Designing for HA and resiliency in Amazon EKS applications

# AWS Prescriptive Guidance

# AWS Prescriptive Guidance: Designing for HA and resiliency in Amazon EKS applications



Services or capabilities described in AWS documentation might vary by Region. To see the differences applicable to the AWS European Sovereign Cloud Region, see the AWS European Sovereign Cloud User Guide.

# Table of Contents

# Designing for high availability and resiliency in Amazon EKS applications

*Haofei Feng, Frank Fan, and Rus Kalakutskiy, Amazon Web Services (AWS)*

*October 2025* (document history)

Ensuring high availability (HA) and resiliency in application design is crucial for achieving near-zero recovery point objective (RPO) and recovery time objective (RTO). As organizations increasingly migrate and modernize their applications to Kubernetes environments, the demand for robust and scalable solutions continues to increase. Amazon Elastic Kubernetes Service (Amazon EKS) helps you to efficiently manage containerized applications at scale.

This guide delves into a set of widely recognized recommendations and best practices for designing and managing Amazon EKS microservice applications. Based on extensive experience and real-world deployments, these insights offer valuable guidance for architects and developers. Implement these recommendations for high performance, reliability, and scalability of your Kubernetes-based applications to achieve robust operations.

# High availability and resilience design considerations

The shared responsibility model becomes more complex with Kubernetes. Amazon EKS control plane availability and resilience are managed by Amazon Web Services (AWS). Your organization manages the data plane, which can significantly affect the performance and availability of your microservices applications.

When designing a highly available and resilient application on Amazon EKS, consider the following components:

- The microservices application: its pods and containers
- The workload data plane: Ingress Controller, pod, system components such as the Amazon Virtual Private Cloud (Amazon VPC) Container Network Interface (CNI), service mesh sidecars, and kube-proxy
- The workload-mangement layer: controllers, admission controllers, network policy engines, and persistent data storage for these components
- The Kubernetes control plane
- Infrastructure: nodes, network, and network appliances

For the first three considerations, which refer to components that run within a Kubernetes cluster, this guide covers the following topics:

- Spreading workloads across nodes and Availability Zones
- Protecting critical workloads with a PDB
- Configuring probes and health checks
- Configuring container lifecycle hooks
- Understanding pod eviction during zonal disruptions

## Spread workloads across nodes and Availability Zones

Distributing a workload across failure domains such as Availability Zones and nodes improves component availability and decreases the chances of failure for horizontally scalable applications. The following sections introduce ways to spread workloads across nodes and Availability Zones.

## Use pod topology spread constraints

Kubernetes pod topology spread constraints instruct the Kubernetes scheduler to distribute pods that are managed by `ReplicaSet` or `StatefulSet` across different failure domains (Availability Zones, nodes, and types of hardware). When you use pod topology spread constraints, you can do the following:

- Distribute or concentrate pods across different failure domains depending on application requirements. For example, you can distribute pods for resilience, and you can concentrate pods for network performance.
- Combine different conditions, such as distributing across Availability Zones and distributing across nodes.
- Specify the preferred action if conditions can't be met:
  - Use `whenUnsatisfiable: DoNotSchedule` with a combination of `maxSkew` and `minDomains` to create hard requirements for the scheduler.
  - Use `whenUnsatisfiable: ScheduleAnyway` to reduce `maxSkew`.

If a failure zone becomes unavailable, the pods in that zone become unhealthy. Kubernetes reschedules the pods while adhering to the spread constraint if possible.

The following code shows an example of using pod topology spread constraints across Availability Zones or across nodes:

```
...
spec:
  selector:
    matchLabels:
      app: <your-app-label>
    replicas: 3
    template:
      metadata:
        labels: <your-app-label>
      spec:
        serviceAccountName: <ServiceAccountName>
...
        topologySpreadConstraints:
        - labelSelector:
            matchLabels:
              app: <your-app-label>
```

```
          maxSkew: 1
          topologyKey: topology.kubernetes.io/zone # <---spread those pods evenly over
 all availability zones
          whenUnsatisfiable: ScheduleAnyway
        - labelSelector:
            matchLabels:
              app: <your-app-label>
          maxSkew: 1
          topologyKey: kubernetes.io/hostname # <---spread those pods evenly over all
 nodes
          whenUnsatisfiable: ScheduleAnyway
```

## Default cluster-wide topology spread constraints

By default, Kubernetes provides a set of topology spread constraints for distributing pods across nodes and Availability Zones:

```
defaultConstraints:
  - maxSkew: 3
    topologyKey: "kubernetes.io/hostname"
    whenUnsatisfiable: ScheduleAnyway
  - maxSkew: 5
    topologyKey: "topology.kubernetes.io/zone"
    whenUnsatisfiable: ScheduleAnyway
```

> **Note**
>
> Applications that need different types of topology constraints can override the cluster-level policy.

