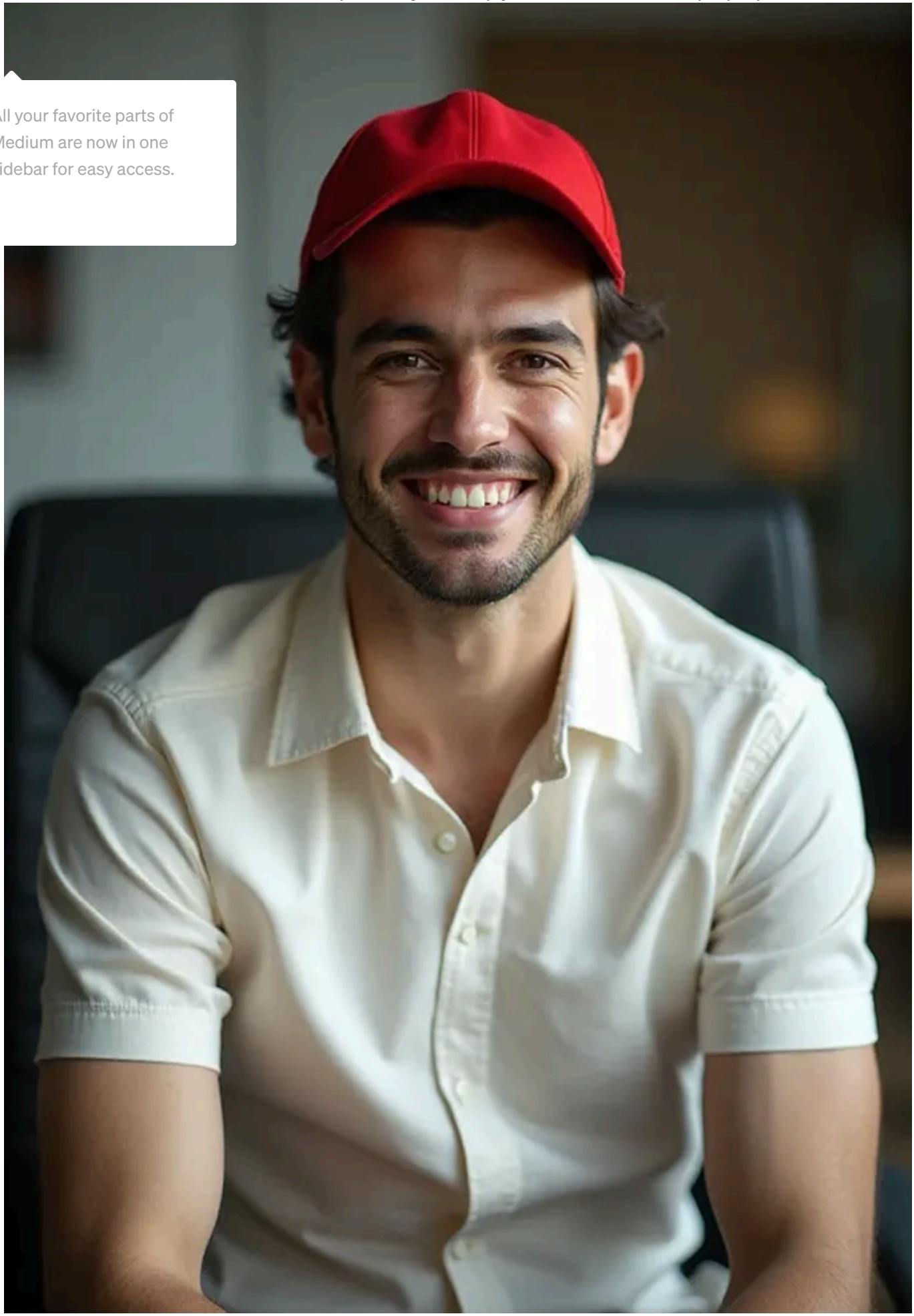
🚀 End-to-End Retail Sales Analytics ETL Pipeline with PySpark (Scalable & Production-Ready)

Mayurkumar Surani Follow 8 min read · Sep 5, 2025

Category: Data Engineering, Big Data, PySpark, ETL, Analytics

🔍 Why This Matters: Business-Driven Data Engineering

In today's data-driven retail landscape, timely, accurate insights from sales data are critical for decision-making. From understanding customer behavior to optimizing inventory and forecasting revenue, ETL (Extract, Transform, Load) pipelines form the backbone of analytics infrastructure.



