PuigFlows

Graceful Shutdown en Kubernetes: Despliegues sin Errores ni Peticiones Perdidas

Graceful Shutdown in Kubernetes: Deploys Without Errors or Dropped Requests

Guía · Intermedio · Proyectos Reales · ~35 min de lectura / ~35 min read
Edición bilingüe: Español + English · puigflows.com · Julio 2026

ESPAÑOL

Graceful Shutdown en Kubernetes: Despliegues sin Errores ni Peticiones Perdidas

Por qué cada deploy puede estar perdiendo peticiones

Cada vez que despliegas una nueva versión, Kubernetes mata pods. Lo mismo ocurre cuando el autoscaler reduce réplicas, cuando un nodo spot desaparece o cuando el scheduler reubica cargas. En un sistema con deploys frecuentes, la terminación de pods no es un evento raro: es rutina diaria.

Si tu servicio no gestiona bien ese apagado, los síntomas son conocidos: errores 502/503 durante cada deploy, conexiones cortadas a mitad de respuesta, transacciones a medias y mensajes de cola procesados dos veces. Lo insidioso es que todo funciona en pruebas —donde nadie mata el proceso— y falla justo en producción, en el peor momento posible.

La buena noticia: el apagado elegante (graceful shutdown) es un patrón bien entendido. Solo hay que implementarlo en las tres capas correctas: la plataforma (Kubernetes), el proceso (tu aplicación) y las dependencias (conexiones, colas y trabajos en curso).

La secuencia de terminación en Kubernetes

Cuando Kubernetes decide eliminar un pod, ocurre lo siguiente:

- El pod pasa a estado Terminating y se retira de los endpoints del Service.
- En paralelo, se ejecuta el hook preStop (si existe) y después se envía SIGTERM al proceso principal de cada contenedor.
- Empieza a correr el presupuesto de terminationGracePeriodSeconds (30 segundos por defecto), que cubre tanto el preStop como el apagado de tu aplicación.
- Si el proceso sigue vivo al agotarse el plazo, recibe SIGKILL: sin apelación y sin limpieza.

El detalle que rompe la mayoría de los deploys está entre los pasos 1 y 2: la retirada de endpoints y el SIGTERM ocurren en paralelo, no en orden. kube-proxy, los ingress controllers y los balanceadores externos tardan segundos en enterarse de que el pod ya no debe recibir tráfico. Durante esa ventana, tu aplicación —que quizá ya empezó a apagarse— sigue recibiendo peticiones nuevas. Resultado: connection refused y 502s.

El primer arreglo: un preStop con sleep

La solución estándar a ese race condition es contraintuitiva: antes de apagarte, espera. Un preStop con un sleep de unos segundos retrasa el SIGTERM y da tiempo a que la infraestructura de red deje de enviarte tráfico mientras tu aplicación sigue atendiendo con normalidad.

```
spec:
  terminationGracePeriodSeconds: 60
  containers:
  - name: api
    image: mi-api:2.4.0
    lifecycle:
      preStop:
        sleep:
          seconds: 8
```

Desde Kubernetes 1.30 existe la acción nativa sleep (estable en 1.32); en versiones anteriores se usa exec con ["sh", "-c", "sleep 8"], que exige que la imagen incluya un shell. Dos reglas importantes: el sleep debe superar el tiempo que tu infraestructura tarda en propagar la retirada del endpoint (5–10 segundos suele bastar), y el terminationGracePeriodSeconds debe ampliarse en consecuencia, porque el preStop consume parte de ese presupuesto.

Manejo de SIGTERM en Python (FastAPI + Uvicorn)

Uvicorn ya captura SIGTERM y deja de aceptar conexiones nuevas mientras termina las que están en curso. Tu trabajo es doble: marcar el pod como «no listo» para que los balanceadores lo saquen de rotación, y cerrar recursos (pools de base de datos, clientes HTTP) después de drenar las peticiones, no antes.

```
import asyncio
import signal
from contextlib import asynccontextmanager
from fastapi import FastAPI, Response

shutting_down = False

@asynccontextmanager
async def lifespan(app: FastAPI):
    app.state.pool = await create_db_pool()

    def on_sigterm():
        global shutting_down
        shutting_down = True  # el readiness probe empezará a fallar

    loop = asyncio.get_running_loop()
    loop.add_signal_handler(signal.SIGTERM, on_sigterm)

    yield  # la aplicación atiende peticiones

    # Aquí Uvicorn ya drenó las peticiones en curso:
    # ahora sí es seguro cerrar los recursos.
    await app.state.pool.close()

app = FastAPI(lifespan=lifespan)

@app.get("/healthz/ready")
async def ready():
    if shutting_down:
        return Response(status_code=503)
    return {"status": "ok"}
```

Arranca Uvicorn con --timeout-graceful-shutdown 25 para poner un tope al drenado: si una petición lleva más de 25 segundos, se cierra igualmente antes de que llegue el SIGKILL. Ese número debe caber dentro del grace period, descontando el preStop.

Manejo de SIGTERM en Node.js (Express)

En Node el método clave es server.close(): deja de aceptar conexiones nuevas y espera a que terminen las peticiones activas. El problema clásico son las conexiones keep-alive inactivas, que mantienen el servidor abierto indefinidamente. Desde Node 18.2 existe closeIdleConnections() precisamente para eso.

```
const express = require("express");

const app = express();
let shuttingDown = false;

app.get("/healthz/ready", (req, res) => {
  res.status(shuttingDown ? 503 : 200).end();
});

const server = app.listen(8080);

process.on("SIGTERM", () => {
  shuttingDown = true; // 1. dejar de anunciarse como listo

  // 2. no aceptar conexiones nuevas y drenar las activas
  server.close(() => {
    console.log("Drenado completo, saliendo");
    process.exit(0);
  });

  // 3. cerrar sockets keep-alive sin peticiones en vuelo
  server.closeIdleConnections();

  // 4. red de seguridad: forzar salida antes del SIGKILL
  setTimeout(() => {
    console.error("Timeout de drenado, forzando cierre");
    server.closeAllConnections();
    process.exit(1);
  }, 25000).unref();
});
```

El error más común aquí es llamar a process.exit(0) directamente en el handler de SIGTERM: eso mata las peticiones en vuelo al instante, que es exactamente lo que intentamos evitar.

El problema del PID 1: cuando SIGTERM nunca llega

Si tu aplicación «ignora» el SIGTERM y siempre muere por SIGKILL tras 30 segundos, casi seguro que la señal no le está llegando. La causa habitual es la forma shell del CMD en el Dockerfile: CMD npm start arranca un shell como PID 1, y ese shell no reenvía señales a sus procesos hijos.

```
# MAL: el shell es PID 1 y se traga el SIGTERM
CMD npm start

# BIEN: forma exec, tu proceso es PID 1
CMD ["node", "server.js"]

# Alternativa robusta: tini como init
RUN apk add --no-cache tini
ENTRYPOINT ["tini", "--"]
CMD ["node", "server.js"]
```

Para comprobarlo: kubectl exec <pod> -- ps aux y verifica qué proceso es PID 1. Si es sh, ya sabes por qué tu graceful shutdown nunca se ejecuta.

Workers de colas: parar de consumir, no de procesar

En un worker el apagado elegante significa otra cosa: al recibir SIGTERM, deja de tomar mensajes nuevos, termina el que tienes entre manos y confirma (ack) antes de salir. Nunca hagas ack antes de procesar: si te matan a mitad, el mensaje se pierde.

```
import signal

class Worker:
    def __init__(self, queue):
        self.queue = queue
        self.running = True
        signal.signal(signal.SIGTERM, self._stop)

    def _stop(self, signum, frame):
        self.running = False  # no tomar más mensajes

    def run(self):
        while self.running:
            msg = self.queue.receive(timeout=5)
            if msg is None:
                continue
            process(msg)  # terminar el mensaje actual
            msg.ack()     # confirmar SOLO tras procesar
        print("Worker parado limpiamente")
```

Dimensiona el grace period según el mensaje más lento: si un trabajo puede tardar 90 segundos, un grace period de 30 garantiza trabajos matados a medias. Y como la entrega al-menos-una-vez es inevitable, tus consumidores deben ser idempotentes —tema que tratamos en la guía de idempotencia de esta misma serie.

Cómo calcular terminationGracePeriodSeconds

El grace period es un presupuesto compartido que empieza a contar en el momento en que el pod entra en Terminating. Súmalo por partes:

- Sleep del preStop: 8 s
- Drenado de la petición o mensaje más lento: 20 s

- Cierre de pools, flush de logs y métricas: 5 s
- Margen de seguridad: 10 s

Total: 43 segundos; redondea a 60. El valor por defecto de 30 segundos se queda corto en cuanto añades un preStop y tienes peticiones de más de unos segundos.

Errores comunes

- Añadir el preStop sin ampliar el grace period: el sleep se come el presupuesto y tu aplicación acaba en SIGKILL.
- process.exit() o sys.exit() directo en el handler: convierte el apagado elegante en uno brutal.
- Cerrar la base de datos antes de drenar: las últimas peticiones fallan con errores de conexión cerrada.
- CMD en forma shell: SIGTERM nunca llega y todo lo demás da igual.
- Ignorar las conexiones keep-alive: server.close() nunca resuelve y siempre sales por timeout.
- Ack antes de procesar en workers: mensajes perdidos cada vez que un pod muere a mitad de un trabajo.

Readiness probes durante el apagado: sal de rotación antes de dejar de responder

En los ejemplos de FastAPI y Express marcamos el pod como «no listo» al recibir SIGTERM. Merece la pena detenerse en por qué ese detalle importa y cómo configurar las probes para que el efecto sea inmediato y no llegue tarde.

Cuando un pod entra en Terminating, Kubernetes lo retira de los endpoints del Service sin consultar a la readiness probe. Entonces, ¿para qué fallarla? Por dos razones. La primera: no todo el tráfico llega a través de los endpoints. Los balanceadores que registran pods directamente como targets —como el AWS Load Balancer Controller en modo IP— hacen sus propios health checks contra el pod, y esos checks sí dependen de tu endpoint de readiness. La segunda: fallar la probe es la única señal universal que entienden todas las capas, desde el kubelet hasta un healthcheck externo. Es barato y elimina toda una clase de carreras.

```
readinessProbe:
  httpGet:
    path: /healthz/ready
    port: 8080
  periodSeconds: 2
  failureThreshold: 2
livenessProbe:
  httpGet:
    path: /healthz/live
    port: 8080
  periodSeconds: 10
  failureThreshold: 3
```

La aritmética importa: con periodSeconds: 2 y failureThreshold: 2, quien dependa de esa probe tarda como mucho unos 4 segundos en dejar de enviarte tráfico desde que empiezas a devolver 503. Ese tiempo tiene que caber dentro del sleep del preStop: si tu sleep es de 8 segundos y la probe necesita 4 para propagarse, vas justo.

Con los valores por defecto (periodSeconds: 10, failureThreshold: 3) necesitarías hasta 30 segundos, más que el grace period entero por defecto.

