OpsCortex

Kubernetes Reliability Playbook

Trupti K.

OPSCORTEX

Kubernetes Reliability Playbook

Production reliability, observability, and incident response, explained the way systems actually work.

Trupti Kolekar · Sr. Site Reliability Engineer

FIRST EDITION · 2026 · PUBLISHED BY OPSCORTEX

Contents

THE PLAYBOOK

APPENDICES

Preface

If you have run Kubernetes in production, you already know the cluster does not care about your weekend. Pods get evicted at 3 a.m. A node goes `NotReady` mid-deploy. A readiness probe you copy-pasted two years ago starts flapping under load. This playbook is for the engineer who has lived some of that: comfortable with `kubectl get pods`, but ready to stop guessing and start reasoning.

My promise is narrow and deliberate. You will leave understanding reliability, not memorizing tools. Commands age. Flags get deprecated. The mental models are what last: why a probe exists, what a controller is actually doing, how disruption budgets interact with node drains. I would rather you understand *why* `OOMKilled` happens than hand you twenty commands to react to it.

Here is the one idea that everything in this book hangs off. **Kubernetes is a control loop.** You declare a desired state. Controllers observe the actual state, compute the difference, and act to close the gap, continuously and forever. Deployments, the scheduler, the HPA, the kubelet, even certificate rotation all run the same loop. Once you see the cluster this way, reliability stops being a bag of tricks and becomes a single question you can always ask: *what is the desired state, what is the actual state, and which loop is responsible for closing the gap?*

Trupti Kolekar

GET STARTED

How to use this playbook

Read it front to back the first time. Each chapter assumes the mindset built by the one before it. After that, treat it as a reference you raid mid-incident.

- **Chapter 1: The Reliability Mindset.** The control-loop model, and how to think about failure before it happens.
- **Chapter 2: Designing Pods That Don't Fall Over.** Requests, limits, probes, and the pod-level choices that keep everything above them up.
- **Chapter 3: Self-Healing and Scaling.** How controllers, the HPA, and autoscaling close the gap, and where they quietly fail.
- **Chapter 4: Surviving Disruptions.** PodDisruptionBudgets, drains, and graceful termination when nodes move under you.
- **Chapter 5: Observability That Actually Helps.** Signals worth collecting, and how to instrument so data answers questions instead of adding noise.
- **Chapter 6: SLO-Based Alerting and On-Call.** Turning reliability targets into alerts that page you for the right reasons only.
- **Chapter 7: Incident Response and Debugging.** A repeatable method for the emergency page, from triage to root cause.
- **Chapter 8: Reliability Guardrails.** Policies, admission control, and defaults that make the safe path the default path.

CHAPTER 1

The Reliability Mindset

Reliability is not a vibe. It is a number you choose on purpose, measure continuously, and spend deliberately. Before you touch a single HPA or PodDisruptionBudget, you need a mental model for *what* "reliable enough" means and *how* Kubernetes actually keeps systems running. This chapter gives you both: the SLI/SLO/error-budget vocabulary that turns reliability into an engineering decision, and the reconciliation control loop that makes Kubernetes self-heal.

Reliability is a feature, not an afterthought

Every system has a reliability target whether or not you wrote it down. The difference between mature teams and the rest is that mature teams chose the number *before* the incident, not during the postmortem. "As reliable as possible" is not a target. It has no budget, no trade-off, and no way to tell engineering "ship the feature" versus "stop and fix." Reliability competes with feature velocity for the same engineering hours. If you don't quantify it, that competition gets resolved by whoever shouts loudest in the incident channel.

SLI vs SLO vs SLA: precise definitions and how they differ

These three are routinely confused. They are not the same thing.

- **SLI (Service Level Indicator):** a *measured* ratio of good events to total events. It is a number you compute from real telemetry, for example `successful requests / total requests`, or `requests faster than 200 ms / total requests`. An SLI is always between 0 and 1 (0 to 100%).
- **SLO (Service Level Objective):** a *target threshold* on an SLI over a defined window. Take the **checkout** service we will follow through this book: its SLO is "99.9% of requests succeed, measured over a rolling 30 days." The window matters as much as the percentage.
- **SLA (Service Level Agreement):** an *external contract* with a customer that carries consequences (refunds, credits, penalties) when violated.

The non-negotiable rule: **your SLO must be stricter than your SLA.** If you promise customers 99.5% (SLA), run your internal SLO at 99.9%. That gap is your safety margin. You want to be alerting and fixing long before you owe anyone money.

```
SLI  = good / total                  (a measured fact)
   SLO  = SLI ≥ 99.9% over 30 days      (a target you chose)
   SLA  = "≥ 99.5% or we credit you"    (a contract, looser than the
   SLO)
```

Error budgets: turning 'how reliable' into an engineering decision

The error budget is the single most useful idea in this chapter:

```
error budget = 100% − SLO
```

Checkout's 99.9% availability SLO permits 0.1% unavailability. Over a 30-day window that is:

```
30 days × 24 h × 60 min × 0.001 = 43.2 minutes of allowed downtime
```

That 43.2 minutes is a *budget you are allowed to spend*. A risky deploy, a chaos experiment, a region failover drill: all of it draws down the same account. The policy that makes this powerful: **while there is budget left, ship features; when the budget is exhausted, freeze risky changes and spend engineering on reliability instead.** The error budget converts an emotional argument ("is this safe?") into a balance check ("do we have budget?").

The nines table and what they actually cost

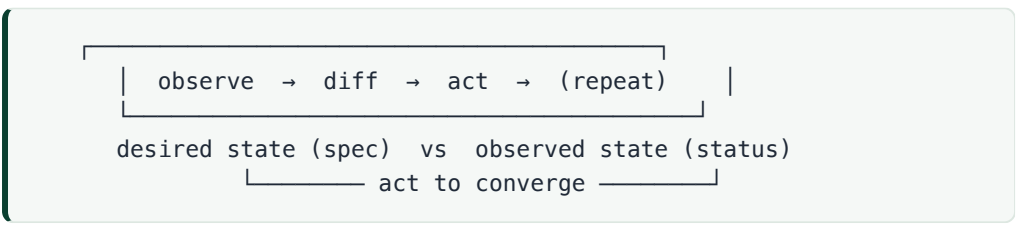
Availability	Downtime / month	Downtime / year
99.9% ("three nines")	~43.2 min	~8.77 h
99.99% ("four nines")	~4.32 min	~52.6 min
99.999% ("five nines")	~26 sec	~5.26 min

Read that table as a cost curve, not a goal ladder. Each extra nine cuts your allowed downtime by 10×, and it roughly multiplies the engineering investment (redundancy, multi-region, automated failover, on-call rigor)

needed to hit it. Five nines means your *entire* yearly downtime budget is shorter than a single node reboot. Chasing nines blindly burns money and velocity for reliability your users may never notice. The error budget exists precisely so you can *stop* climbing and spend that headroom on shipping.

The Kubernetes control loop: declarative desired-state reconciliation

Now the mechanism. Kubernetes is built on **controllers**, and every controller runs the same loop:



You declare *desired* state in an object's `spec`, and Kubernetes records *observed* state in its `status`. A controller continuously compares the two and takes action to close the gap. Critically, this loop is **level-triggered, not edge-triggered**: it reacts to the *current state of the world*, not to a one-time event. If a controller misses an event (it was restarting, the network blipped), it still converges on the next pass because it re-reads reality every cycle. Edge-triggered systems that miss the edge stay broken. Level-triggered systems self-correct.

```
apiVersion: apps/v1
   kind: Deployment
   metadata:
     name: checkout
   spec:
     replicas: 3          # desired state: "I want 3 Pods"
     selector:
       matchLabels: { app: checkout }
     template:
       metadata:
         labels: { app: checkout }
       spec:
         containers:
           - name: checkout
             image: checkout:1.4.2
```

The pieces that run this system start with the **control plane**: `kube-apiserver` (the front door and source of truth, backed by `etcd`), `kube-controller-manager` (which runs the built-in controllers), and `kube-scheduler` (which assigns Pods to nodes). On top of that sits the **kubelet** on every node. The division of labor is clean: the controller-manager reconciles higher-level objects (a Deployment manages a ReplicaSet, which manages Pod count), while the kubelet reconciles the actual Pods on its node against what the API server says should be running there.

Why reconciliation is the foundation of self-healing

Self-healing is not a special feature bolted on. It is an *emergent property* of the reconcile loop. Walk the failure:

```
Pod crashes  →  observed replicas = 2  →  desired = 3
              →  ReplicaSet controller sees diff
              →  creates 1 replacement Pod  →  observed = 3 again
```

Nobody paged anyone. The ReplicaSet controller didn't need to be *told* the Pod died. On its next pass it simply observed that reality (2 running) no longer matched intent (3 desired) and acted. The same reconcile pattern shows up all over: the ReplicaSet controller replaces Pods lost when a node dies, the Deployment controller rolls a stuck rollout forward, and the kubelet restarts a crashed container on its own node per the Pod's `restartPolicy` (that last one is a separate loop, and it keeps the same Pod without changing the replica count). Different controllers, different objects, one idea. Understand this pattern and most of Kubernetes' "magic" stops being magic. It is just `observe → diff → act`, forever.

