Containers

Optimize IP addresses usage by pods in your Amazon EKS cluster

by Umesh Ramesh | on 25 SEP 2020 | in Amazon Elastic Kubernetes Service, Containers | Permalink | Comments | Share

Many enterprise customers adopt multi-account strategy to meet their business needs and at the same time reduce the security blast radius. Customers have had problems maintaining network topology because of constant growth and increased workloads. They can quickly run out of IP space while planning out the VPC Classless Inter-Domain Routing (CIDR). In this blog, we briefly explain two solution options to address this problem, one using custom networking and another without.

Things you should know

Kubernetes: Kubernetes is a popular open source orchestration system of deploying and managing containerized applications, at scale.

Amazon EKS: Amazon Elastic Kubernetes Service (Amazon EKS) is a managed service that makes it easy for you to run Kubernetes on AWS without needing to stand up or maintain your own infrastructure. This fully managed service is designed to provide scalability, security, and integrations with other AWS services like Amazon Elastic Container Registry (Amazon ECR), IAM, and Elastic Load Balancer, to name a few.

AWS VPC CNI plugin: The AWS VPC CNI networking plugin provides native networking capabilities for pods running inside a Virtual Private Cloud (VPC). This plugin is responsible for assigning IPs to the pods whenever the pods are created. In order to assign an IP, the plugin needs to be aware of the underlying instance's ENI capacity and know how many primary and secondary interfaces are supported by the particular instance type.

Customer scenarios:

There are customers with tens or hundreds of accounts and thousands of workloads. As customers scale the EKS clusters and deploy more workloads, the number of pods managed by a cluster can grow easily above thousands of pods. Now, each of these pods will consume an IP address. This scenario might become challenging as the availability of IP addresses on a VPC is limited and it's not always possible to recreate a larger VPC or extend the current VPC's CIDR blocks. It's worth noting that both the worker nodes and the pods themselves require IP addresses. Also, the CNI will reserve IP addresses for future use.

Now, if you have pods executing in your EKS cluster and are already running out of IP addresses on your primary VPC CIDR (e.g. 10.0.0.0/20), we have two options:

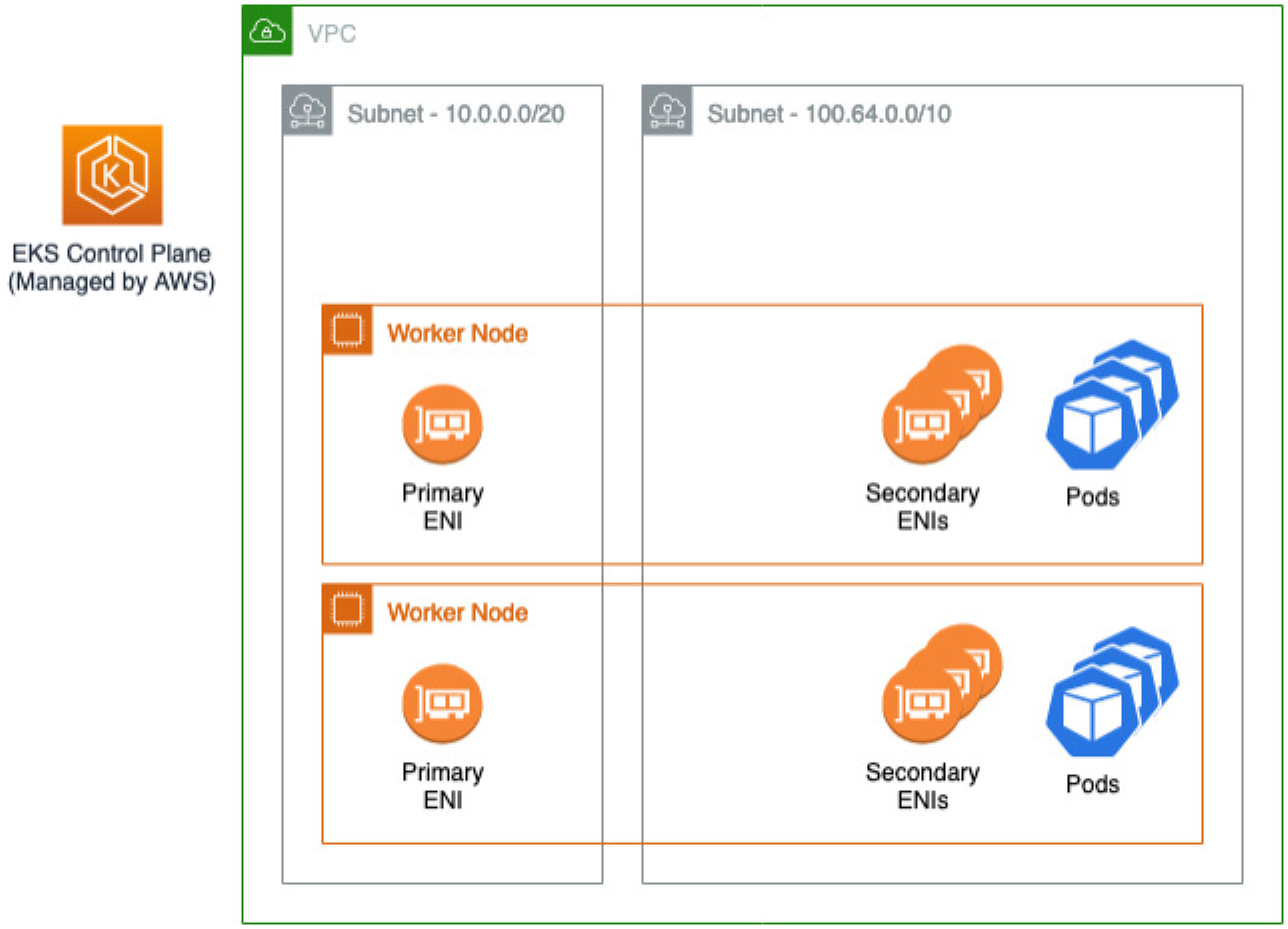
- CNI custom networking: this option uses the custom networking plugin to maintain your workers nodes' IP addresses on your primary VPC CIDR (in this example, 10.0.0.0/20), but move your pods' IP addresses to a larger subnet (e.g. 100.64.0.0/8). In this scenario, you can move all your pods to the new range and still use your 10.0.0.0/20 CIDR IP addresses as the source IP. However, here are some considerations you should know while using this configuration:
  - This solution will result in getting lesser IP addresses for the pods.

