Cheat sheet de Kubernetes

Para escribir y leer manifests con la documentación abierta, operar con kubectl y diagnosticar un pod roto rápido. Docker construye la imagen; Kubernetes la corre, la escala y la reinicia. Cada sección trae la sintaxis como plantilla con variables, las piezas, un ejemplo real del curso, una figura, un truco para recordarla, la trampa típica y un laboratorio opcional para un cluster local. En la versión web, al final hay un diagnosticador de síntomas, un revisor de YAML y un quiz de errores.

Cómo recordarla · Trampa típica · *variable* la cambias tú · Ejemplos en verde

EN LAS FIGURAS: lo que pides (spec) · listo · esperando · falla · GPU · control plane

0 Números y códigos de bolsillo

¿Qué estados, números y códigos conviene saberse de memoria para leer un pod roto en segundos?

SINTAXIS CON VARIABLES

`kubectl get pods -n` *namespace* `-o wide`

Un pod es la unidad mínima de Kubernetes: uno o varios contenedores. Tres columnas cuentan su historia: `STATUS` (en qué paso está), `READY` (contenedores listos / total) y `RESTARTS` (cuántas veces lo reinició el kubelet, el agente que corre en cada máquina o nodo).

`kubectl describe pod` *pod* `-n` *namespace*

Busca `Last State: Terminated` con su `Reason` y su `Exit Code`: cómo murió el intento anterior. Al final, los **Events** dicen qué pasó y en qué orden.

`kubectl logs` *pod* `-n` *namespace* `--previous`

El log del contenedor que murió. Sin `--previous` ves el intento actual, que quizás todavía no falló.

`kubectl explain` *recurso*`.`*campo*

El cluster te explica el campo y, si tiene default, casi siempre lo dice, para tu versión. Ejemplo: `kubectl explain deployment.spec.progressDeadlineSeconds`.

VARIABLES

VARIABLE	QUÉ ES	EJEMPLO
namespace	carpeta lógica donde vive el pod	`classifier`
pod	nombre del pod, como sale en get pods	`triton-5c7d9b8f6d-q2x7k (ilustrativo)`
recurso	tipo de objeto, en minúsculas	`deployment`
campo	ruta del campo dentro del objeto	`spec.strategy.rollingUpdate.maxSurge`
$n_{\text{señal}}$	número de la señal que mató al proceso (kill -l las lista)	`9 = SIGKILL, 15 = SIGTERM`
t_{gracia}	plazo total para apagarse: terminationGracePeriodSeconds	`45 s en el gateway, 60 s en Triton`
t_{preStop}	lo que dura el preStop (un comando que corre antes de apagar); va dentro del plazo, antes del SIGTERM	`15 s en el gateway, 10 s en Triton`
t_{app}	lo que le queda a tu proceso después del SIGTERM	`45 − 15 = 30 s; uvicorn usa hasta 20`
n	número de la espera: el primer reinicio es inmediato, y la espera 1 va antes del segundo	`la 4.ª espera es de 80 s; de la 6.ª en adelante, 300 s`
N	réplicas deseadas; surge y faltantes son los pods de más y de menos que permite un rolling update (el cambio de versión de a poco)	`N = 3 → surge 1 y faltantes 0; N = 8 → 2 y 2`

FÓRMULAS

$\text{exit code} = 128 + n_{\text{señal}} \qquad 137 = 128 + 9\ (\text{KILL}) \qquad 143 = 128 + 15\ (\text{TERM})$

$t_{\text{app}} = t_{\text{gracia}} - t_{\text{preStop}} \qquad t_{\text{gracia}} = 30\ \text{s por defecto}$

$t_{\text{espera}}(n) = \min\left(10 \cdot 2^{n-1},\ 300\right)\ \text{s} \qquad (\text{vuelve a 10 s tras 10 min corriendo bien})$

$\text{surge} = \lceil 0.25N \rceil \qquad \text{faltantes} = \lfloor 0.25N \rfloor \qquad (\text{defaults del rolling update})$

LAS PIEZAS

PIEZA	QUÉ HACE	EJEMPLO
Pending	El cluster aceptó el pod, pero todavía no tiene nodo o no terminó de preparar sus contenedores.	**Triton esperando una GPU libre**
ContainerCreating	Ya tiene nodo: el kubelet baja la imagen, monta volúmenes y crea el contenedor.	**pull de la imagen de Triton (varios GB)**
Running con READY 0/1	El proceso corre, pero falla la readinessProbe (el chequeo de ¿le mando tráfico?): no recibe requests.	**Triton cargando el engine**
CrashLoopBackOff	Se cae una y otra vez; el kubelet espera antes de reiniciarlo: 10, 20, 40… hasta 300 s.	**RESTARTS sube cada vez más lento**
ImagePullBackOff / ErrImagePull	No pudo bajar la imagen; reintenta con espera creciente.	**tag inexistente o nodo sin permiso en el registry (ECR, el de AWS)**
CreateContainerConfigError	No pudo armar el contenedor: falta su configuración (Secret, ConfigMap) o pediste no-root (runAsNonRoot) y la imagen es root.	**secret "hf-token" not found**
Completed / Error	Motivo que anota el runtime de contenedores al terminar: salió con 0 / salió con otro código.	**Error con Exit Code 1: una excepción de Python**
OOMKilled · 137	Superó su límite de memoria: el kernel lo mató con SIGKILL (128 + 9), una señal que no se atrapa.	**pico de RAM al cargar pesos, sin una línea de log**
137 sin OOMKilled	También SIGKILL, por otra causa: casi siempre no salió dentro del plazo de gracia al borrarlo o reiniciarlo.	**Reason: Error y Exit Code: 137 tras un deploy**
143	128 + 15: murió por el SIGTERM (la señal de apágate). Normal en un deploy o al bajar réplicas.	**un pod viejo de un rolling update**
StartError · 128	El runtime (containerd) no pudo ni arrancar el proceso: el comando no existe o no es ejecutable.	**exec: "uvicorn": executable file not found in $PATH**
126 / 127	Solo aparecen si un shell dentro del contenedor los devuelve: no es ejecutable / no existe.	**command: ["sh", "-c", "uvicorn …"] en una imagen sin uvicorn**
Terminating	Tiene orden de borrado: corre el preStop, recibe SIGTERM y tiene hasta el plazo de gracia para salir.	**hasta 45 s en el gateway**
Evicted	El kubelet lo sacó del nodo porque se quedaba sin memoria o sin disco; el pod queda en Failed.	**The node was low on resource: memory**

CAMPO	DEFAULT	EN EL C01	POR QUÉ CAMBIA
terminationGracePeriodSeconds	30 s	gateway 45 · Triton 60	preStop + apagado del servidor + margen
probes: periodSeconds / timeoutSeconds / failureThreshold	10 s / 1 s / 3	readiness del gateway y de Triton: 5 s / 2 s / 2	salir del Service en ≈ 10 s, con 2 s de holgura por chequeo
probes: initialDelaySeconds / successThreshold	0 s / 1	liveness (¿reiniciar?) del gateway: 5 s · Triton: startupProbe (¿ya arrancó?)	la carga del modelo la cubre la startupProbe (60 × 5 s), no un retraso fijo
rollingUpdate.maxSurge	25 %, redondea hacia arriba	gateway 25 % · Triton 1	cada pod extra de Triton es una GPU extra
rollingUpdate.maxUnavailable	25 %, redondea hacia abajo	0 en los dos	nunca bajar de la capacidad actual durante un deploy

CAMPO	DEFAULT	EN EL C01	POR QUÉ CAMBIA
revisionHistoryLimit	10	5	ReplicaSets viejos para rollout undo; 5 alcanzan
progressDeadlineSeconds	600 s	Triton 900	un pod con nodo GPU nuevo tarda ≈ 4–11 min (estimación del curso)
minReadySeconds	0	sin cambiar	el pod cuenta como disponible apenas está Ready
Espera entre reinicios	1.er reinicio inmediato; después 10 s, ×2 hasta 300 s; vuelve a 10 s tras 10 min bien	—	es del kubelet; el tope se ajusta por nodo (maxContainerRestartPeriod, beta desde 1.35)
HPA (autoscaler de réplicas)	cada 15 s · tolerancia 10 % · bajar tras 300 s de estabilización	gateway: baja 1 pod por minuto	no oscilar con el tráfico del día
Service	type ClusterIP · targetPort = port	targetPort: http (por nombre)	el número del puerto puede cambiar sin tocar el Service
ALB: deregistration delay / idle timeout	300 s / 60 s	30 s / 60 s	con 300 s, cada target viejo puede quedar hasta 5 min drenando

Verbos de bolsillo, en orden: `get` (en qué paso está) → `describe` (por qué) → `logs --previous` (qué dijo al morir) → `events` (en qué orden pasó todo). Y `kubectl explain` te da el default de cualquier campo en la versión de tu propio cluster.

Los defaults que más aparecen al leer un manifest, junto a lo que fija el C01. Verificados contra Kubernetes 1.37 y la documentación de AWS, en octubre de 2026.

EJEMPLO REAL · EL APAGADO DEL GATEWAY DEL C01: 45 = 15 + 20 + MARGEN

YAML `reference/c01/k8s/gateway.yaml (extracto; # … marca líneas omitidas)`

```yaml
spec:
  # preStop (15 s) + --timeout-graceful-shutdown de uvicorn (20 s) + margen
  terminationGracePeriodSeconds: 45
  # …
  containers:
    - name: gateway
      # …
      lifecycle:
        preStop:
          exec:
            # El ALB tarda en desregistrar el target: seguimos atendiendo 15 s antes del SIGTERM
            command: ["sleep", "15"]
```

- `terminationGracePeriodSeconds: 45` es el plazo total (t_gracia). Empieza a contar cuando se ordena borrar el pod; sin el campo serían 30 s.
- El `preStop` es un hook: algo que el kubelet corre dentro del contenedor antes de apagarlo. Aquí duerme 15 s dentro del plazo, y el pod sigue atendiendo mientras el balanceador (ALB) deja de mandarle requests.
- Después llega el SIGTERM. A uvicorn le quedan 45 – 15 = 30 s y usa hasta 20 (`--timeout-graceful-shutdown 20` , en su Dockerfile): sobran 10 de margen.
- Si a los 45 s el proceso sigue vivo, llega el SIGKILL: `Exit Code: 137` con `Reason: Error` , no `OOMKilled` . Si el preStop todavía corre al vencer el plazo, el kubelet da una sola prórroga de 2 s.
- Es el mismo plazo que en Docker dabas con `docker stop -t` (10 s por defecto); en Kubernetes lo fija el pod, y su default es 30 s.

CÓMO RECORDARLA

Resta 128 y busca la señal: **137 = 9 = KILL** (memoria si `Reason` dice `OOMKilled` ; si dice `Error` , casi siempre se acabó el plazo) y **143 = 15 = TERM** (apagado pedido). Los defaults que más muerden: **30 s** de gracia, probes **10 s / 1 s / 3** y rollout **25 % / 25 %**.

TRAMPA

Copiar la tabla de exit codes de Docker. Con containerd (el runtime que usan EKS en AWS y kind en tu Mac), un comando que no existe no da 127: da `Reason: StartError` y `Exit Code: 128` , y el 125 no aparece (lo da el comando docker, no el runtime). Y leer `137` como falta de memoria sin mirar `Reason` .

En el curso: Docker y kubectl · Depurar pods: síntoma → comandos → causa típica · Docker y kubectl · kubectl: mirar, entrar y conectar · M6 · Los objetos que sí debes conocer · M6 · Alta disponibilidad: multi-AZ, PDB y N+1

La cadena: del YAML al pod que recibe tráfico

¿Qué pasa, en orden, entre tu kubectl apply y el primer request que llega a un pod?

SINTAXIS CON VARIABLES

`kubectl apply -f` *archivo* `--dry-run=server`

El API server (la puerta del cluster) revisa el esquema, tus permisos y sus reglas de admisión (admission), sin guardar nada. Lo que falla aquí casi siempre fallaría igual en el apply real.

`kubectl apply -f` *archivo*

Guarda el estado deseado en el cluster y vuelve en segundos: **no** espera a que los pods (las unidades que corren tus contenedores) estén listos.

`kubectl rollout status deployment/` *deploy* `-n` *namespace* `--timeout=` *plazo*

Espera el resto de la cadena: pods creados, con nodo, arrancados y Ready. Sale con error si vence el plazo.

`kubectl get endpointslices -n` *namespace* `-l kubernetes.io/service-name=` *servicio*

El último eslabón: las IPs de los pods detrás del Service (el nombre fijo que les reparte tráfico). `<unset>` = ninguna. Ojo: la columna lista también pods no listos; cuáles reciben tráfico lo dice `conditions.ready`.

VARIABLES

VARIABLE	QUÉ ES	EJEMPLO
archivo	manifest (o carpeta) a aplicar	`k8s/gateway.yaml`
deploy	nombre del Deployment	`gateway`
namespace	namespace de los objetos	`classifier`
plazo	cuánto esperar antes de dar el deploy por fallido	`p. ej. 5m para el gateway; 25m para Triton (estimación del curso)`
servicio	nombre del Service	`gateway`
t_{nodo}	esperar un nodo con espacio; si hay que crearlo, minutos	`≈ 2–5 min para un g4dn nuevo (estimación del curso)`
t_{pull}	bajar la imagen al nodo; cero si ya estaba	`segundos para el gateway; minutos para los varios GB de Triton`
t_{arranque}	el proceso arranca y carga lo que necesita	`Triton carga el engine; su startupProbe (¿ya arrancó?) le da hasta 5 min`
$t_{\text{readiness}}$	hasta que pasa la readinessProbe (¿le mando tráfico?)	`chequeo cada 5 s en el gateway`

FÓRMULAS

$$t_{\text{hasta tráfico}} \approx t_{\text{nodo}} + t_{\text{pull}} + t_{\text{arranque}} + t_{\text{readiness}}$$

LAS PIEZAS

PIEZA	QUÉ HACE	EJEMPLO
`kubeconfig y contexto`	Archivo con tus clusters y credenciales; el contexto dice a cuál le habla kubectl ahora.	`kubectl config current-context`
`kubectl apply`	Manda los objetos del archivo al API server: crea los que faltan y actualiza los que cambiaron.	`kubectl apply -f k8s/gateway.yaml`
`API server (kube-apiserver)`	La única puerta del cluster: autentica, revisa permisos (RBAC), valida y guarda.	`rechaza un typo: strict decoding error: unknown field`
`admission`	Paso del API server que completa o revisa el objeto antes de guardarlo: defaults, políticas, webhooks (programas externos).	`el webhook del AWS LB Controller le agrega al pod un readiness gate (esperar a que el ALB lo vea sano)`
`etcd`	La base de datos del cluster: guarda el estado deseado de cada objeto.	`en EKS la opera AWS`
`Deployment controller`	Lee el Deployment y crea un ReplicaSet (mantiene N copias iguales) a partir de su template, el molde del pod.	`gateway-<hash>`
`ReplicaSet controller`	Crea los pods que faltan para llegar a las réplicas pedidas.	`3 pods recién creados, en Pending`
`scheduler (kube-scheduler)`	Elige nodo para cada pod nuevo según lo que pide (requests), los vetos del nodo (taints) y la zona.	`Triton solo en nodos workload=gpu-t4`

PIEZA	QUÉ HACE	EJEMPLO
kubelet	El agente de cada nodo: arranca los contenedores del pod y corre sus probes (chequeos periódicos).	`evento Started`
container runtime (containerd)	Baja la imagen y crea el proceso aislado, como Docker en tu Mac.	`evento Pulled: cuánto tardó el pull`
plugin de red (CNI)	Le da una IP a cada pod; en EKS, una IP de la red privada de AWS (VPC CNI).	`la IP que el balanceador (ALB) registra como target`
readinessProbe	Chequeo periódico de ¿le mando tráfico? Mientras falle, el pod no recibe requests del Service.	`/readyz en el gateway`
EndpointSlice controller	Arma la lista de IP:puerto de los pods de cada Service y marca cuáles están listos.	`label kubernetes.io/service-name=gateway`
kube-proxy y AWS LB Controller	kube-proxy arma reglas de red en cada nodo; el controller registra esas IPs en el ALB (balanceador de AWS).	`target-type ip: el ALB va directo al pod`

TÚ, CON KUBECTL

1. **Manifest (YAML)**
   `kubectl diff -f k8s/gateway.yaml`
   Tu estado deseado, en git. diff muestra qué cambiaría en el cluster (exit 1 si hay cambios).
2. **kubectl apply**
   `kubectl apply -f k8s/gateway.yaml`
   Manda cada objeto al API server con tus credenciales. Vuelve en segundos: no espera pods.

CONTROL PLANE: GUARDA Y DECIDE

3. **API server**
   `kubectl apply --dry-run=server -f k8s/gateway.yaml`
   Autentica, revisa permisos (RBAC), corre admission y valida el esquema. El objeto que rechaza no pasa a ningún eslabón más.
4. **etcd**
   `kubectl get deployment gateway -n classifier -o yaml`
   El objeto queda guardado con su spec; el status lo irán llenando los controllers.
5. **Deployment controller**
   `kubectl get rs -n classifier -l app.kubernetes.io/name=gateway`
   Crea (o escala) un ReplicaSet gateway-<hash> con el template.
6. **ReplicaSet controller**
   `kubectl get pods -n classifier -l app.kubernetes.io/name=gateway`
   Crea los pods que faltan. Nacen Pending: todavía sin nodo.
7. **Scheduler**
   `kubectl get pods -n classifier -o wide`
   Elige nodo según requests, taints y reparto por zona: se llena la columna NODE.

EN EL NODO: ARRANCA EL CONTENEDOR

8. **kubelet + containerd**
   `kubectl events -n classifier --for pod/POD`
   En el nodo: baja la imagen, monta volúmenes y arranca el contenedor (Pulling, Pulled, Created, Started).
9. **Pod Ready**
   `kubectl get pods -n classifier -l app.kubernetes.io/name=gateway -w`
   La readinessProbe /readyz pasa: READY 1/1. Con el readiness gate del ALB, además espera a que el ALB vea sano al pod.

HACIA EL TRÁFICO

10. **EndpointSlice**
    `kubectl get endpointslices -n classifier -l kubernetes.io/service-name=gateway`
    La IP del pod queda como lista para recibir tráfico.
11. **Tráfico**
    El ALB (target-type ip) le manda requests a la IP del pod cuando su health check pasa; dentro del cluster, kube-proxy reparte las conexiones.

Once eslabones entre tu archivo y el primer request. kubectl apply termina en el cuarto, cuando etcd ya guardó el objeto. El resto lo hacen controllers, el scheduler y el kubelet, y cada uno mira al API server.

EJEMPLO REAL · LOS CUATRO ESLABONES QUE UNEN EL YAML CON EL TRÁFICO (GATEWAY DEL C01)

YAML `reference/c01/k8s/gateway.yaml (extracto del Deployment y del Service; # … marca líneas omitidas)`

```yaml
template:
  metadata:
    labels:
      app.kubernetes.io/name: gateway
      # …
  spec:
    # …
```

```yaml
      containers:
        - name: gateway
          # …
          ports:
            - name: http
              containerPort: 8080
          # …
          readinessProbe:
            httpGet:
              path: /readyz
              port: http
# …
apiVersion: v1
kind: Service
# …
spec:
  type: ClusterIP  # el ALB (target-type ip) apunta directo a las IPs de los pods; el Service da los endpoints
  selector:
    app.kubernetes.io/name: gateway
  ports:
    - name: http
      port: 8080
      targetPort: http
```

- Las labels del `template` se copian a cada pod. `app.kubernetes.io/name: gateway` es lo que buscan el ReplicaSet, el Service y el reparto por zona.
- El puerto del contenedor tiene nombre (`name: http`) y el Service apunta a ese nombre (`targetPort: http`), no al número 8080.
- Mientras `/readyz` falle (Triton no responde o el modelo no cargó), el pod corre pero no recibe requests: su endpoint queda marcado como no listo.
- El `selector` del Service busca esas labels en los pods. Un typo aquí no da ningún error: el Service existe, pero sin endpoints.
- `type: ClusterIP` : el ALB (target-type ip) no pasa por la IP del Service. El AWS Load Balancer Controller lee los endpoints y registra directo las IPs de los pods.

CÓMO RECORDARLA

Apply guarda; los controllers construyen; la readiness abre la puerta. Si el apply salió bien y no llega tráfico, recorre la cadena en orden: ¿ReplicaSet? ¿pods? ¿nodo? ¿Ready? ¿endpoints?

TRAMPA

Creer que `kubectl apply` terminó el deploy: solo guardó el estado deseado y sale en segundos aunque la imagen no exista. Lo que pasa después lo dicen `rollout status` , los eventos y los endpoints. Y un selector con un typo no da error: el Service existe, sin endpoints.

En el curso: M6 · Los objetos que sí debes conocer · Docker y kubectl · kubectl: mirar, entrar y conectar · Docker y kubectl · Depurar pods: síntoma → comandos → causa típica · C01 · Archivos de referencia

1 Qué es Kubernetes y cómo se trabaja

¿Qué hace Kubernetes por ti, se programa o se configura, y qué partes tiene un cluster?

SINTAXIS CON VARIABLES

`aws eks update-kubeconfig --region` *región* **`--name`** *cluster*

Agrega el cluster de EKS (el Kubernetes gestionado de AWS) a tu kubeconfig, el archivo de clusters y credenciales de kubectl, y lo deja como contexto actual.

`kubectl config use-context` *contexto*

Cambia de cluster. Un contexto junta cluster, usuario y namespace por defecto; `kubectl config current-context` dice dónde estás **antes** de cambiar algo.

`kubectl get` *kind* *nombre* **`-n`** *namespace* **`-o yaml`**

Los dos estados del objeto, uno arriba del otro: `spec` (lo que pediste) y `status` (lo que el cluster logró hasta ahora).

`kubectl get nodes -L` *label*

Los nodos (las máquinas del cluster), con esa label (etiqueta clave=valor) como columna. En el C01: `-L workload,topology.kubernetes.io/zone`.

