**High-Level Architecture Overview**

**Components:**

1. **Azure Blob Storage (ADLS Gen2):** Stores incoming files.
2. **Azure Event Grid:** Triggers processing upon file upload.
3. **Azure Durable Functions (Java):** Orchestrates file splitting, enrichment, and aggregation.
4. **Azure Queue Storage:** Manages tasks for parallel processing.
5. **Azure Functions (Java):** Processes file chunks concurrently.
6. **Azure Blob Storage:** Stores processed chunks and final aggregated output.
7. **Azure Monitor & Application Insights:** Logs processing details and errors.

## Detailed Component Explanation

### 1. **Azure Blob Storage (ADLS Gen2)**

* **Purpose:** Stores the raw input files.
* **Advantages:** Scalable storage with hierarchical namespace support, ideal for large dataset

### 2. **Azure Event Grid**

* **Purpose:** Detects new file uploads and triggers the Durable Function orchestrator.
* **Advantages:** Provides reliable event-driven architecture with low latency.

### 3. **Azure Durable Functions (Java)**

* **Purpose:** Manages the workflow:
  + Splits the file into chunks based on record count (e.g., 1000 records per chunk).
  + Enqueues messages for each chunk into Azure Queue Storage.
  + Monitors processing and aggregates results.
* **Advantages:** Handles stateful workflows and retries, ensuring reliability

### 4. **Azure Queue Storage**

* **Purpose:** Queues messages for each file chunk to be processed.
* **Advantages:** Decouples processing tasks, allowing scalable and parallel processing

### 5. **Azure Functions (Java)**

* **Purpose:** Processes each file chunk:
  + Reads the chunk from Blob Storage.
  + Applies enrichment logic.
  + Logs any failed records.
  + Writes the processed chunk back to Blob Storage.
* **Advantages:** Serverless compute with automatic scaling based on workload.

### 6. **Azure Blob Storage**

* **Purpose:** Stores the processed chunks and the final aggregated output file.
* **Advantages:** Centralized storage for all processed data, facilitating easy access and further processing.

### 7. **Azure Monitor & Application Insights**

* **Purpose:** Collects logs, metrics, and diagnostics:
  + Tracks processing status.
  + Logs failed records for auditing.
* **Advantages:** Provides insights into system performance and reliability.

## 🚀 Advantages of the Proposed Architecture

* **Scalability:** Azure Functions and Durable Functions automatically scale based on the number of file chunks, ensuring efficient processing regardless of file size.
* **Fault Tolerance:** Durable Functions manage state and retries, ensuring that transient failures do not disrupt the entire workflow.
* **Cost-Effective:** Serverless components mean you only pay for what you use, reducing costs during low-volume periods.
* **Maintainability:** Modular design with clear separation of concerns simplifies maintenance and future enhancements.

## 🛠️ Implementation Considerations

* **Chunk Size Determination:** Choose an optimal chunk size (e.g., 1000 records) to balance between processing efficiency and resource utilization.
* **Error Handling:** Implement robust error logging for failed records, storing them in a separate Blob Storage container for review.
* **Monitoring:** Set up alerts and dashboards in Azure Monitor to track processing times, failures, and system health.

