Adaptive, Resource-Aware Spring Batch Processing: Technical Introduction and Documentation

Purpose and Overview

This implementation is a fully adaptive, resource-aware batch processing system built with Spring Batch.
Its purpose is to process multiple files in priority order—using parallelism and chunked handling—while monitoring real-time CPU and memory usage. The system then automatically adapts its **thread pool size** and **chunk size** at runtime. This guarantees efficient resource utilization, balanced throughput, and robust error handling, closely matching the advanced behaviors you had in your original custom Java implementation but now built with maintainable Spring idioms.

What We're Trying to Do

- **Process multiple input files with different priorities:** Files are sorted and handled by priority (e.g., a file with priority_1 is processed before priority_2).
- **Chunk-oriented, transactional processing:** Each file is split into records (lines), grouped into *chunks* (batches of records), and processed in transactions for reliability and performance.
- **Parallelize work:** Multiple files are handled in parallel steps, using a thread pool so parallelism fits available CPU resources.
- **Live system monitoring:** A background monitoring component checks CPU and memory usage every few seconds.
- **Dynamic adaptivity:** If CPU/memory is abundant, the system increases parallel threads and chunk sizes for speed. Under high load, it reduces both for system stability.
- **Automatic retry and backoff:** On processing errors (e.g., IO hiccups), the system retries chunk processing, using exponential backoff to spread out repeated failures.
- **Ultra-detailed documentation:** Every class, method, and significant line is explained for onboarding, handover, and technical training.

How It Works: Component-by-Component Explanation

1. **Job Launch**
    - Processing begins either automatically on app launch or via JobLauncher.
2. **Partitioning Files by Priority**
    - `PriorityFilePartitioner` scans the input directory, extracts priority from file names, and produces a list of partitions—each one referencing a specific file.
3. **Adaptive Task Executor (Thread Pool)**
    - All work is distributed to a special thread pool (`AdaptiveThreadPoolTaskExecutor`), whose size can be grown or shrunk in real time.
4. **Worker Steps (One per Partition/File)**
    - Each worker processes its assigned file in chunks. Within each chunk:
        - **ItemReader:** Reads a batch of records from the current file.
        - **ItemProcessor:** Applies business logic to each record.
        - **ItemWriter:** Writes the processed records to the target (e.g., database, console).
        - **Checkpointing:** After every chunk, the system commits the transaction and checks for new chunk size from the latest resource assessment.
5. **Resource Monitor**
    - Background thread (`ResourceMonitor`) periodically polls JVM and OS for CPU and memory usage. It then:
        - Increases threads/chunk when ample resources are available.
        - Decreases both when the system is under strain.
        - These changes take effect immediately for new threads and at the next chunk boundary for chunk size.
6. **Retry/Backoff Handler**
    - If a chunk fails (exception during read, process, or write), it's retried up to a fixed number of times with exponential backoff delays.
7. **Job Completion Listener**
    - Logs overall job status when processing completes or fails.