- The solution is comparatively complex as it involves manual calculation & configuration of "max pods."
- The solution is not supported by EKS managed node groups.

- Secondary CIDR with VPC: another option is to deploy new worker nodes with both the instance and pods networking on a new larger CIDR block (e.g. 100.64.0.0/8). In this scenario, after adding the new CIDR to your VPC, you can deploy another node group using the secondary CIDR and drain the original nodes to automatically redeploy the pods onto the new worker nodes.

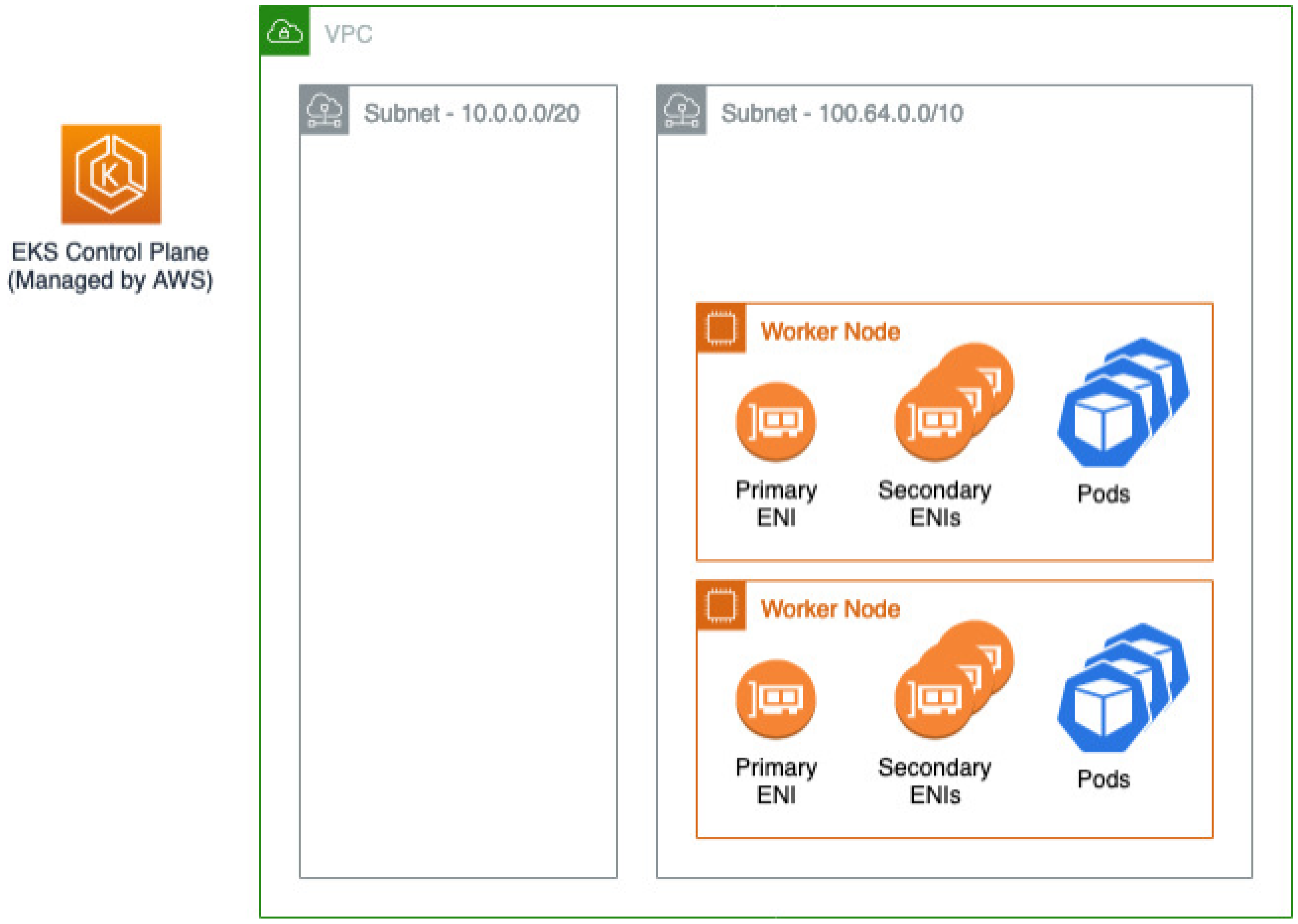
Option 1: CNI custom networking

As shown in the diagram below, the primary Elastic Network Interface (ENI) of the worker node still uses the primary VPC CIDR range (in this case 10.0.0.0/20) but the secondary ENIs for pods use the secondary VPC CIDR Range (in this case 100.64.0.0/10). Now, in order to have the pods using the 100.64.0.0/8 CIDR range, you will have to configure the CNI plugin to use custom networking. You can follow through the steps as documented here. You can also use a script published in this blog to execute and configure the plugin effortlessly.



Option 2: Unique network for nodes and pods.

In this scenario, you can still maintain the EKS cluster as is (the control plane will still live on the original subnet/s) but you'll completely migrate the instances and the pods to a secondary subnet/s. In the example shown on the diagram below, both the primary and the secondary ENIs use the secondary VPC CIDR Range (in this case 100.64.0.0/10).



How to use secondary CIDR with EKS (Option 2):