VARIABLES

VARIABLE	QUÉ ES	EJEMPLO
región	región de AWS del cluster	`us-east-1`
cluster	nombre del cluster de EKS	`el de tu equipo (el C01 no lo fija)`
contexto	nombre del contexto en tu kubeconfig	`docker-desktop en tu Mac; en EKS, el ARN del cluster (su identificador en AWS)`
kind	tipo de objeto	`deployment`
nombre	nombre del objeto	`triton`
namespace	carpeta lógica del objeto (namespace)	`classifier`
label	label de nodo que quieres ver como columna	`workload (vale gpu-t4 en los nodos GPU del C01)`
C_{EKS}	lo que cobra AWS por el control plane, aparte de los nodos	`0.10 US$/h en soporte estándar (14 meses por versión)`

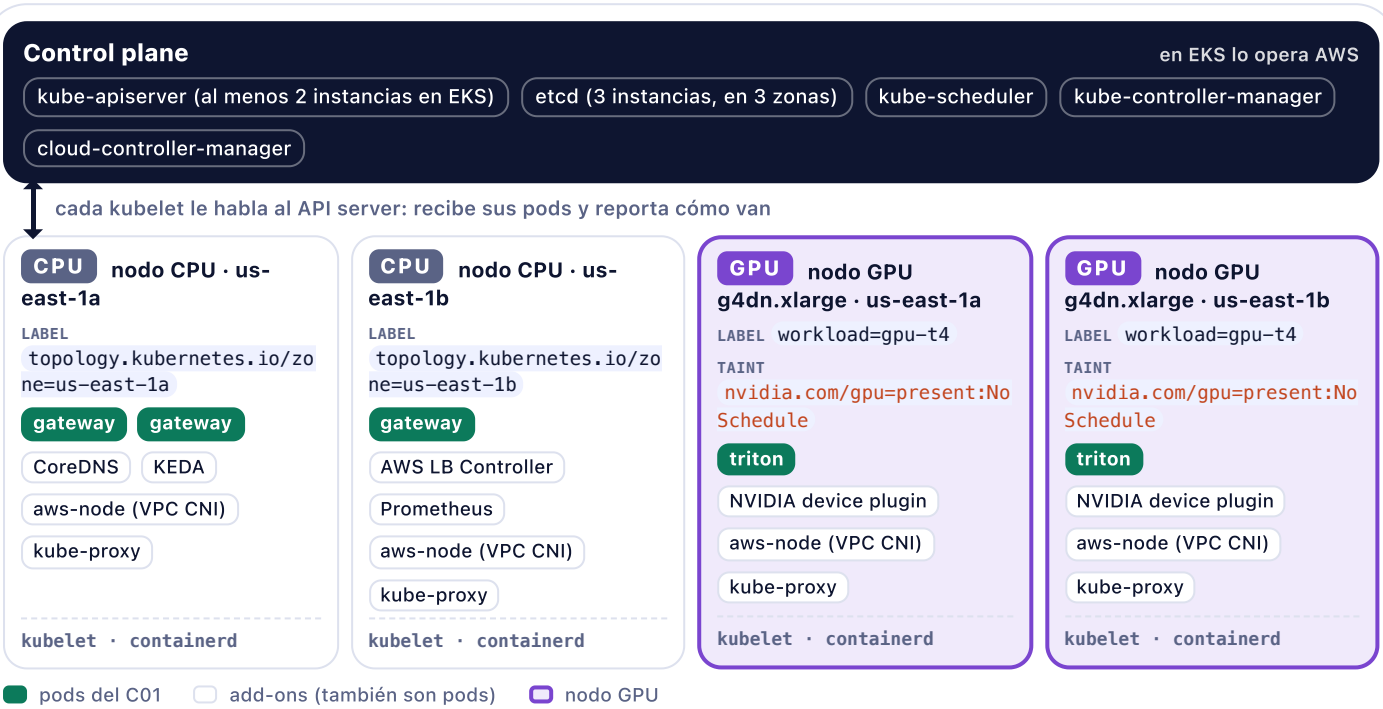
FÓRMULAS

$$C_{\text{EKS}} = 0.10 \,\frac{\text{US\$}}{\text{h}} \times 730 \text{ h} \approx 73 \text{ US\$ al mes por cluster} \qquad (0.60 \text{ US\$/h en soporte extendido})$$

LAS PIEZAS

PIEZA	QUÉ HACE	EJEMPLO
`cluster`	Un grupo de máquinas (nodos) más el control plane que las gobierna.	`el EKS del C01: nodos de CPU y nodos g4dn con T4`
`nodo (node)`	Una máquina, física o virtual (en EKS, una máquina virtual de AWS), donde corren tus pods (grupos de contenedores).	`g4dn.xlarge con la label workload=gpu-t4`
`control plane`	El cerebro: guarda lo que pediste y decide qué hacer. No corre tus pods.	`en EKS lo opera AWS`
`kube-apiserver`	Expone la API HTTP del cluster. Todo pasa por aquí, también kubectl.	`kubectl get pods = un GET a la API`
`etcd`	Base de datos clave-valor, consistente y replicada, con todos los objetos.	`en EKS: 3 instancias en 3 zonas`
`kube-scheduler`	Asigna un nodo a cada pod que todavía no tiene.	`busca un nodo con una T4 libre para Triton`
`kube-controller-manager`	Corre los controllers: lazos que acercan lo real a lo deseado.	`Deployment, ReplicaSet, EndpointSlice`
`kubelet`	El agente de cada nodo: arranca los contenedores de sus pods y los vigila.	`reinicia un Triton que falla la liveness (¿sigue vivo?)`
`container runtime`	Baja imágenes y corre contenedores en el nodo.	`containerd en los nodos de EKS`
`kube-proxy y CNI`	kube-proxy arma las reglas de los Services; el plugin de red (CNI) le da IP a cada pod.	`aws-node (VPC CNI) en cada nodo de EKS`
`add-on`	Software extra que corre como pods del cluster: DNS, métricas, drivers.	`CoreDNS, NVIDIA device plugin, KEDA`

PIEZA	QUÉ HACE	EJEMPLO
controller propio u operator	Programa que vigila objetos y actúa: eso sí es código, y rara vez lo escribes tú.	**KEDA: agrega el kind ScaledObject con una CRD (definición de un tipo nuevo) y crea un HPA (autoscaler de réplicas)**
kubectl	Herramienta de línea de comandos (CLI): lee tu kubeconfig y habla con la API. Tolera una versión menor de diferencia.	**kubectl 1.37 con clusters 1.36 a 1.38**
Amazon EKS	Kubernetes gestionado: AWS opera el control plane; nodos y add-ons siguen siendo tuyos.	**0.10 US$/h por cluster (≈ 73 al mes)**



AWS opera todo el control plane (0.10 US$/h por cluster) y tú no ves esos procesos como pods. Nodos, kubelet, containerd y add-ons corren en tu cuenta. Con EKS Auto Mode, AWS también opera los nodos y varios add-ons.

El cluster del C01 en EKS, simplificado. Arriba, el control plane que opera AWS; abajo, los nodos de tu cuenta, cada uno con kubelet, containerd, kube-proxy y su plugin de red. Los nodos GPU llevan label y taint para que solo Triton caiga ahí.

EJEMPLO REAL · LO QUE EL C01 NECESITA EN EL CLUSTER ANTES DEL PRIMER APPLY

TEXTO **reference/c01/README.md (Desplegar en EKS, prerequisitos)**

```
Prerequisitos del cluster: NVIDIA device plugin, node group de `g4dn.xlarge` con label
`workload=gpu-t4` y taint `nvidia.com/gpu=present:NoSchedule`, AWS Load Balancer Controller, KEDA,
Prometheus (kube-prometheus-stack, con kube-state-metrics), un CNI que aplique NetworkPolicy (VPC CNI
con `enableNetworkPolicy=true`, o Calico) y las imágenes en ECR. Kubernetes ≥ 1.35: 1.34 sale del
soporte upstream el 27 de octubre de 2026 y del soporte estándar de EKS el 2 de diciembre de 2026. Nada
en los manifests depende de 1.34: pasan la validación estricta contra 1.35–1.37 y
`trafficDistribution: PreferSameZone` es GA desde 1.35 (en 1.31–1.33 cámbialo por `PreferClose`).
```

- Nada de esta lista es parte del control plane que opera AWS: son nodos y add-ons que tu equipo instala o configura (con Helm, Terraform o como add-ons de EKS).
- El NVIDIA device plugin es un DaemonSet (un pod por nodo; aquí, en cada nodo GPU) que le anuncia las GPUs al kubelet. Sin él, ningún nodo ofrece `nvidia.com/gpu` y Triton queda Pending.
- La label y el taint (un veto que aleja a los demás pods) del grupo de nodos GPU son la mitad del acuerdo para ubicar a Triton. La otra mitad está en `triton.yaml`: el nodeSelector (en qué nodos puede caer) y la toleration (el permiso para ignorar ese veto). En EKS, un paso del API server, el admission plugin ExtendedResourceToleration, agregaría solo una toleration equivalente, porque Triton pide `nvidia.com/gpu`. Escribirla la deja explícita y sirve en cualquier cluster.
- AWS Load Balancer Controller, KEDA y el Prometheus Operator son controllers: vigilan sus objetos (Ingress, ScaledObject, ServiceMonitor) y actúan. Los kinds nuevos que usa el C01 (ScaledObject, ServiceMonitor) llegan con CRDs: definiciones de tipos que se suman a la API.

- Una NetworkPolicy necesita un plugin de red que la aplique. Sin él, el objeto se acepta y no hace nada.
- ≥ 1.35 se cumple de sobra: en octubre de 2026 la versión más nueva es la 1.37. La 1.34 dejó de recibir parches el 27 de octubre de 2026; en EKS, su soporte estándar termina el 2 de diciembre de 2026.

CÓMO RECORDARLA

Kubernetes funciona como un **termostato para contenedores**: tú fijas el estado deseado en YAML y los controllers corrigen sin parar hacia él. **Se configura** con YAML y `kubectl apply`; tu código vive en la imagen.

TRAMPA

Pensar que `kubectl` le habla a los nodos o a los contenedores: solo le habla al API server, y todo lo demás lo hacen controllers que leen de ahí. Por eso un objeto aceptado todavía no es un pod corriendo. Y en EKS, buscar etcd o el scheduler en `kube-system`: AWS los corre fuera de tus nodos.

En el curso: M6 · Qué resuelve Kubernetes (y cuándo no lo necesitas) · M6 · Los objetos que sí debes conocer · M13 · Managed vs self-hosted: la cuenta de los dos lados · Docker y kubectl · kubectl: mirar, entrar y conectar

2 Anatomía de un manifest

¿Qué partes tiene un manifest, quién escribe cada una y cómo se encuentran los objetos entre sí?

SINTAXIS CON VARIABLES

```
apiVersion: grupo / versión
kind: Kind
metadata:
  name: nombre
  namespace: namespace
  labels:
    clave : valor
spec:
  # el estado deseado: cambia según el kind
```

Lo escribes tú. Los objetos básicos (grupo core) no llevan grupo: `apiVersion: v1` (Service, ConfigMap); el resto sí: `apps/v1`, `networking.k8s.io/v1`.

```
spec:
  selector:
    matchLabels:
      clave : valor
  template:
    metadata:
      labels:
        clave : valor
```

En un Deployment, las labels que busca el selector tienen que estar en las del template (el molde de los pods); si no, el API server lo rechaza.

`kubectl api-resources --api-group=` *grupo*

Qué tipos de objeto (kinds) existen en ese grupo, con su apiVersion y si van dentro de un namespace. Con `--api-group=keda.sh` ves si KEDA (el autoscaler por eventos del C01) está instalado.

`kubectl get pods -n` *namespace* **`-l`** *clave* **`=`** *valor*

Busca por label (etiqueta clave=valor), igual que un Service. Si aquí no sale ningún pod, el Service tampoco encuentra ninguno.

VARIABLES

VARIABLE	QUÉ ES	EJEMPLO
grupo	grupo de la API; el core no tiene	`apps (Deployment), networking.k8s.io (Ingress), keda.sh (ScaledObject)`
versión	versión dentro del grupo	`v1; v1alpha1 en el ScaledObject de KEDA`
Kind	tipo de objeto, en CamelCase	`Deployment`
nombre	nombre único por kind y namespace, en minúsculas	`gateway`
namespace	carpeta lógica del objeto	`classifier`
clave	clave de una label	`app.kubernetes.io/name`
valor	valor de esa label	`gateway`

FÓRMULAS

$$\{\text{pares del selector}\} \subseteq \{\text{labels del pod}\} \implies \text{el objeto gobierna (o apunta a) ese pod}$$

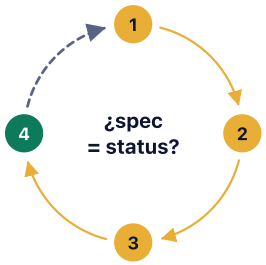
LAS PIEZAS

PIEZA	QUÉ HACE	EJEMPLO
`apiVersion`	Grupo y versión de la API que define el kind. El core no lleva grupo.	`apps/v1 · v1 · keda.sh/v1alpha1`
`kind`	Qué tipo de objeto es, en CamelCase.	`Deployment, Service, ScaledObject`
`metadata.name`	Nombre único dentro de su kind y su namespace, en minúsculas.	`gateway (Deployment) y gateway (Service): kinds distintos`
`metadata.namespace`	Carpeta lógica. Si falta, el objeto va al namespace de tu contexto (casi siempre default).	`classifier`
`metadata.labels`	Etiquetas clave-valor para buscar y agrupar; es lo que miran los selectors (filtros por label).	`app.kubernetes.io/name: gateway`
`metadata.annotations`	Notas para herramientas y controllers; no sirven para seleccionar.	`alb.ingress.kubernetes.io/scheme: internal`
`spec`	El estado deseado: lo escribes tú.	`réplicas, imagen, puertos, probes`
`status`	El estado real: lo escribe el cluster. No va en tu YAML.	`readyReplicas, conditions`

PIEZA	QUÉ HACE	EJEMPLO
spec.selector	Qué pods gobierna el objeto. En un Deployment no se puede cambiar después.	`matchLabels: app.kubernetes.io/name: gateway`
spec.template	El molde de los pods que crea el controller. Cambiarlo dispara un rollout (cambio de versión).	`labels, containers, probes`
--- (separador)	Separa varios objetos dentro de un mismo archivo YAML.	`gateway.yaml: Deployment, Service, HPA e Ingress`
CRD (CustomResourceDefinition)	Agrega un kind nuevo a la API. Sin ella, apply falla con no matches for kind.	`ScaledObject (KEDA), ServiceMonitor (Prometheus Operator)`
ownerReferences	Quién es dueño del objeto: al borrar al dueño, el cluster borra a los hijos.	`Pod → ReplicaSet → Deployment`
app.kubernetes.io/*	Labels recomendadas: name, instance, version, component, part-of, managed-by.	`part-of: classifier`

DESEADO · SPEC

spec del Deployment gateway: 3 réplicas del template con la imagen :3f9c2ab (las 3 las fijó el HPA)



REAL · STATUS

status: 2 réplicas listas, porque vaciaron un nodo para mantenimiento (drain) y desalojaron uno de los pods

1. ve la diferencia: 3 deseadas, 2 que existen
2. el ReplicaSet crea un pod nuevo con el mismo template
3. el scheduler le busca nodo y el kubelet lo arranca
4. pasa la readiness: status vuelve a 3 de 3 y el lazo queda quieto

QUIÉN CORRE EL LAZO: Deployment y ReplicaSet controllers (dentro de kube-controller-manager)

spec es la temperatura que fijas en el termostato y status la del cuarto: el controller los compara sin parar y actúa hasta que coinciden. Tú solo escribes spec.

EJEMPLO REAL · EL DEPLOYMENT DEL GATEWAY: TRES LUGARES CON LABELS, TRES TRABAJOS DISTINTOS

YAML `reference/c01/k8s/gateway.yaml (líneas 4–27)`

```yaml
apiVersion: apps/v1
kind: Deployment
metadata:
  name: gateway
  namespace: classifier
  labels:
    app.kubernetes.io/name: gateway
    app.kubernetes.io/part-of: classifier
spec:
  # Sin "replicas": las decide el HPA de abajo (3–12)
  revisionHistoryLimit: 5
  strategy:
    type: RollingUpdate
    rollingUpdate:
      maxSurge: 25%  # pods nuevos antes de bajar viejos
      maxUnavailable: 0  # capacidad completa durante todo el deploy
  selector:
    matchLabels:
      app.kubernetes.io/name: gateway
  template:
    metadata:
      labels:
        app.kubernetes.io/name: gateway
        app.kubernetes.io/part-of: classifier
```

- `apps/v1` : grupo `apps` , versión `v1` . El Service del mismo archivo usa `v1` a secas porque es del grupo core.
- `metadata.labels` etiqueta al Deployment mismo: sirve para encontrarlo (`kubectl get deploy -l …`), no decide qué pods gobierna.

- `spec.selector.matchLabels` decide qué pods son suyos. En `apps/v1` es obligatorio e inmutable: cambiarlo obliga a borrar y recrear el Deployment.
- `spec.template.metadata.labels` se copia a cada pod. Tiene que contener al selector y puede tener labels de más (aquí, `part-of`).
- No hay `status` ni `replicas`: el status lo escribe el cluster y las réplicas las decide el HPA (el autoscaler de réplicas) del mismo archivo.

CÓMO RECORDARLA

apiVersion + kind dicen qué es; **metadata**, cómo se llama y dónde vive; **spec**, lo que quieres; **status**, lo que hay (lo escribe el cluster). Las **labels** son el pegamento: Services, PodDisruptionBudgets (cuántos pods deben quedar en pie en un mantenimiento) y NetworkPolicies (el firewall entre pods) encuentran pods por labels, nunca por nombre.

TRAMPA

Copiar un manifest y cambiar solo `metadata.name`: si el selector y las labels quedan iguales, dos Deployments pelean por los mismos pods. Y olvidar el namespace en un archivo: el objeto cae en el namespace de tu contexto (casi siempre `default`) y desde `classifier` nadie lo ve.

En el curso: M6 · Los objetos que sí debes conocer | Docker y kubectl · kubectl: mirar, entrar y conectar | C01 · Archivos de referencia

3 Pods, ReplicaSets y Deployments

¿Cómo mantiene Kubernetes N copias de tu servicio y cómo cambia de versión sin cortar?

SINTAXIS CON VARIABLES

```
spec:
  replicas: n
  strategy:
    type: RollingUpdate
    rollingUpdate:
      maxSurge: extra
      maxUnavailable: faltantes
```

Quita `replicas` si un autoscaler (HPA o KEDA) decide cuántas. Defaults de la estrategia: 25 % y 25 %.

`kubectl get deploy,rs,pods -n` *namespace* `-l` *selector*

La cadena completa de un vistazo: el Deployment, sus ReplicaSets (uno por versión del template) y los pods.

`kubectl set image deployment/` *deploy* *contenedor*`=` *imagen* `-n` *namespace*

Cambia la imagen en el template y dispara un rollout (el cambio de versión de a poco). Sirve en un lab; en producción, el cambio va en git y se aplica.

`kubectl rollout history deployment/` *deploy* `-n` *namespace*

Las revisiones guardadas: una por ReplicaSet, el actual y hasta `revisionHistoryLimit` viejos. Es a lo que puede volver `rollout undo`.

VARIABLES

VARIABLE	QUÉ ES	EJEMPLO
n	réplicas deseadas	16 en el vLLM del c10; sin campo en el C01 (las deciden el HPA y KEDA)
extra	maxSurge: pods de más durante el rollout (número o %)	25 % en el gateway, 1 en Triton
faltantes	maxUnavailable: pods de menos durante el rollout	0 en el C01
namespace	namespace del Deployment	classifier
selector	label para filtrar	app.kubernetes.io/name=triton
deploy	nombre del Deployment	triton
contenedor	nombre del contenedor dentro del template	triton
imagen	imagen nueva, con tag	123456789012.dkr.ecr.us-east-1.amazonaws.com/classifier-triton:2026-09-15-resnet50-fp16
N	réplicas deseadas en ese momento	2 (mínimo de KEDA para Triton)
s, u	maxSurge y maxUnavailable como fracción	0.25 por defecto

FÓRMULAS

$\text{surge} = \lceil s \cdot N \rceil$ $\quad \text{faltantes} = \lfloor u \cdot N \rfloor$ $\quad (s = u = 0.25 \text{ por defecto})$

$\text{disponibles} \geq N - \text{faltantes}$ $\quad \text{pods totales} \leq N + \text{surge}$

$\text{GPUs máx. durante el rollout} = (N + \text{surge}) \times \text{GPUs por pod}$ $\quad \text{Triton: } (2+1) \times 1 = 3$

LAS PIEZAS

PIEZA	QUÉ HACE	EJEMPLO
Pod	La unidad mínima: uno o varios contenedores que comparten IP y volúmenes, en un solo nodo.	un pod de Triton = un contenedor + una T4
contenedores del pod	Nacen y mueren con el pod y se hablan por localhost.	un agente de logs junto a la app
sidecar nativo	Contenedor de arranque (initContainer) con restartPolicy: Always: arranca antes y se apaga después del principal (estable desde 1.33).	un proxy o un agente de métricas
restartPolicy	Qué hace el kubelet (agente del nodo) al terminar un contenedor. En un Deployment, el del pod solo admite Always.	Always (Deployment) · OnFailure o Never (Job)
ReplicaSet	Mantiene N pods iguales: si falta uno, lo crea; si sobra, lo borra.	triton-5c7d9b8f6d (ilustrativo)
Deployment	Maneja ReplicaSets, uno por versión del template (el molde del pod), para cambiar de versión de a poco.	triton, gateway

PIEZA	QUÉ HACE	EJEMPLO
pod-template-hash	Label que el Deployment pone en cada ReplicaSet y sus pods para no mezclar versiones.	**triton-5c7d9b8f6d-q2x7k**
RollingUpdate	Estrategia por defecto: sube pods nuevos y baja viejos de a poco.	**gateway y Triton del C01**
Recreate	Borra todos los pods viejos antes de crear los nuevos: hay corte.	**dos versiones que no pueden convivir**
maxSurge / maxUnavailable	Cuántos pods de más y de menos se permiten durante el rollout.	**Triton: 1 y 0**
revisionHistoryLimit	ReplicaSets viejos (en 0 réplicas) que se guardan para volver atrás. Default 10.	**5 en el C01**
progressDeadlineSeconds	Plazo para avanzar. Si vence: Progressing=False con ProgressDeadlineExceeded. No revierte solo.	**900 s en Triton**
minReadySeconds	Segundos que un pod nuevo debe seguir Ready para contar como disponible. Default 0.	**el C01 no lo usa**
StatefulSet · DaemonSet · Job	Otros controllers de pods: identidad estable · un pod por nodo · correr hasta terminar.	**el NVIDIA device plugin es un DaemonSet**

Deployment `triton` lo escribes tú

spec: el template nuevo (otra imagen), maxSurge 1, maxUnavailable 0; las réplicas (2 a 4) las escribe el HPA de KEDA

↓ crea y controla ↑ ownerReferences: apunta a su dueño

ReplicaSet `triton-<hash nuevo>` lo crea el Deployment

sube de 0 a 2 de a un pod; el viejo, triton-<hash viejo>, baja de 2 a 0 y queda guardado para rollout undo

↓ crea y controla ↑ ownerReferences: apunta a su dueño

Pod `triton-<hash nuevo>-<sufijo>` lo crea el ReplicaSet

nace Pending hasta que hay una T4 libre; cuenta como disponible cuando pasa la readiness. Si su contenedor muere, el kubelet lo reinicia dentro del mismo pod

SE ENCUENTRAN POR LABELS: app.kubernetes.io/name=triton (y pod-template-hash, que agrega el Deployment)

Cada nivel maneja al de abajo: el Deployment, a sus ReplicaSets (uno por versión del template), y cada ReplicaSet, a sus pods. Durante un rollout conviven dos ReplicaSets.