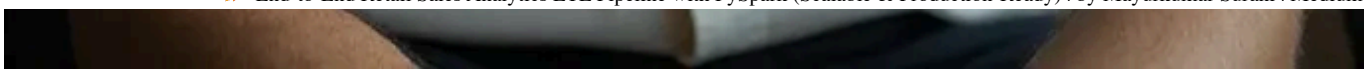


Image by author

u through building a scalable, production-grade ETL he Spark (PySpark) that:

- *Ingests raw retail sales data*
- *Cleans and validates it*
- *Computes key business KPIs*
- *Stores results in an efficient, query-optimized format*

🎯 Target Audience:

- *Data Engineers*
- *Analytics Engineers*
- *Data Scientists*
- *Tech Leads building data platforms*

💡 **Use Case:** A retail company wants to analyze daily sales trends across stores and products to improve inventory planning and marketing strategies.

🧩 Problem Statement: Turning Raw Data into Business Value

📌 Business Challenge

A national retail chain collects transactional data from hundreds of stores. The data arrives in raw JSON files (e.g., from POS systems), but it's

inconsistent, contains duplicates, and lacks structure. Business teams need clean, reliable metrics like:

ıcts

- *Store performance*
- *Monthly sales trends*

Without automation, this process is manual, error-prone, and slow.

📌 Technical Goals

We'll build a modular, reusable, and scalable ETL pipeline that:

1. *Ingests raw sales data*
2. *Cleans and validates data quality*
3. *Transforms into a structured format*
4. *Computes business KPIs*
5. *Stores outputs in Parquet (columnar format) for fast querying*

🔧 **Tech Stack:** PySpark, Python, Local File System (can scale to cloud)

🏗️ Project Architecture: Scalable & Modular Design

We follow a layered, production-ready architecture to ensure maintainability and scalability.

```
retail-etl-pipeline/
|
                                → Spark configurations
                                → Input/output (raw, processed, metrics)

                                → Reusable functions (logging, file ops)
                    /           → Load raw data
                    ation/      → Clean and validate
    ├── metrics/                → Compute KPIs
    └── main.py                 → Orchestrate pipeline
├── tests/                      → Unit & integration tests
├── generate_sample_data.py     → Simulate input
├── requirements.txt            → Dependencies
└── README.md                   → Documentation
```

Best Practices Applied:

- *Separation of concerns*
- *Idempotent operations*
- *Schema enforcement*
- *Logging and error handling*
- *Extensible for cloud (S3, Delta Lake, etc.)*

📩 Step 1: Generate Sample Sales Data (Simulating Real-World Source)

Before processing, we simulate real-world data ingestion.

`generate_sample_data.py`

```
"""
Simulate daily sales data from retail POS systems.
and development.

ort datetime, timedelta

# Configuration
NUM_RECORDS = 1000
OUTPUT_PATH = "data/raw/sales_raw.json"

PRODUCTS = ["Laptop", "Phone", "Tablet", "Headphones", "Mouse", "Keyboard"]
STORES = ["Store_A", "Store_B", "Store_C"]
CITIES = ["New York", "Los Angeles", "Chicago", "Houston", "Phoenix"]

def generate_sales_data():
    records = []
    start_date = datetime(2024, 1, 1)

    for _ in range(NUM_RECORDS):
        sale_date = start_date + timedelta(days=random.randint(0, 30))
        record = {
            "transaction_id": random.randint(10000, 99999),
            "product": random.choice(PRODUCTS),
            "category": "Electronics",
            "quantity": random.randint(1, 5),
            "price_per_unit": round(random.uniform(50, 1500), 2),
            "total_amount": None,
            "store_id": random.choice(STORES),
            "city": random.choice(CITIES),
            "sale_date": sale_date.strftime("%Y-%m-%d"),
            "sales_rep": f"Rep_{random.randint(1, 10)}"
        }
        # Recalculate total to simulate potential data inconsistency
        record["total_amount"] = round(record["quantity"] * record["price_per_un
        records.append(record)

    return records

if __name__ == "__main__":
    data = generate_sales_data()

    with open(OUTPUT_PATH, "w") as f:
        for record in data:
            f.write(json.dumps(record) + "\n")  # One JSON per line
```

```
    print(f"✅ Generated {len(data)} synthetic sales records at {OUTPUT_PATH}")
```


:

This simulates data from Point-of-Sale (POS) systems across multiple locations — a common real-world scenario where data arrives in semi-structured formats (JSON/CSV) and may have inconsistencies.

