PuigFlows

IA y Automatizacion - Guias tecnicas de produccion

Autoescalado con KEDA en Kubernetes: Escalar a Cero, Triggers por Eventos y Ajuste Fino del HPA

Kubernetes Autoscaling with KEDA: Scale to Zero, Event-Driven Triggers and Fine-Tuning the HPA

Autor / Author:	PuigFlows
Tipo / Type:	Guia tecnica
Tiempo de lectura / Reading time:	37 min
Edicion / Edition:	2026-09-09
Idiomas / Languages:	Espanol + English
Fuente / Source:	https://www.puigflows.com/recursos

Documento generado automaticamente a partir del recurso publicado. Contiene la version completa en espanol y en ingles. / Automatically generated from the published resource. Contains the full Spanish and English versions.

Version en espanol

El problema: la CPU no sabe cuánta cola tienes

Un worker que consume mensajes de una cola pasa casi todo su tiempo esperando E/S: recibe un lote, llama a una API externa, escribe en Postgres y vuelve a esperar. Su CPU media ronda el 10 %. Si lo autoescalas contra `cpu: 70%`, ese worker no escalará jamás, por muchos mensajes que se acumulen detrás. Y cuando por fin escale --porque un pico saturó un núcleo-- ya llevarás veinte minutos de retraso acumulado que nadie va a recuperar.

La señal que importa no es la CPU: es el *backlog*. Cuántos mensajes hay sin procesar, cuánto lleva esperando el más antiguo, cuántas peticiones por segundo entran. KEDA existe para conectar esas señales al autoescalado de Kubernetes y, de paso, resolver algo que el HPA no puede hacer solo: bajar a cero réplicas.

Cómo funciona el HPA por debajo

Antes de añadir una pieza nueva conviene entender la que ya tienes. El HorizontalPodAutoscaler vive dentro del `kube-controller-manager` y reevalúa cada objetivo cada 15 segundos por defecto (`--horizontal-pod-autoscaler-sync-period`). En cada ciclo aplica una fórmula que cabe en una línea:

```
desiredReplicas = ceil[ currentReplicas × ( currentMetricValue / desiredMetricValue ) ]
```

Con dos matices que explican el 90 % del comportamiento aparentemente raro:

- **Tolerancia.** Si el cociente cae dentro del ±10 % (`--horizontal-pod-autoscaler-tolerance`, 0.1 por defecto), el HPA no mueve nada. Desde Kubernetes 1.34 puedes fijar una `tolerance` distinta por política de `scaleUp` y `scaleDown`; en 1.33 era alfa tras el feature gate `HPAConfigurableTolerance`.
- **Sólo cuentan los pods listos.** Los pods que aún no pasan el readiness probe, o cuya métrica falta, se descartan del cálculo. Es lo que evita que un arranque lento se interprete como falta de capacidad y dispare una avalancha de réplicas.

El campo `behavior` controla la velocidad, y sus valores por defecto son asimétricos a propósito:

```
behavior:
  scaleUp:
    stabilizationWindowSeconds: 0
    selectPolicy: Max
    policies:
    - type: Percent
      value: 100
      periodSeconds: 15
    - type: Pods
      value: 4
      periodSeconds: 15
  scaleDown:
    stabilizationWindowSeconds: 300
    policies:
    - type: Percent
      value: 100
      periodSeconds: 15
```

Traducido: sube rápido (puede duplicar las réplicas o añadir 4 pods cada 15 s, lo que sea mayor) y baja despacio (mira los últimos 5 minutos y se queda con la recomendación más alta de la ventana). Ese `stabilizationWindowSeconds: 300` es la razón por la que tu deployment tarda cinco minutos en encogerse tras un pico, y es un valor sano: evita el *flapping*.

Lo que el HPA no hará nunca es bajar de `minReplicas: 1`. No es una cuestión de configuración, es del controlador.

Qué añade KEDA exactamente

KEDA (Kubernetes Event-Driven Autoscaling, proyecto graduado de la CNCF) son esencialmente dos componentes:

- El **operator**, que observa tus `ScaledObject` y gobierna el tramo 0 1. Cuando un trigger reporta actividad estando a cero réplicas, el operator parchea directamente el subrecurso `/scale` del Deployment.
- El **metrics adapter**, que implementa `external.metrics.k8s.io` y sirve al HPA los valores de tus triggers: longitud de cola, resultado de una query de Prometheus, lag de un consumer group de Kafka, lo que sea.

Ese reparto es clave para depurar: **de 0 a 1 manda KEDA; de 1 a N manda el HPA**. Por eso `pollingInterval` sólo afecta al arranque desde cero y `cooldownPeriod` sólo al apagado hasta cero.

```
helm repo add kedacore https://kedacore.github.io/charts
helm repo update
helm install keda kedacore/keda --namespace keda --create-namespace

# El adapter debe aparecer como Available=True
kubectl get apiservice v1beta1.external.metrics.k8s.io
```

Tu primer ScaledObject

Un worker que procesa imágenes desde SQS. Ni una línea de más:

```
apiVersion: keda.sh/v1alpha1
kind: ScaledObject
metadata:
  name: image-worker
  namespace: media
spec:
  scaleTargetRef:
    name: image-worker           # Deployment, mismo namespace
  pollingInterval: 15            # cada cuánto consulta KEDA estando a 0
  cooldownPeriod: 120            # espera tras la última actividad antes de volver a 0
  minReplicaCount: 0
  maxReplicaCount: 40
  triggers:
  - type: aws-sqs-queue
    metricType: AverageValue
    metadata:
      queueURL: https://sqs.eu-west-1.amazonaws.com/123456789012/image-jobs
      queueLength: "20"            # objetivo: 20 mensajes por réplica
      activationQueueLength: "1"   # con 1 mensaje ya arranca desde cero
      awsRegion: eu-west-1
    authenticationRef:
      name: keda-aws-irsa
```

Dos conceptos que se confunden constantemente:

- `queueLength: "20"` es un objetivo **por réplica**, no total. Con `metricType: AverageValue` --el valor por defecto de casi todos los scalers-- el HPA divide el backlog entre las réplicas listas. 200 mensajes en cola tienden a 10 pods.
- `activationQueueLength` es el umbral **0 -> 1** y lo evalúa KEDA, no el HPA. Son dos umbrales distintos y por eso existen dos campos. Si pones `minReplicaCount: 1`, el umbral de activación se ignora por

completo.

Aplica y observa. KEDA habrá creado un HPA llamado `keda-hpa-image-worker`:

```
kubectl get scaledobject image-worker -n media
kubectl get hpa keda-hpa-image-worker -n media -w
kubectl describe scaledobject image-worker -n media    # condiciones: Ready, Active, Fallback
kubectl logs -n keda -l app=keda-operator -f
```

La condición `Active` responde a la pregunta más frecuente: "¿por qué sigo a cero?". Si está en `False`, ningún trigger supera su umbral de activación.

Escalar a cero sin castigar al usuario

Bajar a cero ahorra dinero real: en workers intermitentes es habitual recortar entre un 60 % y un 90 % del gasto ocioso. Pero el primer mensaje de cada ráfaga paga el arranque en frío, y ese arranque no es un número, es una suma:

1. **Detección**: hasta `pollingInterval` segundos hasta que KEDA ve el mensaje.
2. **Programación**: segundos si hay nodo con hueco; 1-3 minutos si el cluster autoscaler o Karpenter tiene que levantar uno.
3. **Descarga de imagen**: 0 s con caché en el nodo, decenas de segundos con una imagen de 2 GB en un nodo recién creado.
4. **Arranque de la aplicación**: JVM que calienta, pool de conexiones, modelo que se carga en memoria.

Reglas que aguantan en producción:

- Escala a cero **workers y jobs**, no APIs con SLO de latencia. Para una API HTTP, `minReplicaCount: 1`, o 2 si te importa seguir disponible durante un rolling update.
- Baja `pollingInterval` a 5-15 s en las colas donde el arranque importa. Es una llamada a la API del broker cada N segundos y por ScaledObject: barato, pero no gratis con cientos de objetos.
- Adelgaza la imagen. Un worker distroless de 80 MB arranca en frío en una fracción de lo que tarda uno de 1,5 GB.
- Si tienes un patrón horario conocido, combina un trigger `cron` con el de cola: mantén un mínimo caliente en horario laboral y deja que baje a cero de madrugada.

```
  triggers:
  - type: cron
    metadata:
      timezone: Europe/Madrid
      start: 0 8 * * 1-5        # lunes a viernes, 08:00
      end: 0 20 * * 1-5         # hasta las 20:00
      desiredReplicas: "3"
  - type: aws-sqs-queue
    metadata:
      queueURL: https://sqs.eu-west-1.amazonaws.com/123456789012/image-jobs
      queueLength: "20"
      awsRegion: eu-west-1
```

Con varios triggers, el HPA calcula la recomendación de cada métrica y se queda con la **mayor**. El cron actúa entonces como suelo durante su ventana, y la cola manda el resto del tiempo.

Escalar por métricas de negocio con Prometheus

Los scalers de cola cubren mucho terreno, pero muchas veces la señal correcta ya está en Prometheus: la

profundidad del backlog, la edad del mensaje más antiguo, las peticiones en vuelo. El scaler `prometheus` convierte cualquier query en una métrica de autoescalado.

```
  triggers:
  - type: prometheus
    name: backlog
    metricType: AverageValue
    metadata:
      serverAddress: http://prometheus-operated.monitoring.svc:9090
      query: sum(rabbitmq_queue_messages_ready{queue="invoices"})
      threshold: "50"             # 50 mensajes pendientes por réplica
      activationThreshold: "5"    # por debajo de 5, quédate a cero
      ignoreNullValues: "false"   # query sin datos = error, no = cero
```

Esa última línea es una decisión deliberada. Por defecto, `ignoreNullValues` vale `true` y una query que no devuelve series se interpreta como 0, así que tu deployment se apaga justo cuando el sistema de métricas se rompe. Con `false`, la ausencia de datos cuenta como fallo del scaler y activa el `fallback` que vemos a continuación.

Cuando necesitas combinar señales --escalar por backlog *pero también* si la latencia p95 se dispara-- usa `scalingModifiers`:

```
spec:
  advanced:
    scalingModifiers:
      target: "1"
      metricType: AverageValue
      formula: "max(backlog / 50, p95_latency / 0.3)"
  triggers:
  - type: prometheus
    name: backlog
    metadata:
      serverAddress: http://prometheus-operated.monitoring.svc:9090
      query: sum(rabbitmq_queue_messages_ready{queue="invoices"})
      threshold: "50"
  - type: prometheus
    name: p95_latency
    metadata:
      serverAddress: http://prometheus-operated.monitoring.svc:9090
      query: histogram_quantile(0.95, sum by (le) (rate(job_duration_seconds_bucket[5m])))
      threshold: "0.3"
```

La fórmula se evalúa con el lenguaje *expr* y debe devolver un número, no un booleano. El truco es normalizar cada señal a "unidades de réplica" dividiéndola por su objetivo y quedarse con la más exigente; el `target: "1"` del modificador es entonces el denominador final del HPA. Cada trigger que aparezca en la fórmula necesita obligatoriamente su campo `name`.

Que un scaler caído no tumbe el servicio

Si Prometheus se cae, el scaler falla y el HPA se queda sin métrica. Sin configuración adicional el HPA *congela* el número de réplicas actual, lo que puede ser desastroso si el incidente empezó precisamente con un pico de tráfico. La sección `fallback` define el plan B:

```
spec:
  fallback:
    failureThreshold: 3                 # 3 fallos consecutivos del scaler
    replicas: 6                         # ...y me planto en 6 réplicas
    behavior: currentReplicasIfHigher   # o me quedo con las actuales si son más
```

`currentReplicasIfHigher` es la opción segura por defecto: nunca reduce capacidad durante un apagón de la observabilidad. Dos límites que conviene saber: `fallback` no funciona con triggers de CPU ni de

memoria, y no está soportado en `ScaledJob`.

Ajustar la velocidad desde el propio ScaledObject

```
spec:
  advanced:
    restoreToOriginalReplicaCount: true
    horizontalPodAutoscalerConfig:
      behavior:
        scaleUp:
          stabilizationWindowSeconds: 0
          selectPolicy: Max
          policies:
          - type: Percent
            value: 200
            periodSeconds: 30       # puede triplicar cada 30 s
          - type: Pods
            value: 10
            periodSeconds: 30
        scaleDown:
          stabilizationWindowSeconds: 600
          policies:
          - type: Pods
            value: 2
            periodSeconds: 60       # como mucho, -2 pods por minuto
```

Cómo elegir esos números sin adivinar: mide cuánto tarda un pod tuyo en estar *Ready y produciendo*, no sólo Ready. Si son 90 segundos, un `periodSeconds` de 15 hará que el HPA añada pods seis veces antes de que el primero empiece a trabajar, y sobreescalarás sistemáticamente. Pon `periodSeconds` del orden del tiempo de arranque real.

Para el `scaleDown`, la pregunta es cuánto cuesta equivocarse. Con jobs cortos e idempotentes, bajar rápido es barato. Con jobs de diez minutos que habría que reiniciar desde cero, una ventana de 600 s y un techo de 2 pods por minuto sale mucho más barata que los reintentos.

`restoreToOriginalReplicaCount: true` merece una nota aparte: si borras el ScaledObject, el comportamiento por defecto es dejar el Deployment con las réplicas que tuviera en ese instante, incluido 0. Más de un servicio "desaparecido" tras un `kubectl delete scaledobject` se explica exactamente así.

KEDA por dentro: dos procesos, una API de Kubernetes y un HPA que no escribiste

Antes de ajustar un solo parámetro conviene tener el mapa mental correcto, porque casi todos los problemas raros de KEDA se explican mirando qué componente está fallando. Cuando instalas el chart aparecen dos cargas de trabajo distintas con responsabilidades que no se solapan.

