Highly Available Jenkins AWS EKS Architecture

Matt Kereczman (LINBIT), with contributions from James Bland (AWS) and Welly Siauw (AWS)
Version 5.1, 2022-03-10

Table of Contents

Chapter 1. Introduction

LINBIT SDS is the product name for LINBIT's LINSTOR software and the various plugins that surround it. In this reference architecture, LINBIT SDS will be referenced as "LINSTOR".

LINSTOR is an open source management tool designed to manage block storage devices for Linux server clusters. Its primary use-case is to provide Linux block storage for Kubernetes and other public and private cloud platforms.

LINSTOR layers various storage software native to Linux in order to provide tailored feature sets for the volumes it creates. LINSTOR uses LVM and/or ZFS for pooling and partitioning physical storage as well as enabling snapshots and caching layers. LINSTOR can layer LUKS for encrypted volumes, and VDO for compression and deduplication. The most important storage software LINSTOR manages is DRBD. Layering DRBD enables block level replication, as well as remote attachment of volumes to hosts without a physical replica of a volume, which are both very useful features in cloud solutions like Amazon's EKS (Elastic Kubernetes Service).

EKS provides AWS users with Kubernetes clusters that are both scalable and highly available. Out of the box, EKS clusters will have a default `storageClass` backed by EBS (Elastic Block Store) for stateful Kubernetes workloads. EBS volumes can only be attached to instances in the same AZ (availability zone).

EKS can also use EFS (Elastic File System) backed storage. EFS volumes can be concurrently accessed across AWS AZs, but concurrent access means locking overhead, and therefore isn't as performant as EBS. Stateful workloads with demanding IO requirements, like some Jenkins deployments, will suffer from poor storage performance on EFS.

The gap between performant storage and storage accessible across AZs is where LINSTOR fits into the EKS ecosystem. LINSTOR can be configured to consume unused EBS volumes attached to your EKS worker nodes. LINSTOR will partition them to size using LVM or ZFS, and replicate them synchronously using DRBD to EBS volumes attached to EKS workers in different AZs.

If there is an AZ outage where your Jenkins pod is currently running, the pod will automatically be rescheduled in a different AZ where there is an identical replica of your LINSTOR volume.

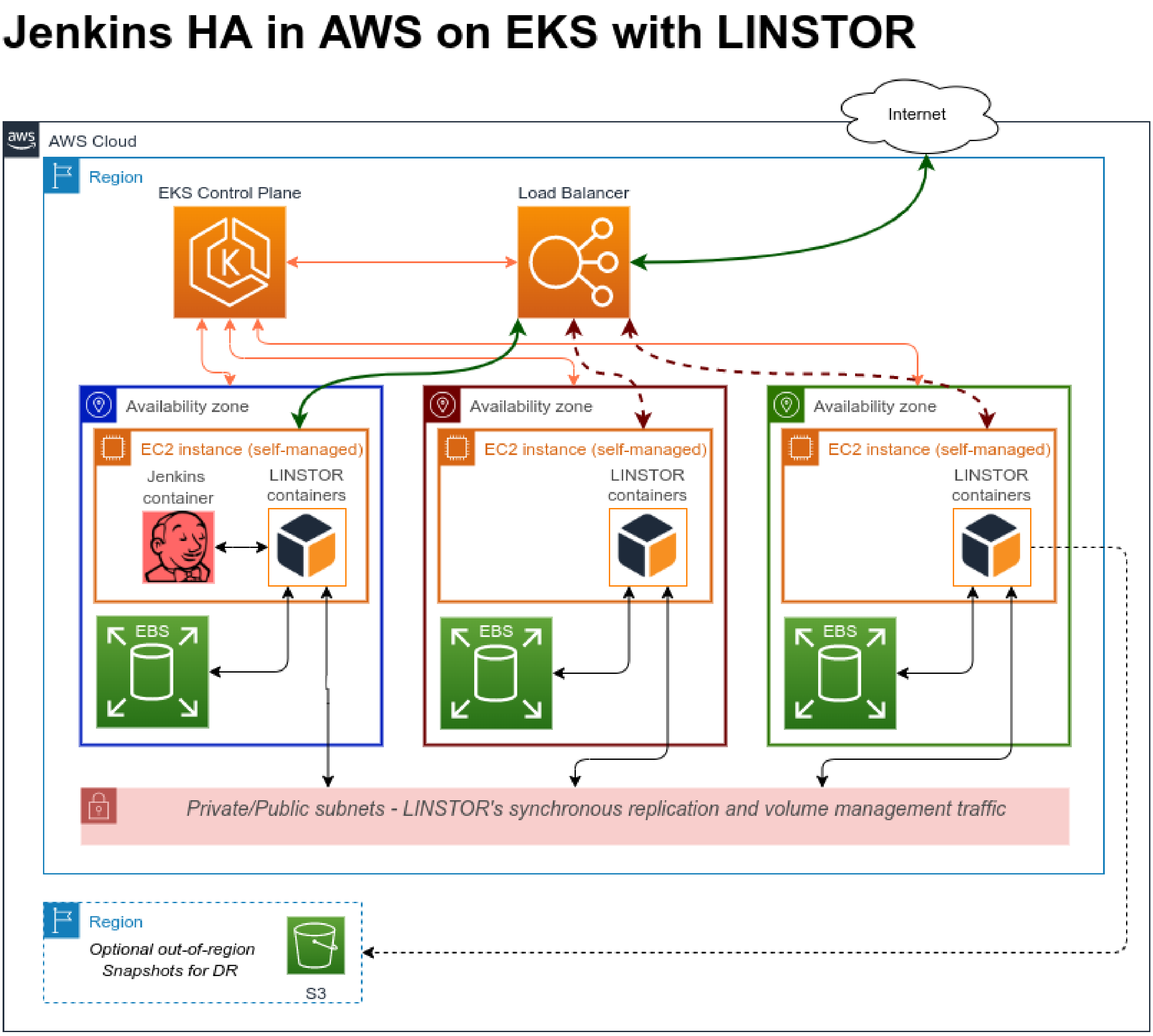
The following reference architecture is specific to deploying HA Jenkins pods onto LINSTOR backed persistent storage in EKS stretching across multiple AZs, but could be adapted for other workloads that need cross AZ replication without sacrificing performance.

The estimated time to deploy the reference architecture outlined in this white paper is 45 minutes.

While LINSTOR is open-source, this reference architecture describes the deployment of LINSTOR from LINBIT's container image repository (http://drbd.io) which is only available to LINBIT customers or through LINBIT customer trial accounts. Contact LINBIT for information on pricing or to begin a trial. Alternatively, you may use LINSTOR SDS' upstream project named Piraeus, without being a LINBIT customer.

1.1. Architectural Diagram

The reference architecture described in this document is depicted in the diagram below.



The diagram shows a singleton Jenkins pod running in an EC2 instance within a single AWS Availability Zone. It writes its persistent data to a LINSTOR volume, which is synchronously replicated to its LINSTOR peer volumes, each in their own different AZ. LINSTOR's control and data planes communicate over private or public subnets attached to each AZ and routed by AWS.

LINSTOR's volumes are backed by EBS volumes in each instance's AZ, which are always identical to each other thanks to LINSTOR's synchronous replication. In the event of an AZ outage, LINSTOR's High Availability controller for StatefulSets will kick in and move the Jenkins workload to another AZ within a few minutes.

The EKS managed load balancer will transparently reroute users to the active AZ whenever the Jenkins pod is migrated.

LINSTOR supports snapshot shipping of volumes to Amazon's S3 to satisfy disaster recovery requirements. The Amazon S3 bucket should be created in a different AWS region than the EKS cluster.

No user data is stored anywhere outside of the EBS volumes used to back LINSTOR's storage-pools and etcd database in this reference architecture.

Chapter 2. Prerequisites

To follow this reference architecture line for line, you're going to need some AWS EKS related tools installed on your workstation, or be familiar enough with AWS services that you can replicate the desired infrastructure without them.

The following tools need to be installed and configured:

- `eksctl`: installation doc
- `kubectl`: installation doc
- `aws cli v2`: installation doc
- `helm 3`: installation doc

It is highly recommended that you do not use your AWS account's root user's access to deploy this reference architecture. Instead, readers should create IAM accounts with least privileges granted. For more information, consult the AWS IAM Security best practices guide.

Chapter 3. Deploying the Solution

The following chapter is broken into sections that will describe how the solution is deployed.