Un error sutil y frecuente: usar el mismo endpoint para liveness y readiness. Si tu liveness probe empieza a fallar durante el drenado —porque devuelves 503 en el endpoint compartido—, el kubelet puede decidir reiniciar el contenedor en mitad del apagado, justo lo contrario de lo que buscas. Sepáralos siempre: liveness responde «¿el proceso está vivo?» y no debe cambiar durante el shutdown; readiness responde «¿quiero tráfico ahora?» y es la que se apaga primero. En los ejemplos anteriores, /healthz/live puede devolver 200 incondicionalmente mientras el proceso exista.

PodDisruptionBudgets: sobrevivir a los drains, upgrades y nodos spot

Todo lo anterior protege una terminación individual. Pero en producción los pods rara vez mueren de uno en uno: un upgrade de nodos drena máquinas enteras, el cluster autoscaler consolida cargas y los nodos spot desaparecen con dos minutos de aviso (en AWS) o treinta segundos (en algunas zonas de GCP). Si todas tus réplicas viven en el mismo nodo y ese nodo se drena, tu graceful shutdown perfecto no evitará una caída total del servicio.

El PodDisruptionBudget (PDB) es el contrato que le dice a Kubernetes cuántas réplicas puedes permitirte perder a la vez por disrupciones voluntarias (drains, evictions del autoscaler, upgrades):

```
apiVersion: policy/v1
kind: PodDisruptionBudget
metadata:
  name: api-pdb
spec:
  minAvailable: 2
  unhealthyPodEvictionPolicy: AlwaysAllow
  selector:
    matchLabels:
      app: api
```

Con minAvailable: 2 y tres réplicas, un kubectl drain solo podrá desalojar un pod a la vez y esperará a que el reemplazo esté Ready antes de tocar el siguiente. El campo unhealthyPodEvictionPolicy: AlwaysAllow (estable desde Kubernetes 1.31) evita el clásico bloqueo en el que un pod que nunca llegó a estar Ready impide drenar el nodo: los pods no sanos se pueden desalojar siempre, porque protegerlos no protege ninguna disponibilidad real.

Dos matices importantes. Primero: el PDB no aplica a disrupciones involuntarias —un kernel panic o la pérdida física del nodo no piden permiso—, para eso están las réplicas repartidas con topologySpreadConstraints entre nodos y zonas. Segundo: el PDB tampoco aplica a los rolling updates del Deployment; eso lo gobiernan maxSurge y maxUnavailable en la estrategia de despliegue. La configuración conservadora habitual es maxSurge: 1, maxUnavailable: 0: primero se crea el pod nuevo, y solo cuando está Ready se termina el viejo. Combinado con el PDB, tienes cubiertos los dos caminos por los que Kubernetes mata pods a propósito.

Para los nodos spot, el aviso de interrupción llega por el plano del proveedor (el instance metadata service en AWS). Herramientas como el AWS Node Termination Handler o el modo de interrupciones de Karpenter convierten ese aviso en un drain ordenado del nodo: cordon, eviction respetando PDBs y, en cada pod, la misma

secuencia de terminación que ya implementaste. Si tu shutdown cabe en el aviso de dos minutos, un nodo spot desapareciendo es un no-evento.

El drenado fuera del clúster: ALB, NLB, ingress y los segundos que no controlas

La secuencia de terminación de Kubernetes termina en el borde del clúster. Si delante tienes un balanceador gestionado, hay otra máquina de estados que también tiene que enterarse de que tu pod se va —y sus tiempos por defecto rara vez coinciden con los tuyos.

Con el AWS Load Balancer Controller en modo IP, cada pod es un target del target group. Cuando el pod entra en Terminating, el controller inicia la deregistración, y el ALB deja de enviar peticiones nuevas pero mantiene las conexiones existentes durante deregistration_delay, que por defecto es de 300 segundos. Cinco minutos. Si tu grace period es de 60 segundos, el ALB puede seguir considerando el target «draining» mucho después de que el pod haya muerto. Ajústalo a algo coherente con tu drenado real:

```
apiVersion: networking.k8s.io/v1
kind: Ingress
metadata:
  annotations:
    alb.ingress.kubernetes.io/target-group-attributes: >
      deregistration_delay.timeout_seconds=30
```

El movimiento complementario son los pod readiness gates: etiquetando el namespace con elbv2.k8s.aws/pod-readiness-gate-inject: enabled, el controller inyecta una condición extra en cada pod que no se cumple hasta que el target está registrado y sano en el ALB. Sin esto, durante un rolling update Kubernetes puede considerar Ready al pod nuevo —y matar al viejo— antes de que el ALB sea capaz de enviarle tráfico: una ventana de 502s que ningún preStop arregla, porque el problema está en el lado de las altas, no en el de las bajas.

Si usas ingress-nginx, el equivalente al drenado vive en worker_shutdown_timeout (240 segundos por defecto): cuánto esperan los workers de NGINX a que las conexiones activas terminen durante un reload o apagado. Y recuerda que el propio controller de ingress es un pod más, con su propia secuencia de terminación: las versiones recientes ya incorporan el retardo y el drenado, pero si has tocado sus probes o su grace period, revisa que no lo hayas roto.

En GKE con container-native load balancing, el ajuste equivalente es el BackendConfig: connectionDraining.drainingTimeoutSec cumple el papel del deregistration delay. Y un último detalle que afecta a los Services tipo LoadBalancer: externalTrafficPolicy. Con Cluster (el valor por defecto), cualquier nodo reenvía tráfico a cualquier pod y las retiradas se propagan vía kube-proxy en todos los nodos; con Local solo reciben tráfico los nodos con pods locales, lo que preserva la IP de origen y ahorra un salto, pero acopla la salud del nodo a la de tus pods: al drenar los últimos pods de un nodo, el health check del balanceador hacia ese nodo también tiene que fallar a tiempo.

Go: apagado elegante con http.Server.Shutdown

La serie cubre Python y Node en los ejemplos principales, pero muchos servicios de plataforma están escritos en Go, y su librería estándar trae el drenado incorporado desde Go 1.8: http.Server.Shutdown deja de aceptar conexiones nuevas, espera a que las peticiones en curso terminen y cierra las conexiones keep-alive inactivas — exactamente la semántica que en Node hay que montar a mano.

```
package main

import (
        "context"
        "log"
        "net/http"
        "os/signal"
        "syscall"
        "time"
)

func main() {
        sigCtx, stop := signal.NotifyContext(
                context.Background(), syscall.SIGTERM, syscall.SIGINT)
        defer stop()

        srv := &http.Server{Addr: ":8080", Handler: routes()}

        go func() {
                if err := srv.ListenAndServe(); err != http.ErrServerClosed {
                        log.Fatalf("servidor: %v", err)
                }
        }()

        <-sigCtx.Done() // SIGTERM recibido
        markNotReady()  // readiness pasa a 503

        // tope al drenado: debe caber en el grace period
        ctx, cancel := context.WithTimeout(
                context.Background(), 25*time.Second)
        defer cancel()

        if err := srv.Shutdown(ctx); err != nil {
                log.Printf("drenado incompleto: %v", err)
        }
        closePools() // pools de BD, clientes, flush de métricas
}
```

Tres detalles que marcan la diferencia. Primero, signal.NotifyContext convierte la señal en un context cancelado: idiomático y componible con el resto de tu código. Segundo, el context con timeout que recibe Shutdown es tu tope de drenado —el equivalente al --timeout-graceful-shutdown de Uvicorn— y debe caber en el grace period tras descontar el preStop. Tercero, Shutdown no espera a las conexiones «secuestradas» (WebSockets vía Hijack): para esas, registra limpieza propia con srv.RegisterOnShutdown y ciérralas tú, como vemos en la siguiente sección.

Conexiones que no terminan solas: gRPC, WebSockets y SSE

Todo el modelo de drenado anterior asume que las peticiones terminan solas si esperas lo suficiente. Las conexiones de larga vida rompen esa premisa: un stream gRPC, un WebSocket o una conexión SSE pueden durar horas. Para ellas, drenar no es esperar: es avisar, cortar con orden y confiar en la reconexión del cliente.

En gRPC el mecanismo nativo es el frame GOAWAY de HTTP/2. GracefulStop() en grpc-go deja de aceptar streams nuevos, envía GOAWAY y espera a que los RPCs en vuelo terminen; como puede esperar para siempre, se combina con un temporizador que fuerza Stop() si el drenado no acaba a tiempo:

```
done := make(chan struct{})
go func() {
        grpcServer.GracefulStop() // GOAWAY + espera RPCs en vuelo
        close(done)
}()
select {
case <-done:
case <-time.After(20 * time.Second):
        grpcServer.Stop() // corta lo que quede
}
```

Para los streams eternos, la defensa es preventiva: keepalive.ServerParameters con MaxConnectionAge y MaxConnectionAgeGrace recicla las conexiones cada pocos minutos, de modo que ningún cliente lleva horas pegado al mismo pod cuando llega el momento de morir. Es la misma idea que rotar conexiones en un pool: limita el radio de impacto de cualquier terminación.

Con WebSockets, el apagado correcto es enviar un close frame con código 1001 («going away») a cada conexión, dar un par de segundos a que los clientes lo procesen y cerrar. El trabajo de verdad está en el cliente: lógica de reconexión con backoff y jitter —la guía de reintentos de esta serie aplica literalmente— y re-suscripción al estado que tuviera. Un cliente WebSocket sin reconexión automática convierte cada deploy en una sesión rota.

SSE es el caso amable: EventSource se reconecta solo. Tu servidor decide el retardo con el campo retry: y, si numera los eventos con id:, el navegador reenvía Last-Event-ID al reconectar, lo que te permite retomar el stream sin perder mensajes. En el apagado basta con cerrar las conexiones SSE sin dramatismo tras marcar el pod como no listo: los clientes volverán… al pod nuevo. El error típico es un balanceador con timeout de idle más corto que tu heartbeat de SSE, que convierte las reconexiones en rutina permanente en lugar de excepción de deploy.

Sidecars: cuando el proxy muere antes que tu aplicación

Durante años, el problema más irritante del graceful shutdown en mallas de servicio fue el orden de muerte: Kubernetes enviaba SIGTERM a todos los contenedores del pod a la vez, el sidecar de red (istio-proxy, linkerd-proxy) moría en segundos, y tu aplicación —que seguía drenando peticiones con la calma que le diste— se quedaba sin red. Las llamadas salientes de esos últimos segundos (confirmar un mensaje, escribir en la base de datos, avisar a otro servicio) fallaban justo durante cada deploy.

Los contenedores sidecar nativos resuelven esto de raíz. Desde Kubernetes 1.28 (estables en 1.33), un sidecar se declara como initContainer con restartPolicy: Always:

```
spec:
  initContainers:
  - name: proxy
    image: mi-proxy:1.2
    restartPolicy: Always   # esto lo convierte en sidecar
  containers:
  - name: api
    image: mi-api:2.4.0
```

La semántica es exactamente la que un proxy necesita: arranca antes que los contenedores principales, y en la terminación el kubelet primero espera a que los contenedores principales terminen y solo después apaga los sidecars, en orden inverso al de arranque. Tu aplicación conserva la red durante todo su drenado. Además, en Jobs el pod ya no se queda colgado en Running porque el sidecar «no sabía» que debía morir: cuando el contenedor principal acaba, el sidecar se apaga solo y el Job completa.