El **operador** (*keda-operator*) observa los recursos *ScaledObject* y *ScaledJob*. Su trabajo es traducir tu declaración a objetos nativos: por cada ScaledObject crea y mantiene un *HorizontalPodAutoscaler* que tú nunca escribes y que se llama *keda-hpa-nombre-de-tu-scaledobject*. También es quien decide las transiciones entre cero y uno, que el HPA no sabe hacer.

El **servidor de métricas** (*keda-operator-metrics-apiserver*) implementa la API de métricas externas de Kubernetes y se registra como *APIService* para *external.metrics.k8s.io/v1beta1*. Cuando el HPA necesita saber cuántos mensajes hay en la cola, no habla con SQS: pregunta a la API de Kubernetes, que enruta la pregunta a este servidor, que a su vez consulta al scaler correspondiente.

```
kubectl get pods -n keda
kubectl get apiservice v1beta1.external.metrics.k8s.io
kubectl get hpa -A | grep keda-hpa

# Ver la metrica exacta que el HPA esta leyendo, tal cual la sirve KEDA:
kubectl get --raw "/apis/external.metrics.k8s.io/v1beta1/namespaces/media/s0-aws-sqs-queue?labelSelect
or=scaledobject.keda.sh/name=image-worker" | jq
```

Ese último comando es la herramienta de diagnóstico más útil que existe para KEDA y casi nadie la usa. Te dice si el problema está en el scaler (no devuelve valor) o en el HPA (recibe el valor y no actúa), que son dos incidentes completamente distintos con dos arreglos completamente distintos.

El conflicto que rompe clusters ya montados

Hay una restricción de Kubernetes que conviene conocer antes de instalar nada: **sólo puede haber un proveedor registrado para *external.metrics.k8s.io* en todo el cluster**. Es un *APIService* de ámbito global, no de namespace.

Si tu cluster ya tiene *prometheus-adapter* sirviendo métricas externas y despliegas KEDA encima, KEDA sobrescribe el registro y el adaptador anterior deja de responder. Los HPA que dependían de él no dan error llamativo: se quedan con la métrica en *unknown* y dejan de escalar. Es un fallo silencioso que puede tardar días en notarse, y suele descubrirse cuando llega el pico de tráfico.

La salida es elegir un único proveedor. Si necesitas métricas de Prometheus, no instales los dos: usa el scaler *prometheus* de KEDA, que hace la misma consulta sin necesidad del adaptador. Si por alguna razón tienes que conservar *prometheus-adapter*, entonces KEDA no puede vivir en ese cluster, y esa es la conversación que hay que tener antes de abrir el pull request, no después.

El webhook de admisión

El chart instala también un *ValidatingWebhookConfiguration* que rechaza ScaledObjects incoherentes en el momento de aplicarlos: dos ScaledObjects apuntando al mismo *scaleTargetRef*, un HPA propio que ya gestiona ese Deployment, un *minReplicaCount* mayor que el *maxReplicaCount*. Es una red de seguridad excelente y también una fuente de sorpresas en CI si tu pipeline aplica manifiestos con *--validate=false* o si el webhook no está disponible: en ese caso los recursos entran sin comprobar y el error aparece en tiempo de ejecución.

Threshold y activationThreshold: los dos números que casi todo el mundo confunde

Si sólo te llevas una idea de esta guía, que sea esta. En un trigger de KEDA hay dos umbrales y significan cosas distintas, gobiernan transiciones distintas y los ejecutan componentes distintos.

- ***threshold*** (o el parámetro equivalente del scaler: *queueLength*, *lagThreshold*, *threshold*...) es el objetivo por réplica que se entrega al HPA. Es el denominador de la fórmula del HPA. Gobierna la escala entre 1 y N.
- ***activationThreshold*** es el valor mínimo de la métrica para que KEDA considere el scaler *activo*. Gobierna únicamente la transición entre 0 y 1, y lo evalúa el operador, no el HPA.

De ahí salen dos consecuencias que explican la mayoría de los comportamientos "raros". La primera: si tu *minReplicaCount* es 1 o más, el scaler está siempre activo y **el *activationThreshold* se ignora por completo**. Ponerlo ahí no hace nada, y el equipo pasa una tarde ajustando un número que no se lee. La segunda: *activationThreshold* no es un porcentaje ni una fracción del *threshold*, es un valor absoluto de la

métrica, y si lo dejas en su valor por defecto de cero, cualquier mensaje suelto despierta el servicio.

```
apiVersion: keda.sh/v1alpha1
kind: ScaledObject
metadata:
  name: image-worker
  namespace: media
spec:
  scaleTargetRef:
    name: image-worker
  minReplicaCount: 0
  maxReplicaCount: 40
  pollingInterval: 15        # cada cuanto consulta KEDA al scaler
  cooldownPeriod: 300        # espera antes de bajar de 1 a 0
  triggers:
    - type: aws-sqs-queue
      metadata:
        queueURL: https://sqs.eu-west-1.amazonaws.com/111122223333/imagenes
        awsRegion: eu-west-1
        queueLength: "20"          # objetivo POR REPLICA -> lo usa el HPA
        activationQueueLength: "5" # despierta desde 0 solo con 5+ mensajes
        scaleOnInFlight: "true"    # cuenta tambien los mensajes en vuelo
```

Con esa configuración, cuatro mensajes sueltos a las tres de la madrugada no arrancan un pod, un nodo y una alerta. A partir de cinco sí. Y una vez despierto, el HPA apunta a veinte mensajes por réplica.

Merece la pena detenerse en *scaleOnInFlight*. Por defecto el scaler de SQS suma los mensajes visibles y los que están siendo procesados (*ApproximateNumberOfMessagesNotVisible*). Eso es lo correcto casi siempre, porque un mensaje en vuelo sigue ocupando un worker. Pero si tu procesamiento es muy largo, la métrica no baja mientras trabajas y el autoescalador cree que la cola sigue llena; en ese caso, ponerlo a *false* hace que la métrica refleje sólo trabajo pendiente. No hay una respuesta universal: depende de si tu cuello de botella es la cola o el tiempo por mensaje.

La aritmética completa, con números

Vale la pena hacer el recorrido entero una vez, porque cuando lo has hecho a mano ya no vuelves a discutir sobre por qué hay siete réplicas y no cinco.

Supón un worker de SQS con *queueLength: "20"*, *minReplicaCount: 0*, *maxReplicaCount: 40* y ahora mismo 3 réplicas corriendo. Llega un pico y la cola sube a 260 mensajes (visibles más en vuelo).

1. KEDA consulta el scaler cada *pollingInterval* y publica el valor en la API de métricas externas. Publica el **total**, 260, con un objetivo de tipo *AverageValue* de 20.
2. El HPA lee 260 y calcula la media por réplica: 260 / 3 = 86,67 mensajes por pod.
3. Aplica la fórmula: *ceil(3 x (86,67 / 20)) = ceil(13) = 13* réplicas.
4. Comprueba el *behavior* de subida. Si tienes una política de "como mucho el 100% cada 60 segundos", no salta a 13 de golpe: va a 6, luego a 12, luego a 13.
5. Cuando la cola se vacía, el HPA calcula 0 réplicas necesarias pero **no baja de *minReplicaCount***. La bajada final de 1 a 0 la hace el operador de KEDA cuando el scaler lleva inactivo el *cooldownPeriod* completo.

De ese recorrido salen dos aclaraciones importantes. La primera es que ***cooldownPeriod* sólo afecta a la transición de 1 a 0**. Si tu servicio baja de 30 réplicas a 4 demasiado rápido, tocar el *cooldownPeriod* no cambia nada: lo que manda ahí es la ventana de estabilización del HPA, que por defecto son 300 segundos y se ajusta en *behavior.scaleDown*. La segunda es que la fórmula usa el techo, así que en cargas pequeñas siempre redondea hacia arriba: con un objetivo de 20 y 21 mensajes, pides 2 réplicas.

```
spec:
  advanced:
    horizontalPodAutoscalerConfig:
      behavior:
        scaleUp:
          stabilizationWindowSeconds: 0      # reaccionar de inmediato al pico
          policies:
            - type: Percent
              value: 100
              periodSeconds: 30
            - type: Pods
              value: 10
              periodSeconds: 30
          selectPolicy: Max                  # la que permita crecer mas rapido
        scaleDown:
          stabilizationWindowSeconds: 300    # no bajar por un valle de 30 s
          policies:
            - type: Percent
              value: 25
              periodSeconds: 60
```

La asimetría es deliberada y es la configuración correcta para casi cualquier worker de cola: subir rápido cuesta dinero durante minutos, bajar rápido cuesta latencia y mensajes reprocesados durante el siguiente pico.

ScaledJob: cuando la unidad de trabajo no es un pod que vive para siempre

Un *ScaledObject* ajusta el número de réplicas de un Deployment que se queda esperando trabajo. Hay cargas donde eso es el modelo equivocado: transcodificaciones de veinte minutos, informes que se generan una vez, migraciones por lotes. Si un pod procesa un mensaje y debe morir, un Deployment te obliga a inventar el bucle de espera, la señal de apagado y la lógica de "ya no queda trabajo". Para eso existe *ScaledJob*: KEDA crea un *Job* de Kubernetes por unidad de trabajo y deja que el planificador y el *backoffLimit* hagan lo suyo.

```
apiVersion: keda.sh/v1alpha1
kind: ScaledJob
metadata:
  name: transcodificador
  namespace: media
spec:
  jobTargetRef:
    parallelism: 1
    completions: 1
    backoffLimit: 2
    template:
      spec:
        restartPolicy: Never
        containers:
          - name: worker
            image: registry.interno/transcodificador:1.9.3
            resources:
              requests: { cpu: "2", memory: 4Gi }
              limits:   { memory: 6Gi }
  pollingInterval: 20
  maxReplicaCount: 60              # tope de Jobs simultaneos (por defecto 100)
  successfulJobsHistoryLimit: 5
  failedJobsHistoryLimit: 20       # deja rastro suficiente para depurar
  scalingStrategy:
    strategy: accurate
  triggers:
    - type: aws-sqs-queue
```

```
    metadata:
      queueURL: https://sqs.eu-west-1.amazonaws.com/111122223333/transcodificar
      awsRegion: eu-west-1
      queueLength: "1"
```

La *scalingStrategy* es la parte que hay que entender antes de copiar el YAML. La estrategia *default* resta los Jobs en ejecución a la cola, lo que está bien para trabajos cortos. La estrategia *accurate* está pensada para colas: en lugar de fiarse de la longitud total, tiene en cuenta los mensajes que aún no están bloqueados por ningún consumidor, de forma que no lanza sesenta Jobs para veinte mensajes que ya están siendo procesados. La estrategia *eager* hace lo contrario a propósito: llena todos los huecos disponibles hasta *maxReplicaCount* para vaciar la cola lo antes posible. Es sobreaprovisionamiento consciente, y sólo tiene sentido cuando el coste de la espera supera claramente el coste de la capacidad ociosa.

Dos detalles operativos que se aprenden por las malas. El primero es que *failedJobsHistoryLimit* con un valor bajo borra la evidencia justo cuando la necesitas: los Jobs fallidos son tu registro de errores en este modelo. El segundo es que un *backoffLimit* alto combinado con un mensaje envenenado produce reintentos infinitos que consumen capacidad sin avanzar; el patrón correcto es *backoffLimit* bajo y una cola de mensajes fallidos aguas abajo, no reintentos indefinidos dentro del cluster.

Autenticación: sacar las credenciales del ScaledObject

El primer ScaledObject de casi todo el mundo lleva una clave de acceso en un *Secret* montado en la carga de trabajo, o peor, en el propio manifiesto. Funciona en la demo y es exactamente lo que no quieres tener el día que alguien exporta el namespace a un repositorio.

KEDA separa la autenticación en un recurso propio: *TriggerAuthentication* (por namespace) y *ClusterTriggerAuthentication* (global). El trigger apunta a él por referencia y el manifiesto del ScaledObject deja de contener secretos.

```
# Opcion 1: secretos ya existentes, sin credenciales en el manifiesto.
apiVersion: keda.sh/v1alpha1
kind: TriggerAuthentication
metadata:
  name: rabbit-auth
  namespace: pagos
spec:
  secretTargetRef:
    - parameter: host
      name: rabbitmq-credenciales
      key: connection-string
---
# Opcion 2 (preferible): identidad de carga de trabajo, sin ningun secreto.
apiVersion: keda.sh/v1alpha1
kind: TriggerAuthentication
metadata:
  name: sqs-auth
  namespace: media
spec:
  podIdentity:
    provider: aws                       # EKS Pod Identity / IRSA
    roleArn: arn:aws:iam::111122223333:role/keda-lector-de-colas
---
apiVersion: keda.sh/v1alpha1
kind: ScaledObject
metadata:
  name: image-worker
  namespace: media
spec:
  scaleTargetRef:
    name: image-worker
```

```
  triggers:
    - type: aws-sqs-queue
      authenticationRef:
        name: sqs-auth
      metadata:
        queueURL: https://sqs.eu-west-1.amazonaws.com/111122223333/imagenes
        awsRegion: eu-west-1
        queueLength: "20"
```

La segunda opción es la que hay que perseguir: en AWS con EKS Pod Identity o IRSA, en Azure con Workload Identity y el proveedor *azure-workload*, en GCP con Workload Identity. Desaparece la clave estática, desaparece la rotación y desaparece el riesgo de filtrarla.