3.1. Create an EC2 Launch Template

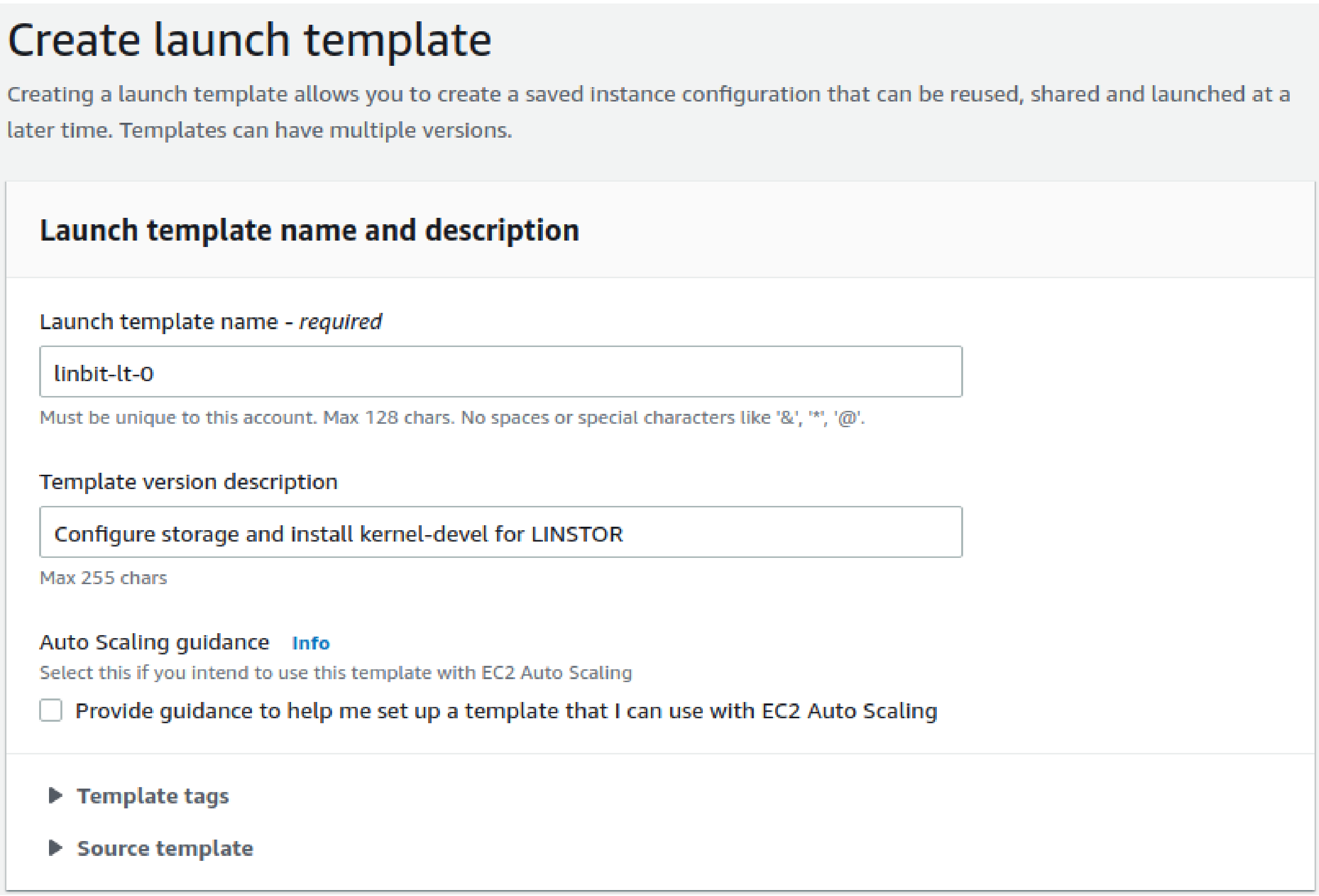
LINSTOR requires an additional unused block device for its storage pools, as well as `kernel-devel` packages present on each EC2 instance for compiling the DRBD kernel module for the Amazon Linux 2.0 kernel. The DRBD kernel module will replicate Jenkins' block device (pvcdata) across AWS AZs.

We can use an EC2 launch template to satisfy these requirements.

Log in to the AWS Management Console and browse to the EC2 Dashboard. In the navigation bar you should see a link to "Launch Templates" nested under the "Instances" drop down; click this link.

Click the "Create launch template" button in the Launch Template console. Only set the options pictured below, as the rest will be configured elsewhere, and duplicated settings will cause failures when launching new instances.

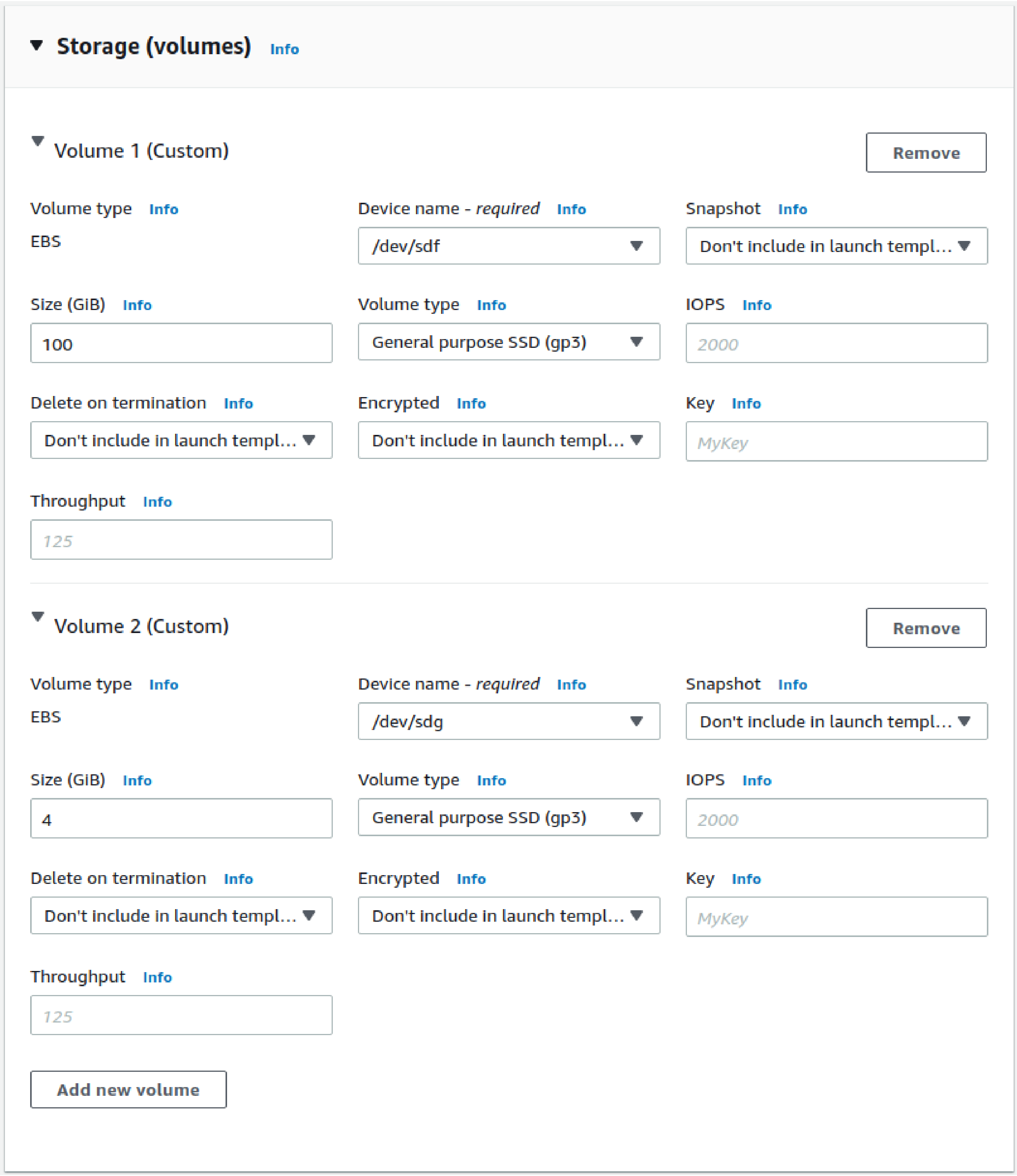
Name and description for launch template



Instance type for launch template

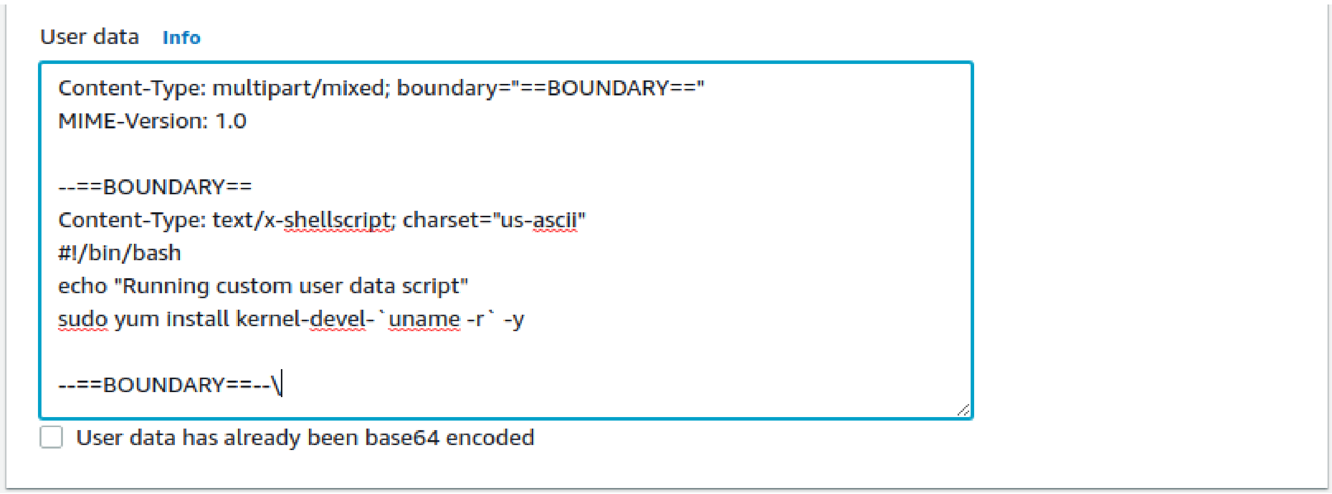
LINSTOR itself is not resource intensive. Memory utilization for a DRBD resource scales with the size of volume and number of replicas. The formula is roughly 32KiB of memory per 1GiB of storage multiplied by the number of peers (*other* nodes with replicas). Size your instances according to your application's requirements.