✓ PRODUCTION CHECKLIST

- ☐ Write down an explicit SLO (percentage **and** window, for example 99.9% over 30 days) for every user-facing service.
- ☐ Define each SLI as a concrete ratio against real telemetry (`good / total`), not a vague "uptime" feeling.
- ☐ Set every internal SLO stricter than the corresponding customer SLA, with a deliberate safety margin.
- ☐ Compute the error budget (`100% − SLO`) in minutes for your window and track burn against it.
- ☐ Agree on an error-budget policy *in writing*: what happens (feature freeze) when the budget is exhausted.
- ☐ Pick a target nine based on user impact and cost, not prestige, and be able to justify why you are *not* going higher.
- ☐ For every workload, set `replicas` (and use a controller like Deployment) so reconciliation has a desired state to enforce.

⚠ **FAILURE MODES TO WATCH**

- **No defined SLO** → reliability arguments become opinion fights; *fix:* commit a number and window before the next incident.
- **SLA tighter than or equal to your SLO** → you breach the contract before your own alerts fire; *fix:* widen the gap so the SLO trips first.
- **Treating the error budget as "never spend it"** → you starve feature velocity for nines users don't notice; *fix:* deliberately spend budget on deploys and experiments while it lasts.
- **Bare Pods instead of a controller** (`kind: Pod` with no Deployment/ReplicaSet) → nothing reconciles desired count, so a dead Pod stays dead; *fix:* always run workloads under a controller.
- **Assuming events are guaranteed** → building logic that breaks if a watch event is missed; *fix:* rely on level-triggered reconciliation that re-reads current state each pass.

CHAPTER 2

Designing Pods That Don't Fall Over

Most Pod restarts in production are self-inflicted. A misconfigured probe, a missing request, a memory limit set too tight: none of these show up in a demo, but under real traffic they trigger restart loops, evictions, and silent latency. This chapter is about configuring the four knobs that decide whether a Pod survives: probes, requests, limits, and the QoS class they imply.

Liveness vs readiness vs startup: three different questions

These three probes answer different questions, and conflating them is the root of most probe pain.

- **Liveness:** "Is this container wedged and unrecoverable?" If a liveness probe fails `failureThreshold` times in a row, the kubelet **restarts the container**.
- **Readiness:** "Can this container serve traffic right now?" If a readiness probe fails, the Pod is **removed from all Service endpoints**, so it gets no traffic, but it is **not restarted**. When the probe passes again, it returns to rotation.
- **Startup:** "Has this container finished booting?" A startup probe **gates** liveness and readiness: while it is running, both are disabled. Liveness

and readiness only begin once the startup probe succeeds for the first time.

```
startup probe ──succeeds once──▶ liveness + readiness take over
      (boot)                          (steady state)
```

The mental model: readiness controls *traffic*, liveness controls *life*, and startup controls *when the other two wake up*.

The classic probe mistakes that cause cascading restarts

Know the defaults, because they are aggressive:

Field	Default
`periodSeconds`	10
`timeoutSeconds`	1
`failureThreshold`	3
`successThreshold`	1
`initialDelaySeconds`	0

Note that `successThreshold` **must be 1** for liveness and startup probes; only readiness may set it higher. And `timeoutSeconds: 1` is brutal: if your health endpoint occasionally takes 1.2s under load, the probe times out, and three of those in a row restart a perfectly healthy container.

The most damaging mistake is pointing a **liveness** probe at a dependency such as a database or downstream API. When that dependency hiccups, every replica fails liveness at once and the kubelet restarts them all together. That cascading restart turns a brief blip into an outage. Liveness should check only *this process's* health. Dependency health belongs in **readiness**, which removes the Pod from traffic without killing it.

The second classic mistake is using a large `initialDelaySeconds` to handle a slow starter. That is a fixed guess: too short and liveness kills the boot, too long and crashes take forever to recover. The correct fix is a **startup probe**:

```
startupProbe:
     httpGet:
       path: /healthz
       port: 8080
     periodSeconds: 10
     failureThreshold: 30   # 30 × 10s = up to 300s to start
   livenessProbe:
     httpGet:
       path: /healthz
       port: 8080
     periodSeconds: 10
     failureThreshold: 3
```

`failureThreshold: 30` × `periodSeconds: 10` allows up to **300 seconds** for startup. As soon as `/healthz` returns 200 once, the startup probe passes and liveness takes over with its tight 30s budget. Slow starters get patience; wedged ones still get killed fast.

Requests vs limits: scheduling vs enforcement

This is the distinction people get wrong most often:

- **Requests** are for the **scheduler**. A Pod is placed on a node only if that node has allocatable capacity left for the Pod's CPU and memory *requests*. Requests also determine the Pod's **QoS class**. Think of a request as a planning number.
- **Limits** are for the **kernel**. They are enforced at runtime through cgroups: CPU through the CFS quota, memory through the OOM killer. A limit is a hard ceiling.

A Pod can request 100m CPU and limit at 1 core: the scheduler reserves 100m, but the kernel lets it burst to a full core when the node has spare capacity. Set requests to what you *normally* need, and set limits to protect the node from a runaway.

QoS classes: Guaranteed, Burstable, BestEffort

The kubelet assigns each Pod a QoS class from its requests and limits. This class decides eviction order under node pressure.

- **Guaranteed:** every container sets `requests == limits` for **both** CPU and memory.
- **BestEffort: no** container sets any requests or limits.
- **Burstable:** anything in between (at least one request set, but not the Guaranteed pattern).

```
resources:        # Guaranteed: requests == limits, both resources
     requests:
       cpu: "500m"
       memory: "512Mi"
     limits:
       cpu: "500m"
       memory: "512Mi"
```

Under **node memory pressure**, the kubelet does not rank by QoS class directly. It looks first at whether a Pod's usage exceeds its **requests**, then at its **Priority**, then at how far usage runs over the request. QoS only *correlates* with the result: a BestEffort Pod sets no requests, so it always exceeds them and tends to go first, while Guaranteed Pods never do and go last. The practical rule survives the detail: a Pod with no requests is first out the door, so if a workload matters, make sure it isn't BestEffort.

OOMKilled: how memory limits kill containers

Memory is **incompressible**: you can't throttle it, you can only take it away. When a container tries to use more than its memory limit, the kernel's **OOM killer** terminates it with **SIGKILL**. The container shows `Reason: OOMKilled` and **exit code 137** (128 + signal 9).

```
kubectl get pod checkout-7d9f8b-4qx2p -o
   jsonpath='{.status.containerStatuses[0].lastState.terminated.reaso
   n}'
   # OOMKilled
```

There is no warning and no graceful shutdown, because SIGKILL can't be caught. The fix is almost never "remove the limit." It is to measure real usage and size the limit above the true peak, including GC headroom for JVM- and Go-style runtimes.

CPU throttling: why limits slow you down instead of killing you

CPU is **compressible**, so a CPU limit never kills your container. It **throttles** it instead. Enforcement uses the Linux **CFS quota** over a **100ms period**. A `500m` limit means 50ms of CPU time per 100ms period; once the container burns that 50ms, it is **paused until the next period begins**.

This is why a latency-sensitive service like **checkout** gets mysteriously slow under load with a low CPU limit: it is hitting the quota and stalling, not running out of CPU on the node. Check `container_cpu_cfs_throttled_periods_total` to confirm. A common production choice is to **omit the CPU limit entirely** while still setting a CPU *request*. The request guarantees scheduling and a fair share, and the absence of a limit means no artificial throttling when the node has spare cycles. Checkout is sized exactly this way:

```
resources:
     requests:
       cpu: "250m"        # scheduling + fair share, no throttle
   ceiling
       memory: "512Mi"
     limits:
       memory: "512Mi"    # memory limit == request: no OOM-from-
   bursting surprises
```

Setting the **memory limit equal to the memory request** removes bursting surprises. The Pod can never grow into memory that isn't reserved for it, so it won't get OOMKilled for borrowing memory the node later needs.

✓ PRODUCTION CHECKLIST

- ☐ Liveness probes check only the local process, never a database or downstream dependency.
- ☐ Dependency health lives in the **readiness** probe, so a blip drains traffic instead of restarting Pods.
- ☐ Slow starters use a **startup probe** (`failureThreshold` × `periodSeconds` ≥ real boot time), not a large `initialDelaySeconds`.
- ☐ Raise `timeoutSeconds` above the default 1s if your health endpoint can be slow under load.
- ☐ Every container sets a **memory request and limit**, with `limit == request` unless you have a measured reason to burst.
- ☐ Set a **CPU request** on every container, and consider omitting the CPU limit for latency-sensitive services.
- ☐ Make production-critical Pods at least **Burstable**, never BestEffort.
- ☐ Confirm QoS with `kubectl get pod <name> -o jsonpath='{.status.qosClass}'`.

