

ELLYPUNK

Mastery

WEEKLY DIGEST / M401

28 September - 2 October 2026



This week

28 September - 2 October 2026

Select a title to turn to its pages, or open the original online.

MONDAY / 28 September 2026

Kubernetes Reliability Week - Part 1 of 5: Map Failure Domains 3

[Open original online](#)

TUESDAY / 29 September 2026

Kubernetes Reliability Week - Part 2 of 5: Design Resilient Workload Placement 8

[Open original online](#)

WEDNESDAY / 30 September 2026

Kubernetes Reliability Week - Part 3 of 5: Set Safe Rollout and Disruption Controls 13

[Open original online](#)

THURSDAY / 1 October 2026

Kubernetes Reliability Week - Part 4 of 5: Diagnose Control-Plane and Workload Failures 18

[Open original online](#)

FRIDAY / 2 October 2026

Kubernetes Reliability Week - Part 5 of 5: Run Production Recovery and Capacity Drills 23

[Open original online](#)

Original artwork is reproduced in full, in publication order.

Copyright Ellypunk. Contains AI-generated artwork.

ellypunk.com

M401
MASTERY

PRACTICE
UNDERSTAND
OPERATE
GO FURTHER

Kubernetes Reliability Week — Part 1 of 5: Map Failure Domains

Map service failure domains and choose a user-facing reliability objective before placement changes.

28
SEPTEMBER
2026

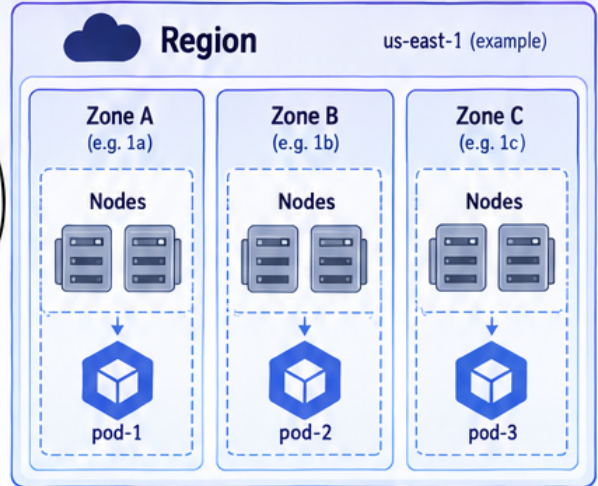
M401
MASTERY

RELIABLE
SYSTEMS
HAPPIER
USERS



A STACK OF FAILURE DOMAINS

Understand the layers that can fail — and how they relate.



Three replicas can still be impacted if they share a zone, or rely on a dependency that's unavailable in that zone.

DEPENDENCIES EXPAND THE BLAST RADIUS

Your service is only as resilient as what it depends on.



Even if the service replicas are healthy, a zone outage can still affect users when dependencies or placement are correlated.

CHOOSE A USER-FACING RELIABILITY OBJECTIVE

Define what success looks like for your users before changing placement.



SERVICE OBJECTIVE

≥ 99.9%

Successful user requests

(for example)

"Design for the experience your users care about."

— ELLY



**M401
MASTERY**
PRACTICE
UNDERSTAND
OPERATE
GO FURTHER

Kubernetes Reliability Week —

Part 1 of 5: Map Failure Domains

Map service failure domains and choose a user-facing reliability objective before placement changes.

28
SEPTEMBER
2026
**M401
MASTERY**

RELIABLE
SYSTEMS
HAPPIER
USERS



SAME
SERVICE.
DIFFERENT
RESILIENCE.
KNOW
WHAT
YOU'RE
GETTING.

FAILURES
HAPPEN.
DESIGN
FOR
THEM.

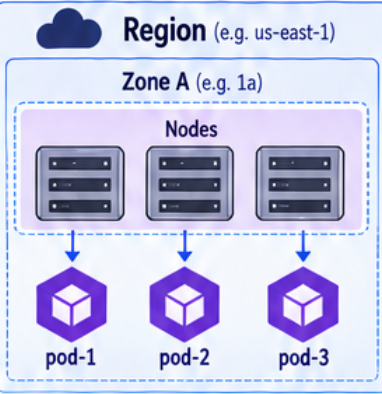
MORE
REPLICAS
SURVIVE
POD FAILURES.
ZONES
REDUCE
CORRELATED
RISK.

REPLICAS VS. ZONE RESILIENCE

Extra replicas and multi-zone placement solve different problems.

3 replicas in one zone

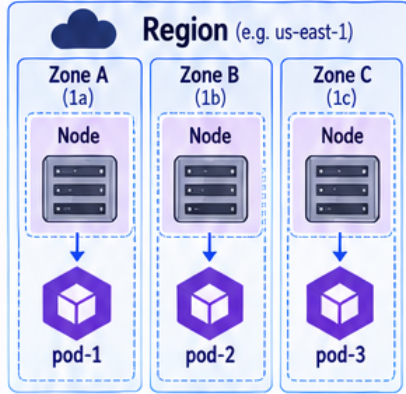
Survives individual pod failures even in a single zone.



- ✓ Can survive individual pod failures.
- ✓ Good for high availability within the zone.
- ! Still at risk if the entire zone fails (e.g. power, network, or cloud event).

One replica per zone

Spreads replicas to reduce correlated zone-failure risk.

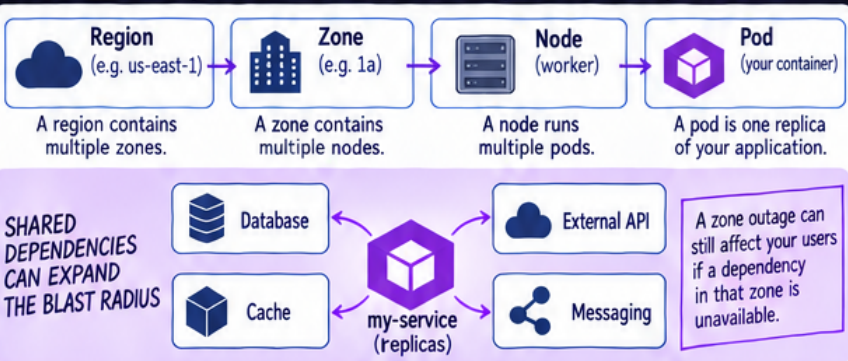


- ✓ Survives pod failures.
- ✓ Reduces risk from a single zone outage.
- ✓ Protects against correlated failures that affect an entire zone.

! Replica count does not prove zone resilience.
You can have 3 replicas and still be unavailable if they are all in the same zone.

KUBERNETES FAILURE DOMAIN HIERARCHY

Understand the layers that can fail — and how your service depends on them.



VERIFY ACTUAL PLACEMENT

Node labels in your cluster show the real topology.

Always verify actual node labels in your cluster. Don't assume where your replicas are running.

KEY TAKEAWAYS

- 1 Extra replicas can survive individual pod failures even in one zone.
- 2 Spreading replicas across zones reduces correlated zone-failure risk.
- 3 Replica count does not prove zone resilience.
- 4 Map dependencies and their failure domains.

KNOW
YOUR
FAILURE
DOMAINS.
MAKE
BETTER
DECISIONS.

