1. [**Resource Lifecycle**](#_vy7feur2ne2w) **2**
2. [Lifecycle](#_u8jaw7r1bzu4) 2
3. [Partial State and Error Handling](#_8si401koe34p) 2
4. [**Cluster Provision:**](#_v8p98wub599i) **3**
5. [AWS STS token generation for Gitlab and AWS CLI](#_86dto6lywqee) 3
6. [Update secret with CI/CD variables and AWS CLI](#_kruufvfis29c) 3
7. [Run pipeline of your cluster provision project](#_6wulm1o3aogm) 4
8. [Trust for Kubernetes Provider](#_wib68ln0eiok) 7
9. [Create KubeConfig for AWS EKS [through command prompt]](#_bc0clzs9w8lz) 7
10. [Create KubeConfig Manually [If any challenges on command prompt]](#_1k63klutmias) 8
11. [Create a kubeconfig file dynamically [Prefered Approach]](#_54msncfhaht2) 10
12. [**Check the cluster health**](#_ypbl1s4pmz) **10**
13. [Before Auto Scaling Cluster Creation](#_sox5xxpp8rd6) 10
14. [After Auto Scaling Cluster Created](#_evb747cdrvtu) 11
15. [**Namespaced objects Vs. Cluster-wide objects**](#_48vumphl0eph) **13**
16. [**Frequently used Kubectl commands**](#_7omiebf43ply) **14**
17. [Source of ‘kubectl explain’ output](#_cxihxo5v8m61) 17
18. [**Evicted Pod**](#_kqernpvgno2e) **17**
19. [Why bother deleting an Evicted Pod?](#_6fl4sua22wn3) 17
20. [But what if you have 100 Evicted pods?](#_tht966ptvoxr) 17
21. [**Metrics Server**](#_g0d1rgd7ksup) **18**
22. [Installing the Kubernetes Metrics Server](#_x8abykpxwvno) 18
23. [**Image Management**](#_yvf1gryem81c) **23**
24. [Purging All Unused or Dangling Images, Containers, Volumes, and Networks](#_wuyx3hp4bx4) 23
25. [docker pull](#_qnyvuzo425as) 23
26. [Convert image from local repository to file system](#_l8yawwwa8l28) 23
27. [**Cluster Autoscaler**](#_z9beapjptls5) **24**
28. [**Enabling IAM user and role access to your cluster**](#_smhewcxbxtb) **24**
29. [**How to Expose the cluster service**](#_5oiv4y6la3gd) **25**
30. [Get the Metrics through endpoint in raw format](#_c3sml6wmygum) 25
31. [**Troubleshoot in your clusters:**](#_vj1tmqx02ubs) **26**
32. [Troubleshoot the pod:](#_197mpnoe3606) 26
33. [Encoded Error Message](#_i10w96z0r4ea) 26
34. [Pods in the pending state](#_ogghxh9q4tu6) 26
35. [Reason for Pending state](#_8e6dj2kp0zoe) 27
36. [**Troubleshoot the memory and storage**](#_5ozcrzczd6mg) **29**
37. [**Prometheus Introduction**](#_9ltcsuxw90v9) **30**
38. [Prometheus Architecture](#_hdesfvs81ddg) 30
39. [MONITORING LINUX HOST METRICS WITH THE NODE EXPORTER](#_k5ewjmns5n7j) 31
40. [Exploring Node Exporter metrics through the Prometheus expression browser](#_b858gm2gc6wt) 31
41. [**Built-in Grafana Dashboard**](#_tqp2rdsr6dyo) **32**
42. [**Create a Docker Credentials file**](#_j15q8jtm8ovy) **33**
43. [Create a Kubernetes Secret](#_etkkikzjwlh) 34
44. [**Reference:**](#_cip4zimpzsdo) **34**

# Resource Lifecycle

Resources have a strict lifecycle, and can be thought of as basic state machines. Understanding this lifecycle can help better understand how Terraform generates an execution plan, how it safely executes that plan, and what th

, me resource provider is doing throughout all of this.

## Lifecycle

A resource roughly follows the steps below:

1. **ValidateResource** is called to do a high-level structural validation of a resource's configuration. The configuration at this point is raw and the interpolations have not been processed. The value of any key is not guaranteed and is just meant to be a quick structural check.
2. **Diff** is called with the current state and the configuration. The resource provider inspects this and returns a diff, outlining all the changes that need to occur to the resource. The diff includes details such as whether or not the resource is being destroyed, what attribute necessitates the **destroy**, old values and new values, whether a value is computed, etc. It is up to the resource provider to have this knowledge.
3. **Apply** is called with the current state and the diff. Apply does not have access to the configuration. This is a safety mechanism that limits the possibility that a provider changes a diff on the fly. Apply must apply a diff as prescribed and do nothing else to remain true to the Terraform execution plan. Apply returns the new state of the resource (or nil if the resource was destroyed).
4. If a resource was just created and did not exist before, and the apply succeeded without error, then the provisioners are executed in sequence. If any provisioner errors, the resource is marked as tainted, so that it will be destroyed on the next apply.

## **Partial State and Error Handling**

If an error happens at any stage in the lifecycle of a resource, Terraform stores a partial state of the resource. This behavior is critical for Terraform to ensure that you don't end up with any zombie resources: resources that were created by Terraform but no longer managed by Terraform due to a loss of state.

$ terraform init

$ terraform plan -out "outfile"

$ terraform apply "outfile"

$ terraform init

$ terraform plan

$ terraform apply -auto-approve

# Cluster Provision:

## AWS STS token generation for Gitlab and AWS CLI

Prerequisites: latest version of AWS CLI should be installed in the server where gitlab runner is running. [For my case] Group Gitlab runner is installed on my laptop with the tag of devsecops. Since we have enabled MFA, we need to generate a token for every session (12 hours validity).

Command:

**$aws sts get-session-token --serial-number arn:aws:iam::594977439087:mfa/sathiyan.sivaprakasam@synapsica.com --token-code 385749**

Output:

{

"Credentials": {

"AccessKeyId": "ASIAYVB3LHVX46IXXKOE",

"SecretAccessKey": "z45wqE/XpWK7l7BTIdTuCjucf4tyHpzGXTYFLevm",

"SessionToken": "IQoJb3JpZ2luX2VjEA0aCXVzLWVhc3QtMSJHMEUCIQDHpztQdFagIbIFrUvGJXw/YygLViyDTEIe67iM7btPFgIgZ0wzyqde6UScbMvCzPbAe57qw4U1Q2lmACPZz6G0GrUq+AEIhv//////////ARADG",

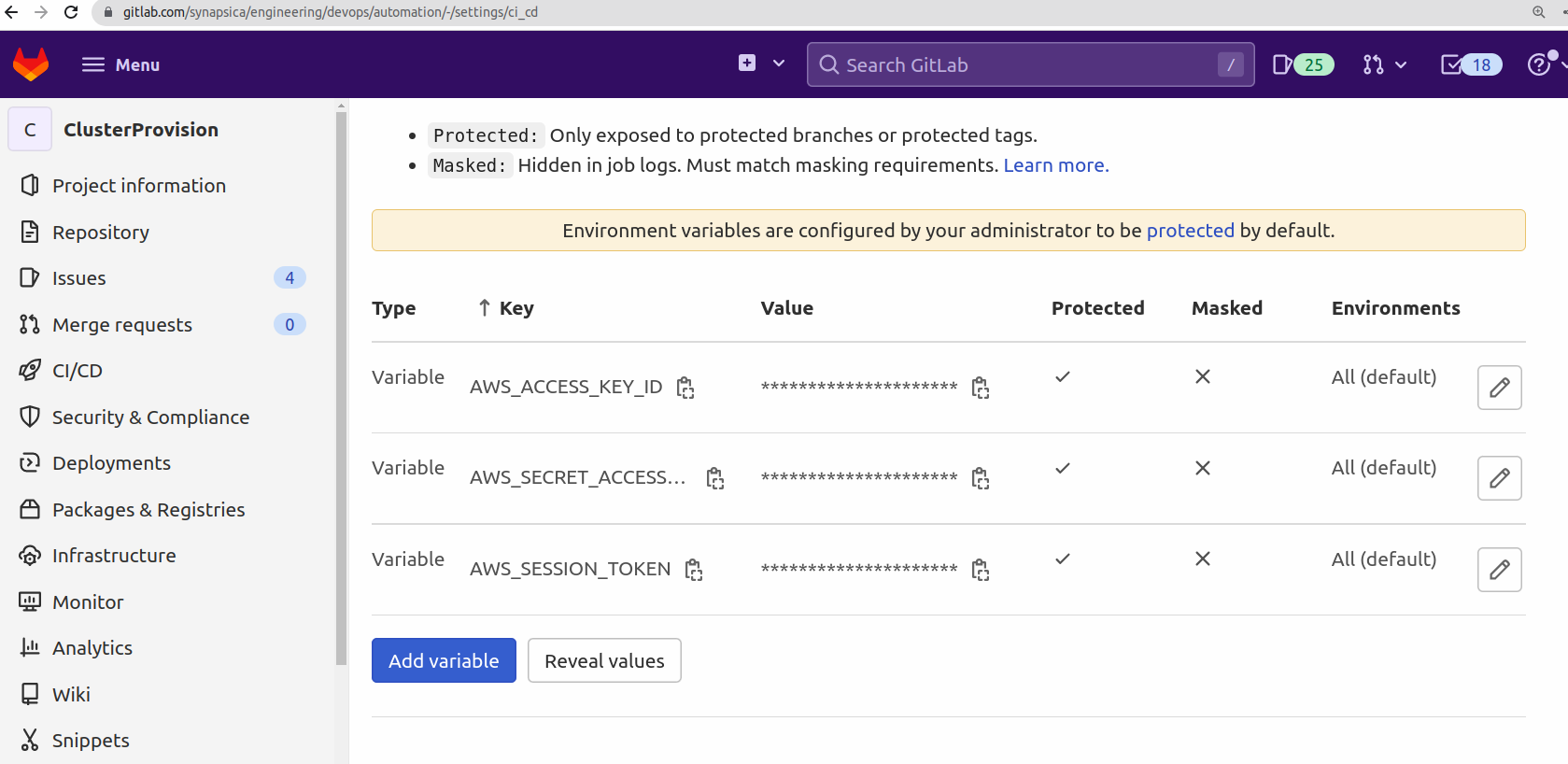
"Expiration": "2022-08-23T16:38:50+00:00"

}

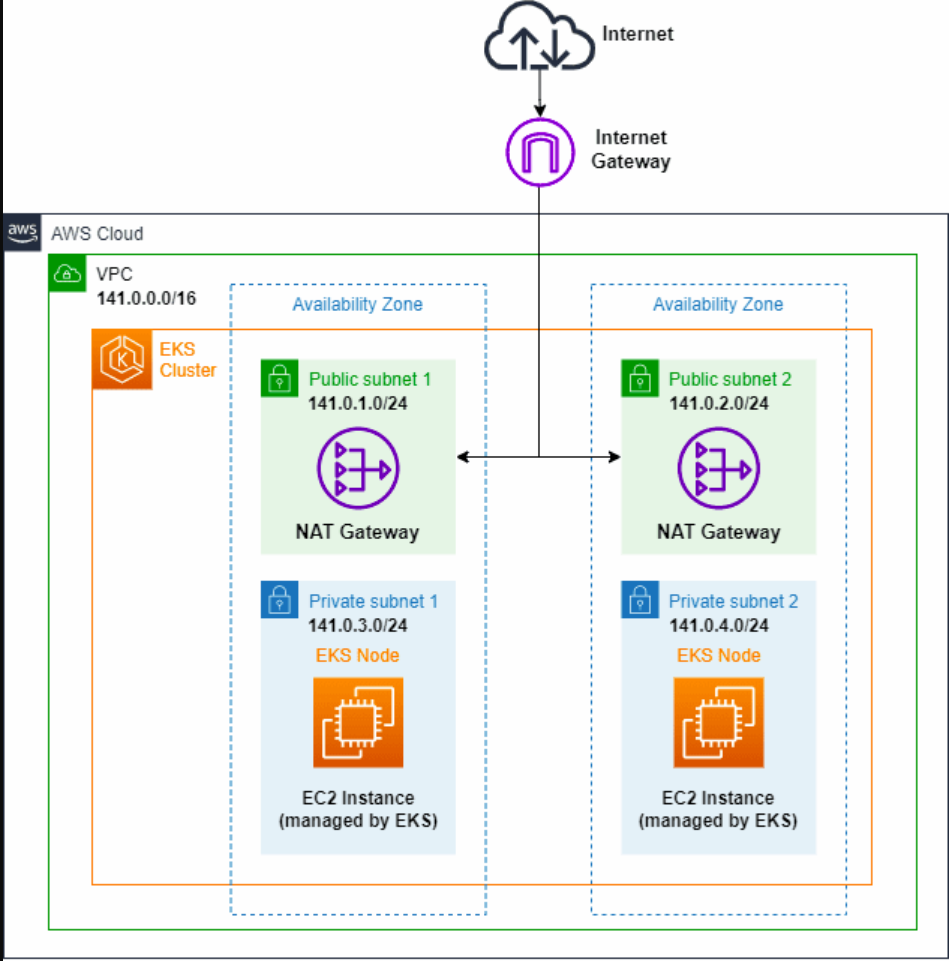
}

## Update secret with CI/CD variables and AWS CLI

Update the above values with CI/CD variables like below.

