Medallion Architecture

Complete Guide to Data Quality Layers

Databricks 14-Days AI Challenge

Day 6: Multi-Hop Data Architecture

January 14, 2026

Agenda

- **Introduction** to Medallion Architecture
- **Why** Medallion Architecture?
- **Bronze Layer**: Raw Data Ingestion
- **Silver Layer**: Cleaned & Validated Data
- **Gold Layer**: Business Aggregates
- **Data Flow** & Transformations
- **Incremental Processing** Patterns
- **Best Practices** for Each Layer

Introduction to Medallion Architecture

Definition

Medallion Architecture (Multi-Hop Architecture) is a data design pattern used to logically organize data in a **lakehouse**.

Raw Data
As-Is Storage

Cleaned
Validated

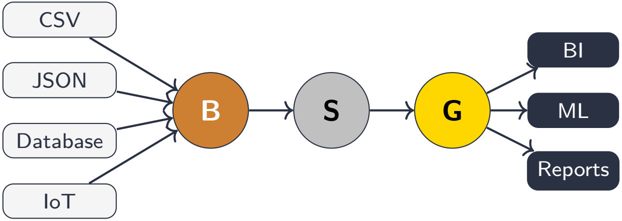
Aggregates
Business-Ready

Principle: Data quality and structure improve as data moves from Bronze to Gold.

The Refining Analogy

Think of it like refining raw ore:

- **Raw ore (Bronze)**
  - ▷ Contains impurities
  - ▷ Unprocessed material
- **Refined metal (Silver)**
  - ▷ Purified
  - ▷ Standardized
- **Finished jewelry (Gold)**
  - ▷ Ready for use
  - ▷ High value



Why Medallion Architecture?

Problems It Solves:

- **Data Quality Issues**
  - Progressive refinement
- **Debugging Difficulties**
  - Raw data preserved in Bronze
- **Schema Changes**
  - Transformations in Silver
- **Reprocessing Needs**
  - Replay from Bronze

Key Benefits:

1. **Data Lineage** - Traceable transformations
2. **Replayability** - Reprocess without re-ingesting
3. **Separation of Concerns** - Clear responsibilities
4. **Quality Gates** - Validation at each layer
5. **Flexibility** - Multiple Gold tables
6. **Performance** - Pre-aggregated queries

The Three Layers at a Glance

Aspect	**Bronze**	**Silver**	**Gold**
Data Quality	Raw, may have issues	Cleaned, validated	Aggregated, business-ready
Schema	Schema-on-read	Schema enforced	Highly structured
Duplicates	May exist	Removed	N/A (aggregated)
Transformations	None (only metadata)	Filtering, cleaning	Aggregations
Users	Data engineers	Engineers, analysts	Business users, BI
Update Pattern	Append-only	Upsert/Merge	Overwrite or Merge

Bronze Layer: Raw Data Ingestion

What is the Bronze Layer?

The **landing zone** for all raw data. Captures data exactly as received with minimal transformation.

Characteristics:

- Raw and Unmodified
- Append-Only pattern
- Full History preserved
- Metadata Enriched
- Schema-on-Read

Formula:

```
Source Data + Metadata =
Bronze
```

Common Metadata Fields:

- `ingestion_timestamp`
- `source_file`
- `source_system`
- `batch_id`

Bronze Layer: Code Example

Listing 1: Bronze Layer Ingestion

```
# BRONZE: Raw ingestion
raw = spark.read.csv(
    "/raw/events.csv",
    header=True,
    inferSchema=True
)

# Add metadata and save to Bronze
raw.withColumn("ingestion_ts", F.current_timestamp()) \
   .write.format("delta") \
   .mode("overwrite") \
   .save("/delta/bronze/events")
```

Code Element	**Purpose**
`header=True`	First row contains column names
`inferSchema=True`	Auto-detect data types
`ingestion_ts`	When data was ingested
`format("delta")`	ACID transactions enabled

Bronze Layer: Best Practices

Practice	Description	Reason
Never modify raw data	Store as received	Audit trails
Use append mode	Add, don't overwrite	Preserve history
Add ingestion metadata	Timestamp, source	Debugging
Partition by date	Ingestion date	Efficient processing
Keep original schema	Don't rename columns	Source comparison
Use Delta format	ACID, time travel	Reliability

Silver Layer: Cleaned and Validated Data

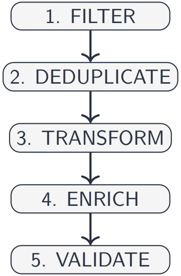
What is the Silver Layer?

Contains **cleaned, validated, and enriched data**. Quality rules applied, duplicates removed.

Characteristics:

- Cleaned Data
- Validated (business rules)
- Deduplicated
- Enriched (derived columns)
- Schema Enforced
- Joined (multiple sources)

Common Operations:



Silver Layer: Code Example

Listing 2: Silver Layer Cleaning

```
# SILVER: Cleaned data
bronze = spark.read.format("delta").load("/delta/bronze/events")

silver = bronze \
    .filter(F.col("price") > 0) \
    .filter(F.col("price") < 10000) \
    .dropDuplicates(["user_session", "event_time"]) \
    .withColumn("event_date", F.to_date("event_time")) \
    .withColumn("price_tier",
        F.when(F.col("price") < 10, "budget")
         .when(F.col("price") < 50, "mid")
         .otherwise("premium"))

silver.write.format("delta").mode("overwrite") \
      .save("/delta/silver/events")
```