M401 MASTERY

PRACTICE
UNDERSTAND
OPERATE
GO FURTHER

Kubernetes Reliability Week — Part 1 of 5: Map Failure Domains

Define user-facing reliability, set a realistic target, and map the failure domains before placement changes.

28
SEPTEMBER
2026

M401
MASTERY

RELIABLE
SYSTEMS
HAPPIER
USERS

REAL
RELIABILITY
STARTS
WITH
THE USER
EXPERIENCE.

Our service
looks resilient...
but wait —
multiple services
share the same
database?

That's a shared
dependency.
A database outage
could still affect
our users.

DIFFERENT
COMPONENTS.
DIFFERENT
FAILURE
DOMAINS.
SAME
USERS.

RELIABILITY
IS A SYSTEM
THINKING
PROBLEM.

1 DEFINE A USER-FACING SLI

Measure the experience your users actually see.



Service Level Indicator (SLI)
Fraction of valid requests served successfully.

$$SLI = \frac{\text{Successful user requests}}{\text{Valid user requests}}$$

IT'S NOT
ABOUT PODS.
IT'S ABOUT
PEOPLE.

- ✓ Count only valid requests (exclude client errors, e.g. 4xx).
- ✓ Measure end-to-end from the user's perspective (e.g. through your ingress or load balancer).
- ✓ Choose a consistent definition across your service.

2 SET AN SLO TARGET AND TIME WINDOW

Set a realistic target and measure the experience, not infrastructure.

Service Level Objective (SLO)
An illustrative example.

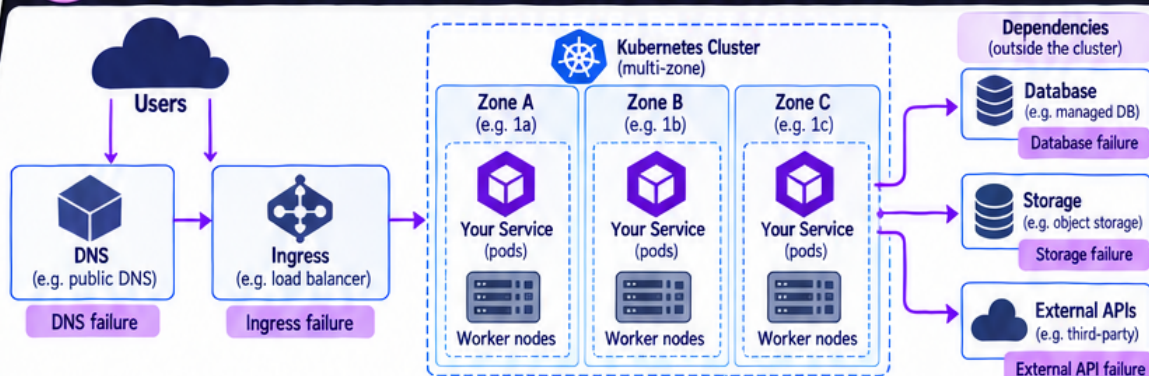
≥ 99.9%
successful user requests
over a 30-day window
(for example)

KEY POINTS

- ✓ Choose a target that fits your product and users.
- ✓ Use a defined time window (e.g. 30 days) to smooth out short-term blips.
- ✓ Measure the user's experience (not how many pods are running).

3 MAP FAILURE DOMAINS FOR WORKLOAD AND DEPENDENCIES

Identify what can fail and how your service depends on other components.



A TOPOLOGY SPREAD POLICY HAS LIMITS

It affects scheduling — it does not guarantee surviving every outage or automatically repair dependencies.

What it does:

- ✓ Helps spread your pods across zones or nodes to reduce correlated failures.
- ✓ Improves availability when a single node or zone has problems.

What it does NOT do:

- ✗ Does not guarantee your service survives every possible outage.
- ✗ Does not make external dependencies (e.g. databases, storage, DNS) highly available.
- ✗ Does not automatically repair failed dependencies.

SPREADING
PODS REDUCES RISK.
IT DOESN'T
REMOVE
DEPENDENCIES.
MAP THEM ALL.

OBSERVE
UNDERSTAND
IMPROVE

M401 MASTERY

PRACTICE
UNDERSTAND
OPERATE
GO FURTHER

Kubernetes Reliability Week — Part 1 of 5: Map Failure Domains

Turn failure domain understanding into a resilient design, and validate it with a controlled drill.

28
SEPTEMBER
2026

M401
MASTERY

RELIABLE
SYSTEMS
HAPPIER
USERS

FROM
A RISK MAP
TO A MORE
RESILIENT
DESIGN.
REAL
RELIABILITY
TAKES
ACTION.

KUBERNETES
RELIABILITY
BUILDS
TRUST

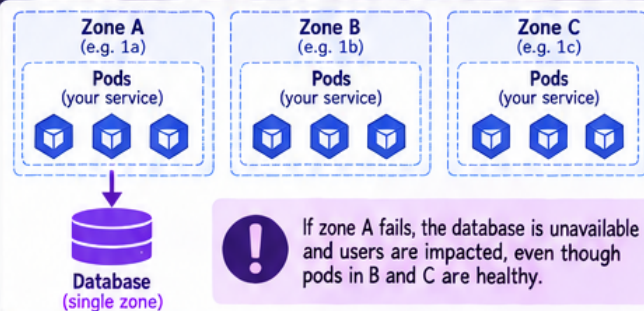
PLAN.
TEST.
LEARN.
STRONGER
SYSTEMS
FOR REAL
USERS.

We know
where it can fail.
Now let's design,
validate and
improve —
step by step.

Resilience
is a habit,
not a hope.

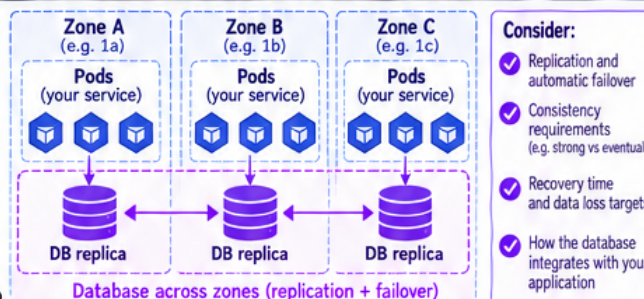
1 PROBLEM A SINGLE ZONE CAN TAKE YOU DOWN

Your service runs in multiple zones, but the database is only in zone A. If zone A has an outage, the database is unavailable and the service can't work, even though pods still run in B and C.



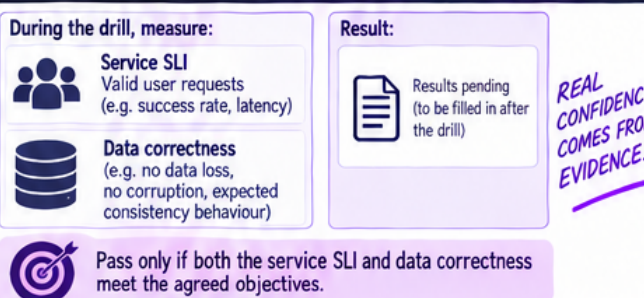
2 ACTION DESIGN FOR CROSS-ZONE AVAILABILITY

Design the database for cross-zone availability and automatic failover, appropriate to your consistency needs.



