**✅ Goal Recap**

* You get **one file per day**, size varies: 100 to 2 million lines.
* Need to **scale compute (CPU/memory)** based on **file size**.
* Processing must be **sequential**, so only **one pod per file**.
* You want to **auto-scale up/down** dynamically.

**✅ Short Answer First**

✅ Yes, you will create **3 separate Kubernetes deployments**:

* file-processor-small
* file-processor-medium
* file-processor-large

Each deployment has:

* Its own **resource config** (CPU/memory).
* Its own **Service Bus queue**.
* Its own **KEDA ScaledObject** watching that queue.

🔁 **Only one of them will be active at a time** — depending on which queue receives the message for today’s file.

**🎯 Step-by-Step: How KEDA + Queues + Pods Work**

**🔹 1. File Uploads to Blob Storage**

* A daily file is uploaded (e.g., 1.6M lines).
* Trigger a **preprocessing step** (Azure Function or Spring Boot service) to:
  + Count/estimate file size.
  + Based on size → route message to:
    - file-small-queue (if <50k lines)
    - file-medium-queue (50k–500k)
    - file-large-queue (>500k lines)

**🔹 2. Kubernetes Deployments (3 Types)**

Each pod image is identical (same Spring Boot + Camel code), but the deployments differ by:

|  |  |  |
| --- | --- | --- |
| **Deployment Name** | **CPU / Memory** | **Service Bus Queue Watched** |
| file-processor-small | 0.25 / 512MB | file-small-queue |
| file-processor-medium | 1 CPU / 2GB | file-medium-queue |
| file-processor-large | 2 CPU / 4GB | file-large-queue |

Each has a corresponding **ScaledObject**.

**🔹 3. KEDA + ScaledObject (per deployment)**

Each ScaledObject is configured like this (example: large tier):

apiVersion: keda.sh/v1alpha1

kind: ScaledObject

metadata:

name: file-processor-large-scaler

spec:

scaleTargetRef:

name: file-processor-large

pollingInterval: 30 # seconds

cooldownPeriod: 300 # scale down after 5 mins of idle

minReplicaCount: 0

maxReplicaCount: 1

triggers:

- type: azure-servicebus

metadata:

queueName: file-large-queue

messageCount: "1"

connectionFromEnv: SERVICEBUS\_CONN

Repeat the above for small and medium, adjusting names and queue.

**🔹 4. At Runtime: Which Pod is Active?**

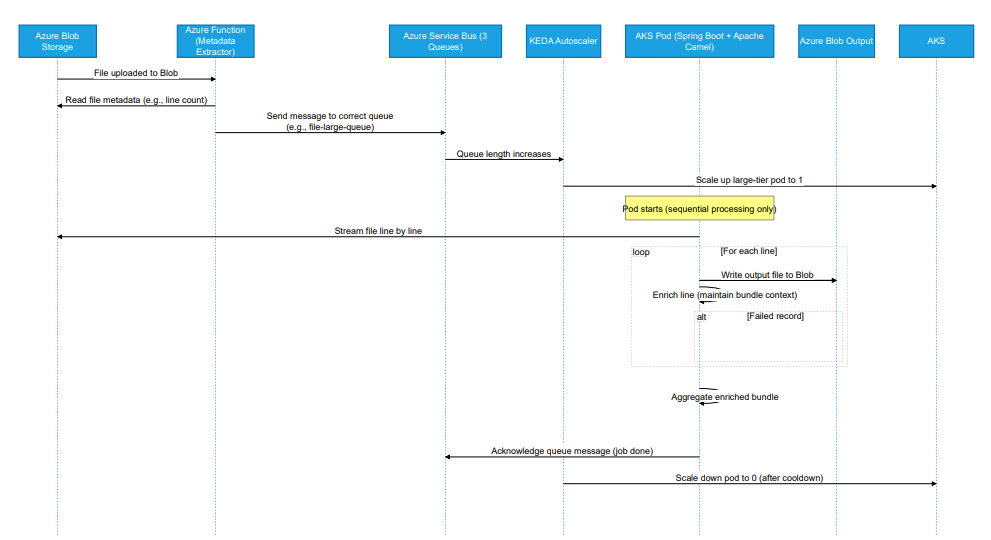
* Only the **deployment corresponding to the queue with a message** will scale **from 0 → 1 pod**.
* **Other 2 deployments remain scaled to 0.**
* Once processing completes (file done, queue empty), KEDA will **scale back down to 0**.

**✅ Benefits of This Approach**

|  |  |
| --- | --- |
| **Feature** | **Advantage** |
| Efficient resource use | Big files → big pods, small files → tiny pods |
| Fully auto-scaled | No compute waste, 0 → 1 → 0 pod lifecycle |
| Easy isolation | One pod = one file = traceable |
| Single shared logic | Same Spring Boot + Camel image for all |

**❗ Important Notes**

* Only one pod active per day.
* No overlap/duplication — job routing ensures **only one queue has a message**.
* KEDA uses **event-driven polling**, no manual pod management needed.



**Full Flow**

**🔹 1. File Upload → Azure Blob Storage**

* A file (e.g., eod-20240518.csv) is uploaded to input-container.