Storage settings for launch template



The larger volume will be used by LINSTOR when provisioning PVs from its storage classes. The smaller volume will be used as an external metadata pool for LINSTOR's DRBD devices. Set the size of the larger volume according to your deployment's storage requirements. The smaller volume should be sized relative to the size of your larger volume; the ratio of 32MiB (metadata) per 1TiB (data), per peer. 4GiB is the smallest volume allowed by EBS, which will work for data volumes up to 64 TiB in size with 3 replicas of each volume.

Advanced settings for launch template



The only advanced setting that needs modification is the "User data" field. This is where we add the `kernel-devel` package to the EKS bootstrapping. You can copy and paste this text from the code block below.

```
Content-Type: multipart/mixed; boundary="==BOUNDARY=="
MIME-Version: 1.0

--==BOUNDARY==
Content-Type: text/x-shellscript; charset="us-ascii"
#!/bin/bash
echo "Running custom user data script"
sudo yum install kernel-devel-`uname -r` -y

--==BOUNDARY==--\
```

Finally, click the "Create template version" button to save the launch template. You should see that your launch template was created successfully.

Follow the link to view your launch template and note the "Launch Template ID" which should be formatted like this: `lt-0123456789abcdefg`. You will need this when creating your EKS cluster configuration.

3.2. Create the EKS cluster using `eksctl`

Write the cluster's configuration changing the settings where appropriate; specifically: `name`, `region`, `launchTemplate.id`.

```
cat << EOF > eksctl-cluster.yaml
apiVersion: eksctl.io/v1alpha5
kind: ClusterConfig
metadata:
  name: linbit-eks
  region: us-west-2
  version: "1.21"
managedNodeGroups:
- name: lb-mng-0
  launchTemplate:
    id: lt-0123456789abcdefg
    version: "1"
  desiredCapacity: 3
EOF
```

There are some AWS regions that have less than three AZs. LINSTOR's high-availability controller does rely on quorum for decision making, so choosing a region with three or more AZs is a requirement.

Create the cluster using the yaml definition.

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

The cluster creation will take some time. You should see that `eksctl` is configuring resources spread across AZs in your configured region.

3.3. Install LINSTOR via Helm 3

Create a namespace for LINSTOR in your EKS cluster.

```
kubectl create namespace linstor
```

Create a Kubernetes secret using the credentials for your LINBIT portal (customer or trial). If you don't have this, you can reach out to LINBIT to start a trial by filling out the Contact Us form. Alternatively, you can use LINSTOR's upstream – community supported but LINBIT developed – named Piraeus Datastore.

```
kubectl create secret docker-registry drbdiocred -n linstor \
  --docker-server=drbd.io --docker-username=<lb-username> \
  --docker-email=<lb-email> --docker-password=<lb-password>
```

Add the LINSTOR repository to helm.

```
helm repo add linstor https://charts.linstor.io
helm repo update
```

LINSTOR stores its internal database in an etcd key-value store by default. LINSTOR's helm repository has helper charts for creating hostpath volumes for etcd's persistence, but in EKS we can skip creating hostpath volumes and rely on the default EBS `storageClass` provided by EKS to create and attach EBS PVs to our instances for LINSTOR's etcd.

Configure LINSTOR's options in a values file that will be passed to helm during installation.

```
cat << EOF > linstor-op-vals.yaml
operator:
  satelliteSet:
    storagePools:
      lvmThinPools:
      - name: ext-meta-pool
        thinVolume: metapool
        volumeGroup: ""
        devicePaths:
        - /dev/nvme1n1
      - name: lvm-thin
        thinVolume: thinpool
        volumeGroup: ""
        devicePaths:
        - /dev/nvme2n1
    kernelModuleInjectionMode: Compile
stork:
  enabled: false
csi:
  enableTopology: true
etcd:
  replicas: 3
haController:
  replicas: 3
EOF
```

Many things can be configured here. The best source of information for the current and complete list of settings is in the LINSTOR User's Guide as well as the upstream Piraeus Helm Reference Table.

Finally, use helm to install LINSTOR, and watch the LINSTOR stack become ready.

```
helm install -n linstor -f ./linstor-op-vals.yaml linstor-op linstor/linstor
watch kubectl get pods -n linstor -o wide
```

Create storageClass definitions for LINSTOR using the examples below:

```
cat << EOF > linstor-sc.yaml
apiVersion: storage.k8s.io/v1
kind: StorageClass
metadata:
  name: "linstor-csi-lvm-thin-r1"
provisioner: linstor.csi.linbit.com
parameters:
  allowRemoteVolumeAccess: "false"
  autoPlace: "1"
  storagePool: "lvm-thin"
  DrbdOptions/Disk/disk-flushes: "no"
  DrbdOptions/Disk/md-flushes: "no"
  DrbdOptions/Net/max-buffers: "10000"
  property.linstor.csi.linbit.com/StorPoolNameDrbdMeta: "ext-meta-pool"
reclaimPolicy: Retain
allowVolumeExpansion: true
volumeBindingMode: WaitForFirstConsumer
---
apiVersion: storage.k8s.io/v1
kind: StorageClass
metadata:
  name: "linstor-csi-lvm-thin-r2"
provisioner: linstor.csi.linbit.com
parameters:
  allowRemoteVolumeAccess: "false"
  autoPlace: "2"
  storagePool: "lvm-thin"
  DrbdOptions/Disk/disk-flushes: "no"
  DrbdOptions/Disk/md-flushes: "no"
  DrbdOptions/Net/max-buffers: "10000"
  property.linstor.csi.linbit.com/StorPoolNameDrbdMeta: "ext-meta-pool"
reclaimPolicy: Retain
allowVolumeExpansion: true
volumeBindingMode: WaitForFirstConsumer
---
apiVersion: storage.k8s.io/v1
kind: StorageClass
metadata:
  name: "linstor-csi-lvm-thin-r3"
provisioner: linstor.csi.linbit.com
parameters:
  allowRemoteVolumeAccess: "false"
  autoPlace: "3"
  storagePool: "lvm-thin"
  DrbdOptions/Disk/disk-flushes: "no"
  DrbdOptions/Disk/md-flushes: "no"
  DrbdOptions/Net/max-buffers: "10000"
  property.linstor.csi.linbit.com/StorPoolNameDrbdMeta: "ext-meta-pool"
reclaimPolicy: Retain
allowVolumeExpansion: true
volumeBindingMode: WaitForFirstConsumer
EOF

kubectl apply -f ./linstor-sc.yaml
```

> These example storageClass definitions will work for the LINSTOR storage pools we've created. Most tuning of LINSTOR resources happens in the storageClass definitions.

3.4. Install Jenkins via Helm 3

To simplify the deployment of Jenkins, we'll use Helm to deploy Jenkins.

Create a namespace for Jenkins in your EKS cluster.

```
kubectl create namespace jenkins
```

Add the Jenkins repository to helm.

```
helm repo add jenkinsci https://charts.jenkins.io
helm repo update
```

We will need to override some of the default values for our environment in Jenkins' Helm chart. Specifically, we'll configure the following options and values:

- `persistence.storageClass: linstor-csi-lvm-thin-r3`
- `persistence.size: "20Gi"`
- `controller.serviceType: LoadBalancer`
- `controller.podLabels:`
  - `linstor.csi.linbit.com/on-storage-lost: remove`

> Adjust the `persistence.size="20Gi"` value according to your requirements.

Create the Jenkins configuration override file:

```
cat << EOF > jenkins-op-vals.yaml
persistence:
  storageClass: linstor-csi-lvm-thin-r3
  size: "20Gi"
controller:
  serviceType: LoadBalancer
  podLabels:
    linstor.csi.linbit.com/on-storage-lost: remove
EOF
```

Install Jenkins with the override options using helm:

```
helm install -n jenkins -f ./jenkins-op-vals.yaml jenkins jenkinsci/jenkins
```

Once running, you should be able to get the admin password and the DNS address for the load balancer using the steps output by helm after installation. The commands should look something like this:

```
kubectl exec --namespace jenkins -it svc/jenkins -c jenkins \
  -- /bin/cat /run/secrets/chart-admin-password && echo

export SERVICE_IP=$(kubectl get svc --namespace jenkins jenkins \
  --template "{{ range (index .status.loadBalancer.ingress 0) }}{{ . }}{{ end }}")
echo http://$SERVICE_IP:8080/login
```

The DNS address of your load balancer is publicly accessible. Be sure to follow your organizations security practices when hardening your Jenkins deployment (e.g. limiting source IP ranges on the load balancer's security group to those of your organization).