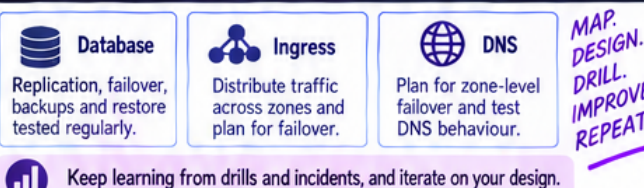
3 CHECK RUN A CONTROLLED DRILL AND MEASURE

In a controlled zone-A failure drill, measure valid user requests and data correctness. Pass only if both meet the agreed objectives.



4 IMPROVE EXPAND THE PLAN

A complete resilience plan includes more than the database.



RESILIENT SYSTEMS
KEEP PEOPLE MOVING FORWARD.

"Reliability is a journey.
built one failure domain at a time."

— ELLY

M401 MASTERY

PRACTICE
UNDERSTAND
OPERATE
GO FURTHER

Kubernetes Reliability Week — Part 1 of 5: Map Failure Domains

From mapping to confidence — key takeaways before you design and place workloads.

28
SEPTEMBER
2026

M401
MASTERY

RELIABLE
SYSTEMS
HAPPIER
USERS

**OBSERVE
UNDERSTAND
DESIGN
TEST
IMPROVE
REAL USERS**

**RESILIENT
SYSTEMS
BUILD
TRUST**

**CURIOSITY
TODAY.
MORE
RESILIENT
TOMORROW.
— ELLY**

Great work!
You've mapped where your service can fail — and how it all connects.
That's the foundation for real resilience.

**PEOPLE
BEHIND
EVERY
REQUEST**

**USERS
SYSTEMS
DEPENDENCIES
RESILIENCE
A BRIGHTER TOMORROW**

**SAME
USERS.
STRONGER
SYSTEMS.**

YOUR RECAP CHECKLIST

Four essentials from Part 1.

- 1 Define a user-facing SLI and SLO window.**
 Start with what success looks like for your users, and set a clear, measurable objective.
USERS DEFINE SUCCESS.
- 2 Map region, zone, node and pod placement.**
 Understand the layers that can fail and how your application is distributed across them.
VISUALISE THE BLAST RADIUS.
- 3 Map shared dependencies and their fault domains.**
 Include databases, caches, external APIs, messaging and other services — and understand how their failures can affect your service.
DEPENDENCIES HAVE DEPENDENCIES.
- 4 Test a controlled failure; measure service and data outcomes before declaring resilience.**
 Validate your design with a controlled test and check both service behaviour and data correctness.
MEASURE. LEARN. IMPROVE.

Elly's Final Tip
 Reliability is measured from the user's path through the whole system.
REAL RELIABILITY LIVES IN THE END-TO-END EXPERIENCE.

You've taken an important step towards more resilient systems. On to the next challenge!

See you in Part 2! ♥

**RELIABLE
SYSTEMS
BRIGHTER
TOMORROWS.
— ELLY**

**Next Up:
Part 2: Design resilient workload placement**

KUBERNETES RELIABILITY WEEK

PART 2 OF 5

DESIGN RESILIENT WORKLOAD PLACEMENT

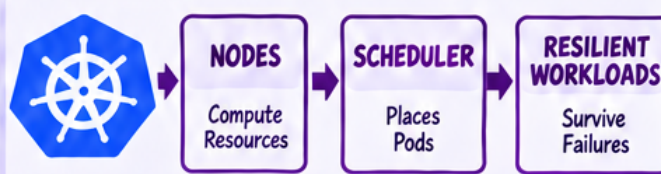


OBJECTIVE

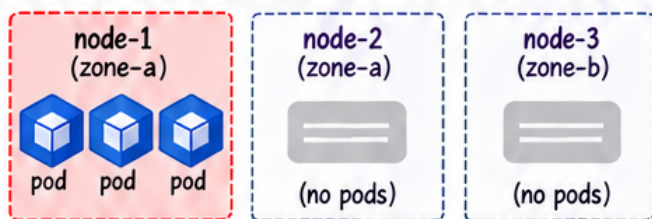


Choose where replicas run so one node failure does not take every replica.

THE BIG PICTURE



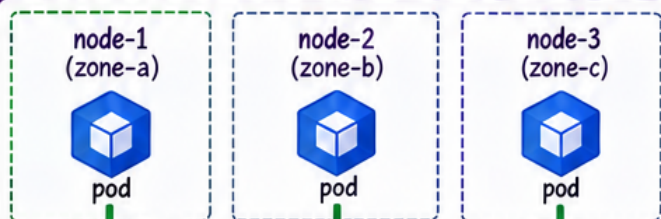
PROBLEM: Three replicas can still share one failure domain.



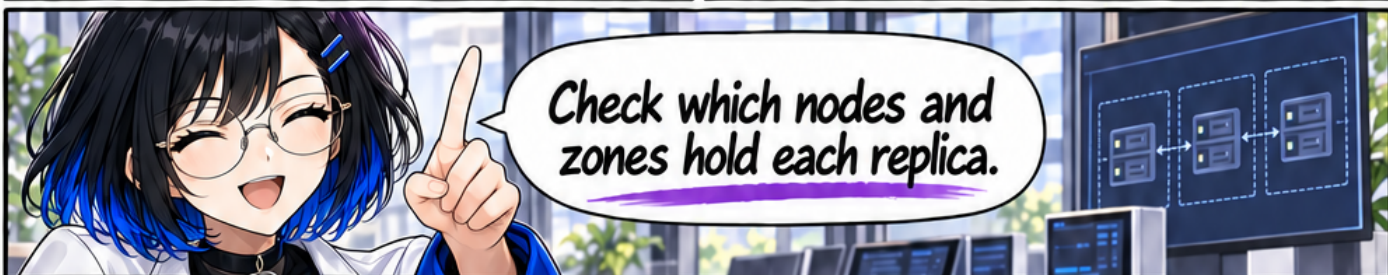
If node-1 fails, all replicas are lost.



BETTER PLACEMENT: Spread replicas across eligible nodes and zones.



A single node failure does not take every replica.



M401 • MASTERY

29 September 2026

PAGE 2 / 5

HOW THE SCHEDULER PLACES PODS

The scheduler filters feasible nodes, then scores them.

Kubernetes Cluster

- 3 Zones
- 6 Nodes
- 9 Pods
- Healthy

Cluster Topology

zone-a	zone-b	zone-c
3 nodes	2 nodes	1 node

NODES SCHEDULER RESILIENCE

1 ELIGIBLE NODES

Resources, selectors, affinity and taints restrict where a Pod can run.

This node doesn't have enough memory.

2 SCORE CANDIDATES

Preferences and topology spread can favour a less crowded zone.

zone-b has fewer Pods, so it scores higher.

3 BIND ONE POD

The scheduler chooses a node for this Pod; future Pods are scheduled separately.

This Pod is scheduled to node-b in zone-b.

Candidate Nodes for Pod

node-a (zone-a) CPU 4 Mem 8Gi Labels: app=web 	node-b (zone-b) CPU 4 Mem 8Gi Labels: app=web 	node-c (zone-c) CPU 4 Mem 2Gi Labels: app=web Insufficient memory
---	---	--

Scoring Eligible Nodes

node-a (zone-a) Current Pods: 3 Less preferred	node-b (zone-b) Current Pods: 1 More preferred
---	---

Pod Scheduled

node-a (zone-a) Pods: 3	node-b (zone-b) Pods: 2	node-c (zone-c) Pods: 0
---	---	---------------------------------------

A desired spread needs enough eligible nodes and capacity.

zone-a 	zone-b 	zone-c
-------------------	-------------------	-------------------

Issue 2 / original page 2 of 5

ELLYPUNK | © 2026 Ellypunk / NeetOps

9 / 27



Kubernetes Cluster

- zone-a Ready
- zone-b Ready
- zone-c Ready

Will this spread keep our workloads safe enough?

