Kubernetes Design Handbook

Prerequisites

➔ RHEL (Amazon Linux 2)

```
sudo yum install -y git curl wget unzip jq
```

➔ Debian (Ubuntu)

```
sudo apt-get install -y git curl wget unzip jq
```

➔ Manjaro (Archlinux)

```
sudo pacman -Syu git curl wget unzip jq --noconfirm
```

➔ eksctl v0.69.0

```
cd /tmp
wget
https://github.com/weaveworks/eksctl/releases/download/v0.69.0/eksctl_Linux
_amd64.tar.gz
tar -xzf eksctl_Linux_amd64.tar.gz
sudo mv eksctl /usr/local/bin/eksctl
sudo chmod +x /usr/local/bin/eksctl
sudo ln -s /usr/local/bin/eksctl /usr/bin/eksctl
eksctl version

# bash completion for eksctl (optional)
echo 'source <(eksctl completion bash)' >>~/.bashrc
```

Note that it is necessary to pick the correct version of the eksctl as it is a very active repository that gets alot of new features and changes everyday.

➔ kubectl v1.21.2

```
cd /tmp
curl -LO https://dl.k8s.io/release/v1.21.2/bin/linux/amd64/kubectl
sudo mv kubectl /usr/local/bin/kubectl
sudo chmod +x /usr/local/bin/kubectl
sudo ln -s /usr/local/bin/kubectl /usr/bin/kubectl
kubectl version --client --short

# bash completion for kubectl (optional)
echo 'source <(kubectl completion bash)' >>~/.bashrc
kubectl completion bash >/etc/bash_completion.d/kubectl
```

Note that it is necessary to install correct version of kubectl as different kubernetes versions has different api classes for different components.

➔ helm 3.7.1

```
cd /tmp
wget https://get.helm.sh/helm-v3.7.1-linux-amd64.tar.gz
tar -xzf helm-v3.7.1-linux-amd64.tar.gz
sudo mv linux-amd64/helm /usr/local/bin/helm
sudo chmod +x /usr/local/bin/helm
sudo ln -s /usr/local/bin/helm /usr/bin/helm
helm version --short

# bash completion for helm (optional)
echo 'source <(helm completion bash)' >>~/.bashrc
```

Walkthrough

➔ Export these environment variables for easier and understandable setup process. It is necessary to use single terminal session or re-export the environment variables

```
export CLUSTERNAME=[bidclips-dev/qa/uat/prod]
export REGION=[us-east-1/ap-south-1]
export AWSACCOUNTID=$(aws sts get-caller-identity --query Account  | cut -d
"\"" -f 2 | cut -d "\"" -f 1)  # 12 digit account id
```

"bidclips" is taken as project name for this walkthrough

➔ Prepare a manifest file for kubernetes cluster

```
eksctl create cluster -f eks-create-cluster.yaml
```

to verify the created nodes execute `kubectl get nodes`
See these reference [1] [2]

Extras: In case of scaling needs, execute this

```
eksctl scale nodegroup --cluster=$CLUSTERNAME --nodes=3 --name=NodeGroup-Name
--nodes-min=2 --nodes-max=4
```

cluster name and nodegroup name are **C**ase **S**ensitive.

➔ **metrics server:** to grab the resources metrics from the cluster to present on kubernetes dashboard

```
DOWNLOAD_URL=$(curl --silent
"https://api.github.com/repos/kubernetes-sigs/metrics-server/releases/latest" | jq -r .tarball_url)
DOWNLOAD_VERSION="v$(grep -o '[^/v]*$' <<<  $DOWNLOAD_URL)"

kubectl apply -f
https://github.com/kubernetes-sigs/metrics-server/releases/download/$DOWNLOAD_VERSION/components.yaml
```

to verify that metrics server is up and running `kubectl get deployment metrics-server -n kube-system`

➔ **kubernetes dashboard:** handy tool to take a look at the kubernetes cluster and its components, more transparently

```
export DASHBOARD_RELEASE=v2.4.0
kubectl apply -f
https://raw.githubusercontent.com/kubernetes/dashboard/$DASHBOARD_RELEASE/aio/deploy/recommended.yaml

kubectl apply -f admin-service-account.yaml
```

choose dashboard version which is compatible with eks kubernetes version from here

Ref to admin-service-account.yaml here
Creating an admin service account is essential because that's what's going to be used to authorise the request to access kubernetes dashboard.

verify that components are ready before proceed further `kubectl -n kubernetes-dashboard get deployment`

start kubernetes proxy to access dashboard

```
kubectl proxy
```

Click me to access dashboard.