High-Level Process Flow

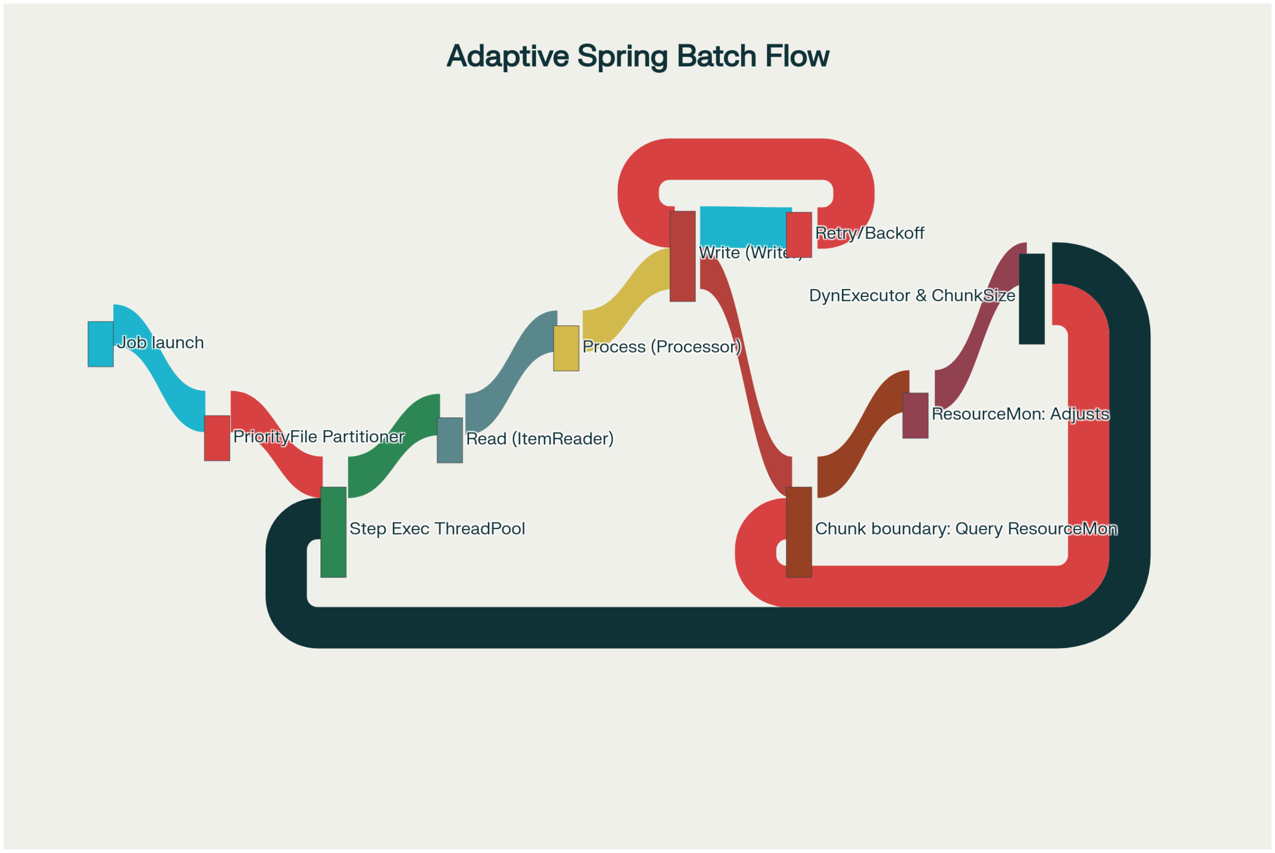
Text Walkthrough

1. **Start**
   The job is launched.
2. **Partition Phase**
   The input directory is scanned. Each file is extracted and assigned a partition, ordered so high priority files are handled first.

3. **Resource Monitor Starts**
   A monitor thread begins checking system CPU and memory every 5 seconds.
4. **Parallel Step Execution**
   Each partition (file) is assigned to a worker thread in the adaptive executor. Multiple partitions can process in parallel.
5. **Chunked Processing**
   For each file (partition):
   - Data is read record-by-record until the current chunk size is reached.
   - Each record runs through the processor logic (validation, transformation, etc.).
   - The entire chunk is then written in bulk to output.
   - At each chunk boundary, the chunk size may be updated (increased or decreased) based on recent system resource checks.
6. **Error Handling with Retry**
   Any chunk failure is retried up to 5 times, with increasing delays using exponential backoff.
7. **Live Adaptivity**
   As the ResourceMonitor detects low or high load, it instructs the thread pool to scale up or down immediately and signals the next chunk executions to use updated chunk sizes.
8. **Completion**
   Processing finishes once all files are fully handled. The listener logs the final status.

