Containers

Operating a multi-regional stateless application using Amazon EKS

by Re Alvarez-Parmar | on 07 DEC 2020 | in Amazon Elastic Kubernetes Service, Containers | Permalink | Share

This post was contributed by Re Alvarez Parmar, Sr Solutions Architect, and Avi Harari, Technical Account Manager.

One of the key benefits of operating on AWS is how easily customers can use AWS's global footprint to run their workloads in multiple regions. Whether you need a multi-region architecture to support disaster recovery or bring your applications closer to your customers, AWS gives you the building blocks to improve your applications' availability, reliability, and latency. This post shows you how you can use Amazon Elastic Kubernetes Service (Amazon EKS) to run applications in multiple AWS Regions and use AWS Global Accelerator to distribute traffic between AWS Regions.

To keep the post simple, we have decided to limit its scope to running a stateless application across multiple AWS Regions; for workloads that require data persistence, customers can rely on features like DynamoDB global tables, Amazon Aurora Global Database, and Amazon S3 cross-region replication. This post proposes a foundational architecture for stateless applications; data and database replication services can complement this architecture to build a blueprint for stateful multi-regional applications.

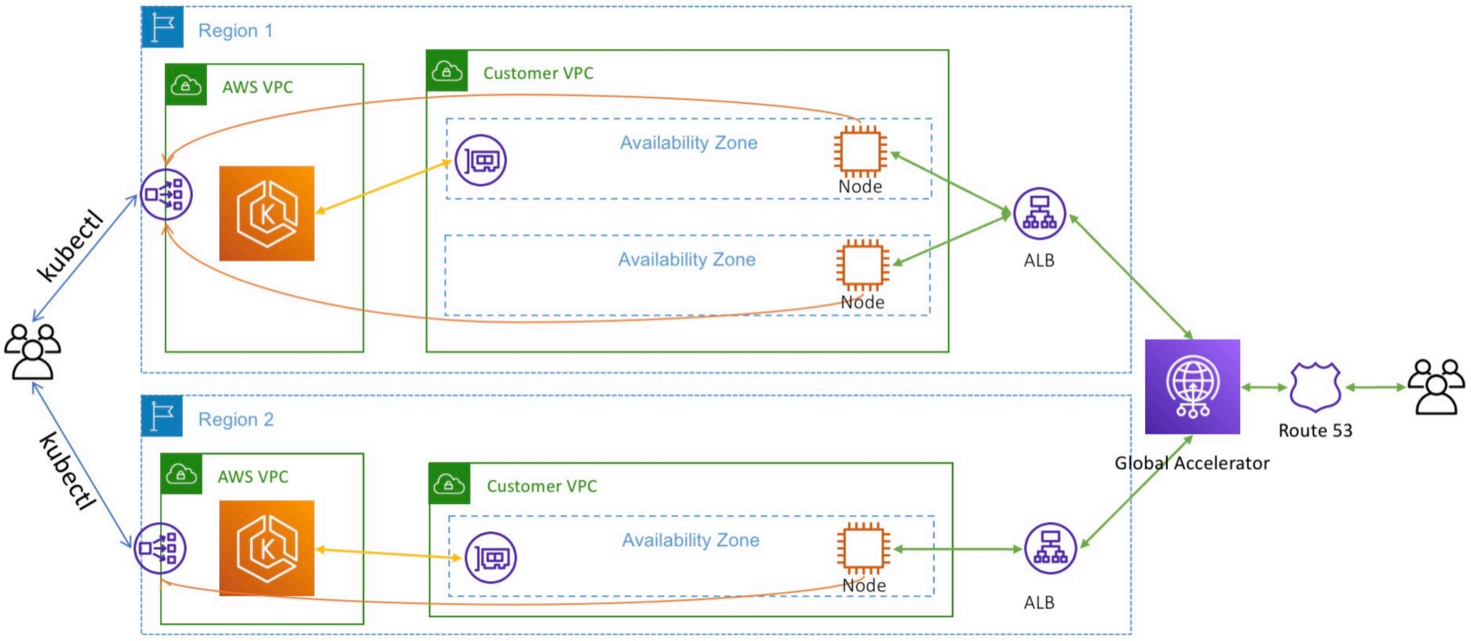
How Amazon EKS helps you go global

Kubernetes' declarative system makes it an ideal platform to operate multi-region deployments. In a declarative system, you declare the desired state, and the system observes the current and desired state and determines the actions required to reach the desired state from the current state. Kubernetes' declarative system makes it easier for you to set up applications and let Kubernetes manage the system state. This allows you to use GitOps tools like Flux and ArgoCD that let you control multiple Kubernetes clusters using a single source of truth. These tools help you minimize configuration drift, an issue to which most concurrent deployments are vulnerable.

Amazon EKS is available in all the 24 AWS Regions, which enables you to operate a consistent, global Kubernetes-backed infrastructure with the help of GitOps and Infrastructure-as-Code tools. You can connect applications hosted in Amazon EKS clusters in multiple AWS Regions over private network using inter-region VPC peering and employ a data replication strategy using AWS Database and Storage services that fit your recovery point (RPO) and recovery time (RTO) objectives. And finally, AWS Global Accelerator's routing capabilities simplify directing traffic to multiple AWS Regions whether you need to implement active-failover, active-active, or more complex scenarios.

Let's explore how you can use AWS Global Accelerator and Amazon Route 53 to distribute traffic to Kubernetes-hosted applications that run in two AWS Regions, a primary region that actively serves traffic and a second region that acts as failover.

Solution



We will deploy a sample application in the two EKS clusters in two different AWS Regions, install AWS Load Balancer Controller in both the clusters, and expose the application in both regions using Application Load Balancer (ALB). Then we will configure these ALBs as endpoints in AWS Global Accelerator. We will configure one region to be the primary and other as failover.

You will need the following to complete the tutorial:

- AWS CLI version 2
- eksctl
- kubectl
- Helm
- Two EKS clusters in different AWS Regions. See creating an Amazon EKS cluster if you don't have one. For this demo, we created two clusters in two different regions using eksctl:
  `eksctl create cluster --name multi-region-blog`
- A domain and a Route 53 hosted zone.

Let's start by setting up a few environment variables:

Bash
```bash
export AWS_REGION_1=<<Primary AWS Region>>
export AWS_REGION_2=<<Secondary AWS Region>>
export EKS_CLUSTER_1=<<EKS cluster name in the primary AWS Region>>
export EKS_CLUSTER_2=<<EKS cluster name in the secondary AWS Region>>
export my_domain=<<example.com Your domain>>
export ACCOUNT_ID=$(aws sts get-caller-identity --query 'Account' --output text)
```

Install the AWS Load Balancer Controller

As we will use an Application Load Balancer for Kubernetes ingress, we have to install the AWS Load Balancer Controller on both EKS clusters. We will use Helm to install the controller. You can skip this step if you have installed the AWS Load Balancer Controller in your cluster already.

> *The AWS ALB Ingress Controller has been renamed to AWS Load Balancer Controller.*

Create an OIDC provider for your clusters:

Bash
```bash
eksctl utils associate-iam-oidc-provider \
  --region $AWS_REGION_1 \
  --cluster $EKS_CLUSTER_1 \
  --approve

eksctl utils associate-iam-oidc-provider \
  --region $AWS_REGION_2 \
  --cluster $EKS_CLUSTER_2 \
  --approve
```

Download the IAM policy document for AWS Load Balancer Controller:

Bash
```bash
curl https://raw.githubusercontent.com/kubernetes-sigs/aws-load-balancer-controller/ma
```

Create an IAM policy for AWS Load Balancer Controller:

Bash
```bash
aws iam create-policy \
    --policy-name AWSLoadBalancerControllerIAMPolicy \
    --policy-document file://awslb-policy.json
```

Create an IAM role, and create a Kubernetes service account in both clusters:

Bash
```bash
eksctl create iamserviceaccount \
  --cluster $EKS_CLUSTER_1 \
  --namespace kube-system \
  --region $AWS_REGION_1 \
  --name aws-load-balancer-controller \
  --attach-policy-arn arn:aws:iam::${ACCOUNT_ID}:policy/AWSLoadBalancerControllerIA
  --override-existing-serviceaccounts \
  --approve

eksctl create iamserviceaccount \
  --cluster $EKS_CLUSTER_2 \
  --namespace kube-system \
  --region $AWS_REGION_2 \
  --name aws-load-balancer-controller \
  --attach-policy-arn arn:aws:iam::${ACCOUNT_ID}:policy/AWSLoadBalancerControllerIA
  --override-existing-serviceaccounts \
  --approve
```

Now that the service account for the AWS Load Balancer Controller is created, let's install the controller. Ensure t
your kubectl's context is set to the primary EKS cluster:

Bash
```
aws eks update-kubeconfig \
  --name $EKS_CLUSTER_1 \
  --region $AWS_REGION_1
```

Install the controller:

Bash
```
helm repo add eks https://aws.github.io/eks-charts
kubectl apply -k "github.com/aws/eks-charts/stable/aws-load-balancer-controller//crds?
helm upgrade -i aws-load-balancer-controller \
  eks/aws-load-balancer-controller \
  -n kube-system \
  --set clusterName=$EKS_CLUSTER_1 \
  --set serviceAccount.create=false \
  --set serviceAccount.name=aws-load-balancer-controller
```

Repeat the steps for the secondary cluster:

Bash
```
aws eks update-kubeconfig \
  --name $EKS_CLUSTER_2 \
  --region $AWS_REGION_2

kubectl apply -k "github.com/aws/eks-charts/stable/aws-load-balancer-controller//crds?
helm upgrade -i aws-load-balancer-controller \
  eks/aws-load-balancer-controller \
  -n kube-system \
  --set clusterName=$EKS_CLUSTER_2 \
  --set serviceAccount.create=false \
  --set serviceAccount.name=aws-load-balancer-controller
```

Once the Helm chart has been applied, you should see:

```
AWS Load Balancer controller installed!
```

Deploy the sample application

cluster, let's deploy the sample application in the secondary cluster first:

Bash
```
cat > sample_application.yaml <<EOF
apiVersion: apps/v1
kind: Deployment
metadata:
  labels:
    app: py-az
  name: py-az
spec:
  replicas: 1
  selector:
    matchLabels:
      app: py-az
  template:
    metadata:
      labels:
        app: py-az
    spec:
      containers:
      - image: public.ecr.aws/b5x3e7x1/eks-py-az
```

Let's repeat the steps but on the primary cluster this time:

Bash
```
aws eks update-kubeconfig \
  --name $EKS_CLUSTER_1 \
  --region $AWS_REGION_1

kubectl apply -f sample_application.yaml
```

```
export Ingress_1=$(kubectl get ingress py-az-ingress \
  -o jsonpath='{.status.loadBalancer.ingress[0].hostname}')
```

You can curl the address of either of your ingresses and verify the response:

Bash
```
$ curl $Ingress_1
Welcome to us-east-1c <-- The response should originate from your primary region
```

Configure Global Accelerator

Now that the service works independently in each region, we have to direct all traffic to the primary region and route it to the failover region only if the primary region experiences issues. The service we will use to route traffic is AWS Global Accelerator.

AWS Global Accelerator is a networking service that sends your user's traffic through AWS's global network infrastructure, improving your internet user performance by up to 60%. It also makes it easier to operate multi-regional deployments by providing you with two static IPs that are anycast from AWS's globally distributed edge locations, giving you a single entry point to your application regardless of how many AWS Regions it's deployed in. Instead of relying on DNS for routing traffic at a global level (which is often problematic if you're dealing with clients that cache DNS results), Global Accelerator can switch traffic routes without requiring DNS changes, or delays caused by DNS propagation and client-side caching.

Let's start by creating an accelerator:

Bash
```
Global_Accelerator_Arn=$(aws globalaccelerator create-accelerator \
  --name multi-region \
  --query "Accelerator.AcceleratorArn" \
  --output text)
```

When you create an accelerator, Global Accelerator provisions two static IP addresses for you. It also assigns a default Domain Name System (DNS) name to your accelerator, similar to `a1234567890abcdef.awsglobalaccelerator.com`, that points to the two static IP addresses. The static IP addresses are advertised globally using anycast from the AWS edge network to your endpoints such as Network Load Balancers, Application Load Balancers, EC2 instances, or Elastic IP addresses.

Next, add a listener to the accelerator that will process inbound connections from clients on TCP port 80:

Bash
```
Global_Accelerator_Listerner_Arn=$(aws globalaccelerator create-listener \
  --accelerator-arn $Global_Accelerator_Arn \
  --region us-west-2 \
  --protocol TCP \
  --port-ranges FromPort=80,ToPort=80 \
  --query "Listener.ListenerArn" \
  --output text)
```

We now need to register the two ALBs we created earlier in the post as endpoints for the listener. An endpoint group routes requests to one or more registered endpoints in AWS Global Accelerator. Configure the endpoints group for the listener:

Bash
```
  EndpointGroupArn_1=$(aws globalaccelerator create-endpoint-group \
    --region us-west-2 \
    --listener-arn $Global_Accelerator_Listerner_Arn \
    --endpoint-group-region $AWS_REGION_1 \
    --query "EndpointGroup.EndpointGroupArn" \
    --output text \
    --endpoint-configurations EndpointId=$(aws elbv2 describe-load-balancers \
      --region $AWS_REGION_1 \
      --query "LoadBalancers[?contains(DNSName, '$Ingress_1')].LoadBalancerArn" \
      --output text),Weight=128,ClientIPPreservationEnabled=True)
```

Repeat the last step to add the endpoint for the failover region to the endpoint group:

Bash
```
  EndpointGroupArn_2=$(aws globalaccelerator create-endpoint-group \
    --region us-west-2 \
    --traffic-dial-percentage 0 \
    --listener-arn $Global_Accelerator_Listerner_Arn \
    --endpoint-group-region $AWS_REGION_2 \
    --query "EndpointGroup.EndpointGroupArn" \
    --output text \
    --endpoint-configurations EndpointId=$(aws elbv2 describe-load-balancers \
      --region $AWS_REGION_2 \
      --query "LoadBalancers[?contains(DNSName, '$Ingress_2')].LoadBalancerArn" \
      --output text),Weight=128,ClientIPPreservationEnabled=True)
```

Because we set the traffic-dial-percentage parameter to ‘0’ for the secondary region’s ALB, all traffic will be routed to the primary region. If the primary region’s endpoint (or its associated backends) fail health checks, Global Accelerator will route traffic to the ALB in the failover region. You can point your web browser to (or `curl`) the accelerator’s DNS name and you should receive a response from your primary region:

Bash

```
GA_DNS=$(aws globalaccelerator describe-accelerator \
  --accelerator-arn $Global_Accelerator_Arn \
  --query "Accelerator.DnsName" \
  --output text)

curl $GA_DNS

Welcome to us-east-1c%  <-- Response from the primary AWS region
```

Configure Route 53

Global Accelerator assigns a DNS name for accelerators, but you may want to use your own domain. In that case, you can use Amazon Route 53 to use a custom domain name with your accelerator.

Amazon Route 53 is a highly available and scalable cloud Domain Name System (DNS) web service. It offers new domain registration, but you can also use Route 53 to manage domains purchased through other registrars. Route 53 organizes your DNS records into "hosted zones" that include the routing logic for a domain (such as example.com) and all of its subdomains. A hosted zone has the same name as the corresponding domain. You can follow this guide if your domain is not managed by Route 53.

Get the hosted zone ID for your domain:

Bash

```
Route53_HostedZone=$(aws route53 list-hosted-zones \
  --query "HostedZones[?Name == '$my_domain.'].[Id]" \
  --output text | cut -d'/' -f 3)

echo $Route53_HostedZone
```

Create Route 53 records:

Bash

```
cat > route53-records.json<<EOF
{
  "Comment": "Multi region Global Accelerator",
  "Changes": [
    {
      "Action": "CREATE",
      "ResourceRecordSet": {
        "Name": "multi-region.$my_domain.",
        "Type": "A",
        "AliasTarget": {
          "HostedZoneId": "Z2BJ6XQ5FK7U4H",
          "DNSName": "$GA_DNS",
          "EvaluateTargetHealth": false
        }
      }
     }
   ]
}
```

Wait a few seconds for DNS changes to propagate and then you can check your service:

Bash
```
$ curl multi-region.$my_domain
Welcome to us-east-1c <-- Response from the primary AWS Region
```

You should see a response originating from your primary AWS Region. Route 53 will route all requests to Global Accelerator, which in turn will send traffic to the EKS cluster in the primary AWS Region, until the primary region health checks. When that happens, Global Accelerator will direct traffic to the failover region.

Simulate a failover

AWS Global Accelerator automatically checks the health of your applications and routes user traffic only to healthy application endpoints. If the health status changes or you make configuration updates, AWS Global Accelerator reacts instantaneously to route your users to the next available endpoint.

Let’s terminate the pods running in the primary EKS cluster. Doing this will leave the service without any backends and the ALB will return a *502 Bad Gateway* error. First, send a request to the ALB in the primary region and verify that you get a response successfully.

Bash
```
curl $Ingress_1
Welcome to us-east-1c% <-- Response from the primary AWS region
```

Now, scale down the deployment in the primary region:

Bash
```
aws eks update-kubeconfig \
  --name $EKS_CLUSTER_1 \
  --region $AWS_REGION_1

kubectl scale deployment py-az --replicas=0
```

`curl` the primary ALB again and you will see a 502 error:

Bash
```
curl $Ingress_1
<html>
<head><title>502 Bad Gateway</title></head>
<body>
<center><h1>502 Bad Gateway</h1></center>
</body>
</html>
```

Wait for Global Accelerator's health checks to fail (in our lab testing, failover took about thirty seconds) and resend a request to the service's A name.

Bash
```
curl multi-region.$my_domain
Welcome to us-east-2b% <-- Response from the failover AWS region
```

This time you should see a response from the failover region. If you scale up the deployment in the primary region, after a few minutes Global Accelerator will send all requests to the primary region again.

Cleanup

Use the following commands to delete resources created during this post:

Bash
```
aws globalaccelerator delete-endpoint-group --endpoint-group-arn $EndpointGroupArn_2
aws globalaccelerator delete-endpoint-group --endpoint-group-arn $EndpointGroupArn_1
aws globalaccelerator delete-listener --listener-arn $Global_Accelerator_Listerner_Arn
aws globalaccelerator update-accelerator --accelerator-arn $Global_Accelerator_Arn --n
# You may have to wait a few seconds until the accelerator is disabled
aws globalaccelerator delete-accelerator --accelerator-arn $Global_Accelerator_Arn
sed -i 's/CREATE/DELETE/g' route53-records.json
aws route53 change-resource-record-sets --hosted-zone-id $Route53_HostedZone --change-
aws eks update-kubeconfig --name $EKS_CLUSTER_1 --region $AWS_REGION_1
helm delete aws-load-balancer-controller -n kube-system
kubectl delete -f sample_application.yaml
aws eks update-kubeconfig --name $EKS_CLUSTER_2 --region $AWS_REGION_2
helm delete aws-load-balancer-controller -n kube-system
kkubectl delete -f sample_application.yaml
```

Conclusion

We showed you how to use Amazon EKS to run applications in multiple regions and use AWS Global Accelerator to route traffic. You can improve the resilience of your multi-regional application by configuring Global Accelerator's health checks to detect failures and route traffic to a failover region automatically.

Even though we have used this architecture to focus on multi-regional deployments primarily, you can also morph this architecture to distribute traffic to workloads running in multiple EKS clusters in different accounts, Virtual Private Clouds (VPCs), or even in the same AWS Region.

TAGS: EKS, k8s, multi-region