➔ **fetch token for admin service account to login to dashboard** [1]

```
kubectl -n kubernetes-dashboard get secret $(kubectl -n
kubernetes-dashboard get sa/eks-admin -o
jsonpath="{.secrets[0].name}") -o go-template="{{.data.token |
base64decode}}"
```

➔ **Setup ALB LoadBalancer Controller (formerly ALB Ingress Controller)**

◆ **IAM Permissions** [1]

1. Create IAM OIDC provider

```
eksctl utils associate-iam-oidc-provider \
  --region $REGION \
  --cluster $CLUSTERNAME \
  --approve
```

2. Download IAM policy for the AWS Load Balancer Controller

```
curl -o iam-policy.json
https://raw.githubusercontent.com/kubernetes-sigs/aws-load-balancer-control
ler/v2.2.4/docs/install/iam_policy.json
```

3. Create an IAM policy called **AWSLoadBalancerControllerIAMPolicy**

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

4. Create a IAM role and ServiceAccount for the AWS Load Balancer controller, use the ARN from the step above

```
eksctl create iamserviceaccount \
  --cluster=$CLUSTERNAME \
  --namespace=kube-system \
  --name=aws-load-balancer-controller \

--attach-policy-arn=arn:aws:iam::$AWSACCOUNTID:policy/AWSLoadBalancerContro
llerIAMPolicy \
  --override-existing-serviceaccounts \
  --approve
```

- **Add Controller to Cluster** [1]

```
export VPCID=$(echo $(aws cloudformation describe-stack-resources
--stack-name eksctl-$CLUSTERNAME-cluster --logical-resource-id VPC --region
$REGION --output json --query "StackResources[0].PhysicalResourceId") | cut
-d "\"" -f 2 | cut -d "\"" -f 1)

echo $VPCID  # confirm that the variable got correct VPC ID
```

1. Add the EKS chart repo to helm

```
helm repo add eks https://aws.github.io/eks-charts
```

2. Install the TargetGroupBinding CRDs if upgrading the chart via `helm upgrade`.

```
kubectl apply -k
"github.com/aws/eks-charts/stable/aws-load-balancer-controller//crds?
ref=master"
```

3. Install the helm chart if using IAM roles for service accounts. NOTE you need to specify both of the chart values `serviceAccount.create=false` and `serviceAccount.name=aws-load-balancer-controller`

```
helm install aws-load-balancer-controller
eks/aws-load-balancer-controller \
  -n kube-system \
  --set clusterName=$CLUSTERNAME \
  --set serviceAccount.create=false \
  --set serviceAccount.name=aws-load-balancer-controller \
  --set vpcId=$VPCID \
  --set region=$REGION
```

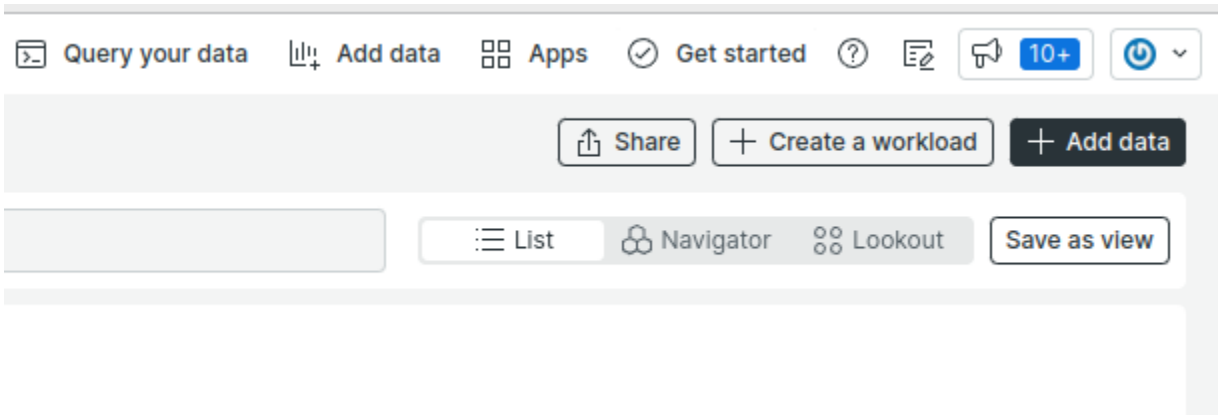
verify the alb ingress controller installation by executing

```
kubectl -n kube-system logs $(kubectl get pods -n kube-system | grep
aws-load-balancer-controller | sed -n 1p | awk '{print $1}')
```