EJEMPLO REAL · EL DEPLOYMENT DE TRITON: ROLLING UPDATE CUANDO CADA POD ES UNA GPU

YAML `reference/c01/k8s/triton.yaml (líneas 12–32)`

```yaml
spec:
  # Sin "replicas": las gestiona el HPA que crea KEDA. Si lo fijas aquí, cada `kubectl apply`
  # resetea la escala y pelea con el autoscaler.
  revisionHistoryLimit: 5  # ReplicaSets viejos para `kubectl rollout undo`
  progressDeadlineSeconds: 900  # un pod con nodo GPU nuevo tarda ≈ 4–11 min: el default de 600 s vence en el caso pesimista
  strategy:
    type: RollingUpdate
    rollingUpdate:
      maxSurge: 1  # de a un pod nuevo: cada pod necesita una GPU libre (Karpenter/cluster-autoscaler agrega el nodo)
      maxUnavailable: 0  # nunca bajar de la capacidad actual durante un deploy
  selector:
    matchLabels:
      app.kubernetes.io/name: triton  # inmutable: debe coincidir con las labels del template
  template:
    metadata:
      labels:
        app.kubernetes.io/name: triton
        app.kubernetes.io/part-of: classifier
    spec:
      # Tiempo total para apagar: preStop (10 s) + --exit-timeout-secs de Triton (30 s) + margen
      terminationGracePeriodSeconds: 60
```

- Sin `replicas`: el número lo pone el HPA (el autoscaler de réplicas de Kubernetes) que KEDA crea y alimenta para Triton (mínimo 2). Si lo fijaras aquí, cada `kubectl apply` lo pisaría.
- `maxSurge: 1` y `maxUnavailable: 0`: con 2 réplicas, el rollout tiene 3 pods un rato. El tercero necesita una T4 libre, o un nodo nuevo: por eso va de a uno.
- `progressDeadlineSeconds: 900`: un pod que espera nodo GPU, imagen y engine puede pasar los 600 s del default. Vencido el plazo, el Deployment marca `ProgressDeadlineExceeded`, pero no revierte solo.
- `revisionHistoryLimit: 5`: guarda 5 ReplicaSets viejos en 0 réplicas. Volver al modelo anterior con `kubectl rollout undo` es re-escalar uno de ellos.
- El comentario del selector lo dice: es inmutable y tiene que coincidir con las labels del template. El Deployment, además, agrega `pod-template-hash` a cada ReplicaSet para no mezclar versiones.

CÓMO RECORDARLA

Deployment → ReplicaSet → Pod: el Deployment cambia de versión, el ReplicaSet cuenta copias y el pod corre. Un rollout es un ReplicaSet nuevo que sube mientras el viejo baja, al ritmo de **maxSurge** (de más) y **maxUnavailable** (de menos).

TRAMPA

Esperar un rollout al editar un ConfigMap (la configuración que vive aparte del pod) o al escalar: solo cambiar `spec.template` dispara uno. Y en un servicio de GPU, el surge por defecto pide GPUs extra que quizás no existen: el pod nuevo queda Pending y el deploy se queda a medias.

En el curso: M6 · Los objetos que sí debes conocer · M6 · Alta disponibilidad: multi-AZ, PDB y N+1 · Docker y kubectl · Rollouts y escalado · M11 · Rollback: criterios automáticos y qué guardar para volver

4 Services, DNS e Ingress

¿Cómo encuentra el tráfico a tus pods si cambian de IP cada vez que se recrean?

SINTAXIS CON VARIABLES

```
apiVersion: v1
kind: Service
metadata:
  name: servicio
spec:
  selector:
    clave : valor
  ports:
    - name: nombre_puerto
      port: puerto
      targetPort: puerto_pod
```

Un nombre y una IP estables delante de los pods (las unidades que corren tus contenedores) que cumplen el selector, el filtro por labels. Sin `type` es ClusterIP: solo se alcanza desde dentro del cluster.

`servicio` . `namespace` `.svc.cluster.local:` `puerto`

El nombre completo en el DNS interno del cluster (CoreDNS). Desde el mismo namespace alcanza con `servicio` ; desde otro, con `servicio` . `namespace` .

```
spec:
  ingressClassName: clase
  rules:
    - http:
        paths:
          - path: ruta
            pathType: tipo
            backend:
              service:
                name: servicio
                port:
                  name: nombre_puerto
```

Un Ingress solo describe reglas HTTP (qué ruta va a qué Service): lo implementa el controller de su clase (`alb` = AWS Load Balancer Controller). Sin controller, no pasa nada.

`kubectl port-forward svc/` `servicio` `puerto_local` `:` `puerto` `-n` `namespace`

Un túnel desde tu laptop hasta **un** pod detrás del Service, para probar sin exponer nada. No sirve para pruebas de carga.

VARIABLES

VARIABLE	QUÉ ES	EJEMPLO
`servicio`	nombre del Service (y de su DNS)	`triton`
`clave`	clave de la label que busca el selector	`app.kubernetes.io/name`
`valor`	valor de esa label	`triton`
`nombre_puerto`	nombre del puerto; obligatorio si el Service tiene varios	`http`
`puerto`	puerto del Service, al que llamas	`8000`
`puerto_pod`	puerto del contenedor, por número o por nombre; si falta, igual a port	`http (= containerPort 8000)`
`namespace`	namespace del Service	`classifier`
`clase`	ingressClassName: qué controller atiende el Ingress	`alb`
`ruta`	path de la regla	`/v1/classify`
`tipo`	pathType: Exact, Prefix o ImplementationSpecific	`Exact`
`puerto_local`	puerto en tu laptop	`8000`
$t_{\text{keep-alive app}}$	cuánto deja la app abierta una conexión sin uso	`75 s (--timeout-keep-alive de uvicorn)`
$t_{\text{idle ALB}}$	cuánto deja el balanceador de AWS (ALB) abierta una conexión sin uso; default 60 s	`60 s en el C01`

FÓRMULAS

$t^{\text{app}}_{\text{keep-alive}} > t^{\text{ALB}}_{\text{idle}}$ $\quad 75\text{ s} > 60\text{ s}$ (si no, el ALB reusa conexiones que la app ya cerró: 502)

LAS PIEZAS

PIEZA	QUÉ HACE	EJEMPLO
Service	Nombre e IP estables delante de los pods que cumplen su selector.	`triton, gateway`
ClusterIP (default)	IP virtual solo dentro del cluster; kube-proxy (reglas de red en cada nodo) reparte por conexión, no por request.	`Triton: nunca expuesto afuera`
NodePort	Abre el mismo puerto (30000–32767 por defecto) en todos los nodos.	`lo necesita un ALB en modo instance`

PIEZA	QUÉ HACE	EJEMPLO
LoadBalancer	Le pide a la nube un balanceador externo para el Service.	en AWS, un balanceador de red (NLB) con el AWS LB Controller
headless (clusterIP: None)	Sin IP virtual: el DNS devuelve directo las IPs de los pods listos.	asr-worker del c13: el router elige el pod
selector	Las labels que busca el Service; solo admite igualdad (clave = valor).	app.kubernetes.io/name: triton
port / targetPort	port: al que llamas. targetPort: el del contenedor; si falta, vale lo mismo que port.	8000 → http
EndpointSlice	Lista de IP:puerto de los pods del Service, con cuáles están listos.	kubernetes.io/service-name=triton
DNS del cluster (CoreDNS)	Traduce el nombre del Service a su IP: servicio.namespace.svc.cluster.local.	triton.classifier.svc.cluster.local
trafficDistribution	Preferencia de ruteo: PreferSameZone manda a pods de la misma zona si hay (estable desde 1.35).	Service triton: no cruzar de zona
Ingress	Reglas HTTP (host, path) hacia Services. Su API está congelada: no recibe cambios nuevos.	/v1/classify → gateway:http
IngressClass y controller	Quién implementa el Ingress; sin controller, el Ingress no hace nada.	ingressClassName: alb
Gateway API	El reemplazo que recomienda Kubernetes para Ingress: Gateway + HTTPRoute.	para diseños nuevos
NetworkPolicy	Firewall entre pods, por labels. Lo aplica el plugin de red, si lo soporta.	solo el gateway entra a Triton:8000

camino del request — fuera del camino

Cliente en la VPC `HTTPS 443`
el backend del marketplace u otro equipo; resuelve el DNS del ALB interno

ALB interno (Ingress gateway)
`443 → regla /v1/classify (Exact)`
lo crea el AWS Load Balancer Controller; health check a /readyz; deregistro en 30 s

target-type ip: el ALB va directo a la IP de cada pod

Pods gateway `containerPort 8080 (http)`
readiness /readyz; preStop de 15 s mientras el ALB lo desregistra

Service triton (ClusterIP, PreferSameZone) `8000 → http`
TRITON_URL = triton.classifier.svc.cluster.local:8000; kube-proxy elige un pod listo

Pods triton `8000 (http) · 8002 (metrics)`
la NetworkPolicy solo deja entrar al gateway al 8000 y al namespace monitoring al 8002

FUERA DEL CAMINO: DE AQUÍ SACA LAS IPS EL CONTROLLER DEL ALB

Service gateway (ClusterIP)
`8080 → targetPort http`
no está en el camino del ALB: el controller lee sus endpoints para registrar targets

EndpointSlice del gateway `IP del pod:8080`
label kubernetes.io/service-name=gateway; un pod que falla la readiness queda como no listo

DNS triton.classifier.svc.cluster.local → IP virtual del Service (ClusterIP), estable aunque los pods cambien de IP

El camino de un request del C01. El ALB va directo a las IPs de los pods del gateway (target-type ip). El gateway llama a Triton por el DNS de su Service, y kube-proxy elige un pod listo, de su zona si hay.

EJEMPLO REAL · EL INGRESS DEL GATEWAY: UN ALB INTERNO QUE VA DIRECTO A LOS PODS

YAML `reference/c01/k8s/gateway.yaml (el Ingress completo, sin 3 líneas de comentario)`

```yaml
apiVersion: networking.k8s.io/v1
kind: Ingress
metadata:
  name: gateway
```

```yaml
  namespace: classifier
  annotations:
    alb.ingress.kubernetes.io/scheme: internal
    # ip: el ALB manda directo a la IP del pod (sin NodePort ni salto extra de kube-proxy)
    alb.ingress.kubernetes.io/target-type: ip
    alb.ingress.kubernetes.io/listen-ports: '[{"HTTPS": 443}]'
    alb.ingress.kubernetes.io/certificate-arn: arn:aws:acm:us-east-1:123456789012:certificate/00000000-0000-0000-0000-
000000000000
    # El health check del ALB usa la misma señal que la readiness de k8s
    alb.ingress.kubernetes.io/healthcheck-path: /readyz
    alb.ingress.kubernetes.io/success-codes: "200"
    # Drenado al sacar un pod: 30 s (el default de 300 s hace eternos los deploys)
    alb.ingress.kubernetes.io/target-group-attributes: deregistration_delay.timeout_seconds=30
    # Idle timeout 60 s < --timeout-keep-alive 75 de uvicorn: el ALB siempre cierra primero (sin 502)
    alb.ingress.kubernetes.io/load-balancer-attributes: idle_timeout.timeout_seconds=60
spec:
  ingressClassName: alb  # lo atiende el AWS Load Balancer Controller
  rules:
    - http:
        paths:
          - path: /v1/classify
            pathType: Exact
            backend:
              service:
                name: gateway
                port:
                  name: http
```

- `ingressClassName: alb`: el AWS Load Balancer Controller ve este objeto y crea un balanceador de aplicaciones de AWS (ALB). Sin ese controller instalado, el Ingress existe y nada pasa.
- `scheme: internal` (también el default): el ALB solo se alcanza desde la red privada (VPC). `internet-facing` lo expondría a internet.
- `target-type: ip`: el ALB registra las IPs de los pods y les manda tráfico directo. El Service gateway no está en el camino: el controller solo lee sus endpoints. El default, `instance`, exigiría un Service NodePort (o LoadBalancer).
- `healthcheck-path: /readyz`: el ALB y la readiness de Kubernetes miran la misma señal. Sin la anotación, el ALB revisaría `/`.
- `deregistration_delay` en 30 s (el default de AWS es 300) y el preStop de 15 s del Deployment trabajan juntos. El pod sigue atendiendo mientras el ALB deja de mandarle requests.
- Solo `/v1/classify`, con `pathType: Exact`: `/metrics`, `/healthz` y `/readyz` no quedan expuestos fuera del cluster.

CÓMO RECORDARLA

Los pods cambian de IP; el Service no. El selector busca pods por labels, el EndpointSlice marca cuáles están listos y el DNS da `servicio.namespace.svc.cluster.local`. El Ingress (o Gateway API) es la puerta HTTP desde afuera, y siempre necesita un controller.

TRAMPA

Un Service con un selector que no coincide no da error: existe, tiene IP y DNS, y cada llamada falla porque no tiene endpoints. Lo ves con `kubectl get endpointslices -l kubernetes.io/service-name=…`. Y una NetworkPolicy en un cluster cuyo plugin de red no la aplica se acepta sin aviso y no bloquea nada.

En el curso: C01 · Serving · M6 · Alta disponibilidad: multi-AZ, PDB y N+1 · Docker y kubectl · kubectl: mirar, entrar y conectar · C01 · Archivos de referencia

5 Probes y apagado ordenado

¿Cómo sabe Kubernetes si tu contenedor ya arrancó, si puede recibir tráfico o si se colgó? ¿Y cómo lo apaga sin cortar requests a la mitad?

SINTAXIS CON VARIABLES

```
startupProbe:
  httpGet:
    path: ruta_listo
    port: puerto
  periodSeconds: periodo
  failureThreshold: fallos
```

Una probe (sonda) es una pregunta periódica del kubelet, el agente de cada nodo, a tu contenedor. La de arranque da **fallos × periodo** segundos; mientras, las otras dos esperan.

```
readinessProbe:
  httpGet:
    path: ruta_listo
    port: puerto
  periodSeconds: periodo
  timeoutSeconds: timeout
  failureThreshold: fallos
```

¿Le mando tráfico? Si falla, el pod sale de los endpoints del Service (la lista de pods que reciben tráfico), sin reiniciarse.

```
livenessProbe:
  httpGet:
    path: ruta_vivo
    port: puerto
  periodSeconds: periodo
  timeoutSeconds: timeout
  failureThreshold: fallos
```

¿Se colgó? Si falla `failureThreshold` veces seguidas, el kubelet mata el contenedor y lo reinicia. Pregunta solo por el proceso, nunca por una dependencia.

```
spec:
  terminationGracePeriodSeconds: gracia
  containers:
    - name: contenedor
      lifecycle:
        preStop:
          sleep:
            seconds: espera
```

Apagado: primero el `preStop` (un paso que corre antes de apagar), luego SIGTERM (la señal de "termina") y, si sigue vivo al vencer `terminationGracePeriodSeconds` (el plazo de gracia), SIGKILL (muerte inmediata). La acción `sleep` es estable desde 1.34 y la corre el kubelet, así que la imagen no necesita un `sleep`; C01 usa `exec` con `sleep 10`.

VARIABLES

VARIABLE	QUÉ ES	EJEMPLO
ruta_listo	ruta que da 200 solo cuando el modelo está cargado	`/v2/health/ready` (Triton), `/readyz` (gateway)
ruta_vivo	ruta barata que solo dice si el proceso responde	`/v2/health/live` (Triton), `/healthz` (gateway)
puerto	número o nombre del puerto del contenedor	`http` (= 8000 en Triton)
periodo	cada cuántos segundos pregunta (periodSeconds; default 10, mínimo 1)	`5`
timeout	segundos que espera la respuesta (timeoutSeconds; default 1)	`2`
fallos	fallos seguidos para declarar la falla (failureThreshold; default 3)	`60` en la startup de Triton; `2` en su readiness
gracia	plazo total del apagado, preStop incluido (default 30)	`60` (Triton), `45` (gateway)
contenedor	nombre del contenedor dentro del pod	`triton`
espera	segundos que el preStop retrasa el SIGTERM	`10` (Triton), `15` (gateway)
t_{arranque}	lo que tu app tarda en quedar lista; tiene que caber en f × p de la startup	carga del engine (el modelo compilado) + warmup (inferencias de calentamiento) ≤ 60 × 5 s = 300 s
f	failureThreshold de esa probe	`60` (startup), `2` (readiness), `3` (liveness) en Triton
p	periodSeconds de esa probe	`5`, `5` y `10` s
t_{detectar}	cuánto tarda en notarse una falla	readiness: 2 × 5 ≈ 10 s; liveness: 3 × 10 ≈ 30 s
t_{gracia}	terminationGracePeriodSeconds	`60` s en Triton
t_{preStop}	duración del preStop; corre dentro del plazo	`10` s
t_{drenado}	lo que el servidor tarda en terminar lo que tiene en curso	hasta `30` s (`--exit-timeout-secs=30`)

VARIABLE	QUÉ ES	EJEMPLO
margen	colchón para el resto del apagado	20 s (60 − 10 − 30)

FÓRMULAS

$$t_{\text{arranque}} \leq f \times p$$

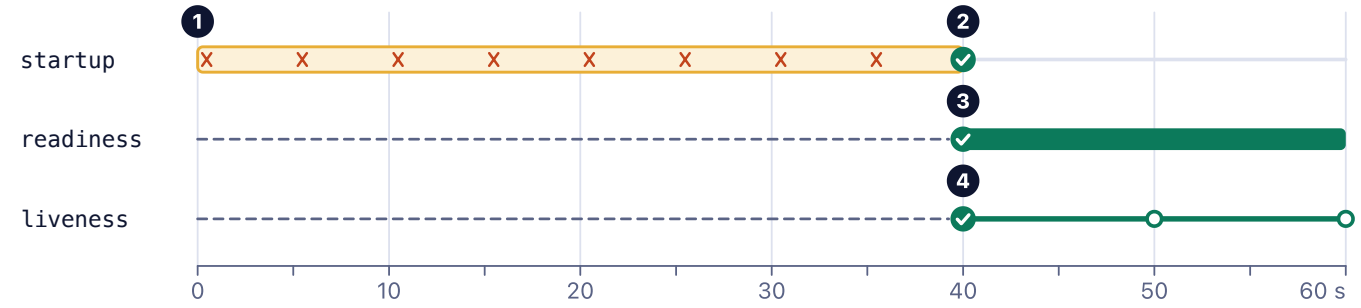
$$t_{\text{detectar}} \approx f \times p$$

$$t_{\text{gracia}} \geq t_{\text{preStop}} + t_{\text{drenado}} + \text{margen}$$

LAS PIEZAS

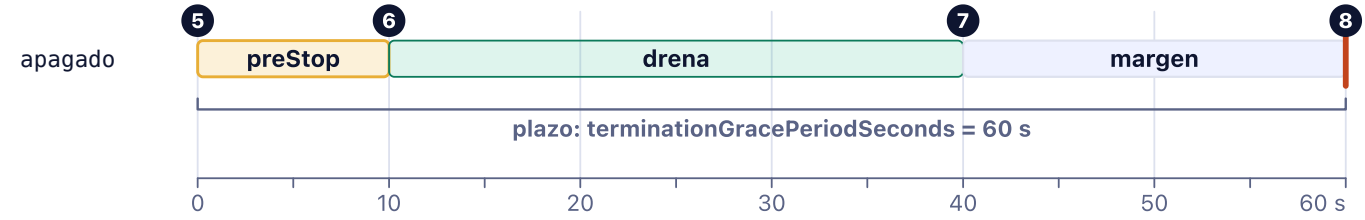
PIEZA	QUÉ HACE	EJEMPLO
startupProbe	¿Ya arrancó? Mientras no pasa, no corren liveness ni readiness. Si agota f × p, el kubelet reinicia el contenedor.	Triton: 60 × 5 s = 5 min
readinessProbe	¿Le mando tráfico? Si falla, el pod sale de los endpoints del Service. Sigue vivo y sigue probando.	READY 0/1 con RESTARTS sin cambio
livenessProbe	¿Se colgó? Si falla f veces seguidas, el kubelet corre el preStop, mata el contenedor y lo reinicia.	/v2/health/live cada 10 s
httpGet · tcpSocket · grpc · exec	Cómo pregunta: GET con respuesta 200–399, puerto que acepta conexión, health check de gRPC que responde SERVING, o comando con exit 0.	exec: ["cat", "/tmp/listo"] (crea procesos en cada probe: en nodos llenos, mejor httpGet)
periodSeconds · timeoutSeconds	Cada cuánto pregunta y cuánto espera la respuesta. Defaults: 10 s y 1 s.	Triton: 5 s y 2 s
failureThreshold · successThreshold	Fallos seguidos para declarar la falla (default 3) y éxitos para volver (default 1; liveness y startup: siempre 1).	readiness de Triton: 2 fallos
initialDelaySeconds	Espera antes de la primera probe (default 0). Con una startupProbe casi nunca hace falta.	liveness del gateway: 5
Condición Ready	Pod listo: todos sus contenedores pasan la readiness y se cumplen los readinessGates. El Service solo manda tráfico a pods Ready.	kubectl get pods → READY 1/1
readinessGates	Condiciones extra para estar Ready. El AWS Load Balancer Controller agrega una: que el ALB (el balanceador de AWS) ya vea sano al pod.	label elbv2.k8s.aws/pod-readiness-gate-inject=enabled en el namespace
lifecycle.preStop	Corre antes del SIGTERM y dentro del plazo. Mantiene el proceso atendiendo mientras el pod sale de los endpoints.	sleep 10 (Triton), sleep 15 (gateway)
terminationGracePeriodSeconds	Plazo total del apagado, preStop incluido. Al vencer, SIGKILL (exit 137); si el preStop sigue, recibe 2 s extra.	60 (Triton), 45 (gateway); default 30
Eventos Unhealthy y Killing	Lo que deja cada falla en kubectl describe pod y kubectl events.	Readiness probe failed: HTTP probe failed with statuscode: 503
HEALTHCHECK del Dockerfile	Kubernetes no lo usa: solo cuentan las probes del manifest.	el del gateway sirve en docker run y en Compose

ARRANQUE · t = 0: arranca el contenedor



1. **0 s** — **startup** · arranca el contenedor; startup cada 5 s: falla (connection refused)
2. **40 s** — startup OK: cabía en 60 × 5 = 300 s
3. **40 s** — readiness OK: Ready, entra a los endpoints
4. **40 s** — liveness empieza: /v2/health/live cada 10 s

APAGADO · t = 0: kubectl delete, scale-down, rollout o drain



5. **0 s** — borrado: Terminating; el endpoint pasa a no listo; corre el preStop (sleep 10)
6. **10 s** — SIGTERM: Triton espera hasta 30 s a que terminen las inferencias en curso
7. **40 s** — Triton sale a más tardar aquí
8. **60 s** — vence el plazo: SIGKILL a lo que quede (exit 137)

la startup prueba y falla · pasa: Ready y en los endpoints · espera a la startup · SIGKILL

Arriba, el arranque de un pod de Triton (t = 0 al arrancar el contenedor; en este ejemplo el engine tarda ≈ 40 s). Abajo, el apagado (t = 0 al borrar el pod), con el plazo de 60 s de C01.

LAS TRES PROBES DE UN VISTAZO

PROBE	PREGUNTA	SI FALLA	EN TRITON (C01)
startupProbe	¿Ya terminó de arrancar?	Al agotar f × p, reinicia el contenedor. Mientras no pasa, las otras dos no corren.	/v2/health/ready cada 5 s, hasta 60 fallos = 5 min
readinessProbe	¿Le mando tráfico?	El pod sale de los endpoints (READY 0/1). No se reinicia.	/v2/health/ready cada 5 s, 2 fallos ≈ 10 s
livenessProbe	¿Se colgó el proceso?	Reinicia el contenedor (con preStop y plazo de gracia).	/v2/health/live cada 10 s, 3 fallos ≈ 30 s

Defaults si no los escribes: periodSeconds 10, timeoutSeconds 1, failureThreshold 3, successThreshold 1, initialDelaySeconds 0. Readiness y liveness no se esperan entre sí: solo la startup frena a las otras dos.