**🔹 2. Blob Triggered → Azure Function Executes**

* Your Azure Function is triggered by the blob upload.
* It reads the file **content**, counts lines, and determines size tier.
* It constructs a message:

json

CopyEdit

{  
 "fileName": "eod-20240518.csv",  
 "lineCount": 1450000,  
 "blobUrl": "<https://.../eod-20240518.csv>"  
}

* Sends the message to the correct **Azure Storage Queue**:
  + <50k → file-small-queue
  + 50k–500k → file-medium-queue
  + >500k → file-large-queue

**🔹 3. KEDA Watches Queues**

* KEDA watches:
  + file-small-queue
  + file-medium-queue
  + file-large-queue

Each has a separate ScaledObject targeting different **Spring Boot Camel Deployments** with appropriate CPU/memory.

✅ When a message appears in file-large-queue, KEDA triggers:

yaml

Deployment: file-processor-large  
Pod: 1 replica

**🔹 4. Apache Camel (Spring Boot) is Scaled & Triggered**

* This pod has a route like:

from("azure-storage-queue:file-large-queue?...")

* That route:
  + Reads the message
  + Downloads the blob using blobUrl
  + Streams it line by line
  + Performs enrichment + aggregation
  + Writes result to output

**✅ Summary: Your Understanding**

|  |  |
| --- | --- |
| **Component** | **Behavior** |
| Azure Function | Detects size, routes message to the correct queue |
| KEDA | Watches queues, auto-scales the matching pod |
| Queue Name | Determines **which processing pod is triggered** |
| Apache Camel App | Reads from that queue and processes the file |

✅ **Yes, the queue name maps directly to the input route** in the Camel processor.

**✅ Best Practice: Separate 3 Camel Projects + Deployments**

Because:

* You want **fine-grained resource scaling** (e.g. large files = more memory).
* You’re using **KEDA to autoscale per queue**, which needs **separate Deployments**.
* You can’t dynamically “shut off” unused routes in a monolithic app.

**🔍 Comparison: 1 App vs 3 Apps**

|  |  |  |
| --- | --- | --- |
| **Criteria** | **Option 1: Single App (3 Routes)** | **Option 2: Separate Projects (Recommended)** |
| ✅ Simple codebase | ✅ Fewer files to manage | 🔶 Needs duplication (can use shared lib) |
| ❌ Resource scaling | ❌ Same resources for all file sizes | ✅ Each pod has custom CPU/memory |
| ❌ Autoscaling precision | ❌ Only 1 pod for all queues (poor isolation) | ✅ KEDA scales only the right queue |
| ✅ Easier to build once | ✅ One Docker image | 🔶 Multiple images to manage |
| ✅ Clean separation | ❌ Mixed logic for small/medium/large files | ✅ Cleanly scoped per file type |
| 🔒 Failure isolation | ❌ One route fail = risk for all | ✅ Failures isolated to individual pods |

**Real-world Analogy:**

Imagine you run 3 workloads:

* 1 CPU job (small files)
* 2 CPU job (medium)
* 4 CPU job (large files)

If you use 1 app:

* You must provision **maximum CPU/RAM** for all routes — **wasteful**
* All 3 routes are always running — even if only 1 file type is present

If you use 3 apps:

* Each app is **tailored, scalable, cost-efficient**
* Only **1 of them runs** when needed

**✅ Recommendation**

Build **3 separate Spring Boot + Camel apps**, one per queue:

* file-processor-small
* file-processor-medium
* file-processor-large
* Use shared module/library for common logic (e.g. enrichment, blob reader)
* Configure **separate Deployment.yaml + ScaledObject.yaml** per queue
* Build/deploy each with independent resource limits

**✅ Updated Recommendation: One Camel App, Multiple Deployments via Parameterization**

🎯 **Best of both worlds**:

* 🟢 Single codebase = Easy maintenance
* 🟢 Route activation is **driven by environment variable**
* 🟢 Use the **same Docker image**, deployed 3 times with different config
* 🟢 KEDA handles queue-based autoscaling

**🧩 How It Works**

**1. One Camel App with route like:**

java

CopyEdit

String queueName = System.getenv("QUEUE\_NAME");

from("azure-storage-queue:" + queueName + "?connectionString=${AZURE\_STORAGE\_CONNECTION\_STRING}")

.routeId("file-processor-route-" + queueName)

.log("[INFO] Processing queue: " + queueName)

.process(...) // same enrichment, aggregation logic

**2. 3 Deployments (AKS YAML)**

Each deployment injects a different env var:

yaml

CopyEdit

env:

- name: QUEUE\_NAME

value: file-small-queue

And KEDA ScaledObject watches that specific queue.

**🔁 Deployment Flow Summary**

| **Pod / App** | **Env Var** | **Scales When** |
| --- | --- | --- |
| file-processor-small | QUEUE\_NAME=... | file-small-queue > 0 |
| file-processor-medium | QUEUE\_NAME=... | file-medium-queue > 0 |
| file-processor-large | QUEUE\_NAME=... | file-large-queue > 0 |

They all use the **same JAR/image**, same logic, same route code — just configured differently at runtime.

**✅ Advantages of This Approach**

| **Benefit** | **Why it matters** |
| --- | --- |
| ✅ Single Codebase | Maintainable, DRY, easy to test |
| ✅ Separate Pods | KEDA scales each independently |
| ✅ Configurable via env var | No code duplication needed |
| ✅ One Docker Image | Simpler CI/CD pipeline |
| ✅ Resource Optimization | You set CPU/RAM limits per Deployment |

**🧠 Pro Tip**

To make this flexible:

* Use a @Component class to register routes dynamically
* Validate QUEUE\_NAME at startup and only activate that route