**High-Level Architecture Diagram**

┌─────────────────────┐

│ File Producer │

│ (SFTP, External App)│

└─────────┬───────────┘

│

▼

┌─────────────────────┐

│ Azure Blob Storage │◄────────────┐

│ (Raw Container) │ │

└─────────┬───────────┘ │

│ │

Triggers Event │

▼ │

┌──────────────────────┐ │

│ Azure Event Grid │ │

└─────────┬────────────┘ │

▼ │

┌────────────────────────────┐ │

│ Azure Durable Function │ │

│ (Orchestrator in Java) │◄─────────┘

└─────┬──────────┬───────────┘

│ │

│ │

┌────────────▼─┐ ┌───▼────────────────────┐

│ File Splitter│ │ Azure Queue Storage │

│ (split into │ │ (1 message per chunk) │

│ chunks N) │ └─────────┬──────────────┘

└──────────────┘ │

▼

┌─────────────────────────────────┐

│ Azure Functions (Java Worker) │

│ - Read Chunk │

│ - Enrich Logic │

│ - Log Failed Records │

│ - Store Success to Blob │

└──────────┬──────────────────────┘

│

┌──────────▼──────────┐

│ Enriched Blob Files │

└──────────┬──────────┘

│

┌─────────────▼─────────────┐

│ Aggregator Durable Func │

│ - Merge enriched chunks │

│ - Create Final Output │

└─────────────┬─────────────┘

│

┌────────────▼────────────┐

│ Final Output File Blob │

└────────────┬────────────┘

│

┌──────────▼─────────┐

│ Application Insights│

│ + Azure Monitor │

└─────────────────────┘

## Component Responsibilities

| **Component** | **Responsibility** |
| --- | --- |
| **Azure Blob Storage** | Input/output data storage. Organized as: raw/, chunks/, processed/, error/ |
| **Azure Event Grid** | Detects new files in Blob and triggers the orchestrator |
| **Durable Function (Orchestration)** | Splits file into chunks, queues tasks, waits for processing completion |
| **Azure Queue Storage** | Stores task metadata for each chunk |
| **Azure Function Workers** | Java-based processors: read → enrich → write → log errors |
| **Aggregator Function** | Gathers all processed chunks, merges and writes final output |
| **Application Insights** | Monitors logs, traces, error records |

**✅ Benefits**

* **Scalable:** Auto-scales Azure Functions to match file size and chunk load.
* **Fault-tolerant:** Errors are isolated per chunk, logged, and do not halt full processing.
* **Serverless Efficiency:** Cost-effective; no infrastructure management required.
* **Observability:** Built-in monitoring and error tracking.
* **Cloud Native:** Uses durable workflows, blob events, and native storage to optimize processing.

## 🔄 Azure Function Auto-Scaling in This Architecture

### ✅ Short Answer:

**You do NOT get one Azure Function instance per chunk.**  
Azure Functions **scale dynamically based on workload** — i.e., **how many chunk messages are waiting in the queue**, constrained by concurrency, CPU, and memory.

## 🔍 Detailed Breakdown:

### 🎯 You have:

* 1 Durable Function that splits a file into N chunks (e.g., 1000 records per chunk).
* Each chunk → 1 message → Azure Queue Storage.
* A single Java-based Azure Function is **bound to the queue**, and it:
  + Triggers on each message
  + Processes the chunk

### 🧠 How scaling works:

* Azure monitors **queue length** and **function execution time**.
* Based on this, it scales out (adds more instances) or scales in (removes instances).
* Each Function instance **can process 1 or more messages concurrently** depending on settings (batchSize, maxConcurrentCalls).

## ⚙️ Example Scenario:

| **Scenario** | **Result** |
| --- | --- |
| 1,000 chunks in queue | Azure starts 5–50 function instances (varies) |
| Each instance processes 1–10 msgs/sec | Parallelism improves throughput |
| Queue drains quickly | Instances automatically **scale down** |

## 📈 Azure Function Scaling Characteristics

| **Metric** | **Behavior** |
| --- | --- |
| **Startup Time** | Java cold start ~2–10s (first trigger) |
| **Max Instances** | Default limit: 200 (App Service Plan / Consumption) |
| **Concurrency** | Configurable: maxConcurrentCalls, batchSize (in host.json) |
| **Downscaling** | Happens automatically after idle time (~1-2 minutes) |
| **Consumption Plan** | Best for burst loads, cost-effective |
| **Premium Plan** | Required for VNET, no cold start, more control |

## ✅ Summary: Scaling Model

* **You deploy ONE function app** (Java Function bound to the queue).
* **Azure auto-creates 1+ instances** of it as needed based on the **queue size and throughput**.
* You don’t create an instance per chunk manually.
* Scaling is automatic and **concurrent chunk processing is handled by multiple function instances** behind the scenes.

Would you like:  
**a.** host.json config for concurrency tuning in Java Azure Function  
**b.** Throughput benchmark estimation for 1M chunks using Consumption vs Premium plan

Top of Form

Bottom of Form

## ⚙️ Example Scenario:

| **Scenario** | **Result** |
| --- | --- |
| 1,000 chunks in queue | Azure starts 5–50 function instances (varies) |
| Each instance processes 1–10 msgs/sec | Parallelism improves throughput |
| Queue drains quickly | Instances automatically **scale down** |

## 📈 Azure Function Scaling Characteristics

| **Metric** | **Behavior** |
| --- | --- |
| **Startup Time** | Java cold start ~2–10s (first trigger) |
| **Max Instances** | Default limit: 200 (App Service Plan / Consumption) |
| **Concurrency** | Configurable: maxConcurrentCalls, batchSize (in host.json) |
| **Downscaling** | Happens automatically after idle time (~1-2 minutes) |
| **Consumption Plan** | Best for burst loads, cost-effective |
| **Premium Plan** | Required for VNET, no cold start, more control |

**✅ Summary: Scaling Model**

* **You deploy ONE function app** (Java Function bound to the queue).
* **Azure auto-creates 1+ instances** of it as needed based on the **queue size and throughput**.
* You don’t create an instance per chunk manually.
* Scaling is automatic and **concurrent chunk processing is handled by multiple function instances** behind the scenes.