Hay un matiz de permisos que conviene decidir pronto. KEDA puede asumir un rol propio del operador para todas las colas del cluster, o cada *TriggerAuthentication* puede apuntar a un rol distinto por aplicación. Lo primero es cómodo y acaba con un rol que puede leer todas las colas de la organización; lo segundo es más manifiestos y mucho menos radio de explosión. Para el permiso que KEDA necesita de verdad --leer atributos de una cola, no consumir mensajes-- la política mínima es corta:

```
{
  "Version": "2012-10-17",
  "Statement": [
    {
      "Effect": "Allow",
      "Action": ["sqs:GetQueueAttributes"],
      "Resource": "arn:aws:sqs:eu-west-1:111122223333:imagenes"
    }
  ]
}
```

Si tu rol de KEDA tiene *sqs:ReceiveMessage* o *sqs:**, alguien copió la política del worker. El autoescalador sólo mira; no consume.

Los scalers que vas a usar de verdad, y sus trampas

KEDA trae más de setenta scalers y en la práctica el 90% de los despliegues usa cuatro. Vale la pena conocer los parámetros que importan de cada uno en lugar de descubrirlos en producción.

Kafka: el lag no es lo que parece

```
triggers:
  - type: kafka
    metadata:
      bootstrapServers: kafka.datos.svc:9092
      consumerGroup: procesador-eventos
      topic: eventos-pedidos
      lagThreshold: "500"
      activationLagThreshold: "50"
      excludePersistentLag: "true"
      allowIdleConsumers: "false"
```

Dos parámetros deciden si esto funciona. *allowIdleConsumers* en *false* impide que KEDA cree más réplicas que particiones tiene el topic, que es la restricción física de Kafka: la réplica número trece de un topic con doce particiones no consume nada y sólo cuesta dinero. Y *excludePersistentLag* resuelve un caso incómodo: una partición cuyo consumidor está bloqueado mantiene un lag que nunca baja, y sin esta opción KEDA escala eternamente intentando arreglar con réplicas un problema que no es de capacidad.

Prometheus: una consulta, un valor

```
triggers:
  - type: prometheus
    metadata:
      serverAddress: http://prometheus.observabilidad.svc:9090
      query: |
        sum(rate(http_requests_in_progress{service="checkout"}[2m]))
      threshold: "40"
      activationThreshold: "5"
      ignoreNullValues: "false"
```

La consulta **debe devolver un único valor escalar**. Si devuelve una serie por pod, el scaler falla o toma un valor arbitrario, y el síntoma es un escalado errático que nadie sabe explicar. Envuelve siempre en *sum()* o *avg()*. Y presta atención a *ignoreNullValues*: en su valor por defecto (*true*), una consulta sin datos se interpreta como cero, lo que hace que un Prometheus caído parezca tráfico cero y tu servicio se apague solo. Ponerlo a *false* convierte ese caso en un error del scaler, que es lo que quieres, porque entonces entra el *fallback*.

Cron: el patrón que ahorra más dinero con menos riesgo

```
triggers:
  - type: cron
    metadata:
      timezone: Europe/Madrid
      start: 0 7 * * 1-5      # lunes a viernes a las 07:00
      end: 0 21 * * 1-5
      desiredReplicas: "12"
  - type: aws-sqs-queue       # sigue reaccionando a picos fuera de horario
    metadata:
      queueURL: https://sqs.eu-west-1.amazonaws.com/111122223333/imagenes
      awsRegion: eu-west-1
      queueLength: "20"
```

Cuando hay varios triggers, KEDA calcula cada uno por separado y **se queda con el mayor**. Eso convierte al trigger de cron en un suelo programado: durante el horario laboral hay doce réplicas calientes cueste lo que cueste, y fuera de él manda la cola. Es la forma más simple de eliminar el arranque en frío exactamente en las horas en que un usuario lo notaría, sin renunciar al escalado a cero de madrugada. Cuidado con la zona horaria: si pones *UTC* y tu negocio es europeo, en verano tus réplicas llegan una hora tarde.

Escalar a cero de verdad: el presupuesto del arranque en frío

Escalar a cero es la promesa más atractiva de KEDA y también la que más incidentes provoca, porque el coste no aparece en la factura sino en la latencia del primer usuario. Antes de activarlo conviene poner números al camino completo desde "llega trabajo" hasta "hay un pod procesándolo":

- **Detección**: hasta un *pollingInterval* completo. Con el valor por defecto de 30 segundos, la media es 15 y el peor caso 30.
- **Planificación**: si hay hueco en un nodo existente, un par de segundos. Si no lo hay, el tiempo que tarde tu autoescalador de nodos.
- **Descarga de la imagen**: cero si está en caché en ese nodo, y entre veinte segundos y varios minutos si no lo está. En la práctica, este suele ser el sumando dominante.
- **Arranque del proceso**: JVM, conexiones al pool, calentamiento de caché, *readinessProbe* con su *initialDelaySeconds*.

Sumado sin optimizar, de cero a útil suelen ser entre cuarenta segundos y tres minutos. Para un worker de cola es irrelevante. Para una API sincrónica es inaceptable, y ahí la respuesta no es "afinar KEDA" sino no escalar a cero: *minReplicaCount: 1* y el ahorro se busca en otro sitio.

Cuando sí quieres escalar a cero, hay tres palancas que funcionan. La primera es reducir el *pollingInterval* a 10 o 15 segundos para los ScaledObjects críticos, asumiendo más llamadas al scaler. La segunda es mantener la imagen caliente en los nodos con un DaemonSet trivial que la descarga:

```
apiVersion: apps/v1
kind: DaemonSet
metadata:
  name: precarga-imagenes
  namespace: media
spec:
  selector:
    matchLabels: { app: precarga-imagenes }
  template:
    metadata:
      labels: { app: precarga-imagenes }
    spec:
      initContainers:
        - name: traer-worker
          image: registry.interno/image-worker:2.4.1   # misma etiqueta que el worker
          command: ["/bin/true"]
      containers:
        - name: pausa
          image: registry.k8s.io/pause:3.9
          resources:
            requests: { cpu: 1m, memory: 8Mi }
```

Cuesta unos milicores por nodo y elimina el sumando más grande del arranque en frío. La tercera palanca es *idleReplicaCount*, que permite tener un estado de reposo distinto de cero; con la advertencia de que, por limitaciones del propio controlador del HPA, el único valor soportado hoy es *0*, así que en la práctica sirve para documentar la intención más que para configurar un uno.

Servicios HTTP y el complemento de KEDA

Para cargas HTTP existe el *HTTP Add-on*, que resuelve el problema de origen: no puedes escalar desde cero por peticiones si no hay nadie que reciba la primera petición. El complemento intercala un *interceptor* que retiene la petición mientras el operador despierta la carga de trabajo, y la reenvía cuando el pod está listo. Puedes configurar una respuesta provisional o un servicio de reserva para el hueco del arranque.

Es una pieza muy útil y hay que adoptarla con los ojos abiertos: el complemento sigue en beta, y el interceptor pasa a estar en el camino crítico de todas las peticiones de ese servicio, con lo que hereda su disponibilidad y su latencia. La primera petición tras un arranque en frío suele añadir entre dos y cinco segundos, y bastante más con imágenes grandes o inicializaciones pesadas. Para un panel interno o un entorno de pruebas es un cambio excelente; para el *checkout*, el cálculo hay que hacerlo con datos.

KEDA no crea nodos: el segundo nivel de autoescalado

Este es el malentendido que más veces convierte una demo perfecta en un incidente. KEDA escala **pods**. Si no hay capacidad en el cluster, esos pods se quedan en *Pending* y tu cola sigue creciendo mientras el panel de KEDA muestra, con toda la razón, que ha pedido cuarenta réplicas.

El autoescalado real tiene dos niveles encadenados y el presupuesto de latencia es la suma de los dos:

```
# El diagnostico de treinta segundos cuando "KEDA no escala":
kubectl get scaledobject -n media          # READY / ACTIVE / FALLBACK
kubectl get hpa -n media                   # TARGETS y REPLICAS
kubectl get pods -n media --field-selector=status.phase=Pending
kubectl describe pod -n media <pod-pendiente> | tail -20   # el evento FailedScheduling lo dice todo
```

Si los pods están en *Pending* con *Insufficient cpu*, el problema no es KEDA: es tu proveedor de nodos. Con Karpenter, un nodo nuevo tarda entre cuarenta y noventa segundos en estar listo; con Cluster Autoscaler y grupos de nodos, más. Ese tiempo se suma al arranque en frío del apartado anterior y es lo que de verdad determina cuánto tarda tu sistema en absorber un pico.

Dos configuraciones que valen su peso en oro cuando combinas KEDA con un autoescalador de nodos. La primera es reservar capacidad con pods de baja prioridad que el planificador desaloja al instante, de forma que siempre haya un nodo con hueco:

```
apiVersion: scheduling.k8s.io/v1
kind: PriorityClass
metadata:
  name: reserva-capacidad
value: -10                        # prioridad negativa: se desaloja el primero
globalDefault: false
description: "Pods de relleno que ceden su sitio a carga real."
---
apiVersion: apps/v1
kind: Deployment
metadata:
  name: reserva
  namespace: capacidad
spec:
  replicas: 3
  selector:
    matchLabels: { app: reserva }
  template:
    metadata:
      labels: { app: reserva }
    spec:
      priorityClassName: reserva-capacidad
      terminationGracePeriodSeconds: 0
      containers:
        - name: pausa
          image: registry.k8s.io/pause:3.9
          resources:
            requests: { cpu: "2", memory: 4Gi }
```

La segunda es proteger a los workers largos de la consolidación de nodos. Karpenter reagrupa la carga para ahorrar dinero, y eso significa mover pods; si tu transcodificación lleva dieciocho minutos, prefieres que no la muevan:

```
metadata:
  annotations:
    karpenter.sh/do-not-disrupt: "true"
```

Con un *PodDisruptionBudget* razonable y esa anotación en los Jobs largos, dejas de perder trabajo por optimizaciones de coste que llegan en mal momento.

Observabilidad: saber qué está haciendo el autoescalador

Un autoescalador sin métricas propias es una caja negra que un día toma una decisión inexplicable. El operador de KEDA expone métricas de Prometheus en el puerto 8080 y hay cuatro que deberían estar en el

panel del equipo desde el primer día.

- *keda_scaler_metrics_value*: el valor que el scaler está entregando al HPA. Comparado con tu propia métrica de negocio, revela desfases y consultas mal escritas.
- *keda_scaler_active*: 1 o 0. Es la respuesta directa a "¿por qué sigue en cero?".
- *keda_scaler_detail_errors_total* y *keda_scaled_object_errors_total*: errores por scaler y por ScaledObject. Un scaler que falla y nadie mira es un escalado que no ocurre.
- *keda_scaler_metrics_latency_seconds*: cuánto tarda en obtener la métrica. Si sube, tu *pollingInterval* deja de cumplirse y el autoescalado se vuelve lento sin que nada dé error.

```
groups:
  - name: keda
    rules:
      - alert: KedaScalerConErrores
        expr: |
          sum by (scaledObject, scaler) (
            rate(keda_scaler_detail_errors_total[10m])
          ) > 0
        for: 10m
        labels: { severity: warning }
        annotations:
          summary: "El scaler {{ $labels.scaler }} falla en {{ $labels.scaledObject }}"
          description: "El autoescalado esta ciego o funcionando por fallback."

      - alert: KedaEnFallback
        expr: keda_scaled_object_paused == 1
        for: 15m
        labels: { severity: warning }
        annotations:
          summary: "ScaledObject {{ $labels.scaledObject }} pausado hace mas de 15 minutos"

      - alert: PodsPendientesConColaCreciendo
        expr: |
          kube_pod_status_phase{phase="Pending", namespace="media"} > 0
          and on() (keda_scaler_metrics_value{scaledObject="image-worker"} > 500)
        for: 5m
        labels: { severity: critical }
        annotations:
          summary: "KEDA pide replicas pero el cluster no tiene sitio"
```

Esa última alerta es la que distingue "el autoescalador no funciona" de "no hay nodos", y es exactamente la pregunta que te vas a hacer a las tres de la mañana.

Operar KEDA: pausar, migrar y convivir con GitOps

Durante un incidente, lo primero que quieres a veces es que el autoescalador deje de opinar. KEDA lo permite con anotaciones, sin borrar nada y sin perder la configuración:

```
# Congelar en un numero concreto de replicas (el HPA deja de actuar):
kubectl annotate scaledobject image-worker -n media \
  autoscaling.keda.sh/paused-replicas="25" --overwrite

# Pausar sin fijar numero: se queda como esta.
kubectl annotate scaledobject image-worker -n media \
  autoscaling.keda.sh/paused="true" --overwrite

# Devolver el control a KEDA:
kubectl annotate scaledobject image-worker -n media \
  autoscaling.keda.sh/paused-replicas- autoscaling.keda.sh/paused-
```

Esto es infinitamente mejor que borrar el ScaledObject a mano, que es lo que suele hacerse con prisas y que además tiene un efecto poco conocido: al eliminar un ScaledObject, KEDA borra el HPA que gestionaba y restaura el número de réplicas que el Deployment tenía antes. Si eso ocurre en mitad de un pico, acabas de reducir tu capacidad justo cuando más la necesitabas.

Migrar desde un HPA existente

La transición desde un HPA de CPU tiene un orden obligatorio, porque dos autoescaladores sobre el mismo Deployment se pelean y el resultado es un pod que aparece y desaparece cada pocos segundos.

1. Crea el ScaledObject en modo observación: *minReplicaCount* y *maxReplicaCount* iguales al HPA actual, y añade la anotación de pausa desde el principio.
2. Compara durante un ciclo de tráfico completo *keda_scaler_metrics_value* con lo que el HPA de CPU está decidiendo. Si KEDA hubiera escalado en momentos muy distintos, tu umbral está mal.
3. **Borra el HPA antiguo.** No lo dejes "por si acaso": el webhook de KEDA lo rechazará o, si el webhook no está, tendrás dos controladores escribiendo el mismo campo.
4. Quita la anotación de pausa y vigila las primeras horas.

