**Proposed Architecture Overview**

1. **File Ingestion and Storage**
   * **Azure Blob Storage:** The incoming EOD file is uploaded to a dedicated blob container. Blob Storage is highly scalable, durable, and cost effective for handling large files. It also integrates well with other Azure services.
2. **Event-Triggered Processing**
   * **Azure Blob Trigger (Function App):** A function is configured with a blob trigger that fires whenever a new file is uploaded. This function initiates the processing pipeline by reading the file metadata and passing it along for splitting.
3. **File Splitting**
   * **File Splitter Service:** Upon the trigger, a dedicated Java/Spring microservice (or a function) reads the file and splits it into smaller chunks based on a configurable threshold (say, 500 records per file, up to a maximum defined by your business logic). Splitting minimizes memory pressure and enables parallel processing.
4. **Queueing for Parallel Processing**
   * **Azure Service Bus (or Storage Queue):** Each split chunk is then pushed as a message onto a queue. This decouples the file ingest from the processing logic and allows the “fan-out” effect where each chunk is processed independently. The queue ensures reliable message delivery and load balancing.
5. **Parallel Record Processing and Enrichment**
   * **Processing Instances (Java Spring Boot Microservices on AKS/Azure Functions):** Multiple instances of your processing service (deployed via Spring Boot in containers on Azure Kubernetes Service, or alternatively via serverless Azure Functions) listen for messages from the queue.
     + Each instance reads its assigned chunk, applies the enrichment logic, and handles exceptions by logging (without retry) any failed records. Auto scaling is enabled so that as workload increases, additional instances process chunks concurrently.
     + The use of Java and Spring Boot gives you robust development, integration, and monitoring capabilities while working with Azure Spring Cloud libraries.
6. **Staging and Aggregation**
   * **Temporary Output Storage:** Processed chunks are temporarily stored in another blob container or a database (e.g., Cosmos DB) as a staging area.
   * **Aggregation Service:** Once all chunks have been processed, an aggregation service is triggered (this could be another function or a scheduled job using Azure Data Factory) that “fans-in” the results by reading from temporary storage and merging them into a single output file.
7. **Final Output and Consumption**
   * **Consolidated Output File:** The aggregated file is then stored back in Azure Blob Storage or pushed to a downstream system for further processing or reporting. You can also expose the file via APIs if needed.

**High-Level Architecture Diagram**

****

### ****Component Details and Advantages****

1. **Azure Blob Storage**
   * **Role:** Acts as the central repository for incoming and outgoing files.
   * **Advantages:** High scalability, integration with triggers, cost-effectiveness, and managed security features.
2. **Azure Blob Trigger (Function App)**
   * **Role:** Automatically detects new file additions and initiates the file processing workflow.
   * **Advantages:** Serverless, event-driven, minimal operational overhead, and cost benefits due to pay-per-use.
3. **File Splitter Service Azure function**
   * **Role:** Splits the file into manageable chunks based on record count thresholds (e.g., 500 records minimum per chunk).
   * **Advantages:** Prevents memory overload, enables concurrent processing, and offers flexibility to adapt to varying file sizes.
4. **Azure Service Bus (or Storage Queue)**
   * **Role:** Decouples the file ingestion from processing logic by queuing each chunk for processing.
   * **Advantages:** Offers reliable message delivery, supports dead-lettering, and enables smooth auto scaling.
5. **Processing Instances (Java Spring Boot on AKS/Azure Functions)**
   * **Role:** Processes each file chunk concurrently, enriches the records, and logs any failed records.
   * **Advantages:** Auto scaling (via AKS or Functions) ensures resources match processing demands; Java Spring Boot provides robust support for business logic, exception handling, and integration with logging and monitoring tools.
6. **Exception Logging**
   * **Role:** Logs records that fail processing for further analysis while not retrying automatically.
   * **Advantages:** Helps maintain system throughput by not retrying problematic records and preserves data logs for audit and debugging. Tools like Application Insights or Log Analytics integrate seamlessly with Azure services.
7. **Temporary Output Storage and Aggregation Service**
   * **Role:** Stores processed chunks temporarily and aggregates them into a single final output file.
   * **Advantages:** Modularizes the workflow with clear separation of concerns (processing vs. aggregation) and leverages Azure Data Factory or scheduled jobs for orchestration.
8. **Final Output File Storage**
   * **Role:** The final, consolidated enriched file is stored ready for consumption or further downstream processing.
   * **Advantages:** Maintains a single source of truth and leverages Azure’s secure, scalable storage options.

### ****Why This Architecture Works Well****

* **Scalability and Resilience:** The decoupled design (using queues and microservices) provides excellent scalability. Auto-scaling features in AKS or Azure Functions ensure that bursts in load are handled without manual intervention.
* **Performance:** Splitting the file into smaller chunks allows parallel processing. Each processor works on a subset, reducing overall processing time even as record counts vary from 100 to 2 million.
* **Fault Isolation and Logging:** By handling exceptions immediately and logging failed records without retries, you maintain a high throughput while ensuring errors are captured and available for future troubleshooting.
* **Technology Synergy:** Java with Spring Boot is tried and tested for building robust microservices, and its integration with Azure (via connectors for Blob Storage, Service Bus, etc.) leverages cloud-native features, making development and maintenance efficient.
* **Cost Effectiveness:** By using serverless components where possible (Blob Trigger, Azure Functions), you only pay for what you use, and by decoupling processing, you avoid over-provisioning resources for a one-size-fits-all system.