The following steps help you set up an Amazon EKS cluster and the managed worker nodes (data plane), followed by steps to drain worker nodes and move the pods to new worker nodes that you planned to retain. We have used Amazon AWS CLI to execute all the steps, which can also be easily automated in your CICD pipeline. You can execute these steps from your local machine, by logging into an Amazon EC2 instance or AWS SSM Session Manager.

Step 1: let's begin by creating Amazon Virtual Private Cloud (VPC) and subnets where we can deploy Amazon EKS clusters. If you plan on using an existing VPC, you can skip this step and directly jump to step 2.

Set the environment variables like CLUSTER_NAME and KEY_NAME that can used for subsequent commands

Bash
```bash
export CLUSTER_NAME=eks-cluster
export KEY_NAME=<an existing Keypair>
```

This CloudFormation stack will deploy a VPC and private subnets on the 10.0.0.0/20 CIDR block. Save the below CloudFormation script as eks-vpc.yaml and execute the following AWS CLI command to deploy the stack.

YAML

```
  NatGateway3:
    Type: AWS::EC2::NatGateway
    Properties:
      AllocationId: !GetAtt NatGateway3EIP.AllocationId
      SubnetId: !Ref PublicSubnet3
      Tags:
        - Key: Name
          Value: !Sub ${EnvironmentName}-ngw-3

  PublicRouteTable:
    Type: AWS::EC2::RouteTable
    Properties:
      VpcId: !Ref VPC
      Tags:
        - Key: Name
```

Bash

```
aws cloudformation create-stack --stack-name $CLUSTER_NAME-vpc \
      --template-body file://eks-vpc.yaml \
      --parameters \
ParameterKey=EnvironmentName,ParameterValue=$CLUSTER_NAME
```

Step 2: You can create the Amazon EKS control plane in a number of ways by using tools like eksctl, cdk8s, or custom CloudFormation templates. For this example, we have gone ahead and used this CloudFormation stack by running the below AWS CLI command, which deploys the cluster on the subnets created in Step 1.