Y una nota sobre GitOps que evita un problema recurrente: Argo CD verá el HPA creado por KEDA como un recurso que no está en tu repositorio, y además verá el campo *replicas* del Deployment cambiando constantemente. Sin configuración, eso es o bien ruido permanente de *OutOfSync* o bien Argo devolviendo las réplicas a lo que dice Git cada pocos minutos, peleándose literalmente con el autoescalador.

```
# En la Application de Argo CD
spec:
  syncPolicy:
    syncOptions:
      - RespectIgnoreDifferences=true
  ignoreDifferences:
    - group: apps
      kind: Deployment
      jsonPointers:
        - /spec/replicas          # el numero lo decide el autoescalador, no Git
```

La regla general: el manifiesto del Deployment en Git no debe declarar *replicas* en absoluto cuando hay un ScaledObject encima. Lo que va a Git es la política de escalado; el número concreto es estado de ejecución.

Caso práctico: de una cola que crecía toda la tarde a un coste un 61% menor

Un servicio de procesamiento de documentos con un worker de SQS, 24 réplicas fijas, todo el día y toda la noche. El patrón de carga: prácticamente nada entre las 22:00 y las 07:00, un pico fuerte de 09:00 a 11:00 cuando las gestorías cargan sus lotes, y una meseta el resto de la jornada. Con réplicas fijas, la noche pagaba capacidad ociosa y el pico de la mañana generaba una cola que tardaba hora y media en drenarse.

Medición previa: 24 pods de 2 vCPU y 4 GiB, 720 horas al mes, un p95 de espera en cola de 41 minutos durante el pico, y cero incidentes de latencia el resto del día.

Lo que se cambió, en este orden:

1. **ScaledObject de SQS** con *queueLength: "30"*, *minReplicaCount: 2*, *maxReplicaCount: 90*. El mínimo de dos, no cero, porque el trabajo llega en ráfagas pequeñas y constantes durante el día y no compensaba el arranque en frío.
2. **Trigger de cron adicional** con un suelo de 10 réplicas entre las 08:45 y las 11:30 en días laborables.

El pico es predecible: no tiene sentido descubrirlo cada mañana con una cola de 4.000 mensajes.

3. ***behavior* asimétrico**: subida sin ventana de estabilización y hasta el 100% cada 30 segundos; bajada con 600 segundos de ventana y un 20% por minuto.
4. **DaemonSet de precarga** de la imagen del worker, que pesaba 1,3 GiB, en el grupo de nodos correspondiente.
5. **Karpenter** con un *NodePool* de instancias spot para este namespace y *do-not-disrupt* en los pods que procesan documentos de más de cien páginas.

Resultado tras tres semanas: media de 9,2 réplicas frente a 24 fijas, p95 de espera en cola durante el pico de 41 a 6 minutos (porque el suelo programado más el escalado agresivo absorben la ráfaga en lugar de digerirla), y la factura de cómputo de ese namespace un 61% más baja. El arranque en frío dejó de ser un problema al eliminar la descarga de imagen: de 95 segundos de mediana a 22.

Lo que no salió bien a la primera: el primer intento usaba *minReplicaCount: 0* y el equipo de soporte reportó que los documentos sueltos que llegaban por la tarde tardaban minutos en procesarse. El *activationQueueLength* estaba en 1, así que cada documento despertaba el sistema entero. Subirlo a 5 y poner el mínimo en 2 resolvió el síntoma y costó menos que la mitad del ahorro conseguido.

Cuánto cuesta esto de verdad, y cuándo KEDA no es la respuesta

El argumento comercial del autoescalado siempre es el ahorro, y suele ser cierto, pero conviene hacer la cuenta completa en lugar de la que sale bien. Hay tres costes que no aparecen en la comparación ingenua entre "24 réplicas fijas" y "media de 9".

El primero es el **coste de las consultas de métricas**. Cada ScaledObject pregunta a su fuente cada *pollingInterval*. Con CloudWatch o con llamadas a la API de SQS eso se factura; con Prometheus no se factura pero se paga en carga de consulta, y una consulta pesada ejecutada cada diez segundos por sesenta ScaledObjects es una forma eficaz de tumbar tu propio Prometheus. La aritmética es simple y conviene hacerla: un ScaledObject con *pollingInterval: 10* son 8.640 consultas al día; cincuenta ScaledObjects son 432.000.

El segundo es el **coste del churn**. Cada arranque descarga imagen, abre conexiones al pool de base de datos, calienta cachés y consume ancho de banda. Un servicio que oscila entre 8 y 30 réplicas cada pocos minutos puede gastar más en arrancar de lo que ahorra en capacidad ociosa, además de castigar a la base de datos con un patrón de conexiones que ningún pooler agradece. Si tus métricas muestran más de una decena de cambios de escala por hora, el problema no se arregla subiendo el máximo: se arregla ampliando la ventana de estabilización de bajada.

El tercero es el **coste operativo**: un componente más que actualizar, un CRD más que revisar en cada actualización del cluster y una causa raíz más que descartar en cada incidente.

Con eso sobre la mesa, hay tres situaciones en las que la respuesta honesta es no usar KEDA:

- **Carga plana y predecible.** Si tu servicio hace lo mismo a las cuatro de la mañana que a mediodía, el autoescalado añade riesgo sin ahorrar nada. Un número fijo bien elegido es más barato y más aburrido, que en producción es una virtud.
- **Cuando el cuello de botella está aguas abajo.** Si tus workers esperan por una base de datos saturada, añadir réplicas empeora la situación: más conexiones, más contención, más lentitud, y la métrica que dispara el escalado sigue subiendo. Aquí el autoescalado convierte un problema de rendimiento en una caída.
- **Cuando ya tienes *prometheus-adapter* y no puedes quitarlo.** Por la restricción del proveedor único de métricas externas que vimos antes, esa decisión hay que tomarla a nivel de cluster, no a nivel de equipo.

Y una comprobación previa que ahorra discusiones: antes de instalar nada, mide durante una semana el número de réplicas que *habrías* necesitado en cada momento y compáralo con el que tienes. Si la diferencia entre el máximo y el mínimo es menor que un factor de dos, el ahorro no va a pagar la complejidad.

Cien ScaledObjects: lo que cambia cuando esto deja de ser un servicio

Todo lo anterior funciona igual con uno o con cien ScaledObjects, salvo tres cosas que sólo se ven a escala.

El operador es un único punto de coordinación. Corre en una sola réplica activa (las demás quedan en espera mediante elección de líder) y evalúa los triggers de forma secuencial dentro de su bucle. Con muchos ScaledObjects y scalers lentos, el intervalo real de sondeo se aleja del configurado. La métrica *keda_scaler_metrics_latency_seconds* es la que avisa, y la solución suele ser subir el *pollingInterval* de lo que no es crítico en lugar de bajarlo en todo.

Los umbrales no se copian entre equipos. Un *queueLength* de 20 es correcto para un worker que tarda 200 ms por mensaje y absurdo para uno que tarda 40 segundos. Cuando la plantilla se propaga por copiar y pegar, acabas con veinte servicios mal ajustados y nadie sabe por qué. Documenta el umbral con el razonamiento al lado, no sólo el número:

```
metadata:
  annotations:
    # 20 mensajes/replica: cada mensaje tarda ~1,5 s, objetivo de drenar
    # la cola en menos de 30 s con la replica ya caliente. Revisado 2026-09.
    escalado.interno/justificacion-umbral: "20 = 30s objetivo / 1,5s por mensaje"
```

La gobernanza se vuelve necesaria. Un *maxReplicaCount* sin techo en un namespace sin *ResourceQuota* es un incidente esperando a que alguien publique un bucle en una cola. La combinación que funciona es sencilla: cuota de recursos por namespace que actúa como límite duro, *maxReplicaCount* por debajo de esa cuota como límite blando, y una política de admisión que rechace ScaledObjects sin máximo declarado.

Los cinco errores que verás en producción

1. GitOps peleándose con el autoescalador

Si tu manifiesto de Deployment declara `replicas: 3` y Argo CD sincroniza cada tres minutos, tienes un bucle garantizado: KEDA escala a 12, Argo lo devuelve a 3, el HPA vuelve a subir. Quita `replicas` del manifiesto (Kubernetes conserva el valor actual al aplicar si el campo no está presente) o excluye el campo de forma explícita:

```
# Application de Argo CD
spec:
  ignoreDifferences:
  - group: apps
    kind: Deployment
    jsonPointers:
    - /spec/replicas
```

2. Dos autoescaladores sobre el mismo Deployment

Un HPA escrito a mano y un ScaledObject apuntando al mismo target se pisan, y el resultado es impredecible. KEDA lo detecta con su admission webhook y rechaza el ScaledObject. Si lo que quieres es migrar un HPA existente, dile a KEDA que adopte la propiedad:

```
metadata:
  annotations:
    scaledobject.keda.sh/transfer-hpa-ownership: "true"
```

3. Mensajes perdidos al escalar hacia abajo

Escalar a cero sin un apagado limpio significa mensajes a medio procesar. El worker debe capturar `SIGTERM`, dejar de pedir trabajo nuevo y terminar el que tiene en curso dentro de `terminationGracePeriodSeconds`:

```
import signal

shutting_down = False

def handle_sigterm(signum, frame):
    global shutting_down
    shutting_down = True      # no abortamos: dejamos de pedir trabajo nuevo

signal.signal(signal.SIGTERM, handle_sigterm)

def run():
    while not shutting_down:
        response = sqs.receive_message(
            QueueUrl=QUEUE_URL,
            MaxNumberOfMessages=10,
            WaitTimeSeconds=20,        # long polling
        )
        for message in response.get("Messages", []):
            process(message)           # debe ser idempotente
            sqs.delete_message(
                QueueUrl=QUEUE_URL,
                ReceiptHandle=message["ReceiptHandle"],
            )
    log.info("SIGTERM recibido, lote terminado, salgo limpiamente")
```

Y en el Deployment, un margen mayor que tu tarea más lenta:

```
spec:
  template:
    spec:
      terminationGracePeriodSeconds: 120
```

Un detalle propio del escalado a cero: con *long polling* de 20 segundos, el worker puede tardar ese tiempo en enterarse de que debe morir, porque está bloqueado dentro de `receive_message`. Súmalo al presupuesto de gracia.

4. Confundir "sin réplicas" con "sin nodos"

KEDA escala pods; no crea nodos. Si el cluster está lleno, tus pods se quedan en `Pending` hasta que el cluster autoscaler o Karpenter provisione capacidad, y eso puede duplicar tu arranque en frío. Mide las dos cosas por separado. Si el tiempo de arranque es crítico, reserva capacidad con pods de relleno de prioridad negativa que el scheduler pueda expulsar al instante:

```
apiVersion: scheduling.k8s.io/v1
kind: PriorityClass
metadata:
  name: overprovisioning
value: -10
globalDefault: false
description: "Pods de relleno, expulsables al instante"
```

5. Escalar por una métrica que el propio escalado no mejora

Si tus workers están bloqueados esperando un pool de 20 conexiones a Postgres, añadir pods no reduce el backlog: aumenta la contención y empeora la latencia de todos. Antes de autoescalar cualquier cosa, comprueba que el cuello de botella escala horizontalmente. Es el fallo de diseño más caro de esta lista, y no lo arregla ningún YAML.

Cómo validarlo antes de confiar

Un experimento de veinte minutos que te ahorra un incidente:

```
# 1. Confirma el estado inicial
kubectl get deploy image-worker -n media -o jsonpath="{.spec.replicas}"

# 2. Inyecta 500 mensajes y cronometra el arranque en frío
python scripts/seed_queue.py --count 500
kubectl get pods -n media -l app=image-worker -w

# 3. Mira lo que ve el HPA: valor actual frente a objetivo
kubectl describe hpa keda-hpa-image-worker -n media

# 4. Consulta la métrica externa cruda que sirve KEDA
kubectl get --raw "/apis/external.metrics.k8s.io/v1beta1/namespaces/media/s0-aws-sqs-image-jobs" | jq
```

Anota tres números: segundos desde el primer mensaje hasta el primer pod Ready, segundos hasta drenar la cola y réplicas máximas alcanzadas. Repite la medición tras cada cambio de `threshold` o de `behavior`. Sin esa línea base, ajustar el autoescalado es adivinar en YAML.

Checklist antes de dar por bueno un ScaledObject

- La métrica del trigger es la que de verdad limita tu throughput, y el sistema escala horizontalmente.
- `threshold` está expresado por réplica, y lo has comprobado con la fórmula del HPA.
- `activationThreshold` configurado si escalas a cero; `minReplicaCount: 1` en todo lo que tenga SLO de latencia.
- `fallback` definido con `currentReplicasIfHigher` para triggers externos.
- `behavior.scaleUp.periodSeconds` del orden del tiempo real de arranque del pod.
- Apagado limpio: SIGTERM manejado, `terminationGracePeriodSeconds` mayor que la tarea más larga, procesamiento idempotente.
- `spec.replicas` fuera del manifiesto o ignorado en GitOps.
- Alertas sobre pods en `Pending` y sobre ScaledObjects en estado `Fallback`.
- Coste medido antes y después. Si no bajó, el escalado a cero no era tu problema.

El autoescalado dirigido por eventos no es magia: es elegir bien la señal, entender quién manda en cada tramo y aceptar que el arranque en frío es un coste que se presupuesta, no un bug que se elimina. Empieza por el worker más caro y ocioso que tengas, mide, y decide con los números delante.

Servicios sincrónicos: escala por concurrencia, no por peticiones por segundo

Casi todo lo anterior habla de colas porque es donde KEDA brilla, pero cada vez más equipos lo usan para APIs. Ahí hay una decisión que se toma mal por defecto: escalar por peticiones por segundo. Es la métrica más fácil de obtener y la peor de las razonables.