### ****Processing Flow and Instance Behavior****

1. **Work Distribution with Split Chunks:** When the file is split into smaller chunks (each containing, for example, 500 records), every chunk is sent as a message into an Azure Service Bus queue. Each message represents a discrete unit of work that a processing instance—a Java Spring Boot microservice deployed in a container on AKS—can work on independently. This design decouples the heavy file-splitting process from the processing stage, allowing the system to tackle many chunks concurrently regardless of whether there are 100 chunks or millions.
2. **Instance Operation:** Each processing instance operates as a stateless service. Once deployed, these instances continuously poll the queue for new messages. Upon receiving a message, an instance retrieves the corresponding file chunk and starts processing by performing:
   * **Record Enrichment:** Applying business logic to transform and enhance data.
   * **Exception Handling:** Logging any failed records (without retrying them) so that they can be reviewed later.
   * **Output Staging:** Writing the successful, enriched data to a temporary location (for example, a staging blob store) for later aggregation into the final output file.
3. **Concurrency and Decoupling:** Because the microservice is stateless and each instance works independently, multiple instances can process different chunks simultaneously. This parallel processing significantly reduces overall processing time and ensures high throughput regardless of the file size.

### ****Autoscaling and Downscaling in AKS****

AKS supports dynamic scaling through Kubernetes’ **Horizontal Pod Autoscaler (HPA)**, which adjusts the number of pod instances based on real-time metrics. Let’s consider how this plays out:

1. **Scaling Out (Autoscaling Up):**
   * **Metric Monitoring:** By default, HPA can monitor CPU or memory usage. However, since our workload is message-driven, custom metrics (such as the number of messages waiting in the Azure Service Bus queue) can be integrated.
   * **Threshold-Based Scaling:** For example, if each pod is expected to efficiently handle a certain number of messages (or if the average CPU usage per pod exceeds a threshold like 70%), the HPA will trigger additional pod replicas.
   * **Load-Driven Expansion:** As more file chunks are added to the queue—say when a particularly large file is ingested—the HPA automatically increases the number of processing instances (pods). This ensures that the backlog is quickly consumed, and the processing of each chunk happens concurrently.
2. **Scaling In (Downscaling):**
   * **Reduced Workload Detection:** When the processing load decreases—indicated by lower CPU usage or a shrinking queue length—the HPA will reduce the number of active pods.
   * **Cost and Resource Efficiency:** This downscaling avoids unnecessary resource consumption and helps keep operational costs in check. The autoscaling mechanism ensures that once all the chunks have been processed and the queue is nearly empty, the system gracefully scales down to just the necessary number of instances.
3. **Custom Metrics and Fine-Tuning:**
   * **Custom Metric Example:** You could set a custom metric to monitor the queue length (e.g., average number of pending messages per pod). If that average climbs above a defined threshold, the HPA scales out; if it drops below a lower threshold, it scales in.
   * **Configuration Flexibility:** Both the minimum and maximum number of pods can be configured based on expected load patterns. For instance, if historical data shows peak times when the file routinely splits into thousands of chunks, you might set a higher maximum to swiftly absorb the burst in workload.

### ****Visual Summary****

Below is a simplified table illustrating how the autoscaling decision might map to workload metrics:

| **Metric** | **Scale Action** | **Description** |
| --- | --- | --- |
| CPU Utilization (e.g., >70%) | Scale Out | New pods are added when the existing pods are hitting high CPU usage. |
| Queue Length per Pod | Scale Out | Additional pods are provisioned if messages per pod exceed a set threshold. |
| CPU Utilization (e.g., <30%) | Scale In | Pods are reduced as the processing load diminishes. |
| Queue Length per Pod | Scale In | When the queue is almost empty, unnecessary pods are decommissioned. |

### ****Advantages of the Approach****

* **Parallelism and Performance:** By processing file splits in parallel, the solution can handle a variable number of records efficiently. This design ensures that even when file sizes are enormous, the total processing time remains low.
* **Elastic Resource Utilization:** AKS’s HPA mechanism adapts the number of processing instances dynamically based on the load. This elasticity means you only use—and pay for—the compute resources you need at any given time.
* **Fault Tolerance:** Since each pod works independently and processes tasks retrieved from the queue, individual pod failures do not jeopardize the overall process. Failed messages remain in the queue until processed by another instance.
* **Maintainability and Scalability:** Deploying microservices as containers in AKS ensures a modular and scalable architecture. Java Spring Boot provides a robust environment to implement the enrichment logic while enabling integration with monitoring and logging tools such as Application Insights.

### How Autoscaling Works in This Context

* **Queue-Driven Workload:** All split chunks are enqueued (for example, in Azure Service Bus), so various pods can concurrently pull and process available chunks. A single pod might process many chunks sequentially or even concurrently if designed to handle multiple threads.
* **Horizontal Pod Autoscaler (HPA):** HPA monitors metrics like CPU usage, memory utilization, or custom metrics (e.g., the length of the queue) to decide if the load is high enough to warrant scaling out. For instance, if the queue builds up because many chunks are waiting, HPA can add additional pods to the pool. Conversely, when the queue diminishes or resource usage drops, HPA will scale in, reducing the number of pods to meet current demand.
* **Cluster Autoscaler:** Additionally, when the HPA adds more pods, the underlying AKS cluster might need to add more nodes if the current nodes don’t have enough spare capacity. The Cluster Autoscaler ensures that there are enough resources (nodes) available to schedule the newly scaled pods.

### Key Points

* **Not One-to-One:** The design doesn’t create one new pod per chunk. Instead, multiple chunks are processed by a dynamic pool of pods.
* **Adaptable Scaling:** Autoscaling is based on aggregate system metrics like CPU load or queue depth, so scaling happens automatically in response to the overall workload, not per individual task.
* **Efficient Resource Utilization:** This approach finds a balance between handling bursts of workload (when many chunks are queued) and cost-effective operation (by reducing pods when the workload is low).

This design ensures that your system efficiently processes varying amounts of data without the overhead of creating a new instance for each chunk.