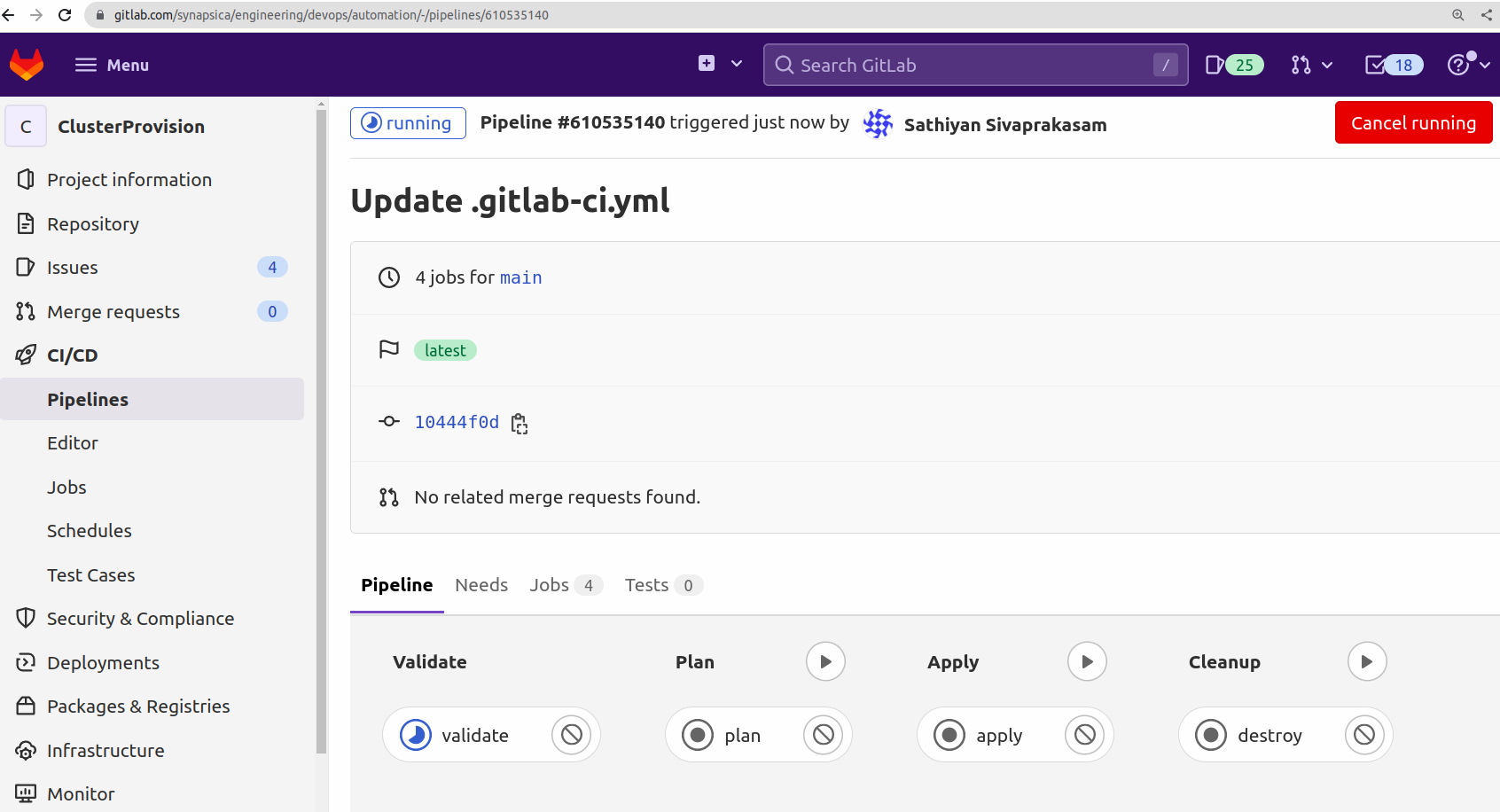


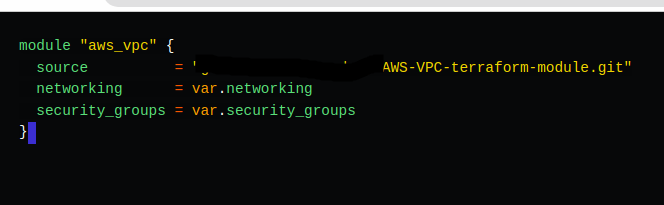
Prerequisites: You need to have privilege for “clusterprovision” project in devops group. You also need to update those values with ~/.aws/credentials also along with some profile name [for instance, profile = mfa].



## Run pipeline of your cluster provision project

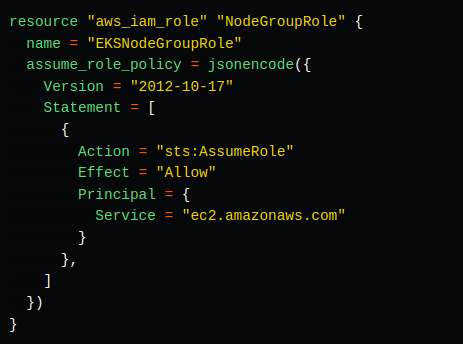
Click on “Run Pipeline”. It executes the .gitlab-ci.yml with the following pipeline.

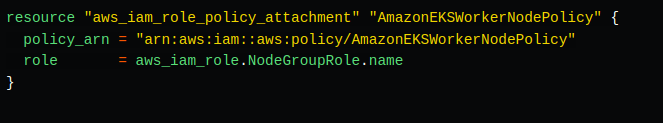


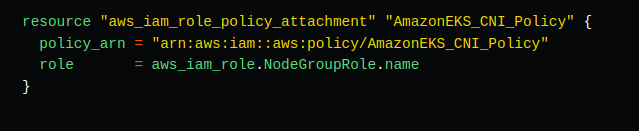


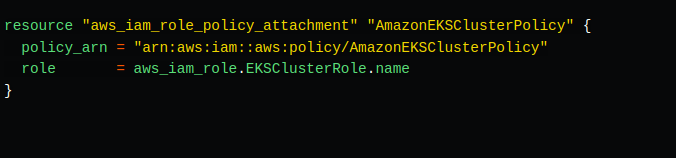


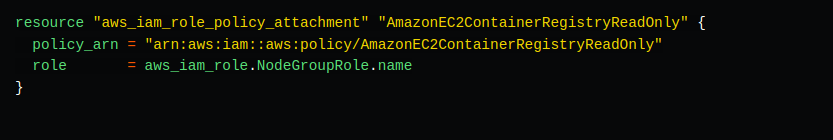












* 1. 

## Trust for Kubernetes Provider

1. provider "kubernetes" {
2. host = data.aws\_eks\_cluster.cluster.endpoint
3. token = data.aws\_eks\_cluster\_auth.cluster.token
4. cluster\_ca\_certificate = base64decode(data.aws\_eks\_cluster.cluster.certificate\_authority.0.data)
5. }
6. or
7. provider "kubernetes" {
8. config\_path = "../eks\_setup/spindle-ai-eks-terraform-cluster\_config"
9. }

### decode the secret

* 1. Just use the base64 program from the coreutils package:
  2. echo QWxhZGRpbjpvcGVuIHNlc2FtZQ== | base64 --decode

## Create KubeConfig for AWS EKS [through command prompt]

**$ aws eks update-kubeconfig --region us-east-1 --name eks-cluster --profile mfa**

Output:

Added new context arn:aws:eks:us-east-1:594977439087:cluster/test-eks-cluster-Mw6qwFqF to /home/sathiyan/.kube/config

# 

# 

# 

# 

# 

## Create KubeConfig Manually [If any challenges on command prompt]

To create your kubeconfig file manually

1. Set values for a few variables by replacing the example values with your own and then running the modified commands.

export region\_code=region-code

export cluster\_name=my-cluster

export account\_id=111122223333

1. Retrieve the endpoint for your cluster and store the value in a variable.

cluster\_endpoint=$(aws eks describe-cluster \

--region $region\_code \

--name $cluster\_name \

--query "cluster.endpoint" \

--output text)

1. Retrieve the Base64-encoded certificate data required to communicate with your cluster and store the value in a variable.

certificate\_data=$(aws eks describe-cluster \

--region $region\_code \

--name $cluster\_name \

--query "cluster.certificateAuthority.data" \

--output text)

1. Create the default ~/.kube directory if it doesn't already exist.

mkdir -p ~/.kube

1. Run the command for your preferred client token method (AWS CLI or AWS IAM authenticator for Kubernetes) to create the config file in the ~/.kube directory. You can specify the following before running one of the commands by modifying the command to include the following:
   * An IAM role – Remove the # at the start of the lines under args:. Replace my-role with the name of the IAM role that you want to perform cluster operations with instead of the default AWS credential provider chain.
   * An AWS CLI named profile – Remove the # at the start of the env: line, and remove # at the start of the lines under it. Replace aws-profile with the name of the profile to use. If you don't specify a profile, then the default profile is used.
   * AWS CLI
   * AWS IAM Authenticator for Kubernetes

Prerequisite

Version 1.16.156 or later of the AWS CLI must be installed on your device.

#!/bin/bash

read -r -d '' KUBECONFIG <<EOF

apiVersion: v1

clusters:

- cluster:

certificate-authority-data: $certificate\_data

server: $cluster\_endpoint

name: arn:aws:eks:$region\_code:$account\_id:cluster/$cluster\_name

contexts:

- context:

cluster: arn:aws:eks:$region\_code:$account\_id:cluster/$cluster\_name

user: arn:aws:eks:$region\_code:$account\_id:cluster/$cluster\_name

name: arn:aws:eks:$region\_code:$account\_id:cluster/$cluster\_name

current-context: arn:aws:eks:$region\_code:$account\_id:cluster/$cluster\_name

kind: Config

preferences: {}

users:

- name: arn:aws:eks:$region\_code:$account\_id:cluster/$cluster\_name

user:

exec:

apiVersion: client.authentication.k8s.io/v1beta1

command: aws

args:

- --region

- $region\_code

- eks

- get-token

- --cluster-name

- $cluster\_name

# - "- --role-arn"

# - "arn:aws:iam::$account\_id:role/my-role"

# env:

# - name: "AWS\_PROFILE"

# value: "aws-profile"

EOF

echo "${KUBECONFIG}" > ~/.kube/config

1. Add the file path to your KUBECONFIG environment variable so that kubectl knows where to look for your cluster configuration.
   * For Bash shells on macOS or Linux:

export KUBECONFIG=$KUBECONFIG:~/.kube/config

* + For PowerShell on Windows:

$ENV:KUBECONFIG="{0};{1}" -f $ENV:KUBECONFIG, "$ENV:userprofile\.kube\config"

1. (Optional) Add the configuration to your shell initialization file so that it is configured when you open a shell.
   * For Bash shells on macOS:

echo 'export KUBECONFIG=$KUBECONFIG:~/.kube/config' >> ~/.bash\_profile

* + For Bash shells on Linux:

echo 'export KUBECONFIG=$KUBECONFIG:~/.kube/config' >> ~/.bashrc

1. Test your configuration.

kubectl get svc

## Create a kubeconfig file dynamically [Prefered Approach]

We can create a kubeconfig file dynamically with the help of a template in eks\_setup module after creating the cluster successfully.

$ export KUBECONFIG=$(pwd)/spindle-ai-eks-terraform-cluster\_config

**$ export KUBECONFIG=../eks\_setup/spindle-ai-eks-terraform-cluster\_config**

**or**

**export KUBECONFIG=~/assignments/devsecops/iac/spindle-scaler/prod/eks/spindle-radiolens-monitor-cluster\_config**

# Check the cluster health

## Before Auto Scaling Cluster Creation

**$aws autoscaling describe-auto-scaling-groups --query "AutoScalingGroups[? Tags[? (Key=='k8s.io/cluster-autoscaler/enabled') && Value=='true']]".AutoScalingGroupName --region us-east-1 --profile mfa**

Output:

[

]

initially empty array for us.

## After Auto Scaling Cluster Created

**$ aws autoscaling --region us-east-1 describe-auto-scaling-groups --profile mfa**

{

"AutoScalingGroups": [

{

"AutoScalingGroupName": "test-eks-cluster-GDl9F3iW-worker-group-120220812100108383000000016",

"AutoScalingGroupARN": "arn:aws:autoscaling:us-east-1:594977439087:autoScalingGroup:9472abf2-dab3-4bf9-81ec-026a440da03c:autoScalingGroupName/test-eks-cluster-GDl9F3iW-worker-group-120220812100108383000000016",

"LaunchConfigurationName": "test-eks-cluster-GDl9F3iW-worker-group-120220812100058478200000014",

"MinSize": 1,

"MaxSize": 3,

"DesiredCapacity": 2,

"DefaultCooldown": 300,

"AvailabilityZones": [

"us-east-1a",

"us-east-1b",

"us-east-1c"

],

"LoadBalancerNames": [],

"TargetGroupARNs": [],

"HealthCheckType": "EC2",

"HealthCheckGracePeriod": 300,

"Instances": [

{

"InstanceId": "i-0a767af5e7d681990",

"InstanceType": "t2.small",

<<more key pair>>

"test-eks-cluster-GDl9F3iW-worker-group-120220812100058478200000014",

"ProtectedFromScaleIn": false

},

{

"InstanceId": "i-0b3514de2aeb49bde",

"InstanceType": "t2.small",

"AvailabilityZone": "us-east-1c",

"LifecycleState": "InService",

"HealthStatus": "Healthy",

"LaunchConfigurationName": "test-eks-cluster-GDl9F3iW-worker-group-120220812100058478200000014",

"ProtectedFromScaleIn": false

}

],

"CreatedTime": "2022-08-12T10:01:08.917000+00:00",

"SuspendedProcesses": [

{

"ProcessName": "AZRebalance",

"SuspensionReason": "User suspended at 2022-08-12T10:02:56Z"

}

],

"VPCZoneIdentifier": "subnet-0eedea74271ec7c2b,subnet-0e954ce7869db4a52,subnet-030a54f66b4b44890",

"EnabledMetrics": [],

"Tags": [

{

"ResourceId": "test-eks-cluster-GDl9F3iW-worker-group-120220812100108383000000016",

"ResourceType": "auto-scaling-group",

"Key": "Name",

"Value": "test-eks-cluster-GDl9F3iW-worker-group-1-eks\_asg",

"PropagateAtLaunch": true

},

{

"ResourceId": "test-eks-cluster-GDl9F3iW-worker-group-120220812100108383000000016",

"ResourceType": "auto-scaling-group",

"Key": "k8s.io/cluster/test-eks-cluster-GDl9F3iW",

"Value": "owned",

"PropagateAtLaunch": true

},

{

"ResourceId": "test-eks-cluster-GDl9F3iW-worker-group-120220812100108383000000016",

"ResourceType": "auto-scaling-group",

"Key": "kubernetes.io/cluster/test-eks-cluster-GDl9F3iW",

"Value": "owned",

"PropagateAtLaunch": true

}

],

"TerminationPolicies": [

"Default"

],

"NewInstancesProtectedFromScaleIn": false,

"ServiceLinkedRoleARN": "arn:aws:iam::594977439087:role/aws-service-role/autoscaling.amazonaws.com/AWSServiceRoleForAutoScaling"

},

Auto scaling groups

[test-eks-cluster-Mw6qwFqF-worker-group-120220813051019680600000017](https://us-east-1.console.aws.amazon.com/ec2/v2/home?region=us-east-1#AutoScalingGroupDetails:id=test-eks-cluster-Mw6qwFqF-worker-group-120220813051019680600000017;view=details)

[test-eks-cluster-Mw6qwFqF-worker-group-220220813051019680400000016](https://us-east-1.console.aws.amazon.com/ec2/v2/home?region=us-east-1#AutoScalingGroupDetails:id=test-eks-cluster-Mw6qwFqF-worker-group-220220813051019680400000016;view=details)

**$ aws autoscaling describe-auto-scaling-groups --query "AutoScalingGroups[? Tags[? (Key=='k8s.io/cluster-autoscaler/enabled') && Value=='true']]".AutoScalingGroupName --region us-east-1**

[

"[test-eks-cluster-Mw6qwFqF-worker-group-120220813051019680600000017](https://us-east-1.console.aws.amazon.com/ec2/v2/home?region=us-east-1#AutoScalingGroupDetails:id=test-eks-cluster-Mw6qwFqF-worker-group-120220813051019680600000017;view=details)"

]

# Namespaced objects Vs. Cluster-wide objects

In Kubernetes, namespaces provide a mechanism for isolating groups of resources within a single cluster. Names of resources need to be unique within a namespace, but not across namespaces. Namespace-based scoping is applicable only for namespaced objects (e.g. Deployments, Services, etc) and not for cluster-wide objects (e.g. StorageClass, Nodes, PersistentVolumes, etc).