Si aún no puedes usar sidecars nativos, las mallas traen paliativos. En Istio, TERMINATION_DRAIN_DURATION_SECONDS (5 segundos por defecto) alarga la espera del proxy antes de cerrar, y la opción EXIT_ON_ZERO_ACTIVE_CONNECTIONS hace que Envoy espere a quedarse sin conexiones activas antes de salir. En Linkerd, los Jobs pueden avisar al proxy explícitamente con un POST a localhost:4191/shutdown al terminar su trabajo. Son soluciones honestas pero frágiles comparadas con el orden de terminación garantizado por el kubelet: si tu clúster está en 1.33 o posterior, migra a sidecars nativos y borra los workarounds.

Kafka, Celery y SQS: apagado elegante en consumidores reales

El patrón del worker genérico —deja de consumir, termina lo que tienes, confirma y sal— se concreta distinto según el sistema de colas. Los tres de más uso tienen sus trampas propias.

En Kafka, lo crítico es salir del grupo de consumidores con orden. Un consumidor que muere por SIGKILL no avisa: el broker tarda session.timeout.ms en darlo por muerto y, mientras, sus particiones están huérfanas. Llamar a consumer.close() en el apagado confirma los offsets pendientes y dispara el rebalanceo inmediatamente:

```
from confluent_kafka import Consumer
import signal

running = True
signal.signal(signal.SIGTERM, lambda s, f: globals().update(running=False))

consumer = Consumer({
    "bootstrap.servers": "kafka:9092",
    "group.id": "pedidos",
    "enable.auto.commit": False,
    "partition.assignment.strategy": "cooperative-sticky",
})
consumer.subscribe(["pedidos"])

try:
    while running:
        msg = consumer.poll(timeout=1.0)
        if msg is None or msg.error():
            continue
        procesar(msg)              # terminar el mensaje actual
        consumer.commit(msg)       # commit SOLO tras procesar
finally:
    consumer.close()  # confirma, abandona el grupo, rebalancea ya
```

El detalle de partition.assignment.strategy: cooperative-sticky merece mención: con la estrategia clásica (eager), cada salida de un consumidor detiene el grupo entero mientras se reasignan todas las particiones; con el rebalanceo cooperativo solo se mueven las particiones del que se va. En un deploy rolling de veinte workers, la diferencia es entre veinte micro-pausas globales y ninguna. Vigila también max.poll.interval.ms: si tu drenado tarda más que ese intervalo (5 minutos por defecto), el broker te expulsa del grupo antes de que termines.

En Celery, SIGTERM ya inicia un warm shutdown: el worker deja de aceptar tareas nuevas y termina las que tiene en curso. Lo que casi nadie configura es task_acks_late = True: sin él, Celery confirma el mensaje al recibirlo, y una tarea matada a medias simplemente se pierde. Con acks_late, la tarea interrumpida vuelve a la cola —lo que, de nuevo, exige tareas idempotentes. Y dimensiona el grace period según tu tarea más lenta, no según tu petición HTTP más lenta: son presupuestos muy distintos.

Con SQS no existe el ack explícito: un mensaje recibido se vuelve invisible durante el visibility timeout (30 segundos por defecto) y reaparece si nadie lo borra a tiempo. El apagado elegante consiste en dejar de pedir mensajes nuevos, terminar y borrar los que tienes, y salir. Si te matan a medias, el mensaje reaparece solo: el sistema es auto-correctivo siempre que el visibility timeout supere tu tiempo de proceso máximo y tus consumidores sean idempotentes. El anti-patrón es el contrario: un visibility timeout de 30 segundos con trabajos de 2 minutos genera procesamiento duplicado constante, con o sin deploys.

Cómo probar el graceful shutdown antes de que producción lo haga por ti

El graceful shutdown es el típico código que nunca se ejecuta en los tests y siempre se ejecuta en producción. Merece pruebas explícitas en tres niveles.

Nivel uno, local y en segundos: docker stop (que envía SIGTERM, espera 10 segundos y remata con SIGKILL) contra tu contenedor mientras le mantienes una petición lenta abierta con curl. Si la petición muere con la conexión cortada, tu handler no funciona o la señal no llega (revisa el PID 1). Este test detecta el 80% de los problemas sin tocar un clúster.

Nivel dos, automatizado en CI. Un test de integración que arranca el servicio real como subproceso, lanza una petición lenta, envía SIGTERM en mitad y comprueba dos cosas: que la petición completa con 200 y que el proceso sale con código 0 dentro del plazo:

```
import signal, subprocess, time, threading
import httpx

def test_sigterm_drena_peticiones_en_vuelo():
    proc = subprocess.Popen(["uvicorn", "app:app", "--port", "8123",
                             "--timeout-graceful-shutdown", "10"])
    time.sleep(1.5)  # esperar el arranque

    estado = {}
    def peticion_lenta():
        # el endpoint tarda ~3 s en responder
        r = httpx.get("http://localhost:8123/lento", timeout=15)
        estado["code"] = r.status_code

    t = threading.Thread(target=peticion_lenta)
    t.start()
    time.sleep(0.5)              # la petición ya está en vuelo
    proc.send_signal(signal.SIGTERM)

    t.join(timeout=15)
    assert estado.get("code") == 200      # se completó, no se cortó
    assert proc.wait(timeout=15) == 0     # salida limpia, sin SIGKILL
```

Nivel tres, el ensayo general: una prueba de carga durante un despliegue real. Lanza k6 o vegeta contra el entorno de staging con tráfico sostenido y, a mitad de la prueba, ejecuta kubectl rollout restart deployment/api. El criterio de éxito es binario: cero respuestas no-2xx durante toda la ventana. Si aparecen 502s, ya sabes en qué capa buscar según el momento: al inicio de la terminación es propagación de endpoints (sube el sleep del preStop o revisa los readiness gates), al final es presupuesto agotado (grace period frente a drenado). Automatiza este escenario y conviértelo en la puerta de calidad de los cambios de infraestructura: es la única prueba que valida todas las capas juntas.

Observabilidad del apagado: demostrar que funciona

Sin métricas, el graceful shutdown se degrada en silencio: alguien acorta el grace period, un sidecar nuevo cambia el orden de muerte, y nadie lo nota hasta la siguiente revisión de incidencias. Tres instrumentos bastan para vigilarlo.

Primero, un gauge de peticiones en vuelo y un log estructurado por fase del apagado, con marcas de tiempo:

```
from prometheus_client import Gauge
import structlog, time

log = structlog.get_logger()
inflight = Gauge("http_inflight_requests",
                 "Peticiones actualmente en proceso")

async def shutdown_sequence(app):
    t0 = time.monotonic()
    log.info("shutdown.start", inflight=inflight._value.get())
    marcar_no_listo()
    log.info("shutdown.not_ready", t=time.monotonic() - t0)
    await drenar_peticiones()
    log.info("shutdown.drained", t=time.monotonic() - t0)
    await cerrar_pools()
    log.info("shutdown.cleanup_done", t=time.monotonic() - t0)
```

Con esa línea temporal en los logs, diagnosticar un apagado lento deja de ser arqueología: ves exactamente qué fase consume el presupuesto. Si shutdown.drained tarda 24 de tus 25 segundos, tu problema son peticiones lentas, no la limpieza.

Segundo, vigila los SIGKILL. Un contenedor que termina con exit code 137 (128 + 9) fue rematado: o el grace period se quedó corto o tu drenado se colgó. Una alerta sobre reinicios con ese código en tus servicios web equivale a una alerta de «el graceful shutdown está roto». La mayoría de agentes de observabilidad (kube-state-metrics incluido) exponen la razón de terminación del último contenedor, así que la consulta es trivial.

Tercero, el panel de la verdad: tasa de respuestas 5xx medida desde el balanceador —no desde el pod, que no ve las peticiones que nunca le llegaron— superpuesta con los eventos de deploy. La firma de un problema de drenado es inconfundible: picos de 502/503 de menos de un minuto, perfectamente alineados con cada rollout. Ese panel, mirado una vez por semana, es la diferencia entre descubrir la regresión en el acto y convivir con ella durante meses.

Pools de conexiones y transacciones: el orden exacto de la limpieza

«Cierra los recursos después de drenar» es fácil de decir; el matiz está en qué significa cerrar un pool con trabajo a medias. Un pool de PostgreSQL como asyncpg o psycopg_pool tiene tres estados posibles al recibir la orden de cierre: conexiones ociosas (se cierran sin más), conexiones prestadas ejecutando una consulta (hay que esperarlas) y conexiones con una transacción abierta (el caso delicado).

La regla general: el cierre del pool debe ser el espejo del drenado HTTP. asyncpg con pool.close() espera a que las conexiones prestadas vuelvan —equivalente al server.close() de Node—, así que si ya drenaste las peticiones, el pool cierra en milisegundos. El anti-patrón es el cierre con timeout cero o terminate(): corta las conexiones con transacciones abiertas, y PostgreSQL hará rollback de todas ellas. No pierdes consistencia (para eso son las transacciones), pero sí el trabajo, y el usuario que recibió un 200 con su escritura a medio confirmar tiene ahora un problema que ningún reintento arregla.

Si usas PgBouncer u otro pooler intermedio, recuerda que también es un servicio con su propio apagado: la orden PAUSE espera a que las transacciones activas terminen y detiene la asignación de nuevas, y SHUTDOWN

WAIT_FOR_SERVERS drena antes de salir. Un PgBouncer matado sin drenar corta todas las sesiones de todas tus aplicaciones a la vez: su graceful shutdown importa más que el de cualquier servicio individual.

Para las escrituras que deben sobrevivir a cualquier interrupción, la respuesta no está en el apagado sino en el diseño: transacciones que agrupan la operación completa y, para los efectos secundarios (publicar eventos, llamar a otros servicios), el patrón transactional outbox que cubrimos en esta misma serie. El apagado elegante reduce la frecuencia de los cortes a mitad de operación; el outbox hace que dejen de importar.

StatefulSets y procesos con estado: traspasar antes de morir

Todo lo anterior asume servicios sin estado, donde cualquier réplica puede atender cualquier petición. Los procesos con estado añaden un paso al apagado: traspasar la responsabilidad antes de salir.

En un StatefulSet, la terminación durante un rolling update es ordenada e inversa (del ordinal más alto al más bajo), y cada pod conserva su identidad y su volumen. El patrón crítico es el del líder: si el pod que muere es el líder de un grupo de consenso o el primario de una base de datos, esperar pasivamente el SIGTERM es mala estrategia, porque el failover automático tarda (segundos de detección más elección). Los sistemas bien diseñados traspasan el liderazgo proactivamente en el preStop o en el handler de SIGTERM: Kafka con el controlled shutdown mueve el liderazgo de particiones antes de salir; Patroni ofrece el switchover explícito para el primario de PostgreSQL; etcd transfiere el liderazgo del raft. El objetivo es convertir un failover (con ventana de indisponibilidad) en un handover (sin ella).

Para trabajos largos con estado en memoria —entrenamientos de modelos, procesos batch de horas—, el apagado elegante significa checkpoint: al recibir la señal, guarda el progreso en almacenamiento duradero y sal; el reemplazo retoma desde el último checkpoint. La frecuencia del checkpoint la dicta el presupuesto: con nodos spot y dos minutos de aviso, un checkpoint que tarda cinco minutos en escribirse es un checkpoint que nunca se completará. Dimensiona el intervalo para que perder el trabajo desde el último punto guardado sea aceptable, porque tarde o temprano lo perderás.