The default constraints set a high `maxSkew`, which isn't useful for deployments that have a small number of pods. As of now, `KubeSchedulerConfiguration` can't be changed in Amazon EKS. If you need to enforce other sets of topology spread constraints, consider using mutating admission controller like in the section below. You can also control default topology spread constraints if you run an alternative scheduler. However, managing custom schedulers adds complexity and can have implications on cluster resilience and HA. For these reasons, we don't recommend using an alternative scheduler for topology spread constraints only.

## The Gatekeeper policy for topology spread constraints

Another option for enforcing topology spread constraints is to use a policy from the Gatekeeper project. Gatekeeper policies are defined at the application level.

The following code examples show the use of a `Gatekeeper OPA` policy for deployment. You can modify the policy for your needs. For example, apply the policy only to deployments that have the label `HA=true`, or write a similar policy using a different policy controller.

This first example shows `ConstraintTemplate` used with `k8stopologyspreadrequired_template.yml`:

```
apiVersion: templates.gatekeeper.sh/v1
kind: ConstraintTemplate
metadata:
  name: k8stopologyspreadrequired
spec:
  crd:
    spec:
      names:
        kind: K8sTopologySpreadRequired
      validation:
        openAPIV3Schema:
          type: object
          properties:
            message:
              type: string
  targets:
    - target: admission.k8s.gatekeeper.sh
      rego: |
        package k8stopologyspreadrequired

        get_message(parameters, _default) =3D msg {
          not parameters.message
          msg :=_default
        }


        get_message(parameters, _default) =3D msg {
          msg := parameters.message
        }

        violation[{"msg": msg}] {
```

```
          input.review.kind.kind ="Deployment"
          not input.review.object.spec.template.spec.topologySpreadConstraint
          def_msg :"Pod Topology Spread Constraints are required for Deployments"
          msg :get_message(input.parameters, def_msg)

        }
```

The following code shows the `constraints` YAML manifest `k8stopologyspreadrequired_constraint.yml`:

```
apiVersion: constraints.gatekeeper.sh/v1beta1
kind: K8sTopologySpreadRequired
metadata:
  name: require-topologyspread-for-deployments
spec:
  match:
    kinds:
      - apiGroups: ["apps"]
        kinds: ["Deployment"]
    namespaces:  ## Without theses two lines will apply to the whole cluster
      - "example"
```

## When to use topology spread constraints

Consider using topology spread constraints for the following scenarios:

- Any horizontally scalable application (for example, stateless web services)
- Applications with active-active or active-passive replicas (for example, NoSQL databases or caches)
- Applications with stand-by replicas (for example, controllers)

System components that can be used for the horizontally scalable scenario, for example, include the following:

- Cluster Autoscaler and Karpenter (with `replicaCount > 1` and `leader-elect = true`)
- AWS Load Balancer Controller
- CoreDNS

# Pod affinity and anti-affinity

In some cases, it's beneficial to ensure that no more than one pod of a specific type is running on a node. For example, to avoid scheduling multiple network-heavy pods on the same node, you can use the anti-affinity rule with the label `Ingress` or `Network-heavy`. When you use `anti-affinity`, you can also use a combination of the following:

- Taints on network-optimized nodes
- Corresponding tolerations on network-heavy pods
- Node affinity or node selector to ensure that network-heavy pods use network-optimized instances

Network-heavy pods are used as an example. You might have different requirements, such as GPU, memory, or local storage. For other usage examples and configuration options, see the Kubernetes documentation.

## Rebalance pods

This section discusses two approaches to rebalancing pods in a Kubernetes cluster. The first uses the Descheduler for Kubernetes. The Descheduler helps to maintain pod distribution by enforcing strategies to remove pods that violate topology spread constraints or anti-affinity rules. The second approach uses the Karpenter consolidation and bin-packing feature. Consolidation continuously evaluates and optimizes resource usage by consolidating workloads onto fewer, more efficiently packed nodes.

We recommend using Descheduler if you aren't using Karpenter. If you're using Karpenter and Cluster Autoscaler together, you can use Descheduler with Cluster Autoscaler for node groups.