**$ kubectl api-resources --namespaced=false**

The following are cluster-wide objects.

| NAME | SHORTNAMES | APIVERSION | KIND |
| --- | --- | --- | --- |
| componentstatuses | cs | v1 | ComponentStatus |
| namespaces | ns | v1 | Namespace |
| nodes | no | v1 | Node |
| persistentvolumes | pv | v1 | PersistentVolume |
| mutatingwebhookconfigurations |  | admissionregistration.k8s.io/v1 | MutatingWebhookConfiguration |
| validatingwebhookconfigurations |  | admissionregistration.k8s.io/v1 | ValidatingWebhookConfiguration |
| customresourcedefinitions | crd,crds | apiextensions.k8s.io/v1 | CustomResourceDefinition |
| apiservices |  | apiregistration.k8s.io/v1 | APIService |
| tokenreviews |  | authentication.k8s.io/v1 | TokenReview |
| selfsubjectaccessreviews |  | authorization.k8s.io/v1 | SelfSubjectAccessReview |
| selfsubjectrulesreviews |  | authorization.k8s.io/v1 | SelfSubjectRulesReview |
| subjectaccessreviews |  | authorization.k8s.io/v1 | SubjectAccessReview |
| certificatesigningrequests | csr | certificates.k8s.io/v1 | CertificateSigningRequest |
| eniconfigs |  | crd.k8s.amazonaws.com/v1alpha1 | ENIConfig |
| flowschemas |  | flowcontrol.apiserver.k8s.io/v1beta1 | FlowSchema |
| prioritylevelconfigurations |  | flowcontrol.apiserver.k8s.io/v1beta1 | PriorityLevelConfiguration |
| ingressclasses |  | networking.k8s.io/v1 | IngressClass |
| runtimeclasses |  | node.k8s.io/v1 | RuntimeClass |
| podsecuritypolicies | psp | policy/v1beta1 | PodSecurityPolicy |
| clusterrolebindings |  | rbac.authorization.k8s.io/v1 | ClusterRoleBinding |
| clusterroles |  | rbac.authorization.k8s.io/v1 | ClusterRole |
| priorityclasses | pc | scheduling.k8s.io/v1 | PriorityClass |
| csidrivers |  | storage.k8s.io/v1 | CSIDriver |
| csinodes |  | storage.k8s.io/v1 | CSINode |
| storageclasses | sc | storage.k8s.io/v1 | StorageClass |
| volumeattachments |  | storage.k8s.io/v1 | VolumeAttachment |

# Frequently used Kubectl commands

**$kubectl get all**

**$ kubectl get nodes -A**

NAME STATUS ROLES AGE VERSION

ip-10-0-2-86.ec2.internal Ready <none> 139m v1.20.15-eks-99076b2

ip-10-0-3-236.ec2.internal Ready <none> 139m v1.20.15-eks-99076b2

1. **$kubectl get events –all-namespaces**
2. **$kubectl apply -f <<name of the yaml>> [to create deployment-pod/service]**
3. **$kubectl get pod**

You can also watch ongoing things through

1. **$kubectl get pod –watch**
2. **$kubectl describe pod <<pod name>>**
3. **$kubectl describe pods -n <<namespace>>**
4. **$kubectl describe service <<service name>>**
5. **$kubectl logs <<name of the pod – So you can get the things what's going on inside the pod creation>>**

Checking the Status of the Pods for All the Namespaces

**Issue**: You want to check the status of the pods for all the namespaces.

**Workaround**: To check the status of all the pods for all the namespaces use the following command:

**$ kubectl get pods -A**

or

**$ kubectl get pods --all-namespaces**

| NAMESPACE | NAME | READY | STATUS | RESTARTS | AGE |
| --- | --- | --- | --- | --- | --- |
| kube-system | aws-node-c55j9 | 1/1 | Running | 0 | 47h |
| kube-system | coredns-65bfc5645f-pz57v | 1/1 | Running | 0 | 47h |
| kube-system | kube-proxy-47bck | 1/1 | Running | 0 | 47h |
| kube-system | kube-proxy-msz7m | 1/1 | Running | 0 | 47h |
| spindle-nuance | spindle-nuance-8464d6f96c-mfdvh | 1/1 | Running | 0 | 47h |

Checking the Physical and Internal IP Details of All the Pods

Issue: You want to check the physical and internal IP details of all the pods.

Workaround: To check the physical and internal IP details of all the pods use the following command:

**$kubectl get pods -n spindle-nuance -o wide**

| NAME | READY | STATUS | RESTARTS | AGE | IP | NODE | NOMINATED NODE | READINESS GATES |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
| spindle-nuance-7684d65bb-7qzsf | 1/1 | Running | 2 (81s ago) | 5m59s | 10.10.2.203 | ip-10-10-2-65.ec2.internal | <none> | <none> |
| spindle-nuance-7684d65bb-9x8qb | 0/1 | Error | 0 | 5m59s | 10.10.3.199 | ip-10-10-3-104.ec2.internal | <none> | <none> |
| spindle-nuance-7684d65bb-fmw7l | 0/1 | Pending | 0 | 46s | <none> | <none> | <none> | <none> |
| spindle-nuance-7684d65bb-gmssw | 0/1 | ContainerStatusUnknown | 1 (104s ago) | 5m | <none> | ip-10-10-2-65.ec2.internal | <none> | <none> |
| spindle-nuance-7684d65bb-jd2pc | 0/1 | OOMKilled | 2 (90s ago) | 3m58s | 10.10.2.124 | ip-10-10-2-65.ec2.internal | <none> | <none> |

To look for errors in the logs of the previous pod that crashed, run the following command:

**$ kubectl logs - - previous YOUR\_POD\_NAME -n NAMESPACE**

**$ kubectl logs coredns-65bfc5645f-m822w -n kube-system --tail 20 --follow**

"livenessProbe": {

"exec": {

"command": [

"/app/grpc-health-probe",

"-addr=:50051"

]

},

"initialDelaySeconds": 60,

<<some more key-value pair>>

},

"readinessProbe": {

"exec": {

"command": [

"/app/grpc-health-probe",

"-addr=:50051"

]

},

"initialDelaySeconds": 1,

<<some more key-value pair>>

},

"terminationMessagePath": "/dev/termination-log",

"terminationMessagePolicy": "File",

"imagePullPolicy": "Always",

"securityContext": {

"capabilities": {

"add": [

"NET\_ADMIN"

]

}

}

}

],

"restartPolicy": "Always",

"terminationGracePeriodSeconds": 10,

"dnsPolicy": "ClusterFirst",

<<some more key-value pair>>

"securityContext": {},

**$ kubectl get events --sort-by='.metadata.creationTimestamp' -A**

exec command will help you to execute OS specific commands on your runtime container [that has exactly the structure of docker image and its path]. For instance, The hosts file content would look like this:

**$ kubectl exec spindle-nuance-84bb657fb4-4x2ks -n spindle-nuance -- cat /etc/hosts**

**$ kubectl exec -it prometheus-prometheus-prometheus-oper-prometheus-0 -- /bin/sh**

**$ kubectl exec spindle-nuance-84bb657fb4-4x2ks -n spindle-nuance -- cat /etc/passwd**

Use the “cat” command to list all the users on the terminal to display all the user account details and passwords stored in the /etc/passwd file of the Linux system.

**$ kubectl apply -f hpa.yml**

horizontalpodautoscaler.autoscaling/spindle-nuance **created**

[If you apply the same file with some modification, status message will be like below]

**$ kubectl apply -f hpa.yml**

horizontalpodautoscaler.autoscaling/spindle-nuance **configured**

## Source of ‘kubectl explain’ output

kubectl explain command is used to show documentation about Kubernetes resources like pod.

Example:

**$ kubectl explain Pods**

The information displayed as output of this command is obtained from the OpenAPI Specification for the Pod. OpenAPI Spec for all the Types is generated by the main API server when it starts up and is maintained by it in memory.

Another Example:

**$ kubectl explain deployment.spec.template.spec.securityContext**

# Evicted Pod

## Why bother deleting an Evicted Pod?

1. When we have too many evicted pods in our cluster, this can lead to network load as each pod, even though it is evicted is connected to the network
2. In case of a cloud Kubernetes cluster, will have blocked an IP address, which can lead to exhaustion of IP addresses too if you have a fixed pool of IP addresses for your cluster.
3. Also, when we have too many pods in Evicted status, it becomes difficult to monitor the pods by running the kubectl get pod command as you will see too many evicted pods, which can be a bit confusing at times.

## But what if you have 100 Evicted pods?

Well, don't worry, we have a special command, which will find all the evicted pods and delete them just by running a single command.

**$kubectl get pod -n studytonight | grep Evicted | awk '{print $1}' | xargs kubectl delete pod -n studytonight**

In the above command, we are searching for the pods with status Evicted and then running kubectl delete pod command for all of them. You can also use the above command to delete pods in any particular status like Running, Evicted, CrashLoopBackOff, etc.

# Metrics Server

Is Metrics API available at your cluster?

**$ kubectl top pods -n spindle-nuance**

error: Metrics API not available

If you get the above error, your cluster doesn’t have Metrics Server.

Other approach to make sure about availability of metrics server:

To see whether the metrics-server is running, or another provider of the resource metrics API (metrics.k8s.io), run the following command:

**$ kubectl get apiservices**

If the resource metrics API is available, the output includes a reference to metrics.k8s.io.

NAME

v1beta1.metrics.k8s.io

If your output doesn’t include an above name reference, then your cluster doesn’t have metrics server.

## Installing the Kubernetes Metrics Server

The Kubernetes Metrics Server is an aggregator of resource usage data in your cluster, and it is not deployed by default in Amazon EKS clusters. The Metrics Server is commonly used by other Kubernetes add ons, such as the [Horizontal Pod Autoscaler](https://docs.aws.amazon.com/eks/latest/userguide/horizontal-pod-autoscaler.html) or the [Kubernetes Dashboard](https://docs.aws.amazon.com/eks/latest/userguide/dashboard-tutorial.html). For more information, see [Resource metrics pipeline](https://kubernetes.io/docs/tasks/debug/debug-cluster/resource-metrics-pipeline/) in the Kubernetes documentation. This topic explains how to deploy the Kubernetes Metrics Server on your Amazon EKS cluster.

Deploy the Metrics Server onto your cluster

**$ kubectl apply -f** [**https://github.com/kubernetes-sigs/metrics-server/releases/latest/download/components.yaml**](https://github.com/kubernetes-sigs/metrics-server/releases/latest/download/components.yaml)

Verify that the metrics-server deployment is running the desired number of pods

**$kubectl get deployment metrics-server -n kube-system**

If you execute top pods after the above fix, you will get results instead of errors.

**$ kubectl top pods -n kube-system**

| NAMESPACE | NAME | CPU(cores) | MEMORY(bytes) |
| --- | --- | --- | --- |
| kube-system | Aws-node-c55j9 | 5m | 50Mi |
| kube-system | Aws-node-cmrrv | 5m | 49Mi |
| kube-system | coredns-65bfc5645f-m822w | 2m | 15Mi |
| kube-system | coredns-65bfc5645f-pz57v | 3m | 15Mi |
| kube-system | Kube-proxy-msz7m | 1m | 20Mi |
| kube-system | metrics-server-6594d67d48-q4ntq | 4m | 15Mi |

**$ kubectl get apiservices**

| NAME | SERVICE | AVAILABLE | AGE |
| --- | --- | --- | --- |
| v1. | Local | True | 7d22h |
| v1.admissionregistration.k8s.io | Local | True | 7d22h |
| v1.apiextensions.k8s.io | Local | True | 7d22h |
| v1.apps | Local | True | 7d22h |
| v1.authentication.k8s.io | Local | True | 7d22h |
| v1.authorization.k8s.io | Local | True | 7d22h |
| v1.autoscaling | Local | True | 7d22h |
| v1.batch | Local | True | 7d22h |
| v1.certificates.k8s.io | Local | True | 7d22h |
| v1.coordination.k8s.io | Local | True | 7d22h |
| v1.events.k8s.io | Local | True | 7d22h |
| v1.networking.k8s.io | Local | True | 7d22h |
| v1.node.k8s.io | Local | True | 7d22h |
| v1.rbac.authorization.k8s.io | Local | True | 7d22h |
| v1.scheduling.k8s.io | Local | True | 7d22h |
| v1.storage.k8s.io | Local | True | 7d22h |
| v1alpha1.crd.k8s.amazonaws.com | Local | True | 7d22h |
| v1beta1.admissionregistration.k8s.io | Local | True | 7d22h |
| v1beta1.apiextensions.k8s.io | Local | True | 7d22h |
| v1beta1.authentication.k8s.io | Local | True | 7d22h |
| v1beta1.authorization.k8s.io | Local | True | 7d22h |
| v1beta1.batch | Local | True | 7d22h |
| v1beta1.certificates.k8s.io | Local | True | 7d22h |
| v1beta1.coordination.k8s.io | Local | True | 7d22h |
| v1beta1.discovery.k8s.io | Local | True | 7d22h |
| v1beta1.events.k8s.io | Local | True | 7d22h |
| v1beta1.extensions | Local | True | 7d22h |
| v1beta1.flowcontrol.apiserver.k8s.io | Local | True | 7d22h |
| v1beta1.metrics.k8s.io | Kube-system/metrics-server | True | 14h |
| v1beta1.networking.k8s.io | Local | True | 7d22h |
| v1beta1.node.k8s.io | Local | True | 7d22h |
| v1beta1.policy | Local | True | 7d22h |
| v1beta1.rbac.authorization.k8s.io | Local | True | 7d22h |
| v1beta1.scheduling.k8s.io | Local | True | 7d22h |
| v1beta1.storage.k8s.io | Local | True | 7d22h |
| v1beta1.vpcresources.k8s.aws | Local | True | 7d22h |
| v2beta1.autoscaling | Local | True | 7d22h |
| v2beta2.autoscaling | Local | True | 7d22h |

Once Metrics Server is deployed, you can query the Metrics API to retrieve current metrics from any node or pod using the below commands.

kubectl get --raw /apis/metrics.k8s.io/v1beta1/nodes/<NODE\_NAME> | jq

kubectl get --raw /apis/metrics.k8s.io/v1beta1/namespaces/<NAMESPACE>/pods/<POD\_NAME> | jq

$ kubectl get --raw /apis/metrics.k8s.io/v1beta1/namespaces/spindle-nuance/pods/spindle-nuance-8968dcb69-bnn42 | jq

{

"kind": "PodMetrics",

"apiVersion": "metrics.k8s.io/v1beta1",

"metadata": {

"name": "spindle-nuance-8968dcb69-bnn42",

"namespace": "spindle-nuance",

"creationTimestamp": "2022-10-12T13:16:59Z",

"labels": {

"app": "spindle-nuance",

"pod-template-hash": "8968dcb69"

}

},

"timestamp": "2022-10-12T13:16:42Z",

"window": "18.473s",

"containers": [

{

"name": "spindle-nuance-container",

"usage": {

"cpu": "31095930n",

"memory": "4744636Ki"

}

}

]

}

$ **kubectl get --raw /apis/metrics.k8s.io/v1beta1/nodes/ip-10-10-2-221.ec2.internal | jq**

{

"kind": "NodeMetrics",

"apiVersion": "metrics.k8s.io/v1beta1",

"metadata": {

"name": "ip-10-10-2-221.ec2.internal",

"creationTimestamp": "2022-10-12T13:24:26Z",

"labels": {

"beta.kubernetes.io/arch": "amd64",

"beta.kubernetes.io/instance-type": "t3.2xlarge",

"beta.kubernetes.io/os": "linux",

"eks.amazonaws.com/capacityType": "ON\_DEMAND",

<<more key-value pairs>>

"topology.kubernetes.io/zone": "us-east-1a"