Autoscaling: cuando el apagado deja de ser un evento y se vuelve rutina

Con un HPA (Horizontal Pod Autoscaler) activo, los pods no mueren solo en los deploys: mueren cada vez que baja la carga. Un servicio que escala entre 5 y 50 réplicas a lo largo del día ejecuta decenas de terminaciones diarias sin que nadie despliegue nada. Eso cambia el cálculo: un bug de apagado que en un servicio estático aparece una vez por semana, con autoscaling aparece cincuenta veces al día.

Dos ajustes mitigan el vaivén. En el propio HPA, el campo behavior.scaleDown con stabilizationWindowSeconds (300 por defecto) y políticas de ritmo evita que una caída puntual de tráfico dispare una ola de terminaciones que habrá que deshacer en minutos. Y en los pods, un coste de arranque alto (caches que calentar, modelos que cargar) es un argumento para escalar hacia abajo despacio: matar es barato, volver a crear no.

Con KEDA y el escalado a cero, el apagado es literalmente constante: la última réplica muere cada vez que la cola se vacía. Ahí el patrón del worker cobra otra dimensión: si el desescalado llega con un mensaje a medio procesar, todo el modelo de acks tardíos e idempotencia es lo único que evita perder o duplicar trabajo. KEDA

además respeta el ciclo normal de terminación, así que tu handler de SIGTERM es el mismo; lo que cambia es la frecuencia con la que se ejercita. En resumen: el autoscaling convierte el graceful shutdown de una preocupación de deploys en una propiedad operativa permanente, y las métricas de la sección de observabilidad pasan de «panel de deploys» a «panel de todos los días».

Cuando el graceful shutdown no basta: diseña para el crash

Una advertencia honesta para cerrar el círculo: el apagado elegante es una optimización de probabilidad, no una garantía. Hay terminaciones que no dan opción a limpieza. Un OOMKill es SIGKILL inmediato, sin SIGTERM previo: si tu servicio muere por memoria, ningún handler se ejecuta (y lo verás como exit code 137 con razón OOMKilled, distinguible del SIGKILL por grace period agotado). Un kernel panic, la pérdida del nodo o un fallo de hardware tampoco avisan. Y un bug que cuelga tu handler de apagado convierte cada terminación amable en una brutal con retardo.

La conclusión de ingeniería es que el graceful shutdown reduce la frecuencia de los finales abruptos, pero la corrección del sistema no puede depender de él. Las piezas que garantizan corrección son las de siempre: transacciones para la atomicidad local, idempotencia para los reintentos, acks tardíos para no perder mensajes, outbox para los efectos secundarios, y checkpoints para el trabajo largo. Si tu sistema solo es correcto cuando los apagados son elegantes, no es correcto.

Esta mentalidad, conocida como crash-only software, sugiere una prueba reveladora: mata a tus servicios con SIGKILL en staging de vez en cuando y verifica que no se pierde ni duplica nada. Si el resultado es aceptable, el graceful shutdown es lo que debe ser —una mejora de experiencia que elimina los errores visibles del caso común—, no una muleta que oculta un diseño frágil. Las dos cosas se refuerzan: el crash-safety te da corrección; el apagado elegante te da deploys invisibles.

Depuración: el pod que no muere

El problema inverso también existe: pods clavados en Terminating durante minutos u horas. La caja de herramientas para diagnosticarlo es corta y va por orden de frecuencia.

Primero, ¿está el proceso ignorando la señal? kubectl describe pod te dice cuánto lleva en Terminating; si supera el grace period más unos segundos, algo raro pasa a nivel de kubelet o de finalizers. Si aún está dentro del plazo, entra al contenedor y mira: ps aux para confirmar quién es PID 1, y si es un shell, ya tienes al culpable de siempre.

Segundo, los finalizers. Un pod (o cualquier objeto) con finalizers pendientes no desaparece hasta que el controlador responsable los retira. kubectl get pod -o yaml y busca metadata.finalizers: si hay entradas de un operador o una malla que ya no funciona, el pod quedará en Terminating para siempre. La salida limpia es arreglar el controlador; editar los finalizers a mano es el último recurso, porque saltarse la limpieza que representaban puede dejar recursos huérfanos (volúmenes adjuntos, IPs, registros externos).

Tercero, los procesos zombi. Si tu contenedor lanza subprocesos y PID 1 no los cosecha (no llama a wait), los hijos muertos se acumulan como zombis y algunos runtimes esperan indefinidamente. tini o el init: true de Docker Compose existen exactamente para esto; en Kubernetes, shareProcessNamespace: true permite además que un contenedor del pod actúe de init para los demás. Y el último recurso de verdad: kubectl delete pod --force

--grace-period=0 no mata el proceso —solo borra el objeto de la API y deja al kubelet la limpieza en segundo plano—, así que puede dejar el proceso corriendo en el nodo un rato más. Con StatefulSets, ese «un rato más» puede significar dos pods con la misma identidad: úsalo solo cuando el nodo ya está perdido.

Otros runtimes: Spring Boot y .NET en dos minutos

El patrón es idéntico en todos los ecosistemas; cambia dónde está el interruptor. En Spring Boot, el drenado viene de serie desde la versión 2.3 y se activa con una línea de configuración:

```
# application.properties
server.shutdown=graceful
spring.lifecycle.timeout-per-shutdown-phase=25s
```

Con eso, el SIGTERM hace que Tomcat (o Netty en aplicaciones reactivas) deje de aceptar conexiones nuevas y espere a las peticiones activas hasta el timeout configurado. Los beans se destruyen después del drenado, en el orden inverso a su creación, así que los pools de conexiones (HikariCP) se cierran cuando ya no hay peticiones que los usen: el orden correcto sale gratis del ciclo de vida del contenedor de Spring.

En .NET, el equivalente es el IHostApplicationLifetime: ApplicationStopping se dispara al recibir la señal (ahí marcas el readiness como no listo y dejas de aceptar trabajo), el host espera a que los servidores drenen, y ApplicationStopped llega al final para la limpieza. El tope se ajusta con HostOptions.ShutdownTimeout, cuyo valor por defecto —30 segundos en las versiones recientes— tiene que caber, una vez más, dentro del grace period de Kubernetes tras descontar el preStop. Sea cual sea el runtime, la checklist no cambia: señal que llega, readiness que cae, drenado con tope, limpieza al final y presupuesto que lo cubre todo.

Jobs y CronJobs: terminar bien también es apagarse bien

Las cargas batch tienen su propia versión de cada problema de esta guía. Un pod de Job puede ser desalojado por un drain o una interrupción de spot como cualquier otro, y recibe el mismo SIGTERM: un proceso batch que hace checkpoint y sale con código distinto de cero será reintentado según backoffLimit, así que asegúrate de que una ejecución interrumpida y su reintento no apliquen el trabajo dos veces —idempotencia, otra vez. Si el trabajo no debe superar nunca una duración, activeDeadlineSeconds lo termina (SIGTERM incluido) al agotarse el presupuesto; y suspend: true permite pausar un Job limpiamente antes de un mantenimiento planificado, en lugar de dejar que el drain lo interrumpa en un punto aleatorio.

Los CronJobs añaden la pregunta de la concurrencia: si la ejecución anterior no ha terminado cuando toca la siguiente, decide concurrencyPolicy. Forbid salta la nueva ejecución; Replace termina la vieja —con la secuencia exacta de SIGTERM de esta guía— y arranca de cero. Replace es, en la práctica, un graceful shutdown programado: si tu batch maneja bien la señal, Replace es seguro; si no, cada solapamiento de horario es una pequeña avería. Y recuerda la nota de sidecars de antes: con sidecars nativos, los pods batch llegan de verdad a Succeeded en lugar de colgarse para siempre porque un proxy se negaba a salir.

El arranque es parte del apagado: probes, minReadySeconds y readiness honesta

Suena paradójico, pero la mitad de un deploy sin errores ocurre en el arranque: el pod viejo solo puede irse sin daño si el nuevo está listo de verdad. Kubernetes considera Ready a un pod cuando pasa su readiness probe; la trampa es una probe que pasa demasiado pronto —el puerto está abierto pero la caché está fría, el modelo no está cargado, el pool de conexiones está vacío. El resultado es un deploy sin 5xx pero con un pico de latencia: técnicamente vivo, prácticamente no listo.

Tres herramientas hacen honesta a la readiness. La startupProbe cubre a los que arrancan lento: hasta que no pasa, ni liveness ni readiness se evalúan, así que un servicio que tarda 90 segundos en calentar no necesita un initialDelaySeconds grotesco en las demás probes. La readiness probe debe verificar lo que servir requiere de verdad: dependencias alcanzables, cachés suficientemente calientes, pool inicializado —no solo «el proceso responde». Y minReadySeconds (en el spec del Deployment) añade un margen final: el pod debe mantenerse Ready ese número de segundos antes de que el rollout lo cuente como disponible y pase a matar al siguiente pod viejo. Es un seguro barato contra pods que pasan la probe una vez y entran en crash-loop: sin él, un rollout rápido puede reemplazar todas las réplicas sanas por rotas antes de que salte la primera alerta. Un arranque bien afinado hace aburrido cada apagado del otro lado del rollout —que es el objetivo de toda esta guía.

Caso práctico: la línea de tiempo de un deploy sin errores

Para atar todas las piezas, este es el minuto y medio de vida de un rollout bien configurado en un servicio real: FastAPI detrás de un ALB en modo IP, tres réplicas, preStop de 8 segundos, grace period de 60, deregistration delay de 30 y readiness gates activados.

- T+0 — kubectl rollout restart. Con maxSurge: 1 y maxUnavailable: 0, Kubernetes crea primero un pod nuevo. Nadie ha matado nada todavía.
- T+12 — El pod nuevo pasa su readiness probe, el controller lo registra en el target group y espera a que el ALB lo declare healthy. Solo entonces el readiness gate se cumple y el pod cuenta como Ready.
- T+15 — Empieza la terminación del pod viejo: pasa a Terminating, sale de los endpoints y el ALB inicia su deregistración (deja de enviarle peticiones nuevas). En paralelo arranca el preStop: 8 segundos de sleep durante los cuales el pod sigue atendiendo con total normalidad las peticiones rezagadas que aún le llegan.
- T+23 — SIGTERM. El handler marca shutting_down = True; la readiness probe empieza a devolver 503. Uvicorn deja de aceptar conexiones nuevas y drena las que están en vuelo. La más lenta tarda 14 segundos.
- T+38 — Drenado completo. Se cierran el pool de PostgreSQL y el cliente HTTP, se hace flush de métricas y logs: 3 segundos.
- T+41 — exit 0. El kubelet ve la salida limpia, muy por debajo del presupuesto de 60 segundos. El pod desaparece. El rollout repite el ciclo con la siguiente réplica.

El antes y el después de este equipo es representativo: antes de tocar nada, cada deploy generaba entre 30 y 60 respuestas 502 (medidas en el ALB) y una alerta de ruido por pico de errores; el post-mortem reveló CMD en forma shell, sin preStop, con el deregistration delay en sus 300 segundos por defecto. Tras aplicar exactamente lo descrito en esta guía, los siguientes cincuenta deploys sumaron cero errores atribuibles a la terminación. Ningún cambio de código de negocio: solo apagado bien hecho.

