Solution Design Document

Project Two: GitHub Archive Data

Version: 1.00

Updated: 1/16/2025

Authors: Collin Gebauer, Josh Ripoli, Ta'Shawn Deshazier

IMPORTANT!

Note 1: *While a lot of effort has been put into this document to make it as generic as possible, you will inevitably find yourself adding additional sections or removing irrelevant ones, and that is perfectly fine.*

Note 2: *Make sure to read the article on documenting a solution design before attempting to use this document. Reading and understanding the article contents will allow you to achieve the intended purpose from documenting a good solution design.*

1. Overview or Introduction

A complete data engineering solution that processes GitHub's public event data through a medallion architecture (Bronze -> Silver -> Gold) using Apache Spark on Databricks. Sparrow Analytics needs fast, reliable aggregations, data will be clean and trustworthy for BI tools

2. Summary of Existing Functionality

- GitHub Events API - GH Archive collects public GitHub activity by consuming the GitHub Events API and stores that data in JSON format (https://www.gharchive.org/)
- Databricks Platform Providing a Distributed System (Spark) and Transformation tools
- Azure Platform Providing
    - Azure Data Lake Storage Gen 2 (ADLSv2)
    - Data Warehouse
    - BI Tools

3. Requirement Details

Data Ingestion Requirements (Source -> Bronze):

- Already Met: Data Source: GHArchive (https://www.gharchive.org/)

Medallion Architecture storage:

- Partitioned, and in parquet format for silver and gold layer

Aggregations:

- Data aggregated by type of GitHub event per hour
- PushEvent data aggregated by ref type – whether the commit is on the main branch
- Breakdown of events by type and number of commits per event
- User activity should be aggregated so that a filterable chart can be populated with breakdowns of user activity by week or month.
- Breakdown of activity by project – find a unique use case

Pipeline:

- Use Apache Spark to read in and flatten JSON data into a tabular format

    - Should be in a parquet format
- Data is partitioned in a time-based schema
    - Each partition is ~128 MB in size
- Gold layer will be in a snowflake schema for use of BI and other analytical tools

4. Assumptions and Prerequisites

- All the JSON files are available and accessible
- The pipeline should work for multiple year's worth of data
- Storage environment should be structured for the different layers of the medallion architecture
- data is partitioned and structured in parquet format for all steps of the medallion architecture
- Actor, repo, payload are valid strings for all rows.
- 'Created_at' is a parseable timestamp for each row and is in UTC (because of "Z" at end of timestamp)

5. High-Level Design

1) Data Source: GHArchive (https://www.gharchive.org/)
2) ADLSv2 Bronze (raw)
3) Bronze_to_Silver (transformations using PySpark, Databricks Notebook)
4) ADLSv2 Silver (partitioned, parquet)
5) Silver_to_Gold (transformations using PySpark, Databricks Notebook)
6) Gold Layer Denormalized into Snowflake Schema

Architecture Diagram

Silver Layer ERD

Gold Layer ERD

6. Low-Level Design

Silver Layer

1. System Overview

The pipeline implements a **Medallion Architecture** transition. It reads compressed JSON files, enforces a complex schema, extracts dimensional data (Actors, Orgs, Repos), and flattens 16 specific GitHub event types into separate tables.

2. Component Design

2.1 Configuration & Globals

The system uses a set of global parameters to control memory overhead and storage optimization:

- **Partition Mode:** spark.sql.sources.partitionOverwriteMode is set to dynamic to ensure idempotency.
- **File Sizing:** Targeted file size is **128MB** with a partition count of **96** to optimize downstream read performance.
- **Batching:** Processing is done in chunks (default Chunk_Size = 288) to prevent driver memory exhaustion during wide transformations.

2.2 Extraction Logic (Methods)

The transformation logic is modularized into specialized extraction functions:

Dimensional Extraction: extract_non_event_tables(df)

This is the core "Star Schema" generator.

- **Actor Table:** Extracts user identities from seven different JSON paths (Main Actor, Issue Assignee, PR Owner, etc.) and performs a prioritized deduplication using Window functions.
- **Repo Table:** Consolidates repository metadata from the base event and nested payload objects (forkee, base repo, head repo).
- **Org Table:** Extracts unique Organization IDs and logins.

Event-Specific Extraction

Each GitHub event type has a dedicated function (e.g., extract_pull_request_events, extract_push_events).

- **Input:** Raw DataFrame.
- **Logic:** Filters by type, selects specific payload fields, and casts types (e.g., id to LongType).
- **Output:** Flattened DataFrame ready for Parquet persistence.

3. Data Flow & Processing Steps

Step 1: File Discovery & Batching

The system scans the Bronze container (gharchive-raw), sorts files chronologically, and groups them by **Year-Month** (YYYY-MM).

Step 2: Incremental Processing Loop

For each month, the pipeline:

1. Reads a batch of JSON files using the raw_data_schema.
2. **Caches** the initial DataFrame to support multiple downstream extraction passes.
3. Executes the dimensional and event extraction methods.

