Here is the comprehensive, deep-dive documentation for **Project 5: The Medallion Architect**.

This project represents the **Gold Standard** of modern Data Engineering. You are moving away from traditional Data Warehousing (Project 4) to the **Lakehouse Architecture**. This is the architecture used by 80% of modern enterprises (Netflix, Uber, Shell, etc.).

PROJECT 5: THE MEDALLION ARCHITECT (Lakehouse)

Role: Lakehouse Architect

Difficulty: Medium (Builder)

Est. Time: 8-10 Hours

---

0. 📚 Topic Brush-Up (Read Before You Start)

To succeed in this project, you must understand the "Lakehouse" paradigm. These concepts are distinct from the SQL Warehouse concepts in Project 4.

A. The Medallion Architecture (Multi-Hop)

- **Bronze Layer (Raw):** The landing zone. Data here is "as-is" from the source. It is append-only and immutable. If the source sends garbage, we store garbage. *Goal: Never lose history.*
- **Silver Layer (Cleaned/Enriched):** The filtering zone. We remove duplicates, fix data types, and join tables (e.g., matching Orders to Customers). This is "Enterprise Quality" data.
- **Gold Layer (Aggregated):** The consumption zone. Data is aggregated for specific use cases (e.g., "Daily Sales by Region"). This is read by Power BI.

B. Delta Lake Internals

- **ACID Transactions on Spark:** Standard Parquet files don't support transactions. If a job fails halfway, you get corrupt files. **Delta Lake** adds a transaction log (_delta_log) to guarantee either *all* data is written or *none* is.
- **Time Travel:** Because Delta keeps a log of every change, you can query data as it existed yesterday using VERSION AS OF or TIMESTAMP AS OF.

- **Schema Enforcement:** Delta prevents "bad data" (wrong columns) from corrupting your table, rejecting writes that don't match the schema.

C. Spark Optimization

- **OPTIMIZE & Z-ORDER:** A specific command that compacts small files into large ones and co-locates related data (like indexing) to make queries 100x faster.

---

1. Executive Summary & Scenario

The Business Problem:

"ShopSmart," a rapidly growing e-commerce platform, is drowning in data.

- **Scenario:** They have raw logs of **Orders**, **Customers**, and **Payments** dumping into the lake every hour.
- **The Issue:**
  - Data Scientists want the **Raw** data to look for hidden patterns.
  - The Finance Team wants **Clean** data to calculate taxes.
  - The CEO wants a **Dashboard** showing "Top Selling Categories."
  - *Current State:* Everyone is querying the raw JSON files, resulting in slow performance and conflicting numbers.

The Goal:

Build a **Medallion Architecture** pipeline in Azure Databricks.

1. **Ingest** raw data into a **Bronze** Delta Table.
2. **Refine** and merge data into a **Silver** Delta Table (joining Orders + Customers).
3. **Aggregate** KPIs into a **Gold** Delta Table for the CEO.

---

2. Architecture Overview

We are building a "Lakehouse"—combining the low cost of a Data Lake with the reliability of a Warehouse.

The Stack:

- **Storage:** Azure Data Lake Gen2 (Separated into 3 containers: bronze, silver, gold).
- **Compute:** Azure Databricks (PySpark & SQL).
- **Format:** Delta Lake (The open-source storage layer).

---

3. Pre-Requisites (Data Generator)

We need relational data (Orders linked to Customers). Run this Python script locally to generate the raw files.

Action:

1. Save as generate_ecommerce_data.py.
2. Run it. You will get orders.csv and customers.csv.
3. Upload them to ADLS:
    - raw-source/orders/orders.csv
    - raw-source/customers/customers.csv