EJEMPLO REAL · LAS PROBES Y EL PRESTOP DE TRITON EN C01

YAML `reference/c01/k8s/triton.yaml (líneas 91–119, contenedor triton; sangría reducida)`

```yaml
startupProbe:
  httpGet:
    path: /v2/health/ready
    port: http
  periodSeconds: 5
  failureThreshold: 60  # hasta 5 min para quedar listo
# readiness: ¿recibe tráfico? ready = servidor + modelo cargado (--strict-readiness)
readinessProbe:
  httpGet:
    path: /v2/health/ready
    port: http
```

```
  periodSeconds: 5
  timeoutSeconds: 2
  failureThreshold: 2  # ~10 s sin estar listo → sale del Service
# liveness: ¿el proceso está colgado? Solo "live", NO "ready": un modelo que se descarga
# no se arregla reiniciando en bucle
livenessProbe:
  httpGet:
    path: /v2/health/live
    port: http
  periodSeconds: 10
  timeoutSeconds: 2
  failureThreshold: 3
lifecycle:
  preStop:
    exec:
      # Al borrar el pod, el Service tarda unos segundos en sacarlo de los endpoints.
      # Seguimos sirviendo 10 s antes del SIGTERM para no cortar requests en vuelo.
      command: ["sleep", "10"]
```

- La `startupProbe` le da a Triton hasta 60 × 5 s = 5 min para cargar el engine y hacer el warmup. Mientras no pasa, el kubelet no corre la liveness: no mata a un pod que solo está cargando.
- Startup y readiness preguntan lo mismo, `/v2/health/ready`, que con `--strict-readiness=true` da 200 solo con el modelo cargado. Antes de eso Triton ni abre el puerto: la probe falla con connection refused, y es normal.
- La readiness saca al pod del Service en ≈ 2 × 5 s = 10 s. `timeoutSeconds: 2` evita el default de 1 s, que bajo carga da falsos negativos.
- La liveness pregunta `/v2/health/live` (el proceso responde), no `ready`: si el modelo no carga, reiniciar en bucle no lo arregla. Un cuelgue se detecta en ≈ 3 × 10 s = 30 s.
- El preStop de 10 s corre dentro de `terminationGracePeriodSeconds: 60` (línea 32): 10 s de preStop + 30 s de `--exit-timeout-secs` + 20 s de margen. El gateway hace lo mismo con 15 + 20 (uvicorn) + 10 = 45, porque el ALB tarda más en dejar de mandarle tráfico que kube-proxy (el que reparte el tráfico de los Services en cada nodo).

CÓMO RECORDARLA

startup = ¿ya arrancó?, readiness = ¿le mando tráfico?, liveness = ¿lo reinicio?, y al apagar, preStop → SIGTERM → SIGKILL dentro de `terminationGracePeriodSeconds`. Como en un restaurante: la startup espera a que la cocina esté lista, la readiness es el letrero de abierto o cerrado (cerrar no despide a nadie) y la liveness reemplaza a quien cocina si se desmayó.

TRAMPA

poner una dependencia en la liveness (Triton en la del gateway, la base de datos en la de tu API): si la dependencia cae, el kubelet reinicia en cascada pods sanos que no arreglan nada, así que las dependencias van en la readiness. Sin `startupProbe`, un modelo que tarda 3 min en cargar muere a manos de la liveness (3 × 10 s con los defaults) antes de terminar, y el pod entra en CrashLoopBackOff (reinicios con esperas cada vez más largas). Y con el `timeoutSeconds` de 1 s por defecto, un proceso saturado falla la liveness justo en el pico: reinicias capacidad cuando más falta.

En el curso: M6 · Los objetos que sí debes conocer · M6 · Alta disponibilidad: multi-AZ, PDB y N+1 · Docker y kubectl · Depurar pods: síntoma → comandos → causa típica · C01 · Modos de falla y degradación · C01 · Archivos de referencia

6 Requests, limits y QoS

¿Cuánta CPU, memoria y GPU reserva tu pod, qué pasa si se pasa, y a quién saca Kubernetes primero cuando a un nodo le falta memoria?

SINTAXIS CON VARIABLES

```
resources:
  requests:
    cpu: "cpu"
    memory: mem
  limits:
    memory: mem
```

El **request** es tu reserva: el scheduler, que elige el nodo, solo te pone donde cabe. El **limit** es el techo. C01: CPU sin limit; memoria con limit = request.

```
resources:
  requests:
    cpu: "cpu"
    memory: mem
    nvidia.com/gpu: gpus
  limits:
    memory: mem
    nvidia.com/gpu: gpus
```

La GPU es un recurso extendido (Kubernetes no la trae de fábrica): la anuncia el device plugin de NVIDIA, un pod que corre en cada nodo GPU. Se pide en `limits`, en números enteros; si también va en `requests`, tiene que ser el mismo número.

```
resources:
  requests:
    cpu: "cpu"
    memory: mem
  limits:
    cpu: "cpu"
    memory: mem
```

Clase de QoS (calidad de servicio) Guaranteed: request = limit de CPU y de memoria en cada contenedor. Suele ser de lo último en salir si al nodo le falta memoria; a cambio, el limit de CPU trae throttling (la CPU se frena al llegar al tope).

```
kubectl describe node nodo
kubectl get pod pod -o
jsonpath='{.status.qosClass}'
kubectl top pod pod --containers
```

Lo que ve el scheduler (Allocatable, Allocated resources), la clase de QoS (calidad de servicio) y el uso real, que requiere metrics-server (el medidor de CPU y memoria).

VARIABLES

VARIABLE	QUÉ ES	EJEMPLO
cpu	cores: "1", "2" (las comillas son opcionales), o milicores con m ("500m" = medio core)	"1" (gateway), "2" (Triton)
mem	bytes con sufijo: Mi y Gi son base 2; M y G, base 10	1Gi (gateway), 6Gi (Triton)
gpus	GPUs enteras. Salvo que configures una forma de compartirla (time-slicing: los pods se turnan; MIG: la GPU partida en pedazos), cada pod se queda con una GPU completa	1 (una T4)
nodo	nombre del nodo, de kubectl get nodes	docker-desktop en tu Mac
pod	nombre del pod, de kubectl get pods	oom (en el lab)
$\text{request}_{\text{nuevo}}$	request del pod que el scheduler quiere colocar; se compara por recurso: CPU, memoria y GPU	Triton: 2 CPU, 6Gi y 1 GPU
Allocatable	lo que el nodo ofrece a los pods: su capacidad menos lo reservado para el sistema operativo, el kubelet (el agente del nodo) y el umbral de desalojo	una g4dn.xlarge tiene 4 vCPU, 16 GiB y 1 T4 de capacidad; Allocatable es algo menor en CPU y memoria
t_{agota}	en qué momento de cada ventana de 100 ms se acaba la cuota de CPU	25 ms
limit	limits.cpu en cores	1
n_{hilos}	hilos usando CPU a la vez	4 decodes en paralelo
u	utilización que mide el HPA (sección 9)	0.6 cores / 1 core = 60 %

FÓRMULAS

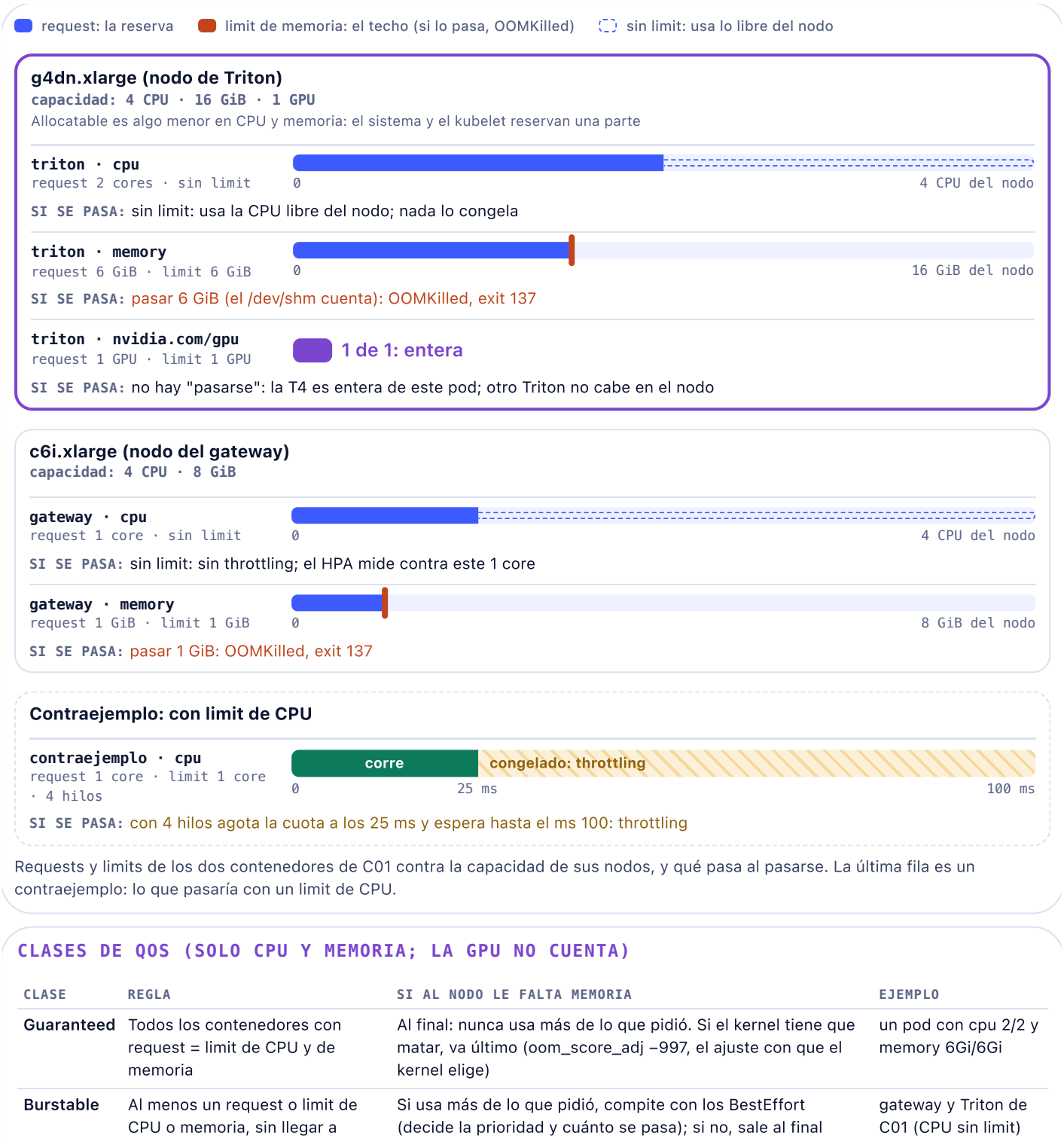
$$\sum_{\text{pods del nodo}} \text{request} + \text{request}_{\text{nuevo}} \leq \text{Allocatable}$$

$$t_{\text{agota}} = \frac{\text{limit} \times 100\text{ms}}{n_{\text{hilos}}}$$

$$u = \frac{\text{uso de CPU}}{\text{request de CPU}}$$

LAS PIEZAS

PIEZA	QUÉ HACE	EJEMPLO
requests.cpu	Reserva para elegir nodo y peso cuando varios contenedores compiten por CPU. Es la base del % del HPA (el autoscaler de réplicas, sección 9).	**"1" en el gateway: 60 % = 0.6 cores**
limits.cpu	Cuota por ventana de 100 ms. Al agotarla, el kernel congela el contenedor hasta la siguiente (throttling). No lo mata.	**C01 no lo pone**
requests.memory	Reserva para elegir nodo. Si al nodo le falta memoria, salen primero los pods que usan más que su request.	**6Gi (Triton)**
limits.memory	Techo duro. Si lo pasa, el kernel mata el proceso (OOMKilled, exit 137) y el kubelet reinicia el contenedor.	**= request en C01**
Unidades	CPU en cores: 500m es medio core. Memoria en bytes: Mi y Gi base 2; M y G base 10.	**cpu: 500m · memory: 512Mi (memory: 400m serían 0.4 bytes)**
Solo limits, sin requests	Kubernetes copia el limit como request, salvo que un LimitRange (defaults por namespace) ponga otro.	**limits.memory: 1Gi → request 1Gi**
nvidia.com/gpu	Recurso extendido que anuncia el device plugin de NVIDIA. Enteros, sin sobreasignar, request = limit.	**una T4 entera por pod de Triton**
Allocatable · Allocated resources	Lo que el nodo ofrece y lo que ya está reservado. El scheduler suma requests, no el uso real.	**kubectl describe node**
Clase de QoS (calidad de servicio)	Guaranteed, Burstable o BestEffort según requests y limits de CPU y memoria. La GPU no cuenta.	**gateway y Triton: Burstable**
ephemeral-storage	Disco local del nodo: lo que el contenedor escribe fuera de volúmenes, sus logs y los emptyDir (carpetas temporales del pod). Pasarse del limit desaloja el pod.	**C04: request 10Gi, limit 30Gi**
emptyDir con medium: Memory	Carpeta en RAM (tmpfs). Lo que escribes ahí cuenta contra el limit de memoria del contenedor.	**/dev/shm de Triton, sizeLimit 1Gi**
LimitRange · ResourceQuota	Defaults de requests y limits por namespace, y topes totales del namespace (también de GPUs).	**requests.nvidia.com/gpu: 4**
Resize en el lugar (in-place)	Cambia CPU y memoria de un pod sin recrearlo (estable desde 1.35). No aplica a GPUs ni cambia la QoS.	**kubectl patch pod … --subresource resize (kubectl ≥ 1.32)**
OOMKilled · Evicted	OOMKilled: el kernel lo mató por memoria (casi siempre, pasó su limit). Evicted: el kubelet sacó el pod porque al nodo le faltaba memoria o disco.	**The node was low on resource: memory.**



Requests y limits de los dos contenedores de C01 contra la capacidad de sus nodos, y qué pasa al pasarse. La última fila es un contraejemplo: lo que pasaría con un limit de CPU.

CLASES DE QOS (SOLO CPU Y MEMORIA; LA GPU NO CUENTA)

CLASE	REGLA	SI AL NODO LE FALTA MEMORIA	EJEMPLO
Guaranteed	Todos los contenedores con request = limit de CPU y de memoria	Al final: nunca usa más de lo que pidió. Si el kernel tiene que matar, va último (oom_score_adj −997, el ajuste con que el kernel elige)	un pod con cpu 2/2 y memory 6Gi/6Gi
Burstable	Al menos un request o limit de CPU o memoria, sin llegar a Guaranteed	Si usa más de lo que pidió, compite con los BestEffort (decide la prioridad y cuánto se pasa); si no, sale al final	gateway y Triton de C01 (CPU sin limit)
BestEffort	Ningún request ni limit de CPU ni de memoria	Entre los primeros: todo lo que usa pasa su request de 0 (oom_score_adj 1000: el kernel lo mata primero)	un pod que solo pide nvidia.com/gpu: 1

El kubelet no mira la clase: saca primero a los pods que usan más memoria de la que pidieron y, entre ellos, decide por prioridad y por cuánto se pasan. La QoS solo sirve para predecir ese orden. Si el nodo se queda sin memoria antes de que el kubelet actúe, mata el kernel, y ahí sí pesa la clase.

EJEMPLO REAL · REQUESTS Y LIMITS DEL GATEWAY Y DE TRITON

YAML `reference/c01/k8s/gateway.yaml (líneas 67–75) y reference/c01/k8s/triton.yaml (líneas 76–84); sangría reducida`

```yaml
resources:
  requests:
    cpu: "1"  # base del HPA: 60 % de utilización = 0.6 cores por pod
    memory: 1Gi  # ≈ 80 MB de Python + 2 decodes de peor caso (≈ 320 MB c/u) + bodies en vuelo; con
PREPROCESS_CONCURRENCY=4 sube a 2Gi (4 × 320 MB ya pasa de 1Gi)
  limits:
```

```
    memory: 1Gi  # límite = request: si se pasa, OOMKilled claro en vez de afectar al nodo
    # SIN límite de CPU a propósito: con límite, CFS reparte la CPU en ventanas de 100 ms;
    # una ráfaga de decodes agota la cuota y el pod queda "throttled" el resto de la
    # ventana → picos de p99 aunque el uso promedio sea bajo. El request ya garantiza su parte.

# … y el contenedor de k8s/triton.yaml (líneas 76–84):
resources:
  requests:
    cpu: "2"  # Triton usa CPU para red, deserializar y el scheduler de batching
    memory: 6Gi  # librerías CUDA + buffers pinned; el modelo en sí vive en la GPU
    nvidia.com/gpu: 1  # recurso extendido del NVIDIA device plugin: una T4 entera por pod
  limits:
    memory: 6Gi  # memoria: límite = request (sin sorpresas de OOM por vecinos)
    nvidia.com/gpu: 1  # las GPUs no se comparten ni se sobre-asignan: request = limit
    # Sin límite de CPU: el nodo es solo para Triton y el throttling de CFS mete latencia
```

- Gateway: request de 1 CPU y ningún limit de CPU, a propósito. Con limit, una ráfaga de decodes agota la cuota a mitad de la ventana de 100 ms y el p99 (la latencia del 1 % de requests más lentos) sube aunque el promedio se vea bajo. El request ya le garantiza su parte.
- Memoria del gateway: 1Gi de request y de limit, medido y no adivinado: ≈ 80 MB de Python + 2 decodes de ≈ 320 MB. La variable `PREPROCESS_CONCURRENCY=2` del Deployment acota la concurrencia para que el pico quepa.
- Triton pide 2 CPU, 6Gi y una GPU, con `nvidia.com/gpu` igual en requests y limits. Una g4dn.xlarge tiene una sola T4: un segundo Triton no cabe en ese nodo (`Insufficient nvidia.com/gpu`, sección 8). Tampoco le alcanzaría la CPU: 2 + 2 cores contra un Allocatable de algo menos de 4.
- El `/dev/shm` de Triton es un emptyDir en RAM de hasta 1Gi (líneas 120–127): cuenta dentro de los 6Gi del limit.
- Los dos pods son Burstable (CPU sin limit). Como su memoria nunca pasa del request, si al nodo le falta memoria salen después que cualquier pod que se pasó del suyo.

CÓMO RECORDARLA

request = la reserva, limit = el techo, y el scheduler solo mira reservas, nunca el uso real. Pasarte de CPU es esperar turno (throttling); pasarte de memoria es que te saquen (OOMKilled, 137).

TRAMPA

`CUDA out of memory` no es OOMKilled: OOMKilled es la RAM del contenedor contra `limits.memory` (exit 137, sin una línea en el log), mientras que el error de CUDA es la memoria de la GPU (VRAM) y aparece en el log, así que subir `limits.memory` no lo arregla. Un pod que solo pide `nvidia.com/gpu` es BestEffort: sin reserva de CPU ni de memoria, primero en salir y sin request contra el cual el HPA pueda medir. Y según el README del device plugin, un contenedor que corre en un nodo GPU sin pedir GPUs puede ver todas las del nodo.

En el curso: M6 · Los objetos que sí debes conocer · M6 · GPUs en Kubernetes · M4 · Compartir una GPU entre modelos · Docker y kubectl · Depurar pods: síntoma → comandos → causa típica · C01 · Archivos de referencia

7 ConfigMaps, Secrets, volúmenes y ServiceAccounts

¿Cómo le das a un pod su configuración, sus secretos, un lugar donde escribir y permisos de AWS, sin meter nada de eso en la imagen?

SINTAXIS CON VARIABLES

```
kubectl create configmap cm
--from-literal=CLAVE=valor
kubectl create secret generic secreto
--from-literal=CLAVE=valor
kubectl rollout restart deployment/deploy
```

Un **ConfigMap** guarda config no secreta (clave–valor); un **Secret**, lo sensible, en base64: codificado, no cifrado. Tras cambiarlos, `rollout restart` recrea los pods para que las variables tomen el valor nuevo.

```
envFrom:
  - configMapRef:
      name: cm
env:
  - name: VAR
    valueFrom:
      secretKeyRef:
        name: secreto
        key: CLAVE
```

Como variables de entorno: todas las claves del ConfigMap, o una variable desde una clave puntual. Se leen una sola vez, al arrancar el contenedor.

```
volumeMounts:
  - name: vol
    mountPath: ruta
    readOnly: true
volumes:
  - name: vol
    configMap:
      name: cm
  - name: tmp
    emptyDir:
      sizeLimit: tamaño
```

Como archivos: un volumen (carpeta montada en el contenedor) con un archivo por clave; se actualiza solo, salvo con `subPath` (montar solo una parte del volumen). El emptyDir es temporal: vive lo que el pod.

```
# en el pod
serviceAccountName: sa
# IRSA: anotación en la ServiceAccount
kubectl annotate serviceaccount sa -n ns
eks.amazonaws.com/role-
arn=arn:aws:iam::cuenta:role/rol
# EKS Pod Identity: asociación en EKS, sin anotación
aws eks create-pod-identity-association
--cluster-name cluster --namespace ns --service-
account sa --role-arn
arn:aws:iam::cuenta:role/rol
```

Una **ServiceAccount** es la identidad del pod. En EKS (el Kubernetes administrado de AWS) hay dos formas de darle un rol de IAM (permisos de AWS) sin llaves: IRSA (roles de IAM para ServiceAccounts) y EKS Pod Identity. En las dos, los pods que ya corrían deben recrearse.