Save the below CloudFormation as eks-control-plane.yaml

YAML

```
      Value: !GetAtt 'ControlPlane.Endpoint'
    CertificateAuthorityData:
```

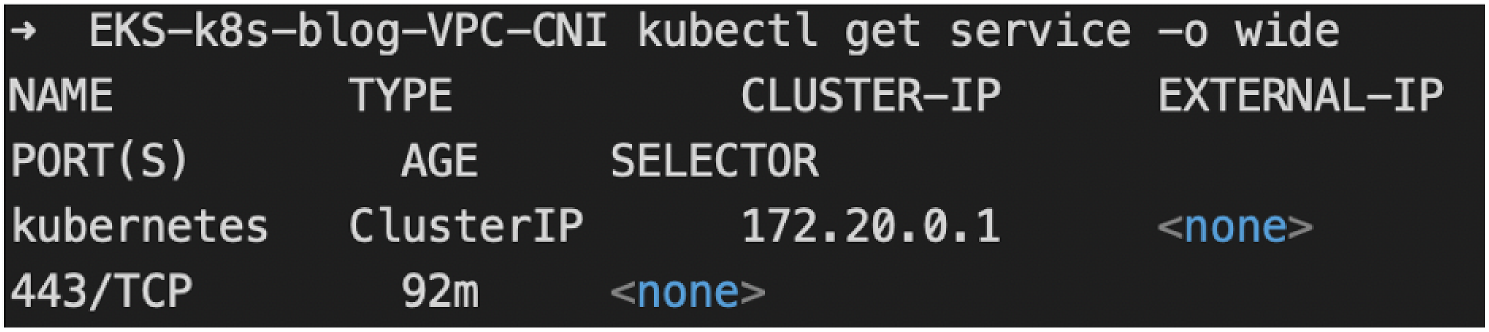
Execute the below commands to run the cloudformation stack.

Bash
```
export VPC_ID=$(aws cloudformation describe-stacks --stack-name $CLUSTER_NAME-vpc --qu

export SUBNETS_IDS=$(aws cloudformation describe-stacks --stack-name $CLUSTER_NAME-vpc

aws cloudformation create-stack --stack-name $CLUSTER_NAME-control-plane \
    --template-body file://eks-control-plane.yaml \
    --capabilities CAPABILITY_NAMED_IAM \
    --parameters \
    ParameterKey=Name,ParameterValue=$CLUSTER_NAME \
    ParameterKey=Vpc,ParameterValue=$VPC_ID \
'ParameterKey=Subnets,ParameterValue="'"$SUBNETS_IDS"'"'
```

Step 3: you will create the kubeconfig file for your cluster using the below command. By default, the resulting configuration file is created at the default kubeconfig path (.kube/config) in your home directory or merged with an existing kubeconfig at that location.

Bash
```
aws eks update-kubeconfig --name $CLUSTER_NAME
```

At this point, you can verify if your service is up using the command “kubectl get svc” as shown below.



```
➜  EKS-k8s-blog-VPC-CNI kubectl get service -o wide
NAME         TYPE          CLUSTER-IP     EXTERNAL-IP
PORT(S)        AGE      SELECTOR
kubernetes   ClusterIP     172.20.0.1     <none>
443/TCP        92m      <none>
```

Step 4: we will use the CloudFormation stack to create the data plane, which is the first set of managed worker node groups, using the below AWS CLI command to deploy the CloudFormation stack. In this command, we feed in some of the inputs such as:

- ClusterName, which is name of the cluster you used in step 1
- A node group name
- Node group auto scaling parameters like MinSize, MaxSize, and Desired Size
- The key pair name to be used for the worker nodes (EC2 instances)
- The security groups to protect the worker nodes
- Subnet id's where the worker nodes should be deployed and the VPC id

Save the below CloudFormation stack as eks-data-plane.yaml

YAML
```
        RemoteAccess:
          Ec2SshKey: !Ref 'KeyName'
          SourceSecurityGroups:
            - !Ref 'ProvidedSecurityGroup'
            - !Ref 'NodeSecurityGroup'
        ScalingConfig:
          MinSize: !Ref 'NodeGroupScalingConfigMinSize'
          DesiredSize: !Ref 'NodeGroupScalingConfigDesiredSize'
          MaxSize: !Ref 'NodeGroupScalingConfigMaxSize'
        Subnets: !Ref 'Subnets'
        Tags:
          Name: !Sub ${NodeGroupName}-NodeGroup
Outputs:
  NodeInstanceRole:
    Description: The node instance role
    Value: !GetAtt 'NodeInstanceRole.Arn'
  NodeSecurityGroup:
    Description: The security group for the node group
    Value: !Ref 'NodeSecurityGroup'
```