### Descheduler for groupless nodes

There's no guarantee that the topology constraints remain satisfied when pods are removed. For example, scaling down a deployment might result in imbalanced pod distribution. However, because Kubernetes uses pod topology spread constraints only at the scheduling stage, pods are left unbalanced across the failure domain.

To maintain a balanced pod distribution in such scenarios, you can use Descheduler for Kubernetes. Descheduler is a useful tool for multiple purposes, such as to enforce the maximum pod age or time to live (TTL), or to improve the use of infrastructure. In the context of resilience and high availability (HA), consider the following Descheduler strategies:

- RemovePodsViolatingTopologySpreadConstraint
- RemovePodsViolatingInterPodAntiAffinity
- RemoveDuplicates

### Karpenter consolidation and bin-packing feature

For workloads that use Karpenter, you can use the consolidation and bin-packing functionality to optimize resource utilization and reduce costs in Kubernetes clusters. Karpenter continuously evaluates pod placements and node utilization, and it attempts to consolidate workloads onto fewer, more efficiently packed nodes when possible. This process involves analyzing resource requirements, considering constraints such as pod affinity rules, and potentially moving pods between nodes to improve overall cluster efficiency. The following code provides an example:

```
apiVersion: karpenter.sh/v1beta1
kind: NodePool
metadata:
  name: default
spec:
  disruption:
    consolidationPolicy: WhenUnderutilized
    expireAfter: 720h
```

For `consolidationPolicy`, you can use `WhenUnderutilized` or `WhenEmpty`:

- When `consolidationPolicy` is set to `WhenUnderutilized`, Karpenter considers all nodes for consolidation. When Karpenter discovers a node that's empty or underused, Karpenter attempts to remove or replace the node to reduce cost.
- When `consolidationPolicy` is set to `WhenEmpty`, Karpenter considers for consolidation only nodes that contain no workload pods.

The Karpenter consolidation decisions are not based solely on CPU or memory utilization percentages that you might see in monitoring tools. Instead, Karpenter uses a more complex algorithm based on pod resource requests and potential cost optimizations. For more information, see the Karpenter documentation.

# Protect critical workloads with a PDB

A pod disruption budget (PDB) is an essential feature for maintaining the high availability of applications in a cluster. The PDB specifies a target size, which is the minimum availability for a particular type of pod. This means that a minimum number of replicas of a particular pod type must be running at any given time. If the number of running replicas falls below the target size, Kubernetes prevents further disruptions to the remaining replicas until the target size is met. PDBs help to ensure that workloads are not affected by these events and can continue to run uninterrupted. When a disruption occurs, Kubernetes attempts to gracefully evict pods from the affected nodes while maintaining the number of replicas specified in the PDB.

You can use a PDB to declare the `minAvailable` and `maxUnavailable` number of replicas. For example, if you want at least three copies of your app to be available, create a PDB that is similar to the following example:

```
apiVersion: policy/v1beta1
kind: PodDisruptionBudget
metadata:
  name: my-svc-pdb
spec:
  minAvailable: 3
  selector:
    matchLabels:
      app: my-svc
```

Setting up PDBs correctly for your applications helps to minimize the disruption during planned or unplanned events. You can use the anti-affinity rule to schedule a deployment's pods on different nodes and avoid PDB delays during node upgrades.

# Configure probes and load balancer health checks

Kubernetes provides several ways to perform application health checks in addition to load balancer health checks. You can run the following Kubernetes built-in probes together with the load balancer health check as a command in the pod's context or as an HTTP/TCP probe to kubelet or the host IP address.

Liveness probes and readiness probes should be different and independent (or at least with different timeout values). If an application has a temporary issue, the readiness probe will mark the

pod as unready until the issue is resolved. If the liveness probe settings aren't correct, the liveness probe might terminate the pod.

## Startup probe

Use startup probes to protect applications that have long initialization cycles. Until the startup probe succeeds, the other probes are disabled.

You can define a maximum time that Kubernetes should wait for application startup. If, after the maximum configured time, the pod still fails the startup probes, the application is terminated, and a new pod is created.

Use startup probes when the startup time of an application is unpredictable. If you know that your application needs 10 seconds to start, use a liveness probe or a readiness probe with `initialDelaySeconds` instead.

