: "quarantine"
  on_expectation_fail_hard: "reject"
  on_expectation_fail_soft: "warn"
```

- `on_new_column`: “propose_soft_rule” → New columns trigger soft rule proposals
- `on_dtype_change`: “quarantine” → Data with type mismatches gets quarantined
- `on_expectation_fail_hard`: “reject” → Records with hard violations are rejected
- `on_expectation_fail_soft`: “warn” → Soft violations trigger a warning

Technical Architecture Insights

Modular Design

The system separates concerns across focused modules:

- **contract.py**: Contract loading and management
- **profiler.py**: Statistical data analysis
- **validator.py**: Rule execution engine
- **proposer.py**: Machine learning-driven rule suggestions

Testing Strategy

Comprehensive test coverage validates core workflows:

- **test_profile_and_propose.py** : **Unit tests** verify individual component behavior
- **test_validator.py** : **Validation tests** ensure error detection accuracy
- **Integration tests** validate end-to-end profiling and proposal generation

Stream Processing Ready

The architecture supports both **batch** and **streaming** validation through Kafka integration, making it suitable for real-time data quality monitoring in production pipelines.

Business Impact

This technical approach delivers:

1. **Automated Quality Gates**: Prevents bad data from entering downstream systems
2. **Adaptive Rule Management**: Reduces manual rule maintenance through ML-driven proposals
3. **Transparent Reporting**: Provides actionable insights for data teams and business stakeholders
4. **Policy Flexibility**: Allows different handling strategies for different types of violations
5. **Continuous Monitoring**: Enables real-time quality assessment in streaming architectures

The system transforms data quality from a reactive, manual process into a **proactive, automated capability** that scales with growing data volumes and evolving business requirements.

Contents