Bash
```
export CLUSTER_SG=$(aws eks describe-cluster --name $CLUSTER_NAME --query 'cluster.reso

export ADDITIONAL_SG=$(aws eks describe-cluster --name $CLUSTER_NAME --query 'cluster.r

aws cloudformation create-stack --stack-name $CLUSTER_NAME-data-plane \
--template-body file://eks-data-plane.yaml \
--capabilities CAPABILITY_NAMED_IAM \
--parameters \
ParameterKey=ClusterControlPlaneSecurityGroup,ParameterValue=$CLUSTER_SG \
ParameterKey=ClusterName,ParameterValue=$CLUSTER_NAME \
ParameterKey=NodeGroupName,ParameterValue=$CLUSTER_NAME-nodegroup \
ParameterKey=KeyName,ParameterValue=$KEY_NAME \
ParameterKey=VpcId,ParameterValue=$VPC_ID \
'ParameterKey=Subnets,ParameterValue="'"$SUBNETS_IDS"'"' \
ParameterKey=ProvidedSecurityGroup,ParameterValue=$ADDITIONAL_SG
```

Before deploying the application, you can use the below command to monitor and wait until all the nodes are in "Ready" state.

Bash
```
kubectl get nodes --watch
```

[illegible]

YAML
```
      ports:
      - containerPort: 80
---
apiVersion: v1
kind: Service
metadata:
  name: nginx
  namespace: default
  labels:
    app: nginx
spec:
  ports:
  - name: http
    port: 80
    protocol: TCP
    targetPort: 80
  selector:
    app: nginx
  type: LoadBalancer
```

Execute the below command to deploy the app.

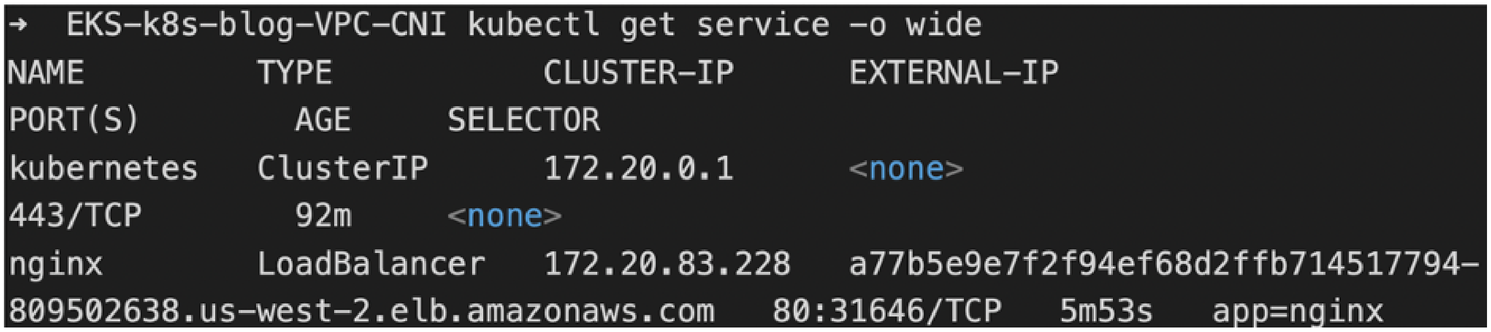
Bash
```
kubectl create -f deployment.yaml
```

This will take a few minutes after which the pods should be in running state. You can verify using the below command:

Bash
```bash
kubectl get Pods --all-namespaces
```

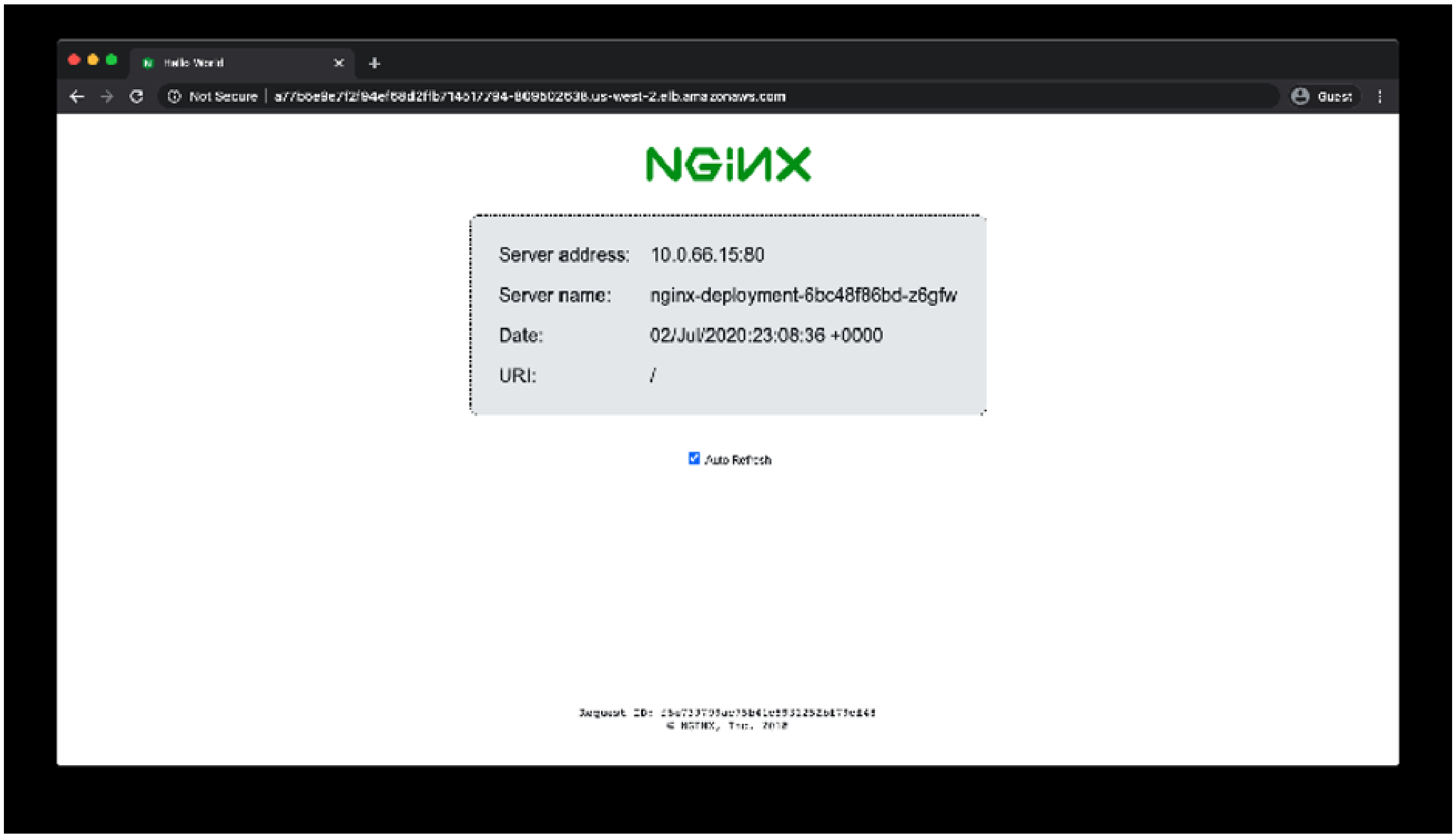
The below command will display the load balancer endpoint of the deployed application for testing:

Bash
```bash
kubectl get service -o wide
```



```
➜  EKS-k8s-blog-VPC-CNI kubectl get service -o wide
NAME         TYPE           CLUSTER-IP      EXTERNAL-IP
PORT(S)        AGE     SELECTOR
kubernetes   ClusterIP      172.20.0.1      <none>
443/TCP        92m     <none>
nginx        LoadBalancer   172.20.83.228   a77b5e9e7f2f94ef68d2ffb714517794-
809502638.us-west-2.elb.amazonaws.com   80:31646/TCP   5m53s   app=nginx
```

Now, when you open this endpoint on a browser, you will observe that the server is deployed on the 10.0.0.0/16 CIDR block.



Hello World
Not Secure | a77b5e9e7f2f94ef68d2ffb714517794-809502638.us-west-2.elb.amazonaws.com
Guest
NGiNX

Server address: 10.0.66.15:80

Server name: nginx-deployment-6bc48f86bd-z6gfw

Date: 02/Jul/2020:23:08:36 +0000

URI: /

☑ Auto Refresh

Request ID: [illegible]

© NGINX, Inc. 2018

Step 6: In this step, we will attach a new CIDR block to the VPC and create new subnets on the 100.64.0.0/16 CIDR block using this CloudFormation stack and the below AWS CLI commands.

You may save the below CloudFormation as eks-vpc-secondary.yaml

YAML
```
puts:

  PrivateSubnets:
    Description: A list of the private subnets
    Value: !Join [ ",", [ !Ref PrivateSubnet1, !Ref PrivateSubnet2 , !Ref PrivateSu

  PrivateSubnet1:
    Description: A reference to the private subnet in the 1st Availability Zone
    Value: !Ref PrivateSubnet1

  PrivateSubnet2:
    Description: A reference to the private subnet in the 2nd Availability Zone
    Value: !Ref PrivateSubnet2

  PrivateSubnet3:
    Description: A reference to the private subnet in the 3rd Availability Zone
    Value: !Ref PrivateSubnet3
```

Run the following commands to deploy the stack.