VARIABLES

VARIABLE	QUÉ ES	EJEMPLO
cm	nombre del ConfigMap	`app-config (seg-worker-config en C04)`
secreto	nombre del Secret	`app-secret`
CLAVE	clave dentro del ConfigMap o del Secret; con envFrom, también es el nombre de la variable	`INFER_TIMEOUT_S`
valor	su valor; en un Secret, la API lo muestra en base64 (codificado, no cifrado)	`0.15`
VAR	nombre de la variable de entorno en el contenedor	`API_KEY`
deploy	Deployment cuyos pods hay que recrear	`gateway`
vol	nombre del volumen; une volumes con volumeMounts	`config`
ruta	carpeta donde aparecen los archivos	`/config (C04)`
tmp	nombre del volumen temporal	`tmp`
tamaño	tope del emptyDir; pasarlo desaloja el pod	`64Mi (gateway), 20Gi (C04)`
sa	ServiceAccount: la identidad del pod	`seg-worker (C04)`
ns	namespace de la ServiceAccount	`c04`
cuenta	ID de la cuenta de AWS (12 dígitos)	`123456789012`
rol	rol de IAM con solo los permisos que ese pod necesita	`c04-seg-worker`
cluster	nombre del cluster de EKS	`prod-ml (ejemplo)`

LAS PIEZAS

PIEZA	QUÉ HACE	EJEMPLO
ConfigMap	Pares clave–valor no secretos, hasta 1 MiB. Llegan como variables de entorno o como archivos.	`MODEL_NAME, INFER_TIMEOUT_S`
Secret	Como un ConfigMap, para datos sensibles. data va en base64 (codificado, no cifrado); stringData acepta texto plano. Tipo default: Opaque.	`kubectl create secret generic`
envFrom · valueFrom	Todas las claves como variables, o una variable desde una clave (configMapKeyRef, secretKeyRef) o desde un campo del pod (fieldRef).	`POD_NAME desde metadata.name (C04)`
Volumen configMap o secret	Cada clave es un archivo. Se actualiza solo (puede tardar hasta ≈ 90 s), salvo con subPath. Los de un Secret viven en RAM (tmpfs).	`/config/worker.yaml (C04)`
kubectl rollout restart	Recrea los pods de un Deployment con rolling update (de a poco): así las variables de entorno toman el valor nuevo.	`kubectl rollout restart deployment/gateway`
immutable · optional	immutable: true impide cambios (solo borrar y recrear). optional: true deja arrancar al pod aunque falte el objeto.	`configMapRef con optional: true`
Cifrado en reposo	Cifra los Secrets dentro de etcd, la base de datos del cluster. En EKS viene activo desde 2025, con KMS (el servicio de llaves de AWS).	`EKS 1.28+; quien puede leer Secrets o crear pods en el namespace igual los lee`
emptyDir	Carpeta temporal del pod: nace vacía, sobrevive a reinicios del contenedor y se borra con el pod. medium Memory: RAM.	`/tmp 64Mi (gateway), /dev/shm 1Gi (Triton)`
PVC (PersistentVolumeClaim)	Pedido de disco que sobrevive al pod. En EBS (el disco de red de AWS): un nodo a la vez y una sola zona.	`accessModes: ["ReadWriteOnce"]; la StorageClass (plantilla de disco) dice qué disco crear; EFS (archivos compartidos de AWS) sirve para ReadWriteMany`
ServiceAccount	Identidad del pod ante la API (permisos por roles, RBAC: sección 11) y, con IRSA o Pod Identity, ante AWS.	`default si no la indicas · kubectl auth can-i get secrets --as=system:serviceaccount:c04:seg-worker -n c04`
automountServiceAccountToken: false	No monta el token de la API en /var/run/secrets/kubernetes.io/serviceaccount: para pods que no hablan con Kubernetes.	`asr-worker (C13)`
IRSA (IAM roles for service accounts)	Anotas la ServiceAccount con el rol. Al crear el pod, un webhook le agrega un token; el SDK (la librería) de AWS lo canjea por credenciales temporales.	`eks.amazonaws.com/role-arn (requiere registrar en IAM el proveedor OIDC del cluster, su emisor de tokens)`
EKS Pod Identity	Asociación rol ↔ ServiceAccount guardada en EKS, sin anotación ni OIDC; un agente por nodo entrega las credenciales.	`aws eks create-pod-identity-association (AWS la recomienda cuando se puede; no funciona en Fargate, los nodos sin servidor de AWS)`
CreateContainerConfigError	Lo más común: una variable de entorno pide un ConfigMap, Secret o clave que no existe. El contenedor no se crea; el kubelet reintenta.	`configmap "app-config" not found (si falta en un volumen: ContainerCreating con FailedMount)`

PIEZA	PARA QUÉ	CÓMO LLEGA AL CONTENEDOR	¿CAMBIA SIN RECREAR EL POD?	DURA HASTA…
ConfigMap	config no secreta: URLs, timeouts, nombre del modelo	variables de entorno o archivos	env: no (rollout restart); archivo: sí, en hasta ≈ 90 s; con subPath: no	que lo borres
Secret	credenciales de terceros y contraseñas	variables de entorno o archivos en tmpfs (RAM)	igual que el ConfigMap	que lo borres

PIEZA	PARA QUÉ	CÓMO LLEGA AL CONTENEDOR	¿CAMBIA SIN RECREAR EL POD?	DURA HASTA…
emptyDir	scratch, /tmp, /dev/shm	volumen vacío al crear el pod	no aplica: lo escribe el pod	que el pod deje el nodo (sobrevive a reinicios del contenedor)
PVC	datos que tienen que sobrevivir al pod	volumen de un disco (EBS: un nodo a la vez, una zona)	no aplica	que borres el PVC (con reclaimPolicy Delete, el default, se borra el disco)
ServiceAccount	identidad ante la API de Kubernetes y, con IRSA o Pod Identity, ante AWS	token montado y variables AWS_*	el token se renueva solo; un rol nuevo exige recrear los pods	que la borres

Cinco formas de darle algo a un pod: para qué sirve cada una, cómo llega y cuánto dura.

EJEMPLO REAL · SERVICEACCOUNT CON IRSA, CONFIGMAP MONTADO Y DOS EMPTYDIR (WORKER DE C04)

YAML `reference/c04/k8s/seg-worker.yaml (líneas 15–20, 31 y 173–193; las líneas # … resumen lo omitido; sangría reducida en la parte del Deployment)`

```yaml
apiVersion: v1
kind: ServiceAccount
metadata:
  name: seg-worker
  namespace: c04
  annotations:
    # … (comentarios: los permisos mínimos del rol, SQS y S3 sin DeleteObject)
    eks.amazonaws.com/role-arn: arn:aws:iam::123456789012:role/c04-seg-worker
# … el Deployment seg-worker usa serviceAccountName: seg-worker; en su contenedor:
    volumeMounts:
      - name: scratch
        mountPath: /scratch  # el WSI descargado (0.5–3 GB) + el archivo de liveness
      - name: tmp
        mountPath: /tmp  # el root filesystem es de solo lectura
      - name: config
        mountPath: /config
        readOnly: true
volumes:
  # emptyDir vive en el disco del kubelet. Para que sea el NVMe local del g6.2xlarge (y no el EBS
  # raíz), el nodo debe montar el instance store como almacenamiento efímero (en Karpenter:
  # instanceStorePolicy: RAID0 en el EC2NodeClass). Un slide a la vez: 20 GiB sobran.
  - name: scratch
    emptyDir:
      sizeLimit: 20Gi
  - name: tmp
    emptyDir:
      sizeLimit: 1Gi
  - name: config
    configMap:
      name: seg-worker-config
```

- La ServiceAccount `seg-worker` lleva la anotación de IRSA. Cada pod que la usa recibe, por un webhook (un servicio que retoca cada pod al crearlo), `AWS_ROLE_ARN` y `AWS_WEB_IDENTITY_TOKEN_FILE`; el SDK de AWS cambia ese token por credenciales temporales del rol. No hay llaves de AWS en el YAML ni en un Secret.
- Con EKS Pod Identity sería la misma ServiceAccount sin anotación: la asociación se crea en EKS (`aws eks create-pod-identity-association`) y el rol confía en `pods.eks.amazonaws.com`. AWS la recomienda cuando se puede; los manifests del curso usan IRSA, que también funciona fuera de EKS.
- El ConfigMap `seg-worker-config` se monta de solo lectura en `/config` y el worker lee `/config/worker.yaml`. Si lo editas, el archivo cambia solo (puede tardar hasta ≈ 90 s), pero el worker tiene que volver a leerlo; una variable de entorno no cambiaría.
- Dos emptyDir: `/scratch` (20Gi para el WSI, la imagen de microscopio que baja el worker) y `/tmp` (1Gi, porque la raíz es de solo lectura). Viven lo que vive el pod y cuentan como almacenamiento efímero (el pod pide 10Gi, con limit de 30Gi). Pasarse de `sizeLimit` desaloja el pod: `Usage of EmptyDir volume "scratch" exceeds the limit "20Gi".`
- C01 no usa ConfigMaps: el gateway recibe su config como `env` literal en el Deployment (`TRITON_URL`, `INFER_TIMEOUT_S`). Cambiar un valor ahí cambia la plantilla del pod y dispara un rollout; cambiar un ConfigMap, no. Por eso Kustomize (la herramienta que arma manifests, la de kubectl apply -k) agrega un hash al nombre del ConfigMap que genera.

CÓMO RECORDARLA

ConfigMap = config, Secret = config sensible, volumen = disco, ServiceAccount = identidad. Un Secret es una carta en un sobre sin sellar: base64 no es cifrado, y quien puede leerlo la lee.

TRAMPA

creer que un Secret está cifrado porque no se lee a simple vista: `data` es base64 y `base64 --decode` lo revela; lo protegen el cifrado en reposo (en EKS, activo por defecto desde 2025) y, sobre todo, quién puede leer Secrets o crear pods en el namespace. Y nada se recarga solo: cambiar un ConfigMap no reinicia los pods (sus variables de entorno siguen con el valor viejo hasta un `kubectl rollout restart`), y anotar la ServiceAccount con un rol tampoco toca a los que ya corren.

En el curso: M15 · Secretos y credenciales · M15 · Autenticación y autorización entre servicios · M6 · Los objetos que sí debes conocer · M5 · Pesos del modelo: ¿dentro de la imagen o al arrancar? · C04 · Archivos de referencia

8 Dónde cae cada pod: nodeSelector, affinity, taints/tolerations, GPUs y nodos que se crean solos

¿Cómo decides en qué nodos puede correr cada pod, cómo apartas los nodos GPU para quien los usa, y qué pasa cuando un pod no cabe en ninguno?

SINTAXIS CON VARIABLES

```
spec:
  nodeSelector:
    label : valor
  tolerations:
    - key: clave_taint
      operator: Equal
      value: valor_taint
      effect: NoSchedule
```

El **scheduler** elige nodo para cada pod nuevo: solo usa nodos con estas labels (etiquetas). Un **taint** (marca en el nodo) rechaza pods; la **toleration** es el permiso para entrar igual.

```
kubectl label nodes nodo label=valor
kubectl taint nodes nodo
clave_taint=valor_taint:NoSchedule
kubectl taint nodes nodo
clave_taint=valor_taint:NoSchedule-
kubectl get nodes -L
label,topology.kubernetes.io/zone
```

Etiquetar, poner y quitar un taint (el guion final lo quita), y ver labels como columnas. En EKS los fija el node group (grupo de máquinas iguales) o el NodePool de Karpenter (un autoscaler de nodos).

```
affinity:
  nodeAffinity:
    requiredDuringSchedulingIgnoredDuringExecution:
      nodeSelectorTerms:
        - matchExpressions:
            - key: label
              operator: In
              values: ["valor", "otro_valor"]
```

Node affinity (afinidad de nodo): como nodeSelector, con operadores (In, NotIn, Exists, DoesNotExist, Gt, Lt) y una versión suave, `preferredDuringSchedulingIgnoredDuringExecution`, con peso de 1 a 100.

```
topologySpreadConstraints:
  - maxSkew: sesgo
    topologyKey: topology.kubernetes.io/zone
    whenUnsatisfiable: si_no_cabe
    labelSelector:
      matchLabels:
        app.kubernetes.io/name: app
```

Reparte las réplicas entre zonas (datacenters separados de una región) o entre nodos. Si no se puede, `DoNotSchedule` deja el pod Pending; `ScheduleAnyway` lo pone igual, con un reparto imperfecto.

VARIABLES

VARIABLE	QUÉ ES	EJEMPLO
label	clave de la label del nodo	`workload`
valor	valor de la label	`gpu-t4`
otro_valor	otro valor aceptado: In acepta una lista	`us-east-1b, si la clave es topology.kubernetes.io/zone y el primer valor us-east-1a`
clave_taint	clave del taint; para GPUs, el nombre del recurso	`nvidia.com/gpu`
valor_taint	valor del taint	`present`
nodo	nombre del nodo, de kubectl get nodes	`docker-desktop en tu Mac`
sesgo	maxSkew: diferencia máxima de réplicas entre zonas	`1`
si_no_cabe	DoNotSchedule (default: queda Pending) o ScheduleAnyway (lo pone igual)	`DoNotSchedule (Triton), ScheduleAnyway (gateway)`
app	label de las réplicas que se cuentan	`triton`
n_{destino}	réplicas que ya hay en la zona candidata	`1 en us-east-1a`
n_z	réplicas en cada zona elegible; cuenta la menor	`0 en us-east-1b`
maxSkew	desbalance permitido	`1: la 2.ª réplica de Triton solo puede ir a 1b`
GPUs del nodo	nvidia.com/gpu en Allocatable (lo que el nodo ofrece a los pods)	`1 en una g4dn.xlarge; 4 en una g4dn.12xlarge`
GPUs por pod	nvidia.com/gpu en limits	`1 (Triton)`

FÓRMULAS

$$n_{\text{destino}} + 1 - \min_z n_z \le \text{maxSkew}$$

$$\text{pods por nodo} = \left\lfloor \frac{\text{GPUs del nodo}}{\text{GPUs por pod}} \right\rfloor$$

LAS PIEZAS

PIEZA	QUÉ HACE	EJEMPLO
Scheduler (kube-scheduler)	Elige nodo para cada pod nuevo: descarta los que no sirven y puntúa el resto. No mueve pods ya puestos.	`evento Scheduled: Successfully assigned …`
Labels de nodo	Etiquetas clave=valor. Kubernetes y la nube ponen algunas; tú agregas las tuyas.	`topology.kubernetes.io/zone, node.kubernetes.io/instance-type, workload=gpu-t4`
nodeSelector	Solo nodos con todas estas labels. Lo más simple para fijar un tipo de nodo.	`workload: gpu-t4`
nodeAffinity	nodeSelector con operadores: required (obligatorio) o preferred (con peso). IgnoredDuringExecution: si la label cambia después, el pod sigue donde está.	`operator: In`
podAffinity · podAntiAffinity	Reglas respecto de otros pods: juntos o separados por nodo o por zona (topologyKey).	`no dos réplicas en el mismo nodo`
topologySpreadConstraints	Reparte réplicas entre zonas o nodos con un desbalance máximo (maxSkew).	`Triton: una por zona`
Taint	Marca del nodo que rechaza pods sin toleration. NoSchedule; PreferNoSchedule, suave; NoExecute, además saca a los que corren.	`nvidia.com/gpu=present:NoSchedule`
Toleration	Permiso del pod para entrar a un nodo con ese taint. No lo obliga a ir ahí.	`key: nvidia.com/gpu, operator: Equal, value: present`
ExtendedResourceToleration	Admission controller (filtro del API server) activo en EKS: a los pods que piden GPU les agrega por su cuenta la toleration nvidia.com/gpu (Exists).	`funciona porque el taint usa el nombre del recurso`
Taints automáticos	not-ready y unreachable (NoExecute): por defecto el pod los tolera 300 s y después se desaloja.	`node.kubernetes.io/unreachable`
PriorityClass y preemption	Prioridad del pod. Si no cabe, el scheduler puede desalojar pods de menor prioridad para hacerle lugar.	`placeholder de prioridad −10 que reserva un nodo GPU (M06)`
FailedScheduling	Evento con el motivo de cada nodo descartado. El pod queda Pending, con PodScheduled=False y reason Unschedulable.	`0/6 nodes are available: …`
Karpenter · Cluster Autoscaler	Ven los pods Unschedulable y agregan un nodo que los satisfaga (label, taint, GPU, zona); lo quitan cuando sobra.	`NodePool de T4 con la label workload: gpu-t4 y el taint`
nodeName	Asigna el nodo a mano y se salta el scheduler, incluidos los taints NoSchedule. Casi nunca lo quieres.	`—`

PODS ↓ · NODOS →	**cpu-1a** c6i.xlarge · us-east-1a	**cpu-1b** c6i.xlarge · us-east-1b	**gpu-1a** g4dn.xlarge · us-east-1a LABEL workload=gpu-t4 TAINT nvidia.com/gpu=present:NoSchedule GPU ocupada ya corre la 1.ª réplica de Triton	**gpu-1b** g4dn.xlarge · us-east-1b LABEL workload=gpu-t4 TAINT nvidia.com/gpu=present:NoSchedule 1 GPU libre
gateway NODESELECTOR *ninguno* TOLERATION *ninguna* PIDE cpu 1, memory 1Gi	✓ cabe	✓ cabe	✗ untolerated taint	✗ untolerated taint
triton (2.ª réplica) NODESELECTOR workload=gpu-t4 TOLERATION nvidia.com/gpu=present:NoSchedule PIDE cpu 2, memory 6Gi, nvidia.com/gpu 1 SPREAD maxSkew 1 por zona	✗ no tiene la label	✗ no tiene la label	✗ Insufficient cpu y nvidia.com/gpu	✓ cabe
triton (3.ª réplica) NODESELECTOR workload=gpu-t4 TOLERATION nvidia.com/gpu=present:NoSchedule PIDE cpu 2, memory 6Gi, nvidia.com/gpu 1 SPREAD maxSkew 1 por zona	✗ no tiene la label	✗ no tiene la label	✗ Insufficient cpu y nvidia.com/gpu	✗ Insufficient cpu y nvidia.com/gpu (ya está la 2.ª)

✓ el scheduler puede ponerlo ahí ✗ el motivo que verías en FailedScheduling nodo GPU

Y DESPUÉS: La 3.ª réplica queda Pending. Karpenter lanza una g4dn.xlarge con la label y el taint de su NodePool; el spread la acepta en cualquiera de las dos zonas (1 + 1 − 1 = 1).

Cuatro nodos de C01 en dos zonas y tres pods. ✓ = el scheduler puede ponerlo ahí; ✗ = el motivo que verías en FailedScheduling. La 3.ª réplica de Triton no cabe en ningún nodo: es la que despierta a Karpenter.

PENDING: QUÉ DICE EL FAILEDSCHEDULING Y QUÉ HACES

MENSAJE (PARTE CENTRAL)	QUÉ SIGNIFICA	QUÉ HACES
Insufficient nvidia.com/gpu	Ningún nodo tiene una GPU libre, o ninguno la anuncia	Espera al autoscaler de nodos o revisa su límite, la cuota y la capacidad de la zona; si el nodo GPU existe y no la muestra en Allocatable, revisa el device plugin (el pod que anuncia las GPUs del nodo)
node(s) didn't match Pod's node affinity/selector	Ningún nodo tiene la label que pide el pod	Corrige la label en el pod o en el NodePool o node group (un typo es lo más común)
node(s) had untolerated taint(s)	Nodos con un taint que el pod no tolera. Desde 1.35 (y 1.34.2) el mensaje no dice cuál; antes: node(s) had untolerated taint {nvidia.com/gpu: present}	Agrega la toleration, o mira el taint con kubectl describe node (bloque Taints)
node(s) didn't match pod topology spread constraints	Ponerlo ahí rompería el maxSkew	Capacidad en la otra zona, o ScheduleAnyway
Insufficient cpu · Insufficient memory	Los requests ya llenan los nodos, aunque el uso real sea bajo	Requests realistas o más nodos

Formato: 0/6 nodes are available: 2 Insufficient nvidia.com/gpu, 4 node(s) didn't match Pod's node affinity/selector. preemption: … El número delante de cada motivo es cuántos nodos lo dieron.

EJEMPLO REAL · CÓMO C01 LLEVA A TRITON A SUS NODOS GPU Y LO REPARTE ENTRE ZONAS

YAML `reference/c01/k8s/triton.yaml (líneas 33–54, spec del template; sangría reducida)`

```yaml
nodeSelector:
  workload: gpu-t4  # solo el node group de g4dn.xlarge (T4, sm_75): el engine se compiló para esa GPU
tolerations:
  # Los nodos GPU tienen el taint nvidia.com/gpu=present:NoSchedule para que ningún pod CPU
  # (gateway, sistema) ocupe una máquina cara; Triton lo tolera explícitamente.
  - key: nvidia.com/gpu
    operator: Equal
    value: present
    effect: NoSchedule
topologySpreadConstraints:
  # Réplicas balanceadas entre zonas (con 2: una por zona). Si cae una AZ, la otra GPU sigue sirviendo (N+1).
  # DoNotSchedule: preferimos un pod Pending (y una alerta) a dos GPUs en la misma AZ.
  # Ojo: con nodeTaintsPolicy por defecto (Ignore), los nodos NotReady de una AZ caída siguen contando
  # como dominio con 0 pods: la zona sana no pasa de 1 réplica y KEDA no puede escalar durante la caída.
  # Aquí es aceptable (1 T4 aguanta el pico de diseño al ≈ 42 %). nodeTaintsPolicy: Honor lo evita, pero
  # un taint transitorio (nodo arrancando o drenando) puede dejar las dos GPUs en la misma AZ: pruébalo en un game day.
  - maxSkew: 1
    topologyKey: topology.kubernetes.io/zone
    whenUnsatisfiable: DoNotSchedule
    labelSelector:
      matchLabels:
        app.kubernetes.io/name: triton
```

- `nodeSelector: workload: gpu-t4` fija el tipo de GPU: el engine de TensorRT se compiló para la T4 (su arquitectura, sm_75) y en otra GPU no carga. Pedir `nvidia.com/gpu` solo asegura un nodo con alguna GPU libre.
- La toleration coincide con el taint `nvidia.com/gpu=present:NoSchedule` de los nodos GPU (`Equal` y el mismo valor). El taint aleja al gateway y a todo pod sin esa toleration (los DaemonSets del sistema, que ponen un pod en cada nodo, como el device plugin, suelen traerla): `gateway.yaml` no tiene toleration, así que nunca cae en una máquina cara.
- En EKS está activo el admission controller (un paso del API server que revisa o modifica cada objeto al crearlo o cambiarlo) `ExtendedResourceToleration`: a cualquier pod que pida `nvidia.com/gpu` le agrega una toleration `nvidia.com/gpu` con `Exists`, porque el taint usa el nombre del recurso como clave. C01 la escribe igual: queda a la vista y funciona en clusters sin ese admission controller.
- El spread con `maxSkew: 1` y `DoNotSchedule`: con 1 Triton en 1a y 0 en 1b, el segundo solo puede ir a 1b (1 + 1 − 0 = 2 en 1a no cumple; 0 + 1 − 0 = 1 en 1b sí). Si en 1b no hay GPU libre, espera Pending a que llegue un nodo ahí.
- Los comentarios marcan la letra chica: con `nodeTaintsPolicy` por defecto (`Ignore`), los nodos caídos de una zona siguen contando con 0 réplicas, así que durante esa caída la zona sana no pasa de una.

CÓMO RECORDARLA

label + nodeSelector atrae; el taint repele y la toleration deja pasar. El nodeSelector es buscar la puerta con cierto letrero; el taint es un control en la puerta que solo deja pasar con pase, y la toleration es el pase: te deja entrar, no te obliga a ir; en nodos GPU usas los dos.

TRAMPA

leer Pending como "faltan nodos" sin leer el mensaje. `node(s) didn't match Pod's node affinity/selector` suele ser un typo en la label, y si ningún NodePool o node group pone esa label, ningún autoscaler va a crear un nodo para ese pod: espera para siempre. Y desde 1.35 (y 1.34.2) el mensaje del taint ya no dice cuál es: míralo en el nodo con `kubectl describe node`.

En el curso: M6 · GPUs en Kubernetes · M6 · Aprovisionar nodos GPU: Cluster Autoscaler y Karpenter · M6 · Alta disponibilidad: multi-AZ, PDB y N+1 · Docker y kubectl · Depurar pods: síntoma → comandos → causa típica · Docker y kubectl · Autoscaling y capacidad

9 Autoscaling: HPA y KEDA

¿Quién decide cuántas réplicas corren, con qué métrica, y qué pasa cuando las réplicas nuevas no tienen nodo donde caer?