Checklist de producción

La lista corta para revisar cada servicio antes de darlo por listo:

- El proceso recibe SIGTERM de verdad: CMD en forma exec o tini como init; verificado con ps aux dentro del contenedor.
- preStop con sleep de 5–10 segundos, y terminationGracePeriodSeconds ampliado para cubrirlo con margen.
- Grace period calculado por partes: preStop + petición/mensaje más lento + limpieza + margen. Nunca el 30 por defecto con preStop añadido.
- Readiness separada de liveness; readiness falla inmediatamente tras SIGTERM con periodSeconds y failureThreshold agresivos.
- Drenado con tope explícito (timeout-graceful-shutdown, server.close + temporizador, srv.Shutdown con context) que cabe en el presupuesto.
- Recursos (pools, clientes, flush) cerrados después de drenar, nunca antes.
- Conexiones keep-alive cerradas explícitamente; conexiones largas (gRPC, WebSocket) con aviso, tope de edad y clientes que reconectan.
- Balanceador ajustado: deregistration delay coherente con el grace period y readiness gates activados si registras pods como targets.
- PDB con unhealthyPodEvictionPolicy: AlwaysAllow, topologySpreadConstraints, y maxSurge/maxUnavailable conservadores.
- Workers: dejar de consumir al recibir la señal, terminar y confirmar tarde (acks_late, commit tras procesar, visibility timeout dimensionado); consumidores idempotentes.
- Sidecars nativos (initContainer con restartPolicy: Always) en clústeres 1.33+, o drenado del proxy configurado en la malla.
- Probado: test de SIGTERM en CI y prueba de carga durante rollout con cero errores como criterio.
- Observado: gauge de in-flight, logs por fase del apagado, alerta sobre exit code 137 y panel de 5xx del balanceador alineado con deploys.

Referencia rápida: qué versión de Kubernetes trae cada pieza

Para saber qué puedes usar según tu clúster:

- 1.28 — Contenedores sidecar nativos (initContainers con restartPolicy: Always), en alpha.
- 1.29 — Sidecars nativos en beta y activados por defecto.
- 1.30 — Acción sleep nativa en preStop (beta): adiós al shell en imágenes distroless.
- 1.31 — unhealthyPodEvictionPolicy: AlwaysAllow en los PDB, estable.
- 1.32 — Acción sleep estable; sleep con cero segundos permitido como no-op configurable.
- 1.33 — Sidecars nativos estables: la solución definitiva al orden de muerte proxy/aplicación.

Si tu clúster va por detrás, cada sección de esta guía indica la alternativa (exec con sh -c "sleep", workarounds de malla, etc.).

Preguntas frecuentes

¿Necesito todo esto para un servicio interno pequeño? No entero. El mínimo digno para cualquier servicio HTTP: forma exec en el CMD, un preStop de 5 segundos, manejo de SIGTERM con drenado y el grace period revisado. Son veinte líneas entre Dockerfile, manifiesto y aplicación, y eliminan la mayoría de los 502 de deploy. El resto de capas se añaden cuando hay balanceadores gestionados, colas o conexiones largas.

¿Por qué sigo viendo 502 con todo configurado? Mide dónde: si los errores aparecen al principio de la terminación, la retirada del endpoint no se propagó a tiempo (sube el sleep del preStop, revisa readiness gates y deregistration delay); si aparecen al final, el presupuesto se agotó (drenado más largo que el grace period restante). El panel de 5xx del balanceador con los eventos de deploy superpuestos responde esta pregunta en segundos.

¿SIGINT o SIGTERM? Kubernetes envía SIGTERM. Maneja ambos con el mismo código —SIGINT te da la misma semántica en local con Ctrl+C— como hacen los ejemplos de Go y Python de esta guía.

¿Un preStop largo no ralentiza mis deploys? Sí: cada réplica tarda al menos preStop + drenado en morir, y un rollout de veinte réplicas lo nota. Es un intercambio deliberado de velocidad por cero errores. Si duele, sube maxSurge para reemplazar más pods en paralelo en lugar de recortar el apagado.

¿Cómo pruebo esto sin clúster? docker stop es tu simulador de Kubernetes de bolsillo: SIGTERM, espera (ajustable con -t) y SIGKILL. Si tu contenedor sobrevive dignamente a docker stop con una petición lenta en vuelo, el 80% del trabajo está hecho.

Conclusión

El graceful shutdown no es una optimización: es la diferencia entre deploys invisibles y deploys que tus usuarios notan. La receta completa cabe en una frase: retrasa el SIGTERM con un preStop, deja de anunciarte como listo, drena lo que está en vuelo, cierra recursos al final y asegúrate de que la señal de verdad llega a tu proceso. Impleméntalo una vez por servicio y cada deploy, escalado o reemplazo de nodo pasará sin que nadie lo note —que es exactamente el objetivo.

ENGLISH

Graceful Shutdown in Kubernetes: Deploys Without Errors or Dropped Requests

Why every deploy might be dropping requests

Every time you ship a new version, Kubernetes kills pods. The same happens when the autoscaler scales down, when a spot node disappears, or when the scheduler rebalances workloads. In a system that deploys frequently, pod termination is not a rare event — it is daily routine.

If your service does not handle that shutdown well, the symptoms are familiar: 502/503 errors during every deploy, connections cut off mid-response, half-finished transactions, and queue messages processed twice. The insidious part is that everything works in testing — where nobody kills the process — and fails precisely in production, at the worst possible moment.

The good news: graceful shutdown is a well-understood pattern. You just have to implement it at the three right layers: the platform (Kubernetes), the process (your application), and the dependencies (connections, queues, and in-flight work).

The termination sequence in Kubernetes

When Kubernetes decides to remove a pod, this is what happens:

- The pod enters the Terminating state and is removed from the Service endpoints.
- In parallel, the preStop hook runs (if defined) and then SIGTERM is sent to the main process of each container.
- The terminationGracePeriodSeconds budget (30 seconds by default) starts ticking — and it covers both the preStop and your application shutdown.
- If the process is still alive when the budget runs out, it gets SIGKILL: no appeal, no cleanup.

The detail that breaks most deploys sits between steps 1 and 2: endpoint removal and SIGTERM happen in parallel, not in order. kube-proxy, ingress controllers, and external load balancers take seconds to learn that the pod should no longer receive traffic. During that window, your application — which may have already started shutting down — keeps receiving new requests. The result: connection refused and 502s.

The first fix: a preStop with sleep

The standard solution to that race condition is counterintuitive: before shutting down, wait. A preStop hook with a sleep of a few seconds delays the SIGTERM and gives the network infrastructure time to stop sending you traffic while your application keeps serving normally.

```
spec:
  terminationGracePeriodSeconds: 60
  containers:
  - name: api
    image: my-api:2.4.0
    lifecycle:
      preStop:
        sleep:
          seconds: 8
```

Since Kubernetes 1.30 there is a native sleep action (stable in 1.32); on older versions you use exec with ["sh", "-c", "sleep 8"], which requires the image to include a shell. Two important rules: the sleep must exceed the time your infrastructure needs to propagate endpoint removal (5–10 seconds is usually enough), and terminationGracePeriodSeconds must be increased accordingly, because the preStop consumes part of that budget.

Handling SIGTERM in Python (FastAPI + Uvicorn)

Uvicorn already catches SIGTERM and stops accepting new connections while finishing the ones in flight. Your job is twofold: mark the pod as "not ready" so load balancers take it out of rotation, and close resources (database pools, HTTP clients) after draining requests, not before.

```
import asyncio
import signal
from contextlib import asynccontextmanager
from fastapi import FastAPI, Response

shutting_down = False

@asynccontextmanager
async def lifespan(app: FastAPI):
    app.state.pool = await create_db_pool()

    def on_sigterm():
        global shutting_down
        shutting_down = True  # the readiness probe will start failing

    loop = asyncio.get_running_loop()
    loop.add_signal_handler(signal.SIGTERM, on_sigterm)

    yield  # the application serves requests

    # By now Uvicorn has drained in-flight requests:
    # it is finally safe to close resources.
    await app.state.pool.close()

app = FastAPI(lifespan=lifespan)

@app.get("/healthz/ready")
async def ready():
    if shutting_down:
        return Response(status_code=503)
    return {"status": "ok"}
```

Start Uvicorn with --timeout-graceful-shutdown 25 to cap the drain: if a request takes longer than 25 seconds, it gets closed anyway before SIGKILL arrives. That number must fit inside the grace period, minus the preStop.

Handling SIGTERM in Node.js (Express)

In Node the key method is server.close(): it stops accepting new connections and waits for active requests to finish. The classic problem is idle keep-alive connections, which keep the server open indefinitely. Since Node 18.2 there is closeIdleConnections() for exactly that.

```
const express = require("express");

const app = express();
let shuttingDown = false;

app.get("/healthz/ready", (req, res) => {
  res.status(shuttingDown ? 503 : 200).end();
});

const server = app.listen(8080);

process.on("SIGTERM", () => {
  shuttingDown = true; // 1. stop advertising as ready

  // 2. stop accepting new connections and drain active ones
  server.close(() => {
    console.log("Drain complete, exiting");
    process.exit(0);
  });

  // 3. close keep-alive sockets with no requests in flight
  server.closeIdleConnections();

  // 4. safety net: force exit before SIGKILL
  setTimeout(() => {
    console.error("Drain timeout, forcing shutdown");
    server.closeAllConnections();
    process.exit(1);
  }, 25000).unref();
});
```

The most common mistake here is calling process.exit(0) directly in the SIGTERM handler: that kills in-flight requests instantly, which is exactly what we are trying to avoid.

The PID 1 problem: when SIGTERM never arrives

If your application "ignores" SIGTERM and always dies by SIGKILL after 30 seconds, the signal is almost certainly not reaching it. The usual cause is the shell form of CMD in the Dockerfile: CMD npm start starts a shell as PID 1, and that shell does not forward signals to its child processes.

```
# BAD: the shell is PID 1 and swallows SIGTERM
CMD npm start

# GOOD: exec form, your process is PID 1
CMD ["node", "server.js"]

# Robust alternative: tini as init
RUN apk add --no-cache tini
ENTRYPOINT ["tini", "--"]
CMD ["node", "server.js"]
```

To verify: kubectl exec <pod> -- ps aux and check which process is PID 1. If it is sh, now you know why your graceful shutdown never runs.

Queue workers: stop consuming, not processing

For a worker, graceful shutdown means something different: on SIGTERM, stop taking new messages, finish the one you are working on, and acknowledge (ack) before exiting. Never ack before processing: if you get killed midway, the message is lost.

```
import signal

class Worker:
    def __init__(self, queue):
        self.queue = queue
        self.running = True
        signal.signal(signal.SIGTERM, self._stop)

    def _stop(self, signum, frame):
        self.running = False  # take no more messages

    def run(self):
        while self.running:
            msg = self.queue.receive(timeout=5)
            if msg is None:
                continue
            process(msg)  # finish the current message
            msg.ack()     # acknowledge ONLY after processing
        print("Worker stopped cleanly")
```

Size the grace period by your slowest message: if a job can take 90 seconds, a 30-second grace period guarantees jobs killed halfway. And since at-least-once delivery is unavoidable, your consumers must be idempotent — a topic we cover in the idempotency guide in this same series.