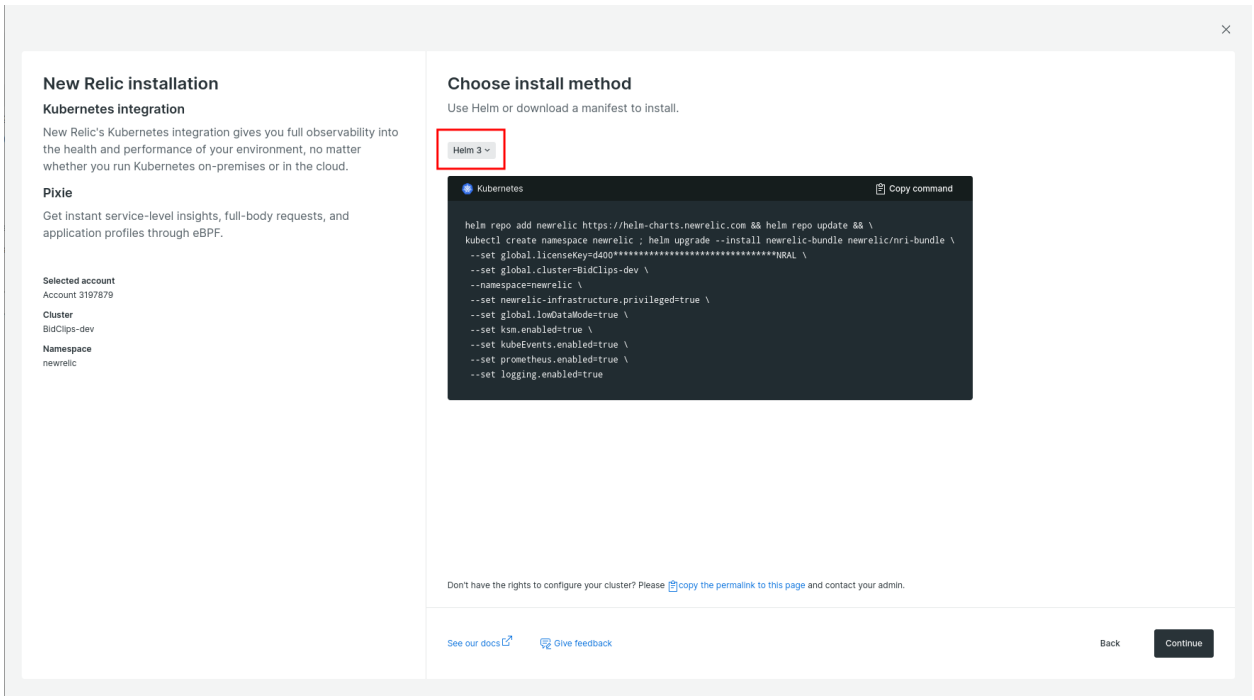
Logging, Monitoring & Alerting

➔ **NewRelic:** It's a pay-as-you-store-your-data kind of service and it's a full fledged Logging, Monitoring, Alerting and Incident Management suite. (and more)

1. Visit https://one.newrelic.com and create an account



2. Click on "Add data" and search "Kubernetes", choose the options as per the need and select "Helm 3" as an installation method



3. Execute the commands it shows and wait till the newrelic agents start sending telemetry data and logs to NewRelic.

➔ **PLG:** Stands for Promtail-Loki-Grafana, is a logging & alerting stack that can be implemented inside the existing kuberntes cluster. All the components will reside inside the cluster itself. It is a stable and robust option to cut costs but is not recommended for production-grade environments and if you need something ironclad.