⚠ FAILURE MODES TO WATCH

- **Liveness probe points at a shared dependency** → all replicas restart together during a blip → move the dependency check to readiness.
- `timeoutSeconds: 1` **on a slow endpoint** → healthy Pods restart-loop under load → raise `timeoutSeconds` and/or `failureThreshold`.
- **Memory limit set too tight** → `OOMKilled`, exit 137, no graceful shutdown → measure real peak usage and raise the limit above it.
- **Low CPU limit on a latency-sensitive service** → CFS throttling and mystery p99 spikes → raise or remove the CPU limit and watch `cfs_throttled_periods_total`.
- **No requests set (BestEffort)** → first Pod evicted under node memory pressure → add CPU and memory requests to make it Burstable or Guaranteed.

CHAPTER 3

Self-Healing and Scaling

Kubernetes "self-heals" not by magic but by *reconciliation*. Every controller runs a loop comparing **desired state** (your spec) to **observed state** (the cluster) and takes action to close the gap. Rollouts, autoscaling, and graceful shutdown are all this same loop applied to different objects. Understand the loop and the rest follows.

Deployments and ReplicaSets

A Deployment doesn't run Pods directly. It manages **ReplicaSets**, and each ReplicaSet manages Pods.

```
Deployment ──owns──▶ ReplicaSet (rev N) ──owns──▶ Pods
                └──────▶ ReplicaSet (rev N-1)  [scaled to 0]
```

When you change the Pod template (a new image, say), the Deployment controller creates a **new ReplicaSet** and scales it up while scaling the old one down. That staged hand-off *is* the rolling update. Old ReplicaSets are kept (scaled to 0), so `kubectl rollout undo` rolls back fast by scaling one of them straight back up, no rebuild required. That rollback is itself a rolling update (same `maxSurge`, `maxUnavailable`, and readiness gates), so it is *fast*, not an instant traffic flip; instant, atomic rollback is a property of blue-green, covered in Chapter 4.

Rollout strategy: maxSurge and maxUnavailable

`RollingUpdate` is governed by two knobs, both defaulting to **25%**:

- **maxSurge:** extra Pods allowed *above* desired count. Rounds **up**.
- **maxUnavailable:** Pods allowed *below* desired count. Rounds **down**.

They **cannot both be 0** (that would forbid all progress). For 10 replicas at defaults: up to 13 Pods may exist, and at least 8 stay available.

```
spec:
     strategy:
       type: RollingUpdate
       rollingUpdate:
         maxSurge: 1
         maxUnavailable: 0   # capacity-preserving: never drop below
   desired
```

`maxUnavailable: 0` plus a correct readiness probe gives a **capacity-preserving** rollout: a new Pod must go Ready before an old one leaves, so the serving fleet never dips below the desired count. Preserving capacity is not the same as zero dropped requests. Whether users feel a blip also depends on connection draining (a `preStop` hook and a SIGTERM handler, later this chapter) and on the load balancer noticing endpoint changes quickly. Capacity-preserving is the safest default for request-serving workloads; pair it with graceful shutdown to get close to no visible downtime.

Horizontal Pod Autoscaler (autoscaling/v2)

HPA adds or removes Pod *replicas*. Use the stable `autoscaling/v2` API, which supports `Resource`, `Pods` (custom), and `External` metrics. The core algorithm:

```
desiredReplicas = ceil( currentReplicas × currentMetric /
   targetMetric )
```

The controller syncs every **15s** (`--horizontal-pod-autoscaler-sync-period`) and applies a **0.1 (10%) tolerance**: it won't act while the metric ratio sits within 10% of target, which prevents thrashing. Scale-**up** is immediate (stabilization window 0s). Scale-**down** waits a **300s** stabilization window by default, so a brief dip doesn't strand you under-provisioned.

Resource utilization targets are measured against the Pod's CPU or memory **request**, so requests **must** be set, or `averageUtilization` has no denominator and HPA reports `<unknown>`. Here is **checkout**'s HPA: it scales on CPU and never drops below 3 replicas, so a lost zone still leaves it serving.

```
apiVersion: autoscaling/v2
   kind: HorizontalPodAutoscaler
   metadata: { name: checkout }
   spec:
     scaleTargetRef:
       apiVersion: apps/v1
       kind: Deployment
       name: checkout
     minReplicas: 3
     maxReplicas: 20
     metrics:
     - type: Resource
       resource:
         name: cpu
         target:
           type: Utilization
           averageUtilization: 70
```

HPA will not scale below **1** unless the alpha `HPAScaleToZero` feature gate is enabled.

VPA and why it conflicts with HPA

The **Vertical Pod Autoscaler** recommends (and in `Auto` or `Recreate` mode sets) CPU and memory **requests**. To apply them it historically **evicts** Pods so they reschedule with new requests. In-place Pod resize (`resizePolicy`, maturing through v1.27+) is the newer, eviction-free path.

The trap: **do not point VPA and HPA at the same CPU or memory signal.** If HPA scales out on CPU utilization while VPA rewrites the per-Pod CPU request, HPA reads higher utilization against a smaller request

and scales out again, a feedback loop with no brakes. VPA tunes **CPU and memory requests** (and by default, `controlledValues: RequestsAndLimits`, the matching limits too, scaled to preserve your request-to-limit ratio; set `controlledValues: RequestsOnly` to leave limits alone), so the safe split is to divide those two signals: let **HPA scale on CPU** (or on a custom or external metric such as requests-per-second or queue depth) and let **VPA manage memory only**. When you cannot separate them cleanly, run VPA in recommendation-only mode and apply its numbers yourself.

Node autoscaling: Cluster Autoscaler vs Karpenter

Pods stay `Pending` if no node fits them. Two tools add nodes:

- **Cluster Autoscaler** scales **predefined node groups** (AWS ASGs, GCP MIGs). It picks a group and bumps its size, so you are constrained to the instance types those groups define.
- **Karpenter** is **groupless**. It watches unschedulable Pods and provisions **right-sized nodes directly** from the cloud provider's API, choosing instance types to fit the pending workload. It is typically **faster** than node-group scaling and bin-packs better.

Graceful shutdown: SIGTERM, preStop, and the grace period

When a Pod is deleted, two things happen **concurrently**: it is removed from Service endpoints, and the kubelet begins termination. The kubelet sequence:

```
preStop hook ⟶ SIGTERM ⟶ wait terminationGracePeriodSeconds
   (default 30s) ⟶ SIGKILL
```

`terminationGracePeriodSeconds` is the **total** budget shared by the preStop hook *plus* your SIGTERM handler. Here is the race: endpoint removal is **eventually consistent**, so for a short window proxies may still route to a Pod that is already terminating. A small `preStop` sleep absorbs that window, letting in-flight requests land before your app starts draining.

```
spec:
     terminationGracePeriodSeconds: 45
     containers:
     - name: checkout
       lifecycle:
         preStop:
           exec: { command: ["sh", "-c", "sleep 5"] }
```

Your app must still **catch SIGTERM** and drain connections within the remaining budget (45s minus 5s here). Relying on SIGKILL drops in-flight requests.

✓ PRODUCTION CHECKLIST

- ☐ Set CPU **and** memory `requests` on every container the HPA targets; without them, utilization HPA silently breaks.
- ☐ Use `maxUnavailable: 0` plus `maxSurge: 1` and a real readiness probe for capacity-preserving rollouts of request-serving apps, then add graceful shutdown to protect in-flight requests.
- ☐ Pin HPA to `autoscaling/v2`, and set `minReplicas ≥ 2` so a single Pod failure isn't an outage.
- ☐ Add a `preStop` sleep of 5 to 10s to any Pod behind a Service to cover endpoint-propagation lag.
- ☐ Ensure your app traps **SIGTERM** and drains within `terminationGracePeriodSeconds`; raise the grace period if drain takes longer.
- ☐ Don't run VPA (`Auto`) and HPA against the **same** CPU or memory signal; give HPA CPU (or a custom/external metric) and restrict VPA to memory, or keep VPA recommend-only.
- ☐ Pair every autoscaled Deployment with a **PodDisruptionBudget**: it gates *voluntary* evictions through the Eviction API (node drains, cluster-autoscaler node consolidation), so an upgrade or node scale-in can't drain your last healthy replicas. A PDB does not throttle the Deployment's own rolling update or an HPA scaling in (both delete Pods directly), and it cannot stop an involuntary node failure.

⚠ FAILURE MODES TO WATCH

- **HPA shows `<unknown>` and never scales** → the target Pods have no CPU or memory request; *fix:* set resource requests on the container.
- **Rollout drops requests despite a "rolling" update** → no readiness probe, so Pods receive traffic before they are ready; *fix:* add a readiness probe and `maxUnavailable: 0`.
- **VPA and HPA thrash replicas up and down forever** → both act on CPU; *fix:* keep HPA on CPU (or a custom/external metric) and restrict VPA to memory, or run VPA recommend-only.
- **Clients see connection resets on every deploy** → the app is SIGKILLed because it ignores SIGTERM or has no preStop; *fix:* handle SIGTERM to drain, and add a preStop sleep.
- **Pods stuck `Pending`, nodes never appear** → the Cluster Autoscaler node group is at max size, or pending Pods request an instance shape no group offers; *fix:* raise the group max or add a matching group, or adopt Karpenter for groupless provisioning.

