Kubernetes Resiliency Guidelines

Document Level Classification

200

Introduction

These guidelines can help you deploy resilient applications that can gracefully execute rolling updates and peacefully survive cluster upgrades. For this to work correctly, you need to understand the behavior of Kubernetes as well as the primitives that enable high availability. For brevity, this guide does not include the best practices for pod or node autoscaling. For additional production best practices for your Kubernetes specs and software, please check out the community-maintained production checklist: https://learnk8s.io/production-best-practices.

Configuration Requirements

At a high level, the configuration requirements for a resilient application running on Kubernetes are:

- An application that properly catches `SIGTERM` messages from the kernel and begins to gracefully shut down and exit with code `0` .
- `Pod` resource requests (CPU and Memory) must be set and sufficient to enable the application to function under production load.
- `Pod` Liveness, Readiness, and Startup Probes. Liveness probes cause your pod to be restarted if they fail. Readiness probes cause your pod to be removed from the load balancer if they fail. Startup probes must pass before your pod is monitored for readiness and liveness and are used to specifically monitor for a potentially extended period of time where the pod is still coming online.
- A `Pod` `terminationGracePeriodSeconds` configuration that is long enough to cover the entire graceful shutdown process in the worst case. This grace period should only be hit when the pod has failed to shut down gracefully and a forceful container kill needs executed to complete the shutdown.
- A soft (or hard) `Pod` `podAntiAffinity` configuration that attempts to space pods apart from each other within your cluster, thus minimizing the disruption if a single node fails.
- Health checks
- A `podDisruptionBudget` configured to target the pods in your application will restrict how many pods can be gracefully evicted concurrently. With a PDB, you can configure the maximum number of pods that can be unavailable (not passing health checks) when nodes are drained.
- [Optional] Use a `podTopologyConstraint` to stripe your pods across AWS availability zones in your region to better mitigate availability zone failure.

Upgrades

When a cluster upgrade happens, the following actions take place:

- A cluster admin installs the version-specific pre-requisites needed by the cluster or AWS prior to the upgrade.
- The AWS EKS (Kubernetes) control plane version is upgraded. Nodes are able to join control planes within two minor revisions of their running Kubelet version.
- A rolling update is begun that does the following to all nodes that are not of the prior version:
    - The node is cordoned. This prevents any new pods from scheduling on this `node`. Existing workloads are retained.
    - The node is drained. This drain operation calls the Kubernetes eviction API on each non-daemonset `pod` on the `node`. The following actions occur for each pod:
        - Pod Disruption Budgets (`pdb`) are respected, which limit the number of pods for a specific application that can be sent shutdown commands simultaneously.
            - The system waits for an "Allowed Disruption" count on the PDB to become available.
        - The pod is sent a `SIGTERM` shutdown signal
        - The `terminationGracePeriodSeconds` of the pod is waited. If the pod exits with code `0`, then the system moves on to evaluate the next pod on the node.
            - If the pod does not exit in `terminationGracePeriodSeconds`, then it will be sent a `SIGKILL` which will cause the kernel to destroy the process memory immediately
        - Kubernetes detects that the number of pods created for the service in question is now less than what is desired.
        - A new pod is spawned by Kubernetes and scheduled on another node in the cluster. This other node could be any

node that is not cordoned and fits the pod's scheduling constraints.
        - If no node is available where the pod fits, the pod goes into the state of `Pending` and waits for a node that it can schedule on.
        - Karpenter immediately identifies that a pod has gone `Pending` and provisions an appropriately sized node to host this pod based on the pod's resource requests.
    - When the node is emptied of pods, it is sent a `Terminate` command via AWS, which is like hitting the power button on a server.
    - The server is removed from the `nodes` list on the Kubernetes control plane
    - All newly created nodes during the cordon, drain, and delete procedure are of the proper new Kubernetes version. Once all of the prior version nodes are deleted, the cluster is fully upgraded.

Application Rollouts

When an application rollout happens:

- A new `deployment` spec is applied to Kubernetes for a service.
- A new `replicaset` is created using the new `deployment` spec and pods are built according to the rollout rules specified within the deployment.
- When these new pods begin to pass all their health checks, one of the older pods from the prior `replicaset` is drained and killed by doing the following:
    - The pod is sent a `SIGTERM` shutdown signal
    - The `terminationGracePeriodSeconds` of the pod is waited. If the pod exits with code `0`, then the system moves on to evaluate the next pod on the node.

    - If the pod does not exit in `terminationGracePeriodSeconds`, then it will be sent a `SIGKILL` which will cause the kernel to destroy the process memory immediately
- This rollout process continues until all pods are upgraded. Several old replicasets are retained to enable rollback, but have their replica counts set to `0`.

Node Failures

When a node fails:

- The node disappears immediately and all pods on it are killed instantly. This is why it is important to ensure you use pod anti-affinity and topology aware scheduling configurations to ensure that the fewest possible pods from any one application are on this node at the same time.
- `kubelet` healthchecks from the node begin to fail with the Kubernetes control plane.
- After several missed heartbeats, the node state is changed to `Unknown`.
- All pods on `Unknown` nodes are considered missing entirely and the number of running pods is updated across all `replicasets` and `statefulsets`.
- Kubernetes sees that the number of running pods requested for these services does not match the number of pods that are actually created and requests that new pods be created in the state `Pending`.
- The new pods are scheduled to a node in the cluster that is in the status `Ready`. If no nodes are available, Karpenter will deploy a new node of the appropriate size and join it to the cluster.
- When the new node joins the cluster and is of the state `Ready`, pods will be scheduled to it.

Pod Disruption Budgets

Pod Disruption Budgets are a resource used for configuring the number of pods in a service/deployment can be taken offline by node maintenance/ compaction or cluster upgrades. They allow you to inform Kubernetes that a service's pods cannot be impacted by more than a defined percent. Kubernetes will evaluate the PDB value of all pods running on a node during events where pods must be shifted to other nodes.

PDBs *do not* impact deployment rollouts – those are configured via a separate series of rollingUpdate options on the Deployment resource.

epic-app supports pod disruption budgets out-of-the-box, and by default will create one where `maxUnavailable` is set to 10% of the running pod count of a deployment.

`kubectl` can be used to inspect the values of PDBs, and see the value that Kubernetes has currently evaluated to be the maximum number of available pods based on the current number running:

```
$ kubectl get pdb
NAME                          MIN AVAILABLE   MAX UNAVAILABLE   ALLOWED
account-policy-service-ci     N/A             10%               1
```

Page Information:
Page ID: 81068403
Space: Cloud Developer Platform
Downloaded: 2025-07-12 04:09:10