SET THE SPREAD RULE

PART 2 OF 5

DESIGN RESILIENT WORKLOAD PLACEMENT



Pod Distribution Across Zones

zone-a	zone-b	zone-c
 2 Pods	 1 Pod	 1 Pod

Healthy Workloads

- zone-a 2 / 2
- zone-b 1 / 1
- zone-c 1 / 1

1 SELECT PODS

Match the workload label so the right replicas are counted.

  
workload=api

2 CHOOSE DOMAIN

topology.kubernetes.io/zone groups eligible nodes by zone.

zone-a
zone-b
zone-c


topology.kubernetes.io/zone

3 SET SKEW

maxSkew: 1 allows at most one Pod count difference when DoNotSchedule applies.

zone-a	zone-b	zone-c
 2	 1	 1

maxSkew: 1
Difference between any two zones ≤ 1

4 CHOOSE FAILURE BEHAVIOUR

DoNotSchedule leaves a Pod Pending if no eligible placement satisfies the rule. ScheduleAnyway still schedules while preferring lower skew.

 Pending

Strict spread can block scheduling when a zone or its capacity is unavailable.

zone-a Ready zone-b Ready zone-c Unavailable



WORKED EXAMPLE: THREE REPLICAS, TWO ZONES



Kubernetes Cluster

- zone-a
- zone-b
- 3 Nodes
- 3 Pods
- Healthy



PROBLEM



All three replicas are on node-a in zone-a. One node outage removes all three.

zone-a

node-a

pod pod pod

zone-b

node-c

(no pods)

ACTION

Add a zone spread constraint for matching workload Pods; keep maxSkew at 1.

zone-a

node-a



zone-b

node-c



CHECK



After replacement Pods are scheduled, inspect the actual placement: zone-a has 2, zone-b has 1.

zone-a

node-a

pod pod

zone-b

node-c

pod

IMPROVEMENT

A zone-a outage leaves the zone-b replica, but capacity and service readiness still need testing.

zone-a

node-a node-b

 (outage)

zone-b

node-c

pod





A spread rule does not move existing Pods by itself.



M401 • MASTERY

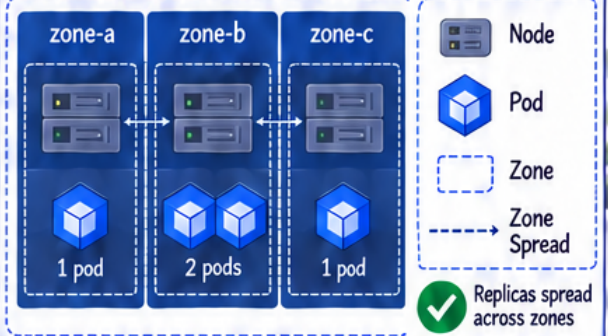
29 September 2026

PAGE 5 / 5

DESIGN RESILIENT WORKLOAD PLACEMENT

KUBERNETES RELIABILITY WEEK PART 2 OF 5

CLUSTER TOPOLOGY (EXAMPLE)



Place replicas so one node failure does not take every replica.

RECAP: WHAT WE BUILT



REPLICAS

Multiple replicas for tolerance.



ELIGIBLE NODES

Nodes that match scheduling rules.



ZONE SPREAD

Distribute across zones and nodes.



VERIFY ACTUAL PLACEMENT

Check real Pod placement in the cluster.

PRODUCTION CHECKLIST



Identify the node and zone failure domains.



Match the intended workload label.



Confirm eligible nodes and capacity in each zone.



Choose DoNotSchedule or ScheduleAnyway deliberately.



Inspect real Pod placement after rollout.



Validate with real placement. The cluster should show replicas across different nodes and zones.

Pod Placement (example)



CAUTION



A PodDisruptionBudget limits supported voluntary evictions; it does not stop a node outage.

ELLY'S FINAL TIP



Test a one-node failure and confirm another ready replica still serves traffic.



ELLYPUNK

© 2026 Ellypunk / NeetOps

KUBERNETES RELIABILITY WEEK

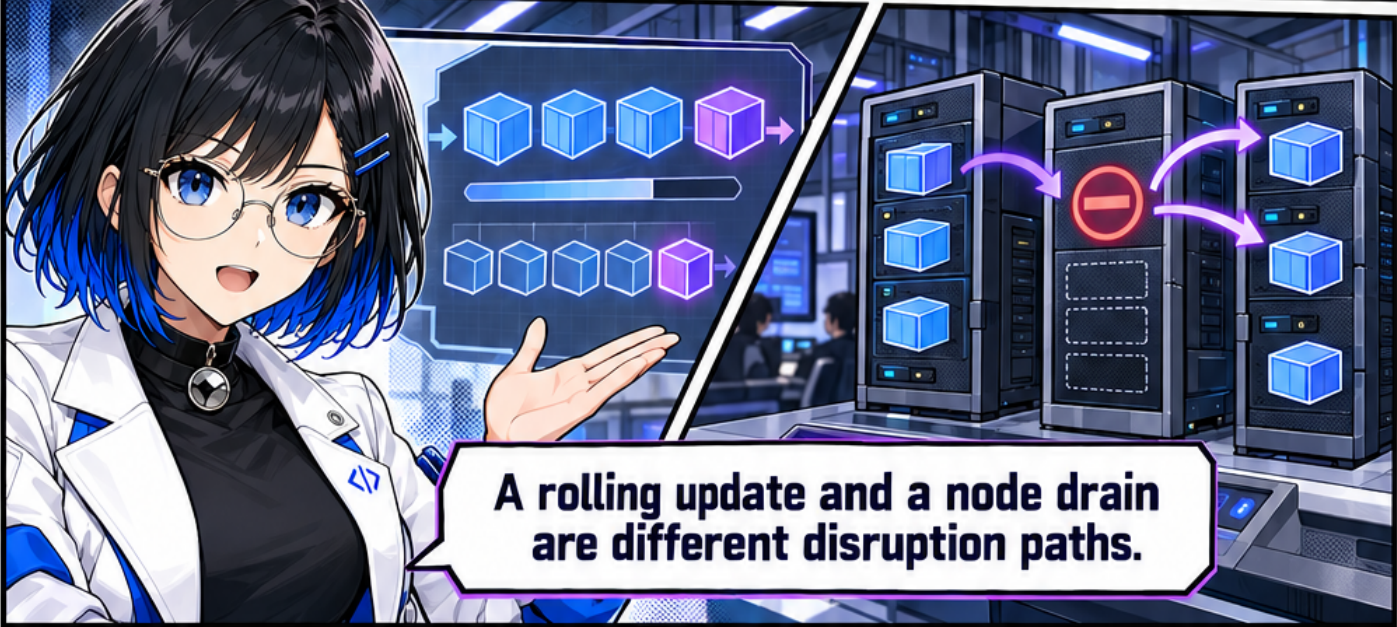


PART 3 OF 5: SAFE ROLLOUT AND DISRUPTION CONTROLS



OBJECTIVE:

Set limits before
a workload
changes.