CHAPTER 4

Surviving Disruptions

Reliability is mostly about what happens when something goes away. Nodes get drained for upgrades, cloud VMs disappear, a kernel panics, an availability zone goes dark. Your job is to ensure that when capacity leaves, whether planned or not, enough survives to keep serving. This chapter covers the mental model and the three tools that enforce it: **PodDisruptionBudgets**, **spreading**, and the right **rollout pattern**.

Voluntary vs involuntary disruptions

Kubernetes draws a hard line between two kinds of disruption, and it matters because only one of them is something you control.

Voluntary disruptions are initiated through the cluster: a `kubectl drain` for a node upgrade, a cluster autoscaler scaling down, an operator that *evicts* a Pod. When these go through the **Eviction API** (`POST .../pods/{name}/eviction`), the Eviction API *respects PodDisruptionBudgets*: if an eviction would push you below your budget, the API returns `429 Too Many Requests` and refuses it until enough Pods are healthy again. The critical nuance: a PDB is enforced *only* by the Eviction API. A plain `kubectl delete pod` (or deleting the Deployment) is a normal, non-policy-controlled DELETE, so it **bypasses the PDB entirely**. This is why you drain and evict, and use controllers that evict, rather than deleting Pods directly, when you want the budget honored.

Involuntary disruptions are everything the cluster did not choose: hardware failure, kernel panic, an OOM kill, a node losing network, someone pulling a power cable. These **bypass PDBs entirely**, because there is no eviction call to intercept. A PDB cannot save you from a dead node.

```
voluntary  → Eviction API → checks PDB → may be blocked (429)
   involuntary → node just dies → PDB never consulted
```

The takeaway: PDBs protect you during *maintenance*, while **spreading and replica count** protect you during *failures*. You need both.

PodDisruptionBudgets: protecting capacity during drains

A PDB tells the Eviction API the minimum service you must keep. It lives in `policy/v1`, and you set **exactly one** of `minAvailable` or `maxUnavailable`, never both.

```
apiVersion: policy/v1
   kind: PodDisruptionBudget
   metadata:
     name: checkout-pdb
   spec:
     maxUnavailable: 1          # at most 1 pod down at a time
     selector:
       matchLabels:
         app: checkout
```

`maxUnavailable: 1` lets a drain evict one Pod, wait for its replacement to go Ready, then evict the next. `minAvailable` can be an absolute number or

a percentage (`minAvailable: 90%`).

The trap: **a PDB with `minAvailable` equal to the full replica count blocks every voluntary eviction.** With 3 replicas and `minAvailable: 3`, the Eviction API can never approve a single eviction, so a node drain hangs forever. Always leave at least one Pod movable. For a Deployment of N replicas, `maxUnavailable: 1` (or `minAvailable: N-1`) is the safe default.

Spreading Pods: anti-affinity and topologySpreadConstraints

A PDB is useless if all your Pods sit on one node, because losing that node takes them all at once, involuntarily, with no PDB protection. You must physically spread them across failure domains.

`topologySpreadConstraints` is the modern, scalable tool. You pick a `topologyKey` (the failure domain), a `maxSkew` (allowed imbalance), and a `whenUnsatisfiable` policy:

```
spec:
     topologySpreadConstraints:
       - maxSkew: 1
         topologyKey: topology.kubernetes.io/zone
         whenUnsatisfiable: DoNotSchedule        # hard
         labelSelector:
           matchLabels:
             app: checkout
       - maxSkew: 1
         topologyKey: kubernetes.io/hostname
         whenUnsatisfiable: ScheduleAnyway       # soft
         labelSelector:
           matchLabels:
             app: checkout
```

`maxSkew: 1` with `topology.kubernetes.io/zone` keeps the per-zone Pod counts within 1 of each other. `DoNotSchedule` is a hard constraint (the scheduler refuses to violate it, leaving Pods `Pending` if it can't comply), while `ScheduleAnyway` is a soft preference (best-effort, and it never blocks scheduling).

Pod anti-affinity does the same thing conceptually, but per Pod relationship. `requiredDuringSchedulingIgnoredDuringExecution` is hard (it forbids co-location outright), while `preferredDuringSchedulingIgnoredDuringExecution` is soft. Anti-affinity is comparatively **expensive to evaluate at scale**, because the scheduler compares every candidate Pod against every existing matching Pod. So for plain zone or node balance, **prefer topology spread**. Reach for anti-affinity when you need "never put these two specific workloads together."

Multi-AZ design and failure domains

A failure domain is the blast radius of one fault. The standard hierarchy runs **container → Pod → node → zone → region**. Design so that no single domain failure takes the whole service.

Practical rules:

- Run replicas across **at least 3 zones** with zone-level spread (`maxSkew: 1`). Two zones means a zone loss halves you; three means you lose a third and degrade gracefully.
- Keep enough headroom that losing one zone's worth of Pods still meets demand. If 3 zones run at 100% utilization, losing one drops you to 67% capacity under full load.
- Remember that regional services (a single-AZ managed database, a load balancer) are their own failure domains. Spreading compute doesn't help if all Pods talk to one zonal datastore.

Safe node drains and cluster upgrades

Node upgrades *are* drains. `kubectl drain` cordons the node (marks it unschedulable), then evicts every Pod through the Eviction API, honoring your PDBs. Two flags you almost always need:

```
kubectl drain ip-10-0-1-5.ec2.internal \
     --ignore-daemonsets \
     --delete-emptydir-data
```

`--ignore-daemonsets` is required because DaemonSet Pods are managed per-node and would otherwise block the drain. `--delete-emptydir-data` is required to evict Pods using `emptyDir` volumes (you are acknowledging that scratch data is lost). Without these flags, the drain refuses to start.

The drain proceeds Pod by Pod, pausing whenever an eviction would breach a PDB and retrying until the budget allows it. This is exactly why a too-strict PDB stalls an upgrade: the rolling node replacement can never get the green light to move the next Pod.

Rolling vs blue-green vs canary

Disruption isn't only infrastructure. Your own deploys disrupt running Pods too. Three patterns, three trade-offs:

- **Rolling update** (the Deployment default): replaces Pods incrementally *in place*, governed by `maxUnavailable` and `maxSurge`. There is no extra full environment and no instant rollback; you roll forward or roll back through the same gradual churn.
- **Blue-green**: stand up a second full environment (green) alongside the live one (blue), then switch traffic instantly at the router or Service. Rollback is immediate (flip back to blue), but you pay **double capacity** during the cutover.
- **Canary**: shift a **small percentage** of traffic to the new version first, watch error rate, latency, and SLOs, then ramp up. This gives the best validation under real traffic, but it needs traffic-splitting (a service mesh, ingress weights, or a progressive-delivery controller).

Rolling is the sane default. Use canary when the blast radius of a bad version is expensive, and use blue-green when you need instant, atomic rollback and can afford the spare capacity.

✓ PRODUCTION CHECKLIST

- ☐ Define a PDB for every customer-facing Deployment; use `maxUnavailable: 1` (or `minAvailable: N-1`), never `minAvailable` equal to the full replica count.
- ☐ Set exactly one of `minAvailable` or `maxUnavailable` per PDB; the API rejects both.
- ☐ Add a zone-level `topologySpreadConstraints` (`maxSkew: 1`, `topology.kubernetes.io/zone`) to every multi-replica workload.
- ☐ Run at least 3 replicas across at least 3 zones, and size headroom so one zone can fail at peak.
- ☐ Test a real `kubectl drain --ignore-daemonsets --delete-emptydir-data` on a node before a live upgrade and confirm it completes.
- ☐ Verify PDBs don't stall drains: drain a node in staging and watch for `Cannot evict pod ... would violate PDB`.
- ☐ Pick a rollout pattern per service and document the rollback steps; wire canary and blue-green to your SLO metrics, not just "Pods are Ready".

⚠ **FAILURE MODES TO WATCH**

- **PDB set to the full replica count** → every node drain hangs indefinitely; *fix:* leave at least one Pod movable (`maxUnavailable: 1`).
- **All replicas land on one node or zone** despite a PDB → one involuntary failure wipes the service; *fix:* add `topologySpreadConstraints` with `maxSkew: 1`.
- **Drain fails to start** with DaemonSet or emptyDir errors → upgrade automation stalls; *fix:* pass `--ignore-daemonsets` and `--delete-emptydir-data`.
- **Two zones instead of three** → a single zone loss removes 50% of capacity and overloads the rest; *fix:* spread across 3 or more zones with headroom for one loss.
- **PDB selector doesn't match the Pods** → the budget silently protects nothing and drains evict everything at once; *fix:* confirm `kubectl get pdb` shows the expected `ALLOWED DISRUPTIONS` and current healthy count.

CHAPTER 5

Observability That Actually Helps

You can't keep a system reliable if you can't see it. But "more dashboards" is not observability. Most teams drown in panels that look busy and answer nothing. This chapter gives you a small set of mental models that tell you *what* to measure, and just enough PromQL to turn those models into queries that survive contact with production.

The four golden signals

Google's SRE book distills monitoring to four signals. If you measure only these, you can catch most user-facing problems:

- **Latency:** how long requests take. Critically, measure latency for *successful* and *failed* requests separately. A fast stream of HTTP 500s will make your average latency look great while users are furious.
- **Traffic:** demand on the system (requests/sec, messages/sec).
- **Errors:** the rate of failing requests (explicit 5xx, or implicit ones such as wrong content).
- **Saturation:** how "full" the system is, measured at the resource closest to its limit (CPU, memory, IO, a connection pool), usually expressed as utilization toward a hard ceiling.