Bash
```
export NGW1=$(aws cloudformation describe-stacks --stack-name $CLUSTER_NAME-vpc --query

export NGW2=$(aws cloudformation describe-stacks --stack-name $CLUSTER_NAME-vpc --query

aws cloudformation create-stack --stack-name $CLUSTER_NAME-vpc-secondary \
 --template-body file://eks-vpc-secondary.yaml \
 --parameters \ ParameterKey=EnvironmentName,ParameterValue=$CLUSTER_NAME \
 ParameterKey=VpcId,ParameterValue=$VPC_ID \
 ParameterKey=NatGateway1,ParameterValue=$NGW1 \
 ParameterKey=NatGateway2,ParameterValue=$NGW2 \
 ParameterKey=NatGateway3,ParameterValue=$NGW3
```

Step 7: In this step, we will deploy the second set of managed worker nodes on the new subnets we created in the previous step using this CloudFormation and the below AWS CLI command:

You will be using the CloudFormation stack created in step 4 (eks-data-plane.yaml).

Bash
```
export SECONDARY_SUBNETS_IDS=$(aws cloudformation describe-stacks --stack-name $CLUSTE

aws cloudformation create-stack --stack-name $CLUSTER_NAME-data-plane-secondary \
--template-body file://eks-data-plane.yaml \
--capabilities CAPABILITY_NAMED_IAM \
--parameters \
ParameterKey=ClusterControlPlaneSecurityGroup,ParameterValue=$CLUSTER_SG \
ParameterKey=ClusterName,ParameterValue=$CLUSTER_NAME \
ParameterKey=NodeGroupName,ParameterValue=$CLUSTER_NAME-nodegroup-secondary \
ParameterKey=KeyName,ParameterValue=$KEY_NAME \
ParameterKey=VpcId,ParameterValue=$VPC_ID \
'ParameterKey=Subnets,ParameterValue="'"$SECONDARY_SUBNETS_IDS"'"' \
ParameterKey=ProvidedSecurityGroup,ParameterValue=$ADDITIONAL_SG
```

You can wait until the nodes are in “Ready” state using the below command:

Bash
```
kubectl get nodes --watch
```

Step 8: Now that we have the application running on subnets using 10.0.0.0/8 CIDR block, we will first issue the cordon command. This command stops scheduling any new pods to these worker nodes. We will then drain these worker nodes deployed in step 4, so that the pods automatically get terminated and recreated in other healthy worker nodes based on subnets using 100.64.0.0/16 CIDR range.

Command to cordon all the nodes running on 10.0.0.0/16 CIDR block.