}

},

"timestamp": "2022-10-12T13:24:21Z",

"window": "20.035s",

"usage": {

"cpu": "156989995n",

"memory": "6003324Ki"

}

}

**$ kubectl describe hpa spindle-nuance -n spindle-nuance**

Name: spindle-nuance

Namespace: spindle-nuance

Labels: <none>

Annotations: <none>

CreationTimestamp: Wed, 10 Aug 2022 20:07:49 +0530

Reference: Deployment/spindle-nuance

Metrics: ( current / target )

resource cpu on pods (as a percentage of request): 112% (563m) / 10%

Min replicas: 1

Max replicas: 5

Behavior:

Scale Up:

Stabilization Window: 0 seconds

Select Policy: Max

Policies:

- Type: Percent Value: 100 Period: 30 seconds

- Type: Pods Value: 3 Period: 30 seconds

Scale Down:

Stabilization Window: 300 seconds

Select Policy: Max

Policies:

- Type: Percent Value: 100 Period: 15 seconds

Deployment pods: 4 current / 4 desired

Conditions:

Type Status Reason Message

---- ------ ------ -------

AbleToScale True SucceededGetScale the HPA controller was able to get the target's current scale

ScalingActive False FailedGetResourceMetric the HPA was unable to compute the replica count: failed to get cpu utilization: unable to get metrics for resource cpu: unable to fetch metrics from resource metrics API: the server is currently unable to handle the request (get pods.metrics.k8s.io)

ScalingLimited True ScaleUpLimit the desired replica count is increasing faster than the maximum scale rate

Events:

Type Reason Age From Message

---- ------ ---- ---- -------

Warning FailedGetResourceMetric 3m48s (x4502 over 19h) horizontal-pod-autoscaler failed to get cpu utilization: missing request for cpu

1. arn:aws:logs:ap-south-1:594977439087:log-group:/aws/kinesisfirehose/KDS-HTP-fkdxO:\*
2. arn:aws:logs:ap-south-1:594977439087:log-group:/aws/kinesisfirehose/KDS-HTP-fkdxO:\*

# Image Management

## Purging All Unused or Dangling Images, Containers, Volumes, and Networks

Docker provides a single command that will clean up any resources — images, containers, volumes, and networks — that are dangling (not tagged or associated with a container):

**$ docker system prune**

To additionally remove any stopped containers and all unused images (not just dangling images), add the -a flag to the command:

**$ docker system prune -a**

## docker pull

After checkout specific branch (for instance, spindle-nuance), Pull an image or a repository from a registry

**$docker login registry.gitlab.com -p /home/sathiyan/.docker/config.json**

**$docker build --pull -t "spindle\_nuance:latest\_nuance\_v1"**

ECR (Elastic Container Registry) is a kind of Docker Hub registry from AWS. Since we are into B2B space, we don’t need to put images outside of our aws account. So ECR is our prefered registry to Docker Hub.

## Convert image from local repository to file system

Let assume the following is your image name along with a tag at your local repository.

How do we need to get that into our file system, so that you can share it to your peer or put inside the gitlab repository.

**$docker save myimage:latest | gzip > myimage\_latest.tar.gz**

**$docker save 594977439087.dkr.ecr.us-east-1.amazonaws.com/spindle/nuance:latest | gzip > spindle\_nuance\_latest.tar.gz**

1. Push the images into ECR (Elastic Container Registry) [if team delivers in the form of tar gz]

**$docker load -i synapsica\_spindle\_v1.9.tar.gz**

**$aws ecr get-login-password --region us-east-1 -- profile mfa | docker login --username AWS --password-stdin 594977439087.dkr.ecr.us-east-1.amazonaws.com**

**$docker tag synapsica-spindle:v1.9 594977439087.dkr.ecr.us-east-1.amazonaws.com/synapsica-spindle:v1.9**

**$docker push 594977439087.dkr.ecr.us-east-1.amazonaws.com/synapsica-spindle:v1.9**

# Cluster Autoscaler

Two parameters should be updated in cluster-autoscaler.yml. Cluster-autoscaler.yml is mostly standard common scripts across all projects except the following two dynamic parameters.

Parameter 1: Cluster name [outcome of eks\_setup module]

**$ aws eks list-clusters --region us-east-1**

{

"clusters": [

"spindle-ai-eks-terraform-cluster"

]

}

Parameter 2: cluster autoscaler role arn [outcome of eks\_cluster\_autoscaler\_role\_arn module]

eks\_ca\_iam\_role\_arn = "arn:aws:iam::594977439087:role/spindle-ai-eks-terraform-cluster-autoscaler"

You need to execute the following command after update the two parameters in cluster-autoscaler.yml

**$kubectl apply -f ./cluster\_autoscaler/cluster-autoscaler.yml**

# Enabling IAM user and role access to your cluster

Access to your cluster using AWS Identity and Access Management (IAM) entities is enabled by the AWS IAM Authenticator for Kubernetes, which runs on the Amazon EKS control plane. The authenticator gets its configuration information from the aws-auth ConfigMap.

You can see which other roles or users currently have access to your cluster with the following command:

$ kubectl describe -n kube-system configmap/aws-auth

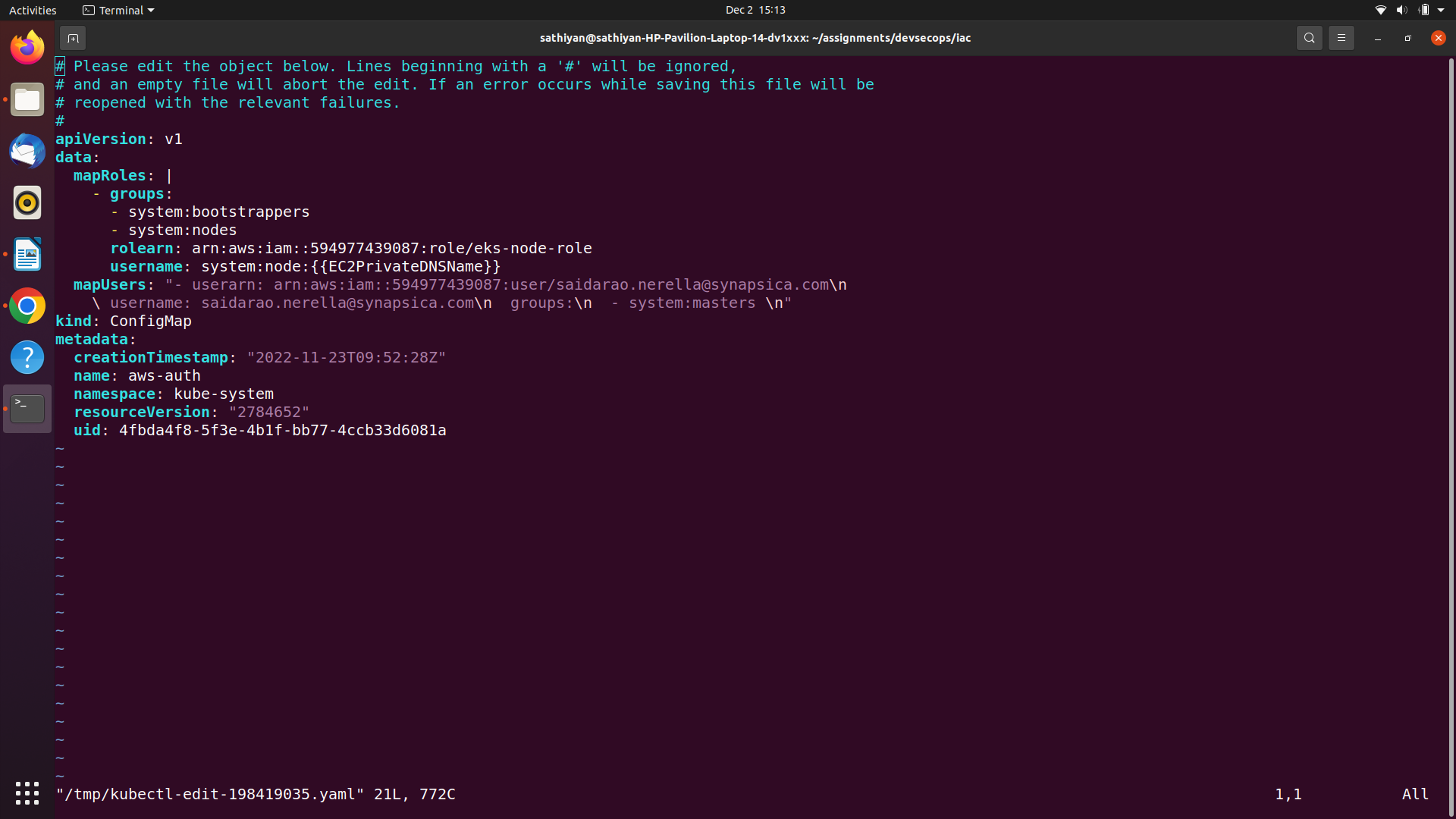
Edit the aws-auth ConfigMap:

$ kubectl edit configmap aws-auth -n kube-system

(or) you can download the yaml file, then you can edit and apply by kubectl.

curl -o aws-auth-cm.yaml <https://s3.us-west-2.amazonaws.com/amazon-eks/cloudformation/2020-10-29/aws-auth-cm.yaml>

$ kubectl apply -f aws-auth-cm.yaml



# How To Setup Kube State Metrics on Kubernetes

Kube State metrics is a service that talks to the Kubernetes API server to get all the details about all the API objects like deployments, pods, daemonsets, Statefulsets, etc.

Primarily it produces metrics in Prometheus format with the stability as the Kubernetes API. Overall it provides kubernetes objects & resources metrics that you cannot get directly from native Kubernetes monitoring components.

Kube state metrics service exposes all the metrics on /metrics URI. [Prometheus](https://devopscube.com/setup-prometheus-monitoring-on-kubernetes/) can scrape all the metrics exposed by Kube state metrics.

Following are some of the important metrics you can get from Kube state metrics.

1. Node status, node capacity (CPU and memory)
2. Replica-set compliance (desired/available/unavailable/updated status of replicas per deployment)
3. Pod status (waiting, running, ready, etc)
4. Ingress metrics
5. PV, PVC metrics
6. Daemonset & Statefulset metrics.
7. Resource requests and limits.
8. Job & Cronjob metrics

## Setup Guide

1. helm repo add prometheus-community <https://prometheus-community.github.io/helm-charts>
2. helm repo update
3. helm install kube-state-metrics prometheus-community/kube-state-metrics
4. helm list <<namespace>> [It will list all the installed helm names in the given namespaces. It can be used for delete those]

## Get the Metrics through endpoint in raw format

[This is through metrics server - for cpu and memory alone]

kubectl get --raw /apis/metrics.k8s.io/v1beta1/namespaces/spindle-nuance/pods/spindle-nuance-8968dcb69-bnn42 | jq

kubectl get --raw /apis/metrics.k8s.io/v1beta1/namespaces/<NAMESPACE>/pods/<POD\_NAME> | jq

# How to Expose the cluster service

You have some cluster service For instance, kube-state-metrics, you can use port forwarding technique to test the service.

$ kubectl get svc --all-namespaces

NAMESPACE NAME TYPE CLUSTER-IP EXTERNAL-IP PORT(S)

default kube-state-metrics ClusterIP 172.20.1.144 <none> 8080/TCP

Let’s test it locally, by exposing the service, run the below command

$ kubectl --namespace default<<or other namespace where service resides>> port-forward svc/kube-state-metrics 8080:8080

Forwarding from 127.0.0.1:8080 -> 8080

Forwarding from [::1]:8080 -> 8080

Handling connection for 8080

Handling connection for 8080

[After port forward] You will get the output for the below url <http://127.0.0.1:8080/metrics>

# Troubleshoot in your clusters:

## Troubleshoot the pod:

### Encoded Error Message

If your application sends sensitive error messages in encoded form, you can get decoded through sts.

**$aws sts decode-authorization-message --encoded-message <<encoded error messages that are from aws/kubernetes>>**

$aws sts decode-authorization-message --encoded-message <<error messages>>

### Pods in the pending state

It can’t be scheduled onto a node. Below - the output of the “describe pod” command to get some insights on why it is in pending state.

**$ kubectl describe pod spindle-nuance-7684d65bb-5577h -n spindle-nuance**

Name: spindle-nuance-7684d65bb-5577h

Priority: 0

Status: Pending

Containers:

spindle-nuance-container:

State: Waiting

Reason: ContainerCreating

Ready: False

Restart Count: 0

Limits:

cpu: 1

Requests:

cpu: 500m

Conditions:

Type Status

Initialized True

Ready False

ContainersReady False

PodScheduled True

Tolerations: node.kubernetes.io/not-ready:NoExecute op=Exists for 300s

node.kubernetes.io/unreachable:NoExecute op=Exists for 300s

Events:

Type Reason Age From Message

---- ------ ---- ---- -------

Warning FailedScheduling 2m29s default-scheduler 0/2 nodes are available: 1 node(s) had taint {node.kubernetes.io/memory-pressure: }, that the pod didn't tolerate, 1 node(s) were unschedulable.