Golden signals are the *what*. RED and USE are two recipes for applying them.

RED for services, USE for resources

RED (coined by Tom Wilkie, building on Rob Ewaschuk's SRE work) is for *request-driven services*, meaning anything that answers requests:

```
Rate      : requests per second
   Errors    : failed requests per second
   Duration  : distribution of request latency
```

USE (Brendan Gregg) is for *resources*: CPU, memory, disk, network, and queues:

```
Utilization : % time the resource was busy
   Saturation  : how much extra work is queued/waiting
   Errors      : error events (e.g. NIC drops, disk errors)
```

Rule of thumb: monitor every **service** with RED, and every **resource** with USE. Checkout's container is a resource (USE: CPU throttling, memory pressure); the checkout API it serves is a service (RED: rate, errors, p99 duration against its 250 ms target).

Metrics, logs, and traces: what each is good for

These three are not interchangeable. Each has a job:

```
Metrics → cheap aggregates over time      → DETECT  ("error rate
   just doubled")
   Traces  → one request across services     → LOCALIZE ("checkout →
   payments is slow")
   Logs    → discrete high-cardinality events→ CONFIRM ("payment
   timeout to gateway X")
```

- **Metrics** are cheap, pre-aggregated numbers. They are great for alerting and trends, and bad at explaining a single request.
- **Traces** follow one request's path across services as parent and child spans. They show you *where* time went and *which hop* failed.
- **Logs** are high-cardinality discrete events. They carry the detail (a stack trace, the exact bad input) but are expensive to query at scale.

The workflow: **metrics detect, traces localize, logs confirm root cause.** Reach for them in that order.

Prometheus and Grafana: the standard stack

Prometheus is a **pull-based** time-series database: it scrapes an HTTP `/metrics` endpoint on each target on an interval and stores the samples. Grafana sits on top for dashboards. In Kubernetes the common path is the kube-prometheus-stack (Prometheus Operator), where you declare scrape targets as `ServiceMonitor` objects rather than editing config by hand:

```
apiVersion: monitoring.coreos.com/v1
   kind: ServiceMonitor
   metadata:
     name: checkout
     labels:
       release: kube-prometheus-stack
   spec:
     selector:
       matchLabels:
         app: checkout
     endpoints:
       - port: metrics      # named port on the Service
         interval: 30s
         path: /metrics
```

Because scraping is pull-based, an instrumented app just exposes `/metrics`. It never needs to know where Prometheus lives.

PromQL basics: rate, histograms, and quantiles

Prometheus has four metric types; the two you'll touch constantly are **counters** and **histograms**.

Counters only ever increase (and reset to 0 on restart). A raw counter value is meaningless, so you almost always wrap it in `rate()` or `increase()` over a time window. `rate()` gives the per-second average over the range and transparently handles resets:

```
# requests/sec, averaged over 5m
   sum(rate(http_requests_total[5m]))

   # error ratio: fraction of requests returning 5xx
   sum(rate(http_requests_total{code=~"5.."}[5m]))
     /
   sum(rate(http_requests_total[5m]))
```

That second query is your RED "Errors" signal in one line. Pick a range (`[5m]`) of at least 4× your scrape interval so you always cover several samples.

For latency (RED "Duration"), instrument with a **histogram**, which exports cumulative `_bucket` counters labelled by `le` ("less than or equal"). Quantiles are computed server-side:

```
histogram_quantile(
     0.99,
     sum(rate(http_request_duration_seconds_bucket[5m])) by (le)
   )
```

Three things bite people here: the `le` label and the `_bucket` suffix are **required**; you must `rate()` the buckets before summing; and the result is an **estimate** computed by linear interpolation within the matched bucket (Prometheus assumes a uniform spread of observations inside it), so its error is bounded by the bucket width and coarse buckets give coarse p99s. Choose buckets around your SLO (for example 50ms, 100ms, 250ms, 500ms, 1s).

Cardinality traps and dashboard discipline

A time series is unique per *combination* of label values. **Cardinality is the product of all label-value counts.** For example, `method` (5) × `path` (20) × `status` (8) = 800 series for one metric, which is fine. The killers are **unbounded** label values:

```
BAD  labels: user_id, request_id, raw URL (/order/8a3f...), email,
   session token
   GOOD labels: method, route template (/order/:id), status_code,
   region
```

Put a user ID in a label and you get one series per user, which means millions of series and a Prometheus that OOMs. The fix: labels must have a small, bounded set of possible values. High-cardinality detail belongs in **logs or traces**, never in metric labels.

Dashboard discipline follows from the same instinct: a panel that doesn't help you make a decision is noise. Build top-down, with one overview row of RED per service and drill-downs below, and template with variables (`$namespace`, `$service`) instead of cloning 40 near-identical panels.

✓ PRODUCTION CHECKLIST

- ☐ Instrument every service with RED (rate, errors, p99 duration) and every node and container with USE.
- ☐ Measure latency for successful and failed requests as separate series.
- ☐ Export latency as a histogram with `_bucket` and `le`, with buckets chosen around your SLO thresholds.
- ☐ Always wrap counters in `rate()` or `increase()`; never alert on a raw counter value.
- ☐ Audit metric labels for unbounded values (IDs, URLs, emails) and move that detail to logs or traces.
- ☐ Define scrape targets declaratively through `ServiceMonitor` or `PodMonitor`, not hand-edited config.
- ☐ Build dashboards top-down with template variables, and delete panels that don't drive a decision.

⚠ **FAILURE MODES TO WATCH**

- **Averaged latency hides pain** → failed-fast and slow-success requests are mixed together; *fix:* split by status and chart quantiles, not `avg`.
- **Cardinality explosion OOMs Prometheus** → a user ID or request ID leaks into a label; *fix:* drop the label with relabeling and move detail to traces or logs.
- `histogram_quantile` **returns NaN or wrong values** → buckets summed without `rate()`, or `le` aggregated away; *fix:* `sum(rate(..._bucket[5m])) by (le)` before the quantile.
- **p99 estimate is imprecise** → buckets are too coarse, so `histogram_quantile` interpolates across a wide bucket and the error is bounded only by that bucket's width; *fix:* add buckets near your SLO latency.
- **Scrapes silently stop** → a renamed port or dropped `ServiceMonitor` label leaves you blind; *fix:* alert on `up == 0` and on `absent()` of key metrics.

CHAPTER 6

SLO-Based Alerting and On-Call

You defined SLOs in Chapter 1 and the metrics to measure them in Chapter 5. This chapter turns them into pages, but only the pages that matter. The goal: when your phone buzzes, a real customer is hurting and there is something you can do about it. Everything else is a ticket, a dashboard, or deleted.

Spending the error budget: policy, not just math

An error budget is the amount of unreliability you are *allowed*. A 99.9% availability SLO over 30 days permits about 43 minutes of badness. That number is useless until it drives decisions. The policy is the real artifact: *budget healthy → ship features fast; budget spent → freeze risky changes and pay down reliability*. Alerting is just the early-warning system for that policy. If you alert without a policy, you generate noise nobody is empowered to act on.

Why alert on symptoms, not causes

Alert on what the user feels, such as an elevated error ratio or latency SLO burn, not on causes like high CPU, a full disk, or pod restarts. There are two reasons. First, cause-based alerts are noisy: CPU can sit at 95% all day with users perfectly happy. Second, they miss novel failures: your next

outage will have a cause you never wrote an alert for, but the *symptom* (errors up, latency up) is always the same. Symptom alerts catch the unknown unknowns. Keep cause metrics on dashboards for diagnosis, but don't page on them.

```
cause (CPU, disk, GC)  →  page?  NO   → dashboard / diagnosis
   symptom (error %, p99) →  page?  YES  → customer is hurting
```

Burn rate: the core idea

Burn rate is the multiple at which you are consuming error budget relative to the sustainable rate. A burn rate of **1** spends the budget exactly evenly across the whole SLO window: you finish the 30 days with zero left, on schedule. A burn rate of **N** exhausts the budget in `window / N`. So at 2x you burn out in 15 days, and at 14.4x you burn out in roughly 50 hours.

Concretely, for a 99.9% SLO the sustainable error ratio is 0.1%. If your current error ratio is 1.44%, that is `1.44% / 0.1% = 14.4x`. Burn rate is just *(observed bad ratio) / (1 – SLO)*.

Multi-window, multi-burn-rate alerts

A single threshold forces a bad trade-off: a long window is slow to fire and slow to reset, while a short window false-alarms on every transient blip. The fix is to require **both** a long window and a short window to be over threshold simultaneously. The short window (recommended: **1/12 of the long**) confirms the problem is *still happening right now*, so the alert fires

fast and, crucially, resolves fast once the burn stops, instead of staying lit while the long window slowly forgets spent budget.

The canonical fast-burn page: a **14.4x** burn rate sustained over **1 hour** consumes **2%** of a 30-day budget. Gate it with a **5-minute** short window so it clears quickly. (`14.4 × 1h / (30 × 24h) = 2%`.)