## Liveness probe

Use liveness probes to detect application issues or whether the process is running without issues. A liveness probe can detect deadlock conditions where the process continues to run but the application becomes unresponsive. When using a liveness probe, do the following:

- Use `initialDelaySeconds` to delay the first probe.
- Don't set the same specification for liveness and readiness probes.
- Don't configure a liveness probe to depend on a factor that is external to your pod (for example, a database).
- Set the liveness probe for a specific `terminationGracePeriodSeconds`. For more information, see the Kubernetes documentation.

## Readiness probe

Use a readiness probe to detect the following:

- Whether the application is ready to accept traffic
- Partial availability, where the application might be temporarily unavailable but is expected to be healthy again after a certain operation completes

Readiness probes help to ensure that the application configuration and dependencies are running without issues or errors, so that the application can serve traffic. However, a poorly configured readiness probe can cause an outage instead of preventing it. Readiness probes that depend on external factors, such as connectivity to a database, can cause all pods to fail the probe. Such failures can result in an outage, and they can lead to a cascading failure from a backend service to other services that used the failed pods.

### Ingress resource and load balancer health checks

Application Load Balancer and Kubernetes `ingress` provide health-check features. For the Application Load Balancer health checks, specify the target ports and path.

> **Note**
>
> For the Kubernetes `ingress`, there will be a deregistration latency. The default for Application Load Balancer is 300 seconds. Consider setting up your ingress resource or load balancer health check by using the same values that you used for your readiness probe.

NGINX also provides a health check. For more information, see the NGINX documentation.

Istio ingress and egress gateways don't have a health check mechanism that's comparable to the HTTP health check from NGINX. However, you can achieve similar functionality by using the Istio circuit breaker or `DestinationRule` outlier detection.

For more information, see Availability and Pod Lifecycle in the *Amazon EKS Best Practices Guide*.

## Configure container lifecycle hooks

During a graceful container shutdown, your application should respond to a `SIGTERM` signal by starting its shutdown so that clients don't experience any downtime. Your application should run cleanup procedures such as the following:

- Saving data
- Closing file descriptors
- Closing database connections
- Completing in-flight requests gracefully
- Exiting in a timely manner to fulfill the pod termination request

Set a grace period that is long enough for cleanup to finish. To learn how to respond to the `SIGTERM` signal, see the documentation for the programming language that you use for your application.

Container lifecycle hooks enable containers to be aware of events in their management lifecycle. Containers can run code implemented in a handler when the corresponding lifecycle hook is executed. Container lifecycle hooks provide a workaround for the asynchronous nature of Kubernetes and the cloud. This approach can prevent the loss of connections that are forwarded to the terminating pod before the ingress resource and `iptables` are updated to not send new traffic to the pod.

Container lifecycle, `Endpoint`, and `EndpointSlice` are part of different APIs. It's important to orchestrate these APIs. However, when a pod is being terminated, the Kubernetes API simultaneously notifies both the kubelet (for container lifecycle) and the `EndpointSlice` controller. For more information, including a diagram, see Gracefully handle the client requests in the *Amazon EKS Best Practices Guide*.

When `kubelet` sends `SIGTERM` to the pod, the `EndpointSlice` controller is terminating the `EndpointSlice` object. That termination notifies the Kubernetes API servers to notify the `kube-proxy` of each node to update `iptables`. Although these actions occur at the same time, there are no dependencies or sequences between them. There is a high chance that the container receives the `SIGKILL` signal much earlier than the `kube-proxy` on each node updates the local `iptables` rules. In that case, possible scenarios include the following:

- If your application immediately and bluntly drops the in-flight requests and connections upon receipt of `SIGTERM`, the clients see `500` errors.
- If your application ensures that all in-flight requests and connections are processed completely upon receipt of `SIGTERM`, during the grace period, new client requests would still be sent to the application container because `iptables` rules might not be updated yet. Until the cleanup procedure closes the server socket on the container, those new requests will result in new connections. When the grace period ends, the new connections that were established after the `SIGTERM` was sent are dropped unconditionally.

To address the previous scenarios, you can implement in-app integration or the PreStop lifecycle hook. For more information, including a diagram, see Gracefully shutdown applications in the *Amazon EKS Best Practices Guide*.

> **Note**
>
> Regardless of whether the application shuts down gracefully, or the result of the `preStop` hook, the application containers are eventually terminated at the end of the grace period through `SIGKILL`.