El problema es que las peticiones por segundo no dicen nada sobre cuánto trabajo hay dentro de tu proceso. Mil peticiones por segundo de 5 ms y cien peticiones por segundo de 900 ms cargan el servicio de forma parecida, pero un umbral basado en RPS trata el segundo caso como un décimo del primero. El día que una dependencia se pone lenta, tus peticiones por segundo *bajan* --porque cada una tarda más-- y el autoescalador reduce réplicas justo cuando hacía falta lo contrario.

La métrica correcta es la **concurrencia**: peticiones en vuelo. Es lo que la ley de Little relaciona directamente con la capacidad, y absorbe los cambios de latencia sin que tengas que tocar nada.

```
triggers:
  - type: prometheus
    metadata:
      serverAddress: http://prometheus.observabilidad.svc:9090
      query: |
        sum(http_requests_in_flight{service="checkout"})
      threshold: "25"          # peticiones simultaneas por replica
      activationThreshold: "3"
      ignoreNullValues: "false"
```

Para elegir el umbral no hace falta intuición: mide. Lanza carga contra *una* réplica aislada, sube la concurrencia poco a poco y anota el punto en el que el percentil 95 empieza a empeorar de forma marcada. Ese es el codo de la curva, y tu umbral debería estar entre el 60% y el 70% de ese valor, para dejar margen mientras arrancan réplicas nuevas.

```
# Encontrar el codo con una sola replica y k6, subiendo concurrencia por escalones
k6 run --vus-max 200 \
  --stage 1m:10 --stage 1m:25 --stage 1m:50 --stage 1m:100 --stage 1m:200 \
  carga.js
```

Y una comprobación de coherencia que evita el bucle más caro: el umbral de concurrencia por réplica no puede superar lo que tu propio servidor admite. Si corres FastAPI con cuatro workers de Uvicorn y un pool de diez conexiones a base de datos, un objetivo de 25 peticiones simultáneas por pod significa que estás encolando dentro del proceso, y añadir réplicas multiplicará las conexiones contra una base de datos que ya es el cuello de botella. El autoescalado no arregla un límite que está aguas abajo; lo hace visible antes.

Preguntas frecuentes

¿KEDA sustituye al HPA? No. Lo genera y lo configura. El HPA sigue haciendo la aritmética entre 1 y N; KEDA le da métricas que el HPA no sabría obtener por su cuenta y añade la transición entre 0 y 1.

¿Puedo tener varios triggers en un mismo ScaledObject? Sí, y es lo habitual. KEDA calcula cada uno y se queda con el número de réplicas mayor. Es un OR, no un AND: basta con que un trigger pida capacidad para que la haya.

¿Qué pasa si mi broker de mensajes se cae? El scaler entra en error. Sin *fallback*, el HPA se queda con la última decisión conocida y deja de reaccionar; con *fallback*, KEDA sirve un número de réplicas fijo que tú defines. Configúralo siempre, y alerta sobre él: un servicio en fallback está funcionando a ciegas.

¿Sirve para StatefulSets? Sí, y para cualquier recurso personalizado que implemente el subrecurso *scale*. Cambia el *scaleTargetRef* y añade *kind*.

¿Cuánto consume KEDA? El operador y el servidor de métricas son ligeros, del orden de decenas de milicores y unos cientos de megabytes en un cluster mediano. Lo que sí crece es el número de llamadas a tus proveedores de métricas: cien ScaledObjects con un *pollingInterval* de 10 segundos son 36.000 consultas por hora, y eso se nota en la factura de CloudWatch o en la carga de tu Prometheus.

¿Puedo escalar por el coste o por la profundidad de una cola de otro equipo? Técnicamente sí, y casi siempre es mala idea. Escala por una métrica que tu propio escalado pueda mejorar. Si añadir réplicas no reduce la métrica, has construido un bucle que sólo sabe crecer.

Glosario

- **ScaledObject**: recurso de KEDA que describe cómo escalar un Deployment, StatefulSet o recurso escalable entre un mínimo y un máximo a partir de uno o varios triggers.
- **ScaledJob**: equivalente para cargas efímeras, donde cada unidad de trabajo es un Job de Kubernetes en lugar de una réplica de un Deployment.
- **Scaler**: el conector concreto que sabe consultar una fuente (SQS, Kafka, Prometheus, cron y setenta más) y devolver un número.
- **Trigger**: la instancia configurada de un scaler dentro de un ScaledObject, con sus metadatos y su umbral.
- **threshold**: valor objetivo por réplica que se pasa al HPA. Gobierna la escala entre 1 y N.
- **activationThreshold**: valor mínimo de la métrica para que el scaler se considere activo. Gobierna sólo la transición entre 0 y 1 y se ignora si el mínimo de réplicas es 1 o más.
- **cooldownPeriod**: tiempo que un scaler debe permanecer inactivo antes de bajar de 1 a 0. No afecta a ninguna otra bajada.
- **pollingInterval**: cada cuánto consulta KEDA al scaler. Define el peor caso de la latencia de detección.
- **fallback**: número de réplicas al que recurrir cuando el scaler está en error.
- **TriggerAuthentication**: recurso que separa las credenciales del ScaledObject, idealmente apuntando a una identidad de carga de trabajo en lugar de a un secreto.
- **Métricas externas**: la API de Kubernetes (*external.metrics.k8s.io*) por la que el HPA obtiene valores que no vienen del cluster. Sólo admite un proveedor por cluster.

English version

The problem: CPU has no idea how deep your queue is

A worker consuming messages from a queue spends most of its life waiting on I/O: it pulls a batch, calls an external API, writes to Postgres, and waits again. Average CPU hovers around 10%. Autoscale it against `cpu: 70%` and it will never scale out, no matter how many messages pile up behind it. And when it finally does scale -- because one spike pinned a core -- you are already twenty minutes behind, and nobody is getting that time back.

The signal that matters is not CPU, it is *backlog*. How many messages are unprocessed, how long the oldest one has been waiting, how many requests per second are arriving. KEDA exists to wire those signals into Kubernetes autoscaling and, along the way, to do something the HPA cannot do on its own: scale down to zero replicas.

How the HPA actually works

Before adding a new moving part, understand the one you already have. The HorizontalPodAutoscaler lives inside `kube-controller-manager` and re-evaluates each target every 15 seconds by default (`--horizontal-pod-autoscaler-sync-period`). On every cycle it applies a formula that fits on one line:

```
desiredReplicas = ceil[ currentReplicas × ( currentMetricValue / desiredMetricValue ) ]
```

With two caveats that explain 90% of the seemingly strange behaviour:

- **Tolerance.** If the ratio falls within ±10% (`--horizontal-pod-autoscaler-tolerance`, 0.1 by default), the HPA does nothing. As of Kubernetes 1.34 you can set a separate `tolerance` per `scaleUp` and `scaleDown` policy; in 1.33 it was alpha behind the `HPAConfigurableTolerance` feature gate.
- **Only ready pods count.** Pods that have not passed their readiness probe, or whose metrics are missing, are excluded from the calculation. That is what stops a slow startup from being read as missing capacity and triggering a replica avalanche.

The `behavior` field controls speed, and its defaults are deliberately asymmetric:

```
behavior:
  scaleUp:
    stabilizationWindowSeconds: 0
    selectPolicy: Max
    policies:
    - type: Percent
      value: 100
      periodSeconds: 15
    - type: Pods
      value: 4
      periodSeconds: 15
  scaleDown:
    stabilizationWindowSeconds: 300
    policies:
    - type: Percent
      value: 100
      periodSeconds: 15
```

Translated: scale up fast (it can double the replicas or add 4 pods every 15s, whichever is larger) and scale down slowly (look back over the last 5 minutes and keep the highest recommendation in that window). That `stabilizationWindowSeconds: 300` is why your deployment takes five minutes to shrink after a spike, and it is a healthy default: it prevents flapping.

What the HPA will never do is go below `minReplicas: 1`. That is not a configuration choice, it is the controller.

What KEDA adds, precisely

KEDA (Kubernetes Event-Driven Autoscaling, a graduated CNCF project) is essentially two components:

- The **operator**, which watches your `ScaledObject` resources and owns the 0 1 range. When a trigger reports activity while you are at zero replicas, the operator patches the Deployment's `/scale` subresource directly.
- The **metrics adapter**, which implements `external.metrics.k8s.io` and feeds the HPA your trigger values: queue length, the result of a Prometheus query, Kafka consumer group lag, whatever you need.

That split is the key to debugging: **KEDA owns 0 to 1; the HPA owns 1 to N**. This is why `pollingInterval` only affects starting from zero, and `cooldownPeriod` only affects shutting down to zero.

```
helm repo add kedacore https://kedacore.github.io/charts
helm repo update
helm install keda kedacore/keda --namespace keda --create-namespace

# The adapter must report Available=True
kubectl get apiservice v1beta1.external.metrics.k8s.io
```

Your first ScaledObject

A worker processing images off SQS. Not one line more than needed:

```
apiVersion: keda.sh/v1alpha1
kind: ScaledObject
metadata:
  name: image-worker
  namespace: media
spec:
  scaleTargetRef:
    name: image-worker           # Deployment, same namespace
  pollingInterval: 15            # how often KEDA polls while at 0
  cooldownPeriod: 120            # wait after last activity before going back to 0
  minReplicaCount: 0
  maxReplicaCount: 40
  triggers:
  - type: aws-sqs-queue
    metricType: AverageValue
    metadata:
      queueURL: https://sqs.eu-west-1.amazonaws.com/123456789012/image-jobs
      queueLength: "20"            # target: 20 messages per replica
      activationQueueLength: "1"   # a single message wakes it from zero
      awsRegion: eu-west-1
    authenticationRef:
      name: keda-aws-irsa
```

Two concepts people mix up constantly:

- `queueLength: "20"` is a **per-replica** target, not a total. With `metricType: AverageValue` -- the default for nearly every scaler -- the HPA divides the backlog by the number of ready replicas. 200 queued messages converge on 10 pods.
- `activationQueueLength` is the **0 -> 1** threshold, and KEDA evaluates it, not the HPA. They are two different thresholds, which is exactly why there are two fields. If you set `minReplicaCount: 1`, the activation threshold is ignored entirely.

Apply it and watch. KEDA will have created an HPA named `keda-hpa-image-worker`:

```
kubectl get scaledobject image-worker -n media
kubectl get hpa keda-hpa-image-worker -n media -w
kubectl describe scaledobject image-worker -n media    # conditions: Ready, Active, Fallback
kubectl logs -n keda -l app=keda-operator -f
```

The `Active` condition answers the most common question of all: "why am I still at zero?". If it reads `False`, no trigger is above its activation threshold.

Scaling to zero without punishing the user

Going to zero saves real money: on intermittent workers it is common to cut 60% to 90% of idle spend. But the first message of every burst pays the cold start, and that cold start is not a single number, it is a sum:

1. **Detection**: up to `pollingInterval` seconds before KEDA even sees the message.
2. **Scheduling**: seconds if a node has room; 1-3 minutes if the cluster autoscaler or Karpenter has to bring one up.
3. **Image pull**: 0s with a warm node cache, tens of seconds for a 2 GB image on a brand new node.
4. **Application startup**: JVM warm-up, connection pool, a model being loaded into memory.

Rules that hold up in production:

- Scale **workers and jobs** to zero, not APIs with a latency SLO. For an HTTP API use `minReplicaCount: 1`, or 2 if you care about staying available during a rolling update.
- Drop `pollingInterval` to 5-15s on queues where startup time matters. It is one broker API call every N seconds, per ScaledObject: cheap, but not free once you have hundreds of them.
- Slim the image down. An 80 MB distroless worker cold-starts in a fraction of the time a 1.5 GB one takes.
- If you have a known daily pattern, combine a `cron` trigger with the queue trigger: keep a warm floor during business hours and let it drop to zero overnight.

```
  triggers:
  - type: cron
    metadata:
      timezone: Europe/Madrid
      start: 0 8 * * 1-5        # Monday to Friday, 08:00
      end: 0 20 * * 1-5         # until 20:00
      desiredReplicas: "3"
  - type: aws-sqs-queue
    metadata:
      queueURL: https://sqs.eu-west-1.amazonaws.com/123456789012/image-jobs
      queueLength: "20"
      awsRegion: eu-west-1
```

With multiple triggers the HPA computes a recommendation per metric and keeps the **largest** one. The cron trigger therefore acts as a floor inside its window, and the queue drives everything outside it.

Scaling on business metrics with Prometheus

Queue scalers cover a lot of ground, but very often the right signal is already in Prometheus: backlog depth, oldest-message age, in-flight requests. The `prometheus` scaler turns any query into an autoscaling metric.

```
  triggers:
  - type: prometheus
    name: backlog
    metricType: AverageValue
    metadata:
      serverAddress: http://prometheus-operated.monitoring.svc:9090
      query: sum(rabbitmq_queue_messages_ready{queue="invoices"})
      threshold: "50"             # 50 pending messages per replica
      activationThreshold: "5"    # below 5, stay at zero
      ignoreNullValues: "false"   # empty query = error, not zero
```

That last line is a deliberate choice. By default `ignoreNullValues` is `true`, and a query returning no series is read as 0 -- so your deployment shuts itself down exactly when your metrics stack breaks. With `false`, missing data counts as a scaler failure and triggers the `fallback` covered below.