⚙️ Step 2: Configure Spark for Performance & Reliability

`config/spark_config.py`

```
"""
Centralized Spark configuration for performance tuning.
Optimized for batch processing on local or cluster mode.
"""
SPARK_CONFIG = {
    "spark.app.name": "RetailSalesETLPipeline",
    "spark.sql.adaptive.enabled": "true",  # Auto-optimize query plans
    "spark.sql.adaptive.coalescePartitions.enabled": "true",  # Reduce small fil
    "spark.sql.parquet.compression.codec": "snappy",  # Efficient storage
    "spark.serializer": "org.apache.spark.serializer.KryoSerializer",  # Faster
    "spark.sql.sources.partitionOverwriteMode": "dynamic"  # Safe overwrite in p
}
```

📌 Why This Matters:

These settings ensure high performance, low latency, and efficient resource usage — essential for scaling to terabytes of data.

🔌 Step 3: Initialize Spark & Utilities

ɹtils.py

```
"""
Initialize Spark session with logging and error handling.
"""
from pyspark.sql import SparkSession
import logging

def get_spark_session(app_name="RetailETL", config=None):
    """
    Create a Spark session with best practices.
    :param app_name: Application name for monitoring
    :param config: Dictionary of Spark configurations
    :return: SparkSession, logger
    """
    builder = SparkSession.builder.appName(app_name)

    if config:
        for key, value in config.items():
            builder.config(key, value)

    spark = builder.getOrCreate()
    spark.sparkContext.setLogLevel("WARN")  # Avoid verbose logs

    # Setup logging
    logging.basicConfig(
        level=logging.INFO,
        format='%(asctime)s [%(levelname)s] %(message)s',
        handlers=[logging.StreamHandler()]
    )
    logger = logging.getLogger(__name__)

    logger.info(f"🚀 Spark session initialized: {app_name}")
    return spark, logger
```

📌 Business Value:

Reliable logging and session management ensure operational visibility — crucial for debugging and monitoring in production.

src/utils/file_utils.py


```
tions with safety checks.

import os
from typing import List

def ensure_dir(path: str):
    """Create directory if it doesn't exist."""
    os.makedirs(path, exist_ok=True)

def list_files(directory: str, extension: str = ".json") -> List[str]:
    """List files with given extension."""
    return [
        os.path.join(directory, f)
        for f in os.listdir(directory)
        if f.endswith(extension)
    ]
```

📌 Scalability Note:

This can be extended to support cloud storage (e.g., `s3a://`, `gs://`) using Hadoop connectors.

📥 Step 4: Ingest Raw Data (Extract Phase)

src/ingestion/data_loader.py

```
"""
Load raw JSON data from file system.
Handles malformed records gracefully.
"""
from pyspark.sql import DataFrame, SparkSession
from src.utils.file_utils import list_files
import logging
```

```
def load_json_data(spark: SparkSession, input_path: str) -> DataFrame:
    """
                    es with schema inference and error tolerance.
                     Active Spark session
                    path: Path to raw data (supports wildcards)
                    rame
```



```
        df = spark.read \
            .option("mode", "DROPMALFORMED") \
            .option("columnNameOfCorruptRecord", "_corrupt_record") \
            .json(input_path)

        row_count = df.count()
        logging.info(f"📩 Successfully loaded {row_count} records from {input_pa
        return df
    except Exception as e:
        logging.error(f"❌ Failed to read data from {input_path}: {str(e)}")
        raise
```

📌 Real-World Relevance:

POS systems often produce malformed or incomplete records. Using `DROPMALFORMED` ensures pipeline resilience.