Use the `preStop` hook with a `sleep` command to delay sending `SIGTERM`. This will help to continue accepting the new connections while the ingress object routes them to the pod. Test the time value of the `sleep` command to ensure that any latency of Kubernetes and other application dependencies are taken into account, as shown in the following example:

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: nginx
spec:
  containers:
    - name: nginx
      lifecycle:
        # This "sleep" preStop hook delays the Pod shutdown until
        # after the Ingress Controller removes the matching Endpoint or EndpointSlice
        preStop:
          exec:
            command:
              - /bin/sleep
              - "20"
              # This period should be turned to Ingress/Service Mesh update latency
```

For more information, see Container hooks in the Kubernetes documentation and Gracefully shutdown applications in the *Amazon EKS Best Practices Guide*.

## Understand pod eviction during zonal disruptions

When a full Availability Zone disruption occurs—that is, when all nodes in that Availability Zone lose connectivity to the Kubernetes control plane—the node lifecycle controller in Kubernetes detects the situation and evicts pods from the impacted zone. Pods on unreachable nodes are marked as `Terminating` and new pods are scheduled on healthy nodes in available Availability Zones. During this period, affected nodes display a `NotReady` status, the scheduler prevents new

pods from being placed on those nodes, and the EndpointSlice controller removes endpoints that are associated with the impaired Availability Zone from service routing until connectivity is restored.

For scenarios that involve partial node failures within a zone—where only a subset of nodes becomes unreachable—the node lifecycle controller applies different eviction behavior. If the disruption persists beyond the configured toleration period (by default, five minutes), pods on disconnected nodes are marked as `Terminating` and new pods are scheduled on healthy nodes in available Availability Zones.

## Implementing Amazon EKS zonal shift for improved resilience

Amazon EKS zonal shift, which integrates with Amazon Application Recovery Controller (ARC), provides a mechanism to proactively manage traffic during Availability Zone impairments. This capability allows temporary redirection of network traffic away from an unhealthy Availability Zone toward healthy zones within the same AWS Region to minimize service disruption.

## Understanding the zonal shift mechanism

Amazon EKS zonal shift addresses east-west traffic (inter-pod communication within the cluster). When zonal shift is configured with Application Load Balancers or Network Load Balancers, it also supports ingress traffic routing. The mechanism operates by coordinating multiple Kubernetes and AWS control plane components to safely redirect traffic without disrupting running workloads. During an active zonal shift, Amazon EKS automatically performs the following coordinated actions:

- **Node cordoning**: All nodes in the impaired Availability Zone are cordoned. This prevents the Kubernetes scheduler from placing new pods on the nodes while it maintains existing workloads.
- **Availability Zone rebalancing suspension**: For managed node groups, Availability Zone rebalancing operations are suspended, and Auto Scaling groups are updated to launch new data plane nodes exclusively in healthy Availability Zones. This ensures that new capacity isn't provisioned in the impaired zone.
- **Endpoint removal**: The EndpointSlice controller removes pod endpoints in the impaired Availability Zone from all relevant EndpointSlices. This ensures that service discovery and load balancing mechanisms route traffic only to pods that are running in healthy Availability Zones.
- **Workload preservation**: Amazon EKS refrains from terminating nodes or evicting pods in the affected Availability Zone. It maintains full capacity in the impaired zone so that when the

zonal shift expires or is canceled, traffic can safely return without requiring additional scaling operations.

## Zonal shift activation methods

You can choose from two approaches to initiate zonal shifts, depending on your operational model:

- Manual zonal shift provides operator-driven control when specific Availability Zone issues are detected through monitoring, alerts, or customer reports. This method requires explicit action through the ARC console, AWS Command Line Interface (AWS CLI), or zonal shift APIs, where operators specify the impaired Availability Zone and define an expiration time for the shift. Manual shifts are appropriate when teams have dedicated monitoring and on-call capabilities and prefer to maintain direct control over traffic management decisions.
- Zonal autoshift authorizes AWS to automatically initiate shifts when ARC detects potential Availability Zone failures or impairments based on internal telemetry and health signals across multiple AWS services, including network metrics, Amazon Elastic Compute Cloud (Amazon EC2), and Elastic Load Balancing. AWS automatically ends an autoshift when indicators show that the issue has been resolved. If you want the highest availability posture with minimal manual intervention, we recommend this approach, because it enables sub-minute response to detected Availability Zone impairments.

## Prerequisites for effective zonal shift

For zonal shift to successfully protect applications during Availability Zone impairments, you must architect your clusters for Multi-AZ resilience before you enable the zonal shift feature:

- **Multi-AZ node distribution**: Provision worker nodes across at least three Availability Zones to ensure sufficient redundancy when one zone becomes unavailable.
- **Capacity planning**: Pre-provision enough compute capacity across healthy Availability Zones to accommodate the full workload when one Availability Zone is removed from service, because scaling operations during an active disruption might encounter insufficient capacity.
- **Pod distribution and pre-scaling**: Deploy multiple replicas of each application across all Availability Zones and pre-scale critical system components such as CoreDNS in every zone. This helps ensure that sufficient capacity remains after a zone is shifted away.

## Recommendations for zonal disruption resilience

- **Enable zonal shift at cluster creation**: For new EKS clusters, enable zonal shift integration with ARC during initial provisioning through the Amazon EKS console, AWS CLI, or infrastructure as code (IaC) tools such as AWS CloudFormation. EKS Auto Mode clusters that are created with quick configuration have zonal shift enabled by default.
- **Select the appropriate activation method**: Choose zonal autoshift for production environments that require maximum availability with automated response, particularly for customer-facing applications where minutes of downtime during an Availability Zone impairment can carry significant business impact. Use manual zonal shift for environments where operations teams prefer to provide explicit approval before traffic shifts, or where application testing and validation are still in progress.
- **Test resilience before production deployment:** Validate cluster behavior under Single-AZ loss by manually initiating test zonal shifts or enabling zonal autoshift practice runs to verify that applications maintain availability, performance remains acceptable, and capacity is sufficient when operating with reduced Availability Zone count. We strongly recommend this testing so you can identify configuration gaps before actual Availability Zone impairments occur.
- **Coordinate with load balancer configuration**: For applications that receive external traffic, enable ARC zonal shift on associated Application Load Balancers and Network Load Balancers to ensure that both ingress traffic and in-cluster east-west traffic shift together during Availability Zone impairments. This coordination prevents scenarios where external requests reach healthy pods but those pods cannot communicate with dependencies in the shifted-away zone.
- **Monitor shift operations**: After you enable zonal shift, configure monitoring and alerting for shift events, including autoshift activations, manual shift initiations, and shift expirations, to maintain operational visibility into traffic management actions and their impact on application behavior.

## Shift completion and recovery

When a zonal shift expires based on its configured duration or is manually canceled after the Availability Zone impairment resolves, the EndpointSlice controller automatically updates all EndpointSlices to reincorporate endpoints in the restored Availability Zone. Traffic gradually returns to the previously impacted zone as clients refresh endpoint information and establish new connections. This enables full cluster capacity utilization without requiring manual intervention or pod rescheduling.

# Conclusion

When you design your architecture for high application availability and resiliency, consider the following components:

- The microservices application (its pods and containers)
- The workload data plane (Ingress Controller, pod, system components such as the Amazon VPC CNI, service mesh sidecars, and kube-proxy)
- The workload-management layer (controllers, admission controllers, network policy engines, and persistent data storage for these components)
- The Kubernetes control plane
- Infrastructure (nodes, network, and network appliances)

To address those component considerations, use the following key strategies:

- To help ensure high availability and fault tolerance, spread workloads across nodes and Availability Zones.
- To protect critical workloads, maintain application stability during disruptions by using pod disruption budgets (PDBs).
- To help ensure that pods are running and serving traffic correctly, configure startup probes, liveness probes, readiness probes, and load balancer health checks.
- To manage container state transitions efficiently, configure container lifecycle hooks.
- To provide control over the eviction process during node failures or maintenance, configure the pod eviction time.

By implementing these practices, you can significantly enhance the reliability and resilience of applications running on Amazon EKS, ensuring robust performance and high availability.

# Resources

- Kubernetes Pod Topology Spread Constraints (Kubernetes documentation)
- Karpenter FAQs (Karpenter documentation)
- Descheduler for Kubernetes (GitHub repository)
- Availability and Pod Lifecycle *Amazon EKS Best Practices Guide*
- Gracefully shutdown applications *Amazon EKS Best Practices Guide*
- [EKS] [request]: Ability to configure pod-eviction-timeout and workarounds (Containers Roadmap repository)

# Document history

The following table describes significant changes to this guide. If you want to be notified about future updates, you can subscribe to an RSS feed.

| Change | Description | Date |
| --- | --- | --- |
| Update | Revised the section about pod evictions during zonal disruptions. | October 29, 2025 |
| Update | Revised the Use pod topology spread constraints section. | January 27, 2025 |
| Initial publication | — | October 23, 2024 |