# Project Requirements

## PySpark Code Standards

1. Expert-Level PySpark Code for Databricks Notebook:

- The code must be clean, reusable, maintainable, efficient, and well-documented.

2. Clean Coding Guidelines:

- Follow the Palantir Style Guide:

[Palantir Style Guide](https://www.palantir.com/docs/foundry/transforms-python/pyspark-style-guide/)

- Adhere to PEP 8 — the Style Guide for Python Code:

[PEP 8](https://peps.python.org/pep-0008/)

### 3. Code Organization and Documentation:

- Ensure the code is well-organized, follows consistent naming conventions, and is thoroughly documented.

- Maintain coherent naming conventions across the project.

- Include type hints, docstrings, and examples in the docstrings for clarity and ease of integration.

- Provide meaningful usage examples for each function parameter within the docstrings.

### 4. Readability and Usability:

- Write code that is easy to understand and use across different PySpark projects.

- Implement input validation and error handling, with engaging and clear error messages (consider using emojis for emphasis).

## Efficiency and Speed

### 1. Performance Optimization:

- Ensure the code is efficient and fast, capable of handling large datasets (over 100 million rows and 60 columns with various data types).

- Optimize the code for scalability and performance.

- Utilize PySpark optimization techniques and the Catalyst optimizer.

### 2. DataFrame API Utilization:

- Use DataFrame API functions optimized by the Catalyst optimizer.

- Employ performance tuning and PySpark execution logic to maximize efficiency.

- Prefer DataFrame API functions over Python functions whenever possible.

- Apply early filtering when feasible.

### 3. Advanced Performance Techniques:

- Improve speed and execution time by:

- Using the Hash `\_CASE\_KEY` column.

- Partitioning by `\_CASE\_KEY` and `EVENTTIME` based on year/month.

- Bucketing data.

- Saving as Delta format.

- Loading, caching, and unpersisting DataFrames to enhance performance.

### 4. Avoidance of Inefficient Practices:

- Avoid unnecessary loops and User-Defined Functions (UDFs) by leveraging PySpark's internal functions.

- Minimize the use of UDFs as they can hinder optimization.

- Avoid using Python's internal functions in favor of PySpark's optimized functions.

## Modular Programming Principles

1. Fundamental Principles:

- Adhere to the Single Responsibility Principle (SRP).

- Maintain consistent naming conventions.

- Ensure modularity in code design.

- Follow the Open/Closed Principle.

- Apply regular refactoring.

- Provide comprehensive documentation and comments.

- Implement the Liskov Substitution Principle.

- Follow the Interface Segregation Principle.

- Adhere to the Dependency Inversion Principle.

- Ensure encapsulation and separation of concerns.

- Follow the DRY (Don't Repeat Yourself) principle.

- Keep the code simple and straightforward (KISS principle).

- Avoid over-engineering (YAGNI principle).

- Prefer composition over inheritance.

- Utilize design patterns where appropriate.

## Additional Essential Requirements

### 1. Error Handling and Logging:

- Implement robust error handling mechanisms to gracefully manage exceptions and errors.

- Integrate comprehensive logging to facilitate debugging and performance monitoring.

- Ensure logs provide meaningful information and are structured for easy analysis.

### 2. Testing and Validation:

- Develop unit tests to validate individual components and functions.

- Create integration tests to ensure all parts of the code work seamlessly together.

- Automate testing processes to ensure code quality and reliability.

### 3. Data Quality and Consistency:

- Implement data validation checks to ensure data quality and consistency.

- Develop procedures to handle missing or inconsistent data.

- Monitor data quality throughout the data processing pipeline.