How to calculate terminationGracePeriodSeconds

The grace period is a shared budget that starts counting the moment the pod enters Terminating. Add it up piece by piece:

- preStop sleep: 8 s
- Draining the slowest request or message: 20 s

- Closing pools, flushing logs and metrics: 5 s
- Safety margin: 10 s

Total: 43 seconds; round up to 60. The default of 30 seconds falls short as soon as you add a preStop and have requests longer than a few seconds.

Common mistakes

- Adding the preStop without increasing the grace period: the sleep eats the budget and your application ends up SIGKILLed.
- Calling process.exit() or sys.exit() directly in the handler: turns a graceful shutdown into a brutal one.
- Closing the database before draining: the last requests fail with closed-connection errors.
- Shell-form CMD: SIGTERM never arrives and nothing else matters.
- Ignoring keep-alive connections: server.close() never resolves and you always exit via the timeout.
- Acking before processing in workers: lost messages every time a pod dies mid-job.

Readiness probes during shutdown: leave rotation before you stop answering

In the FastAPI and Express examples we mark the pod as "not ready" upon receiving SIGTERM. It is worth pausing on why that detail matters and how to configure the probes so the effect is immediate rather than late.

When a pod enters Terminating, Kubernetes removes it from the Service endpoints without consulting the readiness probe. So why fail it at all? Two reasons. First: not all traffic arrives through the endpoints. Load balancers that register pods directly as targets — like the AWS Load Balancer Controller in IP mode — run their own health checks against the pod, and those checks do depend on your readiness endpoint. Second: failing the probe is the one universal signal every layer understands, from the kubelet to an external healthcheck. It is cheap and it eliminates a whole class of races.

```
readinessProbe:
  httpGet:
    path: /healthz/ready
    port: 8080
  periodSeconds: 2
  failureThreshold: 2
livenessProbe:
  httpGet:
    path: /healthz/live
    port: 8080
  periodSeconds: 10
  failureThreshold: 3
```

The arithmetic matters: with periodSeconds: 2 and failureThreshold: 2, anything relying on that probe takes at most about 4 seconds to stop sending you traffic from the moment you start returning 503. That time has to fit inside the preStop sleep: if your sleep is 8 seconds and the probe needs 4 to propagate, you are cutting it close.

With the defaults (periodSeconds: 10, failureThreshold: 3) you would need up to 30 seconds — more than the entire default grace period.

A subtle and frequent mistake: using the same endpoint for liveness and readiness. If your liveness probe starts failing during the drain — because you return 503 on the shared endpoint — the kubelet may decide to restart the container in the middle of the shutdown, the exact opposite of what you want. Always separate them: liveness answers "is the process alive?" and must not change during shutdown; readiness answers "do I want traffic right now?" and is the one that goes dark first. In the earlier examples, /healthz/live can return 200 unconditionally as long as the process exists.

PodDisruptionBudgets: surviving drains, upgrades, and spot nodes

Everything so far protects a single termination. But in production pods rarely die one at a time: a node upgrade drains entire machines, the cluster autoscaler consolidates workloads, and spot nodes vanish with two minutes of warning (on AWS) or thirty seconds (in some GCP zones). If all your replicas live on the same node and that node gets drained, your perfect graceful shutdown will not prevent a full service outage.

The PodDisruptionBudget (PDB) is the contract that tells Kubernetes how many replicas you can afford to lose at once to voluntary disruptions (drains, autoscaler evictions, upgrades):

```
apiVersion: policy/v1
kind: PodDisruptionBudget
metadata:
  name: api-pdb
spec:
  minAvailable: 2
  unhealthyPodEvictionPolicy: AlwaysAllow
  selector:
    matchLabels:
      app: api
```

With minAvailable: 2 and three replicas, a kubectl drain can only evict one pod at a time and will wait for the replacement to be Ready before touching the next. The unhealthyPodEvictionPolicy: AlwaysAllow field (stable since Kubernetes 1.31) avoids the classic deadlock where a pod that never became Ready blocks a node drain: unhealthy pods can always be evicted, because protecting them protects no real availability.

Two important nuances. First: the PDB does not apply to involuntary disruptions — a kernel panic or the physical loss of a node does not ask permission; for that you spread replicas with topologySpreadConstraints across nodes and zones. Second: the PDB also does not apply to Deployment rolling updates; those are governed by maxSurge and maxUnavailable in the deployment strategy. The usual conservative configuration is maxSurge: 1, maxUnavailable: 0: the new pod is created first, and only when it is Ready does the old one get terminated. Combined with the PDB, you have covered both paths by which Kubernetes kills pods on purpose.

For spot nodes, the interruption notice arrives through the provider plane (the instance metadata service on AWS). Tools like the AWS Node Termination Handler or Karpenter interruption handling turn that notice into an orderly node drain: cordon, eviction respecting PDBs and, in each pod, the same termination sequence you already implemented. If your shutdown fits within the two-minute warning, a disappearing spot node is a non-event.

Draining beyond the cluster: ALB, NLB, ingress, and the seconds you do not control

The Kubernetes termination sequence ends at the edge of the cluster. If a managed load balancer sits in front, there is another state machine that also has to learn your pod is leaving — and its defaults rarely match yours.

With the AWS Load Balancer Controller in IP mode, each pod is a target in the target group. When the pod enters Terminating, the controller starts deregistration, and the ALB stops sending new requests but keeps existing connections alive for deregistration_delay, which defaults to 300 seconds. Five minutes. If your grace period is 60 seconds, the ALB may still consider the target "draining" long after the pod has died. Tune it to something consistent with your actual drain:

```
apiVersion: networking.k8s.io/v1
kind: Ingress
metadata:
  annotations:
    alb.ingress.kubernetes.io/target-group-attributes: >
      deregistration_delay.timeout_seconds=30
```

The complementary move is pod readiness gates: by labeling the namespace with elbv2.k8s.aws/pod-readiness-gate-inject: enabled, the controller injects an extra condition into each pod that is not satisfied until the target is registered and healthy in the ALB. Without it, during a rolling update Kubernetes can consider the new pod Ready — and kill the old one — before the ALB is capable of sending it traffic: a window of 502s that no preStop fixes, because the problem is on the registration side, not the removal side.

If you use ingress-nginx, the drain equivalent lives in worker_shutdown_timeout (240 seconds by default): how long NGINX workers wait for active connections to finish during a reload or shutdown. And remember that the ingress controller itself is just another pod, with its own termination sequence: recent versions already ship the delay and drain built in, but if you have customized its probes or grace period, check you have not broken it.

On GKE with container-native load balancing, the equivalent knob is the BackendConfig: connectionDraining.drainingTimeoutSec plays the role of the deregistration delay. One last detail that affects LoadBalancer-type Services: externalTrafficPolicy. With Cluster (the default), any node forwards traffic to any pod and removals propagate via kube-proxy on every node; with Local only nodes with local pods receive traffic, which preserves the source IP and saves a hop, but couples node health to your pods: when the last pods drain off a node, the load balancer health check toward that node also has to fail in time.

Go: graceful shutdown with http.Server.Shutdown

The series covers Python and Node in the main examples, but many platform services are written in Go, and its standard library has shipped draining built in since Go 1.8: http.Server.Shutdown stops accepting new connections, waits for in-flight requests to finish, and closes idle keep-alive connections — exactly the semantics you have to assemble by hand in Node.

```
package main

import (
        "context"
        "log"
        "net/http"
        "os/signal"
        "syscall"
        "time"
)

func main() {
        sigCtx, stop := signal.NotifyContext(
                context.Background(), syscall.SIGTERM, syscall.SIGINT)
        defer stop()

        srv := &http.Server{Addr: ":8080", Handler: routes()}

        go func() {
                if err := srv.ListenAndServe(); err != http.ErrServerClosed {
                        log.Fatalf("server: %v", err)
                }
        }()

        <-sigCtx.Done() // SIGTERM received
        markNotReady()  // readiness starts returning 503

        // cap the drain: it must fit in the grace period
        ctx, cancel := context.WithTimeout(
                context.Background(), 25*time.Second)
        defer cancel()

        if err := srv.Shutdown(ctx); err != nil {
                log.Printf("incomplete drain: %v", err)
        }
        closePools() // DB pools, clients, metrics flush
}
```

Three details make the difference. First, signal.NotifyContext turns the signal into a cancelled context: idiomatic and composable with the rest of your code. Second, the timeout context passed to Shutdown is your drain cap — the equivalent of Uvicorn's --timeout-graceful-shutdown — and it must fit in the grace period after subtracting the preStop. Third, Shutdown does not wait for hijacked connections (WebSockets via Hijack): for those, register your own cleanup with srv.RegisterOnShutdown and close them yourself, as we see in the next section.

Connections that never end on their own: gRPC, WebSockets, and SSE

The entire drain model above assumes requests finish by themselves if you wait long enough. Long-lived connections break that premise: a gRPC stream, a WebSocket, or an SSE connection can last for hours. For them, draining is not waiting: it is announcing, cutting cleanly, and trusting the client to reconnect.

In gRPC the native mechanism is the HTTP/2 GOAWAY frame. GracefulStop() in grpc-go stops accepting new streams, sends GOAWAY, and waits for in-flight RPCs to finish; since it can wait forever, you combine it with a timer that forces Stop() if the drain does not finish in time:

```
done := make(chan struct{})
go func() {
        grpcServer.GracefulStop() // GOAWAY + wait for in-flight RPCs
        close(done)
}()
select {
case <-done:
case <-time.After(20 * time.Second):
        grpcServer.Stop() // cut whatever remains
}
```

For never-ending streams, the defense is preventive: keepalive.ServerParameters with MaxConnectionAge and MaxConnectionAgeGrace recycles connections every few minutes, so no client has spent hours glued to the same pod when its time comes. It is the same idea as rotating connections in a pool: it limits the blast radius of any termination.

With WebSockets, the correct shutdown is to send a close frame with code 1001 ("going away") to each connection, give clients a couple of seconds to process it, and close. The real work lives in the client: reconnection logic with backoff and jitter — the retries guide in this series applies literally — plus re-subscribing to whatever state it held. A WebSocket client without automatic reconnection turns every deploy into a broken session.

SSE is the friendly case: EventSource reconnects on its own. Your server decides the delay with the retry: field and, if you number events with id:, the browser sends Last-Event-ID when reconnecting, letting you resume the stream without losing messages. During shutdown it is enough to close SSE connections without drama after marking the pod not ready: clients will come back — to the new pod. The typical mistake is a load balancer whose idle timeout is shorter than your SSE heartbeat, which turns reconnections into a permanent routine instead of a deploy-time exception.

Sidecars: when the proxy dies before your application

For years, the most irritating graceful-shutdown problem in service meshes was the order of death: Kubernetes sent SIGTERM to all containers in the pod at once, the network sidecar (istio-proxy, linkerd-proxy) died within seconds, and your application — still draining requests with the calm you gave it — lost its network. Outbound calls in those final seconds (acking a message, writing to the database, notifying another service) failed during every single deploy.

