# Message Processor Application - Comprehensive Documentation

**Version:** 1.1.0 (Post-RestTemplate Migration) **Date:** 2025-04-28

## 1. Introduction

This document provides comprehensive information about the Message Processor application, covering its architecture, design, detailed documentation of components, and instructions for usage (configuration, building, running). The application serves as a high-throughput ingestion system designed to receive JSON messages from six distinct pipelines, aggregate them based on configurable rules, and dispatch the merged payloads to downstream REST endpoints.

This version of the documentation reflects the state of the application after migrating the HTTP dispatch mechanism from Spring WebClient (non-blocking) to Spring RestTemplate with HttpEntity (blocking).

## 2. Architecture Overview

The Message Processor is designed as a monolithic Spring Boot application responsible for the following core functions:

1. **Ingestion**: Listens for incoming JSON messages on dedicated HTTP ports for six different pipelines.
2. **Parsing & Validation**: Parses incoming JSON payloads and performs basic validation (e.g., checking for the presence of expected arrays).
3. **Aggregation/Batching**: Groups messages based on pipeline-specific rules:
   * **Per-ID Bucketing**: For LoadPipeline and MultiDestinationRake (MDR), messages are grouped by a specific ID (e.g., LoadID) within a configurable time window.
   * **Global Batching**: For LoadAttribute, MaintenanceBlock, MaintenanceBlockResource, and TrainServiceUpdateActual, messages are collected into a single global batch.
4. **Flushing**: Batches are flushed (merged and prepared for dispatch) based on either a time window expiring or a batch size threshold being reached.
5. **Dispatch**: Sends the aggregated JSON payload to a configured downstream REST endpoint via an HTTP POST request using RestTemplate.
6. **Throttling**: Limits the rate of outgoing HTTP requests per second for each pipeline based on configuration.
7. **Archiving**: Optionally archives both incoming raw messages and the final merged payloads to the filesystem in a structured directory format (timestamped).
8. **Reporting**: Generates daily CSV reports summarizing the dispatch status (SENT, FAILED, SKIPPED) for each batch.
9. **Dead Letter Queue (DLQ)**: Archives payloads that failed to be dispatched successfully.

### System Components

* **Spring Boot Application**: The core framework providing dependency injection, web server, scheduling, and configuration management.
* **Tomcat**: Embedded web server hosting the ingestion endpoints.
* **REST Controllers**: Six controllers, each handling POST requests for a specific pipeline on a dedicated port.
* **Aggregator Service**: The central service containing the business logic for aggregation, flushing, dispatch, throttling, archiving, and reporting.
* **RestTemplate**: Spring’s synchronous HTTP client used for dispatching aggregated messages.
* **Jackson**: Library used for JSON parsing and manipulation.
* **Maven**: Build tool for managing dependencies and packaging the application.
* **External Configuration**: A .properties file (external-config.properties) driving the application’s behavior.

### Data Flow

1. External systems send JSON payloads via HTTP POST to the specific port/endpoint for a given pipeline.
2. The corresponding RestController receives the request.
3. The controller passes the raw JSON string to the AggregatorService.
4. AggregatorService optionally archives the incoming message asynchronously.
5. AggregatorService parses the JSON and adds the relevant data (usually a JSON array) to the appropriate bucket (per-ID MessageBucket or global BatchBucket).
6. If adding the message causes a size threshold to be met, the bucket is flushed immediately.
7. A scheduled task (@Scheduled) runs periodically (e.g., every second) to check for buckets that have exceeded their time window and flushes them.
8. Flushing involves:
   * Merging all messages in the bucket into a single JSON payload.
   * Optionally archiving the merged payload asynchronously.
   * Applying throttling logic (pausing if the rate limit is exceeded).
   * Creating an HttpEntity containing the payload and Content-Type: application/json header.
   * Sending the HttpEntity via RestTemplate.postForEntity to the configured target URL.
   * Handling the response: Logging success/failure to a CSV report and archiving to DLQ on failure.
   * Removing the flushed bucket (for per-ID) or replacing it with a new empty one (for global batch).