🧹 Step 5: Clean & Validate Data (Transform Phase)

`src/transformation/data_cleaner.py`

```
"""
Data cleaning and validation layer.
Ensures data quality before analytics.
"""
from pyspark.sql import DataFrame
from pyspark.sql.functions import col, when, current_timestamp
from pyspark.sql.types import *
```

```
# Define expected schema to enforce consistency
EXPECTED_SCHEMA = StructType([
    StructField("transaction_id", IntegerType(), True),
                product", StringType(), True),
                category", StringType(), True),
                quantity", IntegerType(), True),
                price_per_unit", DoubleType(), True),
                total_amount", DoubleType(), True),
                store_id", StringType(), True),
    StructField("city", StringType(), True),
    StructField("sale_date", StringType(), True),
    StructField("sales_rep", StringType(), True)
])

def clean_sales_data(df: DataFrame) -> DataFrame:
    """
    Apply data quality rules:
    - Schema enforcement
    - Null handling
    - Duplicate removal
    - Business logic validation
    """
    log = __import__('logging').getLogger(__name__)

    # Enforce schema
    df = df.select([col(f.name).cast(f.dataType) for f in EXPECTED_SCHEMA])

    # Convert string date to DateType
    df = df.withColumn("sale_date", col("sale_date").cast("date"))

    # Fill missing values with defaults
    df = df.fillna({
        "quantity": 1,
        "price_per_unit": 0.0,
        "total_amount": 0.0,
        "product": "Unknown",
        "store_id": "Unknown",
        "city": "Unknown"
    })

    # Recalculate total_amount to fix inconsistencies
    expected_total = col("quantity") * col("price_per_unit")
    df = df.withColumn(
        "total_amount",
        when(col("total_amount") != expected_total, expected_total)
        .otherwise(col("total_amount"))
    )

    # Remove duplicates
    initial = df.count()
```

```
    df = df.dropDuplicates(["transaction_id"])
    final = df.count()
    log.info(f"🧹 Removed {initial - final} duplicate transactions.")
```



```
rail
olumn("ingested_at", current_timestamp())

 Cleaned dataset contains {final} valid records.")
```

📌 Business Impact:

Poor data quality leads to wrong decisions. This step ensures trustworthy analytics by fixing common issues:

- *Missing values*
- *Inconsistent totals*
- *Duplicates*
- *Invalid dates*

📊 Step 6: Compute Business KPIs (Analytics Layer)

`src/metrics/business_metrics.py`

```
"""
Generate business intelligence from cleaned data.
"""
from pyspark.sql import DataFrame
from pyspark.sql.functions import sum, count, avg, desc, col, date_format

def compute_daily_revenue(df: DataFrame) -> DataFrame:
    """Total sales per day — for cash flow and trend analysis."""
    return df.groupBy("sale_date") \
             .agg(sum("total_amount").alias("daily_revenue")) \
```

```
            .orderBy("sale_date")

    def compute_top_products(df: DataFrame, top_n: int = 5) -> DataFrame:
                    est-selling products — for inventory and marketing."""
                    upBy("product") \
                    (
                    sum("total_amount").alias("product_revenue"),
                    sum("quantity").alias("total_units_sold")

                .orderBy(desc("product_revenue")) \
                .limit(top_n)

    def compute_store_performance(df: DataFrame) -> DataFrame:
        """Evaluate store efficiency — for operational decisions."""
        return df.groupBy("store_id", "city") \
                .agg(
                    sum("total_amount").alias("revenue"),
                    count("transaction_id").alias("transactions"),
                    avg("total_amount").alias("avg_order_value")
                ) \
                .orderBy(desc("revenue"))

    def compute_monthly_trends(df: DataFrame) -> DataFrame:
        """Monthly revenue trends — for forecasting and planning."""
        return df.withColumn("month", date_format(col("sale_date"), "yyyy-MM")) \
                .groupBy("month") \
                .agg(sum("total_amount").alias("monthly_revenue")) \
                .orderBy("month")
```


📌 Key Business Metrics Explained:

METRIC	BUSINESS USE CASE
Daily Revenue	Monitor cash flow, detect anomalies
Top Products	Optimize inventory, plan promotions
Store Performance	Identify high/low performers, allocate resources
Monthly Trends	Forecast sales, plan budgets

