Problem Description

The project addresses the challenge of processing log files containing various types of log entries and generating aggregated JSON reports for each log category.

Core Functionalities:

1. Log Parsing:
    - Parse log files contain diverse log types (e.g., APM, Application, Request).
    - Extract essential information based on the specific structure of each log type.
2. Log Classification:
    - Categorize logs into distinct groups, such as APM, Application, and Request logs.
3. Log Aggregation:
    - APM Logs: Calculate key metrics (e.g., minimum, maximum, median, and average) for attributes like CPU and memory usage.
    - Application Logs: Count log entries based on severity levels (e.g., INFO, DEBUG, ERROR).
    - Request Logs: Compute response time statistics and group HTTP status codes by API endpoints.
4. Output Generation:
    - Generate and save JSON reports for each log type into separate files (e.g., apm.json, application.json, and request.json).

Future Extensibility:

1. Support for New Log Types:
    - The architecture is designed to allow seamless integration of additional log types, such as Security Logs, without significant modifications to the existing system.
2. Support for Additional File Formats:
    - Extendable processing other file formats (e.g., CSV) for input and output, ensuring the application's versatility across different data sources and requirements.

Design Patterns

1.Factory Method Pattern

- What it does: Decouples the creation of objects (parsers, aggregators, file processors) from their usage.
- How it's used:
    1. LogParserFactory:
        - Creates the appropriate parser for each log type (e.g., AppBaseLogFileParser, RequestLogParser) based on the log structure.
    2. LogAggregatorFactory:
        - Creates the appropriate aggregator (e.g., MetricsDataAggregator, RequestDataAggregator) for each log type.
    3. FileProcessorFactory:
        - Selects the appropriate file processor implementation (e.g., TextFileProcessor, JsonFileProcessor) based on the file extension.
        - Ensures future extensibility by allowing additional file formats (e.g., CSV) without modifying the FileHandler.

2.Singleton Pattern

- What it does: Ensures a class has only one instance and provides a global point of access to it.
- How it's used:
    - FileIOHandler:
        - Ensures that only one instance of FileIOHandler exists in the application.
        - Implements a thread-safe, lazy initialization mechanism using double-checked locking and the volatile keyword.
        - Centralizes access to file operations (readFile, writeFile, and ensureOutputFiles), preventing multiple instances from being created and used inconsistently.

3.Chain of Responsibility

- What it does: Passes a request along a chain of handlers, where each handler can either handle the request or pass it to the next one.
- How it's used:
    - LogValidator:
        - Sequentially validates log entries by checking if required fields are present in the attributes map.
        - Each validation step (e.g., checking for required keys, verifying field correctness) acts as a "handler" in the chain.
        - Future validations (e.g., value type checks, range checks) can be added without modifying existing logic, making it extensible.
    - Purpose:
        - Centralizes and organizes validation logic.
        - Allows logs with invalid fields to be rejected systematically without duplicating validation logic elsewhere.

Consequences of Using Design Patterns

1. Factory Method Pattern

Advantages:

- Decouples Object Creation: Separates logic for creating objects (e.g., parsers, aggregators, file processors) from their usage.
- Extensibility: New log types or file formats can be added by creating new factory methods or classes without altering existing code.
- Centralized Logic: Reduces duplication by centralizing object creation, simplifying maintenance.
- Code Reusability: Promotes reuse of creation logic across the system.

Trade-offs:

- Increased Complexity: Requires more classes and methods, adding layers to the codebase.

- Dependency on Factories: Poorly implemented factories may become bottlenecks or single points of failure.

2. Singleton Pattern

Advantages:

- Ensures Single Instance: Prevents conflicts by allowing only one instance of FileIOHandler across the application.
- Global Access Point: Provides a consistent and centralized interface for managing file operations.
- Thread Safety: Safeguards correctness in multi-threaded environments with thread-safe implementations.

Trade-offs:

- Global State Dependency: Creates hidden dependencies as all components rely on the same global instance.
- Testing Challenges: Mocking or replacing the singleton during unit testing can be difficult without abstraction.
- Tight Coupling: Reduces flexibility as components become tightly coupled to the singleton instance.

3. Chain of Responsibility

Advantages:

- Separation of Concerns: Breaks validation into discrete steps, making the logic easier to maintain and extend.
- Extensibility: New validation rules can be added without modifying existing code.
- Reusability: Validation handlers can be reused across different contexts.
- Dynamic Behavior: Allows adding, removing, or reordering handlers at runtime.

Trade-offs:

- Potential Overhead: Passing a request through multiple handlers may introduce performance costs for long chains.
- Debugging Complexity: Tracing requests through the chain can be challenging, especially with conditional logic.
- Dependency on Chain Order: Improperly ordered handlers may produce incorrect outcomes, requiring careful design.

Class Diagram: