Containers

A deeper look at Ingress Sharing and Target Group Binding in AWS Load Balancer Controller

by Elamaran Shanmugam, Ratnopam Chakrabarti, Re Alvarez-Parmar, and Praseeda Sathaye | on 23 MAR 2023 | in Amazon Elastic Kubernetes Service, Amazon VPC, Amazon VPC, Containers, Elastic Load Balancing | Permalink |
Share

Introduction

AWS Load Balancer Controller is a Kubernetes controller that integrates Application Load Balancers (ALB) and Network Load Balancers (NLB) with Kubernetes workloads. It allows you to configure and manage load balancers using Kubernetes Application Programming Interface (API). Based on our conversations with customers, we identified two AWS Load Balancer Controller features that need further explanation. In this post, we show how you can reduce costs by using Ingress Grouping and integrate existing load balancers using target group binding.

Kubernetes has two ways to expose Services to clients external to the cluster. Kubernetes Service of type LoadBalancer or Ingress resources. Both methods rely on AWS Elastic Load Balancing under the hood. Network Load Balancer (NLB) is designed to handle high amounts of traffic and is optimized to handle layer 4 (Transmission Control Protocol (TCP) and User Datagram Protocol (UDP)) traffic. And Application Load Balancer (ALB) is designed to handle web traffic and is optimized to handle layer 7 (HTTP and HTTPS) traffic. The AWS Load Balancer Controller automates the management of ALBs and NLBs' lifecycle and configuration as you deploy workloads in your cluster.

Kubernetes ingress and Application Load Balancer

In Kubernetes, Ingress is an API object that provides an external, load-balanced IP address to access the services in a cluster. It acts as a layer 7 (HTTP/HTTPS) reverse proxy and allows you to route traffic to different services based on the requested host and URL path. The AWS Load Balancer controller provisions and configures an ALB on your behalf whenever you create an Ingress.

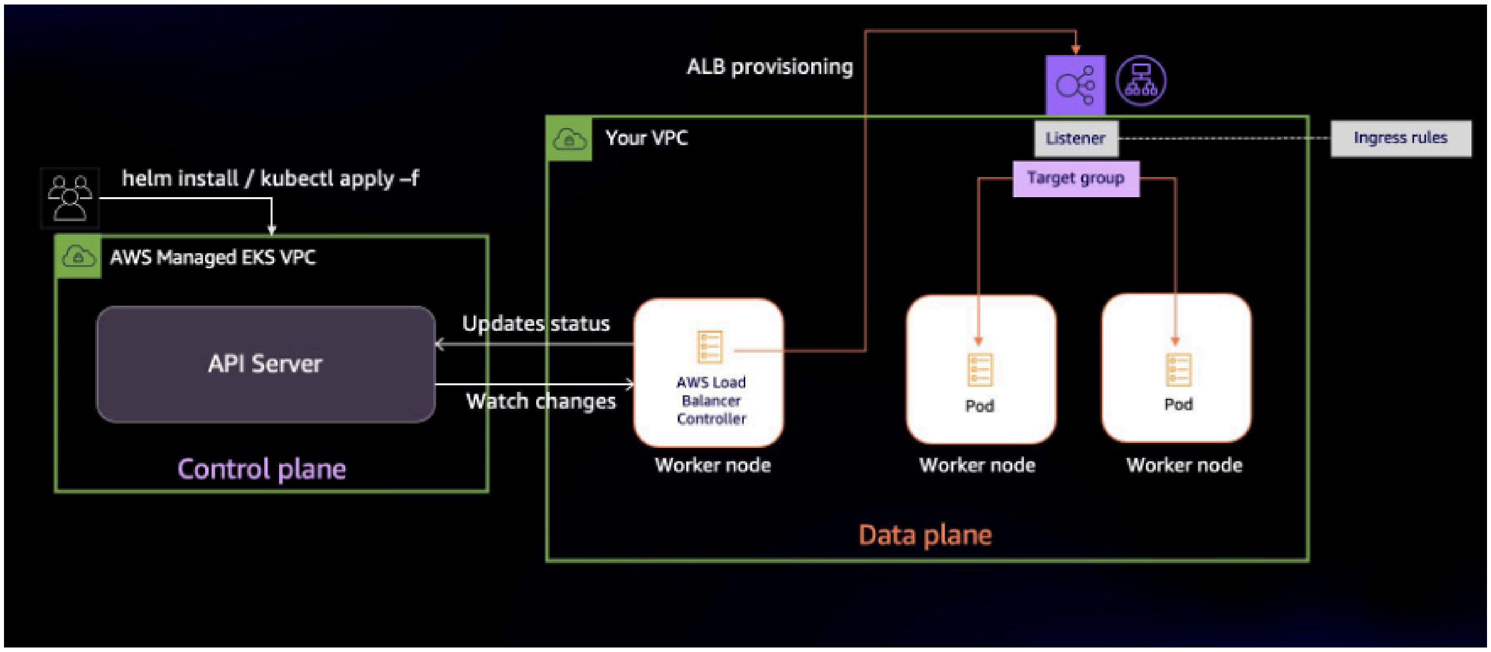
In a microservice architecture, it's common to have multiple services deployed within a single Kubernetes cluster. Each service may have different requirements for external access, routing rules, Secure Sockets Layer (SSL)/ Transport Layer Security (TLS) termination, and so on. In such cases, it may be necessary to use multiple Ingress resources to manage external access to the services. Applications typically include Ingress resource definition in deployment artifacts (along with Deployments, Services, Volumes, etc.) since they contain application-specific routing rules. Separating Ingress resources is a good practice because it allows teams to modify their ingress resources without affecting traffic routing for other applications in the cluster.

Although using the AWS Load Balancer Controller for ingresses can be beneficial; however, there's a disadvantage to this approach. The controller creates an Application Load Balancer (ALB) for each Ingress, which can result in a higher number of load balancers than necessary. This can lead to increased costs since each ALB incurs an hourly charge. Having a separate load balancer for each Service may quickly become too expensive. You can reduce costs by sharing ALBs across multiple Ingresses, thereby minimizing the number of ALBs needed.

To minimize the number of ALBs in your architecture, you can use the AWS Load Balancer Controller to group Ingresses. The controller offers an IngressGroup feature that enables you to share a single ALB with multiple Ingress resources. This allows you to consolidate your load balancers and reduce costs. Additionally, the ingress group can include ingresses from different namespaces, which makes it easier to manage access to your microservices across multiple namespaces.

Ingress groups in action

Let's walk through the process of sharing ALBs with multiple Ingresses. For this demonstration, we'll deploy four web applications in two different namespaces. Then, we'll show you how multiple ingresses can be configured and grouped to share a single ALB. We'll use group.name annotations to enable grouping of multiple ingress resources.

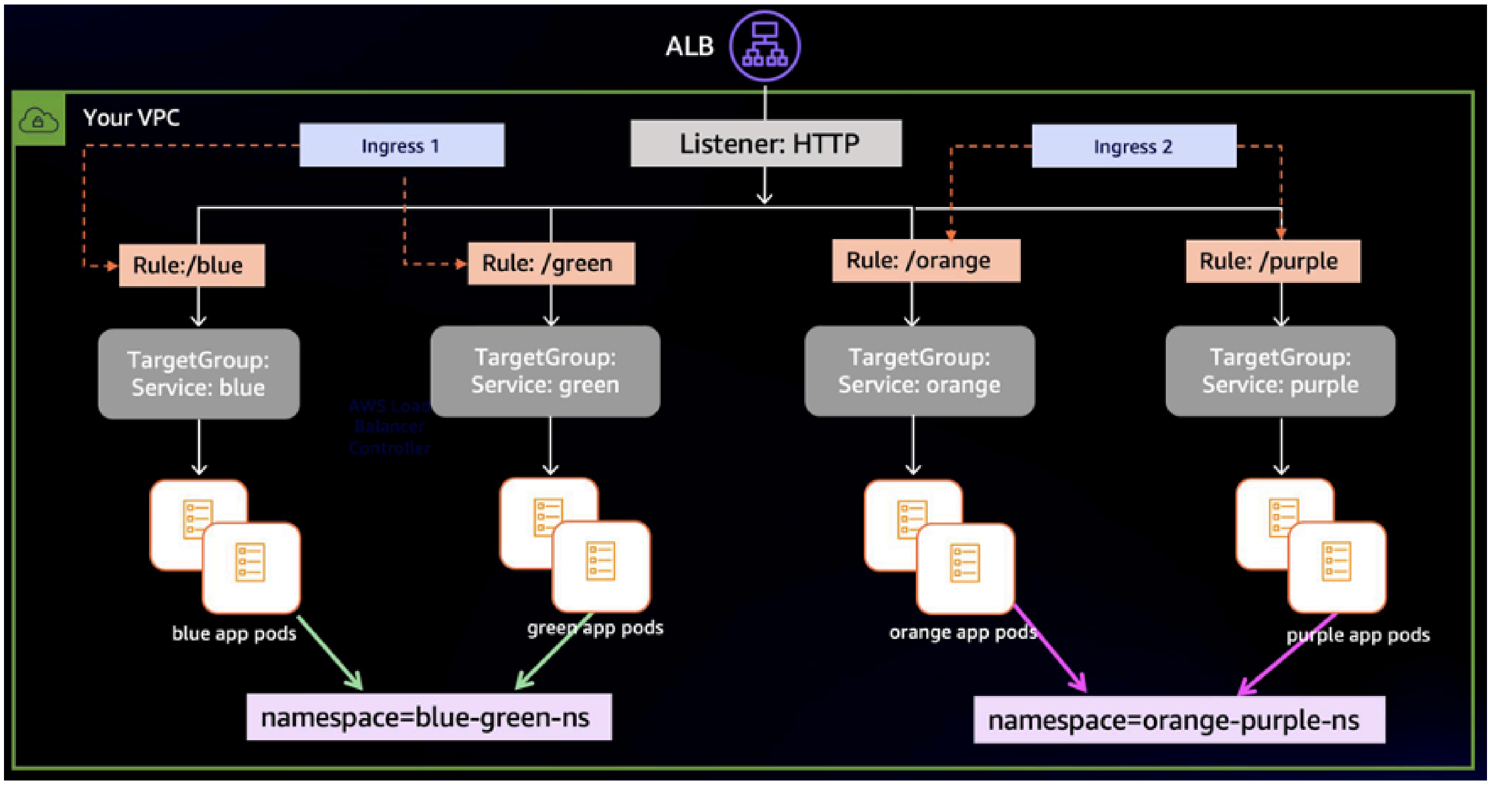


The diagram above shows the operations the AWS Load Balancer Controller performs once installed. It watches the Kubernetes API server for updates to Ingress resources. When it detects changes, it updates resources such as the Application Load Balancer, listeners, target groups, and listener rules.

- A Target group gets created for every Kubernetes Service mentioned in the ingress resource
- Listeners are created for every port defined in the ingress resource's annotations
- Listener rules (also called ingress rules) are created for each path in Ingress resource definition

In this post, we'll run four variants of a web application that renders a web page with different background colors. The blue and green apps runs in the blue-green-ns namespace, and the orange and purple apps run in the orange-purple-ns namespace. With apps deployed, we'll create two ingress resources named blue-green-ingress and orange-purple-ingress. Ingress rules configure the path-based routing.

In the diagram below, when ALB receives traffic, it routes requests to Pods based on the ingress rules. The blue-green-ingress ingress has the routing rules to access blue and green web apps, and deploys in the blue-green-ns namespace. Similarly, the orange-purple-ingress ingress has the routing rules to access orange and purple web apps, and deploys in namespace orange-purple-ns.



Solution overview

You'll need the following things to follow along:

- An existing Amazon Elastic Kubernetes Service (Amazon EKS) Cluster with an existing node group
- AWS Load Balancer Controller installed in the cluster
- Tools required on a machine with access to the AWS and Kubernetes API Server. This could be your local or a remote system or an AWS Cloud9 You'll need these tools installed:
    - AWS Command Line Interface (AWS CLI)
    - eksctl
    - kubectl
    - Helm
    - Docker

The code is available on Github. Start by cloning the code and deploy the sample application:

```
git clone https://github.com/aws-samples/containers-blog-maelstrom.git
kubectl apply -f containers-blog-maelstrom/aws-lb-controller-blog/ingress-grouping/
```

The output should show the resources you've created:

```
namespace/blue-green-ns created
deployment.apps/green-app created
deployment.apps/blue-app created
service/green-service created
service/blue-service created
ingress.networking.k8s.io/blue-green-ingress created
namespace/orange-purple-ns created
deployment.apps/orange-app created
deployment.apps/purple-app created
service/orange-service created
service/purple-service created
ingress.networking.k8s.io/orange-purple-ingress created
```

Check the status of resources in namespace blue-green-ns.

```
kubectl -n blue-green-ns get all
```

The Pods should be in running state:

```
NAME                             READY   STATUS    RESTARTS   AGE
pod/blue-app-9b7bd7578-gjqrm     1/1     Running   0          5d23h
pod/blue-app-9b7bd7578-sgjvd     1/1     Running   0          5d23h
pod/green-app-695664547f-lmq4b   1/1     Running   0          5d23h
pod/green-app-695664547f-lrjh8   1/1     Running   0          5d23h

NAME                     TYPE       CLUSTER-IP      EXTERNAL-IP   PORT(S)        AGE
service/blue-service     NodePort   172.20.93.229   <none>        80:32402/TCP   5d23h
service/green-service    NodePort   172.20.63.132   <none>        80:30106/TCP   5d23h

NAME                         READY   UP-TO-DATE   AVAILABLE   AGE
deployment.apps/blue-app     2/2     2            2           5d23h
deployment.apps/green-app    2/2     2            2           5d23h
```

Similarly, verify that the application Pods are operational in the orange-purple-ns namespace.

We also have two Ingress resources:

```
kubectl get ingress -A
```

You can see that both Ingress resources have the same ADDRESS:

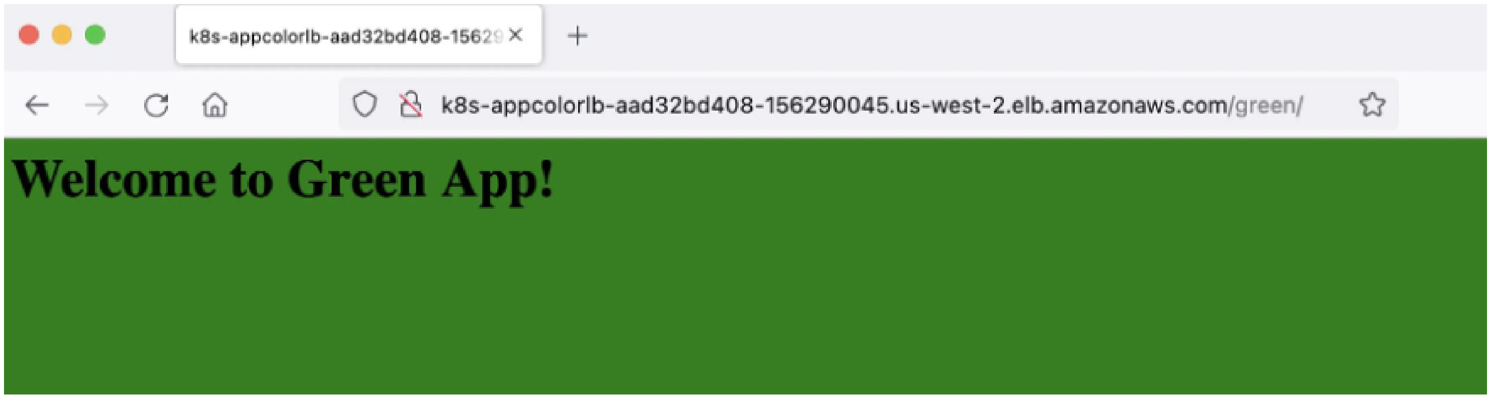
```
NAMESPACE           NAME                     CLASS    HOSTS    ADDRESS
blue-green-ns       blue-green-ingress       alb      *        k8s-appcolorlb-9527f4eb57-1
orange-purple-ns    orange-purple-ingress    alb      *        k8s-appcolorlb-9527f4eb57-1
```

We didn't end up with one ALB for each Ingress is because we have grouped Ingresses. Before looking further at the Ingress, let's verify the routing works the way we want it to.

Get the address assigned to the ALB:

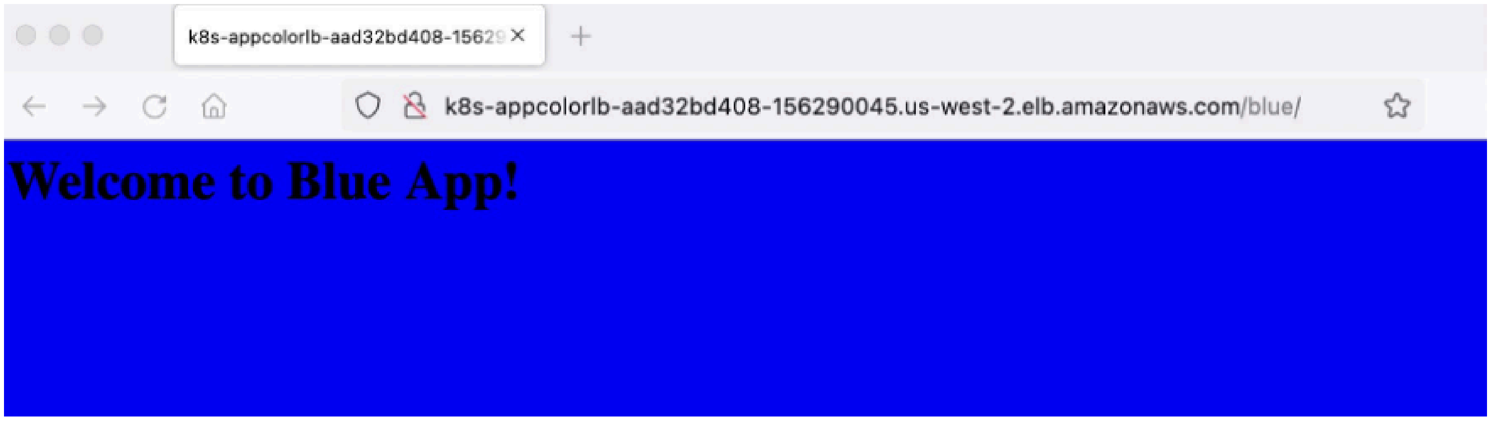
```
kubectl get ingress blue-green-ingress -n blue-green-ns \
  -o=jsonpath="{'http://'}{.status.loadBalancer.ingress[].hostname}{'\n'}"
```

Navigate to the address of the ALB at /green path. A webpage with green background (as shown in the following diagram) indicates that the routing is working as intended.


k8s-appcolorlb-aad32bd408-1562
k8s-appcolorlb-aad32bd408-156290045.us-west-2.elb.amazonaws.com/green/

Welcome to Green App!

Similarly, the /blue, /orange, and /purple paths should show a page each with their corresponding background color.


k8s-appcolorlb-aad32bd408-15629
k8s-appcolorlb-aad32bd408-156290045.us-west-2.elb.amazonaws.com/blue/
Welcome to Blue App!

Let’s get back to the reason for having just one ALB for both Ingresses. Describe either of the Ingresses and you’d notice that they include the `alb.ingress.kubernetes.io/group.name` annotation.

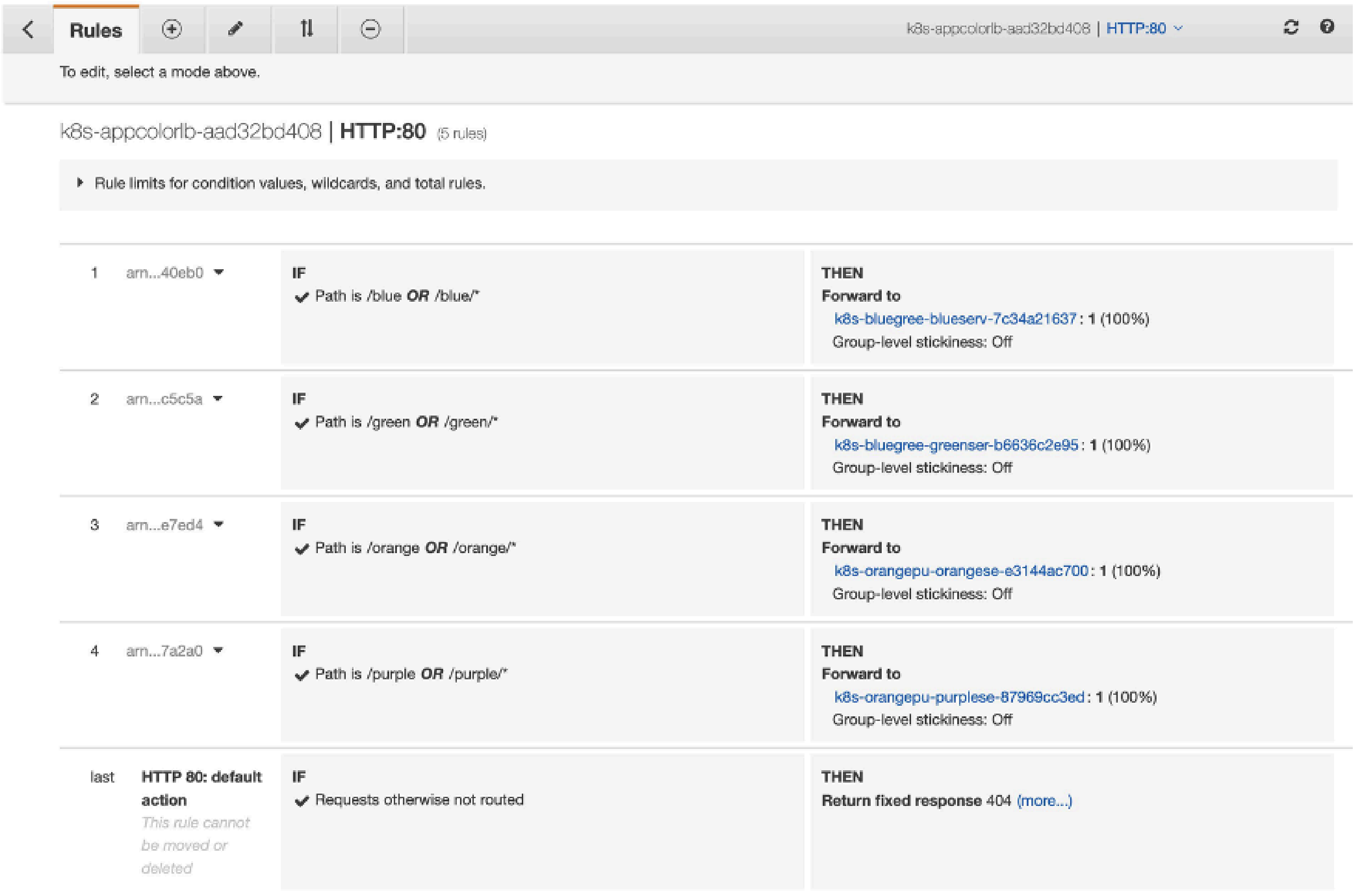
```
kubectl -n blue-green-ns describe ingress blue-green-ingress
```

Below are the Ingress annotations from the blue-green-ingress.yaml:

```
  annotations:
    alb.ingress.kubernetes.io/scheme: internet-facing
    alb.ingress.kubernetes.io/target-type: ip
    alb.ingress.kubernetes.io/group.name: app-color-lb
```

By adding an `alb.ingress.kubernetes.io/group.name` annotation, you can assign a group to an Ingress. Ingresses with same group.name annotation form an IngressGroup. An IngressGroup can have Ingresses in multiple namespaces.

In our example, both blue-green-ingress and orange-purple-ingress use app-color-lb as the value of this annotation, which puts them in the same group. This enables the ALB to route traffic for both the Ingresses based on their corresponding routing rules. You can view the ingress rules as they are configured as listener rules in ALB. The screenshot below shows the Rules that the AWS Load Balancer Controller created:


Rules

k8s-appcolorlb-aad32bd408 | HTTP:80

To edit, select a mode above.

k8s-appcolorlb-aad32bd408 | **HTTP:80** (5 rules)

Rule limits for condition values, wildcards, and total rules.

		IF	THEN
1	arn...40eb0	✓ Path is /blue *OR* /blue/*	**Forward to** k8s-bluegree-blueserv-7c34a21637 : 1 (100%) Group-level stickiness: Off
2	arn...c5c5a	✓ Path is /green *OR* /green/*	**Forward to** k8s-bluegree-greenser-b6636c2e95 : 1 (100%) Group-level stickiness: Off
3	arn...e7ed4	✓ Path is /orange *OR* /orange/*	**Forward to** k8s-orangepu-orangese-e3144ac700 : 1 (100%) Group-level stickiness: Off
4	arn...7a2a0	✓ Path is /purple *OR* /purple/*	**Forward to** k8s-orangepu-purplese-87969cc3ed : 1 (100%) Group-level stickiness: Off
last	**HTTP 80: default action** *This rule cannot be moved or deleted*	✓ Requests otherwise not routed	**Return fixed response** 404 (more...)

With IngressGroup, a single ALB serves as the entry point for all four applications. This change in design reduces the number of ALBs we need and results in cost savings.

Design considerations with IngressGroup

An important aspect to consider before using IngressGroup in a multi-tenant environment is conflict resolution. When AWS Load Balancer Controller configures ingress rules in ALB, it uses the group.order field to set the order of evaluation. If you don't declare a value for group.order, the Controller defaults to 0.

ALB determines how to route requests by applying the ingress rules. The order of rule evaluation is set by the group.order field. Rules with lower order value are evaluated first. By default, the rule order between Ingresses within an IngressGroup is determined by the lexical order of Ingress's namespace/name.

This default ordering of ingress rules can lead to application-wide incorrect routing if any of the Ingresses have misconfigured rules. Teams should create explicit ingress rules to avoid routing conflicts between multiple teams or applications and use group.order to set the order of execution.

Here's another key consideration with using IngressGroup in a shared environment. The name of the IngressGroup is across the cluster as an IngressGroup can have Ingresses in multiple namespaces. Teams should avoid naming collisions by using unambiguous values for group.name. The AWS Load Balancer Controller currently doesn't offer fine grained controls to control access to an IngressGroup. So, avoid giving your group a generic name like

MyIngressGroup, because someone else in the cluster may create an Ingress with the same name, which adds their Ingress to your group. If they create higher priority rules, they may highjack your application's traffic.

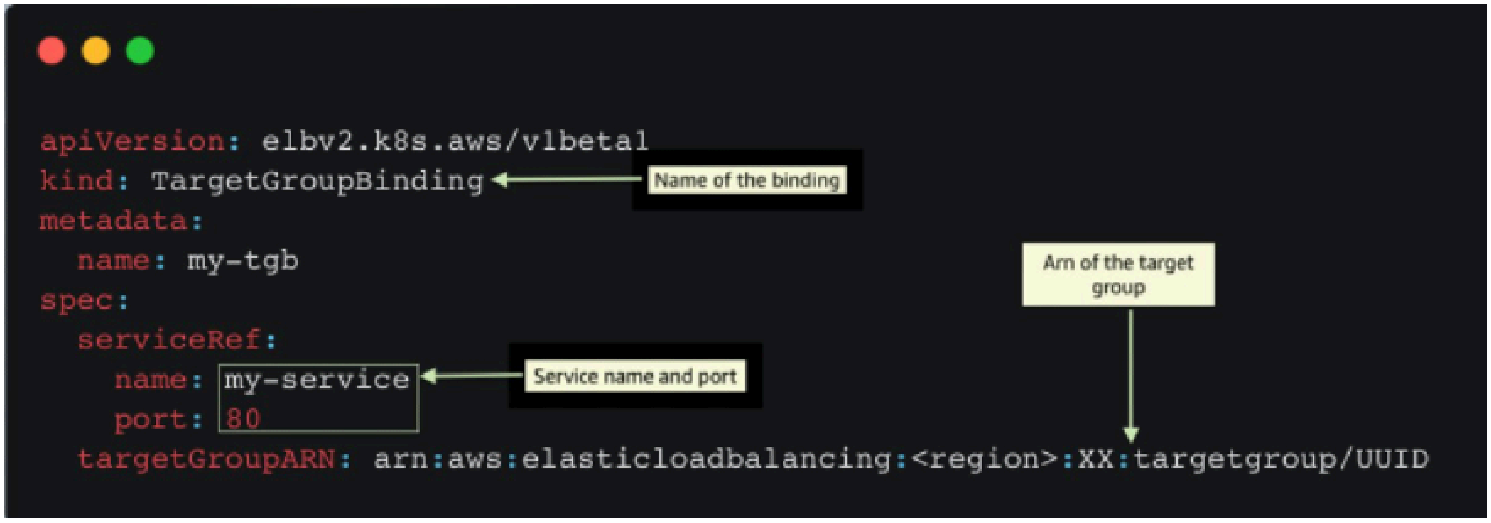
AWS ALB (Application Load Balancer) has several limits on the number of rules that can be configured per listener. These limits are in place to prevent overloading the load balancer and impacting its performance. For example, each ALB listener can have up to 100 rules by default. It's important to check the AWS documentation for the latest information on these limits.

Decouple Load Balancers and Kubernetes resources with TargetGroupBinding

In the previous section, we described the operations the AWS Load Balancer performs when you create an Ingress. The controller creates an ALB when a new Ingress resource is created. If an Ingress is part of IngressGroup, the controller merges ingress rules across Ingresses and configures the ALB, listeners, and rules. The lifecycle of the Load Balancer is tied to the associated one or more Ingresses. If you delete Ingress, the AWS Load Balancer Controller deletes the ALB if there are no other ingresses in the group. The controller creates and configures load balancers to route traffic to your applications.

There are a few scenarios in which customers prefer managing a load balancer themselves. They separate the creation and deletion load balancers from the lifecycle of a Service or Ingress. We have worked with customers that do not give Amazon EKS clusters the permission to create load balancers. In other situations, teams wanted to migrate workloads to Amazon EKS but wanted to preserve the load balancer. In both scenarios, teams needed to the ability to use a pre-existing load balancer to expose Kubernetes Services.

TargetGroupBinding is a custom resource managed by the AWS Load Balancer Controller. It allows you to expose Kubernetes applications using existing load balancers. A TargetGroupBinding resource binds a Kubernetes Service with a load balancer target group. When you create a TargetGroupBinding resource, the controller automatically configures the target group to route traffic to a Service. Here's an example of a TargetGroupBinding resource:



```
apiVersion: elbv2.k8s.aws/v1beta1
kind: TargetGroupBinding
metadata:
  name: my-tgb
spec:
  serviceRef:
    name: my-service
    port: 80
  targetGroupARN: arn:aws:elasticloadbalancing:<region>:XX:targetgroup/UUID
```

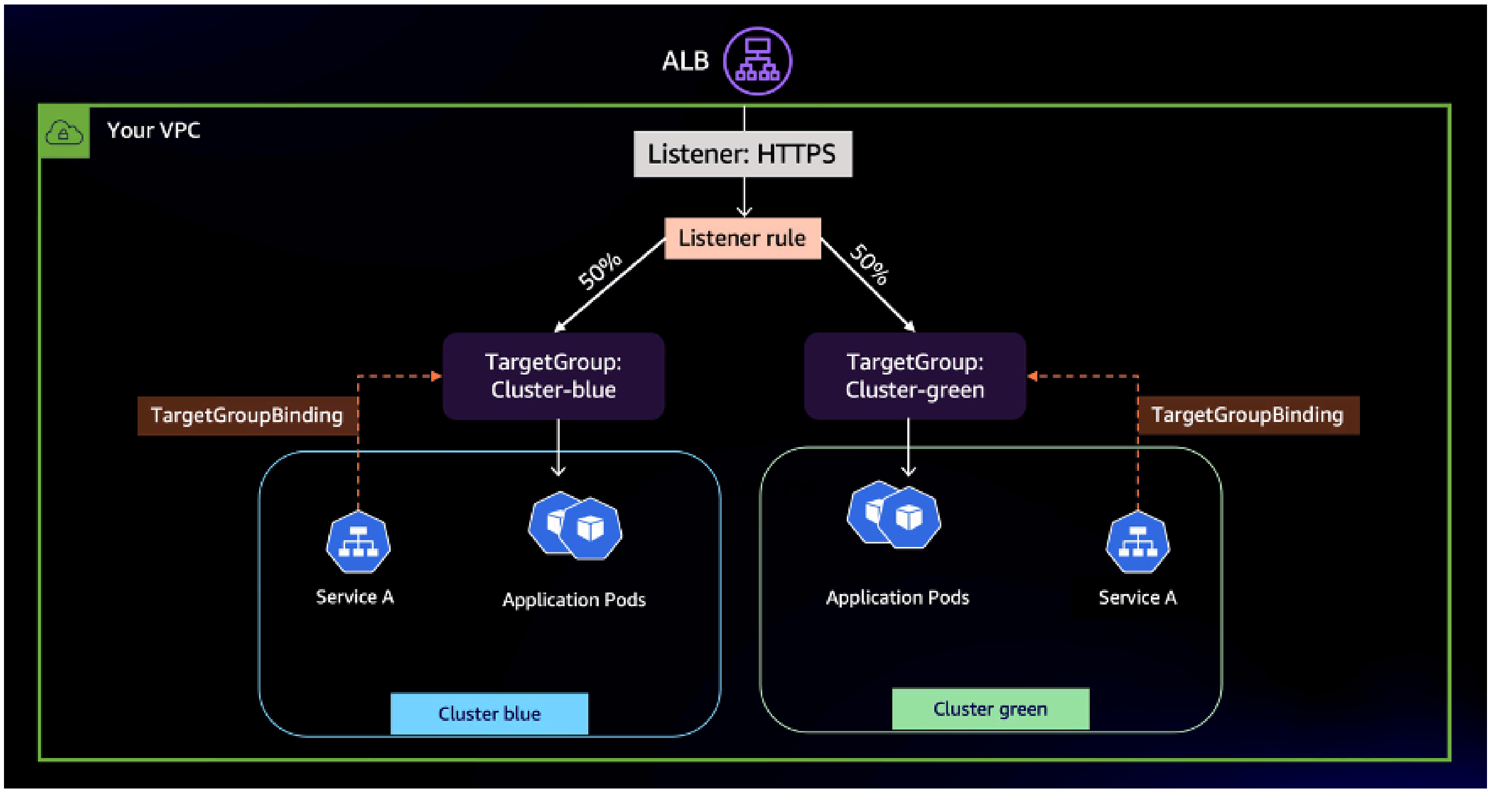
There is one key consideration when using TargetGroupBinding. The type of service used in Kubernetes has a direct relation with the target type configuration in the Elastic Load Balancer. For example if the type of Kubernetes service is `NodePort` then you would need to configure the target type as `instance` to match the implementation. Similarly if the type of Kubernetes service is `ClusterIP` then you would configure the target type as `ip`. In our walkthrough below we will be using the latter approach. For more information on this please read the load balancing sections in the Amazon EKS User Guide.

An obvious advantage is that the load balancer remains static as you create and delete Ingresses or even clusters. The lifecycle of the load balancer becomes independent from the Service(s) it exposes. Another benefit is you can use an pre-existing ALB to distribute traffic to multiple Amazon EKS clusters.

Load balance application traffic across clusters

ALB can distribute traffic to multiple backends using weighted target groups. You can use this feature to route traffic to multiple clusters by first creating a target group for each cluster and then binding the target group to Services in multiple clusters. This strategy allows you to control the percentage of traffic you send to each cluster.

Such traffic controls are especially useful when performing blue/green cluster upgrades. You can migrate traffic from the older cluster to the newer in a controlled manner.



Customers also use this architecture to improve workload resilience in multi-cluster environments. There are customers that deploy their applications to multiple Kubernetes clusters simultaneously. By doing this, they eliminate any Kubernetes cluster from becoming a single point of failure. In case one cluster experiences disrupting events, you can remove it from load balancer targets.

Walkthrough

To demonstrate how TargetGroupBinding works in action, we're going to deploy two versions of a web application in an Amazon EKS cluster. We'll name the deployments black and red. Both applications run in their dedicated namespaces. We'll distribute traffic evenly between the two replicas of our application using an ALB.

We'll create two Services to expose the applications. We will then associate these Services with target groups by creating TargetGroupBinding resources.

Prerequisites:

To be able to follow along, you'll need:

- An Amazon EKS cluster with AWS Load Balancer Controller installed
- Ability to create Application Load Balancer, target groups, and listener

Setup Load Balancing

Set the IDs of your VPC and public subnets in environment variables:

```
VPC_ID=vpc-id
PUB_SUBNET_IDS=$(aws ec2 describe-subnets --filter Name=vpc-id,Values=$VPC_ID --query
```

Create an Application Load Balancer:

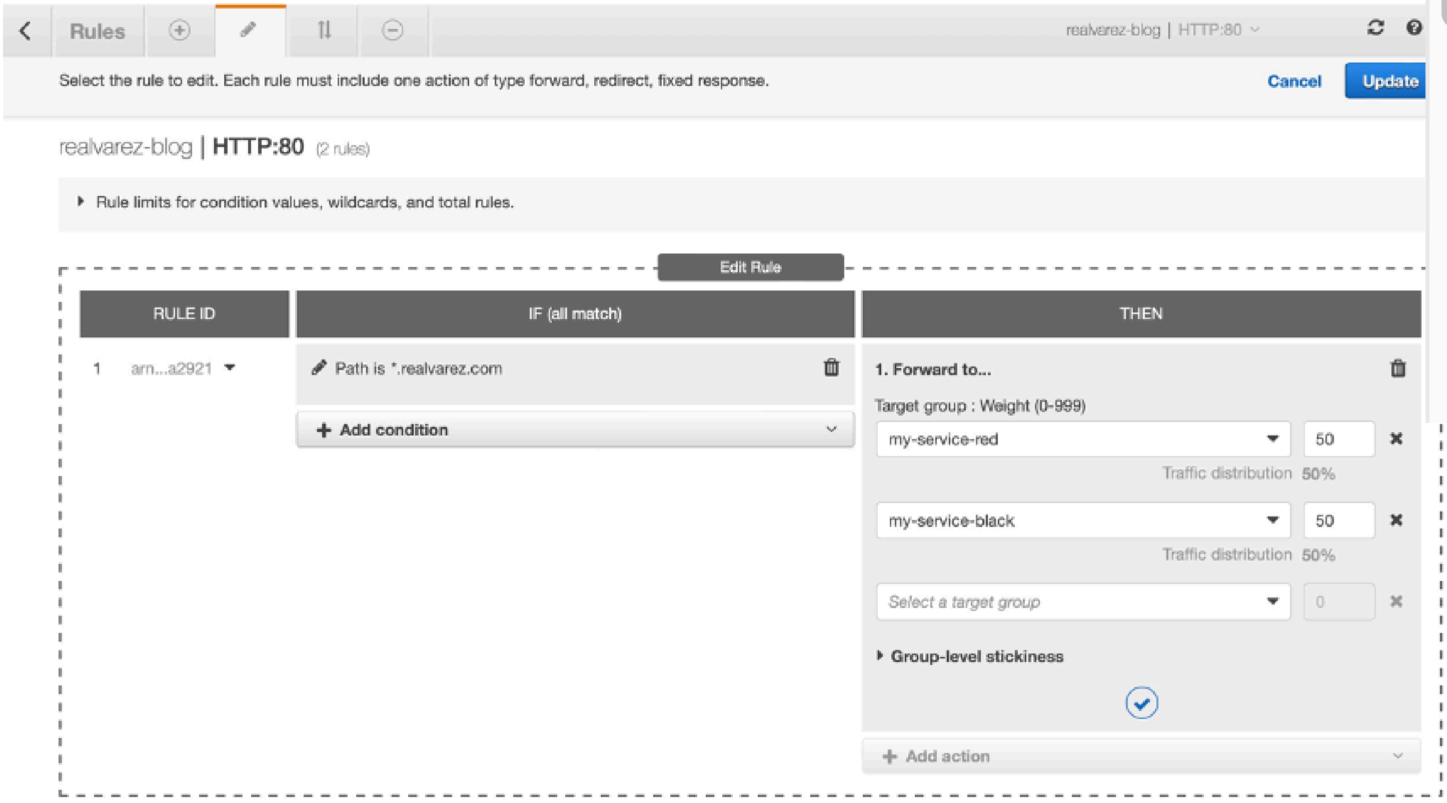
```
ALB_ARN=$(aws elbv2 create-load-balancer \
  --name TargetGroupBinding-Demo-ALB\
  --type application \
  --subnets ${PUB_SUBNET_IDS} \
  --query 'LoadBalancers[].LoadBalancerArn' \
  --output text)
```

Create a listener and two target groups in the same VPC as your Amazon EKS cluster:

```
RED_TG=$(aws elbv2 create-target-group \
  --name my-service-red \
  --port 80 \
  --protocol HTTP \
  --target-type ip \
  --vpc-id $VPC_ID \
  --query 'TargetGroups[].TargetGroupArn' \
  --output text)

BLACK_TG=$(aws elbv2 create-target-group \
  --name my-service-black \
  --port 80 \
  --protocol HTTP \
  --target-type ip \
  --vpc-id $VPC_ID \
  --query 'TargetGroups[].TargetGroupArn' \
  --output text)
```

Here's a screenshot of the target group rules in AWS Management Console after applying the configuration:



Select the rule to edit. Each rule must include one action of type forward, redirect, fixed response.

realvarez-blog | HTTP:80 (2 rules)

Rule limits for condition values, wildcards, and total rules.

RULE ID	IF (all match)	THEN
1 arn...a2921	Path is *.realvarez.com	1. Forward to... Target group : Weight (0-999) my-service-red 50 Traffic distribution 50% my-service-black 50 Traffic distribution 50% Select a target group 0 Group-level stickiness

Deploy applications

Now that we have an Application Load Balancer and the two target groups created, we associate the two target groups with corresponding Services. Let's create manifests for the two target group binding Custom Resource Definition (CRD)s:

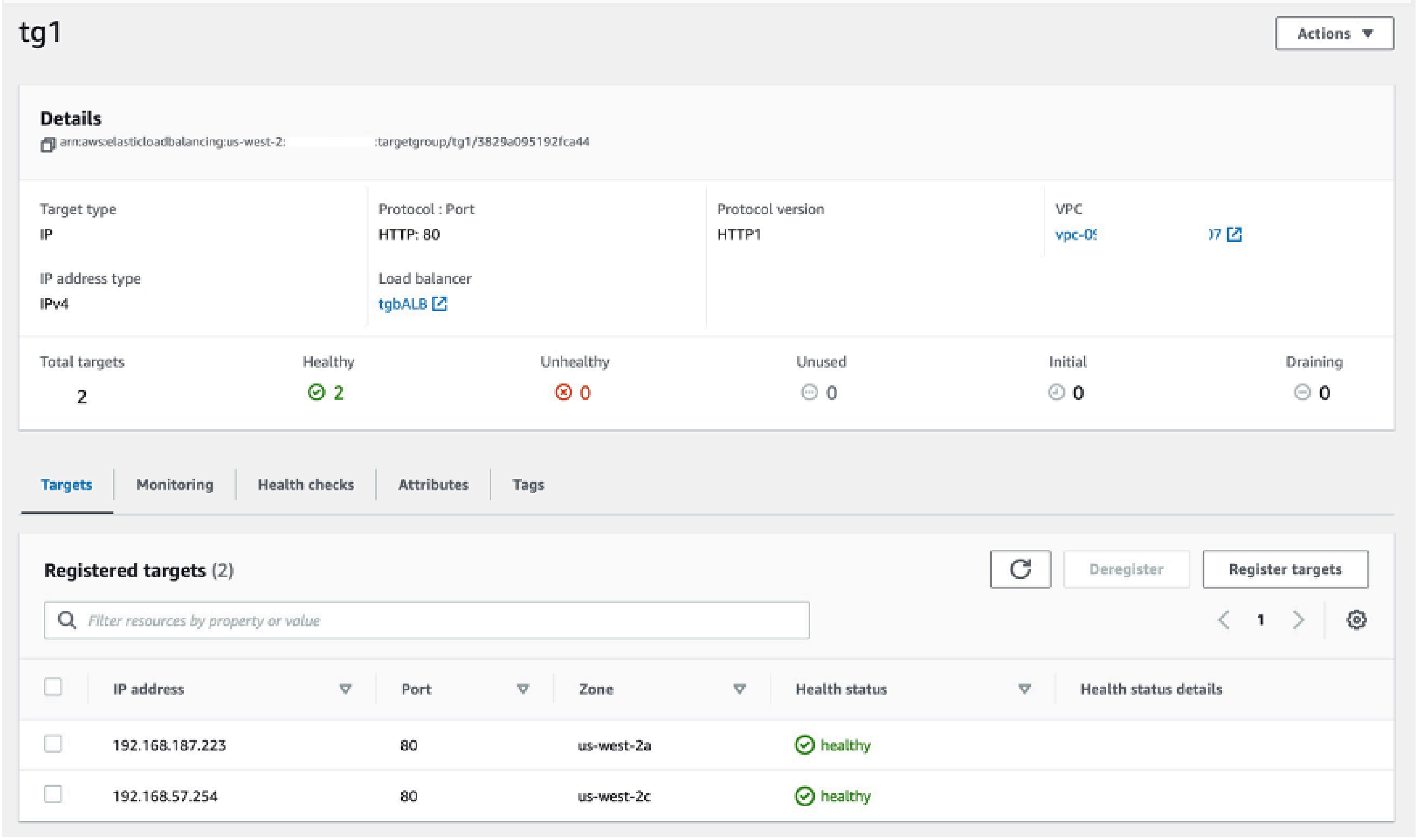
```
# Red Service target group binding
cat > containers-blog-maelstrom/aws-lb-controller-blog/target-grp-binding/red-app-t
apiVersion: elbv2.k8s.aws/v1beta1
kind: TargetGroupBinding
metadata:
  name: red-tgb
  namespace: red-ns
spec:
  serviceRef:
    name: red-service
    port: 80
  targetGroupARN: ${RED_TG}
EOF

# Black Service target group binding
cat > containers-blog-maelstrom/aws-lb-controller-blog/target-grp-binding/black-app
 apiVersion: elbv2.k8s.aws/v1beta1
```

Next, create the application and target group bindings in your cluster:

```
kubectl apply -f containers-blog-maelstrom/aws-lb-controller-blog/target-grp-binding/
```

Let's switch back to the AWS Management Console to visualize this configuration. Navigate to either of the targe groups and you'll see that the AWS Load Balancer Controller has registered corresponding Pods as targets.



Details
arn:aws:elasticloadbalancing:us-west-2: :targetgroup/tg1/3829a095192fca44

Target type	Protocol : Port	Protocol version	VPC
IP	HTTP: 80	HTTP1	vpc-0 7
IP address type	Load balancer		
IPv4	tgbALB		

Total targets	Healthy	Unhealthy	Unused	Initial	Draining
2	2	0	0	0	0

Registered targets (2)

IP address	Port	Zone	Health status	Health status details
192.168.187.223	80	us-west-2a	healthy	
192.168.57.254	80	us-west-2c	healthy	

Once the targets are in healthy status, navigate to the DNS name of the ALB and open it in a browser. You may get a page with a black or blue background. Refresh the page and the colors should alternate.

Note: if you receive a timeout error when accessing the page, verify that the ALB's security groups have an inbound rule to permit HTTP traffic from your IP address.

If you get pages with alternating background, ALB is forwarding your requests to the two Services running in their respective namespace. We can even move /black service to another cluster to load balance traffic between multiple Amazon EKS clusters.

Clean up

You will continue to incur cost until deleting the infrastructure that you created for this post. Use the following commands to clean up the created AWS resources for this demonstration.

```
kubectl delete -f containers-blog-maelstrom/aws-lb-controller-blog/ingress-grouping/
kubectl delete -f containers-blog-maelstrom/aws-lb-controller-blog/target-grp-binding/
aws elbv2 delete-load-balancer --load-balancer-arn $ALB_ARN
aws elbv2 delete-listener --listener-arn $LISTENER_ARN
aws elbv2 delete-target-group --target-group-arn $RED_TG
aws elbv2 delete-target-group --target-group-arn $BLACK_TG
```

Conclusion

In this post, we showed you how to save costs by sharing an ALB with multiple ingress resources. We also explored how to decouple the lifecycle of load balancers from that of Service and ingress resources in a Kubernetes cluster. Finally, we learned how you can use ALB weighted target groups to route traffic to multiple clusters using AWS Load Balancer Controller's TargetGroupBinding feature.

For more information, see the following references:

- AWS Load Balancer Controller LiveDoc
- Different Annotations supported
- Github Link for the AWS Load Balancer Controller project
- Previous post on AWS Load Balancer Controller

TAGS: Amazon EKS, Amazon VPC, Elastic Load Balancing, load balancer