A rolling update and a node drain
are different disruption paths.

TWO DISRUPTION CONTROLS



Deployment maxUnavailable limits unavailable replicas during a rolling update.

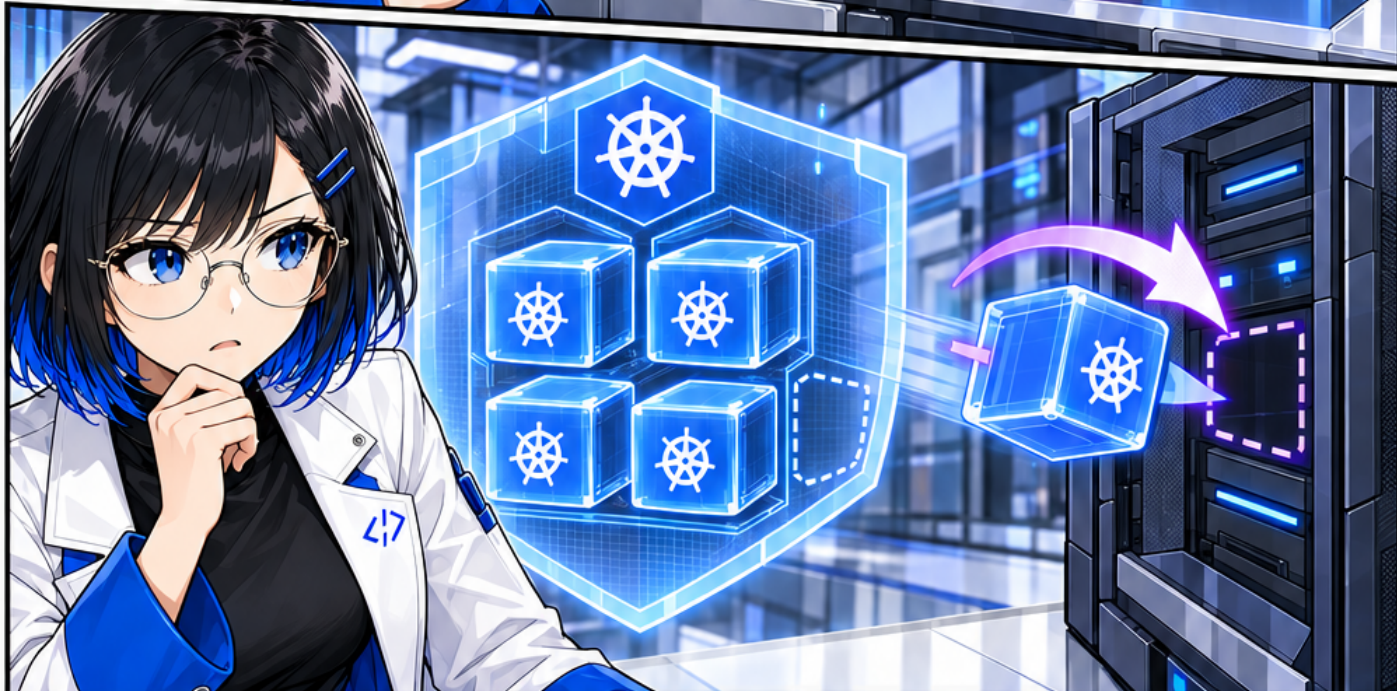
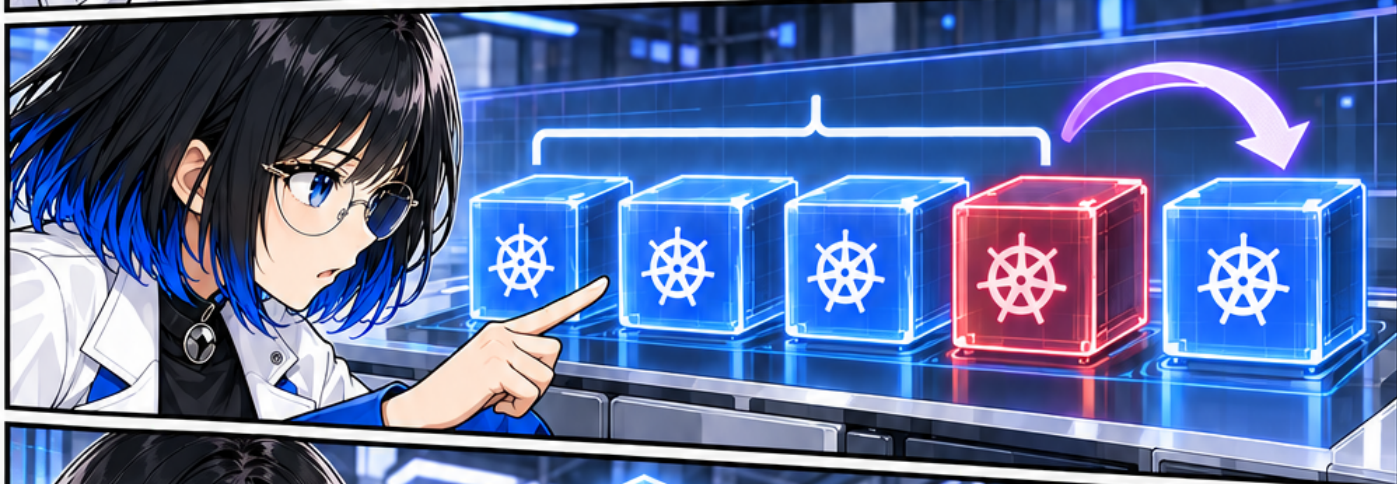
A PDB governs voluntary evictions, including a node drain.

A PDB does not constrain the Deployment rolling update itself.

Set the envelope



For five replicas, a rollout limit of one unavailable pod keeps at least four available under the Deployment's rollout calculation. A PDB with four minimum available can allow one healthy pod eviction during a drain. Confirm selectors match.



KUBERNETES RELIABILITY WEEK

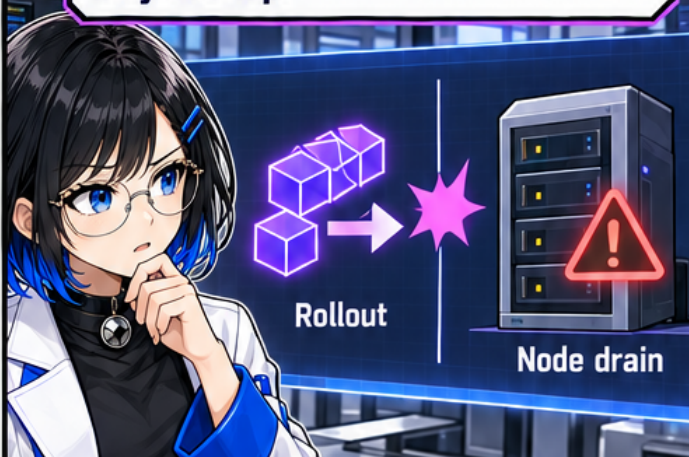
PART 3 OF 5: SAFE ROLLOUT AND DISRUPTION CONTROLS

Protect a five-replica service



1 Problem:

a rollout and a node drain may overlap.