🚦 Step 7: Orchestrate the Pipeline


```
"""
Main ETL orchestration script.
Executes the full pipeline: Extract → Transform → Load → Metrics.
"""
from src.utils.spark_utils import get_spark_session
from src.utils.file_utils import ensure_dir
from src.ingestion.data_loader import load_json_data
from src.transformation.data_cleaner import clean_sales_data
from src.metrics.business_metrics import *
from config.spark_config import SPARK_CONFIG
import logging

# Paths
RAW_PATH = "data/raw/"
PROCESSED_PATH = "data/processed/cleaned_sales"
METRICS_DIR = "data/metrics/"

def run_pipeline():
    spark, logger = get_spark_session("RetailSalesETL", SPARK_CONFIG)

    try:
        # 1. Extract: Load raw data
        logger.info("📦 Loading raw sales data...")
        raw_df = load_json_data(spark, RAW_PATH)
        raw_df.printSchema()

        # 2. Transform: Clean and validate
        logger.info("🧹 Cleaning and validating data...")
        cleaned_df = clean_sales_data(raw_df)
        cleaned_df.cache().count()  # Cache for reuse

        # 3. Load: Save cleaned data
        logger.info("💾 Saving cleaned data in Parquet...")
        ensure_dir(PROCESSED_PATH)
        cleaned_df.write.mode("overwrite").parquet(PROCESSED_PATH)

        # 4. Metrics: Compute KPIs
        logger.info("📊 Generating business intelligence...")
        metrics = {
            "daily_revenue": compute_daily_revenue(cleaned_df),
```

```
            "top_products": compute_top_products(cleaned_df),
            "store_performance": compute_store_performance(cleaned_df),
            "monthly_trends": compute_monthly_trends(cleaned_df)
```



```
            l metrics
            -(METRICS_DIR)
             df in metrics.items():
            = f"{METRICS_DIR}/{name}"
            df.write.mode("overwrite").parquet(path)
            logger.info(f"📈 Saved metric: {name} → {path}")

        logger.info("🎉 ETL Pipeline completed successfully!")

    except Exception as e:
        logger.error(f"💥 Pipeline failed: {str(e)}")
        raise
    finally:
        spark.stop()
        logger.info("🔚 Spark session terminated.")

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

📌 Orchestration Benefits:

- Single entry point for automation
- Error resilience with try/except
- Reusable components for future pipelines

▶️ How to Run the Pipeline

1. Install Dependencies

```
pip install pyspark==3.5.0
```

2. Generate Test Data


```
ample_data.py
```

3. Run ETL

```
python src/main.py
```

4. View Results

```
# Example: Load top products
from pyspark.sql import SparkSession
spark = SparkSession.builder.appName("Test").getOrCreate()
spark.read.parquet("data/metrics/top_products").show()
```

💾 Output Structure

