# Project Requirements

### **Code Structure and Organization**

* **Guidelines:** Follow the standard Python PEP8 guidelines and the Palantir Style Guide: [Palantir Style Guide](https://www.palantir.com/docs/foundry/transforms-python/pyspark-style-guide/)
* **Write modular code:** Modularize your code by creating reusable functions. Break down large functions into smaller, reusable components. Follow the **Single Responsibility Principle**. Group related operations into logical blocks or modules. The code must be clean, reusable, maintainable, efficient, and well-documented. Write code that is easy to understand and easy to use across different PySpark projects.
* **Avoid Hardcoding:** Use configuration files or environment variables for settings.
* **Immutability:** Use immutable data structures where possible.
* **Functional Programming Principles:** Favor pure functions and avoid side effects.
* **Scalability:** Ensure code can handle large datasets efficiently.
* **Maintainability:** Write maintainable code with future developers in mind. Implement input validation and error handling, with engaging and clear error messages (consider using emojis for emphasis).

1. **Data Processing and Transformations**
   * Use **DataFrame API** over RDDs for better performance and optimization.
   * Chain DataFrame operations efficiently to avoid creating multiple intermediate DataFrames.
   * Use select() with column expressions rather than withColumn() for better performance.
   * Leverage built-in functions in pyspark.sql.functions for common operations.
   * Avoid using UDFs (User-Defined Functions) unless necessary.
   * Prefer native Spark functions.
   * Lazy Evaluation: Leverage PySpark's lazy evaluation to optimize the execution plan. Only trigger actions (count, collect, saveAsTextFile) when necessary.
   * Use built-in functions (select, filter, groupBy, join) which are optimized by Catalyst.
   * Partitioning: Ensure optimal partitioning of data to balance workload and reduce shuffling.
   * Use repartition and coalesce to manage partitions efficiently.
   * Broadcast Variables: Use broadcast variables for small datasets to avoid repeated data transfer across nodes.
   * Caching and Persistence: Cache frequently accessed data using cache or persist to speed up subsequent operations.
   * Avoid Wide Transformations: Minimize wide transformations (e.g., groupByKey, reduceByKey) which trigger shuffles.
   * Use narrow transformations (map, filter) wherever possible.
   * Data Serialization: Use efficient serialization formats like Kryo to reduce serialization overhead.
   * Predicate Pushdown: Take advantage of predicate pushdown in data sources (e.g., Parquet) to filter data at the storage level.
   * Avoid Using collect: Avoid using collect on large datasets as it brings all data to the driver, causing memory issues.
   * Use actions that operate on distributed data (e.g., count, foreach).
   * Vectorized UDFs: Use Pandas UDFs (vectorized UDFs) for better performance when applying custom functions to columns.
   * Join Optimizations: Use broadcast joins for small datasets to reduce shuffle.
   * Ensure join keys are well-partitioned to avoid skew.
   * Skew Handling: Handle data skew by salting keys or using custom partitioning strategies.
2. **Data Loading and Saving**
   1. Use schema definitions explicitly while loading data to improve performance and avoid issues with data types.
   2. Prefer Parquet or ORC formats over CSV or JSON for better performance and compression.
   3. Use coalesce() or repartition() appropriately to manage the number of output files.
3. **Performance Optimization**
   1. Use broadcast variables for small lookup tables to avoid shuffling large datasets.
   2. Cache or persist DataFrames that are reused multiple times within a workflow.

* 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.
* 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.
* 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.

1. **Error Handling and Logging**
   1. Implement robust error handling using try-except blocks and custom exception classes.
   2. Log key events and metrics using Spark's logging facilities or external logging libraries.
   3. Use meaningful log messages and appropriate log levels (INFO, DEBUG, ERROR).
2. **Testing and Validation**
   1. Write unit tests for your PySpark code using unittest or pytest frameworks.
   2. Use pyspark.sql.Row and SparkSession for creating test data and running tests.
   3. Validate your transformations and output data for correctness and consistency.
   4. Test and validate each function and consider edge cases before creating a function.
3. **Documentation and Readability**

* Document your code with meaningful docstrings and comments.
* Provide clear and concise descriptions of the purpose and functionality of your functions and classes.
* Use descriptive variable and function names to enhance readability.
* 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.