Chapter 4. Validating the Solution

The following chapter contains sections that will describe validation tests and their expected outcomes.

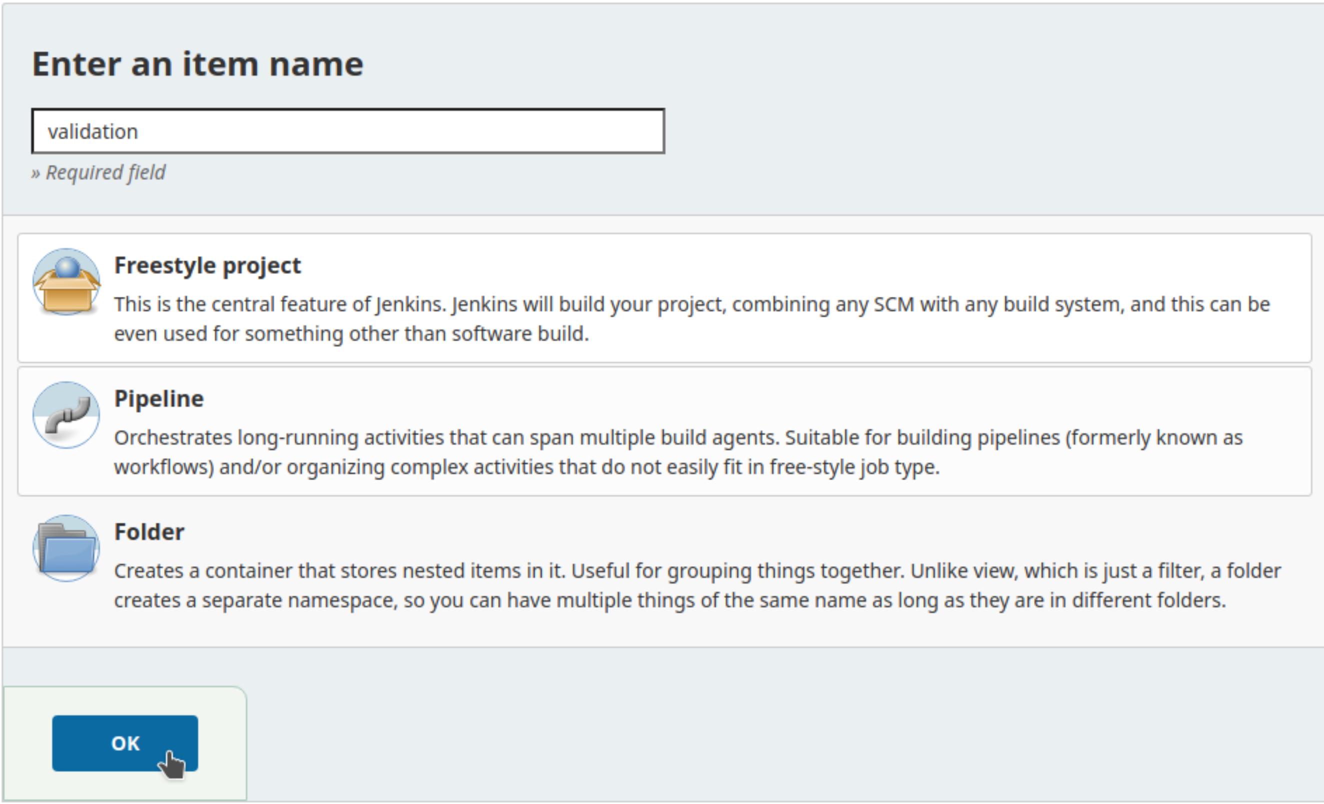
4.1. Failover Verification

The general idea here is to fail/terminate the EC2 instance that is currently running the singleton Jenkins pod. It should be migrated to another AZ without user interaction, with access to the persistent data as the pod in the original AZ.

Log in to the Jenkins instance using the outputs from the helm installation. From there, create a job to validate that your persistent data remains intact after an instance or AZ failure.

Here’s an example job for completeness, but feel free to test however you see fit.

Create a new item as a Freestyle project



Give it some description and add an "Execute shell" build step to the, "Build", section of the project. I'm writing a 10MiB file from `/dev/urandom/` and taking a sha1sum of it in my example.

Add an execute shell build step

Save the project, then click on "Build Now" to start a build. Once complete, view and note the "Console output" from the successful job. Near the bottom you'll see output similar to this:

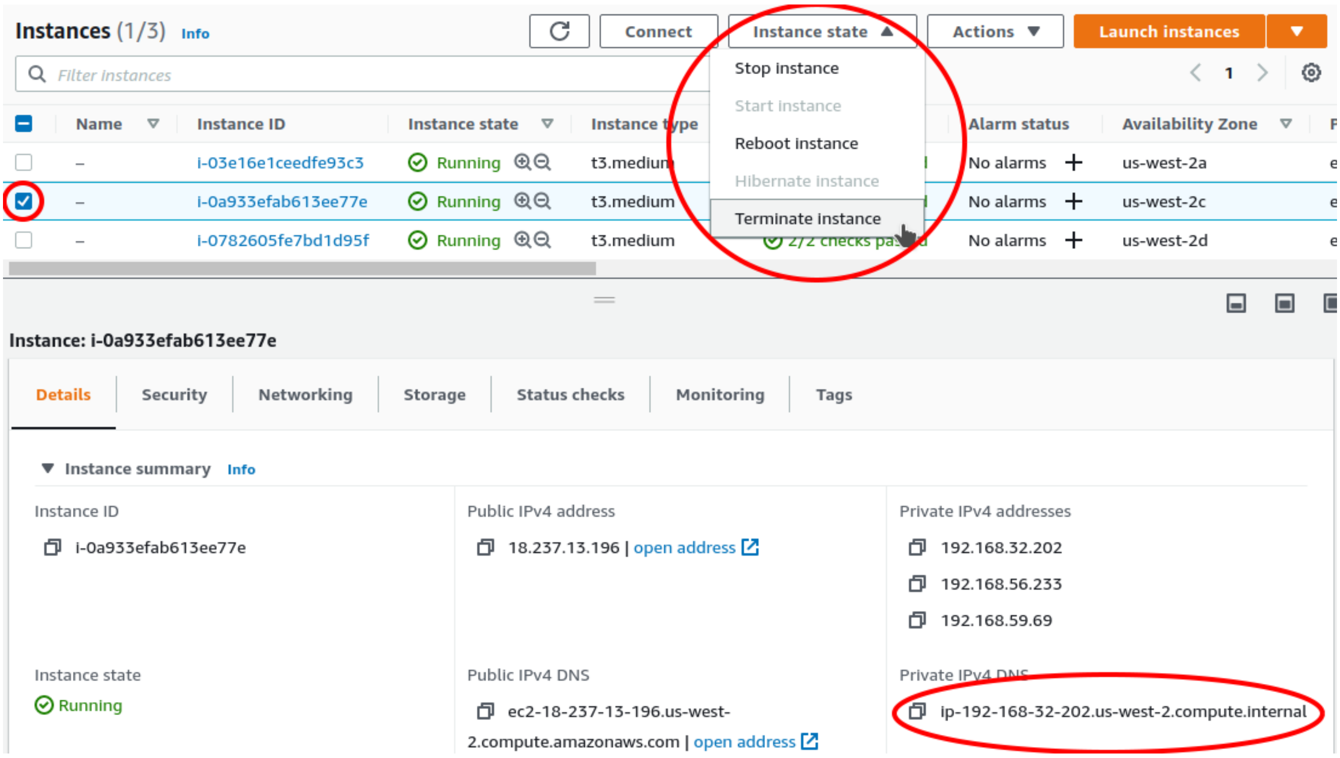
```
...snip...
[validation] $ /bin/sh -xe /tmp/jenkins5820634365303503379.sh
+ dd if=/dev/urandom of=./validation.dump bs=1M count=10
10+0 records in
10+0 records out
10485760 bytes (10 MB, 10 MiB) copied, 0.402937 s, 26.0 MB/s
+ sha1sum ./validation.dump
683df872bb28d55c449914957b635c34f65719d8  ./validation.dump
Finished: SUCCESS
```

The above output is what we want to see once a pod is moved to another AZ. To simulate an instance failure, we can terminate the instance running our singleton Jenkins pod. Retrieve the node name using the following command:

```
kubectl get pods -n jenkins jenkins-0 -o jsonpath='{.spec.nodeName}'
ip-192-168-32-202.us-west-2.compute.internal
```

Browse to your AWS Management Console, find the EC2 instance with the DNS name output by the command above, and terminate the instance.

Terminate EC2 instance running jenkins-0



Use something like the following command to view the status of the pods and nodes in the cluster. You should see the `jenkins-0` pod `Terminating` shortly after the node failure is noticed by the EKS control plane. Shortly after that, the node is being rescheduled on a worker in a different AZ.

```
watch "kubectl get pods -n jenkins -o wide; kubectl get nodes"
```

Once the pod is running again, you'll be able to log back into the Jenkins dashboard and see your persistent data has survived the AZ migration.

4.2. Performance Expectations

Replicating writes synchronously between AZs does have some performance overhead. However, that overhead is minimal due to Amazon's low-latency fiber networks that connect the data centers comprising any of the AWS Regions.

IO performance will differ depending on the type of EBS storage selected when creating the launch template. In this reference architecture, 100GiB **gp3** (general purpose SSD) volumes were used, which have a baseline performance of 3000 IOPS and 125MB/s of throughput at any volume size. IOPS and throughput on **gp3** volumes can be scaled up to 16,000 and 1,000MB/s respectively.

Using `fio`, an open source IO workload generator, performance was measured while simulating various IO patterns and block sizes on LINSTOR volumes. Those same workloads were tested against EBS volumes without LINSTOR as a baseline for comparison in our benchmarks.

Pure read test results are omitted below. This is because LINSTOR provisions thin volumes in this reference architecture, and reads that would be pulled from an unallocated block return immediately, which unrealistically inflates the results.

Table 1. EBS Baseline Results

	4k	16k	32k	64k
randrw	3024	3024	3024	2065
randwrite	3025	3025	3025	2065
readwrite	3025	3025	4132	2065
write	3025	3025	4133	2066

Table 2. LINSTOR Single Replica (no replication) Results

	4k	16k	32k	64k
randrw	3461	3361	3055	2929
randwrite	2389	2253	1890	1990
readwrite	3254	3308	3919	2093
write	3029	3015	3206	2010

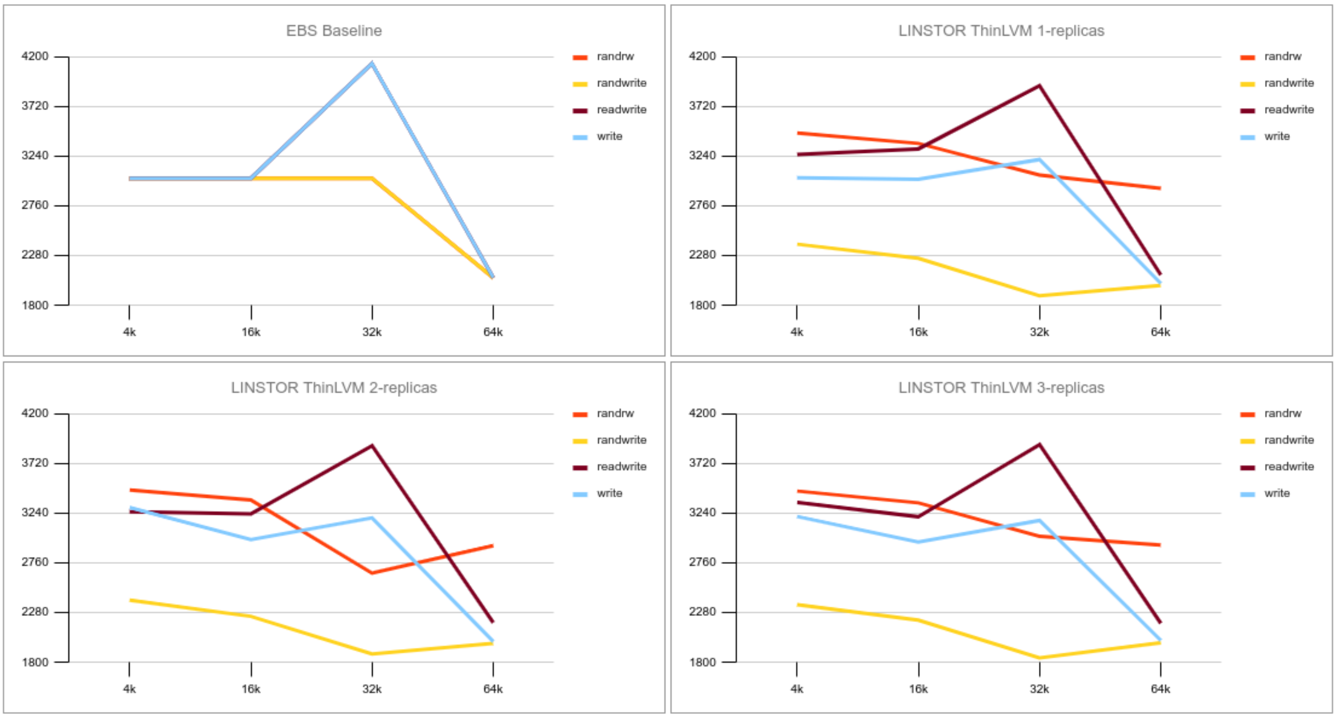
Table 3. LINSTOR Two Replica Results

	4k	16k	32k	64k
randrw	3461	3365	2659	2924
randwrite	2398	2243	1880	1981
readwrite	3252	3232	3890	2182
write	3292	2984	3192	2001

Table 4. LINSTOR Three Replica Results

	4k	16k	32k	64k
randrw	3452	3338	3014	2931
randwrite	2355	2207	1843	1988
readwrite	3343	3204	3901	2176
write	3207	2959	3168	2011

EBS Baseline and LINSTOR Replication Results Graphs



Chapter 5. Maintaining the Solution

The following chapter's sections provide guidance for day-2 operations that will help the reader maintain the solution once deployed.

5.1. Snapshot Shipping to S3

Starting with LINSTOR operator 1.8.0, you can configure LINSTOR VolumeSnapshotClasses in Kubernetes that are backed by Amazon's S3. This enables out-of-region snapshot shipping for LINSTOR volumes, therefore protecting your data from AWS region outages or natural disasters that might span an entire AWS region.

5.1.1. Installing snapshot-controller into EKS

EKS does not provide a snapshot-controller in their Kubernetes distribution. This is required for CSI drivers that support snapshots, therefore one must be installed.

LINSTOR's upstream project, Piraeus, provides Helm charts to assist users with installing the external-snapshotter from the Kubernetes Storage SIG.

Since Helm does not support management of CRDs, we need to manually create these in preparation for Helm deployment of the external-snapshotter.

```
# using this variable to "split" a long URL for PDF readability
EXT_SNAP_URL="https://raw.githubusercontent.com/kubernetes-csi/external-snapshotter"

# create the CRDs
kubectl apply -f \
  ${EXT_SNAP_URL}/v5.0.1/client/config/crd/snapshot.storage.k8s.io_volumesnapshotclasses.yaml
kubectl apply -f \
  ${EXT_SNAP_URL}/v5.0.1/client/config/crd/snapshot.storage.k8s.io_volumesnapshots.yaml
kubectl apply -f \
  ${EXT_SNAP_URL}/v5.0.1/client/config/crd/snapshot.storage.k8s.io_volumesnapshotcontents.yaml
```

Create a namespace for the snapshot-controller.

```
kubectl create namespace snapshot-controller
```

Add the Helm chart repository from the Piraeus project, and deploy the snapshot-controller into the newly created namespace.

```
helm repo add piraeus-charts https://piraeus.io/helm-charts/
helm repo update
helm install -n snapshot-controller snapshot-validation-webhook \
  piraeus-charts/snapshot-validation-webhook
helm install -n snapshot-controller snapshot-controller \
  piraeus-charts/snapshot-controller --wait
```

Once Helm returns control, you should see you have the `snapshot-controller` as well as the necessary `snapshot-validation-webhook` pods running in your `snapshot-controller` namespace.

```
kubectl get pods -n snapshot-controller
NAME                                         READY   STATUS      RESTARTS   AGE
snapshot-controller-685b89b4fc-trrzz         1/1     Running     0          39m
snapshot-validation-webhook-9866866dc-794rr  1/1     Running     0          41m
snapshot-validation-webhook-test-invalid-body 0/1    Completed   0          40m
snapshot-validation-webhook-test-valid-body  0/1     Completed   0          40m
```

5.1.2. Create an S3 Bucket

Create an S3 bucket for LINSTOR's Jenkins snapshots. When you create the bucket, it's recommended to use the default settings "ACL disabled" and "Block *all* public access".

You will need to know the name and region of the S3 bucket when configuring your `VolumeSnapshotClass`.