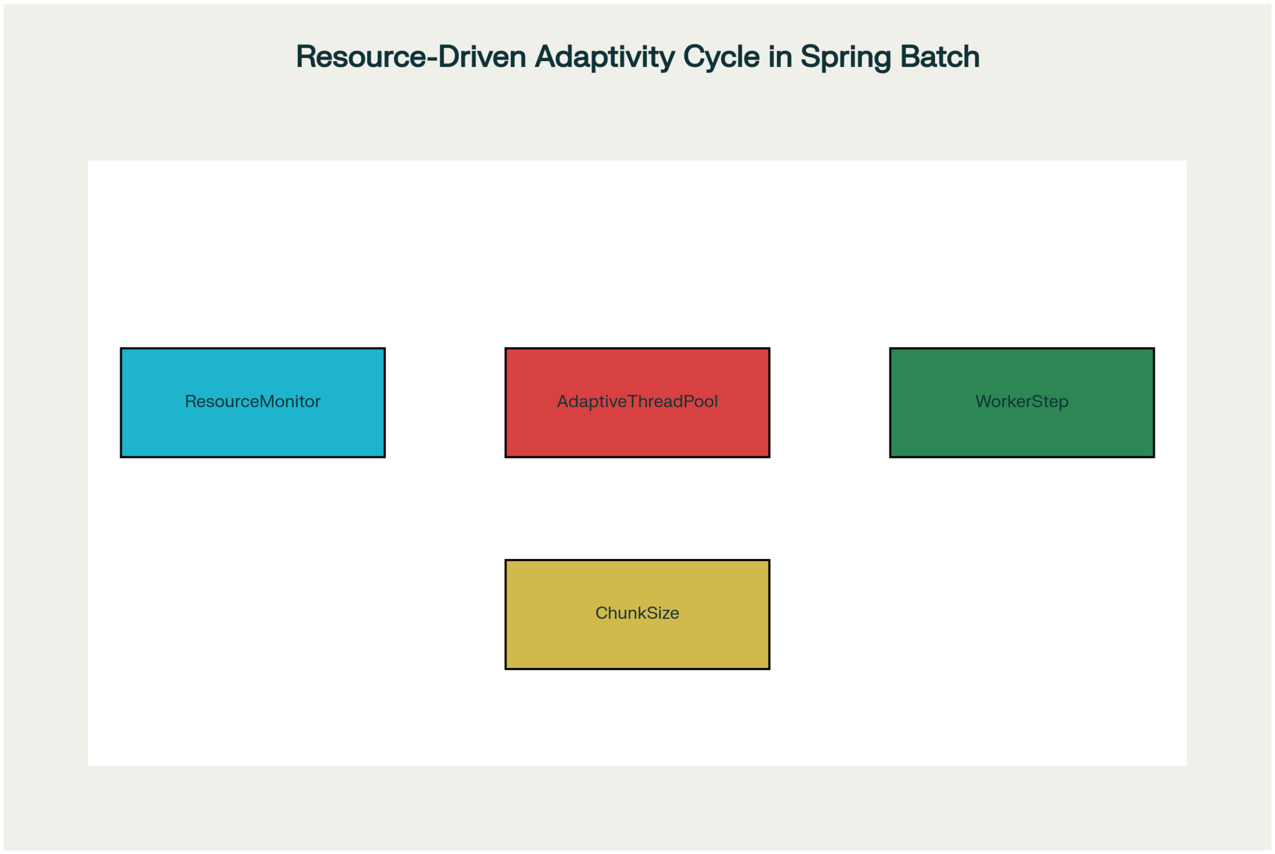
Core Flow Diagram

1. Adaptive Resource-aware Spring Batch Architecture

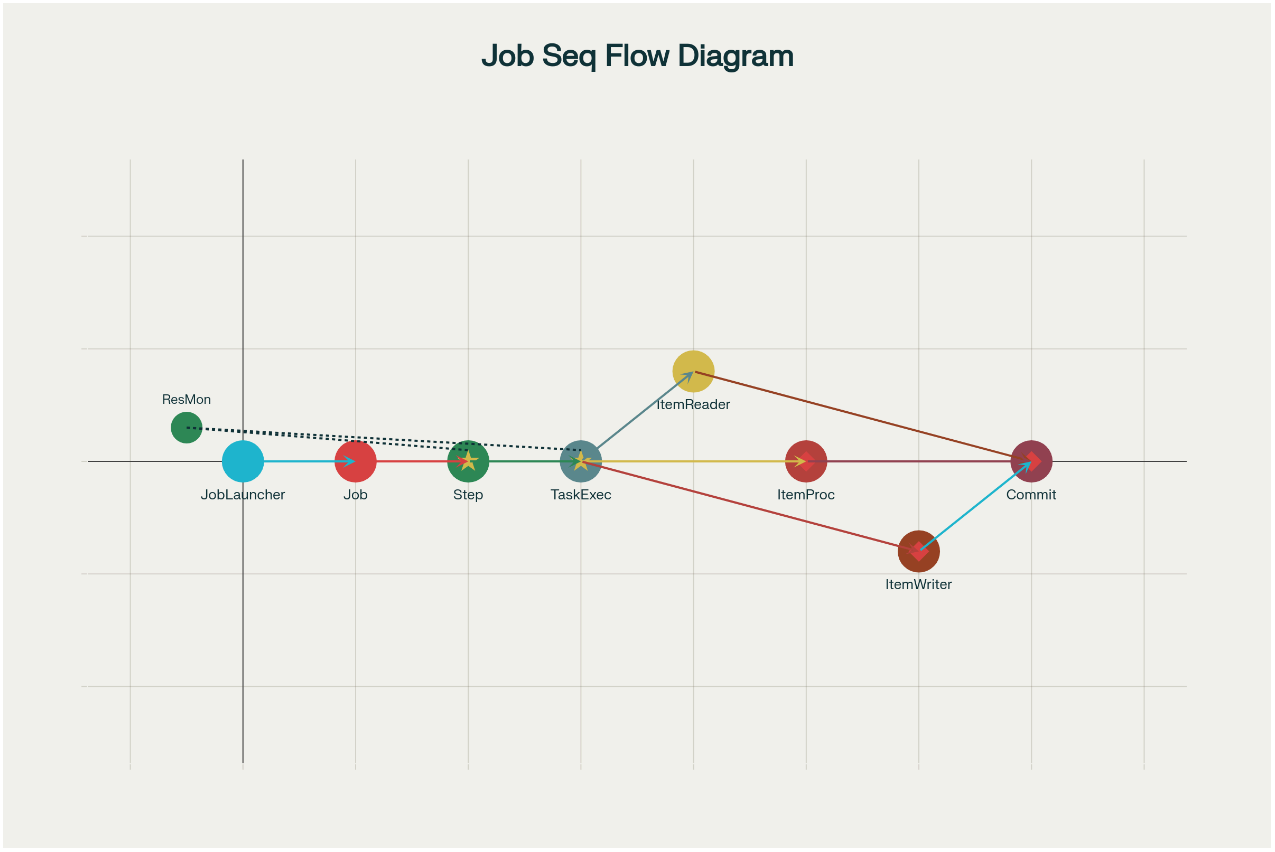


Adaptive Resource-aware Spring Batch Architecture Flow Diagram

2. Resource-Driven Adaptivity Cycle in Spring Batch



3. Spring Batch Execution Sequence with Adaptive Resource-Awareness



Spring Batch Execution Sequence with Adaptive Resource-Awareness

Summary Table: Key Components and Adaptivity

Component	Role	Resource Adaptation
PriorityFilePartitioner	Selects and sorts files for input partitions	No
AdaptiveThreadPoolTaskExecutor	Runs partitions in parallel; size adjusted at runtime	Yes (CPU/memory)
ResourceMonitor	Monitors JVM and OS, adjusts executor/chunk dynamically	Yes (live metrics)
FlatFileItemReader	Reads batch records from current file	Indirect
MyProcessor	Transforms/validates each record	Indirect
MyWriter	Writes processed chunk(s) to output	Indirect
Chunk Size Provider	Sets how many records per transaction/batch	Yes (dynamic)
Retry Handler	Retries failed chunks, with exponential backoff	Static (config)
JobListener	Reports final status/logging	N/A

How This Differs from Standard Spring Batch

Most Spring Batch solutions are **config-static**—their thread pool and chunk sizes are set at startup and never change mid-execution.
This solution is fully adaptive:

- Thread pool and chunk sizes are "live-tuned" at runtime based on hardware load.
- You get the resilience, observability, and extensibility of Spring Batch PLUS maximum use of JVM/OS resources without manual tuning or re-deployment.