Create a separate namespace for this stack

```
kubectl create namespace plg-stack
```

Install PLG stack

```
helm repo add loki https://grafana.github.io/loki/charts

helm repo update

helm upgrade --install loki loki/loki-stack \
  --namespace=plg-stack \
  --set
grafana.enabled=true,prometheus.enabled=false,prometheus.alertmanager.persi
stentVolume.enabled=false,prometheus.server.persistentVolume.enabled=false,
loki.persistence.enabled=true,loki.persistence.size=100Gi,grafana.persisten
ce.enabled=true,grafana.persistence.size=20Gi,loki.config.table_manager.ret
ention_deletes_enabled=true,loki.config.table_manager.retention_period=144h
```

Fetch grafana password

```
kubectl get secret loki-grafana --namespace=plg-stack -o
jsonpath="{.data.admin-password}" | base64 --decode ; echo
```

Change the `loki-grafana` service type to NodePort

```
ports:
  - name: http
    nodePort: 30098
    port: 80
    protocol: TCP
    targetPort: 3000
  type: NodePort
```

Create an ingress controller/create a rule in your existing ingress controller to route the request to PLG stack and access it from a domain/subdomain

Creating Kubernetes Components

There are four fundamental hence mandatory components of Kubernetes component manifest file. These four components can be found in any of the definitions of the Kubernetes component

1. apiVersion
2. kind
3. metadata
4. spec

➔ **Namespaces**[1]**:** In Kubernetes, namespaces provides 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).

- Create Kubernetes namespace by executing

```
kubectl create namespace app-stack
```

➔ **Deployments**[1]**:** A Deployment provides declarative updates for Pods and ReplicaSets.

- When the component is stateless and it requires to have multiple replicas which are self-managed, the Deployment component is the one
- The easiest way to create a component scaffolding is

```
kubectl create deployment nginx --image nginx --replicas 2 --namespace
default --port 80 --dry-run client --output=yaml
```

- This will print a version-specific Kubernetes deployment component structure which you can modify as your need

```
apiVersion: apps/v1
kind: Deployment
metadata:
  labels:
    app: nginx
  name: nginx
  namespace: default
spec:
  replicas: 2
  selector:
    matchLabels:
      app: nginx
  template:
    metadata:
      labels:
```

```
      app: nginx
  spec:
    containers:
    - image: nginx
      name: nginx
      ports:
      - containerPort: 80
```

➔ **Services**[1]**:** An abstract way to expose an application running on a set of Pods as a network service. With Kubernetes, you don't need to modify your application to use an unfamiliar service discovery mechanism. Kubernetes gives Pods their IP addresses and a single DNS name for a set of Pods and can load-balance across them.

- ◆ The easiest way to scaffold a service is

```
kubectl create service nodeport nginx --tcp 80 --dry-run=client -o yaml
```

```
apiVersion: v1
kind: Service
metadata:
  creationTimestamp: null
  labels:
    app: nginx
  name: nginx
spec:
  ports:
  - name: "80"
    port: 80
    protocol: TCP
    targetPort: 80
  selector:
    app: nginx
  type: NodePort
```

- ◆ It will print a skeleton of the service component you can modify it as per need

➔ **ConfigMaps**[1]**:** A ConfigMap is an API object used to store non-confidential data in key-value pairs. Pods can consume ConfigMaps as environment variables, command-line arguments, or as configuration files in a volume. A ConfigMap allows you to decouple environment-specific configuration from your container images so that your applications are easily portable.

- ConfigMaps can be created using a single file or a single directory, using the following commands

```
kubectl create configmap app-config --from-file nginx.yaml
--dry-run=client -o yaml
```

- It will scaffold a basic ConfigMap component, which you can modify as needed

```
apiVersion: v1
data:
  nginx.yaml: |
    apiVersion: apps/v1
    kind: Deployment
    metadata:
      labels:
        app: nginx
      name: nginx
      namespace: app-stack
    spec:
      replicas: 2
      selector:
        matchLabels:
          app: nginx
      template:
        metadata:
          labels:
            app: nginx
        spec:
          containers:
          - image: nginx
            name: nginx
            ports:
            - containerPort: 80
kind: ConfigMap
metadata:
  name: app-config
```

➔ **Ingress**[1]**:** An API object that manages external access to the services in a cluster, typically HTTP. Ingress may provide load balancing, SSL termination, and name-based virtual hosting.

➔ **DaemonSets**[1]**:** A DaemonSet ensures that all (or some) Nodes run a copy of a Pod. As nodes are added to the cluster, Pods are added to them. As nodes are removed from the cluster, those Pods are garbage collected. Deleting a DaemonSet will clean up the Pods it created. The best use-case of a daemonset is a log collector and metrics agent.

➔ **StatefulSets**[1]**:** StatefulSet is the workload API object used to manage stateful applications. It's best not to use the StatefulSet unless it is absolutely necessary. The very common use case of this component is an implementation of elasticsearch or mongodb (multiple persistent replicas with sticky identity)