## 3. Detailed Design and Component Documentation

### 3.1. AggregationServiceApplication.java

* **Purpose**: Main entry point and configuration class for the Spring Boot application.
* **Annotations**: @SpringBootApplication, @EnableScheduling.
* **Key Beans**:
  + ServletWebServerFactory: Configures the embedded Tomcat server. It sets the primary port (for LoadPipeline) and adds five additional Connector instances, each listening on a different port specified in external-config.properties. This allows each pipeline controller to have its own dedicated port.
  + RestTemplate: Provides a shared, injectable RestTemplate instance built using RestTemplateBuilder. This client is used by AggregatorService for dispatching messages.
* **Configuration**: Uses @Value annotations to inject port numbers from external-config.properties.

### 3.2. AggregatorService.java

* **Purpose**: Core service containing the main business logic.
* **Dependencies**: Injects RestTemplate.
* **Key Fields**:
  + mapper: ObjectMapper instance for JSON processing.
  + restTemplate: Shared client for HTTP dispatch.
  + ioPool: ExecutorService (fixed thread pool) for handling asynchronous I/O tasks (archiving, CSV writing).
  + lpBuckets, mdrBuckets: ConcurrentHashMap<String, MessageBucket> for per-ID aggregation.
  + laBatch, mbBatch, mbrBatch, tsuBatch: AtomicReference<BatchBucket> for global batch aggregation.
  + @Value annotated fields: Numerous fields injecting configuration parameters (time windows, sizes, URLs, flags, paths, limits) for each of the six pipelines from external-config.properties.
  + Counters (lpCounter, mdrCounter, etc.) and Window Start Timestamps: Used for throttling logic.
* **Key Methods**:
  + process<PipelineName>(String json, String source): (e.g., processLoadPipeline) Entry point for ingesting a message for a specific pipeline. Handles optional incoming archiving, JSON parsing, adding data to the appropriate bucket, and triggering size-based flushes.
  + flush<PipelineName>Buckets() / flush<PipelineName>ByTime(): @Scheduled methods that periodically check and trigger time-based flushes for the respective pipelines.
  + flush<PipelineName>(String id, MessageBucket bucket) / flush<PipelineName>Batch(BatchBucket batch): Private methods responsible for handling the flush logic for a specific bucket (merging, archiving, dispatching via sendWithThrottleAndReport).
  + sendWithThrottleAndReport(...): Central method for dispatching. Applies throttling, creates HttpHeaders and HttpEntity<String>, calls restTemplate.postForEntity, handles success/error (writes CSV, archives to DLQ on error). **Note:** Uses a synchronous RestTemplate call.
  + throttle(...): Implements the rate-limiting logic (currently uses Thread.sleep(1) when limit is exceeded).
  + archive(String json, String root): Asynchronously writes the given JSON string to a timestamped file structure under the specified root directory.
  + writeCsv(...): Asynchronously appends a status record to a daily CSV file.
  + shutdown(): @PreDestroy method to gracefully shut down the ioPool.
* **Inner Classes**:
  + MessageBucket: Holds messages (ConcurrentLinkedQueue<ArrayNode>) and a start timestamp for per-ID aggregation.
  + BatchBucket: Holds messages (ConcurrentLinkedQueue<ArrayNode>) and a start timestamp for global batch aggregation.

### 3.3. Controller Classes (\*Controller.java)

* **Purpose**: Expose HTTP POST endpoints for message ingestion.
* **Structure**: Each controller is annotated with @RestController and @RequestMapping (e.g., @RequestMapping("/receive-loadpipeline")).
* **Dependencies**: Injects AggregatorService.
* **Method**: Contains a single @PostMapping method (receive(@RequestBody String json)).
* **Functionality**: Accepts the raw JSON payload from the request body and delegates processing to the corresponding process<PipelineName> method in AggregatorService. Returns a simple ResponseEntity.ok(...) acknowledgement.