```
data/
├── processed/
│   └── cleaned_sales/        # Clean, structured Parquet (ready for ML/reportin
└── metrics/
    ├── daily_revenue/
    ├── top_products/
    ├── store_performance/
    └── monthly_trends/
```

✅ Optimized for Analytics:

ports column pruning, predicate pushdown, and high
l for BI tools like Power BI, Tableau, or Athena.

SEO-Friendly Summary: Key Takeaways

TOPIC	SUMMARY
ETL Pipeline	A robust system to extract, clean, and analyze retail sales data
PySpark	Enables scalable processing — works locally or on clusters (Databricks, EMR)
Business Metrics	Daily revenue, top products, store performance, monthly trends
Data Quality	Schema validation, null handling, deduplication
Scalability	Modular design supports cloud migration (S3, Delta Lake, Airflow)
Production Ready	Logging, error handling, idempotency, audit trail

🚀 Future Enhancements (Scalability Roadmap)

1. **Cloud Integration**
   → Replace local paths with `s3a://` or `abfs://` for AWS/Azure
2. **Orchestration with Airflow**
   → Schedule daily runs using **`SparkSubmitOperator`**
3. **Delta Lake**
   → Enable ACID transactions, time travel, and upserts

4. **Monitoring & Alerts**

→ Integrate with Prometheus/Grafana or Datadog

ecks

ectations or Deequ to validate KPIs

→ Serve metrics via FastAPI for dashboard consumption

📚 Conclusion

This end-to-end ETL pipeline demonstrates how data engineers can turn raw, messy sales data into actionable business intelligence using PySpark.

It's designed to be:

- ✅ Modular — Easy to extend and maintain
- ✅ Scalable — Ready for cloud and big data
- ✅ Business-Aligned — Delivers real-world KPIs
- ✅ SEO-Optimized — For visibility and knowledge sharing

Whether you're building your first pipeline or scaling an enterprise data platform, this blueprint provides a solid foundation.

💬 *Feedback? Let me know how you'd extend this pipeline — I'd love to hear your ideas!*

Data Engineering | Pyspark | Spark | Python | Programming

Mayurkumar Surani

330 following

Data Scientist | Machine Learner | Digital Citizen

Follow

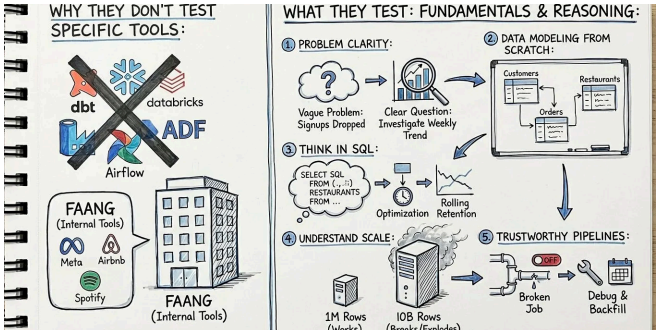