<<some more messages here>>

Normal Pulling 58s kubelet Pulling image "594977439087.dkr.ecr.us-east-1.amazonaws.com/spindle/nuance:v1.9"

### Reason for Pending state

Pending state can happen because:

1. There are no nodes available
2. There are insufficient resources on the available worker nodes
   1. This can be due to not enough available cpu/disk/ram/pods on the worker node to accommodate the pods
3. Nodes have taints that which pods can’t tolerate
4. You have defined an occupied hostPort

kubectl describe node ip-10-10-2-118.ec2.internal

Name: ip-10-10-2-118.ec2.internal

Roles: <none>

Labels: beta.kubernetes.io/arch=amd64

beta.kubernetes.io/instance-type=t3.medium

beta.kubernetes.io/os=linux

eks.amazonaws.com/capacityType=ON\_DEMAND

eks.amazonaws.com/nodegroup=spindle-ai-eks-terraform-cluster-private-ng

eks.amazonaws.com/nodegroup-image=ami-06ce408c7d97c5f2d

failure-domain.beta.kubernetes.io/region=us-east-1

failure-domain.beta.kubernetes.io/zone=us-east-1a

k8s.io/cloud-provider-aws=f1e7420fe24818b3ea0df8f6de77268f

kubernetes.io/arch=amd64

kubernetes.io/hostname=ip-10-10-2-118.ec2.internal

kubernetes.io/os=linux

node.kubernetes.io/instance-type=t3.medium

topology.kubernetes.io/region=us-east-1

topology.kubernetes.io/zone=us-east-1a

Annotations: node.alpha.kubernetes.io/ttl: 0

volumes.kubernetes.io/controller-managed-attach-detach: true

CreationTimestamp: Wed, 31 Aug 2022 12:57:36 +0530

Taints: DeletionCandidateOfClusterAutoscaler=1661932171:PreferNoSchedule

Unschedulable: false

Lease:

HolderIdentity: ip-10-10-2-118.ec2.internal

AcquireTime: <unset>

RenewTime: Wed, 31 Aug 2022 13:36:24 +0530

Conditions:

Type Status LastHeartbeatTime LastTransitionTime Reason Message

---- ------ ----------------- ------------------ ------ -------

MemoryPressure False Wed, 31 Aug 2022 13:35:25 +0530 Wed, 31 Aug 2022 13:30:24 +0530 KubeletHasSufficientMemory kubelet has sufficient memory available

DiskPressure False Wed, 31 Aug 2022 13:35:25 +0530 Wed, 31 Aug 2022 12:57:34 +0530 KubeletHasNoDiskPressure kubelet has no disk pressure

PIDPressure False Wed, 31 Aug 2022 13:35:25 +0530 Wed, 31 Aug 2022 12:57:34 +0530 KubeletHasSufficientPID kubelet has sufficient PID available

Ready True Wed, 31 Aug 2022 13:35:25 +0530 Wed, 31 Aug 2022 12:57:56 +0530 KubeletReady kubelet is posting ready status

Addresses:

InternalIP: 10.10.2.118

Hostname: ip-10-10-2-118.ec2.internal

InternalDNS: ip-10-10-2-118.ec2.internal

Capacity:

attachable-volumes-aws-ebs: 25

cpu: 2

ephemeral-storage: 20959212Ki

hugepages-1Gi: 0

hugepages-2Mi: 0

memory: 3965444Ki

pods: 17

Allocatable:

attachable-volumes-aws-ebs: 25

cpu: 1930m

ephemeral-storage: 18242267924

hugepages-1Gi: 0

hugepages-2Mi: 0

memory: 3410436Ki

pods: 17

System Info:

Machine ID: ec2c512cb9dcdd250cad4b2db03ddc5e

<<some more key-value pair>>

Namespace Name CPU Requests CPU Limits Memory Requests Memory Limits Age

--------- ---- ------------ ---------- --------------- ------------- ---

kube-system aws-node-qgrsc 25m (1%) 0 (0%) 0 (0%) 0 (0%) 11m

kube-system kube-proxy-lmvvj 100m (5%) 0 (0%) 0 (0%) 0 (0%) 11m

Allocated resources:

(Total limits may be over 100 percent, i.e., overcommitted.)

Resource Requests Limits

-------- -------- ------

cpu 125m (6%) 0 (0%)

<<some more rows>>

attachable-volumes-aws-ebs 0 0

Events:

Type Reason Age From Message

---- ------ ---- ---- -------

Normal Starting 38m kube-proxy

Normal Starting 11m kube-proxy

Normal NodeHasNoDiskPressure 39m (x2 over 39m) kubelet Node ip-10-10-2-118.ec2.internal status is now: NodeHasNoDiskPressure

Normal NodeAllocatableEnforced 39m kubelet Updated Node Allocatable limit across pods

Normal Starting 39m kubelet Starting kubelet.

Normal NodeHasSufficientPID 39m (x2 over 39m) kubelet Node ip-10-10-2-118.ec2.internal status is now: NodeHasSufficientPID

Normal RegisteredNode 38m node-controller Node ip-10-10-2-118.ec2.internal event: Registered Node ip-10-10-2-118.ec2.internal in Controller

Normal NodeReady 38m kubelet Node ip-10-10-2-118.ec2.internal status is now: NodeReady

Normal ScaleDown 11m cluster-autoscaler marked the node as toBeDeleted/unschedulable

Warning EvictionThresholdMet 11m (x5 over 35m) kubelet Attempting to reclaim memory

Normal NodeHasInsufficientMemory 11m (x3 over 35m) kubelet Node ip-10-10-2-118.ec2.internal status is now: NodeHasInsufficientMemory

Normal ClusterAutoscalerCleanup 10m cluster-autoscaler removed ToBeDeletedTaint taint from node ip-10-10-2-118.ec2.internal

Normal NodeHasSufficientMemory 6m10s (x5 over 39m) kubelet Node ip-10-10-2-118.ec2.internal status is now: NodeHasSufficientMemory

## Troubleshoot the memory and storage

This issue happened due to the lack of temporary storage while processing such as applications process their jobs and store temporary, cache data. To resolve this issue, you must dive into your pod, and check when the process running which device location cost your available storage by command df-h, and observe the available capacity size. You can create an pvc (with hostpath, or other ways) which has larger size and mount into pod's directory which store their temporary data.

curl --location --request POST 'http://afc2153d20f6541adbc082fe2235cb05-1550214045.us-east-1.elb.amazonaws.com:7000/api/v1/spindle/getprediction' --form 'aiProduct="spindle"' --form 'studyId="456.456.35.4645.6567.567.457"' --form 'version="2"' --form 'type="zip"' --form 'file=@"Downloads/ANON\_PATIENT\_0659\_MR\_1643812383.zip"'

To get the free memory on the node

free -h -s 5 -c 20

to understand the port used status:

sudo lsof -i -P -n | grep LISTEN

## Unlock the terraform acquired state

Manually unlock the state for the defined configuration.

This will not modify your infrastructure. This command removes the lock on the state for the current configuration. The behavior of this lock is dependent on the backend being used. Local state files cannot be unlocked by another process.

Usage: terraform force-unlock [options] LOCK\_ID

Manually unlock the state for the defined configuration.

This will not modify your infrastructure. This command removes the lock on the state for the current configuration. The behavior of this lock is dependent on the backend being used. Local state files cannot be unlocked by another process.

# Exposing multiple ports/services on same Load Balancer in Kubernetes

Scenario: I have to expose Kibana (5601), Apache Storm (8080) & say nginx (80), all on the same Load Balancer (public IP) on Kubernetes.

Implementation:

You can have multiple tags on the same workload (Deployment /Service /Pod /etc.). And then Kubernetes will internally link objects with similar tags together.

### **Step 1: Create an external facing Load Balancer**

apiVersion: v1

kind: Service

metadata:

name: external-lb

spec:

type: LoadBalancer

ports:

- name: elk-kibana

port: 5601

targetPort: "5601-port"

- name: storm-nimbus

port: 8080

targetPort: "8080-port"

- name: nginx

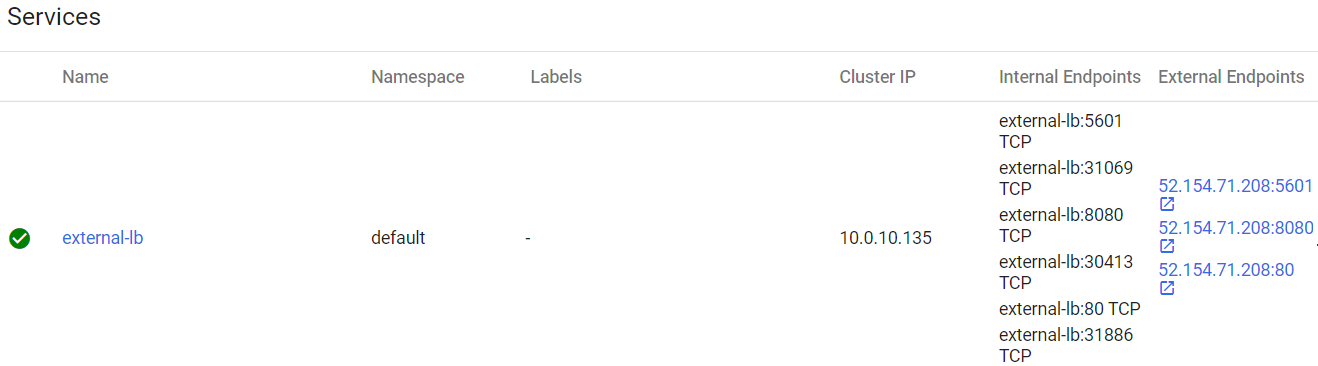
port: 80

targetPort: "80-port"

selector:

lbtype: external

Nothing fancy in the yaml file other than a label architecture as an important point – selector (lbtype: external)



### **Step 2: Create deployment/services to setup Kibana, Apache Storm & Nginx**

Kibana : Kind - Deployment

—-------------------- —-----------

apiVersion: apps/v1

kind: Deployment

metadata:

name: elk-kibana

labels:

app: elk-kibana

spec:

replicas: 1

selector:

matchLabels:

lbtype: external

template:

metadata:

labels:

lbtype: external

app: elk-kibana

spec:

containers:

- name: elk-kibana

image: elk-kibana:7.4.0-2

imagePullPolicy: Always

resources:

limits:

cpu: 1500m

memory: "2Gi"

ports:

- containerPort: 5601

name: "5601-port"

Nimbus: Kind - Service with StatefuleSet

—------------------------------ —------------------

apiVersion: v1

kind: Service

metadata:

name: storm-nimbus

labels:

app: storm-nimbus

spec:

ports:

- port: 8080

targetPort: 8080

name: 8080-port

selector:

app: storm-nimbus

---

apiVersion: apps/v1

kind: StatefulSet

metadata:

name: storm-nimbus

spec:

selector:

matchLabels:

lbtype: external

serviceName: storm-nimbus

replicas: 2

updateStrategy:

type: RollingUpdate

template:

metadata:

labels:

app: storm-nimbus

lbtype: external

spec:

containers:

- name: storm-nimbus

imagePullPolicy: Always

image: storm-nimbus-clustered:1.2.1

resources:

requests:

memory: "2G"

ports:

- containerPort: 8080

name: 8080-port

volumeMounts:

- name: storm-nimbus-data

mountPath: /var/lib/storm

- name: storm-logs

mountPath: /usr/share/storm/logs

env:

- name: node.name

valueFrom:

fieldRef:

fieldPath: metadata.name

volumes:

- name: storm-logs

hostPath:

path: /var/log/containers/storm-nimbus/default

volumeClaimTemplates:

- metadata:

name: storm-nimbus-data

labels:

app: storm-nimbus

spec:

accessModes: [ "ReadWriteOnce" ]

storageClassName: managed-premium-retain

resources:

requests:

storage: 10Gi

Nginx: Kind - Deployment

—---------------------------- —------------

apiVersion: apps/v1

kind: Deployment

metadata:

name: nginx

labels:

app: nginx

spec:

replicas: 1

selector:

matchLabels:

lbtype: external

template:

metadata:

labels:

lbtype: external

app: nginx

spec:

containers:

- name: nginx

image: nginx:latest

tty: true

imagePullPolicy: Always

resources:

limits:

cpu: 500m

memory: "1Gi"

ports:

- containerPort: 80

name: "80-port"

In all of the above deployment files, if you see carefully, there are multiple labels, one for selector and one for template. Selector will link each workload with external-lb (based of label name) whereas, to individually track deployment/service, we are using another label "app".

"lbtype: external" is set as selector for external-lb and so all the objects will be linked to external-lb with same label.

# Advanced traffic routing in Kubernetes using Ingress resources

Ingress can be used to expose your Pods running behind a Service object to the external world using HTTP and HTTPS routes.

Some cons on expose your application using Service objects directly, especially the LoadBalancer Service:

1. This approach works fine only in cloud environments where you have cloud-controller-manager running and by configuring external load balancers to be used with this type of Service.
2. Each LoadBalancer Service requires a separate instance of the cloud load balancer, which brings additional costs and maintenance overhead.

We are going to introduce Ingress and Ingress Controller, which can be used in any type of environment to provide routing and load-balancing capabilities for your application.

We have learned about Service objects, which can be used to expose Pods to load-balanced traffic, both internal as well as external. Internally, they are implemented as virtual IP addresses managed by kube-proxy at each of the Nodes. We are going to do a quick recap of different types of services:

1. ClusterIP
2. NodePort
3. LoadBalancer

To make it easier to explain, we will assume that we have a Deployment running three replicas of Pods running the nginx container, which has the following YAML manifest:

apiVersion: apps/v1

kind: Deployment

metadata:

name: nginx-deployment-example

spec:

replicas: 3

selector:

matchLabels:

environment: test

template:

metadata:

labels:

environment: test

spec:

containers:

- name: nginx

image: nginx:1.17

ports:

- containerPort: 80

The Pod exposes TCP port 80, which is used by the nginx process to serve the requests, and we will now discuss the details of using the ClusterIP Service to expose this Deployment internally.

## The ClusterIP Service

Let's now take a look at the ClusterIP Service type. This type of Service exposes Pods using internally visible virtual IP addresses managed by kube-proxy on each Node. This means that the Service will be reachable from within the cluster only. Consider the following manifest for the service:

apiVersion: v1

kind: Service

metadata:

name: nginx-deployment-example-clusterip

spec:

selector:

environment: test

type: ClusterIP

ports:

- port: 8080

protocol: TCP

targetPort: 80

The ClusterIP Service is configured in such a way that it will map requests coming from its IP and TCP port 8080 to the container's TCP port 80. The actual ClusterIP address is assigned dynamically, unless you specify one explicitly in the specifications. The internal DNS Service in a Kubernetes cluster is responsible for resolving the nginx-deployment-example name to the actual ClusterIP address as a part of service discovery.

Following diagrams that represent how the Service types are implemented logically. In fact, under the hood, kube-proxy is responsible for managing the virtual IP addresses on the Nodes and modifying all forwarding rules. So, services exist only as a logical concept inside the cluster. There is no physical process that runs inside the cluster for each Service and does the proxying.

We have visualized the ClusterIP Service principles in the following diagram:

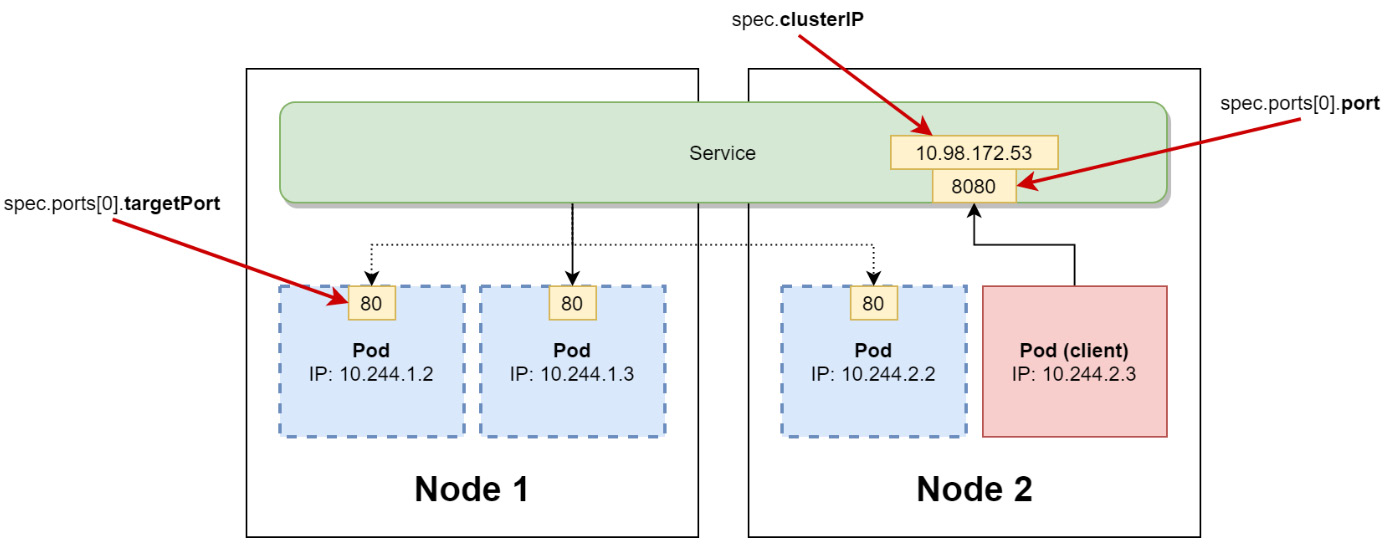


Figure – ClusterIP Service

The diagram includes references to the Kubernetes objects specifications to make it easier to understand the connections. ClusterIP Services are the most basic type of Service in Kubernetes and they are part of other Service types that allow Pods to be exposed to external traffic: NodePort and LoadBalancer.