2 Action:

allow at most one unavailable during rollout; require four available for voluntary eviction.



3 Check:

watch ready replicas and PDB allowed disruptions before draining.



4 Improvement:

schedule the drain after rollout stabilises.



The PDB does not constrain the Deployment rolling update.

ROLLOUT AND DRAIN CHECKLIST



1. Set Deployment rollout limits.



2. Check readiness and progress.



3. Set a PDB for voluntary evictions.



4. Drain only when allowed disruptions are sufficient.



A PDB is not a rollout brake; verify Deployment and PDB controls separately.



KUBERNETES RELIABILITY WEEK—

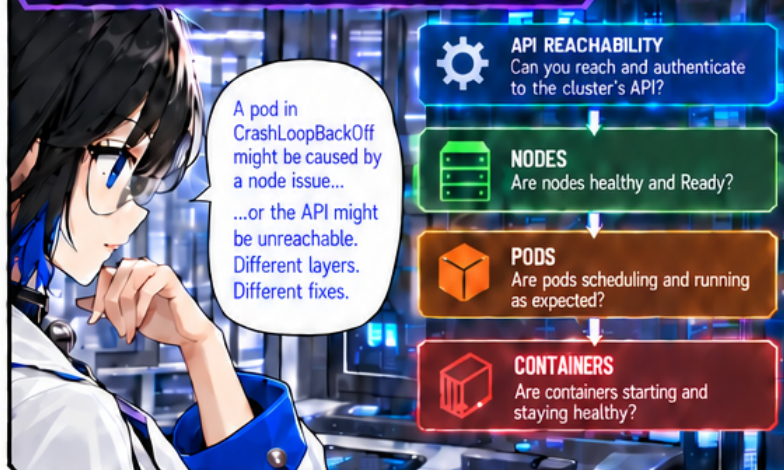
PART 4 OF 5: DIAGNOSE CONTROL-PLANE AND WORKLOAD FAILURES



Objective: Separate cluster access, node health, scheduling and container failures before changing anything.



SAME OUTAGE. FOUR POSSIBLE LAYERS.



DIAGNOSE BEFORE YOU CHANGE.



FIRST: IS THE CONTROL PLANE REACHABLE?

Start with read-only checks to separate access, network and authentication issues before looking at workloads.



Before changing anything, confirm your client can reach and talk to the control plane.

No successful API response means pod listings may be unavailable. Do not infer every Pod failed.

01 Confirm intended kubeconfig context and network path.

Linux / macOS / Windows

```
kubectl config current-context
```

- Verify you are using the expected cluster context.
- Confirm the API server address and network path are reachable from your client (DNS, routing, firewall).
- Use read-only checks; do not change cluster resources at this stage.



02 Check basic API server reachability.

```
kubectl cluster-info
```

Linux / macOS / Windows: same kubectl command; configured client and read permission required.



03 If unreachable, distinguish the failure type before touching workloads.

- Determine whether the issue is name resolution (DNS), TLS (certificate), authentication/authorization, or the API server itself.
- Only after confirming the control-plane is reachable should you proceed to investigate workloads such as Pods.



DECISION TREE: WHAT DOES `kubectl cluster-info` SHOW?



CLIENT CANNOT CONNECT

Network, DNS or TLS issue

```
Unable to connect to the server:
dial tcp: lookup cluster.example.com:
no such host
...
```



FORBIDDEN / AUTH ERROR

Client reaches API, but not authorized

```
Error from server (Forbidden):
user "dev-user" cannot get
path "/" at the cluster scope
...
```



API REPLIES

Control plane is reachable

```
Kubernetes control plane is running at
https://10.0.0.1:6443
CoreDNS is running at
https://10.0.0.10:6443/api/v1/namespaces/
kube-system/services/kube-dns:dns/proxy
...
```

CHECK CLIENT CONNECTIVITY

- DNS resolution for the API server address
- Network reachability (routing, firewall)
- Correct API server address and port
- TLS certificate validity and trust chain



CHECK AUTHENTICATION / RBAC

- Verify client credentials (user, token, certificate or workload identity)
- Confirm kubeconfig context and user
- Check RBAC permissions for cluster-level access



PROCEED TO WORKLOAD CHECKS

- The API server is reachable and responding.
- You can now investigate workloads (e.g. Pods, Deployments, Services) if needed.



IMPORTANT

If there is no successful API response, pod listings may be unavailable. Do not infer that every Pod failed. Resolve the control-plane access issue first.



NEXT: NODE OR SCHEDULING?



Goal: Use read-only kubectl commands to identify whether the issue is a node problem or a scheduling problem. Inspect the right details before you change anything.



You need:

- Installed kubectl
- Correct kubeconfig
- RBAC read permissions (Linux / macOS / Windows)



Check cluster state first. Are the nodes Ready, or are pods failing to schedule?

We'll decide using observability, not assumptions.



NODES	
node-1	Ready
node-2	NotReady
node-3	Ready

PODS	
web	Running
api	Pending
worker	Running

Determine if it's a node issue or a scheduling issue. Don't assume the application code is broken.

1. CHECK NODES

Get the current node status.

Linux / macOS / Windows
(requires kubectl, correct kubeconfig, read permissions)

```
kubectl get nodes
```

NAME	STATUS	ROLES	AGE	VERSION
node-1	Ready	worker	12d	v1.31.0
node-2	NotReady	worker	12d	v1.31.0
node-3	Ready	worker	12d	v1.31.0
...				



If any node shows NotReady, inspect the node conditions and check for resource pressure or taints.



NODE NOTREADY?



Inspect node conditions and pressure/taints.

Linux / macOS / Windows
(requires kubectl, correct kubeconfig, read permissions)

```
kubectl describe node <node-name>
```

Look for:

- Conditions (Ready, DiskPressure, MemoryPressure, PIDPressure, etc.)
- Taints that may prevent scheduling
- Node capacity and allocatable resources
- Recent events on the node

2. CHECK PODS

List all pods across namespaces.

Linux / macOS / Windows
(requires kubectl, correct kubeconfig, read permissions)

```
kubectl get pods -A
```

NAMESPACE	NAME	READY	STATUS	RESTARTS	AGE
default	web-5c7d9	1/1	Running	0	3h
kube-system	coredns-7b	1/1	Running	0	2d
app	api-6f9d8	0/1	Pending	0	12m
...					



If pods are Pending, inspect the scheduling events and the requested resources. Don't assume the application code is broken.



POD PENDING?



Inspect scheduling events and requested resources.

Linux / macOS / Windows
(requires kubectl, correct kubeconfig, read permissions)

```
kubectl describe pod <pod-name> -n <namespace>
```

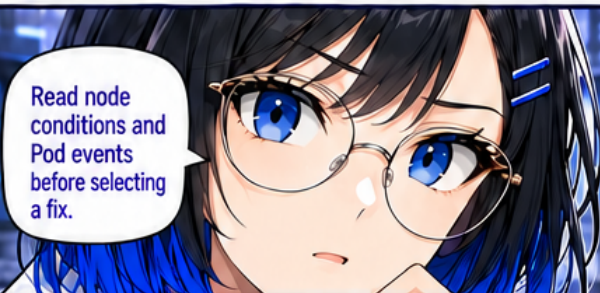
Look for:

- Events that explain why scheduling failed (e.g. insufficient resources, node selectors, taints, affinity/anti-affinity)
- Requested CPU and memory resources
- Namespace-level constraints (resource quotas, limit ranges, etc.)