No responses yet

Luke C

What are your thoughts?

More from Mayurkumar Surani

Mayurkumar Surani



Mayurkumar Surani

Implementing Slowly Changing Dimension Type 2 (SCD2) with...

...sions are fundamental ...using that track...

Why FAANG Companies Don't Test You on dbt, Snowflake, or...

The Hard Truth About Tech Interviews

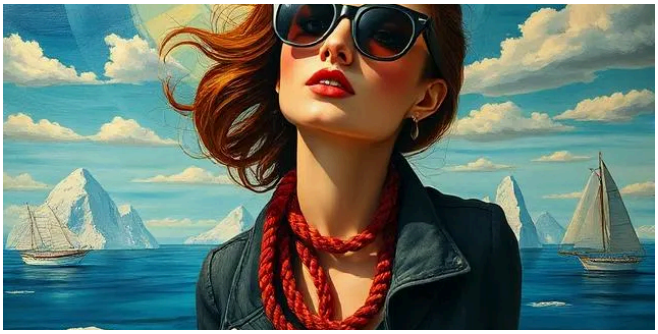
Dec 10, 2025 7

Mayurkumar Surani

Python for Data Engineering 2025: Complete Beginner's Guide with...

Master Python fundamentals for data engineering in 2025. Learn variables, data...

Nov 2, 2025 193 4



Mayurkumar Surani

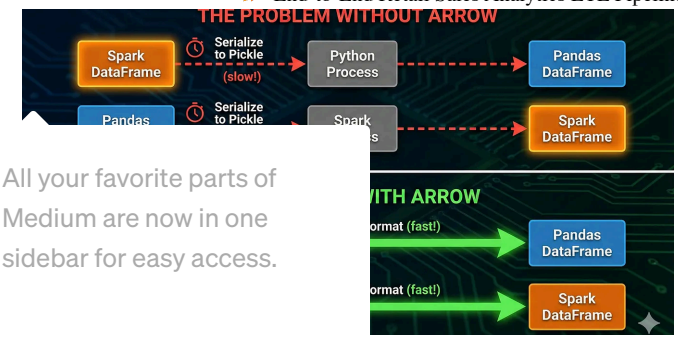
Ultimate Data Engineering Interview Guide: 150+ SQL &...

Master data engineering interviews with 150+ hands-on SQL/PySpark questions. Includes...

Dec 20, 2025 4

See all from Mayurkumar Surani

Recommended from Medium



Anchit Gupta

Level Up Your ETL Code with Arrow in PySpark

How Apache Arrow can supercharge your PySpark pipelines and save you hours of...

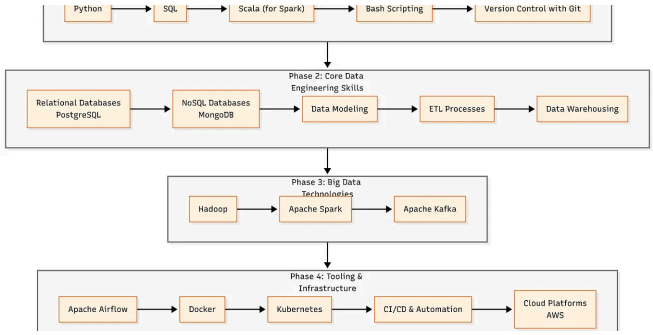
Dec 25, 2025 20

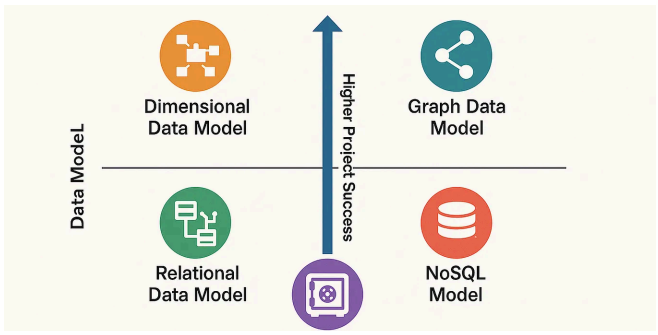
In Stackademic by Sai Kumar Devulapelli

PySpark Interview for Data Engineers: Patterns, Pitfalls, and...

Interviewers look for correctness at scale, performance awareness, and production...

Aug 26, 2025 3





Harish Anbalagan

Complete Data Engineer Roadmap with 3000+ Free Learning

A step-by-step Data Engineer roadmap with 3,000+ free resources and expert guidance

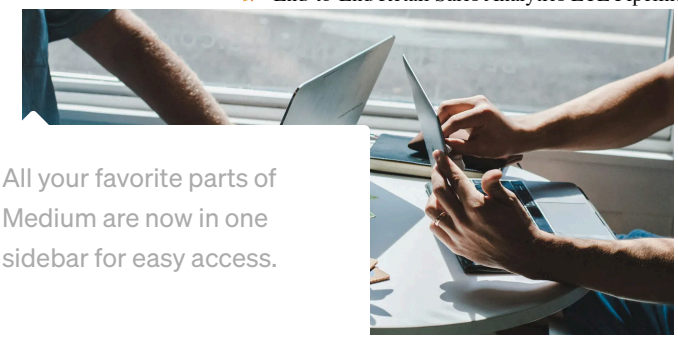
Oct 14, 2025 74

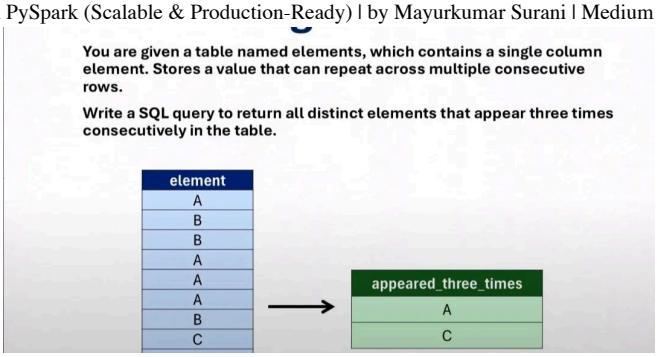
Rohan Dutt

10 Data Models Every Data Engineer Must Know (Before They...

Core modeling patterns that determine scalability, correctness, and long-term...

Dec 27, 2025 68 2





Jagadesh Jamjala

Mid-Level Data Engineer Interview Scenario

(SQL Server → Snowflake ETL Pipeline)

Dec 7, 2025 8 2

In Towards Dev by Mohit Daxini

American Express Interview question—Data Engineer II

This is a classic “consecutive rows” problem, very common in DE interviews (AmEx, Uber,...

Dec 18, 2025 86 1

See more recommendations