SINTAXIS CON VARIABLES

```
apiVersion: autoscaling/v2
kind: HorizontalPodAutoscaler
metadata:
  name: nombre
spec:
  scaleTargetRef:
    apiVersion: apps/v1
    kind: Deployment
    name: deployment
  minReplicas: min
  maxReplicas: max
  metrics:
    - type: Resource
      resource:
        name: cpu
        target:
          type: Utilization
          averageUtilization: porcentaje
```

Un **HPA** (HorizontalPodAutoscaler) revisa una métrica cada 15 s y ajusta cuántas réplicas tiene el Deployment. Con `Utilization`, el objetivo es un % del request de CPU: sin request, no escala.

```
behavior:
  scaleUp:
    stabilizationWindowSeconds: 0
    policies:
      - type: Pods
        value: pods_por_periodo
        periodSeconds: periodo
  scaleDown:
    stabilizationWindowSeconds: ventana
```

La velocidad: cuánto puede subir por período y cuánto espera antes de bajar. Si escribes `policies`, reemplazan a las del default.

```
apiVersion: keda.sh/v1alpha1
kind: ScaledObject
metadata:
  name: nombre
spec:
  scaleTargetRef:
    name: deployment
  minReplicaCount: min
  maxReplicaCount: max
  triggers:
    - type: prometheus
      metricType: Value
      metadata:
        serverAddress: url_prometheus
        query: query
        threshold: "umbral"
```

KEDA (Kubernetes Event-driven Autoscaling) es un operador (un controlador que instalas). Con este ScaledObject crea el HPA `keda-hpa-` + nombre y le sirve la métrica. Escribe `minReplicaCount`: su default es 0.

VARIABLES

VARIABLE	QUÉ ES	EJEMPLO
nombre	nombre del HPA o del ScaledObject	gateway · triton
deployment	Deployment que escala	gateway · triton
min	piso de réplicas	3 (gateway), 2 (Triton)
max	techo de réplicas	12 (gateway), 4 (Triton)
porcentaje	objetivo: % del request de CPU, promedio de los pods	60
pods_por_periodo	pods que puede agregar como mucho en cada período	1 (Triton)
periodo	largo del período de la política, en s (máximo 1800)	60
ventana	ventana de estabilización al bajar, en s (default 300)	300 (gateway), 600 (Triton)
url_prometheus	dirección del Prometheus (la base de métricas) del cluster	http://prometheus-operated.monitoring.svc.cluster.local:9090
query	consulta en PromQL (el lenguaje de Prometheus) que devuelve un solo número	espera media en la cola de Triton, en ms
umbral	objetivo de la métrica; en KEDA va como texto	10
ns	namespace	classifier
deseadas	réplicas que pide el HPA; después las recortan min, max y behavior	8 (gateway); 5 → 4 (Triton)
actuales	réplicas actuales; con métricas Value de KEDA (External), las Ready	5 (gateway); 2 (Triton)
métrica	valor actual: % de CPU promedio o lo que devuelve la query	90 %; 25 ms
objetivo	el target del HPA o el threshold de KEDA	60 %; 10 ms
métrica total	suma de la métrica en todos los pods (mensajes en cola, req/s)	120 mensajes en una cola de SQS (M06)
objetivo por pod	cuánto de esa métrica le toca a cada réplica	15 → [120 / 15] = 8 workers

```
kubectl get hpa -n ns -w
kubectl describe hpa nombre -n ns
kubectl get scaledobject -n ns
kubectl get hpa keda-hpa-nombre -n ns
```

TARGETS es actual/objetivo, como `cpu: 45%/60%`; si donde va el número dice unknown, no llega la métrica. `describe` muestra las condiciones (`ScalingActive`, `ScalingLimited`) y los eventos.

VARIABLE	QUÉ ES	EJEMPLO
t_{servir}	de que la carga sube a que el pod nuevo atiende	≈ 5–13 min en C01 (M06)
t_{detectar}	Prometheus lee la métrica (scrape) + ventana de la query + ciclo del HPA	≈ 0.5–1.5 min
t_{nodo}	lanzar la instancia, unirse al cluster y que el device plugin (el pod que anuncia las GPUs del nodo) anuncie la GPU	≈ 2–5 min
t_{imagen}	descargar la imagen de Triton (≈ 9 GB)	≈ 2–5 min
t_{modelo}	arrancar Triton, cargar el engine y pasar la readiness	≈ 15–75 s

FÓRMULAS

$$\text{deseadas} = \left\lceil \text{actuales} \times \frac{\text{métrica}}{\text{objetivo}} \right\rceil$$

$$\left| \frac{\text{métrica}}{\text{objetivo}} - 1 \right| \leq 0.1 \Rightarrow \text{no cambia}$$

$$\text{deseadas} = \left\lceil \frac{\text{métrica total}}{\text{objetivo por pod}} \right\rceil \quad \text{(AverageValue)}$$

$$t_{\text{servir}} \approx t_{\text{detectar}} + t_{\text{nodo}} + t_{\text{imagen}} + t_{\text{modelo}}$$

LAS PIEZAS

PIEZA	QUÉ HACE	EJEMPLO
HPA (HorizontalPodAutoscaler)	Controlador que cada 15 s calcula réplicas con una métrica y cambia replicas del Deployment.	`kubectl autoscale deployment gateway --min=3 --max=12 --cpu=60%` (kubectl ≥ 1.34; antes, `--cpu-percent=60`)
minReplicas · maxReplicas	Piso y techo. El HPA los impone aunque no consiga la métrica.	3–12 (gateway), 2–4 (Triton)
Resource + Utilization	Uso de CPU como % del request, promedio de los pods. Sin request, el HPA no puede medir ni actuar.	`averageUtilization: 60`
Pods · Object · External	Métricas que no son CPU ni memoria: por pod, de un objeto del cluster o de afuera (colas, queries).	KEDA publica External
metrics-server	Publica CPU y memoria (la API metrics.k8s.io). Sin él: TARGETS cpu: <unknown>/60% y kubectl top no funciona.	Docker Desktop no lo trae
behavior: policies	Cuánto puede cambiar por período, en pods o en %. Si hay varias, gana la que permite más (selectPolicy: Max).	Triton: +1 pod cada 60 s
stabilizationWindowSeconds	Al bajar, usa la recomendación más alta de la ventana (default 300 s): baja despacio, sin oscilar. Al subir: 0.	600 en Triton
Tolerancia	Si métrica / objetivo queda entre 0.9 y 1.1, no hace nada. Por HPA: behavior.scaleUp.tolerance y scaleDown.tolerance (estable desde 1.37).	64 % contra 60 %: sin cambio
ScaledObject (KEDA)	Le dices a KEDA qué escalar y con qué evento; KEDA crea y alimenta un HPA. No agregues otro.	keda-hpa-triton (keda-hpa- + el nombre)
metricType: Value · AverageValue	Value: proporcional a las réplicas Ready (tiempos). AverageValue: total ÷ objetivo por pod (colas, req/s).	Triton: Value, umbral 10 ms

PIEZA	QUÉ HACE	EJEMPLO
minReplicaCount · cooldownPeriod · pollingInterval	Defaults 0, 300 s y 30 s. KEDA decide 0 ⇔ 1; el HPA, 1 ⇔ N. Escribe el mínimo.	`2 en Triton: nunca baja a 0`
ignoreNullValues	Con "false", una query vacía es un error visible; con el default (true), un 0 silencioso.	`"false" en C01`
Karpenter · Cluster Autoscaler	Ven pods Pending por falta de lugar y agregan nodos; los quitan cuando sobran.	`NodePool (grupo de nodos de Karpenter) de T4; Cluster Autoscaler quita un nodo tras 10 min sin hacer falta`
kubectl scale con autoscaler	Dura poco: el HPA vuelve a su número (si subiste, al ritmo de su scaleDown). Cambia min/max en git.	`--replicas=6 con máximo 4 → vuelve a 4 en ≈ 15 s`

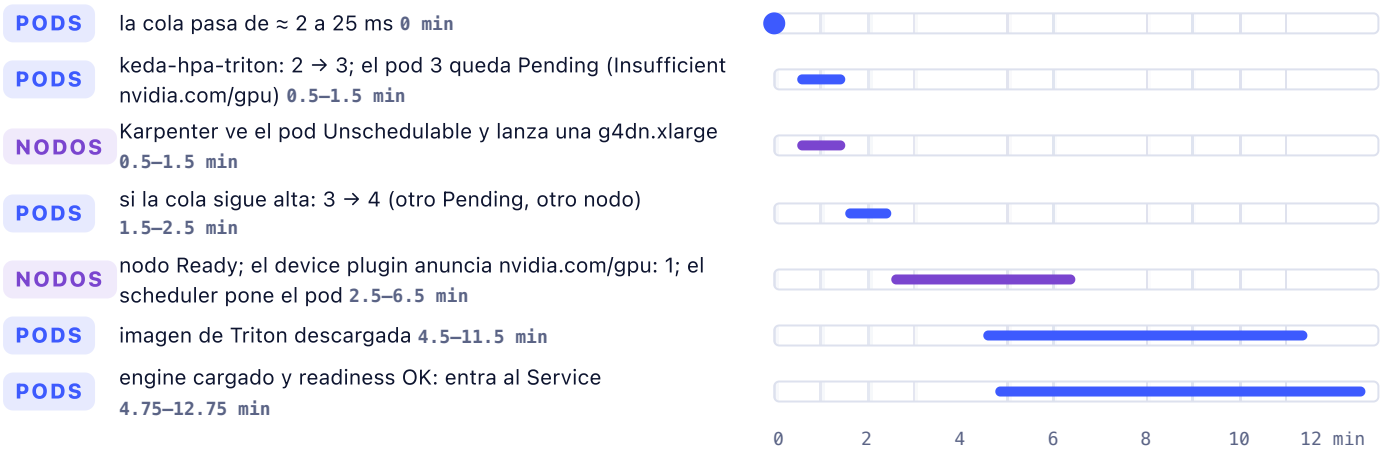
MÉTRICA: espera media en la cola de Triton (ms) · `metricType: Value`



1. ⌈2 × 25 / 10⌉ = 5
2. recortado al máximo: 4
3. behavior: +1 pod cada 60 s → 3, después 4

réplica Ready · la pide el HPA · no entra: pasa el máximo

LÍNEA DE TIEMPO, EN MINUTOS: CADA BARRA VA DEL INICIO AL FIN DEL RANGO



AL BAJAR: el HPA espera 600 s y quita 1 pod cada 300 s (borrar réplicas no pasa por el PodDisruptionBudget); el nodo que queda vacío lo retira Karpenter (consolidación: quita o cambia nodos que sobran) o Cluster Autoscaler (tras 10 min sin hacer falta).

KEDA y el HPA deciden en segundos; el nodo GPU, la imagen y el modelo tardan minutos. Tiempos de M06 para C01, sin nodo libre, con el inicio y el fin de cada rango.

QUIÉN DECIDE QUÉ, Y CUÁNTO TARDA EN C01

PIEZA	DECIDE	CON QUÉ	TIEMPO EN C01
KEDA (operador)	Crear el HPA; pasar de 0 a 1 y de 1 a 0 réplicas	Los triggers del ScaledObject	No aplica: el mínimo es 2
HPA keda-hpa-triton	Cuántas réplicas entre el mínimo y el máximo (2–4)	⌈Ready × métrica / umbral⌉ y el behavior	Cada 15 s; +1 pod por minuto
Scheduler	En qué nodo cae cada pod nuevo	Requests, labels, taints y el reparto entre zonas	Segundos; si no cabe, Pending
Karpenter (o Cluster Autoscaler)	Cuántos nodos	Pods Unschedulable y las reglas del NodePool	≈ 2–5 min por nodo GPU
kubelet, device plugin y Triton	Cuándo sirve el pod nuevo	Pull de la imagen, carga del engine, readiness	≈ 2–5 min + ≈ 15–75 s

Del primer síntoma al primer request del pod nuevo: ≈ 5–13 min. Por eso C01 nunca baja de 2 réplicas (una por zona, y cada una aguanta sola el pico de diseño), y los picos cortos los absorben ese margen y el load shedding (rechazar rápido, con un 503, lo que no

cabe), no el autoscaling.

EJEMPLO REAL · EL SCALEDOBJECT DE TRITON: KEDA ALIMENTA UN HPA CON EL TIEMPO EN COLA

YAML **reference/c01/k8s/keda-triton.yaml (líneas 15–16 y 36–60; las líneas # … resumen lo omitido)**

```yaml
# … ScaledObject "triton" (namespace classifier) → Deployment triton
  minReplicaCount: 2  # nunca menos de 2: una GPU por AZ (N+1), aunque 1 sola cubra el pico
  maxReplicaCount: 4  # techo de costo: 4 × g4dn.xlarge; si no alcanza, TritonQueueTimeHigh avisa
  # … pollingInterval, cooldownPeriod y el behavior del HPA (ver notas)
  triggers:
    - type: prometheus
      # Value (no AverageValue): réplicas deseadas = ceil(actuales × valor / umbral). Proporcional,
      # como corresponde a una latencia; AverageValue dividiría el valor entre réplicas, lo que tiene
      # sentido para un caudal (req/s), no para un tiempo.
      metricType: Value
      metadata:
        # Prometheus del cluster (kube-prometheus-stack). Con Amazon Managed Prometheus se agrega
        # autenticación SigV4 vía TriggerAuthentication.
        serverAddress: http://prometheus-operated.monitoring.svc.cluster.local:9090
        # Si la query vuelve vacía (Prometheus no scrapea Triton: falta el ServiceMonitor de
        # k8s/monitoring.yaml), "false" la vuelve un error visible del scaler (eventos del ScaledObject).
        # Con el default (true) KEDA la toma como 0 y Triton se queda en 2 réplicas sin que nada avise.
        ignoreNullValues: "false"
        # Espera media en cola en ms (misma fórmula que la alerta TritonQueueTimeHigh).
        # clamp_min evita 0/0 = NaN sin tráfico; si todo falla (0 éxitos) el valor se dispara y escala.
        query: |
          sum(rate(nv_inference_queue_duration_us{model="classifier"}[2m]))
            /
          clamp_min(sum(rate(nv_inference_request_success{model="classifier"}[2m])), 1e-9)
            / 1000
        # Objetivo: 10 ms en cola. Normal con max_queue_delay de 2 ms es ≈ 1–3 ms; > 10 ms es saturación.
        # Ojo: la cola es no lineal (casi 0 hasta que la GPU se satura). Alternativa más estable:
        # AverageValue sobre sum(rate(nv_inference_count[1m])) con umbral ≈ 400 img/s por réplica.
        threshold: "10"
```

- Con este ScaledObject, KEDA crea el HPA `keda-hpa-triton` (mín. 2, máx. 4) con una métrica External. Cada 15 s el HPA le pide el número al servidor de métricas de KEDA, que corre la query en Prometheus; con mínimo ≥ 1, `pollingInterval` solo refresca el estado del ScaledObject.
- `metricType: Value` : deseadas = ⌈Ready × ms / 10⌉. Normal, ≈ 2 ms: ⌈2 × 0.2⌉ = 1, y se queda en 2 (el mínimo). Saturada, 25 ms: ⌈2 × 2.5⌉ = 5, y se queda en 4 (el máximo). `AverageValue` dividiría la métrica entre las réplicas: sirve para colas y req/s, no para un tiempo.
- El behavior omitido (`advanced.horizontalPodAutoscalerConfig`) sube +1 pod cada 60 s y baja −1 cada 300 s, después de 600 s de estabilización: de 2 a 4 réplicas pasa al menos un minuto, más el nodo de cada una. `cooldownPeriod` solo aplica al bajar a 0, que aquí nunca pasa.
- `ignoreNullValues: "false"` : si falta el ServiceMonitor (el objeto que le dice a Prometheus qué leer) y la query vuelve vacía, el scaler da error (`prometheus metrics 'prometheus' target may be lost, the result is empty`) y lo ves en los eventos del ScaledObject. Con el default (`true`) sería un 0 silencioso y Triton nunca pasaría de 2.
- El gateway usa un HPA normal (`gateway.yaml`): CPU al 60 % de su request de 1 core. 5 pods al 90 % → ⌈5 × 90 / 60⌉ = ⌈7.5⌉ = 8. Sube hasta +100 % cada 30 s y baja de a 1 pod por minuto después de 300 s.
- Los dos lazos se encadenan así: el HPA pide la 3.ª réplica, no hay T4 libre y el pod queda Pending (`Insufficient nvidia.com/gpu`). Karpenter lo ve Unschedulable y lanza una g4dn.xlarge con la label y el taint de su NodePool; el device plugin anuncia la GPU y recién ahí el scheduler pone el pod. De Pending a Ready, ≈ 4–11 min según el comentario de `progressDeadlineSeconds` en `triton.yaml` .

CÓMO RECORDARLA

el Deployment es un termostato (mantiene N copias); el HPA mueve la perilla N con una regla proporcional; KEDA le acerca métricas de afuera; Karpenter agrega nodos cuando las copias no caben. Pods en segundos, nodos en minutos.

TRAMPA

fijar réplicas a mano con un autoscaler activo: un `kubectl scale` no dura (el HPA vuelve a su número) y, si dejas `replicas:` en el YAML, cada `kubectl apply` resetea la escala y pelea con el autoscaler; por eso los Deployments de C01 no lo tienen. Para cambiar capacidad, cambia `minReplicas` o `minReplicaCount` en git. Y `maxReplicaCount: 20` no te da 20 GPUs: cada réplica necesita un nodo, cuota y capacidad en la zona.

En el curso: M6 · HPA: por qué escalar por CPU no sirve para un servicio de GPU · M6 · KEDA: escalar por eventos y hasta cero · M6 · Aprovisionar nodos GPU: Cluster Autoscaler y Karpenter · M6 · Cold starts de punta a punta · Docker y kubectl · Autoscaling y capacidad · C01 · Serving

10 Rollouts, rollback, PodDisruptionBudget y canary

¿Cómo cambias de versión sin cortar el servicio, cómo vuelves atrás y qué te protege cuando alguien vacía un nodo?

SINTAXIS CON VARIABLES

`kubectl rollout status deploy/` *nombre* **`-n`** *ns* **`--timeout=`** *plazo*

Un **rollout** es el cambio de versión de un Deployment: entran pods nuevos y salen los viejos, poco a poco. Este comando espera a que termine y sale con error si se traba.

`kubectl rollout history deploy/` *nombre* **`-n`** *ns*
`kubectl rollout undo deploy/` *nombre* **`-n`** *ns* **`--to-revision=`** *n*

Cada cambio del template (el molde de los pods) crea una **revisión**: un ReplicaSet nuevo, y los viejos quedan guardados en 0 réplicas. `undo` vuelve a una revisión; sin `--to-revision`, a la anterior.

```
apiVersion: policy/v1
kind: PodDisruptionBudget
metadata: { name: nombre, namespace: ns }
spec:
  minAvailable: disponibles
  selector: { matchLabels: { app.kubernetes.io/name: app } }
```

Un **PodDisruptionBudget** (PDB) dice cuántos pods deben seguir listos cuando alguien vacía un nodo a propósito (un **drain**: sacarle los pods para apagarlo o actualizarlo). Usa `minAvailable` o `maxUnavailable`, nunca los dos.

```
strategy:
  canary:
    steps:
      - setWeight: peso
      - pause: { duration: espera }
```

Argo Rollouts es un controlador que se instala aparte. Con él, un **canary** manda un porcentaje chico del tráfico a la versión nueva y espera antes de subirlo. `pause: {}` espera hasta que alguien lo promueva.

VARIABLES

VARIABLE	QUÉ ES	EJEMPLO
nombre	nombre del Deployment o del PDB	`triton, gateway`
ns	namespace: la carpeta lógica donde vive el objeto	`classifier`
plazo	cuánto espera el comando antes de rendirse (no cambia el Deployment)	`25m (Triton: nodo GPU, imagen y engine, un pod a la vez)`
n	número de revisión, sacado de rollout history	`3`
disponibles	pods que deben quedar listos (número o %)	`1`
app	label que comparten los pods que proteges	`triton`
peso	porcentaje del tráfico para la versión nueva	`5, después 25 (el canary de M11)`
espera	cuánto se queda en ese porcentaje	`4h`
r	réplicas deseadas del Deployment	`2 en Triton`
maxSurge	pods de más durante el rollout; si es %, se redondea hacia arriba	`1 en Triton (default 25 %)`
maxUnavailable	pods de menos durante el rollout; si es %, se redondea hacia abajo	`0 en Triton (default 25 %)`
sanos	pods del PDB con la condición Ready en True	`2`
minAvailable	el mínimo que pide el PDB	`1 → 1 eviction permitida`

FÓRMULAS

$$\text{pods}_{\text{máx}} = r + \lceil r \cdot \text{maxSurge} \rceil \qquad \text{listos}_{\text{mín}} = r - \lceil r \cdot \text{maxUnavailable} \rceil$$

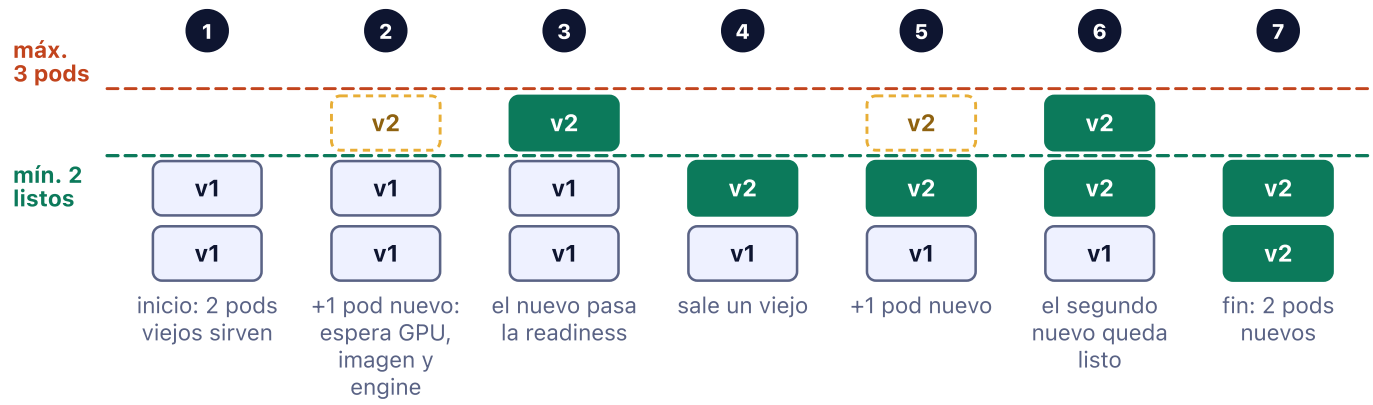
$$\text{evicciones permitidas} = \max(0,\ \text{sanos} - \text{minAvailable})$$

LAS PIEZAS

PIEZA	QUÉ HACE	EJEMPLO
`kubectl rollout status`	Espera el fin del rollout. Sale con error si vence el plazo del Deployment o --timeout.	`deploy/triton --timeout=25m`
`kubectl rollout history`	Lista las revisiones guardadas; con --revision=N muestra el template de una.	`REVISION · CHANGE-CAUSE`
`kubectl rollout undo`	Vuelve a la revisión anterior (o a --to-revision=N) creando una revisión nueva con el template viejo.	`deployment.apps/triton rolled back`

PIEZA	QUÉ HACE	EJEMPLO
`kubectl rollout restart`	Recrea todos los pods con la misma imagen, de a poco (rolling update).	releer un Secret que se lee al arrancar
`kubectl rollout pause · resume`	Congela el Deployment: los cambios no arrancan rollout hasta resume. En pausa no hay undo.	juntar varios cambios en uno
`kubernetes.io/change-cause`	Anotación del Deployment que se copia a la revisión: llena la columna CHANGE-CAUSE.	"classifier 2026-09-15-resnet50-fp16"
`revisionHistoryLimit`	Cuántos ReplicaSets viejos guarda para volver atrás. Default 10; con 0 no hay undo.	5 en C01
`progressDeadlineSeconds`	Plazo para avanzar. Si vence: Progressing=False con ProgressDeadlineExceeded. Default 600 s. No revierte solo.	900 en Triton
`PodDisruptionBudget`	Mínimo de pods listos cuando alguien desaloja pods a propósito: drain, upgrade del cluster o un autoscaler que apaga nodos.	minAvailable: 1 en Triton
`unhealthyPodEvictionPolicy: AlwaysAllow`	Deja desalojar pods que corren pero no están Ready, aunque el PDB esté al límite. El default (IfHealthyBudget) no.	un pod en CrashLoopBackOff que traba un drain
`kubectl drain NODO`	Marca el nodo para que no reciba pods nuevos y desaloja los que tiene, respetando los PDB.	--ignore-daemonsets
`Rollout (Argo Rollouts)`	Recurso que reemplaza al Deployment y agrega pasos de canary: setWeight, pause, analysis.	el gateway de C01 en M11
`AnalysisTemplate`	Consulta métricas (Prometheus) durante el canary y lo aborta si un criterio falla.	5xx > 0.5 % aborta
`kubectl argo rollouts promote · abort`	promote avanza un canary en pausa; abort lo detiene y vuelve a la versión estable.	promote gateway -n classifier

r = 2 réplicas **maxSurge 1** → nunca más de 3 pods **maxUnavailable 0** → nunca menos de 2 listos



v1 versión vieja (sirve) v2 nueva, todavía no Ready v2 nueva y Ready mínimo de listos máximo de pods

Triton en C01: 2 réplicas, maxSurge 1, maxUnavailable 0. Nunca hay menos de 2 pods listos ni más de 3 en total. Si el pod nuevo no llega a Ready, todo se queda en el paso 2: los viejos siguen sirviendo, a los 900 s aparece ProgressDeadlineExceeded y volver es kubectl rollout undo.