READ-ONLY DIAGNOSTIC APPROACH

- ✓ Use read-only commands to gather evidence.
- ✓ Check nodes first, then pods.
- ✓ Inspect the relevant details before changing anything.
- ✓ Do not perform any destructive action.

Read node conditions and Pod events before selecting a fix.



A PENDING POD IS A CLUE

A Pending pod can mean the scheduler cannot find a suitable node (for example, not enough CPU), not that the control-plane or API is broken.



1 API reachable; nodes Ready.

First, confirm the control-plane is healthy. The API is reachable and the nodes are Ready. So the cluster itself looks good.

Control Plane

kube-apiserver

etcd

controller-manager

scheduler

Healthy

Healthy

Healthy

Healthy

Nodes

node-1

node-2

node-3

Ready

Ready

Ready

\$ kubectl get nodes

NAME	STATUS	ROLES	AGE
node-1	Ready	<none>	5d
node-2	Ready	<none>	5d
node-3	Ready	<none>	5d

2 One workload stays Pending.

Now look at the workloads. Most pods are Running, but one pod is Pending. This Pending pod needs closer inspection.

Pods (namespace: app)

NAME	STATUS	RESTARTS	AGE
web-7d6f8	Running	0	2m
api-6c9df	Running	0	2m
worker-0	Pending	0	2m
db-5f4b9	Running	0	2m
cache-7c8d	Running	0	2m

\$ kubectl get pods -n app

NAME	STATUS	RESTARTS	AGE
web-7d6f8	Running	0	2m
api-6c9df	Running	0	2m
worker-0	Pending	0	2m
db-5f4b9	Running	0	2m
cache-7c8d	Running	0	2m

3 Describe its scheduling event.

Describe the Pending pod to see the scheduler's reason. The event shows:

FailedScheduling: Insufficient cpu

This means there isn't enough CPU available on any node to fit this pod.

Events (worker-0)

TYPE	REASON	MESSAGE
Warning	FailedScheduling	FailedScheduling: Insufficient cpu

\$ kubectl describe pod worker-0 -n app

Events:

Type	Reason	Age	From	Message
Warning	FailedScheduling	37s	default-scheduler	FailedScheduling: Insufficient cpu

4 Compare CPU requests with node allocatable capacity. Review impact before changing requests or capacity.

Compare the pod's CPU request with the nodes' allocatable capacity. If the request is larger than what's available on any node, the pod will stay Pending. Before changing requests or adding capacity, review the impact on other workloads.

Pod CPU Request (worker-0)

CPU request 2000m (2 CPU)

Node Allocatable CPU

node-1	1500m (1.5 CPU)
node-2	1500m (1.5 CPU)
node-3	1000m (1 CPU)

The pod requests 2000m CPU, but the largest node allocatable is 1500m. No node can fit it, so it remains Pending.



Trace the Failure Boundary



Find the first layer that fails, check its evidence, and move to the next safe check.
Follow the path from client to running workload.

1 API connection fails?

Check context, credentials, and control-plane access.



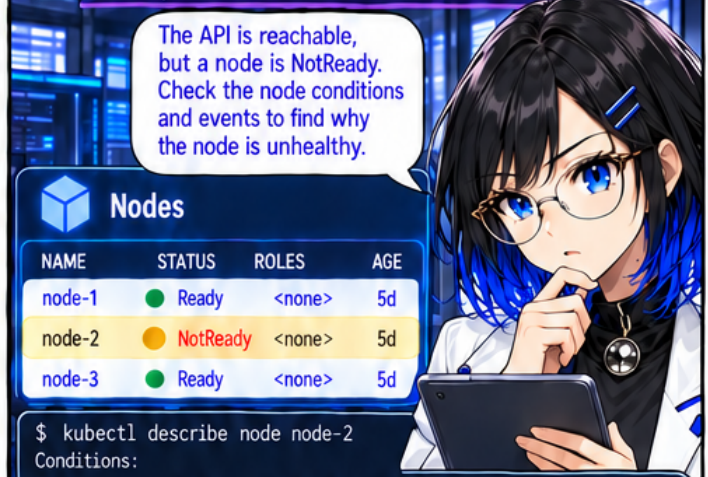
If you cannot reach the API, verify your kubeconfig context, credentials, and that the control-plane API is accessible.



```
$ kubectl cluster-info
Unable to connect to the server: connection timed out
```

2 API responds; node NotReady?

Inspect node conditions and events.



The API is reachable, but a node is NotReady. Check the node conditions and events to find why the node is unhealthy.

Nodes

NAME	STATUS	ROLES	AGE
node-1	Ready	<none>	5d
node-2	NotReady	<none>	5d
node-3	Ready	<none>	5d

```
$ kubectl describe node node-2
Conditions:
  Type      Status  LastHeartbeatTime  Reason
  Ready     False   2m                 KubeletNotReady
  MemoryPressure  False   2m                 KubeletHasSufficientMemory
  DiskPressure  False   2m                 KubeletHasNoDiskPressure
  ...
```

3 Nodes Ready; Pod Pending?

Inspect scheduling events and available capacity.



Nodes are Ready, but the pod is Pending. Check the pod's scheduling events and whether there is enough allocatable capacity for the request.

Pods (namespace: app)

NAME	STATUS	NODE	AGE
web-7d6f8	Running	node-1	2m
api-6c9df	Running	node-1	2m
worker-0	Pending	<none>	2m
db-5f4b9	Running	node-3	2m

```
$ kubectl describe pod worker-0 -n app
Events:
  Type      Reason      Age   From          Message
  Warning   FailedScheduling  37s   default-scheduler  0/3 nodes are available: insufficient cpu.
```

4 Pod Running; service fails?

Inspect container logs and readiness.



The pod is Running, but the service still fails. Check the container logs and readiness to find the reason.

Containers (pod/web-7d6f8)

NAME	READY	RESTARTS	AGE
web	0/1	0	2m
sidecar	1/1	0	2m

Logs (web)

```
$ kubectl logs web-7d6f8 -n app
2026-10-01T08:12:31Z Starting server...
2026-10-01T08:12:32Z ERROR: dependency unavailable
2026-10-01T08:12:33Z Retrying in 10s...
```

Readiness Probe

```
Type: HTTP GET
Path: /ready
Status: Failure
Last probe: 2s ago
```



Record the first failing layer, its evidence, and the next safe check.



Client



API Server



Node



Pod



Application



KUBERNETES RELIABILITY WEEK – PART 5 OF 5

RUN PRODUCTION RECOVERY AND CAPACITY DRILLS

Objective: Practice a bounded node-loss drill; measure recovery and spare capacity.