Native sidecar containers solve this at the root. Since Kubernetes 1.28 (stable in 1.33), a sidecar is declared as an initContainer with restartPolicy: Always:

```
spec:
  initContainers:
  - name: proxy
    image: my-proxy:1.2
    restartPolicy: Always   # this is what makes it a sidecar
  containers:
  - name: api
    image: my-api:2.4.0
```

The semantics are exactly what a proxy needs: it starts before the main containers, and on termination the kubelet first waits for the main containers to finish and only then shuts down the sidecars, in reverse start order. Your application keeps its network for the entire drain. As a bonus, Jobs no longer hang in Running because the sidecar "did not know" it should die: when the main container finishes, the sidecar shuts down on its own and the Job completes.

If you cannot use native sidecars yet, the meshes ship mitigations. In Istio, TERMINATION_DRAIN_DURATION_SECONDS (5 seconds by default) extends how long the proxy waits before closing, and the EXIT_ON_ZERO_ACTIVE_CONNECTIONS option makes Envoy wait until it has no active connections before exiting. In Linkerd, Jobs can tell the proxy explicitly with a POST to localhost:4191/shutdown when their work is done. These are honest but fragile solutions compared to the kubelet-guaranteed termination order: if your cluster is on 1.33 or later, migrate to native sidecars and delete the workarounds.

Kafka, Celery, and SQS: graceful shutdown in real consumers

The generic worker pattern — stop consuming, finish what you have, acknowledge, and exit — takes a different concrete shape in each queue system. The three most used ones have traps of their own.

In Kafka, the critical part is leaving the consumer group in an orderly way. A consumer that dies by SIGKILL gives no notice: the broker takes session.timeout.ms to declare it dead and, meanwhile, its partitions are orphaned. Calling consumer.close() during shutdown commits pending offsets and triggers the rebalance immediately:

```
from confluent_kafka import Consumer
import signal

running = True
signal.signal(signal.SIGTERM, lambda s, f: globals().update(running=False))

consumer = Consumer({
    "bootstrap.servers": "kafka:9092",
    "group.id": "orders",
    "enable.auto.commit": False,
    "partition.assignment.strategy": "cooperative-sticky",
})
consumer.subscribe(["orders"])

try:
    while running:
        msg = consumer.poll(timeout=1.0)
        if msg is None or msg.error():
            continue
        process(msg)              # finish the current message
        consumer.commit(msg)      # commit ONLY after processing
finally:
    consumer.close()  # commits, leaves the group, rebalances now
```

The partition.assignment.strategy: cooperative-sticky detail deserves a mention: with the classic (eager) strategy, every consumer departure stops the whole group while all partitions are reassigned; with cooperative rebalancing only the partitions of the departing member move. In a rolling deploy of twenty workers, the difference is

between twenty global micro-pauses and none. Also watch max.poll.interval.ms: if your drain takes longer than that interval (5 minutes by default), the broker kicks you out of the group before you finish.

In Celery, SIGTERM already initiates a warm shutdown: the worker stops accepting new tasks and finishes the ones in progress. What almost nobody configures is task_acks_late = True: without it, Celery acknowledges the message upon receipt, and a task killed halfway is simply lost. With acks_late, the interrupted task returns to the queue — which, again, requires idempotent tasks. And size the grace period by your slowest task, not your slowest HTTP request: they are very different budgets.

With SQS there is no explicit ack: a received message becomes invisible for the visibility timeout (30 seconds by default) and reappears if nobody deletes it in time. Graceful shutdown means: stop requesting new messages, finish and delete the ones you hold, and exit. If you get killed halfway, the message reappears on its own: the system is self-correcting as long as the visibility timeout exceeds your maximum processing time and your consumers are idempotent. The anti-pattern is the reverse: a 30-second visibility timeout with 2-minute jobs generates constant duplicate processing, deploys or not.

How to test graceful shutdown before production does it for you

Graceful shutdown is the classic code path that never runs in tests and always runs in production. It deserves explicit testing at three levels.

Level one, local and in seconds: docker stop (which sends SIGTERM, waits 10 seconds, and finishes with SIGKILL) against your container while you hold a slow request open with curl. If the request dies with a cut connection, your handler does not work or the signal never arrives (check PID 1). This test catches 80% of problems without touching a cluster.

Level two, automated in CI. An integration test that starts the real service as a subprocess, fires a slow request, sends SIGTERM mid-flight, and checks two things: the request completes with 200 and the process exits with code 0 within the deadline:

```
import signal, subprocess, time, threading
import httpx

def test_sigterm_drains_inflight_requests():
    proc = subprocess.Popen(["uvicorn", "app:app", "--port", "8123",
                             "--timeout-graceful-shutdown", "10"])
    time.sleep(1.5)  # wait for startup

    result = {}
    def slow_request():
        # the endpoint takes ~3 s to respond
        r = httpx.get("http://localhost:8123/slow", timeout=15)
        result["code"] = r.status_code

    t = threading.Thread(target=slow_request)
    t.start()
    time.sleep(0.5)              # the request is now in flight
    proc.send_signal(signal.SIGTERM)

    t.join(timeout=15)
    assert result.get("code") == 200      # completed, not cut off
    assert proc.wait(timeout=15) == 0     # clean exit, no SIGKILL
```

Level three, the dress rehearsal: a load test during a real deploy. Run k6 or vegeta against staging with sustained traffic and, halfway through, execute kubectl rollout restart deployment/api. The success criterion is binary: zero non-2xx responses across the whole window. If 502s appear, the timing tells you which layer to inspect: at the start of termination it is endpoint propagation (raise the preStop sleep or check the readiness gates); at the end it is an exhausted budget (grace period versus drain). Automate this scenario and make it the quality gate for infrastructure changes: it is the only test that validates every layer together.

Shutdown observability: proving it works

Without metrics, graceful shutdown degrades silently: someone shortens the grace period, a new sidecar changes the order of death, and nobody notices until the next incident review. Three instruments are enough to keep watch.

First, an in-flight requests gauge and a structured log per shutdown phase, with timestamps:

```
from prometheus_client import Gauge
import structlog, time

log = structlog.get_logger()
inflight = Gauge("http_inflight_requests",
                 "Requests currently being processed")

async def shutdown_sequence(app):
    t0 = time.monotonic()
    log.info("shutdown.start", inflight=inflight._value.get())
    mark_not_ready()
    log.info("shutdown.not_ready", t=time.monotonic() - t0)
    await drain_requests()
    log.info("shutdown.drained", t=time.monotonic() - t0)
    await close_pools()
    log.info("shutdown.cleanup_done", t=time.monotonic() - t0)
```

With that timeline in the logs, diagnosing a slow shutdown stops being archaeology: you see exactly which phase consumes the budget. If shutdown.drained takes 24 of your 25 seconds, your problem is slow requests, not cleanup.

Second, watch for SIGKILLs. A container that ends with exit code 137 (128 + 9) was finished off: either the grace period fell short or your drain hung. An alert on restarts with that exit code in your web services is equivalent to an alert saying "graceful shutdown is broken". Most observability agents (kube-state-metrics included) expose the last container termination reason, so the query is trivial.

Third, the panel of truth: 5xx response rate measured at the load balancer — not at the pod, which never sees the requests that failed to reach it — overlaid with deploy events. The signature of a drain problem is unmistakable: 502/503 spikes shorter than a minute, perfectly aligned with each rollout. That panel, checked once a week, is the difference between catching the regression immediately and living with it for months.

Connection pools and transactions: the exact order of cleanup

"Close resources after draining" is easy to say; the nuance is what closing a pool with work in progress actually means. A PostgreSQL pool like asyncpg or psycopg_pool has three possible states when the close order arrives: idle connections (closed immediately), borrowed connections running a query (you wait for them), and connections with an open transaction (the delicate case).

The general rule: pool shutdown should mirror the HTTP drain. asyncpg with pool.close() waits for borrowed connections to come back — the equivalent of Node's server.close() — so if you already drained requests, the pool closes in milliseconds. The anti-pattern is closing with a zero timeout or terminate(): it cuts connections with open transactions, and PostgreSQL will roll all of them back. You do not lose consistency (that is what transactions are for), but you do lose the work, and the user who got a 200 with their write half-committed now has a problem no retry will fix.

If you run PgBouncer or another intermediate pooler, remember it is also a service with its own shutdown: the PAUSE command waits for active transactions to finish and stops handing out new ones, and SHUTDOWN WAIT_FOR_SERVERS drains before exiting. A PgBouncer killed without draining cuts every session of every application at once: its graceful shutdown matters more than any individual service's.

For writes that must survive any interruption, the answer is not in the shutdown but in the design: transactions that group the full operation and, for side effects (publishing events, calling other services), the transactional outbox pattern we cover in this same series. Graceful shutdown reduces how often operations get cut mid-flight; the outbox makes those cuts stop mattering.

StatefulSets and stateful processes: hand over before you die

Everything so far assumes stateless services, where any replica can serve any request. Stateful processes add a step to the shutdown: hand over responsibility before exiting.

In a StatefulSet, termination during a rolling update is ordered and reversed (highest ordinal to lowest), and each pod keeps its identity and volume. The critical pattern is the leader's: if the dying pod is the leader of a consensus group or the primary of a database, passively waiting for SIGTERM is a bad strategy, because automatic failover takes time (seconds of detection plus an election). Well-designed systems hand over leadership proactively in the preStop or the SIGTERM handler: Kafka's controlled shutdown moves partition leadership before exiting; Patroni offers an explicit switchover for the PostgreSQL primary; etcd transfers raft leadership. The goal is to turn a failover (with an unavailability window) into a handover (without one).

For long-running jobs with in-memory state — model training, multi-hour batch processes — graceful shutdown means checkpointing: on the signal, persist progress to durable storage and exit; the replacement resumes from the last checkpoint. Checkpoint frequency is dictated by the budget: with spot nodes and a two-minute warning, a checkpoint that takes five minutes to write is a checkpoint that will never complete. Size the interval so that losing the work since the last saved point is acceptable, because sooner or later you will lose it.

Autoscaling: when shutdown stops being an event and becomes routine

With an active HPA (Horizontal Pod Autoscaler), pods do not die only on deploys: they die every time load drops. A service scaling between 5 and 50 replicas over the day executes dozens of terminations daily without anyone deploying anything. That changes the math: a shutdown bug that shows up once a week in a static service shows up fifty times a day with autoscaling.

Two settings soften the churn. On the HPA itself, the behavior.scaleDown block with stabilizationWindowSeconds (300 by default) and rate policies keeps a momentary traffic dip from triggering a wave of terminations you will have to undo minutes later. And on the pods, a high startup cost (caches to warm, models to load) is an argument for scaling down slowly: killing is cheap, recreating is not.

With KEDA and scale-to-zero, shutdown is literally constant: the last replica dies every time the queue empties. There the worker pattern gains another dimension: if scale-down arrives with a message half-processed, the whole model of late acks and idempotency is the only thing preventing lost or duplicated work. KEDA also respects the normal termination cycle, so your SIGTERM handler is the same; what changes is how often it gets exercised. In short: autoscaling turns graceful shutdown from a deploy concern into a permanent operational property, and the metrics from the observability section go from "deploy panel" to "everyday panel".

When graceful shutdown is not enough: design for the crash

An honest warning to close the loop: graceful shutdown is a probability optimization, not a guarantee. Some terminations allow no cleanup. An OOMKill is an immediate SIGKILL, with no prior SIGTERM: if your service dies from memory, no handler runs (and you will see it as exit code 137 with reason OOMKilled, distinguishable from the SIGKILL of an exhausted grace period). A kernel panic, node loss, or hardware failure gives no warning either. And a bug that hangs your shutdown handler turns every gentle termination into a delayed brutal one.