Python

```python
import csv
import random
from datetime import datetime, timedelta

def generate_data():
    # 1. Generate Customers
    customers = []
    print("Generating Customers...")
    for i in range(1, 101): # 100 Customers
        customers.append({
            "customer_id": f"C{i:03d}",
            "name": f"Customer_{i}",
            "city": random.choice(["New York", "London", "Tokyo", "Paris"]),
            "email": f"user{i}@example.com"
        })

    with open("customers.csv", "w", newline='') as f:
        w = csv.DictWriter(f, fieldnames=customers[0].keys())
        w.writeheader()
        w.writerows(customers)

    # 2. Generate Orders
    orders = []
    print("Generating Orders...")
    for i in range(1, 1001): # 1000 Orders
```

```
        order_date = datetime.now() - timedelta(days=random.randint(0, 30))
        orders.append({
            "order_id": f"O{i:05d}",
            "customer_id": f"C{random.randint(1, 100):03d}", # Link to Customer
            "amount": round(random.uniform(10.0, 500.0), 2),
            "order_date": order_date.strftime("%Y-%m-%d"),
            "status": random.choice(["COMPLETED", "PENDING", "CANCELLED"])
        })

    with open("orders.csv", "w", newline='') as f:
        w = csv.DictWriter(f, fieldnames=orders[0].keys())
        w.writeheader()
        w.writerows(orders)

    print("✅ Data Generated: customers.csv, orders.csv")

if __name__ == "__main__":
    generate_data()
```

---

4. The Design Phase (Interactive)

As the Architect, you must define the transformation logic for each layer.

Challenge A: The Silver Logic (Normalization)

Scenario: The raw orders file has a customer_id. The raw customers file has email and city.

Task: In the **Silver** layer, do we join them into one big "Master Table"? Or do we keep them separate but cleaned?

- *Design Principle:* Usually, Silver stays "Normalized" (3rd Normal Form) or "Denormalized" depending on read patterns.
- *Decision:* Let's Denormalize (Join) them in Silver to make Gold aggregations faster. We will create a Silver_Sales_Enriched table.

Challenge B: Handling Updates (Upserts)

Scenario: An order status changes from PENDING to COMPLETED.

Task: If we just "Append" the new row to Silver, we will have two rows for Order #101.

- *Solution:* We need a **Merge (Upsert)** strategy. "If Order ID matches, Update the Status. If not, Insert."

Challenge C: Gold Aggregation

Scenario: The CEO wants "Total Sales by City by Day."

Task: Define the Schema for the Gold Table.

- *Columns:* City, OrderDate, TotalAmount, OrderCount.

---

5. Implementation Guide (Step-by-Step)

Phase 1: Environment Setup

1. **ADLS Gen2:** Create 3 containers: bronze, silver, gold.
2. **Databricks:** Create a Notebook named Medallion_Pipeline.
3. **Mounting/Access:** Use the code from Project 3 to connect to your storage.

Phase 2: The Bronze Layer (Raw Ingestion)

Goal: Convert CSV to Delta (Parquet with Logs). Speed is key.

Python

```
# 1. Read Raw CSVs
df_orders_raw = spark.read.format("csv").option("header",
"true").load("abfss://raw-source@.../orders.csv")
df_cust_raw = spark.read.format("csv").option("header",
"true").load("abfss://raw-source@.../customers.csv")

# 2. Add Metadata (Ingestion Date)
from pyspark.sql.functions import current_timestamp
df_orders_bronze = df_orders_raw.withColumn("ingestion_date", current_timestamp())
df_cust_bronze = df_cust_raw.withColumn("ingestion_date", current_timestamp())

# 3. Write to Bronze (Delta Format)
# We overwrite here for simplicity, but in prod, this would be .mode("append")
df_orders_bronze.write.format("delta").mode("overwrite").save("abfss://bronze@.../orders")
df_cust_bronze.write.format("delta").mode("overwrite").save("abfss://bronze@.../customers")

print("✅ Bronze Layer Populated")
```

Phase 3: The Silver Layer (Cleaning & Joining)

Goal: Remove Cancelled orders, Join with Customers, Enforce Schema.