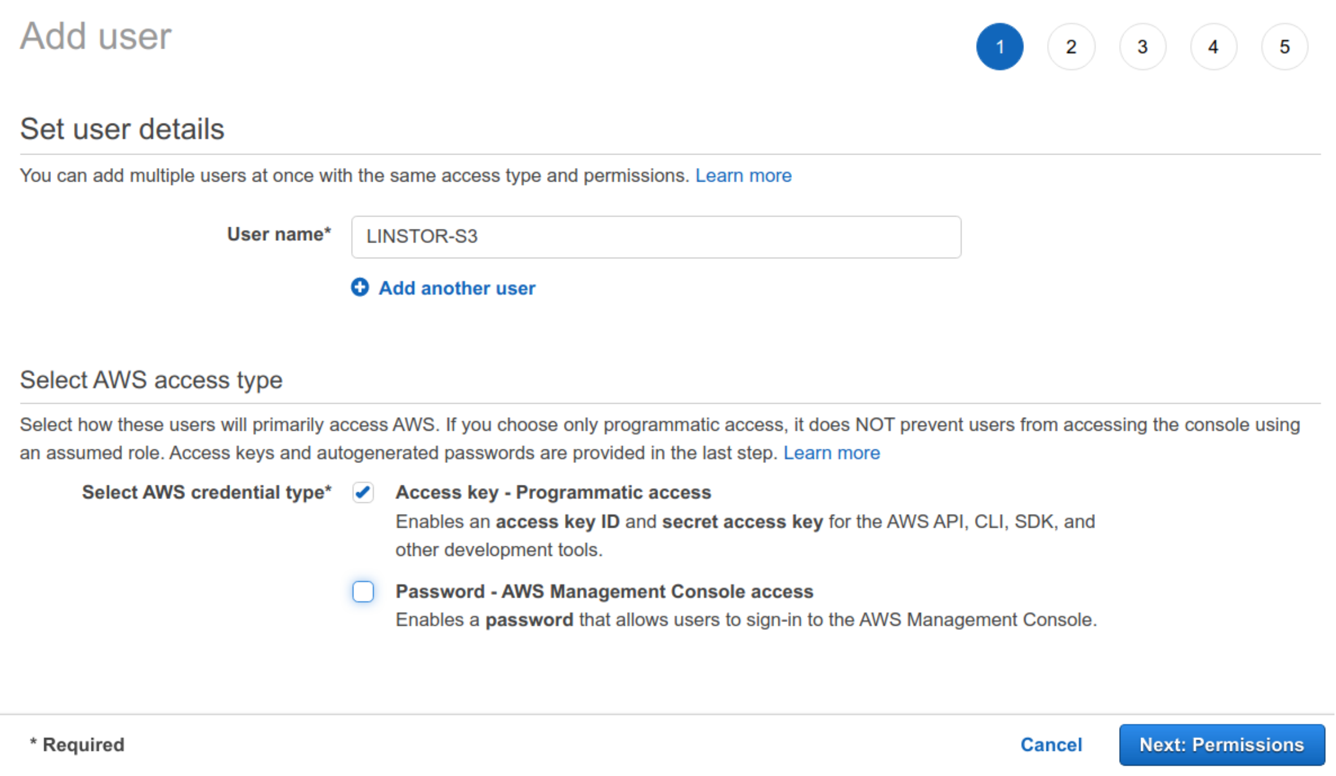
5.1.3. Create IAM User with S3 Access

LINSTOR requires programmatic access to an AWS user with sufficient privileges for reading and writing to S3. The access key and secret access key must be stored in a Kubernetes secret that LINSTOR can access. It's best practice to create an IAM user with least privilege for this type of access.

Browse to IAM in your AWS console, then select "Users" under "Access management". Click the "Add users" button to begin creation.

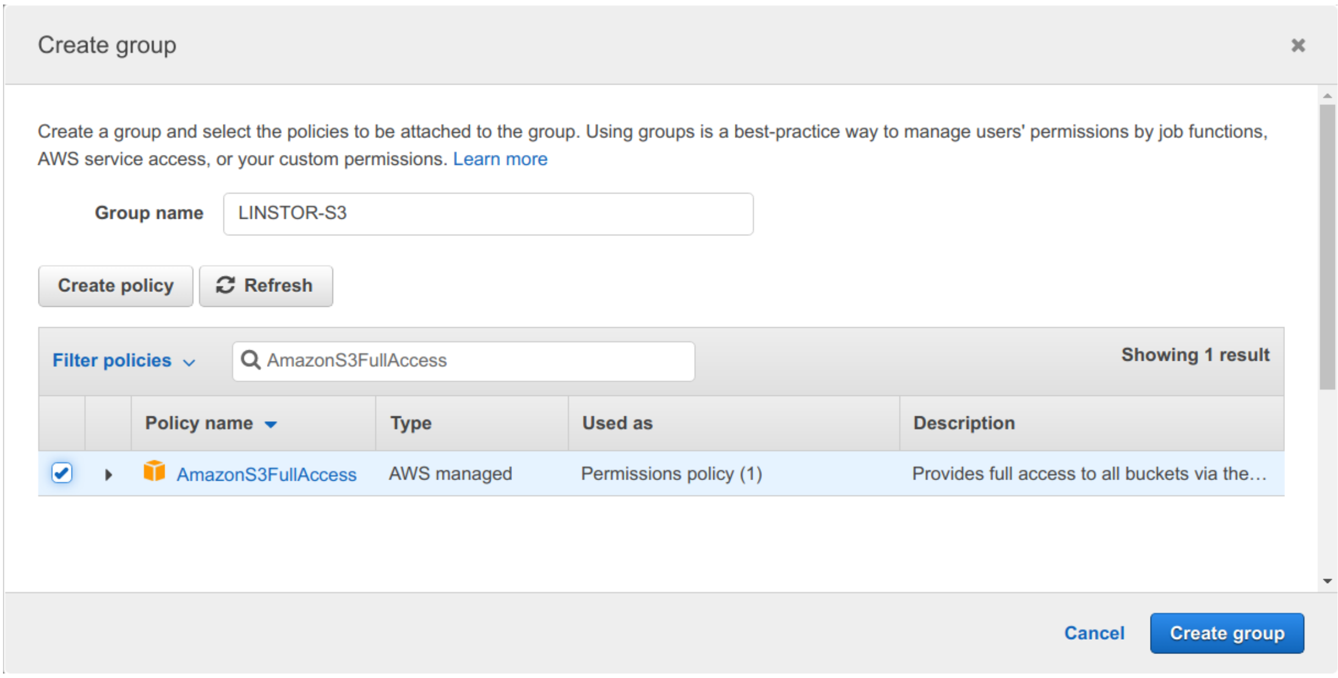
Name the user something obvious, and select the option for "Access key - Programmatic access", click next to proceed.

Create New AWS User for LINSTOR



Create a new group and apply only the "AmazonS3FullAccess" policy.

Create New Group Limited to S3 Access



You can add the user to an existing group if one already exists with the appropriate policies under your account.

Optionally, you can add tags to the user. Finally, review and create the new user making sure to securely store the new users access key and secret access keys.

5.1.4. Add IAM User as Kubernetes Secret

Using the access key and secret access key for the IAM account created for LINSTOR's S3 access, create the Kubernetes secret for LINSTOR.

```
kubectl create secret generic linstor-csi-s3-access -n linstor \
  --type=linstor.csi.linbit.com/s3-credentials.v1 \
  --from-literal=access-key=REDACTED \
  --from-literal=secret-key=REDACTED
kubectl patch secrets linstor-csi-s3-access -n linstor \
  --type=merge -p '{"immutable":true}'
```

5.1.5. Add LINSTOR Volume Snapshot Class

Add a `VolumeSnapshotClass` to Kubernetes using information specific to your S3 bucket.

```
cat << EOF > linstor-s3-snapclass.yaml
kind: VolumeSnapshotClass
apiVersion: snapshot.storage.k8s.io/v1
metadata:
  name: linstor-csi-snapshot-class-s3
driver: linstor.csi.linbit.com
deletionPolicy: Delete
parameters:
  snap.linstor.csi.linbit.com/type: S3
  snap.linstor.csi.linbit.com/remote-name: s3-us-west-2
  snap.linstor.csi.linbit.com/allow-incremental: "false"
  snap.linstor.csi.linbit.com/s3-bucket: name-of-bucket-123
  snap.linstor.csi.linbit.com/s3-endpoint: http://s3.us-west-2.amazonaws.com
  snap.linstor.csi.linbit.com/s3-signing-region: us-west-2
  snap.linstor.csi.linbit.com/s3-use-path-style: "false"
  # Secret to store access credentials
  csi.storage.k8s.io/snapshotter-secret-name: linstor-csi-s3-access
  csi.storage.k8s.io/snapshotter-secret-namespace: linstor
EOF

kubectl apply -f linstor-s3-snapclass.yaml
```

Service Endpoints in AWS generally follow the convention of, **`protocol://service-code.region-code`.amazonaws.com**. S3 utilizes HTTP(S). Therefore, our bucket in the us-west-2 region from the example above uses the s3-endpoint: http://s3.us-west-2.amazonaws.com

5.1.6. Create a Volume Snapshot of Jenkins PVC

Finally, you can create snapshots of your Jenkins volume using Kubernetes native patterns. Jenkins Helm charts deploy Jenkins onto a PVC named "jenkins". Create an S3 snapshot of this PVC:

```
cat << EOF > jenkins-s3-snap.yaml
apiVersion: snapshot.storage.k8s.io/v1
kind: VolumeSnapshot
metadata:
  name: jenkins-dr-snapshot-0
  namespace: jenkins
spec:
  volumeSnapshotClassName: linstor-csi-snapshot-class-s3
  source:
    persistentVolumeClaimName: jenkins
EOF

kubectl apply -f jenkins-s3-snap.yaml
```

After a short while, depending on the size of your volume, you should see that the snapshot has become "Ready", at which point you should be able to see the snapshot in your S3 bucket via the AWS S3 management console.