## NodePort service

This type of Service is similar to the ClusterIP Service but additionally, it can be reached by any cluster node IP address and specified port. To achieve that, kube-proxy exposes the same port on each Node in the range 30000-32767 (which is configurable) and sets up forwarding so that any connections to this port will be forwarded to ClusterIP.

Let's take a look at an example YAML manifest of the NodePort Service:

apiVersion: v1

kind: Service

metadata:

name: nginx-deployment-example-nodeport

spec:

selector:

environment: test

type: NodePort

ports:

- port: 8080

nodePort: 31001

protocol: TCP

targetPort: 80

In this case, TCP port 31001 is used as the external port on each Node. If you do not specify nodePort, it will be allocated dynamically using the range. For internal communication, this Service still behaves like a simple ClusterIP Service, and you can use its ClusterIP address.

These principles have been visualized in the following diagram:

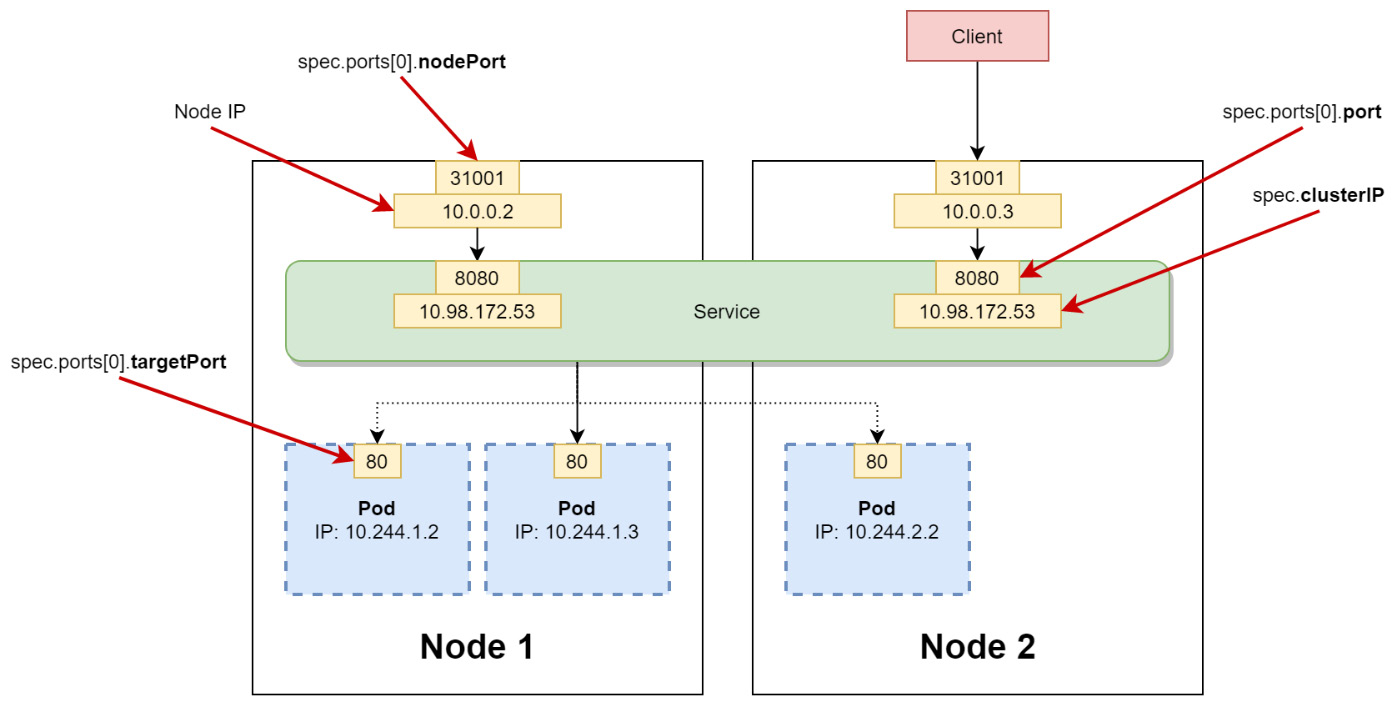


Figure – NodePort service

You will need the NodePort Service if you want to set up load balancing externally, without using any Kubernetes constructs. They can also be used for communicating directly with the Pods, but exposing the external IP addresses of Nodes is usually not a good idea in terms of security. Please note that many other components rely on NodePort Services; for example, they are used in LoadBalancer Service implementations or when exposing Ingress Controller.

We will now do a quick recap of the LoadBalancer Service.

## The LoadBalancer service

This is the second type of Service that allows external traffic to the Pods. The LoadBalancer Service is usually used in cloud environments where you have software-defined networking (SDN), and you can configure load balancers on demand that redirect traffic to your cluster. The automatic provisioning of load balancers in the cloud is done by vendor-specific plugins in cloud-controller-manager. This type of service combines the approach of the NodePort Service with an additional external load balancer in front of it, which routes traffic to NodePorts.

Let's take a look at an example YAML manifest of the LoadBalancer Service:

apiVersion: v1

kind: Service

metadata:

name: nginx-deployment-example-lb

spec:

selector:

environment: test

type: LoadBalancer

ports:

- port: 8080

protocol: TCP

targetPort: 80

When you create such a Service, it will behave as a NodePort Service with a randomly assigned port and a cloud load balancer that serves requests at TCP port 8080 and forwards them to NodePorts. The external IP address of the service is provided by the load balancer and is also available in the Service object in .status.loadBalancer.ingress[0].ip. These principles have been visualized in the following diagram:

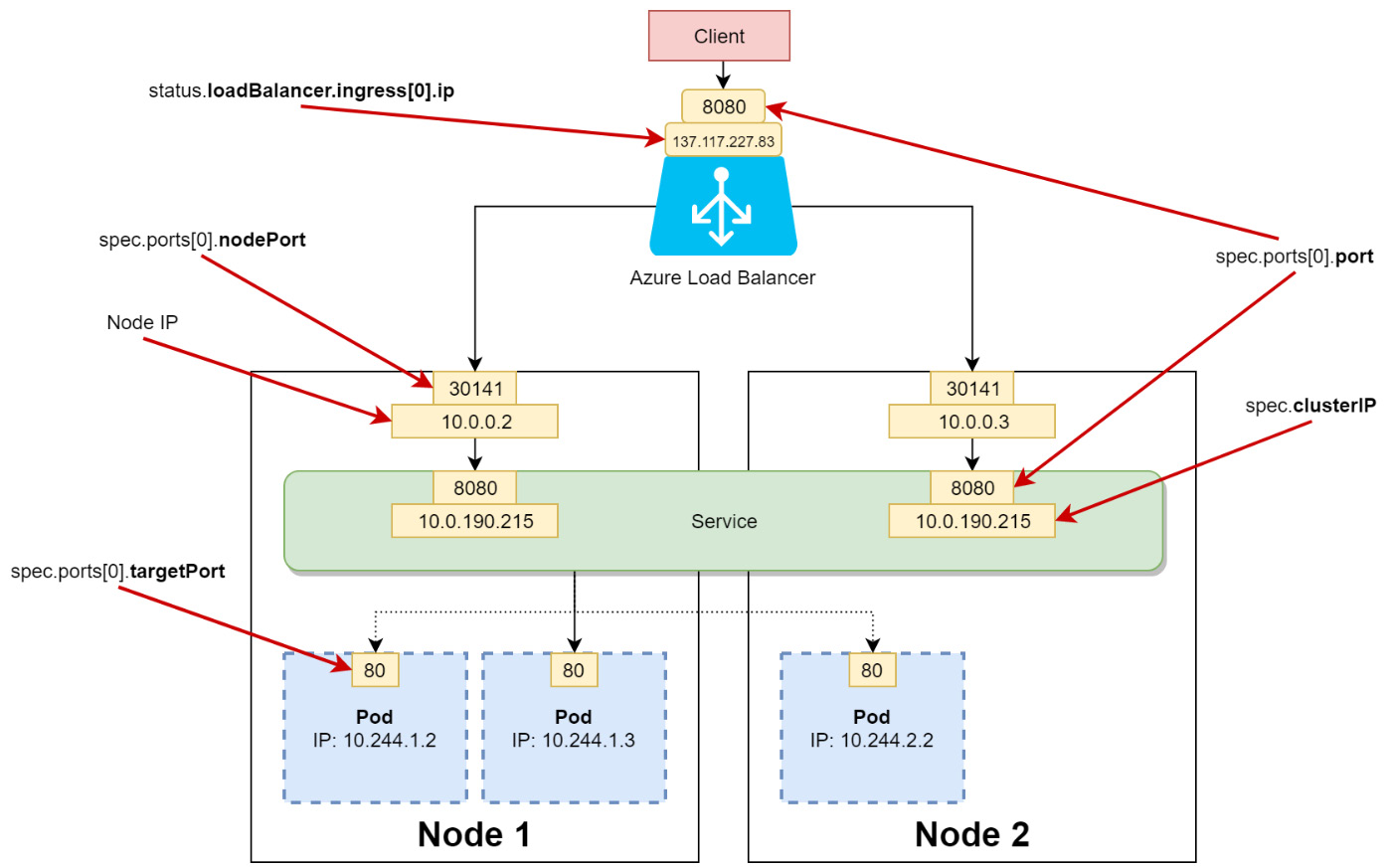


Figure – LoadBalancer service

You can still, of course, use the service internally via its ClusterIP.

It may seem appealing to always use Kubernetes services for allowing external traffic to the cluster, but there are a few disadvantages of using them all the time. We will now introduce the Ingress object and discuss why it is needed and when it should be used instead of Services to manage external traffic.

## Introducing the Ingress object

In the previous section, we did a short recap of Service objects in Kubernetes and their role in routing traffic. From the perspective of external traffic, the most important are the NodePort Service and the LoadBalancer Service. In general, the NodePort Service can only be used in conjunction with a different routing and load balancing component, as exposing multiple external endpoints on all Kubernetes Nodes is not secure. This leaves us with the LoadBalancer Service, which, under the hood, relies on NodePort. There are a few problems with this type of Service in some use cases:

1. The LoadBalancer Service is used for L4 load balancing, which means it is done at OSI layer 4 (transport). The load balancer can make the decisions based on the TCP/UDP protocol.
2. Applications that use HTTP or HTTPS protocols often require L7 load balancing, which is done at OSI layer 7 (application).
3. The L4 load balancer cannot do HTTPS traffic termination and offloading.
4. You cannot implement name-based virtual hosting using the same L4 load balancer for multiple domain names.
5. You need an L7 load balancer to implement path-based routing. For example, configuring requests to https://<loadBalancerIp>/service1 to be redirected to the Kubernetes Service named service1, and requests to https://<loadBalancerIp>/service2 to be redirected to the Kubernetes Service named service2 is not possible with the L4 load balancer – it is not aware of the HTTP(S) protocol.
6. You need an L7 load balancer if you want to implement features such as sticky sessions or cookie affinity.

In Kubernetes, you can solve these problems using an Ingress object, which can be used for implementing and modeling L7 load balancing. The Ingress object is used for defining the routing and balancing rules only, for example, which path should be routed to which Kubernetes Service.

Let's take a look at an example YAML manifest file, example-ingress.yaml, for Ingress:

apiVersion: networking.k8s.io/v1beta1

kind: Ingress

metadata:

name: example-ingress

annotations:

nginx.ingress.kubernetes.io/rewrite-target: /

spec:

rules:

- http:

host

paths:

- path: /service1

pathType: Prefix

backend:

serviceName: example-service1

servicePort: 80

- path: /service2

pathType: Prefix

backend:

serviceName: example-service2

servicePort: 80

Simply put, Ingress is an abstract definition of routing rules for your Services. Alone, it is not doing anything; it requires the Ingress Controller to actually process and implement these rules – you can apply the manifest file, but at this point, it will have no effect. But first, we will explain how the Ingress HTTP routing rules are built. Each of these rules in the specification contains the following:

Optional host: In the example, we are not using this field, so the rule that we defined is applied to all incoming traffic. If the field value is provided, then the rule applies only to requests that have this host as the destination – you can have multiple hostnames resolving to the same IP address. The host field supports wildcards.

List of path routings: Each of the paths has an associated Ingress backend that you define by providing serviceName and servicePort. In the preceding example, all requests arriving at the path with the prefix /service1 will be routed to Pods of the example-service1 Service, and all requests arriving at the path with the prefix /service2 will be routed to Pods of the example-service2 Service. The path fields support prefixes and exact matching, and it is also possible to use implementation-specific matching, which is carried out by the underlying Ingress Controller.

In this way, you can configure complex routing rules that involve multiple Services in the cluster, but externally they will be visible as a single endpoint with multiple paths available. This is especially useful when you create API gateways in microservice architecture for frontend or client applications.

To materialize Ingress objects, we need to have an Ingress Controller installed in the cluster.

## Using nginx as an Ingress Controller

An Ingress Controller is a Kubernetes controller that is deployed manually to the cluster, most often as a DaemonSet or a Deployment object that runs dedicated Pods for handling incoming traffic load balancing and smart routing. It is responsible for processing the Ingress objects (which specify that they especially want to use the Ingress Controller) and dynamically configuring real routing rules. A commonly used Ingress controller for Kubernetes is ingress-nginx (https://www.nginx.com/products/nginx/kubernetes-ingress-controller), which is installed in the cluster as a Deployment of an nginx web host with a set of rules for handling Ingress API objects. The Ingress Controller is exposed as a Service with a type that depends on the installation – in cloud environments, this will be LoadBalancer.

The installation of ingress-nginx is described for different environments in the official documentation: https://kubernetes.github.io/ingress-nginx/deploy/. Note that it is also possible to use Helm to install this Ingress Controller, which makes management and upgrades a bit easier. For cloud environments, the installation is usually very simple and involves applying a single YAML manifest file, which creates multiple Kubernetes objects.

$ kubectl apply -f

https://raw.githubusercontent.com/kubernetes/ingress-nginx/master/deploy/static/provider/cloud/deploy.yaml

namespace/ingress-nginx created

serviceaccount/ingress-nginx created

configmap/ingress-nginx-controller created

...

Now, we can also create our example Services together with the Ingress object that defines routings for them. First, we will create the Service objects and Deployment objects. There is no need to explain the YAML manifests as they are simple web servers printing out a welcome message with information on which Service you have reached:

$ kubectl apply -f https://raw.githubusercontent.com/PacktPublishing/Kubernetes-for-Beginners/master/Chapter21/02\_ingress/example-services.yaml

deployment.apps/example-service1 created

deployment.apps/example-service2 created

service/example-service1 created

service/example-service2 created

Next, we can apply the Ingress object to the cluster that we created earlier:

$ kubectl apply -f ./example-ingress.yaml

ingress.networking.k8s.io/example-ingress created

By this time, the LoadBalancer Service that is running as part of Ingress Controller should already be functional and have an external IP address available. You can get it using the following command:

$ kubectl describe svc -n ingress-nginx ingress-nginx-controller

...