EJEMPLO REAL · C01: CÓMO CAMBIA DE VERSIÓN TRITON Y QUÉ LO PROTEGE EN UN DRAIN

YAML `reference/c01/k8s/triton.yaml (extracto: estrategia del Deployment y PodDisruptionBudget; # … marca líneas omitidas)`

```yaml
apiVersion: apps/v1
kind: Deployment
metadata:
  name: triton  # kube_deployment_status_replicas_available{deployment="triton"} → alerta TritonNotReady
  namespace: classifier
  # …
spec:
  # Sin "replicas": las gestiona el HPA que crea KEDA. Si lo fijas aquí, cada `kubectl apply`
  # resetea la escala y pelea con el autoscaler.
```

```yaml
  revisionHistoryLimit: 5  # ReplicaSets viejos para `kubectl rollout undo`
  progressDeadlineSeconds: 900  # un pod con nodo GPU nuevo tarda ≈ 4–11 min: el default de 600 s vence en el caso pesimista
  strategy:
    type: RollingUpdate
    rollingUpdate:
      maxSurge: 1  # de a un pod nuevo: cada pod necesita una GPU libre (Karpenter/cluster-autoscaler agrega el nodo)
      maxUnavailable: 0  # nunca bajar de la capacidad actual durante un deploy
  # …
---
apiVersion: policy/v1
kind: PodDisruptionBudget
metadata:
  name: triton
  namespace: classifier
spec:
  # Evicciones VOLUNTARIAS (drain de nodos, upgrades del cluster, consolidación de Karpenter):
  # siempre queda al menos 1 Triton sirviendo. No protege de fallas involuntarias (eso es N+1).
  minAvailable: 1
  selector:
    matchLabels:
      app.kubernetes.io/name: triton
```

- `maxSurge: 1` y `maxUnavailable: 0`: cada pod nuevo necesita una GPU libre **antes** de que se vaya uno viejo. Si no hay, queda Pending y el rollout espera; los viejos siguen sirviendo.
- `progressDeadlineSeconds: 900`: un pod de GPU en un nodo nuevo tarda ≈ 4–11 min (nodo, imagen, engine). Con el default de 600 s, un rollout sano pero lento podría marcarse como fallido.
- `revisionHistoryLimit: 5` guarda 5 ReplicaSets viejos (el default es 10): son las versiones a las que `rollout undo` puede volver. En C01 el modelo va horneado en la imagen, así que volver al tag anterior es volver al modelo anterior.
- El PDB frena un `kubectl drain`, o a Karpenter (el autoscaler de nodos) cuando junta pods para apagar un nodo, si eso dejaría 0 Triton listos. No frena tu propio rollout (eso lo controla `maxUnavailable`) ni la muerte de un nodo (eso lo cubre tener 2 réplicas).
- Con 2 réplicas, `minAvailable: 1` permite 1 evicción (desalojo) a la vez. Con `minAvailable: 2` no se permitiría ninguna y los upgrades del cluster se quedarían trabados.
- Con `maxUnavailable: 0`, un `rollout undo` también necesita una GPU libre para el pod que vuelve: el rollback no es instantáneo.

CÓMO RECORDARLA

status → history → undo. Cada cambio del template deja una revisión, y volver atrás es elegir una. El PDB cuida los drains; `maxUnavailable` cuida tu rollout.

TRAMPA

creer que Kubernetes vuelve atrás solo. Un rollout trabado marca `ProgressDeadlineExceeded` y se queda ahí: el rollback lo haces tú, o una herramienta como Argo Rollouts. Y con GitOps (una herramienta como Argo CD aplica lo que está en git), un `rollout undo` dura hasta el próximo sync: el rollback de verdad es un revert en git.

En el curso: M6 · Alta disponibilidad: multi-AZ, PDB y N+1 · M11 · Canary: porcentajes, tiempos y criterios para avanzar · M11 · Rollback: criterios automáticos y qué guardar para volver · M11 · Blue/green y rollback instantáneo · Docker y kubectl · Rollouts y escalado · C01 · Modos de falla y degradación

11 kubectl: mirar, entrar, depurar

¿Cómo le preguntas al cluster qué hay, qué cambiaría y si tienes permiso, antes de tocar nada?

SINTAXIS CON VARIABLES

`kubectl` *verbo* *tipo* / *nombre* **`-n`** *ns* **`-o`** *formato*

Casi todo kubectl tiene esta forma: un verbo, un recurso, el **namespace** (la carpeta lógica donde vive) y el formato de salida. Para mirar o cambiar algo, kubectl siempre pasa por el **API server**, la puerta de entrada del cluster.

`kubectl config current-context`
`kubectl config use-context` *contexto*
`kubectl config set-context --current --namespace=` *ns*

Un **contexto** junta cluster, usuario y namespace por defecto. Vive en tu **kubeconfig**, el archivo de clusters y credenciales de kubectl (`~/.kube/config`). Míralo antes de cambiar algo.

`kubectl get` *tipo* / *nombre* **`-n`** *ns* **`-o`**
`jsonpath='{` *ruta* **`}{"\n"}'`**
`kubectl explain` *tipo* **`.`** *campo*
`kubectl diff -f` *archivo*

Mirar fino: un campo exacto con **JSONPath** (una ruta dentro del objeto, como `.status.readyReplicas`), la documentación de un campo y qué cambiaría si aplicas (exit 0 = nada, 1 = hay diferencias).

`kubectl exec -it -n` *ns* *pod* **`-c`** *contenedor* **`--`** *comando*
`kubectl port-forward -n` *ns* **`svc/`** *nombre* *local* **`:`** *puerto*
`kubectl auth can-i` *verbo* *tipo* **`-n`** *ns*

Entrar a un contenedor, abrir un túnel a un pod y preguntar si puedes. **RBAC** (los permisos de Kubernetes) decide si puedes hacer las dos primeras; `auth can-i` se lo pregunta.

VARIABLES

VARIABLE	QUÉ ES	EJEMPLO
verbo	lo que haces	`get, describe, delete · en can-i: list, create`
tipo	clase de recurso, con nombre largo o corto	`deployment (deploy), service (svc), pods (po)`
nombre	nombre del objeto	`triton`
ns	namespace	`classifier`
formato	cómo imprime la respuesta	`wide, yaml, json, name, jsonpath='…'`
contexto	entrada del kubeconfig: cluster + usuario + namespace	`docker-desktop · en EKS, por defecto el ARN del cluster (su identificador en AWS)`
ruta	camino al campo dentro del objeto; empieza con punto	`.status.readyReplicas`
campo	camino al campo para explain, sin punto inicial	`spec.progressDeadlineSeconds`
archivo	manifest local	`k8s/triton.yaml`
pod	nombre del pod, o deploy/NOMBRE para que elija uno	`deploy/triton`
contenedor	contenedor dentro del pod	`triton`
comando	lo que corres adentro	`nvidia-smi, sh`
local	puerto en tu máquina	`8000`
puerto	puerto del Service (o del pod)	`8000`

LAS PIEZAS

PIEZA	QUÉ HACE	EJEMPLO
`kubectl config get-contexts`	Lista tus contextos; el asterisco marca el activo.	`docker-desktop · arn:aws:eks:…:cluster/NOMBRE`
`-n NS · -A`	Un namespace, o todos a la vez (--all-namespaces).	`-n classifier · -n keda`
`-o wide`	Agrega columnas: en pods, la IP y el nodo.	`en qué nodo GPU cayó cada Triton`
`-o yaml`	El objeto completo como lo guarda el cluster, con defaults y status. Oculta managedFields (quién cambió cada campo).	`kubectl get deploy triton -o yaml`
`-o jsonpath='{…}'`	Saca solo los campos que pides. Para scripts y respuestas cortas.	`'{.status.readyReplicas}'`
`-l · --show-labels · -L`	Filtra por label, muestra todas, o pone una label como columna.	`-l app.kubernetes.io/name=triton`
`-w`	Se queda mirando y muestra cada cambio.	`kubectl get pods -w`

PIEZA	QUÉ HACE	EJEMPLO
kubectl explain	Documentación de un campo, servida por tu cluster y su versión.	**explain deploy.spec.progressDeadlineSeconds**
kubectl diff -f	Qué cambiaría si aplicas, calculado por el servidor. Exit 0 sin cambios, 1 con cambios.	**diff -f k8s/triton.yaml**
kubectl api-resources	Todos los tipos: nombre corto, grupo y si viven en un namespace.	**pdb, hpa, scaledobjects**
kubectl exec -it … -- sh	Abre una terminal, o corre un comando, dentro de un contenedor que ya corre.	**-- nvidia-smi**
kubectl port-forward	Túnel de tu máquina a un solo pod, sin exponer el servicio.	**svc/triton 8000:8000**
kubectl debug -it --image --target	Agrega un contenedor temporal con herramientas a un pod que corre.	**imágenes sin shell**
kubectl auth can-i	Pregunta si puedes hacer algo. Responde yes o no.	**create pods --subresource=exec**

QUIERES SABER	COMANDO	QUÉ MIRAR
¿En qué cluster estoy?	`kubectl config current-context`	El nombre del contexto; en EKS, el ARN del cluster
¿Qué namespace uso si no pongo -n?	`kubectl config view --minify -o jsonpath='{..namespace}'`	Vacío = default
¿Qué pods hay y dónde corren?	`kubectl get pods -n classifier -o wide`	STATUS, READY, RESTARTS y NODE
¿Qué tiene de verdad el cluster?	`kubectl get deploy triton -n classifier -o yaml`	La spec con defaults y el status
¿Qué significa este campo?	`kubectl explain deploy.spec.progressDeadlineSeconds`	Tipo, descripción y default
¿Cómo se llama ese recurso?	`kubectl api-resources`	SHORTNAMES, APIVERSION y NAMESPACED
¿Qué cambiaría si aplico?	`kubectl diff -f k8s/triton.yaml`	Líneas con - y +; exit 0 o 1
¿Puedo hacerlo?	`kubectl auth can-i create pods --subresource=exec -n classifier`	yes o no
¿Qué puedo hacer en total?	`kubectl auth can-i --list -n classifier`	Recursos y verbos permitidos
¿Responde desde mi máquina?	`kubectl port-forward -n classifier svc/triton 8000:8000`	Forwarding from 127.0.0.1:8000 -> 8000

Pregunta → comando → qué mirar. De arriba hacia abajo, de lo que solo lee a lo que ya abre una conexión a un pod.

EJEMPLO REAL · C01: UNA SESIÓN DE INSPECCIÓN DE TRITON, DE MENOR A MAYOR RIESGO

BASH · ZSH — Comandos de ref-comandos (kubectl-basicos), con los nombres de reference/c01/k8s/triton.yaml

```
# ¿Dónde estoy? Antes de tocar nada
kubectl config current-context      # en EKS: arn:aws:eks:us-east-1:123456789012:cluster/…
kubectl config set-context --current --namespace=classifier

# ¿Qué hay y dónde corre?
kubectl get pods -l app.kubernetes.io/name=triton -o wide
kubectl get deploy triton -o jsonpath='{.spec.strategy.rollingUpdate}{"\n"}'   # {"maxSurge":1,"maxUnavailable":0}

# ¿Qué cambiaría si aplico lo que está en git?
kubectl diff -f k8s/triton.yaml; echo "exit=$?"     # 0 = igual, 1 = hay diferencias

# ¿Puedo entrar? exec necesita create sobre pods/exec
kubectl auth can-i create pods --subresource=exec

# Túnel: localhost:8000 y :8002 → los mismos puertos de UN pod detrás de svc/triton
kubectl port-forward svc/triton 8000:8000 8002:8002

# En otra terminal
curl -s localhost:8000/v2/health/ready -o /dev/null -w "%{http_code}\n"   # 200 = listo
curl -s localhost:8002/metrics | grep nv_inference_queue_duration_us     # la señal de KEDA
```

```
# ¿Triton ve la GPU y cuánta memoria ocupa?
kubectl exec -it deploy/triton -- nvidia-smi
```

- Con `set-context --current --namespace=classifier` ya no repites `-n classifier`. Queda guardado en tu kubeconfig para ese contexto: la próxima vez, míralo.
- `kubectl diff` le pide al API server un **dry-run** (calcula el resultado sin guardarlo): el servidor aplica sus defaults y su validación, así que ves lo que de verdad cambiaría, no una comparación de texto con tu archivo.
- `port-forward` a `svc/triton` elige **un** pod y se corta si ese pod muere. Sirve para una prueba de humo, no para un load test: no reparte como el Service.
- `exec` y `port-forward` piden el verbo `create` sobre `pods/exec` y `pods/portforward`. Desde 1.35 el API server lo exige también cuando kubectl se conecta por WebSocket, el canal que usan las versiones nuevas; antes, por esa vía alcanzaba con `get`. En producción suelen estar restringidos: `auth can-i` responde antes de que lo intentes.
- `exec deploy/triton` elige un pod del Deployment. Para uno en particular, usa el nombre del pod.

CÓMO RECORDARLA

¿dónde? → ¿qué hay? → ¿qué es? → ¿qué cambiaría? → ¿puedo?: `config current-context`, `get`, `explain`, `diff`, `auth can-i`. Recién después, `apply` o `exec`.

TRAMPA

correr algo que cambia el cluster sin mirar el contexto. Un `delete` en producción, cuando creías estar en staging, no se deshace. Y con GitOps (una herramienta como Argo CD aplica lo que está en git), un `kubectl edit` o un `scale` se pierde en el próximo sync: el cambio va en git.

En el curso: Docker y kubectl · kubectl: mirar, entrar y conectar · Docker y kubectl · Qué de esto sirve en la entrevista · M15 · Secretos y credenciales · C01 · Archivos de referencia

12 Diagnóstico: el método y la tabla de estados

Tu pod no funciona: ¿qué miras primero, qué después, y qué te dice cada estado?

SINTAXIS CON VARIABLES

`kubectl get pods -n` *ns* `-o wide`

Estado: en qué fase se trabó (STATUS), si está listo para recibir tráfico (READY), cuántas veces se reinició (RESTARTS) y en qué nodo corre.

`kubectl describe pod` *pod* `-n` *ns*
`kubectl events -n` *ns* `--for pod/` *pod*

Eventos: avisos del scheduler (el que elige nodo) y del kubelet (el agente que corre los pods en cada nodo), en orden. Casi siempre el porqué está aquí.

`kubectl logs` *pod* `-n` *ns* `-c` *contenedor* `--previous`

Logs: lo que escribió el intento que murió. Sin `--previous` ves el intento actual, que quizá todavía no falló.

`kubectl exec -it` *pod* `-n` *ns* `-c` *contenedor* `-- sh`
`kubectl debug -it` *pod* `-n` *ns* `--image=busybox:1.38 --target=` *contenedor*

Exec: mirar por dentro (variables, archivos, red). Si la imagen no trae shell, `debug` agrega un contenedor temporal con herramientas.

VARIABLES

VARIABLE	QUÉ ES	EJEMPLO
ns	namespace del pod	`classifier`
pod	nombre del pod (sale de get pods)	`triton-5f6b…-7tq2m`
contenedor	contenedor dentro del pod; hace falta si hay más de uno	`triton`
exit code	lo que muestra describe en Last State	`137`
$n_{\text{señal}}$	número de la señal que terminó el proceso	`9 = SIGKILL (OOMKilled, o venció el plazo de gracia); 15 = SIGTERM`
k	número de reinicio seguido	`1, 2, 3…`
t_{espera}	espera antes de reiniciar en CrashLoopBackOff	`10, 20, 40… hasta 300 s; vuelve a 10 s tras 10 min corriendo bien`

FÓRMULAS

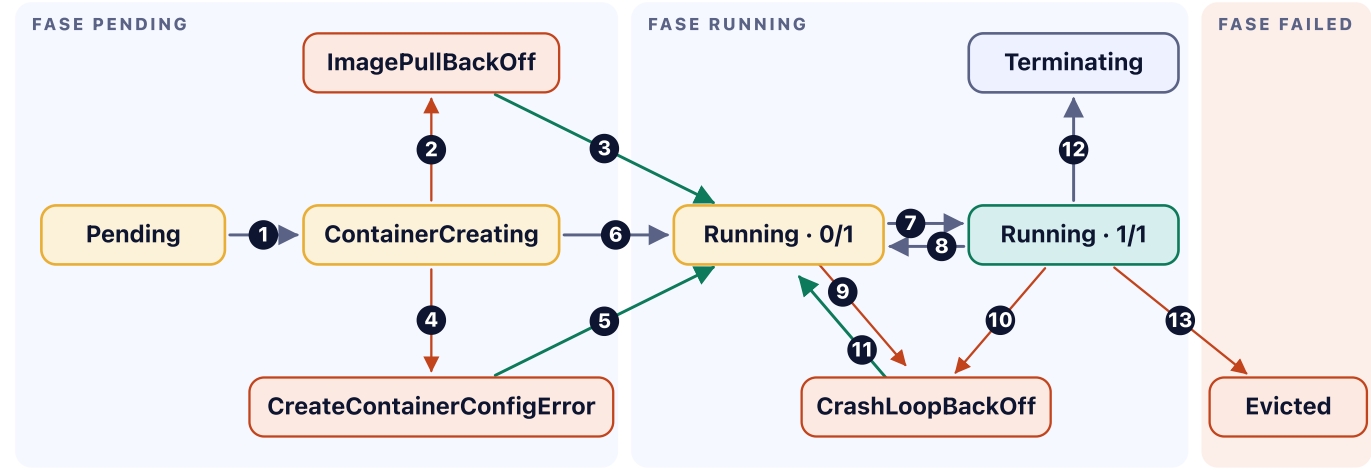
$$\text{exit code} = 128 + n_{\text{señal}} \qquad 137 = 128 + 9\ (\text{SIGKILL}) \qquad 143 = 128 + 15\ (\text{SIGTERM})$$

$$t_{\text{espera}} = \min\left(10\ \text{s} \cdot 2^{k-1},\ 300\ \text{s}\right)$$

LAS PIEZAS

PIEZA	QUÉ HACE	EJEMPLO
`Pending`	El pod existe, pero no tiene nodo todavía, o espera algo antes de crear contenedores.	`FailedScheduling: Insufficient nvidia.com/gpu`
`ContainerCreating`	Ya tiene nodo: baja la imagen, monta los volúmenes y crea el contenedor.	`la imagen de Triton, de varios GB`
`ErrImagePull · ImagePullBackOff`	No pudo bajar la imagen; reintenta con esperas crecientes, hasta 5 min.	`tag inexistente, 403 de ECR`
`CreateContainerConfigError`	No puede armar el contenedor: falta un Secret, un ConfigMap o una clave, o choca runAsNonRoot (pediste no ser root).	`secret "triton-token" not found`
`Running · READY 0/1`	El proceso corre, pero falla la readiness (el chequeo de ¿le mando tráfico?): no recibe requests del Service.	`Triton cargando el engine`
`Running · READY 1/1`	Listo: está en los endpoints del Service, la lista de pods a los que manda tráfico.	`lo normal`
`CrashLoopBackOff`	El contenedor muere una y otra vez; el kubelet espera más antes de cada reinicio.	`exit 1 al cargar el modelo`
`OOMKilled (en Last State)`	Pasó su límite de memoria y el kernel lo mató: exit 137, sin línea en el log.	`pesos leídos a RAM`
`Error · Completed`	El contenedor terminó: con un código distinto de 0, o con 0.	`un CMD que no se queda sirviendo`
`RunContainerError · StartError`	containerd (crea los contenedores en EKS y kind) no encuentra o no puede ejecutar el comando: StartError, exit 128.	`executable file not found in $PATH`
`Terminating`	Se borra: corre el preStop (paso previo al apagado), llega el SIGTERM y el plazo de gracia pone el tope.	`hasta 60 s en Triton`

PIEZA	QUÉ HACE	EJEMPLO
Evicted	El kubelet lo sacó porque al nodo le faltó memoria o disco. Fase Failed.	`The node was low on resource: memory`
Unknown	No se pudo saber el estado del pod, casi siempre porque su nodo no responde.	`nodo NotReady`



esperando · listo · falla · apagándose

1. **Pending → ContainerCreating**: el scheduler le asigna nodo
2. **ContainerCreating → ImagePullBackOff**: no baja la imagen (antes, ErrImagePull)
3. **ImagePullBackOff → Running · 0/1**: el pull por fin funciona
4. **ContainerCreating → CreateContainerConfigError**: falta un Secret, ConfigMap o clave
5. **CreateContainerConfigError → Running · 0/1**: creas lo que faltaba
6. **ContainerCreating → Running · 0/1**: arranca el proceso
7. **Running · 0/1 → Running · 1/1**: la readiness pasa
8. **Running · 1/1 → Running · 0/1**: la readiness falla: sale del Service
9. **Running · 0/1 → CrashLoopBackOff**: el contenedor termina (exit 1, OOMKilled; en un Deployment, también exit 0)
10. **Running · 1/1 → CrashLoopBackOff**: muere y vuelve a morir
11. **CrashLoopBackOff → Running · 0/1**: reinicio tras la espera
12. **Running · 1/1 → Terminating**: delete, rollout o scale-down
13. **Running · 1/1 → Evicted**: al nodo le falta memoria o disco

Pending, Running, Succeeded, Failed y Unknown son las cinco fases del pod. El resto es el motivo del contenedor que kubectl muestra en STATUS; READY dice si pasa la readiness.