When you need to combine signals -- scale on backlog *but also* if p95 latency blows up -- use `scalingModifiers`:

```
spec:
  advanced:
    scalingModifiers:
      target: "1"
      metricType: AverageValue
      formula: "max(backlog / 50, p95_latency / 0.3)"
  triggers:
  - type: prometheus
    name: backlog
    metadata:
      serverAddress: http://prometheus-operated.monitoring.svc:9090
      query: sum(rabbitmq_queue_messages_ready{queue="invoices"})
      threshold: "50"
  - type: prometheus
    name: p95_latency
    metadata:
      serverAddress: http://prometheus-operated.monitoring.svc:9090
      query: histogram_quantile(0.95, sum by (le) (rate(job_duration_seconds_bucket[5m])))
      threshold: "0.3"
```

The formula is evaluated with the *expr* language and must return a number, not a boolean. The trick is to normalise every signal into "replica units" by dividing it by its own target and then keep the most demanding one; the modifier's `target: "1"` becomes the final HPA denominator. Every trigger referenced in the formula must have a `name` field.

Making sure a broken scaler does not take the service down

If Prometheus goes down, the scaler fails and the HPA is left without a metric. With no extra configuration the HPA *freezes* the current replica count, which is disastrous if the incident started with a traffic spike in the first place. The `fallback` section defines your plan B:

```
spec:
  fallback:
    failureThreshold: 3                 # 3 consecutive scaler failures
    replicas: 6                         # ...then park at 6 replicas
    behavior: currentReplicasIfHigher   # or keep the current count if it is higher
```

`currentReplicasIfHigher` is the safe default: it never removes capacity during an observability outage. Two limits worth knowing: `fallback` does not work with CPU or memory triggers, and it is not supported on `ScaledJob`.

Tuning the speed from the ScaledObject itself

```
spec:
  advanced:
    restoreToOriginalReplicaCount: true
    horizontalPodAutoscalerConfig:
      behavior:
        scaleUp:
          stabilizationWindowSeconds: 0
          selectPolicy: Max
          policies:
          - type: Percent
            value: 200
            periodSeconds: 30       # can triple every 30s
          - type: Pods
            value: 10
            periodSeconds: 30
        scaleDown:
          stabilizationWindowSeconds: 600
          policies:
          - type: Pods
            value: 2
            periodSeconds: 60       # at most -2 pods per minute
```

How to pick those numbers without guessing: measure how long one of your pods takes to be *Ready and producing*, not just Ready. If that is 90 seconds, a `periodSeconds` of 15 lets the HPA add pods six times before the first one does any work, and you will overshoot every single time. Set `periodSeconds` to roughly your real startup time.

For `scaleDown`, the question is what a mistake costs. With short idempotent jobs, shrinking fast is cheap. With ten-minute jobs that would have to restart from scratch, a 600s window and a ceiling of 2 pods per minute is far cheaper than the retries.

`restoreToOriginalReplicaCount: true` deserves its own note: if you delete the ScaledObject, the default behaviour leaves the Deployment at whatever replica count it happened to have at that moment -- including 0. More than one service that "disappeared" after a `kubectl delete scaledobject` is explained by exactly this.

Inside KEDA: two processes, one Kubernetes API and an HPA you never wrote

Before tuning a single parameter it helps to have the right mental model, because almost every strange KEDA problem is explained by working out which component is failing. When you install the chart, two distinct workloads appear with non-overlapping responsibilities.

The **operator** (*keda-operator*) watches *ScaledObject* and *ScaledJob* resources. Its job is to translate your declaration into native objects: for every ScaledObject it creates and maintains a *HorizontalPodAutoscaler* you never wrote, named *keda-hpa-your-scaledobject-name*. It is also what decides the transitions between zero and one, which the HPA cannot do.

The **metrics server** (*keda-operator-metrics-apiserver*) implements the Kubernetes external metrics API and registers itself as an *APIService* for *external.metrics.k8s.io/v1beta1*. When the HPA needs to know how many messages are in the queue, it does not talk to SQS: it asks the Kubernetes API, which routes the question to this server, which in turn queries the relevant scaler.

```
kubectl get pods -n keda
kubectl get apiservice v1beta1.external.metrics.k8s.io
kubectl get hpa -A | grep keda-hpa
```

```
# See the exact metric the HPA is reading, exactly as KEDA serves it:
kubectl get --raw "/apis/external.metrics.k8s.io/v1beta1/namespaces/media/s0-aws-sqs-queue?labelSelect
or=scaledobject.keda.sh/name=image-worker" | jq
```

That last command is the most useful KEDA diagnostic there is and almost nobody uses it. It tells you whether the problem is in the scaler (returns no value) or in the HPA (gets the value and does nothing), which are two completely different incidents with two completely different fixes.

The conflict that breaks existing clusters

There is a Kubernetes constraint worth knowing before you install anything: **there can be only one registered provider for *external.metrics.k8s.io* in the entire cluster**. It is a cluster-scoped *APIService*, not a namespaced one.

If your cluster already runs *prometheus-adapter* serving external metrics and you deploy KEDA on top, KEDA overwrites the registration and the previous adapter stops answering. The HPAs that depended on it do not throw an obvious error: their metric goes to *unknown* and they stop scaling. It is a silent failure that can take days to notice, and it usually gets noticed when the traffic peak arrives.

The way out is to pick a single provider. If you need Prometheus metrics, do not install both: use KEDA's *prometheus* scaler, which runs the same query with no adapter needed. If for some reason you must keep *prometheus-adapter*, then KEDA cannot live in that cluster, and that is the conversation to have before opening the pull request, not after.

The admission webhook

The chart also installs a *ValidatingWebhookConfiguration* that rejects incoherent ScaledObjects at apply time: two ScaledObjects pointing at the same *scaleTargetRef*, a hand-written HPA already managing that Deployment, a *minReplicaCount* greater than *maxReplicaCount*. It is an excellent safety net and also a source of surprises in CI if your pipeline applies manifests with *--validate=false*, or if the webhook is unavailable: in that case the resources go in unchecked and the error surfaces at runtime.

threshold and activationThreshold: the two numbers almost everyone confuses

If you take one idea away from this guide, make it this one. A KEDA trigger has two thresholds, they mean different things, they govern different transitions, and different components enforce them.

- ***threshold*** (or the scaler's equivalent parameter: *queueLength*, *lagThreshold*, *threshold*...) is the per-replica target handed to the HPA. It is the denominator in the HPA formula. It governs scaling between 1 and N.
- ***activationThreshold*** is the minimum metric value for KEDA to consider the scaler *active*. It governs only the transition between 0 and 1, and the operator evaluates it, not the HPA.

Two consequences follow, and they explain most of the "weird" behaviour. First: if your *minReplicaCount* is 1 or more, the scaler is always active and **the *activationThreshold* is ignored entirely**. Setting it there does nothing, and the team spends an afternoon tuning a number nobody reads. Second: *activationThreshold* is not a percentage or a fraction of *threshold*, it is an absolute metric value, and if you leave it at its default of zero, a single stray message wakes the service up.

```
apiVersion: keda.sh/v1alpha1
kind: ScaledObject
metadata:
  name: image-worker
  namespace: media
spec:
  scaleTargetRef:
    name: image-worker
  minReplicaCount: 0
  maxReplicaCount: 40
  pollingInterval: 15        # how often KEDA queries the scaler
  cooldownPeriod: 300        # wait before going from 1 to 0
  triggers:
    - type: aws-sqs-queue
      metadata:
        queueURL: https://sqs.eu-west-1.amazonaws.com/111122223333/images
        awsRegion: eu-west-1
        queueLength: "20"          # PER-REPLICA target -> used by the HPA
        activationQueueLength: "5" # only wake from 0 with 5+ messages
        scaleOnInFlight: "true"    # also count in-flight messages
```

With that configuration, four stray messages at three in the morning do not start a pod, a node and an alert. From five on, they do. And once awake, the HPA targets twenty messages per replica.

scaleOnInFlight is worth pausing on. By default the SQS scaler adds visible messages and those being processed (*ApproximateNumberOfMessagesNotVisible*). That is right almost always, because an in-flight message still occupies a worker. But if your processing is very long, the metric does not drop while you work and the autoscaler believes the queue is still full; setting it to *false* makes the metric reflect only pending work. There is no universal answer: it depends on whether your bottleneck is the queue or the time per message.

The full arithmetic, with numbers

It is worth walking the whole path once, because after doing it by hand you stop arguing about why there are seven replicas and not five.

Take an SQS worker with *queueLength: "20"*, *minReplicaCount: 0*, *maxReplicaCount: 40* and 3 replicas running right now. A peak arrives and the queue rises to 260 messages (visible plus in-flight).

1. KEDA queries the scaler every *pollingInterval* and publishes the value on the external metrics API. It publishes the **total**, 260, with an *AverageValue* target of 20.
2. The HPA reads 260 and computes the per-replica average: 260 / 3 = 86.67 messages per pod.
3. It applies the formula: *ceil(3 x (86.67 / 20)) = ceil(13) = 13* replicas.
4. It checks the scale-up *behavior*. With a policy of "at most 100% every 60 seconds", it does not jump to 13 at once: it goes to 6, then 12, then 13.
5. When the queue drains, the HPA computes 0 replicas needed but **will not go below *minReplicaCount***. The final 1-to-0 step is done by the KEDA operator once the scaler has been inactive for the full *cooldownPeriod*.

Two clarifications fall out of that walk. First, ***cooldownPeriod* only affects the 1-to-0 transition**. If your service drops from 30 replicas to 4 too quickly, touching *cooldownPeriod* changes nothing: what rules there is the HPA stabilization window, 300 seconds by default, tuned in *behavior.scaleDown*. Second, the formula uses a ceiling, so at small loads it always rounds up: with a target of 20 and 21 messages, you get 2 replicas.

```
spec:
  advanced:
    horizontalPodAutoscalerConfig:
      behavior:
        scaleUp:
          stabilizationWindowSeconds: 0      # react to the peak immediately
          policies:
            - type: Percent
              value: 100
              periodSeconds: 30
            - type: Pods
              value: 10
              periodSeconds: 30
          selectPolicy: Max                  # whichever grows fastest
        scaleDown:
          stabilizationWindowSeconds: 300    # do not shrink on a 30 s dip
          policies:
            - type: Percent
              value: 25
              periodSeconds: 60
```

The asymmetry is deliberate and it is the right configuration for nearly any queue worker: scaling up fast costs money for a few minutes, scaling down fast costs latency and reprocessed messages during the next peak.

ScaledJob: when the unit of work is not a pod that lives forever

A *ScaledObject* adjusts the replica count of a Deployment that sits waiting for work. Some workloads make that the wrong model: twenty-minute transcodes, reports generated once, batch migrations. If a pod should process one message and die, a Deployment forces you to invent the wait loop, the shutdown signal and the "no work left" logic. That is what *ScaledJob* is for: KEDA creates one Kubernetes *Job* per unit of work and lets the scheduler and *backoffLimit* do their thing.

```
apiVersion: keda.sh/v1alpha1
kind: ScaledJob
metadata:
  name: transcoder
  namespace: media
spec:
  jobTargetRef:
    parallelism: 1
    completions: 1
    backoffLimit: 2
    template:
      spec:
        restartPolicy: Never
        containers:
          - name: worker
            image: registry.internal/transcoder:1.9.3
            resources:
              requests: { cpu: "2", memory: 4Gi }
              limits:   { memory: 6Gi }
  pollingInterval: 20
  maxReplicaCount: 60              # cap on concurrent Jobs (default 100)
  successfulJobsHistoryLimit: 5
  failedJobsHistoryLimit: 20       # leave enough trace to debug
  scalingStrategy:
    strategy: accurate
  triggers:
    - type: aws-sqs-queue
      metadata:
        queueURL: https://sqs.eu-west-1.amazonaws.com/111122223333/transcode
        awsRegion: eu-west-1
        queueLength: "1"
```

The *scalingStrategy* is the part to understand before copying the YAML. The *default* strategy subtracts running Jobs from the queue, which is fine for short work. The *accurate* strategy is designed for queues: instead of trusting total length, it accounts for messages not yet locked by any consumer, so it does not launch sixty Jobs for twenty messages that are already being processed. The *eager* strategy deliberately does the opposite: it fills every available slot up to *maxReplicaCount* to drain the queue as fast as possible. It is conscious over-provisioning, and it only makes sense when the cost of waiting clearly exceeds the cost of idle capacity.

Two operational details you learn the hard way. First, a low *failedJobsHistoryLimit* deletes the evidence exactly when you need it: failed Jobs are your error log in this model. Second, a high *backoffLimit* combined with a poison message produces endless retries that burn capacity without progress; the right pattern is a low *backoffLimit* and a dead-letter queue downstream, not indefinite retries inside the cluster.

Authentication: getting credentials out of the ScaledObject

Almost everyone's first ScaledObject carries an access key in a *Secret* mounted on the workload, or worse, in the manifest itself. It works in the demo and it is exactly what you do not want the day somebody exports the namespace to a repository.

KEDA separates authentication into its own resource: *TriggerAuthentication* (namespaced) and *ClusterTriggerAuthentication* (cluster-wide). The trigger references it and the ScaledObject manifest stops containing secrets.

```
# Option 1: existing secrets, no credentials in the manifest.
apiVersion: keda.sh/v1alpha1
kind: TriggerAuthentication
metadata:
  name: rabbit-auth
  namespace: payments
spec:
  secretTargetRef:
    - parameter: host
      name: rabbitmq-credentials
      key: connection-string
---
# Option 2 (preferable): workload identity, no secret at all.
apiVersion: keda.sh/v1alpha1
kind: TriggerAuthentication
metadata:
  name: sqs-auth
  namespace: media
spec:
  podIdentity:
    provider: aws                       # EKS Pod Identity / IRSA
    roleArn: arn:aws:iam::111122223333:role/keda-queue-reader
---
apiVersion: keda.sh/v1alpha1
kind: ScaledObject
metadata:
  name: image-worker
  namespace: media
spec:
  scaleTargetRef:
    name: image-worker
  triggers:
    - type: aws-sqs-queue
      authenticationRef:
        name: sqs-auth
      metadata:
        queueURL: https://sqs.eu-west-1.amazonaws.com/111122223333/images
        awsRegion: eu-west-1
        queueLength: "20"
```

The second option is the one to aim for: on AWS with EKS Pod Identity or IRSA, on Azure with Workload Identity and the *azure-workload* provider, on GCP with Workload Identity. The static key disappears, the rotation disappears and the risk of leaking it disappears.