5.1.7. Restore a Volume Snapshot from S3

When restoring a volume from a snapshot stored in S3 there are two possible scenarios you will find yourself in.

You may be restoring Jenkins' PVC into a foreign cluster, which is any cluster other than Jenkins' origin cluster. This would be the case if you've redeployed your EKS cluster into a new region, or simultaneously lost all instances comprising the original cluster.

Alternatively, you may be restoring Jenkins' PVC into the origin cluster as a way to roll Jenkins' state back to a previous point in time.

Restoring Snapshots into Foreign Cluster

If you've redeployed the solution into a new cluster, there are steps that must be performed "manually" within the LINSTOR controller pod before restoring will be possible.

We must create a `remote` object within LINSTOR that references the S3 bucket containing our snapshots. To do this, use the commands below making the appropriate substitutions for your S3 bucket.

```
# set the LINSTOR controller pod name to a variable
linstor_k8s=$(kubectl get pods -n linstor \
  -l app.kubernetes.io/instance=linstor-op-cs \
  -o custom-columns=":metadata.name" --no-headers)
# create the LINSTOR remote object referencing your S3 bucket
kubectl exec -it -n linstor $linstor_k8s -- linstor \
  remote create s3 s3-origin \
  http://s3.<AWS_REGION>.amazonaws.com \
  <S3_BUCKET_NAME> <AWS_REGION> \
  <ACCESS_KEY> <SECRET_KEY>
```

Recreate your LINSTOR snapshot-class and LINSTOR S3 secret referencing your IAM user as described in the respective sub-section of this section of this guide.

List the snapshots available in your S3 remote:

```
kubectl exec -it -n linstor $linstor_k8s -- linstor \
  backup list s3-us-west-2-origin
```

We will need to reference the snapshot name using the portion following the ^. You can use regex and grep to filter for the portion that is needed using the command below.

```
kubectl exec -it -n linstor $linstor_k8s -- linstor \
  backup list s3-us-west-2-origin | grep -o snapshot[a-Z0-9\-]*
```

Use the truncated name to populate the `VolumeSnapshotContent` below, and apply the yaml to import the snapshot from S3 into Kubernetes.

```
cat << EOF > snap-from-s3.yaml
apiVersion: snapshot.storage.k8s.io/v1
kind: VolumeSnapshotContent
metadata:
  name: restored-snapshot
  namespace: jenkins
spec:
  deletionPolicy: Delete
  driver: linstor.csi.linbit.com
  source:
    snapshotHandle: <PLACE_BACKUP_NAME_HERE>
  volumeSnapshotClassName: linstor-csi-snapshot-class-s3
  volumeSnapshotRef:
    apiVersion: snapshot.storage.k8s.io/v1
    kind: VolumeSnapshot
    name: restored-snapshot
    namespace: jenkins
---
apiVersion: snapshot.storage.k8s.io/v1
kind: VolumeSnapshot
metadata:
  name: restored-snapshot
  namespace: jenkins
spec:
  source:
    volumeSnapshotContentName: restored-snapshot
  volumeSnapshotClassName: linstor-csi-snapshot-class-s3
EOF

kubectl apply -f snap-from-s3.yaml
```

You should then see that you have a `VolumeSnapshot` within Kubernetes which you can use to populate a new PVC using standard Kubernetes patterns.

```
cat << EOF > restore-snap-to-pvc.yaml
apiVersion: v1
kind: PersistentVolumeClaim
metadata:
  name: jenkins-restored
  namespace: jenkins
spec:
  storageClassName: linstor-csi-lvm-thin-r3
  accessModes:
  - ReadWriteOnce
  dataSource:
    name: restored-snapshot
    kind: VolumeSnapshot
    apiGroup: snapshot.storage.k8s.io
  resources:
    requests:
      storage: 20G
EOF

kubectl apply -f restore-snap-to-pvc.yaml
```

With the snapshot restored into a PVC named, `jenkins-restored`, you can deploy Jenkins with the restored PVC by setting the `persistence.existingClaim` key in the Jenkins chart values.

```
cat << EOF > jenkins-op-vals.yaml
persistence:
  storageClass: linstor-csi-lvm-thin-r3
  existingClaim: jenkins-restored
  size: "20Gi"
controller:
  serviceType: LoadBalancer
  podLabels:
    linstor.csi.linbit.com/on-storage-lost: remove
EOF

helm install -n jenkins -f ./jenkins-op-vals.yaml jenkins jenkinsci/jenkins
```

Restoring Snapshots into Origin Cluster

Restoring snapshots from a remote into the origin cluster is far simpler as the `VolumeSnapshot` and `VolumeSnapshotContent` objects are already present. Therefore, you may use the typical Kubernetes patterns for restoring a snapshot to a PVC.

Identify the snapshot you'd like to restore from.

```
kubectl get -n jenkins volumesnapshot
```

Use the name of the snapshot above to populate the yaml below.

```
cat << EOF > restore-snap-to-pvc.yaml
apiVersion: v1
kind: PersistentVolumeClaim
metadata:
  name: jenkins-restored
  namespace: jenkins
spec:
  storageClassName: linstor-csi-lvm-thin-r3
  accessModes:
  - ReadWriteOnce
  dataSource:
    name: jenkins-dr-snapshot-0
    kind: VolumeSnapshot
    apiGroup: snapshot.storage.k8s.io
  resources:
    requests:
      storage: 20G
EOF

kubectl apply -f restore-snap-to-pvc.yaml
```

With the snapshot restored into a PVC named, `jenkins-restored`, you can deploy Jenkins with the restored PVC by setting the `persistence.existingClaim` key in the Jenkins chart values.

```
cat << EOF > jenkins-op-vals.yaml
persistence:
  storageClass: linstor-csi-lvm-thin-r3
  existingClaim: jenkins-restored
  size: "20Gi"
controller:
  serviceType: LoadBalancer
  podLabels:
    linstor.csi.linbit.com/on-storage-lost: remove
EOF

helm install -n jenkins -f ./jenkins-op-vals.yaml jenkins jenkinsci/jenkins
```

If the original Jenkins deployment is still installed, you can use the `helm upgrade` to move the deployment onto the restored PVC.

```
helm upgrade -n jenkins jenkins jenkinsci/jenkins --reuse-values \
  --set persistence.existingClaim=jenkins-restored
```

5.2. Health Monitoring with Prometheus Operator

Starting with LINSTOR operator v1.5.0, you can use Prometheus to monitor LINSTOR components. The operator will set up monitoring containers alongside the existing components and make them available as a `Service` in Kubernetes.

If you use the Prometheus Operator, the LINSTOR Operator will also set up the `ServiceMonitor` instances. The metrics will automatically be collected by the Prometheus instance associated to the operator, assuming you configure the Prometheus Operators access to the LINSTOR namespace.

Detailed steps for building and customizing the Prometheus Operator can be found in the Project's README.md on Github. The steps below will deploy the Prometheus Operator and Grafana dashboards specifically for this reference architecture.

5.2.1. Building Prometheus Operator

You need go installed and in your path in order to install the build dependencies. Once you have go, prepare your build environment and install build dependencies using go:

```
mkdir my-kube-prometheus; cd my-kube-prometheus
go get github.com/jsonnet-bundler/jsonnet-bundler/cmd/jb
go get github.com/brancz/gojsontoyaml
go get github.com/google/go-jsonnet/cmd/jsonnet
jb init
jb install github.com/prometheus-operator/kube-prometheus/jsonnet/kube-prometheus@release-0.8
jb update
```

`release-0.8` of the Prometheus Operator or better is required as it deploys Grafana 7.5.4 which is a requirement of LINBIT's Grafana Dashboard.

Create a customized `.jsonnet` configuration, download the build script and build the manifests:

```
cat << EOF > linstor-jenkins-ns.jsonnet
local kp = (import 'kube-prometheus/main.libsonnet') + {
  values+:: {
    common+: {
      platform: 'eks',
      namespace: 'monitoring',
    },
    prometheus+:: {
      namespaces: ["default", "kube-system", "linstor", "jenkins"],
    },
    kubePrometheus+: {
      platform: 'eks',
    },
  },
};

{ ['00namespace-' + name]: kp.kubePrometheus[name] for name in std.objectFields(kp.kubePrometheus) } +
{ ['0prometheus-operator-' + name]: kp.prometheusOperator[name] for name in std.objectFields(kp.prometheusOperator) } +
{ ['node-exporter-' + name]: kp.nodeExporter[name] for name in std.objectFields(kp.nodeExporter) } +
{ ['kube-state-metrics-' + name]: kp.kubeStateMetrics[name] for name in std.objectFields(kp.kubeStateMetrics) } +
{ ['alertmanager-' + name]: kp.alertmanager[name] for name in std.objectFields(kp.alertmanager) } +
{ ['prometheus-' + name]: kp.prometheus[name] for name in std.objectFields(kp.prometheus) } +
{ ['grafana-' + name]: kp.grafana[name] for name in std.objectFields(kp.grafana) }
EOF

wget https://raw.githubusercontent.com/prometheus-operator/kube-prometheus/release-0.8/build.sh -O build.sh
chmod +x ./build.sh
./build.sh linstor-jenkins-ns.jsonnet
```

You should now have a `manafests` directory full of the yaml manifests needed to deploy the Prometheus Operator to monitor LINSTOR. Deploy the Prometheus Operator into EKS:

```
kubectl apply -f manifests
```

If any of the manifests fail to apply, simply run the command above again. Some resources need to start before others can be applied.