Silver Layer: Code Breakdown

Operation	Code	Purpose
Validation	`filter(price > 0)`	Remove invalid prices
Validation	`filter(price < 10000)`	Remove outliers
Deduplication	`dropDuplicates([...])`	One record per session
Transformation	`to_date("event_time")`	Extract date
Enrichment	`F.when(...)`	Categorize prices

Price Tier Logic:

Price Range	Tier
\$0.01 - \$9.99	budget
\$10.00 - \$49.99	mid
\$50.00+	premium

Silver Layer: Best Practices

Practice	Description
Document business rules	Record why each filter exists
Handle nulls explicitly	Filter, fill default, or keep
Use meaningful names	Rename cryptic columns
Standardize formats	Consistent dates, units
Add quality flags	Indicators for cleaned records
Partition appropriately	Usually by business date
Enable schema enforcement	Use Delta schema evolution

Gold Layer: Business Aggregates

What is the Gold Layer?

Contains **business-level aggregates and metrics** ready for consumption by analysts and BI tools.

Characteristics:

- Pre-aggregated
- Business-Focused
- Consumption-Ready
- Multiple Domain Tables
- Denormalized

Example Gold Tables:

product_performance
views, purchases, revenue

user_metrics
total_spent, ltv

daily_summary
revenue, orders

Gold Layer: Code Example

Listing 3: Gold Layer Aggregation

```
# GOLD: Aggregates
silver = spark.read.format("delta").load("/delta/silver/events")

product_perf = silver.groupBy("product_id", "product_name") \
    .agg(
        F.countDistinct(F.when(F.col("event_type")=="view",
                               "user_id")).alias("views"),
        F.countDistinct(F.when(F.col("event_type")=="purchase",
                               "user_id")).alias("purchases"),
        F.sum(F.when(F.col("event_type")=="purchase",
                     "price")).alias("revenue")
    ).withColumn("conversion_rate",
                 F.col("purchases")/F.col("views")*100)

product_perf.write.format("delta").mode("overwrite") \
            .save("/delta/gold/products")
```

Gold Layer: Conversion Rate Calculation

Formula

$$\text{Conversion Rate} = \frac{\text{Purchases}}{\text{Views}} \times 100$$

Example Calculation:

- Views: 1000 unique users
- Purchases: 50 unique users
- Conversion Rate: $\frac{50}{1000} \times 100 = 5\%$

Sample Gold Table Output:

product_id	product_name	views	purchases	revenue	conv_rate
P001	Laptop	5000	250	$249,750	5.0%
P002	Mouse	8000	1200	$35,880	15.0%
P003	Keyboard	3500	420	$20,958	12.0%

Gold Layer: Best Practices

Practice	Description
Design for consumers	Understand user questions
Pre-calculate metrics	Avoid query-time calculations
Use meaningful names	`conversion_rate` not `cr`
Document calculations	Record formulas, definitions
Consider multiple tables	Different domains
Optimize for queries	Partition by filter columns
Include timestamps	When was aggregate updated?

Incremental Processing Patterns

Why Incremental Processing?

Processing all data every time is expensive. Handle only new or changed data.

Pattern	When to Use	Pros	Cons
Append	New data only	Simple, fast	Duplicates if replayed
Merge	Updates to records	Handles updates	More complex
Overwrite Partition	Time-based data	Idempotent	Full partition reprocess
CDC	Real-time updates	Efficient	Requires infrastructure

Incremental Processing: Merge Pattern

Listing 4: Merge/Upsert Pattern

```
from delta.tables import DeltaTable

silver_table = DeltaTable.forPath(spark, "/delta/silver/events")
new_bronze = spark.read.format("delta").load("/delta/bronze/events") \
    .filter(F.col("ingestion_ts") > last_silver_update)

silver_table.alias("target").merge(
    new_bronze.alias("source"),
    "target.event_id = source.event_id"
).whenMatchedUpdate(set={
    "price": "source.price",
    "updated_ts": "current_timestamp()"
}).whenNotMatchedInsert(values={
    "event_id": "source.event_id",
    "price": "source.price"
}).execute()
```

Patterns to Follow vs. Anti-Patterns

Do This:

- **Single Responsibility**
  - ▷ Each layer does one thing well
- **Idempotent Pipelines**
  - ▷ Running twice = same result
- **Schema Evolution**
  - ▷ Plan for changes
- **Metadata Tracking**
  - ▷ Know data origin
- **Modular Gold**
  - ▷ Multiple tables for use cases

Don't Do This:

- **Transforming in Bronze**
  - ▷ Loses raw data
- **Skipping Silver**
  - ▷ No validated layer
- **One Mega Gold Table**
  - ▷ Slow queries
- **No Incremental Logic**
  - ▷ Reprocesses everything
- **Missing Metadata**
  - ▷ Can't track lineage

Summary: Key Takeaways

1. **Medallion Architecture** provides structured approach to organizing lakehouse data

2. **Three layers serve distinct purposes:**
   - **Bronze**: Raw data preservation + metadata
   - **Silver**: Cleaned, validated, enriched data
   - **Gold**: Business-ready aggregates

3. Each layer **builds on the previous**, improving data quality progressively

4. **Incremental processing** is essential for production systems

5. **Best practices** ensure maintainability, performance, and reliability

Quick Reference

Thank You!

Day 6: Medallion Architecture Complete

Questions?

LinkedIn: Yash Kavaiya | Gen AI Guru