There is a permissions nuance worth deciding early. KEDA can assume one operator-wide role for every queue in the cluster, or each *TriggerAuthentication* can point at a different role per application. The first is convenient and ends with a role that can read every queue in the organisation; the second is more manifests and far less blast radius. For the permission KEDA actually needs -- read a queue's attributes, not consume messages -- the minimum policy is short:

```
{
  "Version": "2012-10-17",
  "Statement": [
    {
      "Effect": "Allow",
      "Action": ["sqs:GetQueueAttributes"],
      "Resource": "arn:aws:sqs:eu-west-1:111122223333:images"
    }
  ]
}
```

If your KEDA role has *sqs:ReceiveMessage* or *sqs:**, somebody copied the worker's policy. The autoscaler only looks; it does not consume.

The scalers you will actually use, and their traps

KEDA ships more than seventy scalers and in practice 90% of deployments use four. It is worth knowing the parameters that matter for each instead of discovering them in production.

Kafka: lag is not what it looks like

```
triggers:
  - type: kafka
    metadata:
      bootstrapServers: kafka.data.svc:9092
      consumerGroup: event-processor
      topic: order-events
      lagThreshold: "500"
      activationLagThreshold: "50"
      excludePersistentLag: "true"
      allowIdleConsumers: "false"
```

Two parameters decide whether this works. *allowIdleConsumers* set to *false* stops KEDA from creating more replicas than the topic has partitions, which is Kafka's physical constraint: replica thirteen on a twelve-partition topic consumes nothing and only costs money. And *excludePersistentLag* handles an awkward case: a partition whose consumer is stuck holds a lag that never drops, and without this option KEDA scales forever trying to fix with replicas a problem that is not about capacity.

Prometheus: one query, one value

```
triggers:
  - type: prometheus
    metadata:
      serverAddress: http://prometheus.observability.svc:9090
      query: |
        sum(rate(http_requests_in_progress{service="checkout"}[2m]))
      threshold: "40"
```

```
      activationThreshold: "5"
      ignoreNullValues: "false"
```

The query **must return a single scalar value**. If it returns one series per pod, the scaler fails or picks an arbitrary value, and the symptom is erratic scaling nobody can explain. Always wrap in *sum()* or *avg()*. And pay attention to *ignoreNullValues*: at its default (*true*), a query with no data is read as zero, which makes a dead Prometheus look like zero traffic and your service switches itself off. Setting it to *false* turns that case into a scaler error, which is what you want, because then *fallback* kicks in.

Cron: the pattern that saves the most money with the least risk

```
triggers:
  - type: cron
    metadata:
      timezone: Europe/Madrid
      start: 0 7 * * 1-5      # Monday to Friday at 07:00
      end: 0 21 * * 1-5
      desiredReplicas: "12"
  - type: aws-sqs-queue       # still reacts to peaks outside business hours
    metadata:
      queueURL: https://sqs.eu-west-1.amazonaws.com/111122223333/images
      awsRegion: eu-west-1
      queueLength: "20"
```

With multiple triggers, KEDA evaluates each independently and **takes the largest**. That turns the cron trigger into a scheduled floor: during business hours there are twelve warm replicas whatever happens, and outside them the queue rules. It is the simplest way to remove cold starts exactly in the hours a user would notice, without giving up scaling to zero overnight. Watch the timezone: set *UTC* with a European business and your replicas arrive an hour late all summer.

Really scaling to zero: the cold start budget

Scaling to zero is KEDA's most attractive promise and also the one that causes the most incidents, because the cost does not show up on the invoice but in the first user's latency. Before enabling it, put numbers on the full path from "work arrives" to "a pod is processing it":

- **Detection**: up to one full *pollingInterval*. At the default 30 seconds, the average is 15 and the worst case 30.
- **Scheduling**: a couple of seconds if there is room on an existing node. If there is not, however long your node autoscaler takes.
- **Image pull**: zero if it is cached on that node, and between twenty seconds and several minutes if it is not. In practice this is usually the dominant term.
- **Process startup**: JVM, connection pool, cache warm-up, *readinessProbe* with its *initialDelaySeconds*.

Added up without optimisation, zero to useful is typically forty seconds to three minutes. For a queue worker that is irrelevant. For a synchronous API it is unacceptable, and there the answer is not "tune KEDA" but do not scale to zero: *minReplicaCount: 1* and find the savings elsewhere.

When you do want to scale to zero, three levers work. The first is dropping *pollingInterval* to 10 or 15 seconds for critical ScaledObjects, accepting more scaler calls. The second is keeping the image warm on the nodes with a trivial DaemonSet that pulls it:

```
apiVersion: apps/v1
kind: DaemonSet
metadata:
  name: image-prepull
  namespace: media
spec:
  selector:
    matchLabels: { app: image-prepull }
  template:
    metadata:
      labels: { app: image-prepull }
    spec:
      initContainers:
        - name: pull-worker
          image: registry.internal/image-worker:2.4.1   # same tag as the worker
          command: ["/bin/true"]
      containers:
        - name: pause
          image: registry.k8s.io/pause:3.9
          resources:
            requests: { cpu: 1m, memory: 8Mi }
```

It costs a few millicores per node and removes the biggest term in the cold start. The third lever is *idleReplicaCount*, which allows an idle state other than zero; with the caveat that, because of limitations in the HPA controller itself, the only supported value today is *0*, so in practice it documents intent more than it configures a one.

HTTP services and the KEDA add-on

For HTTP workloads there is the *HTTP Add-on*, which solves the chicken-and-egg problem: you cannot scale from zero on requests if nobody is there to receive the first request. The add-on inserts an *interceptor* that holds the request while the operator wakes the workload, and forwards it once the pod is ready. You can configure a placeholder response, a fallback service, or both for the cold start gap.

It is a very useful piece and should be adopted with eyes open: the add-on is still beta, and the interceptor sits in the critical path of every request to that service, inheriting its availability and its latency. The first request after a cold start typically adds two to five seconds, and considerably more with large images or heavy initialisation. For an internal dashboard or a test environment it is an excellent change; for checkout, do the maths with data.

KEDA does not create nodes: the second autoscaling tier

This is the misunderstanding that most often turns a perfect demo into an incident. KEDA scales **pods**. If there is no capacity in the cluster, those pods sit in *Pending* and your queue keeps growing while the KEDA dashboard shows, entirely correctly, that it asked for forty replicas.

Real autoscaling has two chained tiers and the latency budget is the sum of both:

```
# The thirty-second diagnosis when "KEDA is not scaling":
kubectl get scaledobject -n media          # READY / ACTIVE / FALLBACK
kubectl get hpa -n media                   # TARGETS and REPLICAS
kubectl get pods -n media --field-selector=status.phase=Pending
kubectl describe pod -n media <pending-pod> | tail -20   # the FailedScheduling event says it all
```

If pods are *Pending* with *Insufficient cpu*, the problem is not KEDA: it is your node provider. With Karpenter, a new node takes forty to ninety seconds to be ready; with Cluster Autoscaler and node groups, longer. That time adds to the cold start above and is what really determines how fast your system absorbs a peak.

Two configurations worth their weight in gold when you combine KEDA with a node autoscaler. The first is reserving capacity with low-priority pods the scheduler evicts instantly, so there is always a node with room:

```
apiVersion: scheduling.k8s.io/v1
kind: PriorityClass
metadata:
  name: capacity-reservation
value: -10                       # negative priority: evicted first
globalDefault: false
description: "Filler pods that give up their slot to real work."
---
apiVersion: apps/v1
kind: Deployment
metadata:
  name: reservation
  namespace: capacity
spec:
  replicas: 3
  selector:
    matchLabels: { app: reservation }
  template:
    metadata:
      labels: { app: reservation }
    spec:
      priorityClassName: capacity-reservation
      terminationGracePeriodSeconds: 0
      containers:
        - name: pause
          image: registry.k8s.io/pause:3.9
          resources:
            requests: { cpu: "2", memory: 4Gi }
```

The second is protecting long-running workers from node consolidation. Karpenter repacks workloads to save money, and that means moving pods; if your transcode runs for eighteen minutes, you would rather it were not moved:

```
metadata:
  annotations:
    karpenter.sh/do-not-disrupt: "true"
```

With a sensible *PodDisruptionBudget* and that annotation on long Jobs, you stop losing work to cost optimisations that arrive at the wrong moment.

Observability: knowing what the autoscaler is doing

An autoscaler without its own metrics is a black box that one day makes an inexplicable decision. The KEDA operator exposes Prometheus metrics on port 8080 and four of them belong on the team dashboard from day one.

- *keda_scaler_metrics_value*: the value the scaler is handing the HPA. Compared against your own business metric, it exposes lag and badly written queries.
- *keda_scaler_active*: 1 or 0. The direct answer to "why is it still at zero?".
- *keda_scaler_detail_errors_total* and *keda_scaled_object_errors_total*: errors per scaler and per ScaledObject. A failing scaler nobody watches is scaling that does not happen.
- *keda_scaler_metrics_latency_seconds*: how long it takes to fetch the metric. If it climbs, your *pollingInterval* stops being honoured and autoscaling gets slow without anything reporting an error.

```
groups:
  - name: keda
    rules:
      - alert: KedaScalerErroring
        expr: |
          sum by (scaledObject, scaler) (
            rate(keda_scaler_detail_errors_total[10m])
          ) > 0
        for: 10m
        labels: { severity: warning }
        annotations:
          summary: "Scaler {{ $labels.scaler }} failing on {{ $labels.scaledObject }}"
          description: "Autoscaling is blind or running on fallback."

      - alert: KedaPausedTooLong
        expr: keda_scaled_object_paused == 1
        for: 15m
        labels: { severity: warning }
        annotations:
          summary: "ScaledObject {{ $labels.scaledObject }} paused for over 15 minutes"

      - alert: PendingPodsWithGrowingQueue
        expr: |
          kube_pod_status_phase{phase="Pending", namespace="media"} > 0
          and on() (keda_scaler_metrics_value{scaledObject="image-worker"} > 500)
        for: 5m
        labels: { severity: critical }
        annotations:
          summary: "KEDA is asking for replicas but the cluster has no room"
```

That last alert is what separates "the autoscaler is broken" from "there are no nodes", and it is exactly the question you will be asking at three in the morning.

Operating KEDA: pausing, migrating and living with GitOps

During an incident, sometimes the first thing you want is for the autoscaler to stop having opinions. KEDA allows that with annotations, without deleting anything and without losing the configuration:

```
# Freeze at a specific replica count (the HPA stops acting):
kubectl annotate scaledobject image-worker -n media \
  autoscaling.keda.sh/paused-replicas="25" --overwrite

# Pause without fixing a number: it stays as it is.
kubectl annotate scaledobject image-worker -n media \
  autoscaling.keda.sh/paused="true" --overwrite

# Hand control back to KEDA:
kubectl annotate scaledobject image-worker -n media \
  autoscaling.keda.sh/paused-replicas- autoscaling.keda.sh/paused-
```

This is vastly better than deleting the ScaledObject by hand, which is what people do in a hurry and which has a little-known effect: when you delete a ScaledObject, KEDA removes the HPA it managed and restores the replica count the Deployment had before. If that happens mid-peak, you just cut your capacity exactly when you needed it most.

Migrating from an existing HPA

Moving off a CPU HPA has a mandatory order, because two autoscalers on the same Deployment fight and the result is a pod appearing and disappearing every few seconds.

1. Create the ScaledObject in observation mode: *minReplicaCount* and *maxReplicaCount* matching the

current HPA, with the pause annotation set from the start.

2. Over a full traffic cycle, compare *keda_scaler_metrics_value* against what the CPU HPA is deciding. If KEDA would have scaled at very different moments, your threshold is wrong.
3. **Delete the old HPA.** Do not leave it "just in case": KEDA's webhook will reject it or, if the webhook is absent, you will have two controllers writing the same field.
4. Remove the pause annotation and watch the first few hours.

And a GitOps note that prevents a recurring problem: Argo CD will see the HPA created by KEDA as a resource absent from your repository, and it will see the Deployment's *replicas* field changing constantly. Unconfigured, that means either permanent *OutOfSync* noise or Argo resetting replicas to whatever Git says every few minutes, literally fighting the autoscaler.

```
# In the Argo CD Application
spec:
  syncPolicy:
    syncOptions:
      - RespectIgnoreDifferences=true
  ignoreDifferences:
    - group: apps
      kind: Deployment
      jsonPointers:
        - /spec/replicas          # the number is the autoscaler's, not Git's
```

The general rule: the Deployment manifest in Git should not declare *replicas* at all when a ScaledObject sits on top. What goes in Git is the scaling policy; the concrete number is runtime state.

What this actually costs, and when KEDA is not the answer

The sales pitch for autoscaling is always savings, and it is usually true, but do the full calculation rather than the flattering one. Three costs never show up in the naive comparison between "24 fixed replicas" and "an average of 9".

The first is the **cost of metric queries**. Every ScaledObject queries its source every *pollingInterval*. With CloudWatch or SQS API calls that is billed; with Prometheus it is not billed but it is paid in query load, and one heavy query run every ten seconds by sixty ScaledObjects is an effective way to bring down your own Prometheus. The arithmetic is simple and worth doing: one ScaledObject at *pollingInterval: 10* is 8,640 queries a day; fifty ScaledObjects are 432,000.

The second is the **cost of churn**. Every start pulls an image, opens database pool connections, warms caches and consumes bandwidth. A service oscillating between 8 and 30 replicas every few minutes can spend more on starting than it saves on idle capacity, while punishing the database with a connection pattern no pooler enjoys. If your metrics show more than a dozen scale changes an hour, the fix is not raising the maximum: it is widening the scale-down stabilization window.