Step 3: Persistence Strategy

- **Event Data:** Written using write_partition_by_month, partitioned by event_year_month.
- **Dimension Tables:** Written using write_table. To avoid "Small File Problem," these are coalesced to a single file per partition (coalesce(1)).
- **Write Mode:** Uses overwrite for the first batch of a month and append for subsequent chunks within the same month.

Step 4: Final Deduplication (Post-Processing)

Since actors and repos can appear in multiple events across different files, a final global deduplication is performed:

1. Read the written Parquet files.
2. Apply Window.partitionBy("id").orderBy(...) to keep the record with the most complete metadata (non-null types/admins).
3. Overwrite the Silver path with the clean, unique dataset.

4. Error Handling & Optimization

- **Memory Management:** Explicit use of df.unpersist() at the end of every batch loop to clear the Spark cache.
- **Schema Enforcement:** A strict StructType schema is applied during the read phase to handle the deeply nested and sometimes inconsistent GitHub API JSON structure.
- **Idempotency:** Dynamic partitioning ensures that re-running a specific batch does not duplicate data in other partitions.

5. Database Schema (Silver Layer)

The final output consists of a relational-style storage structure

Gold Layer

This Low-Level Design (LLD) detail the transformation of structured relational data from the **Silver Layer** into an optimized Snowflake Schema in the **Gold Layer**. This stage focuses on creating analytical datasets, fact tables, and dimensions for business intelligence.

1. System Design & Architectural Patterns

The Silver-to-Gold pipeline transitions from a normalized relational structure to a **Dimensional Model (Snowflake Schema)**.

- **Pattern:** Dimensional Modeling (Fact and Dimension tables).
- **Storage Format:** Parquet with specific partitioning strategies to optimize query performance.
- **Optimization:** Uses **Broadcast Joins** for small dimension tables (e.g., event_type_dim) to eliminate large-scale data shuffling.

2. Component Breakdown

2.1 Dimensional Table Generation

The pipeline creates several dimensions to provide context to the facts:

- **event_type_dim**: Extracts unique event types and assigns a surrogate key (type_id) using monotonically_increasing_id().
- **datetime_dim**: Generates a high-granularity time dimension from a sequence. It calculates attributes like is_weekend, quarter, and day_part (e.g., Morning, Afternoon) for deep temporal analysis.
- **user_dim**: Derived from the Silver actor table, mapping raw IDs to business-friendly terms like username.
- **repo_dim**: Finalized repository metadata for analytical consumption.

2.2 Fact Table Engineering

Fact tables are engineered by joining Silver tables with the newly created dimensions:

- **event_hourly_fact**: Aggregates event counts by datetime_id and event_type_id. This table enables rapid "events per hour" trend analysis.
- **weekly_repo_activity**: Provides a weekly snapshot of repository health, calculating event_count, commit_count (filtered on PushEvent), and unique_users_id_count.
- **user_activity_weekly_fact / monthly_fact**: These tables track user engagement over time. They utilize date truncation and integer-based week_id/month_id for efficient partitioning.
- **push_event_fact**: Specialized analysis of push behavior. It uses regex (refs/heads/(.*)) to categorize branches into types like main_or_master or develop_family.

3. Step-by-Step Data Flow

1. **Ingestion**: Read Parquet files from the Silver path using pre-defined schemas (event_df_schema, push_event_df_schema, etc.).
2. **Dimension Creation**: Generate static and semi-static dimensions (datetime, event_type).
3. **Broadcasting**: Identify small dimensions to be broadcast during joins to the large event_data fact table.
4. **Aggregation**: Execute groupBy operations at multiple grains:
    a. **Hourly**: For real-time dashboarding.

b. **Weekly/Monthly**: For long-term trend analysis.

5. **Classification**: Apply business logic (e.g., determining the day_part or branch_type) using Spark when/otherwise and regexp_extract functions.

4. Partitioning

Fact tables are partitioned by time-based keys (week_id, month_id, event_type_id) to support time-series queries.

5. Gold Layer Schema (Entity Relationship)

The final architecture consists of:

- **Primary Dimensions**: datetime_dim, user_dim, repo_dim, event_type_dim.
- **Primary Facts**: event_hourly_fact, user_activity_weekly_fact, weekly_repo_activity, push_event_fact.

7. Impact Analysis

No impact on existing systems

8. Out-of-scope

- Real-time streaming or near real-time updates
- Access to info for GitHub private repositories

9. Risks and Mitigation

Risk: GitHub event types contain different nested structures, and schemas may evolve over time.

Mitigation: Pipeline is designed to have modularity so that new functions to acquire new nested data may be added with minimal to no impact to existing code inside of the pipeline

Risk: Large data volume and performance bottlenecks
Mitigation:
1) Partition data by year/month / day / hour
2) Control file sizes (~128 MB)
3) Avoid wide transformations whenever possible
4) Tune Spark configurations