International Journal of Management, IT & Engineering
Vol. 14 Issue 12, December 2024,
ISSN: 2249-0558 Impact Factor: 7.119
Journal Homepage: http://www.ijmra.us, Email: editorijmie@gmail.com
Double-Blind Peer Reviewed Refereed Open Access International Journal - Included in the International Serial Directories Indexed & Listed at: Ulrich's Periodicals Directory ©, U.S.A., Open J-Gate as well as in Cabell's Directories of Publishing Opportunities, U.S.A

How to achieve Zero-downtime deployments and Graceful shutdown of K8 pods running on EKS

Jatla.Prasanna Senior Backend Engineer at One Real

Achieving zero-downtime deployments and graceful shutdowns for Kubernetes pods in Amazon EKS can be a challenge, particularly for Spring Boot applications handling live traffic through an Ingress load balancer. With CI/CD pipelines and tools like ArgoCD, deployments are frequent, but poorly managed pod shutdowns can cause issues such as HTTP 502 errors.

To understand the reason why this happens, first let's look at the Pod life cycle. When a rolling deployment starts, new pods get created and once the pod is ready, the old pod starts terminating. To terminate a pod there are 2 processes that run in parallel.

For an application using Amazon's Application Load Balancer (ALB) and a NodePort service, traffic flows through the ALB to nodes and then is routed by kube-proxy using iptables rules to reach the appropriate pod.

What happens internally at kubernetes control plane level with instance type alb:

1. Kube-apiserver receives the pod deletion request and updates the state of the pod to Terminating at Etcd;
2. Endpoint Controller deletes the IP of the pod from the Endpoint object; In the latest versions of kubernetes, a new concept of EndpointSlices was introduced.
3. Kuber-proxy updates the rules of iptables according to the change of the Endpoint object, and no longer routes traffic to the deleted pod.
4. While all these is happening, Kube-apiserver sends signal to Kubelet to clean up container-related resources at the node, such as storage, network;
5. Kubelet sends SIGTERM to the container; if there are no configurations for the process within the container, the container will exit at once.If the container didn't exit within the default 30s, Kubelet will send SIGKILL and force it to exit.

This termination process involves multiple asynchronous steps that can lead to a brief period where the load balancer tries to route requests to a pod that is already shutting down. This creates a race condition where kube-proxy might not yet be aware of the terminated state, leading to temporary connection failures or HTTP 502 errors.

To overcome this issue, we need to implement below changes:

1. Implement a delay in processing shutdown signals by handling SIGTERM signals in Spring Boot applications, ensuring all in-flight requests are completed before termination. In Spring Boot (if using version 2.3 or higher) to delay shutdown until current requests are complete, set below properties

```
server.shutdown=graceful
spring.lifecycle.timeout-per-shutdown-phase=20s
```

2. Add a preStop hook: A preStop hook is a lifecycle hook in Kubernetes that runs just before Kubernetes sends a SIGTERM signal to terminate the container. By adding a delay (like a sleep command) in this hook, Kubernetes effectively "pauses" the termination process, allowing time for other services in the cluster to be informed about the pod's impending shutdown. The preStop hook is executed by the Kubelet, which waits until the hook's commands finish running before it sends the SIGTERM signal to the container. And another config that plays a vital role is

terminationGracePeriodSeconds.The terminationGracePeriodSeconds parameter sets the time (in seconds) that Kubernetes will wait after sending the SIGTERM signal before it sends a SIGKILL signal to forcibly terminate the pod. By default, Kubernetes sets this period to 30 seconds. The countdown starts as soon as SIGTERM is sent. If the application doesn't shut down within the configured grace period, Kubernetes issues a SIGKILL, which immediately terminates the container. The goal is to give the application enough time to finish processing any remaining requests before it's forcibly killed. If your application takes, for example, 30 seconds to handle in-flight requests during a shutdown, you should set terminationGracePeriodSeconds to 50 seconds to account for the preStop hook's sleep period plus any additional processing time.

```
terminationGracePeriodSeconds: 50
containers:
 - image: {{ .Values.image }}:{{ .Values.version }}
   imagePullPolicy: Always
   name: demo-api
   lifecycle:
     preStop:
       exec:
         command:
           - /bin/sh
           - -c
           - sleep 30
```

But, even after changing this with our ingress of instance type and service of node-port on EKS still routing traffic to terminating pod. We resolved this problem by moving to ingress of ip-type and service of Cluster-type.