A practical tier set:

Severity	Burn rate	Long / short window	Budget consumed
Page	14.4x	1h / 5m	2%
Page	6x	6h / 30m	5%
Ticket	3x	24h	10%
Ticket	1x	72h	10%

Fast burns page you (something is on fire now); slow burns open a ticket (you have days to fix it).

Here is the fast-burn page as Prometheus rules. Assume a recording rule such as `job:slo_errors:ratio_rate5m` exposes the bad-event ratio over each window:

```
groups:
  - name: slo-burn-rate
    rules:
    - alert: ErrorBudgetFastBurn
      # 14.4x over 1h, gated by 5m so it resolves quickly
      expr: |
        job:slo_errors:ratio_rate1h{service="checkout"} > (14.4 *
  0.001)
        and
        job:slo_errors:ratio_rate5m{service="checkout"} > (14.4 *
  0.001)
      for: 2m
      labels:
        severity: page
      annotations:
        summary: "Checkout burning error budget at >14.4x (2% in
  1h)"
        runbook_url: "https://runbooks.internal/checkout/error-
  budget-burn"
    - alert: ErrorBudgetSlowBurn
      # 6x over 6h, gated by 30m
      expr: |
        job:slo_errors:ratio_rate6h{service="checkout"} > (6 *
  0.001)
        and
        job:slo_errors:ratio_rate30m{service="checkout"} > (6 *
  0.001)
      for: 5m
      labels:
        severity: page
      annotations:
        summary: "Checkout burning error budget at >6x (5% in 6h)"
        runbook_url: "https://runbooks.internal/checkout/error-
  budget-burn"
```

`0.001` is `1 - 0.999`, the budget for a 99.9% SLO. Swap it for your own target. The `and` of two windows is the whole trick.

Killing alert fatigue and writing runbooks

Every paging alert must link a runbook, and every runbook must answer four questions: **what the alert means**, **how to confirm real customer impact**, **first diagnostic steps**, and **safe mitigations** (roll back, drain a node, scale up, fail over). If an alert has no clear action, it does not deserve to wake a human. Make it a ticket or a dashboard panel instead. A page without a runbook is a page that takes twice as long to resolve.

```
# Runbook: Checkout error-budget burn
   **Means:** checkout error ratio is high enough to spend the 30-day
   budget fast.
   **Confirm impact:** Grafana "Checkout SLO" → is the user-facing
   error % actually up?
   **Diagnose:** recent deploy? (kubectl rollout history) upstream
   5xx? DB latency?
   **Mitigate:** `kubectl rollout undo deploy/checkout`; if upstream,
   fail over region.
```

Treat every page in retro as one of two things: actionable (keep it, improve the runbook) or noise (tune the threshold or delete the alert). Alerts are code: they need maintenance, not accumulation.

Humane on-call rotations

Reliability includes the humans. Sustainable on-call starts with a cap on pages per shift; a common target is no more than about 2 actionable pages per 12-hour shift, and beyond that you fix the underlying noise before the next rotation. Staff enough engineers (roughly 6 to 8) so no one is on call more than about 1 week in 4 or 5. Compensate on-call explicitly, with pay or time off in lieu. And make "every page is actionable or deleted" a standing rule, reviewed weekly. Burned-out engineers cause outages, so a humane rotation is a reliability control, not an HR nicety.

✓ PRODUCTION CHECKLIST

- ☐ Write SLO burn-rate alerts as multi-window, multi-burn-rate rules, where both the long and short windows must exceed threshold.
- ☐ Implement the fast-burn page at 14.4x (1h/5m, 2% budget) and the slower page at 6x (6h/30m, 5% budget).
- ☐ Route 3x/24h and 1x/72h slow burns to tickets, not pages.
- ☐ Page only on symptoms (error ratio, latency SLO burn), and keep CPU, disk, and restart alerts on dashboards.
- ☐ Attach a runbook URL to every paging alert, and ensure it covers meaning, impact confirmation, diagnosis, and mitigation.
- ☐ Encode an error-budget policy (freeze risky changes when budget is spent) and tie alert severities to it.
- ☐ Review every page weekly: keep-and-improve, or tune-and-delete. Track pages per shift.

⚠ **FAILURE MODES TO WATCH**

- **Single long-window alert only** → it fires slowly and stays lit for hours after recovery because spent budget lingers; *fix:* add a short window (1/12 of the long) as an `and` gate.
- **Paging on causes (CPU, disk)** → on-call floods with non-customer-facing noise and still misses novel outages; *fix:* move cause metrics to dashboards and page on symptoms.
- **Wrong budget constant in the expr** → using `0.01` for a 99.9% SLO makes the alert 10x too quiet, so it never fires; *fix:* set the threshold to `burn_rate × (1 − SLO)`, for example `14.4 × 0.001`.
- **Alerts without runbooks** → every page becomes a from-scratch investigation and MTTR balloons; *fix:* block merge of any paging rule that lacks a `runbook_url`.
- **Understaffed rotation** → too few engineers means too-frequent shifts and burnout, which causes more outages; *fix:* staff 6 to 8 per rotation and cap actionable pages per shift.

CHAPTER 7

Incident Response and Debugging

When a Pod misbehaves, the goal is not to guess. It is to follow a fixed path that turns symptoms into root cause in under a minute. We will run that path against a live incident: an hour after a routine deploy, **checkout** started failing and the fast-burn alert from Chapter 6 paged. Kubernetes already wrote down what went wrong; you just need to read it in the right order.

The repeatable first-look workflow

Three commands, always in this order:

```
kubectl describe pod <pod>          # events are at the BOTTOM
   kubectl get events --sort-by=.lastTimestamp -n <ns>
   kubectl logs <pod> [-c <container>]
   kubectl logs <pod> --previous       # the PRIOR crashed instance
```

`describe` tells you the **state machine** of the Pod: phase, container statuses, restart counts, and an Events block at the bottom that is the kubelet and scheduler narrating what they tried and why it failed. Read the Events first, because most root causes are spelled out there in plain English.

`get events --sort-by=.lastTimestamp` widens the lens to the whole namespace and orders events chronologically, so you see the sequence (scheduled → pulled → started → killed) rather than a single Pod's slice. Events are retained for only about 1 hour by default, so capture them early.

`logs` is the application's own voice. The critical flag is `--previous`: when a container has already crashed and restarted, plain `logs` shows the *new* (possibly healthy-looking) instance. `--previous` shows the dead one, and that is where the stack trace and the real error live.

```
describe → "what did K8s try?"   (scheduler/kubelet view)
   events   → "in what order?"      (namespace timeline)
   logs     → "what did the app say?" (--previous = the corpse)
```

Pending: the scheduler couldn't place the Pod

`Pending` means the Pod was accepted by the API server but **no node was chosen**. The scheduler's reason is in the Pod's events (`FailedScheduling`). The usual suspects:

- **Insufficient CPU or memory:** the Pod's *requests* exceed any node's allocatable. Message: `0/5 nodes are available: 3 Insufficient cpu`.
- **Unsatisfiable nodeSelector or affinity:** no node matches the labels you required.
- **Taints without tolerations:** `node had taint {key: value}, that the pod didn't tolerate`.

- **Unbound PVC:** the Pod waits on a PersistentVolumeClaim that has no matching PersistentVolume or no provisioner.

```
kubectl describe pod <pod> | grep -A5 Events
   kubectl describe node <node> | grep -A5 Allocatable
```

Fix it by lowering requests, relaxing constraints, adding a toleration, or fixing the StorageClass. Deleting and recreating the Pod changes nothing.

CrashLoopBackOff and ImagePullBackOff

CrashLoopBackOff does *not* mean "crash loop is the bug." It means the container starts, exits, and the kubelet is **backing off** restarts with exponential delay (10s, 20s, 40s … **capped at 5 minutes**). The backoff is the kubelet behaving correctly. The root cause is in the exited container's logs:

```
kubectl logs <pod> --previous
```

Common causes: bad config or env, a missing dependency at startup, failing migrations, or a process that exits 0 because there is nothing to keep it alive.

ImagePullBackOff / ErrImagePull means the kubelet cannot pull the image at all. `describe` shows the exact error, so read it literally:

- Wrong image name or tag (a typo, or a tag that doesn't exist).
- A private registry with no `imagePullSecret`.

- Registry rate limiting (for example, Docker Hub anonymous pull limits).

```
kubectl describe pod <pod> | grep -A3 'Failed'
   # Fix private-registry pulls:
   kubectl create secret docker-registry regcred \
     --docker-server=<registry> --docker-username=<u> --docker-
   password=<p>
```

Then reference `regcred` under `imagePullSecrets` in the Pod spec.

OOMKilled and Evicted: two different killers

These look similar but come from different layers.

OOMKilled (exit code 137) is a *container* event: the process exceeded its memory **limit**, and the kernel's OOM killer terminated it. You'll see `Reason: OOMKilled` in the container's `lastState`. Fix it by raising `resources.limits.memory`, or by fixing the leak. Note that 137 = 128 + 9 (SIGKILL). This is exactly what hit checkout: the deploy an hour earlier added an in-memory cart cache, real traffic pushed the process past its 512Mi limit, and the OOM killer started terminating replicas. Each kill fed the next `CrashLoopBackOff` as Pods restarted straight back into the same pressure.

