ETL Data Pipeline for Loan Default Prediction

Business Problem

Financial institute needs ML-ready data for loan default prediction and risk management

Raw CSVs → Bronze → Silver → Gold → ML Models

⚠️ **Pipeline Scope: Delivers Gold Feature + Label Stores — maximize data cleaning & engineering while keeping ML options open until ML development requirements finalized**

- ✓Production-grade ETL pipeline following Medallion Architecture
- ✓Processing 24 months of data across 4 data sources
- ✓Enable data-driven loan decisions and risk management

Data Strategy & Architecture

Building trust through progressive data refinement

In Plain English: We process data in three stages, each improving quality—like refining raw ore into pure gold.

🥉 **Bronze:** Store raw data exactly as received (audit trail)

🥈 **Silver:** Clean & validate data (fix errors, standardize formats)

🥇 **Gold:** Business-ready data (optimized for ML models)

💡 **Business Value:** Each layer has clear purpose—bronze for compliance, silver for quality, gold for ML

TECHNICAL PySpark for distributed processing + Parquet for columnar storage & schema evolution

Prediction Framework

When and how do we predict loan defaults?

The Question: At loan application time (Month 0), can we predict if the customer will default 6 months later?

Month 0
Apply for loan

Month 6
Observe outcome

30+ DPD
= Default

⚠️ **Default = 30+ Days Past Due (DPD) at Month 6 — this is our prediction target**

📊 Demographics

💰 Financials

🖱️ Clickstream

📈 Loan History

💡 **Why Month 6?** Data shows default patterns stabilize at this point—early enough to act, late enough to be reliable

Data Sources & Integration

What data do we have to work with?

The Challenge: Data lives in 4 separate files. We need to combine them by matching customer IDs.

📈 **lms_loan_daily.csv**
Loan performance over time (137K+ records - tracks each loan by month)

💰 **features_financials.csv**
Credit scores & income (12,500 customers - one snapshot per person)

👤 **features_attributes.csv**
Age, occupation, family status (12,500 customers - one snapshot per person)

🖱️ **feature_clickstream.csv**
Website activity (215K+ records across 8K+ customers - monthly snapshots)

💡 **Integration Strategy:** Match all data by Customer_ID, keeping all loan records even if some customer data is missing (left join)

Data Quality Strategy

How do we handle imperfect data?

The Challenge: Real-world data is messy. Do we delete bad records or try to save them?

💡 **Our Philosophy: "Flag but Preserve" — Keep data for ML training, but mark quality issues**

FLAG_ONLY	DROP_ROW	QUARANTINE	DEFAULT_VALUE
Mark issue, keep data	Delete if critical	Isolate extreme outliers	Fill missing values

💡 **Why This Matters:** Maximizes ML training data while maintaining transparency about data quality

TECHNICAL SIMPLIFIED (actual code in utils/data_processing_silver_table.py:20-26, 96-103, 132-136)

```python
# Enum definition (lines 20-26)
class InvalidDataStrategy(Enum):
    FLAG_ONLY = "flag_only"     # Keep record, add validation flag
    DROP_ROW = "drop_row"        # Remove entire record
    QUARANTINE = "quarantine"   # Move to separate table

# Usage example: Conservative approach
df = validate_and_handle_non_negative_amount(
    df, "Annual_Income", strategy=InvalidDataStrategy.FLAG_ONLY
)

# Quarantine logic handles extreme outliers (lines 132-136)
```

Silver Layer: Modular Processing

```
BaseSilverProcessor (ABC)
├── ValidationMixin (data quality methods)
├── LoanDailySilverProcessor (strict financial)
├── FinancialsSilverProcessor (conservative + quarantine)
├── AttributesSilverProcessor (moderate validation)
└── ClickstreamSilverProcessor (lenient defaults)
```

Data Handling Philosophy

- **Financial:** Strict validation for critical fields, flagging for analysis
- **Clickstream:** Lenient defaults (missing clicks = 0)
- **Demographics:** Flag edge cases but preserve for ML
- **Data Lineage:** Keep SSN/Name in silver, exclude from gold

✓Clean, extensible architecture with SilverProcessorFactory

✓Data-source-specific validation rules

✓Easy to add new data sources

Gold Layer: ML-Ready Stores

Feature Store Design

Time-series features joined on **Customer_ID** with **mob=0** (application time)

Feature Engineering Highlights

- **Credit_History_Age:** "10 Years 9 Months" → Credit_History_Age_Months (integer)
- **Payment_Behavior:** Extract spending_level (binary) + value_size (ordinal 0-2)
- **Payment_of_Min_Amount:** One-hot encode YES/NO/NM
- **Credit_Mix:** GOOD/STANDARD/BAD with missing as reference
- **Type_of_Loan:** Multi-label binarization (has_payday_loan, has_mortgage_loan, etc.)

Label Store Logic

Binary classification: 1 = default (30+ DPD at mob=6), 0 = performing

✓Point-in-time correctness (no future leakage)

✓Incremental processing with date-based partitioning

Preventing Data Leakage #1

Making sure we don't "cheat" by using future information

The Problem: Customer info changes over time. If we use 2024 data to predict a 2023 loan, we're cheating—using information we didn't have yet!

⏰ **Rule: Only use customer data that existed AT or BEFORE loan application time (Month 0)**

💡 **Why This Matters:** Ensures model predictions are realistic—we only use information that would have been available when making the actual loan decision

TECHNICAL SIMPLIFIED (actual code in utils/data_processing_gold_table.py:92, 258-265)

```
# 1. Filter to Month 0 (application time)
df_features = loan_daily.filter(col("mob") == 0)

# 2. Join with temporal constraint
df_features = df_features.join(
    attributes_table,
    on=[Customer_ID match,
        attributes.snapshot_date <= loan.snapshot_date],  # KEY!
    how="left"
)
```

Preventing Data Leakage #2

Handling time-series customer behavior data

The Problem: Clickstream has many rows per customer (one per month snapshot). We don't know loan application dates until Gold layer.

📊 **Solution at Gold Layer: When creating features for each loan (mob=0), look back 6 months and average the customer's clickstream activity**

💡 **Why 6 Months?** Arbitrary choice—captures recent behavior patterns. Can be tuned as a hyperparameter (3, 9, 12 months, etc.) based on model performance

TECHNICAL SIMPLIFIED (actual code in utils/data_processing_gold_table.py:305-341)

```
# At Gold layer (after we know loan application dates):
# 1. For each loan at mob=0, define lookback window
window_start = F.date_sub(col('loan_application_date'), 180)  # 6 months back

# 2. Join all clickstream snapshots within window
df_with_clickstream = df_features.join(clickstream,
    on=[Customer_ID match,
        clickstream.snapshot_date >= window_start,
        clickstream.snapshot_date <= loan_application_date],  # Past only!
    how="left")

# 3. Group by loan and calculate average across those months
agg_exprs = [F.avg(f"fe_{i}").alias(f"avg_fe_{i}_last_6m")
             for i in range(1, 21)]
```

Pipeline Boundary & Next Steps

Data Pipeline Complete ✓

ML-ready feature store and label store (can be joined on `loan_id`)

🔀 **Handoff Point: Data Pipeline → ML Pipeline**

ML Pipeline: Data Preparation
Join feature store + label store → training dataset

ML Pipeline: Data Splitting
Create train/validation/test sets + OOT (Out-of-Time) dataset for temporal validation

ML Pipeline: Model Development
Additional feature engineering, model training, evaluation, deployment

Foundation for ML Operations **Clear Pipeline Separation**