The engineering conclusion is that graceful shutdown reduces the frequency of abrupt endings, but system correctness cannot depend on it. The pieces that guarantee correctness are the usual ones: transactions for local atomicity, idempotency for retries, late acks so messages are not lost, the outbox for side effects, and checkpoints for long work. If your system is only correct when shutdowns are graceful, it is not correct.

This mindset, known as crash-only software, suggests a revealing test: kill your services with SIGKILL in staging from time to time and verify nothing is lost or duplicated. If the result is acceptable, graceful shutdown is what it should be — an experience improvement that removes the visible errors of the common case — not a crutch hiding a fragile design. The two reinforce each other: crash-safety gives you correctness; graceful shutdown gives you invisible deploys.

Debugging: the pod that will not die

The reverse problem also exists: pods stuck in Terminating for minutes or hours. The diagnostic toolbox is short and goes in order of frequency.

First, is the process ignoring the signal? kubectl describe pod tells you how long it has been Terminating; if it exceeds the grace period plus a few seconds, something odd is happening at the kubelet or finalizer level. If it is still within budget, get into the container and look: ps aux to confirm who PID 1 is, and if it is a shell, you have found the usual culprit.

Second, finalizers. A pod (or any object) with pending finalizers does not disappear until the responsible controller removes them. kubectl get pod -o yaml and look for metadata.finalizers: if there are entries from an operator or a mesh that no longer runs, the pod will stay Terminating forever. The clean fix is repairing the controller; hand-editing finalizers is the last resort, because skipping the cleanup they represented can leave orphaned resources (attached volumes, IPs, external registrations).

Third, zombie processes. If your container spawns subprocesses and PID 1 does not reap them (never calls wait), dead children pile up as zombies and some runtimes wait indefinitely. tini, or Docker Compose's init: true, exist exactly for this; in Kubernetes, shareProcessNamespace: true additionally lets one container in the pod act as init for the rest. And the true last resort: kubectl delete pod --force --grace-period=0 does not kill the process — it only deletes the API object and leaves the kubelet to clean up in the background — so the process may keep running on the node a while longer. With StatefulSets, that "a while longer" can mean two pods with the same identity: use it only when the node is already lost.

Other runtimes: Spring Boot and .NET in two minutes

The pattern is identical across ecosystems; what changes is where the switch lives. In Spring Boot, draining ships out of the box since version 2.3 and is enabled with one line of configuration:

```
# application.properties
server.shutdown=graceful
spring.lifecycle.timeout-per-shutdown-phase=25s
```

With that, SIGTERM makes Tomcat (or Netty in reactive applications) stop accepting new connections and wait for active requests up to the configured timeout. Beans are destroyed after the drain, in reverse creation order, so connection pools (HikariCP) close once no requests are using them: the correct order falls out of Spring's container lifecycle for free.

In .NET, the equivalent is IHostApplicationLifetime: ApplicationStopping fires on the signal (that is where you mark readiness as not ready and stop accepting work), the host waits for the servers to drain, and ApplicationStopped arrives at the end for cleanup. The cap is tuned with HostOptions.ShutdownTimeout, whose default — 30 seconds in recent versions — must fit, once again, inside the Kubernetes grace period after subtracting the preStop. Whatever the runtime, the checklist does not change: a signal that arrives, readiness that drops, a capped drain, cleanup at the end, and a budget that covers it all.

Jobs and CronJobs: finishing well is also shutting down well

Batch workloads have their own version of every problem in this guide. A Job pod can be evicted by a drain or a spot interruption like any other, and it receives the same SIGTERM: a batch process that checkpoints and exits with a non-zero code will be retried according to backoffLimit, so make sure an interrupted run and its retry do not double-apply work — idempotency again. If the job must never exceed a duration, activeDeadlineSeconds terminates it (SIGTERM included) when the budget runs out; and suspend: true lets you pause a Job cleanly before a planned maintenance instead of letting the drain interrupt it at a random point.

CronJobs add the concurrency question: if the previous run has not finished when the next one is due, concurrencyPolicy decides. Forbid skips the new run, Replace terminates the old one — with the exact SIGTERM sequence from this guide — and starts fresh. Replace is, in effect, a scheduled graceful shutdown: if your batch handles the signal well, Replace is safe; if it does not, every overlapping schedule is a small outage. And remember the sidecar note from earlier: with native sidecars, batch pods actually reach Succeeded instead of hanging forever because a proxy refused to exit.

Startup is part of shutdown: probes, minReadySeconds, and honest readiness

It sounds paradoxical, but half of a zero-error deploy happens at startup: the old pod can only leave without damage if the new one is truly ready. Kubernetes considers a pod Ready when its readiness probe passes; the trap is a probe that passes too early — the port is open but the cache is cold, the model is not loaded, the connection pool is empty. The result is a deploy with no 5xx but with a latency spike: technically alive, practically not ready.

Three tools make readiness honest. The startupProbe covers slow starters: until it succeeds, liveness and readiness are not evaluated, so a service that takes 90 seconds to warm up does not need a grotesquely large initialDelaySeconds on its other probes. The readiness probe itself should verify what serving actually requires: dependencies reachable, caches warm enough, pool initialized — not just "the process responds". And minReadySeconds (in the Deployment spec) adds a final safety margin: the pod must stay Ready for that many seconds before the rollout considers it available and moves on to killing the next old pod. It is cheap insurance against pods that pass the probe once and immediately crash-loop: without it, a fast rollout can replace every healthy replica with broken ones before the first alert fires. A well-tuned startup makes every shutdown on the other side of the rollout boring — which is the point of this entire guide.

Case study: the timeline of an error-free deploy

To tie all the pieces together, here is the minute and a half in the life of a well-configured rollout on a real service: FastAPI behind an ALB in IP mode, three replicas, an 8-second preStop, a 60-second grace period, a 30-second deregistration delay, and readiness gates enabled.

- T+0 — kubectl rollout restart. With maxSurge: 1 and maxUnavailable: 0, Kubernetes creates a new pod first. Nothing has been killed yet.
- T+12 — The new pod passes its readiness probe, the controller registers it in the target group and waits for the ALB to declare it healthy. Only then is the readiness gate satisfied and the pod counts as Ready.
- T+15 — Termination of the old pod begins: it enters Terminating, leaves the endpoints, and the ALB starts deregistration (no new requests are sent to it). In parallel the preStop starts: 8 seconds of sleep during which the pod keeps serving the stragglers that still arrive, completely normally.
- T+23 — SIGTERM. The handler sets shutting_down = True; the readiness probe starts returning 503. Uvicorn stops accepting new connections and drains the in-flight ones. The slowest takes 14 seconds.
- T+38 — Drain complete. The PostgreSQL pool and the HTTP client close, metrics and logs are flushed: 3 seconds.
- T+41 — exit 0. The kubelet sees the clean exit, well under the 60-second budget. The pod disappears. The rollout repeats the cycle with the next replica.

This team's before-and-after is representative: before touching anything, each deploy generated between 30 and 60 502 responses (measured at the ALB) plus a noisy error-spike alert; the post-mortem revealed a shell-form CMD, no preStop, and the deregistration delay at its 300-second default. After applying exactly what this guide describes, the next fifty deploys added up to zero errors attributable to termination. No business code changes: just shutdown done right.

Production checklist

The short list to review for each service before calling it done:

- The process actually receives SIGTERM: exec-form CMD or tini as init; verified with ps aux inside the container.
- preStop with a 5–10 second sleep, and terminationGracePeriodSeconds increased to cover it with margin.

- Grace period computed piece by piece: preStop + slowest request/message + cleanup + margin. Never the default 30 with a preStop added.
- Readiness separate from liveness; readiness fails immediately after SIGTERM with aggressive periodSeconds and failureThreshold.
- Drain with an explicit cap (timeout-graceful-shutdown, server.close + timer, srv.Shutdown with context) that fits the budget.
- Resources (pools, clients, flushes) closed after draining, never before.
- Keep-alive connections closed explicitly; long-lived connections (gRPC, WebSocket) with notice, a max age, and clients that reconnect.
- Load balancer tuned: deregistration delay consistent with the grace period, and readiness gates enabled if you register pods as targets.
- PDB with unhealthyPodEvictionPolicy: AlwaysAllow, topologySpreadConstraints, and conservative maxSurge/maxUnavailable.
- Workers: stop consuming on the signal, finish and acknowledge late (acks_late, commit after processing, properly sized visibility timeout); idempotent consumers.
- Native sidecars (initContainer with restartPolicy: Always) on 1.33+ clusters, or proxy drain configured in the mesh.
- Tested: a SIGTERM test in CI and a load test during a rollout with zero errors as the pass criterion.
- Observed: in-flight gauge, per-phase shutdown logs, an alert on exit code 137, and a load-balancer 5xx panel aligned with deploys.

Quick reference: which Kubernetes version ships each piece

To know what you can use depending on your cluster:

- 1.28 — Native sidecar containers (initContainers with restartPolicy: Always), in alpha.
- 1.29 — Native sidecars in beta and enabled by default.
- 1.30 — Native sleep action in preStop (beta): goodbye to shells in distroless images.
- 1.31 — unhealthyPodEvictionPolicy: AlwaysAllow in PDBs, stable.
- 1.32 — Sleep action stable; zero-second sleep allowed as a configurable no-op.
- 1.33 — Native sidecars stable: the definitive fix for the proxy/application order of death.

If your cluster lags behind, each section of this guide points at the alternative (exec with sh -c "sleep", mesh workarounds, and so on).

Frequently asked questions

Do I need all of this for a small internal service? Not all of it. The dignified minimum for any HTTP service: exec-form CMD, a 5-second preStop, SIGTERM handling with a drain, and a reviewed grace period. That is twenty lines across Dockerfile, manifest, and application, and it removes most deploy-time 502s. The remaining layers come in when you add managed load balancers, queues, or long-lived connections.

Why do I still see 502s with everything configured? Measure where: if errors appear at the start of termination, endpoint removal did not propagate in time (raise the preStop sleep, check readiness gates and the deregistration delay); if they appear at the end, the budget ran out (drain longer than the remaining grace period). The load-balancer 5xx panel with deploy events overlaid answers this question in seconds.

SIGINT or SIGTERM? Kubernetes sends SIGTERM. Handle both with the same code — SIGINT gives you the same semantics locally with Ctrl+C — as the Go and Python examples in this guide do.

Does a long preStop slow down my deploys? Yes: each replica takes at least preStop + drain to die, and a twenty-replica rollout feels it. It is a deliberate trade of speed for zero errors. If it hurts, raise maxSurge to replace more pods in parallel instead of trimming the shutdown.

How do I test this without a cluster? docker stop is your pocket Kubernetes simulator: SIGTERM, a wait (tunable with -t), then SIGKILL. If your container survives docker stop with dignity while a slow request is in flight, 80% of the work is done.

Conclusion

Graceful shutdown is not an optimization: it is the difference between deploys nobody notices and deploys your users feel. The whole recipe fits in one sentence: delay SIGTERM with a preStop, stop advertising as ready, drain what is in flight, close resources last, and make sure the signal actually reaches your process. Implement it once per service and every deploy, scale-down, or node replacement will pass unnoticed — which is exactly the goal.