Technical Report: Stateless Proxy Upload Gateway

1. Executive Summary

Objective: Enable high-volume file ingestion with zero client-side orchestration, ensuring robustness, scalability, and data consistency.

Solution: This project implements a **Stateless Proxy Upload Gateway** using a **Time-Partitioned State Machine**. The system accepts raw file uploads, automatically generates metadata, and orchestrates downstream processing asynchronously using **MinIO** (Object Storage) and **Kafka** (Messaging).

Key Outcomes:

- **Scalability**: Stateless ingestion tier allows horizontal scaling to handle increasing load.
- **Reliability**: Decoupled ingestion and processing layers prevent backpressure from affecting uploads.
- **Simplicity**: Clients only need to upload files; the server handles batching, metadata generation, and recovery.
- **Maintainability**: No SQL database required for state management; "State" is derived from the file system structure (S3/MinIO) and time.

2. Problem Statement

Traditional file ingestion systems often suffer from:

1. **Client Complexity**: Clients must manage batch IDs, generate metadata, and handle partial failure retries.
2. **State Management Overhead**: maintaining a database to track file status adds latency and operational complexity.
3. **Tight Coupling**: Synchronous processing of uploads blocks clients and creates bottlenecks.

3. Solution Architecture

3.1 Design Pattern: Time-Partitioned State Machine

Instead of a database row tracking every file, we use the storage system itself as the source of truth.

- **State = Directory Location**: A file's location determines its state.
  - `tmp-bucket/data/{batchId}/`: **Ingesting**
  - `tmp-bucket/ready-to-process/{yyyy}/{MM}/{dd}/{HH}/{batchId}`: **Ready for Processing**
  - `prod-bucket/data/{batchId}/`: **Processed & Live**
- **Time Partitioning**: "Ready" markers are organized by hour. This allows the processor to scan small, distinct windows of time rather than the entire bucket, ensuring operations remain O(1) relative to total dataset size.

3.2 System Components

1. **Ingestion Service (Spring Boot)**:
   - Exposes `POST /api/batches/upload`.
   - Generates a unique **Batch ID (UUID)**.

   - Uploads files to MinIO `tmp-bucket`.
   - **Auto-Generates Metadata**: Creates companion JSON files for routing.
   - Writes an "implicit completion marker" to the `ready-to-process` path.

2. **Storage Layer (MinIO)**:

   - **Temporary Bucket**: Holds raw uploads. configured with a **7-Day TTL** Lifecycle Policy to auto-expire old data.
   - **Production Bucket**: Long-term storage for processed files.

3. **Batch Processor (Background Worker)**:

   - **Schedule**: Runs every 5 minutes (configurable).
   - **Logic**: Scans the *current* and *previous* hour's `ready-to-process` prefixes.
   - **Action**: Atomically promotes files to `prod-bucket` and publishes metadata events to **Kafka**.

4. **Messaging Layer (Kafka)**:

   - **Topic Alpha**: Receives `*-meta1.json` events.
   - **Topic Beta**: Receives `*-meta2.json` events.
   - Decouples file movement from downstream business logic (e.g., ETL jobs, notifications).

---

4. Technical Implementation Details

4.1 Ingestion Flow

The client performs a single `multipart/form-data` POST.

```
// IngestionService.java
String batchId = UUID.randomUUID().toString();
// 1. Upload Original File
minioClient.putObject(... objectName ...);
// 2. Generate Metadata
minioClient.putObject(... "meta1.json" ...);
minioClient.putObject(... "meta2.json" ...);
// 3. Mark Ready
minioClient.putObject(... "ready-to-process/2026/01/21/10/" + batchId ...);
```

4.2 Background Processing

The `BatchProcessor` ensures *at-least-once* delivery. It is idempotent:

```java
// BatchProcessor.java
public void processTimeWindow(ZonedDateTime time) {
    String prefix = "ready-to-process/" +
time.format(DateTimeFormatter.ofPattern("yyyy/MM/dd/HH"));
    // List all batches in this hour
    for (Result<Item> batch : minioClient.listObjects(prefix)) {
        // Check if already processed (check prod-bucket)
        if (isProcessed(batchId)) continue;

        // Copy to Prod & Send to Kafka
        processBatch(batchId);
    }
}
```

4.3 Support & Recovery

A manual API exists to re-trigger processing for any time range, useful for incident recovery or bug fixes.

- **Endpoint**: `POST /api/batches/reprocess?start=...&end=...`
- **Mechanism**: Re-uses the exact same window scanning logic as the background job.

---

5. Deployment & Configuration

The solution is fully containerized using Docker Compose.

- **MinIO**: `minio/minio:RELEASE.2023-01-31T02-24-40Z`
- **Kafka**: `confluentinc/cp-kafka:7.3.0`
- **Application**: Java 17 / Spring Boot 3.0.0

Network Resilience: Kafka is configured with `KAFKA_ADVERTISED_LISTENERS` supporting distinct internal (Docker) and external (Host) access, facilitating easy local debugging and monitoring.

6. Conclusion

This architecture delivers a robust, high-throughput ingestion system. By removing state management from the application layer and leveraging the distinct capabilities of Object Storage and Kafka, we achieved a solution that is easy to scale, simple to operate, and resilient to failure.