When using an ALB with instance-type configuration, the load balancer targets **EC2 instances** (nodes) within the EKS cluster, rather than individual pods. In this setup, the ALB routes traffic to any node in the cluster that can then route requests to the appropriate pods. Traffic routing depends on **kube-proxy**, which uses iptables on the nodes to forward traffic to the correct pod.Where as with IP-type ALB, the load balancer targets the **IP addresses of the pods** directly rather than the nodes. This setup allows the ALB to communicate directly with the running pods.

Kubernetes Service Types: ClusterIP and NodePort

1. **ClusterIP**:
    - The default service type in Kubernetes, ClusterIP, is internal to the cluster and provides a stable IP address for pods that can only be accessed within the cluster.
    - When used with an IP-type ALB, it ensures that only active, healthy pods are routed to by the load balancer, avoiding issues where requests reach terminating pods.
2. **NodePort**:
    - NodePort services expose a port on each node in the cluster, enabling external access to services running in the cluster by targeting the nodes directly.
    - Combined with an instance-type ALB, NodePort requires kube-proxy to handle traffic routing from nodes to pods. This can lead to delays in updating routing rules during pod terminations, as kube-proxy might still route traffic to terminating pods, causing temporary errors.

Below is the yaml for our ingress resource for ip-type

```
kind: Ingress
metadata:
 name: demo-ip-type
 namespace: demo
 annotations:
   alb.ingress.kubernetes.io/scheme: internet-facing
   alb.ingress.kubernetes.io/group.name: alb-ip-type
   alb.ingress.kubernetes.io/target-type: ip
   alb.ingress.kubernetes.io/ip-address-type: ipv4
   alb.ingress.kubernetes.io/target-group-attributes:
deregistration_delay.timeout_seconds=30
```

And service yaml looks like this:

```
kind: Service
metadata:
 name: demo-api
spec:
 selector:
   app: demo-api
 type: ClusterIP
 ports:
```

```
  - name: http
    protocol: TCP
    port: 80
    targetPort: 8080
```

Our service type needs to be changed to ClusterIP from NodePort. Changing this configuration creates a new service and a new ALB. Now, when we view this ALB in the AWS console, we can see pod IP addresses directly in the target group, instead of the node/instance IPs that were present with the instance-type load balancer. This change means that the ALB now receives updates from Kubernetes components whenever there are changes in pod states. The ALB also performs health checks directly on the pods at regular intervals (as configured). Setting the interval too low, however, can result in all ALB instances continuously probing all the pods, which can lead to significant processing overhead for the ALB. We found that the default value of 15 seconds with AWS ALB worked well for us. With these settings, we were able to achieve truly zero-downtime deployments.

Why Moving from Instance-Type with NodePort to IP-Type with ClusterIP Helps

Switching to an **IP-type ALB** with a **ClusterIP service** improves routing stability during rolling deployments because:

- **Direct Pod Targeting**: The ALB communicates directly with the pods, reducing dependency on kube-proxy for routing updates.
- **Quicker Removal of Terminating Pods**: The ALB's awareness of pod lifecycle events (like terminating status) allows it to quickly stop routing to terminating pods, lowering the chance of 502 errors.

Using an IP-type ALB with ClusterIP offers a more reliable approach to zero-downtime deployments, especially when working with frequent updates and traffic-sensitive applications in Kubernetes.

Another important component that is worth mentioning is the AWS ALB controller. We have an AWS alb controller running on our kubernetes cluster which is responsible for creating these ingress resources. It is specifically designed to help manage AWS load balancers for applications running in Kubernetes clusters on AWS. It automates the creation and management of Application Load Balancers (ALBs) and Network Load Balancers (NLBs) in response to Kubernetes Ingress and Service resources, simplifying the deployment and scaling of services on AWS. When a Kubernetes Ingress resource is created in an EKS cluster (Elastic Kubernetes Service), the AWS ALB Controller interprets the Ingress specifications and automatically provisions an ALB or NLB, according to the configuration. This means that you don't need to manually set up and configure load balancers in AWS.

When a Kubernetes Ingress resource is created in an EKS cluster (Elastic Kubernetes Service), the AWS ALB Controller interprets the Ingress specifications and automatically provisions an ALB or NLB, according to the configuration. This means that you don't need to manually set up and configure load balancers in AWS.

References

[1]https://aws.github.io/aws-eks-best-practices/networking/loadbalancing/loadbalancing/#availability-and-pod-lifecycle

[2]https://easoncao.com/zero-downtime-deployment-when-using-alb-ingress-controller-on-amazon-eks-and-prevent-502-error/

[3]https://www.thoughtworks.com/en-us/insights/blog/cloud/shutdown-services-kubernetes
[4]https://cloud.theodo.com/en/blog/application-load-balancer-aws

Author Profile

Prasanna is working as Senior Backend Engineer at OneReal.
Email : Prasanna.webtech93@gmaail.com