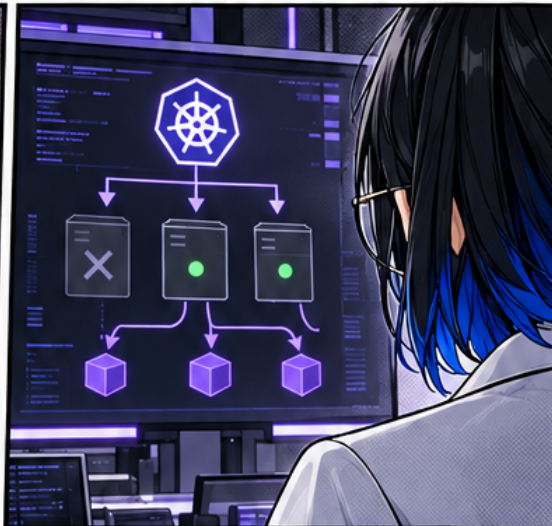
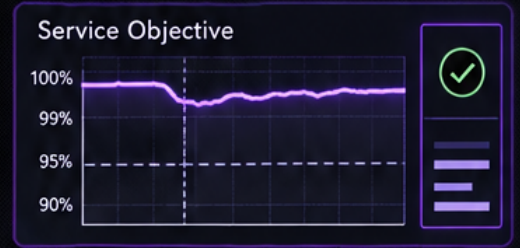
1 NODE FAILURE



2 WORKLOADS RESCHEDULE



3 WATCH SERVICE OBJECTIVE



M401 · MASTERY

2 October 2026

PAGE 2

BOUND THE DRILL



1

Choose one noncritical node.

Production traffic continues

```
graph LR; A[Node A (Healthy)] --> B[Node B (Drill Target)]; B --> C[Node C (Healthy)];
```

2

Record baseline latency and error rate.

Latency (p95)

Time	Latency (p95)
10:00	200
10:15	220
10:30	180
10:45	210

Error Rate

Time	Error Rate
10:00	0.5%
10:15	0.5%
10:30	0.5%
10:45	0.5%

3

Check disruption budget and spare capacity.

Disruption Budget

Allowed Unavailable	1
Currently Unavailable	0
Remaining	1

Spare Capacity

CPU Headroom	28%
Memory Headroom	32%
Pod Headroom	25%

4

Set abort criteria and rollback owner.

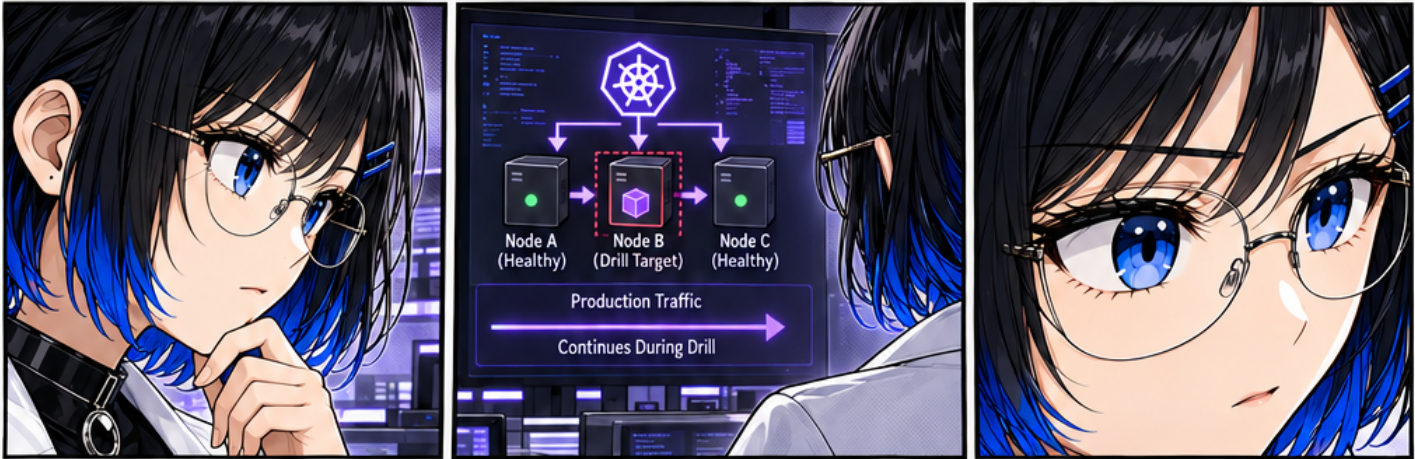
Abort Criteria

- Error rate > 5%
- Latency p95 > 3× baseline
- User-facing errors detected
- SLO at risk

Rollback Owner

On-call Engineer
Primary: Alex Chen
Backup: Priya Nair

Do not start if the workload lacks headroom.



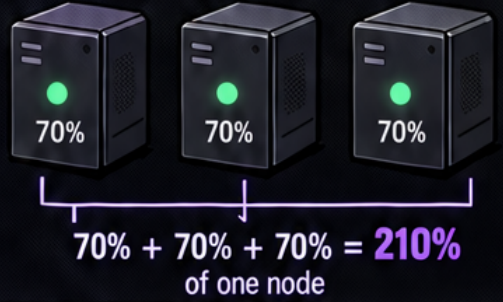
TEST CAPACITY BEFORE FAILURE

Understand whether your remaining nodes can handle the load after a failure.



1 TOTAL LOAD BEFORE FAILURE

3 equal nodes × 70% load
= 210% of one node.



2 AFTER ONE NODE LOSS

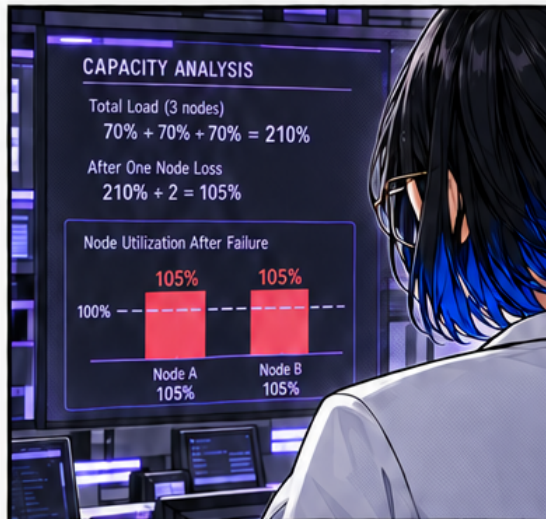
210% ÷ 2 = 105% per survivor.



3 CONCLUSION

! The remaining pair cannot fit this load; add headroom before the drill.

Caveat Real scheduling also depends on pod requests and placement rules.



RUN AND OBSERVE THE DRILL



1 Isolate the chosen node under change control.



2 Watch pod readiness, p95 latency and errors.



3 Compare recovery time with objective.



4 Abort and restore if thresholds are crossed.



Do not treat a successful eviction as service recovery.

RECOVERY DRILL CLOSEOUT



1 RECORD BASELINE AND ABORT THRESHOLD.

Baseline Metrics		Abort Threshold	
CPU	42%	Errors	> 5%
Memory	58%	Latency	> 500 ms
Latency	28 ms	RPS	< 50%
Errors	0.1%	Availability	< 99%
RPS	1,250		

2 CONFIRM SPARE NODE CAPACITY AND DISRUPTION BUDGET.



3 MEASURE READINESS, LATENCY, ERRORS AND RECOVERY TIME.

Readiness
100%

Latency
32 ms

Errors
0.1%

Recovery Time
78 s

4 RESTORE THE NODE AND VERIFY SERVICE HEALTH.



! CONCLUSION

The 3 x 70% example lacks one-node headroom; do not run that drill unchanged.

TIP

Compare the recovery result with the objective before calling the drill successful.