Evicted is a *node* decision: the kubelet detected **node pressure** (memory or disk) and evicted Pods to reclaim resources. The kubelet does **not** sort by QoS class directly. It ranks Pods by whether their usage exceeds their **requests** first, then by **Pod Priority**, then by how far

usage runs over requests. QoS only *correlates* with the outcome: BestEffort Pods set no requests, so they always exceed them, while Guaranteed Pods never do, which is why in practice BestEffort tends to be evicted before Burstable before Guaranteed. An evicted Pod's status shows `Reason: Evicted` with a message like `The node was low on resource: memory`.

```
kubectl get pod checkout-7d9f8b-4qx2p -o
   jsonpath='{.status.containerStatuses[0].lastState.terminated.reaso
   n}'
   # OOMKilled
```

For checkout the mitigation was one command: `kubectl rollout undo deploy/checkout` put the previous image back and stopped the burn within minutes. The durable fix came after: size the memory limit to real P99 and bound the cache, rather than let it grow unmetered.

The durable fix for eviction is **setting requests correctly** so QoS reflects reality, plus node-level memory and disk headroom. Bumping one Pod's limit is not enough.

Ephemeral debug containers

Modern images are often distroless, with no shell, no `ps`, and no `curl`. You can't `exec` into them. `kubectl debug` attaches an **ephemeral container** into the running Pod, sharing the target's process namespace, without a restart:

```
kubectl debug -it <pod> --image=busybox --target=<container> -- sh
```

`--target=<container>` is what shares the PID namespace, so from busybox you can see and inspect the target's processes (`ps`, `/proc/<pid>`), its network, and its localhost ports. Ephemeral containers cannot be removed once added, and they can't have ports or resources. They exist only to debug, and they vanish when the Pod is deleted.

MTTR, blameless postmortems, and RCA

The headline incident metric is **MTTR**, mean time to recovery: the span from detection to service restored. Everything above exists to drive MTTR down, because a fast, repeatable diagnosis path beats heroics.

After recovery, run a **blameless postmortem**. The premise: incidents come from **systemic and process gaps**, not bad individuals. You write down the timeline, the contributing causes, and what the system *allowed* to happen, then produce **tracked action items** with owners and due dates. The reason blameless is non-negotiable: punishing people makes them stop reporting, and hidden incidents are the ones that recur and compound. A good RCA asks "why did our system let this happen and not catch it?", not "who pushed the button?"

✓ PRODUCTION CHECKLIST

- ☐ On every incident, run `describe` → `get events --sort-by=.lastTimestamp` → `logs --previous` before touching anything.
- ☐ Capture events within the hour; they expire (about 1h default TTL) and are then gone.
- ☐ For `Pending`, read the `FailedScheduling` event and compare Pod requests to node `Allocatable`.
- ☐ For `CrashLoopBackOff`, always read `--previous` logs, and never "fix" the backoff itself.
- ☐ Set both memory `requests` and `limits` so the QoS class is intentional, not accidental BestEffort.
- ☐ Pre-create an `imagePullSecret` for every private registry and reference it in the Pod spec.
- ☐ Keep a `kubectl debug` runbook for distroless images so on-call isn't improvising at 3 a.m.
- ☐ Every postmortem ends with tracked, owned, dated action items, not just a writeup.

⚠ FAILURE MODES TO WATCH

- **Reading current logs after a crash and seeing nothing wrong** → you are looking at the restarted instance; *fix:* use `kubectl logs --previous`.
- **Treating `Pending` as a node outage and recreating the Pod** → it is a scheduling constraint; *fix:* read the event and address requests, affinity, taints, or the PVC.
- **Bumping a Pod's memory limit to stop evictions** → eviction is node pressure on low-QoS Pods; *fix:* set requests correctly and add node headroom instead.
- **`exec` failing on a distroless image and assuming the Pod is broken** → there is no shell; *fix:* use `kubectl debug --target` with an ephemeral container.
- **Losing the root cause because events aged out** → they default to about 1h retention; *fix:* snapshot `get events` and `describe` into the incident channel immediately.

CHAPTER 8

Reliability Guardrails

Most incidents are not caused by exotic failures. They are caused by a `latest` tag that moved, a Pod with no limits that ate a node, a credential that could do far more than it needed, or a known CVE that nobody scanned for. Guardrails are cheap controls that stop these classes of failure *before* they reach production. This chapter is about wiring them in so reliability is the default, not a heroic afterthought.

Security as reliability: least-privilege RBAC

A compromised or buggy workload with broad permissions is a reliability problem, not just a security one. It can delete other teams' Deployments or exhaust the API server. Kubernetes RBAC is **additive and deny-by-default**: nothing is allowed until a binding grants it, and there are no deny rules. You grant the minimum and nothing more.

Two scopes matter:

- **Role / RoleBinding:** namespaced. Permissions apply only within one namespace.
- **ClusterRole / ClusterRoleBinding:** cluster-wide, and the only way to grant access to cluster-scoped resources like nodes or PersistentVolumes.

Least privilege in practice means no wildcard verbs or resources (`verbs: ["*"]`, `resources: ["*"]`), scoping to specific resource names where you can, and **never binding the built-in `cluster-admin` ClusterRole to a workload's ServiceAccount**. A pod that only reads ConfigMaps should get exactly that:

```
apiVersion: rbac.authorization.k8s.io/v1
   kind: Role
   metadata:
     namespace: payments
     name: configmap-reader
   rules:
   - apiGroups: [""]
     resources: ["configmaps"]
     verbs: ["get", "list", "watch"]
```

Audit with `kubectl auth can-i --list --as=system:serviceaccount:payments:myapp` to see exactly what a ServiceAccount can do.

Pair RBAC with **Pod Security Admission** (built in since v1.25, replacing the removed PodSecurityPolicy). It enforces three standards (**Privileged**, **Baseline**, and **Restricted**) per namespace through labels, with no third-party controller:

```
apiVersion: v1
   kind: Namespace
   metadata:
     name: payments
     labels:
       pod-security.kubernetes.io/enforce: restricted
       pod-security.kubernetes.io/warn: restricted
```

`enforce` rejects violating Pods, while `warn` and `audit` surface them without blocking, so you can roll it out gradually.

Image scanning and supply chain: Trivy and SBOMs

You cannot keep reliable what you cannot account for. **Trivy** scans container images, filesystems, and IaC (Terraform, Kubernetes manifests, Dockerfiles) for known CVEs and misconfigurations, and it can emit an **SBOM** in CycloneDX or SPDX format.

```
# Fail the CI build on fixable HIGH/CRITICAL CVEs only
   trivy image --exit-code 1 --severity HIGH,CRITICAL --ignore-
   unfixed checkout:1.4.2

   # Generate a CycloneDX SBOM for the artifact registry
   trivy image --format cyclonedx --output sbom.json checkout:1.4.2
```

Supply-chain hygiene has three moving parts: **pin image digests** (`myapp@sha256:…`, not a moving tag), **scan in CI** on every build, and **fail the build on fixable critical vulnerabilities** (`--ignore-unfixed` keeps you from blocking on CVEs with no patch yet). The SBOM becomes

your answer to "are we affected?" the next time a Log4Shell-class CVE lands: you query artifacts instead of guessing.

GitOps as a reliability control

GitOps (**Argo CD** or **Flux**) makes Git the single source of truth and runs a controller that **continuously reconciles** live cluster state back to the repo. The reliability wins are concrete:

- **Auditable changes:** every change is a reviewed, attributable commit.
- **Easy rollback:** `git revert`, and the controller converges the cluster back.
- **Drift detection and correction:** a manual `kubectl edit` is flagged as drift and, optionally, reverted automatically.

```
Git repo (desired)  ──►  GitOps controller  ──►  Cluster (actual)
          ▲                                          │
          └──── drift detected ──┘  reconcile loop
```

This kills the most common change-related outage: the undocumented hotfix that no one can reproduce or roll back.

Policy-as-code admission guardrails

RBAC controls *who* can act; **policy-as-code** controls *what* is allowed to exist. **OPA Gatekeeper** and **Kyverno** run as validating and mutating admission webhooks and reject non-compliant resources *at apply time*, before they ever schedule. Common guardrails: block Pods without

resource limits, reject privileged containers or `hostPath` mounts, and forbid the `latest` image tag.

```
apiVersion: kyverno.io/v1
   kind: ClusterPolicy
   metadata:
     name: require-resource-limits
   spec:
     validationFailureAction: Enforce
     rules:
     - name: require-limits
       match:
         any:
         - resources:
             kinds: ["Pod"]
       validate:
         message: "CPU and memory limits are required."
         pattern:
           spec:
             containers:
             - resources:
                 limits:
                   memory: "?*"
                   cpu: "?*"
```

Roll policies out in `Audit` mode first, watch the violation count, then flip to `Enforce`.