EJEMPLO REAL · C01: UN DEPLOY DEJA UN TRITON NUEVO EN CRASHLOOPBACKOFF

BASH · ZSH — Comandos de ref-comandos (depurar) con los nombres de reference/c01/k8s/triton.yaml; salidas abreviadas, forma típica con kubectl 1.3x

```
# Deploy de un Triton nuevo: el pod nuevo no arranca y el rollout no termina
kubectl get pods -n classifier -l app.kubernetes.io/name=triton -o wide
# NAME              READY  STATUS             RESTARTS      AGE  IP          NODE
# triton-7d9c…-x2k4p  1/1    Running            0             3d   10.0.1.12   ip-10-0-1-5…
# triton-7d9c…-q8n7d  1/1    Running            0             3d   10.0.2.40   ip-10-0-2-9…
# triton-5f6b…-7tq2m  0/1    CrashLoopBackOff   4 (40s ago)   6m   10.0.1.77   ip-10-0-1-31…

kubectl describe pod -n classifier triton-5f6b…-7tq2m
#     Last State:  Terminated
#       Reason:    Error
#       Exit Code: 1
#   Warning  BackOff  …  kubelet  Back-off restarting failed container triton in pod …

kubectl logs -n classifier triton-5f6b…-7tq2m --previous | tail -n 20
# el error al cargar model.plan: Triton sale si un modelo no carga (--exit-on-error)

kubectl rollout undo deploy/triton -n classifier     # ya con la causa: vuelve al tag anterior
kubectl rollout status deploy/triton -n classifier --timeout=25m
```

- El orden importa: `get` dice dónde se trabó; `describe`, por qué (Last State, Exit Code y Events); `logs --previous`, qué dijo el proceso al morir.
- Exit 1 con `Reason: Error` es tu aplicación: no subas memoria. Exit 137 con `Reason: OOMKilled` sí es memoria.

- `4 (40s ago)` en RESTARTS: cuántas veces y hace cuánto fue el último. Entre intentos, la espera crece: 10 s, 20 s, 40 s… hasta 5 min. Ese default sigue igual en 1.37; desde 1.35, un admin puede bajar el tope por nodo en la configuración del kubelet.
- Con `maxSurge: 1` y `maxUnavailable: 0` (entra un pod nuevo antes de que salga uno viejo), los dos Triton viejos siguen sirviendo: no hay caída, así que diagnosticas sin apuro antes de decidir.
- Guarda `describe` y `logs --previous` antes de borrar nada: los eventos duran 1 h por defecto, y borrar el pod se lleva su log.
- Con containerd (EKS, kind) la tabla de exit codes no es la de Docker: un comando que no existe o no es ejecutable da `Reason: StartError` y `Exit Code: 128`; 126 y 127 solo salen si un shell dentro del contenedor los devuelve, y 125 no aparece.

CÓMO RECORDARLA

estado → eventos → logs → exec. `get` dice en qué fase se trabó; `describe` y `events`, por qué; `logs --previous`, qué dijo al morir; `exec`, cómo se ve por dentro.

TRAMPA

borrar el pod o hacer `rollout restart` para “arreglarlo”. Se lleva el log del intento que murió, y el pod nuevo sale igual de roto. Y no te quedes en STATUS: `CrashLoopBackOff` es el síntoma; la causa está en el Exit Code y en el log.

En el curso: Docker y kubectl · Depurar pods: síntoma → comandos → causa típica · Docker y kubectl · Autoscaling y capacidad · C01 · Modos de falla y degradación · M6 · Los objetos que sí debes conocer · M6 · Cold starts de punta a punta

13 Errores que debes reconocer rápido

Lees el mensaje: ¿quién lo dijo, qué significa, qué corres primero y dónde está la explicación?

SINTAXIS CON VARIABLES

`Error from server (`*motivo*`):` *mensaje*

Lo rechazó el **API server** (la puerta de entrada del cluster): permisos, validación o algo que no existe. Ese objeto no se creó ni cambió.

`The` *Kind* **`"`** *nombre* **`" is invalid:`** *campo* **`:`** *detalle*

La validación rechazó tu YAML: un valor imposible, o un campo que ya no se puede cambiar.

`Warning` *motivo* *edad* *origen* *mensaje*

Un evento en `describe` o `kubectl events`: lo dijo el scheduler, el kubelet o un controlador. Explica por qué el pod no avanza.

`Last State: Terminated · Reason:` *motivo* **`· Exit Code:`** *codigo*

Cómo murió el intento anterior del contenedor. Lo que pasó adentro está en `kubectl logs --previous`.

VARIABLES

VARIABLE	QUÉ ES	EJEMPLO
motivo	la razón corta, en CamelCase	`Forbidden`, `NotFound` · `FailedScheduling`, `BackOff` · `OOMKilled`, `Error`
mensaje	el texto que explica la razón	`pods is forbidden: User "ana" cannot list resource "pods"…`
Kind	el tipo de objeto	`Deployment`
nombre	el nombre del objeto	`gateway`
campo	la ruta del campo con problema	`spec.selector`
detalle	qué tiene de malo	`field is immutable`
edad	hace cuánto, y cuántas veces se repitió	`2m (x10 over 6m)`
origen	quién lo emitió	`default-scheduler`, `kubelet`, `cluster-autoscaler`
codigo	exit code del proceso	`137`

LAS PIEZAS

PIEZA	QUÉ HACE	EJEMPLO
`Error from server (Forbidden)`	Casi siempre RBAC (permisos): te falta ese verbo sobre ese recurso en ese namespace. También: una cuota llena.	`cannot list resource "pods"`
`Error from server (NotFound)`	No existe con ese nombre en ese namespace; a veces el namespace es otro.	`deployments.apps "triton" not found`
`Error from server (AlreadyExists)`	Hiciste create de algo que ya existe; apply lo actualizaría.	`kubectl create -f dos veces`
`Error from server (BadRequest)`	El servidor no puede procesar el pedido: un campo desconocido, o logs de un contenedor que no arrancó.	`strict decoding error: unknown field`
`Error from server (Conflict)`	Otro cambió el objeto entre que lo leíste y lo escribiste. Reintenta sobre la versión nueva.	`the object has been modified`
`The … is invalid:`	La validación rechazó un valor: formato, rango o un campo inmutable.	`spec.selector: … field is immutable`
`error: You must be logged in to the server`	El API server no acepta tus credenciales: vencidas o de una identidad que no conoce.	`(Unauthorized)`
`The connection to the server … was refused`	No hay API server en esa dirección: cluster apagado o sin kubeconfig.	`localhost:8080`
`Unable to connect to the server`	No llega a la dirección: red, VPN, DNS o un cluster que solo se alcanza desde su red privada.	`dial tcp …:443: i/o timeout`
`error: … (sin from server)`	Lo dijo kubectl mismo: un flag, un archivo que no parsea, un tipo que no existe en el cluster.	`the server doesn't have a resource type`
`Warning FailedScheduling`	El scheduler no encontró nodo; cuenta los nodos por motivo.	`0/5 nodes are available`

PIEZA	QUÉ HACE	EJEMPLO
Warning Failed · BackOff	El kubelet no pudo bajar la imagen, armar o arrancar el contenedor, y reintenta.	**Back-off pulling image**
Warning Unhealthy	Falló una probe: Readiness, Liveness o Startup probe failed.	**HTTP probe failed with statuscode: 503**
Last State: Terminated	Cómo murió el intento anterior: Reason y Exit Code.	**OOMKilled · 137**

LOS 26 MENSAJES

0/3 nodes are available: 3 Insufficient nvidia.com/gpu.

El scheduler no encontró un nodo con una GPU libre: todas están asignadas (cada pod se lleva una entera), o el nodo no anuncia nvidia.com/gpu porque el device plugin (el pod de NVIDIA que anuncia las GPUs del nodo) no corre ahí. El pod queda Pending; si el autoscaler de nodos puede, agrega uno.

PRIMERO:

```
kubectl get nodes -o 'custom-columns=NAME:.metadata.name,GPU:.status.allocatable.nvidia\.com/gpu'
```

→ Sección 8 · Dónde cae cada pod: nodeSelector, affinity, taints/tolerations, GPUs y nodos que se crean solos

node(s) had untolerated taint {nvidia.com/gpu: present} · node(s) had untolerated taint(s)

Hay nodos, pero con un taint (una marca que rechaza pods) que tu pod no tolera. Según la versión, el mensaje nombra el taint (hasta 1.33, y en 1.34.0–1.34.1) o no (desde 1.35, y 1.34.2 en adelante): míralo en el nodo. En EKS, un pod que pide nvidia.com/gpu recibe sola la toleration de ese taint (la agrega ExtendedResourceToleration, un paso del API server): si un pod de GPU ve este mensaje, busca otro taint.

PRIMERO:

```
kubectl get nodes -o 'custom-columns=NAME:.metadata.name,TAINTS:.spec.taints[*].key'
```

→ Sección 8 · Dónde cae cada pod: nodeSelector, affinity, taints/tolerations, GPUs y nodos que se crean solos

node(s) didn't match Pod's node affinity/selector

Ningún nodo tiene las labels que pide tu nodeSelector o tu affinity (las reglas de a qué nodos puede ir el pod). En C01, Triton pide workload=gpu-t4: si el grupo de nodos de GPU no tiene esa label, el pod no cae en ningún lado.

PRIMERO: `kubectl get nodes -L workload`

→ Sección 8 · Dónde cae cada pod: nodeSelector, affinity, taints/tolerations, GPUs y nodos que se crean solos

Failed to pull image "…": … not found · Back-off pulling image "…"

El kubelet no pudo bajar la imagen: primero ErrImagePull y después ImagePullBackOff, que reintenta con esperas crecientes, hasta 5 min. not found es un tag o un nombre mal escrito; 403 Forbidden o 401, permisos del registry. El texto del medio depende del runtime (containerd en EKS y kind).

PRIMERO: `kubectl describe pod POD -n classifier`

→ Sección 12 · Diagnóstico: el método y la tabla de estados

no match for platform in manifest · exec /…: exec format error

La imagen es de otra arquitectura: la construiste en tu Mac (arm64) y los nodos son amd64. Si su índice (la lista de variantes por arquitectura) no trae amd64, falla el pull; si es una imagen de una sola arquitectura, baja bien y muere al arrancar, con exec format error en el log.

PRIMERO:

```
docker buildx imagetools inspect REGISTRY/IMAGEN:TAG
```

→ Sección 12 · Diagnóstico: el método y la tabla de estados

Back-off restarting failed container triton in pod triton-…

El evento del CrashLoopBackOff: el contenedor muere y el kubelet espera cada vez más antes de reiniciarlo (10 s, 20 s, 40 s… hasta 5 min). Es el síntoma; la causa está en el log del intento anterior y en el Exit Code.

PRIMERO: `kubectl logs POD -n classifier --previous`

→ Sección 12 · Diagnóstico: el método y la tabla de estados

container has runAsNonRoot and image will run as root

Pediste no correr como root (runAsNonRoot: true), pero la imagen arranca como root y no fijaste otro usuario: el kubelet no crea el contenedor (CreateContainerConfigError). Las imágenes de NGC (el catálogo de contenedores de NVIDIA) arrancan como root; por eso C01 fija runAsUser: 1000 en Triton.

PRIMERO:

```
kubectl get pod POD -n classifier -o jsonpath='{.spec.securityContext}'
```

→ Sección 12 · Diagnóstico: el método y la tabla de estados

Readiness probe failed: HTTP probe failed with statuscode: 503

La readiness respondió 503 (o cualquier código fuera de 200–399): el pod sale de los endpoints del Service (la lista de pods que reciben tráfico), sin reiniciarse. En C01, el /readyz del gateway da 503 a propósito cuando Triton no está listo.

PRIMERO: `kubectl logs POD -n classifier --tail=50`

→ Sección 5 · Probes y apagado ordenado

`Liveness probe failed: Get "http://…/healthz": context deadline exceeded (Client.Timeout exceeded while awaiting headers) · Container gateway failed liveness probe, will be restarted`

La liveness no respondió dentro de timeoutSeconds (1 s por defecto) y, tras failureThreshold fallos seguidos, el kubelet reinició el contenedor. Bajo carga suele ser un timeout corto o una liveness cara, no un proceso colgado.

PRIMERO:
```
kubectl get pod POD -n classifier -o jsonpath='{.spec.containers[0].livenessProbe}'
```

→ Sección 5 · Probes y apagado ordenado

`Last State: Terminated · Reason: OOMKilled · Exit Code: 137`

El contenedor pasó su limits.memory y el kernel lo mató, sin dejar una línea en el log. Es RAM, no VRAM; en un servicio de modelos suele ser el pico al cargar los pesos.

PRIMERO: `kubectl describe pod POD -n classifier`

→ Sección 6 · Requests, limits y QoS

`The node was low on resource: memory.`

Un desalojo (eviction): al nodo le faltó memoria y el kubelet sacó pods, primero los que usaban más de lo que pidieron. El pod queda en fase Failed con Reason: Evicted. Si dice ephemeral-storage, lo que se llenó es el disco.

PRIMERO: `kubectl describe node NODO`

→ Sección 6 · Requests, limits y QoS

`CUDA out of memory. Tried to allocate …`

PyTorch se quedó sin memoria de la GPU (VRAM). Es una excepción en el log (torch.OutOfMemoryError, o torch.cuda.OutOfMemoryError según la versión), no un OOMKilled. Se arregla con batch o resolución más chicos, menos procesos por GPU u otra GPU, no subiendo limits.memory.

PRIMERO: `kubectl exec -n classifier POD -- nvidia-smi`

→ Sección 6 · Requests, limits y QoS

`503 Service Temporarily Unavailable (la respuesta del ALB)`

Lo generó el balanceador, no tu app: el target group (la lista de pods a los que el ALB manda tráfico) no tiene targets registrados. En C01 pasa si ningún gateway pasa /readyz (por ejemplo, con Triton caído). Si los targets registrados están todos unhealthy, el ALB igual les manda tráfico (fail open). Ojo: el gateway de C01 también responde 503, con su JSON y Retry-After, cuando corta carga; ese no es el ALB.

PRIMERO:
```
kubectl get endpointslices -n classifier -l kubernetes.io/service-name=gateway
```

→ Sección 4 · Services, DNS e Ingress

`Failed build model due to couldn't auto-discover subnets: unable to resolve at least one subnet`

El AWS Load Balancer Controller no encontró subredes para crear el ALB: les falta el tag kubernetes.io/role/elb (ALB público) o kubernetes.io/role/internal-elb (interno, como en C01). El Ingress se queda sin dirección.

PRIMERO: `kubectl describe ingress gateway -n classifier`

→ Sección 4 · Services, DNS e Ingress

`Error: secret "triton-token" not found · couldn't find key MODEL_NAME in ConfigMap classifier/gateway-config`

CreateContainerConfigError: el pod usa un Secret, un ConfigMap o una clave que no existe en su namespace. El kubelet reintenta solo cuando lo creas.

PRIMERO: `kubectl get secret,configmap -n classifier`

→ Sección 7 · ConfigMaps, Secrets, volúmenes y ServiceAccounts

---

`Not authorized to perform sts:AssumeRoleWithWebIdentity`

IRSA (el pod recibe un rol de IAM, es decir, permisos de AWS, a través de su ServiceAccount) no funciona: el pod no puede asumir ese rol. Casi siempre, la trust policy del rol (quién puede usarlo) no coincide con el namespace y el nombre del ServiceAccount, o apunta al OIDC provider (el emisor de tokens) de otro cluster. Lo verás en el log de tu app (lo dice el SDK de AWS), no en los eventos.

PRIMERO:
`kubectl get serviceaccount SA -n classifier -o yaml`

→ Sección 7 · ConfigMaps, Secrets, volúmenes y ServiceAccounts

---

`Error from server (Forbidden): pods is forbidden: User "…" cannot list resource "pods" in API group "" in the namespace "classifier"`

RBAC (los permisos de Kubernetes): tu identidad no tiene ese verbo sobre ese recurso en ese namespace. El mensaje dice quién, qué verbo, qué recurso y dónde.

PRIMERO: `kubectl auth can-i list pods -n classifier`

→ Sección 11 · kubectl: mirar, entrar, depurar

`error: You must be logged in to the server (Unauthorized)`

El API server no acepta tus credenciales: vencieron, o son de una identidad que el cluster no conoce. En EKS: tu terminal usa otra identidad de IAM (otro usuario o rol de AWS), o esa identidad no tiene un access entry (el permiso de EKS que la conecta con el cluster) ni una entrada en el ConfigMap aws-auth, el método anterior.

PRIMERO: `aws sts get-caller-identity`

→ Sección 11 · kubectl: mirar, entrar, depurar

`The connection to the server localhost:8080 was refused - did you specify the right host or port?`

kubectl no encontró un cluster: sin kubeconfig apunta a localhost:8080, donde no hay nada. Con el cluster local apagado, o un contexto que apunta a otro lado, sale el mismo mensaje con otra dirección.

PRIMERO: `kubectl config current-context`

→ Sección 11 · kubectl: mirar, entrar, depurar

`no matches for kind "ScaledObject" in version "keda.sh/v1alpha1" · ensure CRDs are installed first`

El cluster no conoce ese tipo: falta su CRD (CustomResourceDefinition, la definición del tipo nuevo, que instala KEDA o el operador que sea). Instala el componente antes de aplicar sus objetos; si ya está, revisa el apiVersion.

PRIMERO: `kubectl get crd scaledobjects.keda.sh`

→ Sección 2 · Anatomía de un manifest

`error: deployment "triton" exceeded its progress deadline`

El rollout no avanzó dentro de progressDeadlineSeconds (600 s por defecto; 900 en Triton): Progressing=False con ProgressDeadlineExceeded. Kubernetes no vuelve atrás solo; los pods viejos siguen sirviendo.

PRIMERO:

```
kubectl get pods -n classifier -l
app.kubernetes.io/name=triton -o wide
```

→ Sección 10 · Rollouts, rollback, PodDisruptionBudget y canary

`Cannot evict pod as it would violate the pod's disruption budget.`

Un desalojo voluntario (un kubectl drain, el upgrade de un grupo de nodos) chocó con el PodDisruptionBudget: ahora no se permite sacar ese pod. kubectl drain reintenta cada 5 s. Si el PDB pide tantos pods como réplicas, nunca va a poder.

PRIMERO: `kubectl get pdb -n classifier`

→ Sección 10 · Rollouts, rollback, PodDisruptionBudget y canary

`failed to get cpu utilization: missing request for cpu in container gateway of Pod gateway-…`

El HPA (el autoscaler de pods) mide la CPU como porcentaje del request, y ese contenedor no tiene requests.cpu: no puede calcular la utilización y no escala.

PRIMERO: `kubectl describe hpa gateway -n classifier`

→ Sección 9 · Autoscaling: HPA y KEDA

`prometheus metrics 'prometheus' target may be lost, the result is empty`

KEDA consultó a Prometheus y la respuesta vino vacía. En C01 aparece como error a propósito (ignoreNullValues: "false"): casi siempre falta el ServiceMonitor (el objeto que le dice a Prometheus qué leer) y Prometheus no lee las métricas de Triton. Con el default (true), KEDA lo tomaría como 0, en silencio.

PRIMERO:

```
kubectl describe scaledobject triton -n classifier
```

→ Sección 9 · Autoscaling: HPA y KEDA

`strict decoding error: unknown field "spec.template.spec.containers[0].resource"`

Un campo que no existe en ese lugar: un typo (resource en vez de resources) o mala indentación. kubectl valida en el servidor en modo estricto por defecto, así que lo rechaza en vez de ignorarlo.

PRIMERO:

```
kubectl explain
deployment.spec.template.spec.containers.resources
```

→ Sección 2 · Anatomía de un manifest

`spec.template.metadata.labels: Invalid value: …: `selector` does not match template `labels``

El selector del Deployment no coincide con las labels del template de pods: el Deployment no podría encontrar sus propios pods. Pon las mismas labels en los dos lados, como app.kubernetes.io/name: gateway en C01.

PRIMERO:

```
kubectl apply -f k8s/gateway.yaml --dry-run=server
```

→ Sección 2 · Anatomía de un manifest

EJEMPLO REAL · CÓMO LEER UN FAILEDSCHEDULING LARGO: TRITON NO ENCUENTRA GPU

TEXTO — Forma típica del evento en Kubernetes 1.3x con Cluster Autoscaler; los nombres y la label son los de reference/c01/k8s/triton.yaml

```
$ kubectl describe pod -n classifier triton-5f6b…-9xk2w
…
Events:
  Type     Reason            Age   From               Message
  ----     ------            ---   ----               -------
  Warning  FailedScheduling  2m    default-scheduler  0/5 nodes are available: 2 Insufficient nvidia.com/gpu, 3 node(s)
didn't match Pod's node affinity/selector. preemption: 0/5 nodes are available: 2 No preemption victims found for
incoming pod, 3 Preemption is not helpful for scheduling.
  Normal   TriggeredScaleUp  2m    cluster-autoscaler  pod triggered scale-up: [{… 2->3 (max: 4)}]
```

- Léelo por partes: `0/5 nodes are available` es el total; después viene cuántos nodos fallaron por cada motivo. Esa lista es la causa.
- Aquí, los 2 nodos GPU no tienen una T4 libre (`Insufficient nvidia.com/gpu`) y los 3 de CPU no tienen la label `workload=gpu-t4` . Lo segundo está bien: Triton no debe caer ahí.
- Lo que sigue a `preemption:` dice que sacar otros pods no ayudaría. No es la causa: no hay nada que arreglar ahí.
- `TriggeredScaleUp` es del Cluster Autoscaler (el que agrega nodos): pidió un nodo GPU más. El pod espera unos minutos y se programa solo. Si en cambio ves `NotTriggerScaleUp` , no encontró un grupo de nodos que pueda crecer y le sirva al pod: están en su máximo, ninguno trae la label o la GPU, o quedaron en pausa tras fallar (por ejemplo, sin cuota).
- Si el motivo fuera un taint (una marca del nodo que rechaza pods), el texto depende de la versión: hasta 1.33 (y en 1.34.0–1.34.1) nombra el taint (`{nvidia.com/gpu: present}`); desde 1.35 (y 1.34.2 en adelante) solo dice `untolerated taint(s)` : míralo en el nodo.

CÓMO RECORDARLA

quién habla dice dónde mirar. `Error from server` es el API server (permisos, validación, no existe); un `Warning` en los eventos es el scheduler o el kubelet; `Last State` es cómo murió el proceso; lo demás lo dijo tu app y está en `kubectl logs` .

TRAMPA

leer solo la primera línea, o solo STATUS. En un `FailedScheduling` , el conteo por motivo dice qué arreglar y lo que sigue a `preemption:` no es la causa. Y algunos textos cambian según la versión: desde 1.35 (y 1.34.2) el scheduler ya no nombra el taint, y desde 1.33 el conflicto de zona de un volumen dice `didn't match PersistentVolume's node affinity` (antes, `had volume node affinity conflict`).

En el curso: Docker y kubectl · Depurar pods: síntoma → comandos → causa típica · Docker y kubectl · kubectl: mirar, entrar y conectar · Docker y kubectl · Rollouts y escalado · M6 · GPUs en Kubernetes

---

Comandos y campos para Kubernetes 1.37 (octubre de 2026), revisados contra la documentación oficial: kubernetes.io, Amazon EKS, el AWS Load Balancer Controller, KEDA, NVIDIA y Argo Rollouts. Los laboratorios no se ejecutaron al escribir esta hoja: si tu terminal dice otra cosa, créele a tu terminal. Los ejemplos salen de `reference/` del curso.

© 2026 Erik Palacios · Contenido bajo CC BY-NC-ND 4.0 · ML desde cero · Reportar un error