- This approach is ideal for on-prem or cloud VMs with fluctuating batch load, or for maximizing throughput/cost on limited resources.

Usage Example / Next Steps

- **Deploy** the provided project, drop in your batch files, and adjust thresholds (CPU %, RAM %) as needed for your environment.
- Swap out the processor/writer to apply your real business logic or output requirements.
- Use the included code and diagrams for onboarding, documentation, or as a foundation for more advanced resource-adaptive orchestration.

References for Deeper Study (on demand)

- Spring Batch documentation on [architecture], [chunk processing], [scalability], and [dynamic configuration].[1] [2] [3] [4] [5] [6]
- Videos showing dynamic adaptivity in enterprise batch frameworks.[7] [8] [9]

This document can be used directly as your project's onboarding, architecture description, and technical justification for adopting a resource-adaptive Spring Batch workflow.

⁂

1. https://docs.spring.io/spring-batch/reference/spring-batch-architecture.html
2. https://terasoluna-batch.github.io/guideline/5.0.0.RELEASE/en/Ch02_SpringBatchArchitecture.html
3. https://stackoverflow.com/questions/55610830/spring-batch-dynamic-chunk-size
4. https://docs.spring.io/spring-batch/docs/4.1.x/reference/html/scalability.html
5. https://docs.spring.io/spring-batch/reference/step/chunk-oriented-processing.html
6. https://dzone.com/articles/spring-batch-a-batch-framework-introduction
7. https://www.youtube.com/watch?v=ZdWzewx81wA
8. https://www.youtube.com/watch?v=hr2XTbKSdAQ
9. https://www.youtube.com/watch?v=m-gp6CS55U0