Right-sizing for cost without hurting reliability

Requests drive scheduling, so getting them wrong hurts reliability in both directions. **Set requests close to real P95/P99 usage** so the

scheduler bin-packs efficiently. **Over-requesting** wastes node capacity and money, because pods reserve headroom they never use. **Under-requesting** risks eviction and noisy-neighbor contention when real usage exceeds the reservation.

Measure before you guess. The **Vertical Pod Autoscaler** in recommendation-only mode (`updateMode: "Off"`) reports target requests from observed usage without mutating Pods, which is a safe way to find P90-ish targets (its default target percentile is 0.9, with a P95 `upperBound` for headroom). A practical pattern: set CPU *requests* near P95 and omit CPU *limits* (CPU is compressible, so throttling beats eviction), but always set *memory* limits near P99, since memory is incompressible and overage means an OOM kill. This is the guardrail checkout was missing when its cart cache grew unmetered in Chapter 7: VPA in recommendation mode would have surfaced a data-driven memory target from real usage (roughly P90, with a P95 `upperBound` for headroom), and a memory limit set above that headroom, plus a bound on the cache itself, turns that outage into a non-event.

A pragmatic intro to AIOps and anomaly detection

Static thresholds miss seasonal and trending behavior: 200 RPS is normal at midday and alarming at midnight. **AIOps and anomaly detection** learn those normal patterns (seasonality, trends) and flag deviations. Used well, this is a *signal to investigate*, not a pager.

Keep the hierarchy straight: **SLO burn-rate alerts remain your authoritative paging source** (Chapter 6). Anomaly detection is a secondary lens that surfaces "something changed" earlier, such as rising p99 latency or an unusual restart pattern, often before an SLO is breached. Two rules keep it from becoming noise: never page solely on an anomaly, and **always pair an anomaly with an action** (a runbook link or a dashboard to open). An anomaly with no defined response is just a notification people learn to ignore.

✓ PRODUCTION CHECKLIST

- ☐ Run `kubectl auth can-i --list --as=system:serviceaccount:<ns>:<sa>` for every production ServiceAccount, and remove wildcard verbs or resources and any `cluster-admin` binding.
- ☐ Label every namespace with `pod-security.kubernetes.io/enforce` (Baseline at minimum, Restricted for sensitive workloads); start with `warn` before `enforce`.
- ☐ Add a Trivy scan to CI that fails on `--severity HIGH,CRITICAL --ignore-unfixed` and publishes a CycloneDX or SPDX SBOM per build.
- ☐ Pin all production images to digests (`@sha256:…`) and ban the `latest` tag through a Kyverno or Gatekeeper policy.
- ☐ Manage cluster state through Argo CD or Flux with auto-sync and drift correction enabled, and make `git revert` the standard rollback.
- ☐ Deploy a policy controller and enforce three basics: resource limits required, no privileged containers, and no `hostPath`.
- ☐ Run VPA with `updateMode: "Off"` (recommendation only), then set CPU requests near P95 and memory limits from real P99 usage with headroom above it.
- ☐ Wire anomaly alerts to a non-paging channel, each with an attached runbook, and keep SLO burn-rate as the only pager.

⚠ **FAILURE MODES TO WATCH**

- **A workload bound to `cluster-admin`** → once compromised or buggy, it deletes other namespaces' resources; *fix:* scope it to a namespaced Role with explicit verbs.
- **A `latest` tag** → it silently rolls a broken image on the next pod reschedule; *fix:* pin digests and block `latest` at admission.
- **Pods with no memory limits** → they get OOM-killed, or take down a node, under load; *fix:* enforce limits through policy and right-size memory to P99.
- **A new admission policy flipped straight to `Enforce`** → it blocks every deploy cluster-wide; *fix:* roll out in `Audit` or `warn` mode and watch violations first.
- **Too many anomaly alerts** → the team drowns in them and starts ignoring the pager; *fix:* keep SLO burn-rate authoritative and attach an action to every anomaly.

APPENDIX A

Appendix A: The kubectl debugging cheat sheet

Fast, correct commands for an active incident. Replace `POD`, `NODE`, and `NS` as needed. Every flag below is valid on Kubernetes v1.29+.

Triage

```
# Cluster-wide pod health, sorted to surface the broken ones
   kubectl get pods -A -o wide --sort-by=.status.phase   # use when: first 30
   seconds, "what's not Running?"

   # Live resource usage for pods (needs metrics-server)
   kubectl top pods -A --sort-by=memory                  # use when: chasing OOM /
   memory pressure

   # Node-level capacity and usage
   kubectl top nodes                                     # use when: deciding if
   this is a node-pressure problem
```

Pods

```
# Full object state: events, probe results, restart reasons, QoS class
   kubectl describe pod POD -n NS            # use when: any single pod misbehaves;
   read the Events tail first

   # Last-terminated container's reason (exit code, OOMKilled, etc.)
   kubectl get pod POD -n NS -o
   jsonpath='{.status.containerStatuses[0].lastState.terminated}'   # use when:
   CrashLoopBackOff, need the exit reason

   # Logs from the previous (crashed) container instance
   kubectl logs POD -n NS --previous        # use when: container restarted and
   current logs are empty

   # Exec into a running container to poke around
   kubectl exec -it POD -n NS -- sh         # use when: you need to test
   DNS/connectivity from inside the pod

   # Attach an ephemeral debug container to a running pod (no rebuild)
   kubectl debug POD -n NS -it --image=busybox --target=CONTAINER   # use when: the
   app image has no shell/tools
```

Nodes and scheduling

```
# Node conditions, taints, allocatable vs. capacity, and recent events
   kubectl describe node NODE              # use when: pods Pending or a node went
   NotReady

   # Why is THIS pod unschedulable? The scheduler writes the reason into events
   kubectl get events -n NS --field-selector reason=FailedScheduling   # use when:
   a pod is stuck Pending

   # What's actually running on a node, across namespaces
   kubectl get pods -A --field-selector spec.nodeName=NODE -o wide     # use when:
   draining or isolating a bad node
```

Networking

```
# Does the Service have healthy backend endpoints?
   kubectl get endpointslices -n NS -l kubernetes.io/service-name=SVC   # use when:
   Service resolves but connections fail; empty endpoints means no ready pods

   # Inspect Service selector, ports, and type
   kubectl describe svc SVC -n NS          # use when: traffic isn't reaching
   pods; check the selector matches pod labels
```

Events and logs

```
# All events in a namespace, newest last
   kubectl get events -n NS --sort-by=.lastTimestamp   # use when: you need the
   timeline of what just happened

   # Stream logs from all pods behind a label, prefixed by pod name
   kubectl logs -n NS -l app=NAME --all-containers --prefix -f   # use when:
   tailing a whole deployment during a rollout
```

APPENDIX B

Appendix B: Top production failure modes

Symptom	Likely cause	First move
`CrashLoopBackOff`	App exits on startup: bad config, a missing secret or env var, a failed migration, or a panic	`kubectl logs POD --previous`; read the exit code via `lastState.terminated`
`ImagePullBackOff` / `ErrImagePull`	Wrong image tag, a private registry without `imagePullSecrets`, or a rate limit	`kubectl describe pod` and read the Events line; verify the tag and registry auth
`OOMKilled` (exit 137)	Container hit its memory limit, or the limit was set too low for real usage	`kubectl top pod`; compare actual usage to `resources.limits.memory`; raise the limit or fix the leak

Symptom	Likely cause	First move
Pod stuck `Pending` / unschedulable	No node fits: insufficient CPU or memory, taints, node selector or affinity, or an unbound PVC	`kubectl describe pod`, then read Events; check requests vs. allocatable and taints/tolerations
Readiness probe flapping	Probe too aggressive, or the app is slow under load, so the pod is pulled from endpoints intermittently	Raise `timeoutSeconds` or `failureThreshold`; add a `startupProbe` for slow boots
DNS resolution failures	CoreDNS overloaded or down, a NodeLocal DNSCache issue, or bad `ndots` search expansion	Test from the pod: `kubectl exec -- nslookup svc`; check CoreDNS pods and logs in `kube-system`
Node `NotReady`	kubelet down, network plugin failure, or resource pressure (disk, PID, or memory)	`kubectl describe node`, then read Conditions; check the kubelet and CNI on the node

Symptom	Likely cause	First move
PVC stuck `Pending`	No matching StorageClass or PV, a zone mismatch, or provisioner failure	`kubectl describe pvc`; verify the StorageClass exists and the provisioner is healthy
CPU throttling	Usage hitting the CPU limit: the cgroup throttles, latency spikes, and there is no OOM	`kubectl top pod`; check throttling metrics; raise or remove the CPU limit, or right-size it
Pods evicted	Node under disk or memory pressure, so the kubelet evicts the lowest-QoS pods first	`kubectl describe node`, then read pressure conditions; free disk, set requests equal to limits for Guaranteed QoS
Rollout stuck / not progressing	New pods never go Ready: failed probes, or `maxUnavailable` or a PDB blocking	`kubectl rollout status deploy/NAME`; `kubectl describe` the new ReplicaSet for the block reason

Symptom	Likely cause	First move
TLS cert expired	A webhook, Ingress, or serving cert is past expiry: cert-manager is not renewing, or a manual cert lapsed	Check cert expiry (`openssl x509 -enddate`); inspect the cert-manager `Certificate` and `order` status