LoadBalancer Ingress: 137.117.227.83

...

At this point, we can visualize what is happening behind Ingress Controller in the following diagram:

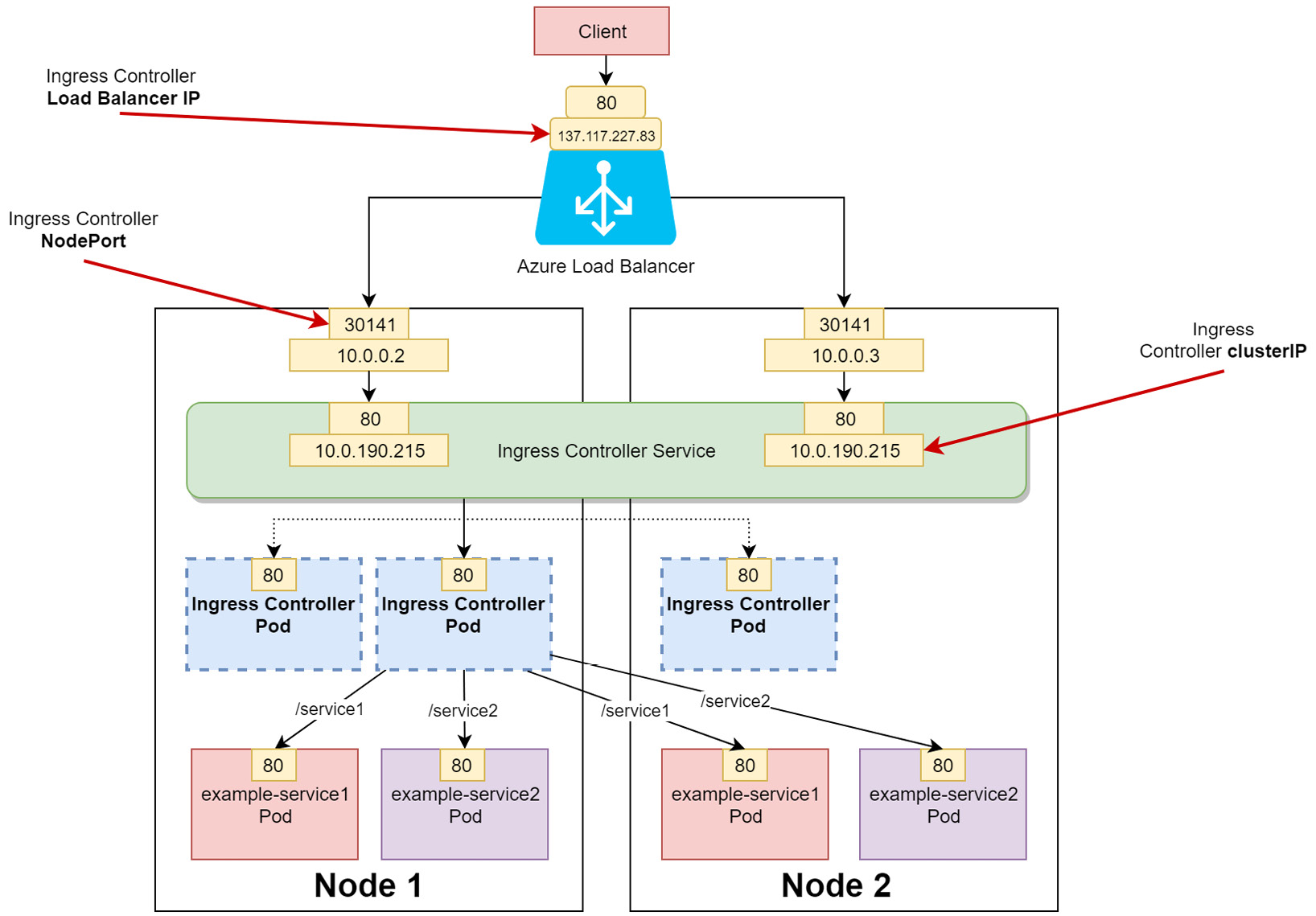


Figure – Using nginx as Ingress Controller in a cloud environment

When you perform an HTTP request to http://<ingressServiceLoadBalancerIp>/service1, the traffic will be routed by nginx to example-service1. Similarly, when you use the /service2 path, the traffic will be routed to example-service2. Note that you are using only one cloud load balancer for this operation, and that the actual routing to Kubernetes Services is performed by the Ingress Controller Pods using path-based routing.

In practice, you need to set up SSL certificates for your HTTP endpoints to ensure proper security. In our examples, we are not doing that for simplicity and to make the demonstrations clearer.

Let's verify this in practice. In your web browser, navigate to the /service1 path. In our case, this will be http://137.117.227.83/service1. You will see that you are served by example-service1 Pods:

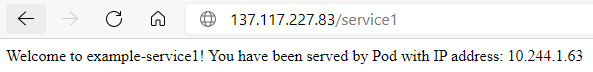


Figure – Routing to example-service1 via the nginx Ingress Controller

Now, navigate to the /service2 path. In our case, this will be http://137.117.227.83/service2. You will see that you are served by example-service2 Pods:

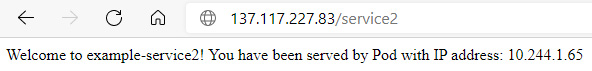


Figure – Routing to example-service2 via the nginx Ingress Controller

Congratulations! You have successfully configured Ingress and Ingress Controller in your cluster.

Summary

We have explained advanced traffic routing approaches in Kubernetes using Ingress objects and Ingress Controllers. At the beginning, we did a brief recap of Kubernetes Service types. We refreshed our knowledge regarding ClusterIP, NodePort, and LoadBalancer Service objects. Based on that, we introduced Ingress objects and Ingress Controller and explained how they fit into the landscape of traffic routing in Kubernetes. Now, you know that simple Services are commonly used when L4 load balancing is required, but if you have HTTP or HTTPS endpoints in your applications, it is better to use L7 load balancing offered by Ingress and Ingress Controllers. You learned how to deploy the nginx web server as Ingress Controller and we tested this on example Deployments.

For more information regarding autoscaling in Kubernetes, please refer to the following Packt books:

The Complete Kubernetes Guide, by Jonathan Baier, Gigi Sayfan, Jesse White (https://www.packtpub.com/virtualization-and-cloud/complete-kubernetes-guide)

Getting Started with Kubernetes – Third Edition, by Jonathan Baier, Jesse White (https://www.packtpub.com/virtualization-and-cloud/getting-started-kubernetes-third-edition)

Kubernetes for Developers, by Joseph Heck (https://www.packtpub.com/virtualization-and-cloud/kubernetes-developers)

You can also refer to the following official documentation:

Kubernetes documentation (https://kubernetes.io/docs/home/), which is always the most up-to-date source of knowledge regarding Kubernetes in general.

A list of many available Ingress Controllers can be found at the following link: https://kubernetes.io/docs/concepts/services-networking/ingress-controllers/.

For Amazon EKS, there is AWS Load Balancer Controller. You can find more information in the official documentation: https://docs.aws.amazon.com/eks/latest/userguide/alb-ingress.html

# Starting a message queue service

Reference Link: <https://kubernetes.io/docs/tasks/job/coarse-parallel-processing-work-queue/>

# Fine Parallel Processing Using a Work Queue

<https://kubernetes.io/docs/tasks/job/fine-parallel-processing-work-queue/>

In practice you could set up a message queue service once in a cluster and reuse it for many jobs, as well as for long-running services.

Start RabbitMQ as follows:

kubectl create -f <https://raw.githubusercontent.com/kubernetes/kubernetes/release-1.3/examples/celery-rabbitmq/rabbitmq-service.yaml>

service "rabbitmq-service" created

kubectl create -f <https://raw.githubusercontent.com/kubernetes/kubernetes/release-1.3/examples/celery-rabbitmq/rabbitmq-controller.yaml>

replicationcontroller "rabbitmq-controller" created

1. helm repo add airflow-stable<https://airflow-helm.github.io/charts>
2. helm repo update
3. helm install RELEASE\_NAME airflow-stable/airflow

--namespace NAMESPACE \

--version CHART\_VERSION

1. helm install --generate-name airflow-stable/airflow --namespace spindle-nuance --version [8.6.1](https://github.com/airflow-helm/charts/releases/tag/airflow-8.6.1)
   1. **RELEASE\_NAME is airflow-1666420923**

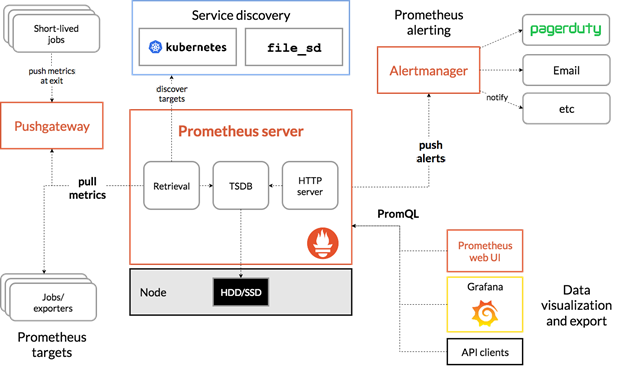
# Prometheus Introduction

1. An open-source monitoring system with a dimensional data model, flexible query language, efficient time series database and modern alerting approach (i.e., metrics information is stored with the timestamp at which it was recorded, alongside optional key-value pairs called labels).
2. Prometheus comes with its language called “PromQL” to query the metrics it scrapes from various sources.
3. Target endpoints can be added to Prometheus via Service Discovery, using node exporters, through static configuration, or client libraries.

## Prometheus Architecture

Prometheus mostly works through a Pull-based model of gathering data exposed via HTTP endpoints of the targets. A push-based data gathering can be achieved through a Push-gateway setup. Prometheus architecture can be divided into **five** components.

1. The central **Prometheus Server** is the heart of the monitoring system.
2. **Targets:** which are the producers of metrics, the metric data on them are exposed to the Prometheus Server via Client Libraries (To access application code-related metrics available in multiple supported languages. These libraries format the metrics in Prometheus readable form), Exporters (are used when the application source code is not accessible, they live alongside the application. Ex: Database Exporter, Node Exporter for host stats, HaProxy Exporter for storage, messaging & HTTP).
3. **Service Discovery:** a mechanism for discovering services like EC2 instances and K8S on the cloud and scraping data from them.
4. **Storage:** Prometheus persists the data on the local host or in a cloud environment; an external persistent storage volume can be used.
5. **Alerting & Visualizing:** The expression browser is a web-based UI where these metrics can be visualized in a central location. Using the Alertmanager component of Prometheus, alerts can be set and managed.



## MONITORING LINUX HOST METRICS WITH THE NODE EXPORTER

Node Exporter is **a Prometheus exporter for server level and OS level metrics with configurable metric collectors**. It helps us in measuring various server resources such as RAM, disk space, and CPU utilization.

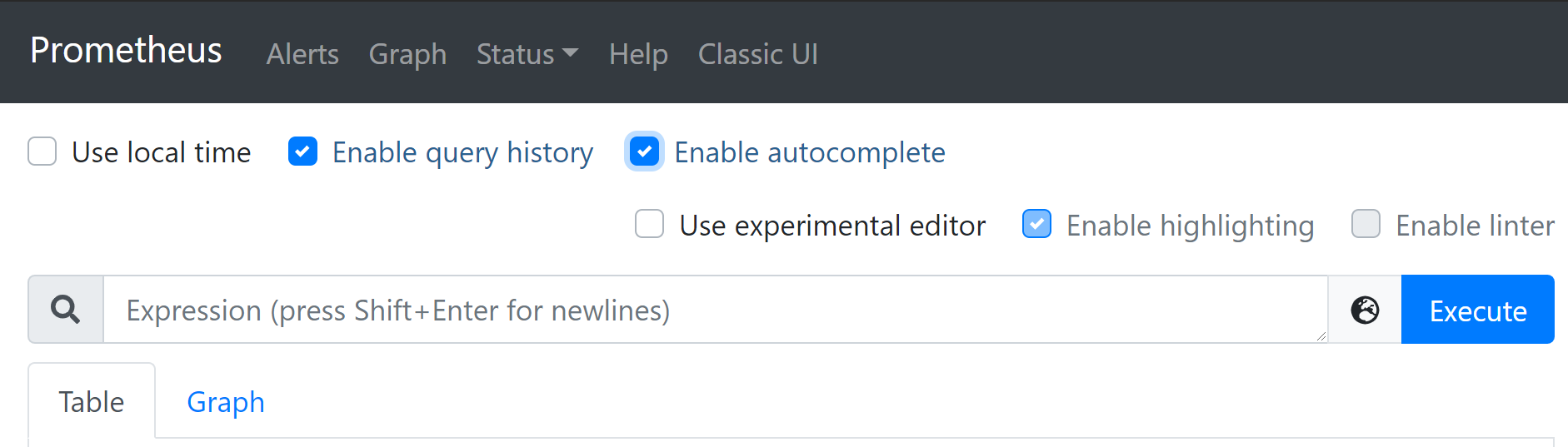
The Prometheus Node Exporter exposes a wide variety of hardware- and kernel-related metrics.

[if node is inside EKS/Kubernetes cluster, Node Exporter can be installed as a DaemonSet kind in the kubernetes cluster]

1. cd opt
2. touch node\_exporter\_download.tar.gz
3. sudo wget -O /opt/node\_exporter\_download.tar.gz https://github.com/prometheus/node\_exporter/releases/download/v1.4.0/node\_exporter-1.4.0.linux-amd64.tar.gz
4. sudo tar xf node\_exporter\_download.tar.gz
5. sudo mv node\_exporter-1.4.0.linux-amd64 node\_exporter
6. cd node\_exporter
7. ./node\_exporter
8. open 9100 tcp in security group
9. <<public ip or dns>>:9100/metrics (Once the Node Exporter is installed and running, you can verify that metrics are being exported by browsing or cURLing the /metrics endpoint)

### Exploring Node Exporter metrics through the Prometheus expression browser

Now that Prometheus is scraping metrics from a running Node Exporter instance, you can explore those metrics using the Prometheus UI (aka the expression browser). Navigate to <<ip or dns of prometheus>>:9090/graph in your browser and use the main expression bar at the top of the page to enter expressions. The expression bar looks like this:



Metrics specific to the Node Exporter are prefixed with node\_ and include metrics like node\_cpu\_seconds\_total and node\_exporter\_build\_info.

| **Metric** | **Meaning** |
| --- | --- |
| rate(node\_cpu\_seconds\_total{mode="system"}[1m]) | The average amount of CPU time spent in system mode, per second, over the last minute (in seconds) |
| node\_filesystem\_avail\_bytes | The filesystem space available to non-root users (in bytes) |
| rate(node\_network\_receive\_bytes\_total[1m]) | The average network traffic received, per second, over the last minute (in bytes) |

# Built-in Grafana Dashboard

Grafana community has developed around 5800 dashboards (still growing). We can use it in our product. You can filter by category, panel, datasource and community social feed like rating/download/review. You can import that into your existing grafana in the cluster.