Python

```
# 1. Read from Bronze
bronze_orders = spark.read.format("delta").load("abfss://bronze@.../orders")
bronze_cust = spark.read.format("delta").load("abfss://bronze@.../customers")

# 2. Transformation Logic
# - Filter out CANCELLED orders
# - Cast 'amount' to Double (it was string in CSV)
from pyspark.sql.functions import col

silver_orders = bronze_orders.filter(col("status") != "CANCELLED") \
                             .withColumn("amount", col("amount").cast("double"))

# 3. Join with Customers (Enrichment)
# We want City and Name attached to the order
silver_enriched = silver_orders.join(bronze_cust, "customer_id", "inner") \
                               .select(
                                  "order_id", "customer_id", "name", "city",
                                  "amount", "order_date", "status"
                               )

# 4. Write to Silver
# Using MERGE (Upsert) is complex in PySpark API, so for this step we will use "overwrite"
# to simulate a full reload. In advanced projects, we use DeltaTable.merge().
silver_enriched.write.format("delta").mode("overwrite").save("abfss://silver@.../sales_enriched")

print("✅ Silver Layer Populated (Cleaned & Joined)")
```

Phase 4: The Gold Layer (Business Aggregates)

Goal: Create the "CEO's Dashboard" table.

Python

```
# 1. Read from Silver
silver_df = spark.read.format("delta").load("abfss://silver@.../sales_enriched")

# 2. Aggregate
from pyspark.sql.functions import sum, count

gold_daily_sales = silver_df.groupBy("city", "order_date") \
                    .agg(
                      sum("amount").alias("total_revenue"),
                      count("order_id").alias("total_orders")
                    )

# 3. Write to Gold
gold_daily_sales.write.format("delta").mode("overwrite").save("abfss://gold@.../daily_sales_by_city")

print("✅ Gold Layer Populated. Ready for Power BI.")
```

Phase 5: Optimization (The "Senior" Step)

Delta tables can get fragmented. We need to optimize them.

SQL

```
%sql
-- Register the table first
CREATE TABLE IF NOT EXISTS gold_sales
USING DELTA
LOCATION 'abfss://gold@.../daily_sales_by_city';

-- Compact small files and index by City (Z-Order)
OPTIMIZE gold_sales ZORDER BY (city);
```

---

6. Verification & Time Travel

Verification 1: Check the Gold Data

Python

```
display(spark.read.format("delta").load("abfss://gold@.../daily_sales_by_city"))
```

- *Success Criteria:* You should see a neat table with City, Date, Revenue. No "Cancelled" orders included.

Verification 2: Time Travel (The "Magic" of Delta)

Let's see what the Silver table looked like *before* we ran the job (if you ran it twice).

SQL

```
%sql
-- View history of operations
DESCRIBE HISTORY delta.`abfss://silver@.../sales_enriched`
```

You can query version 0 using:

Python

```
df_v0 = spark.read.format("delta").option("versionAsOf", 0).load("abfss://silver@.../sales_enriched")
display(df_v0)
```

---

7. Project Defense (Thesis Questions)

Q1. The "GDPR" Delete

"A user (Customer_42) invokes their 'Right to be Forgotten'. You need to delete them from Bronze, Silver, and Gold. How do you do this efficiently in Delta Lake? Does standard DELETE FROM physically remove the data immediately, or does it just mark it? How do you ensure it is legally 'gone' (Vacuum)?"

Q2. Schema Evolution

"Next month, the source system adds a discount_code column to the Orders CSV. Your current write.save() command might fail or ignore it. How do you enable .option("mergeSchema", "true") to automatically update your Delta table structure?"

Q3. Streaming vs. Batch

"We built this as a Batch job (running once a day). If we wanted to make this Real-Time using spark.readStream, what changes would we need to make to the Silver and Gold logic? (Hint: Can you overwrite a table in Streaming? Or must you Append?)"

Q4. Partitioning Strategy

"In the Gold table, we Z-Ordered by City. Why didn't we Partition by Date? What is the rule of thumb for file size before you decide to create a physical partition folder?"

Ready for the Deep End?

Completing this project marks your transition from "ETL Developer" to **"Data Engineer"**.

Reply **"Project 5 Done"** when you have successfully run the Medallion Pipeline and tested Time Travel.

We will then tackle **Project 6**, which deals with **Complex Nested JSON**—a notoriously difficult interview topic.