### 3.4. pom.xml

* **Purpose**: Maven project configuration.
* **Parent**: spring-boot-starter-parent (provides dependency and plugin management).
* **Key Dependencies**:
  + spring-boot-starter-web: Includes Spring MVC, embedded Tomcat, Jackson, and core Spring features.
  + spring-boot-starter-test: For testing.
  + spring-boot-starter-webflux: **Commented out** after migration to RestTemplate.
* **Properties**: Sets java.version to 11 and spring.config.name to external-config (instructs Spring Boot to look for external-config.properties instead of application.properties).
* **Plugins**:
  + spring-boot-maven-plugin: Packages the application as an executable JAR.
  + maven-compiler-plugin: Configures Java source/target versions.

### 3.5. external-config.properties (Conceptual)

* **Purpose**: Externalizes all configurable parameters.
* **Format**: Standard Java properties file (key=value pairs).
* **Key Sections (Example Properties)**:
  + **Server Ports**: loadpipeline.server.port=8081, mdr.server.port=8082, …
  + **Aggregation Time Windows (seconds)**: consolidation.loadpipeline.timeframe=60, consolidation.loadattribute.timeframe=30, …
  + **Aggregation Batch Sizes**: bucket.loadpipeline.flush.size=100, bucket.loadattribute.flush.size=50, …
  + **Archiving Enable/Disable**: archiving.loadpipeline.enabled=true, archiving.mdr.enabled=false, …
  + **Archive Paths**: archive.loadpipeline.incoming.root=/path/to/archive/lp/incoming, archive.loadpipeline.merged.root=/path/to/archive/lp/merged, …
  + **Target URLs**: target.loadpipeline.rest.url=http://downstream-service/api/loadpipeline, target.mdr.rest.url=http://downstream-service/api/mdr, …
  + **Target Enable/Disable**: target.loadpipeline.rest.enabled=true, target.mdr.rest.enabled=false, …
  + **Throttling Enable/Disable**: throttling.loadpipeline.enabled=true, …
  + **Throttling Limits (requests/sec)**: throttling.loadpipeline.limit=10, …
  + **DLQ Paths**: deadletterqueue.loadpipeline=/path/to/dlq/lp, …
  + **Report Prefixes**: report.loadpipeline.prefix=/path/to/reports/report\_lp\_, …

## 4. User Usage Guide

This section details how to configure, build, and run the application.

### 4.1. Prerequisites

* **Java Development Kit (JDK)**: Version 11 or later (java -version).
* **Apache Maven**: Version 3.6 or later (mvn -version).

### 4.2. Configuration (external-config.properties)

1. **Create the File**: Create a file named external-config.properties.
2. **Populate Properties**: Add and configure the necessary properties as described in Section 3.5. Pay close attention to:
   * **Ports**: Ensure the chosen ports are available.
   * **Paths**: Ensure all specified directory paths (archives, DLQ, reports) exist and the application has write permissions.
   * **URLs**: Set the correct downstream service endpoints.
   * **Enable/Disable Flags**: Configure which features (archiving, dispatch, throttling) are active for each pipeline.
3. **Placement**: Place this file in the directory from which you intend to run the application JAR.

### 4.3. Building the Application

1. Navigate to the project’s root directory (containing pom.xml).
2. Execute the Maven command: bash mvn clean package
3. This will create an executable JAR file (e.g., message-processor-1.0.0.jar) in the target/ subdirectory.

### 4.4. Running the Application

1. Copy the generated JAR file (e.g., target/message-processor-1.0.0.jar) to your deployment directory.
2. Ensure the configured external-config.properties file is in the **same directory**.
3. Start the application: bash java -jar message-processor-1.0.0.jar *(Adjust the JAR filename if necessary)*
4. The application will start, logging output to the console, and begin listening on the configured ports.