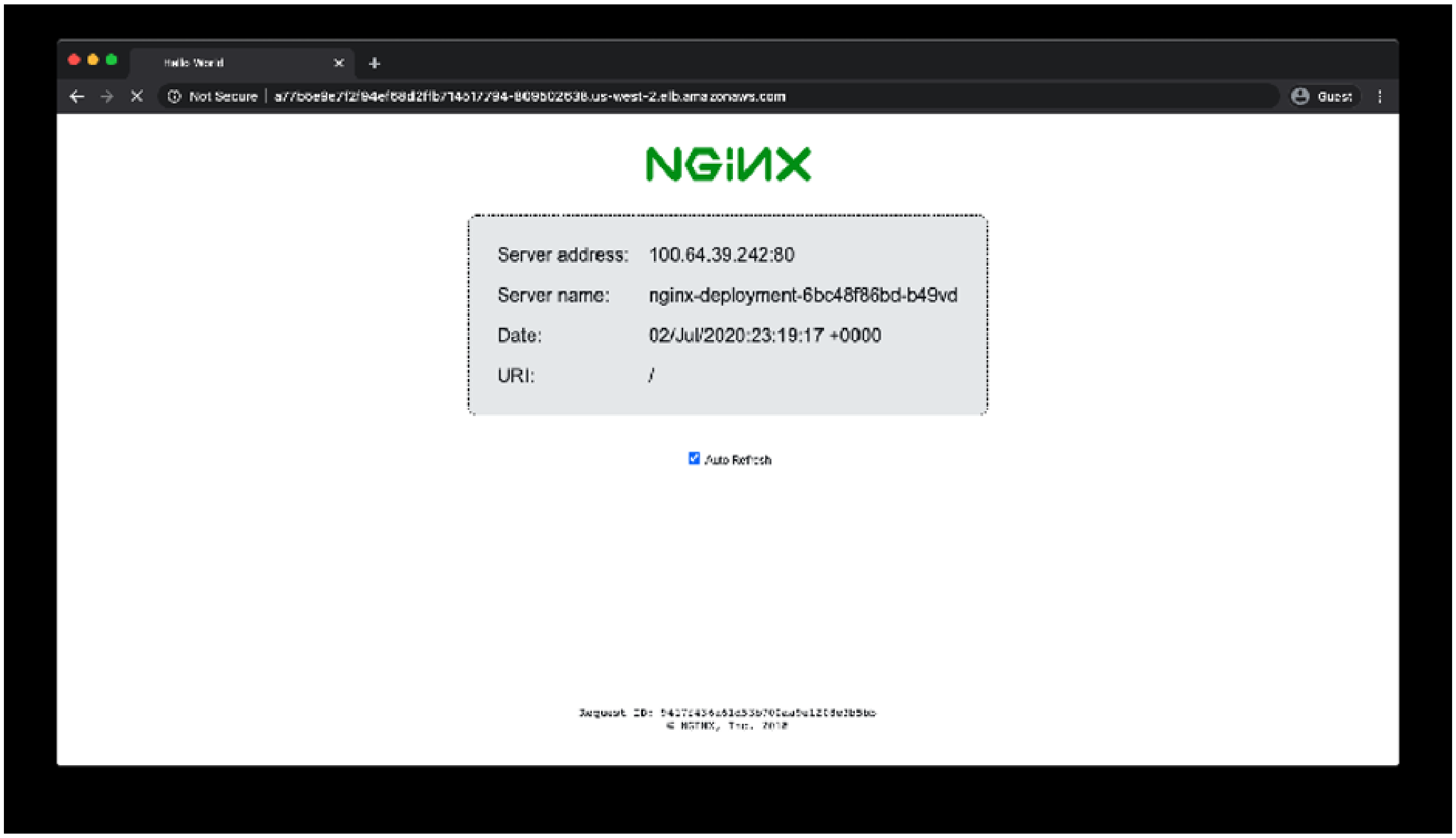
Bash
```
kubectl get nodes --no-headers=true | awk '/ip-10-0/{print $1}' | xargs kubectl cordon
```

Command to drain all the nodes running on 10.0.0.0/16 CIDR block. You need to use –ignore-daemonsets flag in order to drain nodes with daemonsets and use –delete-local-data flag to overide and delete any pods that utilize an emptyDir volume.

Bash
```
kubectl get nodes --no-headers=true | awk '/ip-10-0/{print $1}' | xargs kubectl drain
```

You can verify that the endpoint is now running on 100.64.0.0/16 CIDR block by accessing the endpoint again using the below command:

Bash
```bash
kubectl get service -o wide
```


Hello World
Not Secure | a77b0e9e7f2f94ef68d2f1b714517794-809502638.us-west-2.elb.amazonaws.com
Guest
NGINX
Auto Refresh

Server address: 100.64.39.242:80
Server name: nginx-deployment-6bc48f86bd-b49vd
Date: 02/Jul/2020:23:19:17 +0000
URI: /

Request ID: [illegible]
© NGINX, Inc. 2018

Conclusion

You should now be able to choose the most appropriate solution if you are running into an issue of limited IP addresses in your existing VPC CIDR range. Also, it is very important to plan out your VPC CIDR ranges across your multiple accounts to make sure you do not have overlaping IP addresses requiring complex NATing resolutions. We highly recommend reading this blog that explains various networking patterns, especially for customers having hybrid environments with connectivity to their data centers.

Further reading references:

If you are new to Kubernetes, this should help you understand the basics. Also, you can check our documentation if you are new to Amazon EKS. With the adoption of Kubernetes, Jeremy Cowan explains in this blog on how the problem IP address shortage escalates with increase in workload and also provides a solution to this problem.

About the Authors:

Jose Olcese

Jose is a Principal Cloud Application Architect with Amazon Web Services where he helps customers build cutting-edge, cloud-based solutions. Jose has over 20 years of experience in software development for a variety of different industries and has helped hundreds of customers to integrate Identity solutions with AWS. Outside of work, Jose enjoys spending time with his family, running and building things.

Umesh Kumar Ramesh

Umesh is a Sr. Cloud Infrastructure Architect with AWS who delivers proof-of-concept projects, topical workshops, and leads implementation projects. He holds a Bachelor's degree in Computer Science & Engineering from the National Institute of Technology, Jamshedpur (India). Outside of work, he enjoys watching documentaries, biking, practicing meditation and discuss spirituality.

TAGS: EKS

Comments

0 Comments 4 **Prashanth ▾**

Start the discussion...

♡ **Share** **Best** **Newest** **Oldest**

Be the first to comment.

Subscribe **Privacy** **Do Not Sell My Data**