Assignment 2: Data Contracts Implementation

GitHub Repository:
`https://github.com/sps1001/MLOps-Sahilpreet_Singh-B23CS1061/tree/Assignment-2`

Abstract

This report documents the implementation of data contracts for four distinct industry scenarios following the Open Data Contract Standard (ODCS). Data contracts serve as formal agreements between data producers and consumers, ensuring quality, reliability, and semantic clarity. The implementation demonstrates logical-physical separation, quality validation, PII governance, and circuit breaker patterns across ride-sharing, e-commerce, IoT, and financial transaction domains.

1 Introduction

As data infrastructure scales, the dependence on informal data exchanges leads to fragile pipelines, cascading failures, and semantic drift. **Data contracts** provide a formal, versioned agreement between producers and consumers, establishing clear expectations for schema, quality, freshness, and governance.

1.1 Motivation

Traditional data pipelines suffer from:

- **Tight Coupling:** Consumers directly depend on physical database schemas
- **Silent Failures:** Schema changes break downstream systems without warning
- **Quality Degradation:** No automated validation at pipeline boundaries
- **Governance Gaps:** PII and sensitive data lack proper tagging

Data contracts address these issues by acting as *interface specifications* that decouple consumers from implementation details.

1.2 Assignment Objectives

This assignment implements data contracts for four scenarios:

1. **Ride-Share:** Comprehensive contract with logical mapping and negotiation
2. **E-commerce:** Enum mapping and invalid code rejection
3. **IoT:** Range validation for sensor data
4. **FinTech:** Regex pattern enforcement with hard circuit breakers

2 Methodology

2.1 Contract Specification Format

All contracts follow the **Open Data Contract Standard (ODCS) v0.9.3**, structured as YAML documents with the following sections:

- `info`: Ownership, contact, classification
- `schema`: Logical field definitions with types and constraints
- `sla`: Freshness, availability, retention guarantees
- `quality`: Executable validation rules with enforcement policies
- `lineage`: Source systems and consumer registrations

2.2 Validation Infrastructure

A custom Python validation script performs:

```
$ python validate_contracts.py
```

Listing 1: Validation Script Execution

1. **YAML Syntax Validation:** Using `yamllint`
2. **Structural Validation:** Verifying ODCS required sections
3. **Scenario-Specific Validation:** Checking business rules, PII tagging, quality thresholds

3 Implementation Details

3.1 Scenario 1: Ride-Share (CityMove)

3.1.1 Problem Statement

The Dynamic Pricing ML model crashed for 4 hours when the database column `cost_total` was renamed to `fare_final`, costing $50,000 in lost revenue.

3.1.2 Solution

Implemented a stable logical interface that decouples the ML model from physical schema changes.

Key Features:

- **Logical Mapping:**
  - `r_id` → `ride_id`
  - `ts_start` → `pickup_timestamp`
  - `pax_id` → `passenger_id`
  - `d_rating` → `driver_rating`
  - `fare_final` → `fare_amount`
  - `dist_m` → `distance_meters`
- **PII Tagging:** `passenger_id` marked as `pii: true`
- **SLA:** 30-minute freshness threshold for model retraining
- **Quality Rules:**
  1. `fare_amount >= 0` (prevents negative fares)

2. `driver_rating >= 1.0 AND driver_rating <= 5.0`
3. `distance_meters` completeness: 100%

Negotiation Summary: Documented in contract header, specifying producer (Data Engineering), consumer (ML Team), and agreed trade-offs.

3.2 Scenario 2: E-commerce (Flash Sale)

3.2.1 Problem Statement

Marketing dashboard crashed during Black Friday when it received `status_code: 7`, which had no handler. Negative order totals from a bug also corrupted revenue reports.

3.2.2 Solution

- **Enum Mapping:** Physical codes mapped to logical states:
  - `2` → `PAID`
  - `5` → `SHIPPED`
  - `9` → `CANCELLED`
- **Quality Rules:**
  1. `order_total >= 0` (hard enforcement)
  2. `status IN ('PAID', 'SHIPPED', 'CANCELLED')`
  3. `status_code IN (2, 5, 9)` — rejects unmapped codes

3.3 Scenario 3: IoT (Smart Thermostat)

3.3.1 Problem Statement

Failed sensors default to `9999°C`, skewing the "Average Home Temperature" metric. Battery readings outside [0.0, 1.0] indicate malfunction.

3.3.2 Solution

- **Schema Constraints:**
  - `temperature_c`: `minimum: -30`, `maximum: 60`
  - `battery_level`: `minimum: 0.0`, `maximum: 1.0`
- **Quality Rules:**
  1. `temperature_c >= -30 AND temperature_c <= 60`
  2. `battery_level >= 0.0 AND battery_level <= 1.0`

  Both rules use `enforcement: hard` to reject invalid readings at the publication gate.

3.4 Scenario 4: FinTech (Transaction Log)

3.4.1 Problem Statement

Legacy system upgrade caused some account IDs to be 8 characters instead of 10, causing fraud detection model to silently fail lookups.

3.4.2 Solution

- **Regex Pattern:** `^[A-Z0-9]{10}$`
  - Exactly 10 characters
  - Uppercase letters (A-Z) and digits (0-9) only
- **Hard Circuit Breaker:**
  - `enforcement: hard` — blocks pipeline on violation
  - Description explicitly states: "BLOCK PIPELINE on violation"
- Applied to both `account_source_id` and `account_destination_id`

4 Validation and Verification

4.1 Validation Approach

All contracts were validated using a two-tier approach:

1. **Syntax Validation:** `yamllint` ensures YAML correctness
2. **Semantic Validation:** Custom Python script validates ODCS structure and scenario requirements

5 Key Design Decisions

5.1 Logical vs. Physical Separation

All contracts expose *logical* field names (e.g., `fare_amount`) while documenting *physical* mappings (e.g., `physical_mapping: fare_final`). This allows:

- Database refactoring without breaking consumers
- Business-friendly naming conventions
- Centralized transformation logic

5.2 Enforcement Policies

All quality rules use `enforcement: hard`, meaning violations **block the pipeline** rather than just logging warnings. This is appropriate for:

- Financial data (FinTech)
- ML model inputs (Ride-Share)
- Compliance-critical systems

5.3 Validation at Publication Gate

Contracts are enforced at the *publication gate* (Point C in the data pipeline), ensuring consumers only receive validated data. This prevents:

- Cascading failures downstream
- Silent data quality degradation
- Consumer-side defensive programming

6 Conclusion

Data contracts successfully decouple consumers from physical storage details and provide a robust mechanism for data governance. By implementing these contracts, organizations can shift from reactive firefighting to proactive quality assurance.

6.1 Benefits Demonstrated

- **Reliability:** Circuit breakers prevent bad data from propagating
- **Agility:** Producers can refactor without breaking consumers
- **Governance:** PII tagging enables automated compliance
- **Observability:** Quality metrics provide early warning signals

6.2 Future Work

- Integrate contracts into CI/CD pipelines
- Implement contract versioning and deprecation policies
- Build contract registry for discovery and lineage tracking
- Add automated SLA monitoring and alerting