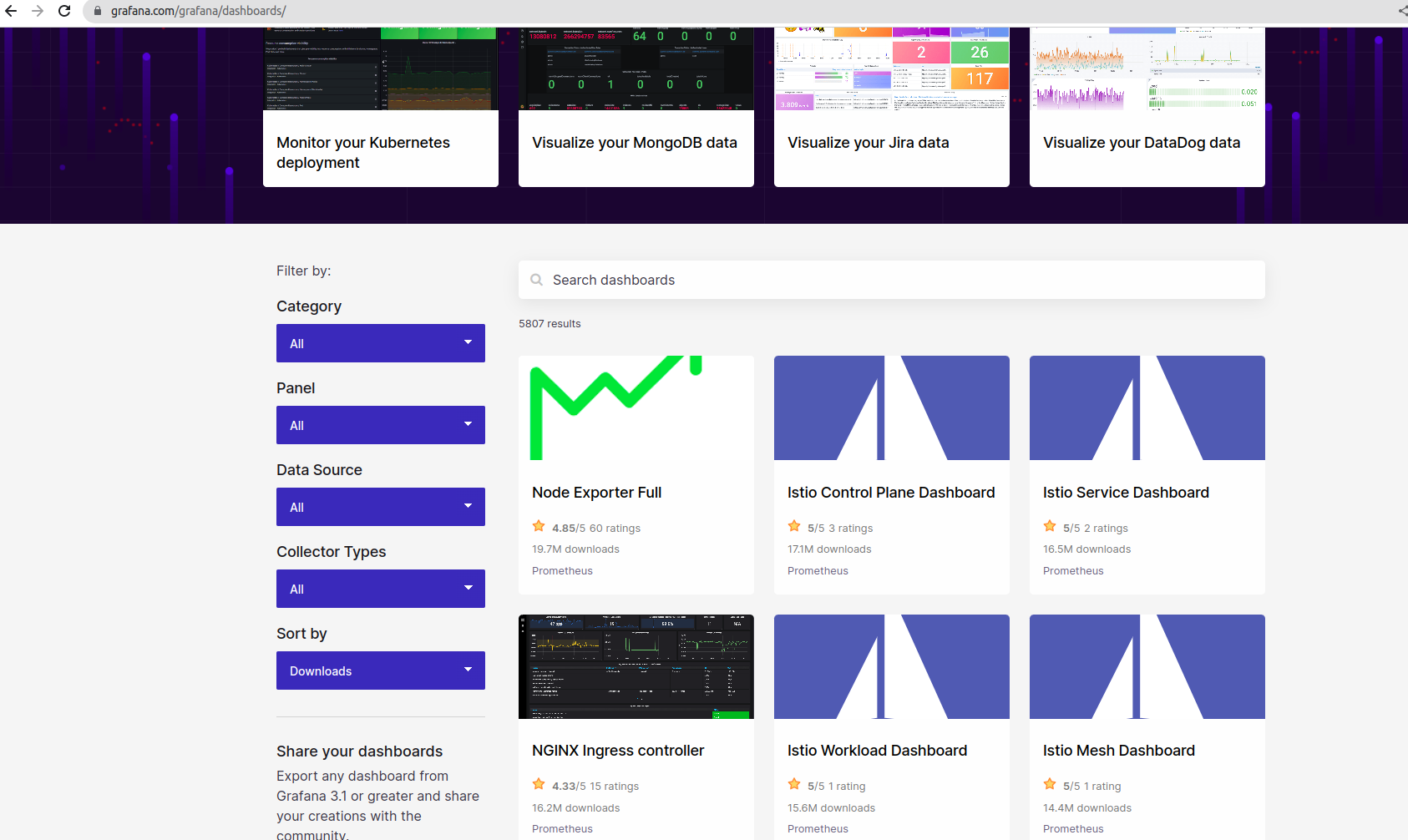
Every dashboard has a unique dashboard id, for instance, <https://grafana.com/grafana/dashboards/1860-node-exporter-full/> gnetId is the key in the json model.

"description": "Node Exporter Full Prometheus Dashboard",

"editable": true,

"fiscalYearStartMonth": 0,

"gnetId": 1860,



# Create a Docker Credentials file

In order for Kubernetes to use the credentials, we need to first give it the credentials, and then assign those credentials to either the service account that will be used to pull the images, or specify them directly on the deployment files that need to pull these images.

So first let's create the secret.

The format of the secret is in the format of a .dockerconfigjson file.

The general format of this file is

{

"auths": {

"https://registry.gitlab.com":{

"username":"REGISTRY\_USERNAME",

"password":"REGISTRY\_PASSWORD",

"email":"REGISTRY\_EMAIL", //We didn’t use email

"auth":"BASE\_64\_BASIC\_AUTH\_CREDENTIALS (see below)"

}

}

}

BASE\_64\_BASIC\_AUTH\_CREDENTIALS:

The base 64 basic credentials mentioned above are the username and password in basic credentials format username:password , encoded with base64 format.

To achieve this simply run :

echo -n "REGISTRY\_USERNAME:REGISTRY\_PASSWORD" | base64

Create a file with above mentioned json format, and then base64 encode it for the Kubernetes secret.

cat .dockerconfigjson | base64

This will output the base64 you need for the registry secret.

## Create a Kubernetes Secret

Create a file called registry-credentials.yml

# KEDA - scale through external metrics

kubectl apply -f <https://github.com/kedacore/keda/releases/download/v2.8.1/keda-2.8.1.yaml>

namespace/keda created

customresourcedefinition.apiextensions.k8s.io/clustertriggerauthentications.keda.sh created

customresourcedefinition.apiextensions.k8s.io/scaledjobs.keda.sh created

customresourcedefinition.apiextensions.k8s.io/scaledobjects.keda.sh created

customresourcedefinition.apiextensions.k8s.io/triggerauthentications.keda.sh created

serviceaccount/keda-operator created

clusterrole.rbac.authorization.k8s.io/keda-external-metrics-reader created

clusterrole.rbac.authorization.k8s.io/keda-operator created

rolebinding.rbac.authorization.k8s.io/keda-auth-reader created

clusterrolebinding.rbac.authorization.k8s.io/keda-hpa-controller-external-metrics created

clusterrolebinding.rbac.authorization.k8s.io/keda-operator created

clusterrolebinding.rbac.authorization.k8s.io/keda-system-auth-delegator created

service/keda-metrics-apiserver created

deployment.apps/keda-metrics-apiserver created

deployment.apps/keda-operator created

apiservice.apiregistration.k8s.io/v1beta1.external.metrics.k8s.io created

# Reference:

1. Reference - Terraform Skill Development

<https://learn.hashicorp.com/tutorials/terraform/blue-green-canary-tests-deployments>

<https://registry.terraform.io/modules/terraform-aws-modules/eks/aws/latest>

<https://www.youtube.com/watch?v=rv6MOvpXU90>

<https://www.clickittech.com/devops/terraform-kubernetes-deployment/>

<https://www.youtube.com/watch?v=rv6MOvpXU90>

<https://cloud.google.com/docs/terraform/best-practices-for-terraform>

<https://www.terraform.io/language/modules/develop/refactoring>

<https://www.terraform.io/language>

Reference - Terraform with Gitlab

<https://blog.wimwauters.com/devops/2019-11-23-aws_terraform_gitlab/>

<https://about.gitlab.com/topics/gitops/gitlab-enables-infrastructure-as-code>

<https://dev.to/stack-labs/deploying-production-ready-gitlab-on-amazon-eks-with-terraform-3coh>

Reference – Terraform State Lock

https://jhooq.com/terraform-state-file-locking/

Reference – Kubernetes Course

<https://www.youtube.com/watch?v=X48VuDVv0do>

<https://www.youtube.com/watch?v=EQNO_kM96Mo>

Reference - Cluster Autoscaling – creation:

<https://hands-on.cloud/how-to-set-up-amazon-eks-cluster-using-terraform/#h-creating-cluster-autoscaler-iam-role-using-terraform>

Reference - Scaling user stories

https://github.com/kubernetes/enhancements/blob/master/keps/sig-autoscaling/853-configurable-hpa-scale-velocity/README.md

Reference – Mapping document between docker compose and kubernetes

https://github.com/docker/compose-on-kubernetes/blob/master/docs/mapping.md

Reference – PV and PVC

https://www.containiq.com/post/kubernetes-persistent-volumes

Reference – Few steps to troubleshoot the Pod

https://www.youtube.com/watch?v=FMUAuNrbpcI

Reference - Kubectl

<https://www.containiq.com/post/kubectl-cheat-sheet>

Reference - AWS CLI MFA

<https://www.youtube.com/watch?v=BNpbGHhk5Tc>

Reference - Prometheus and Grafana

<https://github.com/prometheus/cloudwatch_exporter>

<https://github.com/bibinwilson/kubernetes-prometheus>

<https://gitlab.com/xavki/presentation-prometheus-grafana>

<https://github.com/gurpreet0610/Deploy-Prometheus-Grafana-on-Kubernetes>

<https://medium.com/@gurpreets0610/deploy-prometheus-grafana-on-kubernetes-cluster-e8395cc16f91>

References - Cost Savings for EKS

<https://www.middlewareinventory.com/blog/terraform-eks-spot-instances>

<https://aws.amazon.com/blogs/containers/cost-optimization-for-kubernetes-on-aws/>

<https://aws.amazon.com/blogs/compute/cost-optimization-and-resilience-eks-with-spot-instances/>

<https://aws.amazon.com/blogs/containers/saving-money-pod-at-time-with-eks-fargate-and-aws-compute-savings-plans/>

<https://codeberg.org/hjacobs/kube-downscaler>

AWS workshop

<https://tf-eks-workshop.workshop.aws/500_eks-terraform-workshop/560_nodegroup/nodeg-tf.html>

<https://medium.com/@ealexhaywood/automate-gitlab-issue-creation-with-prometheus-and-gitlab-alerts-76fe1177eacb>

<https://dev.to/aws-builders/apache-airflow-in-eks-cluster-dgo>

<https://learnk8s.io/scaling-celery-rabbitmq-kubernetes>

<https://www.linkedin.com/pulse/exposing-multiple-portsservices-same-load-balancer-sunil-agarwal/>

<https://matthewpalmer.net/kubernetes-app-developer/articles/kubernetes-ingress-guide-nginx-example.html>

<https://www.youtube.com/watch?v=GhZi4DxaxxE>

<https://www.stacksimplify.com/aws-eks/aws-alb-ingress/learn-aws-alb-ingress-on-aws-eks/>

<https://github.com/PacktPublishing/The-Kubernetes-Bible/tree/master/Chapter21>

Cloud Native Learning

<https://github.com/cncf/curriculum.git>

<https://github.com/antonputra/tutorials/tree/main/lessons>

## [**Step 1 - Install NGINX Ingress Controller using Helm**](https://platform9.com/learn/tutorials/nginix-controller-helm#step-1---install-nginx-ingress-controller-using-helm)

An ingress controller, because it is a core component of Kubernetes, requires specific configuration to be performed at the cluster level as part of installation. Lets look into what happens behind the scenes when you install NGINX ingress using helm. If you'd like to skip this, navigate to the bottom of this section to see the actual steps to install the NGINIX ingress controller.

The recommended configuration for NGINX uses three Kubernetes ConfigMaps:

* Base Deployment
* TCP configuration
* UDP configuration

A Kubernetes service account is required to run NGINX as a service within the cluster. The service account needs to have following roles:

* A cluster role to allow it to get, list, and read the configuration of all services and events. This role could be limited if you were to have multiple ingress controllers installed within the cluster. But in most cases, limiting access for this service account may not be needed.
* A namespace-specific role to read and update all the ConfigMaps and other items that are specific to the NGINX Ingress controller’s own configuration.

To install an NGINX Ingress controller using Helm, add the nginx-stable repository to helm, then run helm repo update . After we have added the repository we can deploy using the chart nginx-stable/nginx-ingress.

1. helm repo add nginx-stable <https://helm.nginx.com/stable>
2. helm repo update
3. helm install nginx-ingress nginx-stable/nginx-ingress --set rbac.create=true -n monitoring

## [Step 2 - Validate that NGINX is Running](https://platform9.com/learn/tutorials/nginix-controller-helm#step-2---validate-that-nginx-is-running)

Make sure the NGINX Ingress controller is running.

## [Step 3 - Exposing Services using NGINX Ingress Controller](https://platform9.com/learn/tutorials/nginix-controller-helm#step-3---exposing-services-using-nginx-ingress-controller)

Now that an ingress controller is running in the cluster, you will need to create services that leverage it using either host, URI mapping, or even both.

Sample of a host-based service mapping through an ingress controller using the type “Ingress”:

<https://kubernetes.github.io/ingress-nginx/deploy/#aws>

<https://www.fairwinds.com/blog/intro-to-kubernetes-ingress-set-up-nginx-ingress-in-kubernetes-bare-metal>

<https://devopscube.com/setup-ingress-kubernetes-nginx-controller/>

<https://github.com/scriptcamp/nginx-ingress-controller>

<https://towardsdatascience.com/how-to-set-up-ingress-controller-in-aws-eks-d745d9107307>

<https://kubernetes.io/docs>

[https://helm.sh](https://helm.sh/)

<https://aws.amazon.com/eks/>

[https://eksctl.io](https://eksctl.io/)

<https://github.com/cncf/curriculum>

<https://kubernetes-sigs.github.io/aws-load-balancer-controller/v2.4/>

<https://github.com/aws/eks-charts>

[https://artifacthub.io](https://artifacthub.io/)

<https://www.youtube.com/watch?v=ZfjpWOC5eoE>

<https://github.com/antonputra/tutorials/tree/main/lessons/112>

--set ingress.enabled=true

--set ingress.ingressClassName=“internet-facing-ingress”

--set ingress.hosts=“your host name”

[alb.ingress.kubernetes.io/scheme](http://alb.ingress.kubernetes.io/scheme): internet-facing

[kubernetes.io/ingress.class](http://kubernetes.io/ingress.class): alb

[alb.ingress.kubernetes.io/certificate-arn](http://alb.ingress.kubernetes.io/certificate-arn): arn:aws:acm:us-east-1:594977439087:certificate/578df8b2-7df0-4fb8-ba4d-7b4261cc1cb9

[alb.ingress.kubernetes.io/listen-ports](http://alb.ingress.kubernetes.io/listen-ports): '[{"HTTP": 80}, {"HTTPS":443}]'

alb.ingress.kubernetes.io/actions.viewer-redirect: '{"Type": "redirect", "RedirectConfig": { "Protocol": "HTTPS", "Port": "443", "Path":"/", "Query": "#{query}", "StatusCode": "HTTP\_301"}}'

[alb.ingress.kubernetes.io/actions.ssl-redirect](http://alb.ingress.kubernetes.io/actions.ssl-redirect): '{"Type": "redirect", "RedirectConfig": { "Protocol": "HTTPS", "Port": "443", "StatusCode": "HTTP\_301"}}'

[external-dns.alpha.kubernetes.io/hostname](http://external-dns.alpha.kubernetes.io/hostname):echo.radiolensdashboard.com