The third is the **operational cost**: one more component to upgrade, one more CRD to review on every cluster upgrade and one more root cause to rule out in every incident.

With that on the table, there are three situations where the honest answer is not to use KEDA:

- **Flat, predictable load.** If your service does the same thing at four in the morning as at midday, autoscaling adds risk and saves nothing. A well-chosen fixed number is cheaper and more boring, which in production is a virtue.
- **When the bottleneck is downstream.** If your workers are waiting on a saturated database, adding replicas makes it worse: more connections, more contention, more slowness, and the metric driving the scaling keeps climbing. Here autoscaling turns a performance problem into an outage.

. **When you already run *prometheus-adapter* and cannot remove it.** Because of the single external metrics provider constraint above, that decision belongs at cluster level, not team level.

And a preliminary check that saves arguments: before installing anything, measure for a week the replica count you *would* have needed at each moment and compare it with what you run. If the spread between maximum and minimum is less than a factor of two, the savings will not pay for the complexity.

A hundred ScaledObjects: what changes when this stops being one service

Everything above works the same with one or a hundred ScaledObjects, except three things that only show up at scale.

The operator is a single coordination point. It runs as one active replica (the others stand by via leader election) and evaluates triggers sequentially inside its loop. With many ScaledObjects and slow scalers, the real polling interval drifts from the configured one. The *keda_scaler_metrics_latency_seconds* metric is what warns you, and the fix is usually raising *pollingInterval* on what is not critical rather than lowering it everywhere.

Thresholds do not copy between teams. A *queueLength* of 20 is right for a worker taking 200 ms per message and absurd for one taking 40 seconds. When the template spreads by copy-paste, you end up with twenty badly tuned services and nobody knows why. Document the threshold with the reasoning next to it, not just the number:

```
metadata:
  annotations:
    # 20 messages/replica: each message takes ~1.5 s, target is draining
    # the queue in under 30 s with the replica already warm. Reviewed 2026-09.
    scaling.internal/threshold-rationale: "20 = 30s target / 1.5s per message"
```

Governance becomes necessary. An uncapped *maxReplicaCount* in a namespace with no *ResourceQuota* is an incident waiting for somebody to publish a loop into a queue. The combination that works is simple: a resource quota per namespace as the hard limit, *maxReplicaCount* below that quota as the soft limit, and an admission policy that rejects ScaledObjects with no declared maximum.

Case study: from a queue that grew all afternoon to a 61% lower bill

A document-processing service with an SQS worker, 24 fixed replicas, all day and all night. The load pattern: essentially nothing between 22:00 and 07:00, a heavy peak from 09:00 to 11:00 when accounting firms upload their batches, and a plateau for the rest of the working day. With fixed replicas, the night paid for idle capacity and the morning peak built a queue that took an hour and a half to drain.

Baseline measurement: 24 pods at 2 vCPU and 4 GiB, 720 hours a month, a p95 queue wait of 41 minutes during the peak, and zero latency incidents the rest of the day.

What was changed, in this order:

1. **An SQS ScaledObject** with *queueLength: "30"*, *minReplicaCount: 2*, *maxReplicaCount: 90*. Minimum two, not zero, because work arrives in small constant bursts during the day and the cold start was not worth it.
2. **An additional cron trigger** with a floor of 10 replicas between 08:45 and 11:30 on weekdays. The peak is predictable: there is no sense in rediscovering it every morning with a 4,000-message queue.
3. **Asymmetric *behavior***: scale-up with no stabilization window and up to 100% every 30 seconds;

scale-down with a 600-second window and 20% per minute.

4. **A prepull DaemonSet** for the worker image, which weighed 1.3 GiB, on the relevant node group.
5. **Karpenter** with a spot *NodePool* for this namespace and *do-not-disrupt* on pods processing documents over one hundred pages.

Result after three weeks: an average of 9.2 replicas against 24 fixed, p95 queue wait during the peak down from 41 to 6 minutes (because the scheduled floor plus aggressive scaling absorbs the burst instead of digesting it), and that namespace's compute bill 61% lower. Cold start stopped being a problem once the image pull was removed: from a 95-second median to 22.

What did not work first time: the initial attempt used *minReplicaCount: 0* and support reported that single documents arriving in the afternoon took minutes to process. *activationQueueLength* was at 1, so every document woke the whole system. Raising it to 5 and setting the minimum to 2 fixed the symptom and cost less than half the savings achieved.

The five mistakes you will hit in production

1. GitOps fighting the autoscaler

If your Deployment manifest declares `replicas: 3` and Argo CD syncs every three minutes, you have a guaranteed loop: KEDA scales to 12, Argo pulls it back to 3, the HPA scales up again. Remove `replicas` from the manifest (Kubernetes keeps the current value on apply when the field is absent) or exclude the field explicitly:

```
# Argo CD Application
spec:
  ignoreDifferences:
  - group: apps
    kind: Deployment
    jsonPointers:
    - /spec/replicas
```

2. Two autoscalers on the same Deployment

A hand-written HPA and a ScaledObject pointing at the same target will fight, and the outcome is unpredictable. KEDA catches this with its admission webhook and rejects the ScaledObject. If your goal is to migrate an existing HPA, tell KEDA to take ownership:

```
metadata:
  annotations:
    scaledobject.keda.sh/transfer-hpa-ownership: "true"
```

3. Losing messages on scale-down

Scaling to zero without a clean shutdown means half-processed messages. The worker must catch `SIGTERM`, stop asking for new work, and finish what it already has within `terminationGracePeriodSeconds`:

```
import signal

shutting_down = False

def handle_sigterm(signum, frame):
    global shutting_down
    shutting_down = True      # do not abort: just stop pulling new work
```

```
signal.signal(signal.SIGTERM, handle_sigterm)

def run():
    while not shutting_down:
        response = sqs.receive_message(
            QueueUrl=QUEUE_URL,
            MaxNumberOfMessages=10,
            WaitTimeSeconds=20,        # long polling
        )
        for message in response.get("Messages", []):
            process(message)           # must be idempotent
            sqs.delete_message(
                QueueUrl=QUEUE_URL,
                ReceiptHandle=message["ReceiptHandle"],
            )
    log.info("SIGTERM received, batch finished, exiting cleanly")
```

And on the Deployment, a grace period longer than your slowest task:

```
spec:
  template:
    spec:
      terminationGracePeriodSeconds: 120
```

One detail specific to scale-to-zero: with 20-second long polling, the worker can take that long to even notice it should die, because it is blocked inside `receive_message`. Add it to your grace budget.

4. Confusing "no replicas" with "no nodes"

KEDA scales pods; it does not create nodes. If the cluster is full, your pods sit in `Pending` until the cluster autoscaler or Karpenter provisions capacity, which can double your cold start. Measure the two separately. If startup time is critical, reserve headroom with negative-priority placeholder pods the scheduler can evict instantly:

```
apiVersion: scheduling.k8s.io/v1
kind: PriorityClass
metadata:
  name: overprovisioning
value: -10
globalDefault: false
description: "Placeholder pods, evictable on demand"
```

5. Scaling on a metric that scaling does not improve

If your workers are blocked waiting on a 20-connection Postgres pool, adding pods does not drain the backlog: it increases contention and makes latency worse for everyone. Before autoscaling anything, verify that the bottleneck scales horizontally. It is the most expensive design mistake on this list, and no amount of YAML fixes it.

How to validate it before trusting it

A twenty-minute experiment that saves you an incident:

```
# 1. Confirm the starting state
kubectl get deploy image-worker -n media -o jsonpath="{.spec.replicas}"

# 2. Inject 500 messages and time the cold start
python scripts/seed_queue.py --count 500
kubectl get pods -n media -l app=image-worker -w
```

```
# 3. See what the HPA sees: current value vs target
kubectl describe hpa keda-hpa-image-worker -n media

# 4. Query the raw external metric KEDA is serving
kubectl get --raw "/apis/external.metrics.k8s.io/v1beta1/namespaces/media/s0-aws-sqs-image-jobs" | jq
```

Write down three numbers: seconds from first message to first Ready pod, seconds to drain the queue, and peak replica count. Repeat the measurement after every change to `threshold` or `behavior`. Without that baseline, tuning autoscaling is guessing in YAML.

Checklist before you call a ScaledObject done

- The trigger metric is the one actually limiting your throughput, and the system scales horizontally.
- `threshold` is expressed per replica, and you verified it against the HPA formula.
- `activationThreshold` is set if you scale to zero; `minReplicaCount: 1` on anything with a latency SLO.
- `fallback` defined with `currentReplicasIfHigher` for external triggers.
- `behavior.scaleUp.periodSeconds` is roughly your real pod startup time.
- Clean shutdown: SIGTERM handled, `terminationGracePeriodSeconds` longer than the slowest task, idempotent processing.
- `spec.replicas` removed from the manifest or ignored in GitOps.
- Alerts on `Pending` pods and on ScaledObjects entering `Fallback`.
- Cost measured before and after. If it did not drop, scale-to-zero was not your problem.

Event-driven autoscaling is not magic: it is picking the right signal, understanding who owns which range, and accepting that cold start is a cost you budget for rather than a bug you eliminate. Start with the most expensive idle worker you own, measure it, and decide with the numbers in front of you.

Synchronous services: scale on concurrency, not requests per second

Almost everything above talks about queues because that is where KEDA shines, but more and more teams use it for APIs. There is one decision that goes wrong by default there: scaling on requests per second. It is the easiest metric to get and the worst of the reasonable ones.

The problem is that requests per second says nothing about how much work is inside your process. A thousand requests per second at 5 ms and a hundred requests per second at 900 ms load the service similarly, but an RPS threshold treats the second case as a tenth of the first. The day a dependency slows down, your requests per second *drop* -- because each one takes longer -- and the autoscaler removes replicas exactly when you needed the opposite.

The right metric is **concurrency**: requests in flight. That is what Little's Law ties directly to capacity, and it absorbs latency changes without you touching anything.

```
triggers:
  - type: prometheus
    metadata:
      serverAddress: http://prometheus.observability.svc:9090
      query: |
        sum(http_requests_in_flight{service="checkout"})
      threshold: "25"          # concurrent requests per replica
      activationThreshold: "3"
      ignoreNullValues: "false"
```

Picking the threshold needs no intuition: measure it. Run load against *one* isolated replica, raise concurrency in steps and note where the p95 starts degrading sharply. That is the knee of the curve, and your threshold should sit between 60% and 70% of it, leaving headroom while new replicas start.

```
# Find the knee with a single replica and k6, stepping concurrency up
k6 run --vus-max 200 \
  --stage 1m:10 --stage 1m:25 --stage 1m:50 --stage 1m:100 --stage 1m:200 \
  load.js
```

And one sanity check that avoids the most expensive loop: the per-replica concurrency target cannot exceed what your own server accepts. If you run FastAPI with four Uvicorn workers and a ten-connection database pool, a target of 25 concurrent requests per pod means you are queueing inside the process, and adding replicas will multiply connections against a database that is already the bottleneck. Autoscaling does not fix a limit that lives downstream; it just makes it visible sooner.

FAQ

Does KEDA replace the HPA? No. It generates and configures one. The HPA still does the arithmetic between 1 and N; KEDA feeds it metrics it could not obtain on its own and adds the 0-to-1 transition.

Can I have several triggers on one ScaledObject? Yes, and it is common. KEDA evaluates each and takes the largest replica count. It is an OR, not an AND: one trigger asking for capacity is enough to get it.

What happens if my message broker goes down? The scaler goes into error. Without *fallback*, the HPA keeps its last known decision and stops reacting; with *fallback*, KEDA serves a fixed replica count you define. Always configure it, and alert on it: a service in fallback is running blind.

Does it work for StatefulSets? Yes, and for any custom resource implementing the *scale* subresource. Change the *scaleTargetRef* and add *kind*.

How much does KEDA consume? The operator and metrics server are light, on the order of tens of millicores and a few hundred megabytes on a mid-sized cluster. What does grow is the number of calls to your metric providers: a hundred ScaledObjects at a 10-second *pollingInterval* is 36,000 queries an hour, and that shows up on the CloudWatch bill or in your Prometheus load.

Can I scale on cost, or on another team's queue depth? Technically yes, and it is almost always a bad idea. Scale on a metric your own scaling can improve. If adding replicas does not reduce the metric, you have built a loop that only knows how to grow.

Glossary

- **ScaledObject**: the KEDA resource describing how to scale a Deployment, StatefulSet or scalable resource between a minimum and a maximum, driven by one or more triggers.
- **ScaledJob**: the equivalent for ephemeral workloads, where each unit of work is a Kubernetes Job rather than a Deployment replica.
- **Scaler**: the specific connector that knows how to query a source (SQS, Kafka, Prometheus, cron and seventy more) and return a number.
- **Trigger**: a configured instance of a scaler inside a ScaledObject, with its metadata and threshold.
- **threshold**: the per-replica target passed to the HPA. Governs scaling between 1 and N.
- **activationThreshold**: the minimum metric value for the scaler to count as active. Governs only the 0-to-1 transition and is ignored when the minimum replica count is 1 or more.
- **cooldownPeriod**: how long a scaler must stay inactive before dropping from 1 to 0. It affects no other

scale-down.

- **pollingInterval**: how often KEDA queries the scaler. It defines the worst case of detection latency.
- **fallback**: the replica count to fall back on when the scaler is in error.
- **TriggerAuthentication**: the resource that separates credentials from the ScaledObject, ideally pointing at a workload identity rather than a secret.
- **External metrics**: the Kubernetes API (*external.metrics.k8s.io*) through which the HPA obtains values not originating in the cluster. Only one provider per cluster is allowed.