**Alternative Configuration Location:** If external-config.properties is elsewhere, specify its path during startup:

java -jar message-processor-1.0.0.jar --spring.config.location=file:/path/to/your/external-config.properties

### 4.5. Stopping the Application

* **Foreground**: Press Ctrl+C in the terminal.
* **Background**: Use standard OS commands (e.g., kill <pid>, systemctl stop <service-name>).

## 5. Technical Analysis and Recommendations

(This section incorporates the analysis previously generated)

### 5.1. Strengths

* **Clear Pipeline Separation**: Distinct handling and configuration per pipeline.
* **Configuration-Driven**: High flexibility via external-config.properties.
* **Asynchronous I/O**: Non-blocking archiving and reporting using ExecutorService.
* **Concurrency Handling**: Appropriate use of ConcurrentHashMap and AtomicReference.
* **Resource Management**: Graceful shutdown of the I/O pool via @PreDestroy.

### 5.2. Areas for Improvement

1. **Error Handling (sendWithThrottleAndReport)**: Catches generic Exception. **Recommendation**: Catch specific RestClientException subclasses for better diagnostics and handling (e.g., retries vs. immediate DLQ).
2. **Throttling Implementation**: Uses inefficient and imprecise Thread.sleep(1). **Recommendation**: Use a dedicated library (Guava RateLimiter, Resilience4j RateLimiter) and implement proper window tracking.
3. **I/O Error Handling (archive, writeCsv)**: Uses e.printStackTrace(). **Recommendation**: Use a proper logging framework (SLF4j) and consider retries for transient I/O errors.
4. **RestTemplate Configuration**: Uses default timeouts. **Recommendation**: Configure explicit connection and read timeouts via RestTemplateBuilder (potentially configurable).
5. **Magic Strings**: Uses hardcoded status strings (“SENT”, “FAILED”). **Recommendation**: Define as constants.
6. **Lack of Metrics**: Limited operational visibility. **Recommendation**: Integrate Spring Boot Actuator and Micrometer for exposing metrics (counts, latency, errors).
7. **Synchronous HTTP Calls**: RestTemplate is blocking. **Recommendation**: If non-blocking is critical, explore AsyncRestTemplate or dedicate a thread pool for HTTP calls.
8. **Lack of Circuit Breaking**: Vulnerable to downstream failures. **Recommendation**: Implement circuit breaking (Resilience4j, Spring Cloud Circuit Breaker).

### 5.3. Migration Analysis (WebClient -> RestTemplate)

* **Advantages**: Simpler code, fewer dependencies (WebFlux removed), explicit try-catch error handling.
* **Drawbacks**: Blocking nature impacts scalability, higher thread consumption per request, loss of reactive features (e.g., backpressure).

## 6. Future Architecture Recommendations

(This section incorporates the recommendations previously generated)

1. **Message Queue Integration**: Replace direct HTTP dispatch with sending to a queue (Kafka, RabbitMQ) for resilience and decoupling.
2. **Microservice Decomposition**: Split pipelines into separate microservices if scaling or deployment needs differ.
3. **Containerization (Docker)**: Package for easier deployment and environment consistency.
4. **Orchestration (Kubernetes)**: Deploy on Kubernetes for scaling, resilience, and resource management.
5. **API Gateway**: Introduce for centralized concerns like authentication, rate limiting, and routing.
6. **Monitoring & Alerting**: Implement comprehensive monitoring (Prometheus, Grafana) and alerting.

## 7. Conclusion

The Message Processor application provides a configurable framework for ingesting and aggregating messages. The migration to RestTemplate simplifies the dispatch logic but introduces blocking behavior. By addressing the improvement areas outlined (especially error handling, throttling, and timeout configuration) and considering the architectural recommendations, the application can be made more robust, scalable, and operationally transparent.