After deploying, forward ports to Prometheus and Grafana in order to test and configure:

```
kubectl --namespace monitoring port-forward svc/prometheus-k8s 9090 &
kubectl --namespace monitoring port-forward svc/grafana 3000 &
```

Point your web browser at http://localhost:3000, and set a secure password for Grafana. Add a dashboard by importing LINBIT's Grafana dashboard for DRBD.

If everything was done correctly, you should see a dashboard similar to the one depicted below:

This is a good starting point for configuring alerts. In a healthy cluster, you would never see a DRBD device become *out-of-sync*, a DRBD device *Without Quorum*, or a DRBD device in the *Standalone* state.

If you want to access Grafana's dashboard without the port forward, expose the deployment to create a LoadBalancer service and retrieve the public DNS name of the LoadBalancer:

```
kubectl expose deployment -n monitoring grafana --type=LoadBalancer --name=grafana-lb

export GRAF_IP=$(kubectl get svc -n monitoring grafana-lb --template \
  "{{ range (index .status.loadBalancer.ingress 0) }}{{ . }}{{ end }}")
echo http://$GRAF_IP:3000
```

Exposing Grafana makes it publicly accessible unless you have tightened up the VPC security group.

5.3. Backup and Recovery of LINSTOR's etcd Key-value Store

LINSTOR stores its database in an etcd cluster deployed by the LINSTOR operator. This is the only stateful component in the LINSTOR control plane and should be backed up regularly.

5.3.1. Creating an etcd Backup

To create a backup of the etcd database and copy it to your control host, run:

```
kubectl exec -n linstor linstor-op-etcd-0 -- etcdctl snapshot save /tmp/backup.db
kubectl cp -n linstor linstor-op-etcd-0:/tmp/backup.db backup.db
```

The above commands will copy the `backup.db` to the current working directory on your local machine.

5.3.2. Restoring an etcd Backup

To restore LINSTOR's etcd database from a backup, you must follow the procedure outlined below, waiting for the previous step to complete before continuing onto the next:

- Touch a `nostart` file in each etcd pod to prevent etcd startup
- Restart the etcd pods with etcd stopped
- Delete the existing cluster's data directory in each etcd pod
- Copy the `backup.db` to each etcd pod
- Restore the backup in each etcd pod
- Delete the `nostart` file in each etcd pod
- Restart the etcd pods with etcd no longer stopped
- Restart the LINSTOR controller pod

Touch the `nostart` file in each etcd pod and restart each etcd pod:

```
# Touch the nostart file
for p in $(kubectl get pods -n linstor -o=jsonpath="{.items[*].metadata.name}" -l app=etcd); do
  kubectl exec -n linstor $p -- touch /var/run/etcd/nostart
done

# Restart etcd pods
for p in $(kubectl get pods -n linstor -o=jsonpath="{.items[*].metadata.name}" -l app=etcd); do
  kubectl delete -n linstor pod $p
done
```

Once the etcd pods have been rescheduled and report running, delete the old etcd cluster's data directory in each etcd pod, and copy the backed up database file to each etcd pod. Then, restore the backup in each etcd pod:

```
# Delete the existing etcd cluster
for p in $(kubectl get pods -n linstor -o=jsonpath="{.items[*].metadata.name}" -l app=etcd); do
  kubectl exec -n linstor $p -- rm -rf /var/run/etcd/default.etcd
done

# Copy backup.db etcd to pods
for p in $(kubectl get pods -n linstor -o=jsonpath="{.items[*].metadata.name}" -l app=etcd); do
  kubectl cp -n linstor backup.db $p:/tmp/restore.db
done

# Restore backup into etcd
for p in $(kubectl get pods -n linstor -o=jsonpath="{.items[*].metadata.name}" -l app=etcd); do
  kubectl exec -n linstor $p -- etcdctl snapshot \
    --wal-dir /var/run/etcd/default.etcd/member/wal \
    --data-dir /var/run/etcd/default.etcd \
    restore /tmp/restore.db
done
```

Finally, delete the `nostart` file from each etcd pod and restart the etcd pods. This will start the etcd cluster with the backup restored:

```
# Delete nostart file
for p in $(kubectl get pods -n linstor -o=jsonpath="{.items[*].metadata.name}" -l app=etcd); do
  kubectl exec -n linstor $p -- rm /var/run/etcd/nostart
done

# Restart etcd pods
for p in $(kubectl get pods -n linstor -o=jsonpath="{.items[*].metadata.name}" -l app=etcd); do
  kubectl delete -n linstor pod $p
done
```

Once the etcd pods are rescheduled and running, restart the LINSTOR controller pod to refresh its in memory database from the restored etcd cluster:

```
# Restart the LINSTOR controller
kubectl delete -n linstor pod \
  $(kubectl get pods -n linstor -o=jsonpath="{.items[*].metadata.name}" -l
app.kubernetes.io/instance=linstor-op-cs)
```

5.4. Routine Maintenance

The following tasks should be performed on a regular basis following your organizations policies for such things, or to address critical events such as bug fixes or security breaches.

5.4.1. Upgrading LINSTOR

A LINSTOR Deployment on Kubernetes can be upgraded to a new release using Helm.

Before upgrading to a new release, you should ensure you have an up-to-date backup of the LINSTOR database as described in the Backup and Recovery section of this reference architecture.

Upgrades will update to new versions of the following components:

- LINSTOR Operator Deployment
- LINSTOR Controller
- LINSTOR Satellite
- LINSTOR CSI Driver
- LINSTOR HA Controller
- etcd

Some versions require special steps, please take a look at LINBIT's LINSTOR User Guide's section on release specific upgrades before proceeding.

If there are no special steps for the current upgrade path, the commands to upgrade to a new LINSTOR operator version is as follows:

```
helm repo update
helm -n linstor upgrade --reuse-values linstor-op linstor/linstor
```

If you need to make any new customizations during the upgrade you can retrieve and edit your set values from the current deployment.

```
# Retrieve the currently set options
helm -n linstor get values linstor-op
USER-SUPPLIED VALUES:
USER-SUPPLIED VALUES: null
etcd:
  replicas: 3
haController:
  replicas: 3
operator:
  satelliteSet:
    kernelModuleInjectionMode: Compile
    storagePools:
      lvmThinPools:
      - devicePaths:
        - /dev/nvme1n1
        name: ext-meta-pool
        thinVolume: metapool
        volumeGroup: ""
      - devicePaths:
        - /dev/nvme2n1
        name: lvm-thin
        thinVolume: thinpool
        volumeGroup: ""
stork:
  enabled: false

# Save current options
helm -n linstor get values linstor-op > linstor-op-vals-orig.yaml

# Modify values as needed.
vim linstor-op-vals-orig.yaml

# Start the upgrade
helm -n linstor upgrade linstor-op linstor/linstor -f linstor-op-vals-orig.yaml
```

This triggers the rollout of new pods. After a short wait, all pods should be running and ready. Check that no errors are listed in the status section of `LinstorControllers`, `LinstorSatelliteSets`, and `LinstorCSIDrivers`.

> During the upgrade process, provisioning of volumes and attach/detach operations might not work. Existing volumes and volumes already in use by a pod will continue to work without interruption.

5.4.2. Rotating drbd.io Secret

LINBIT will not force a password reset on the drbd.io account used to pull LINSTOR's container images. However, it's a best practice to rotate passwords on a regular basis. Additionally, LINBIT may reset your password upon request. Regardless of the case, you'll need to update the respective Kubernetes secret.

The best way to do so is using the following approach:

```
kubectl create secret docker-registry drbdiocred -n linstor \
  --docker-server=drbd.io --docker-username=<lb-username> \
  --docker-email=<lb-email> --docker-password=<lb-password> \\
  --dry-run=client -o yaml | kubectl apply -f -
```

Following this approach will allow `kubectl` to record its annotation for tracking changes to the resource in its spec.

Chapter 6. Support for the Solution

To receive support for the solution if you haven't already contact LINBIT's sales team via the Contact Us page on LINBIT's website. Someone will be in contact with you during US or EU business hours depending on which country you're located in.

If you already have support through LINBIT you may open a ticket by emailing support@linbit.com. You will receive a confirmation email stating your ticket has been created, and within which SLAs you should expect a response from LINBIT's engineering team.

Generally speaking, LINBIT SDS (LINSTOR) is only sold with LINBIT's Enterprise support tier. Our Enterprise support tier includes:

- LINSTOR & DRBD update support (certified binary repository and container registry)
- Unlimited incidents covered
- 24/7 email and phone service desk availability
- 1 hour maximum emergency initial response time [1]
- 4 hours maximum initial response time
- 30 day installation support
- Remote installation/troubleshooting and performance tuning via SSH or VPN [2]
- Optional - Remote installation by WebEx, Zoom, etc. [3]
- Access to all necessary software components from LINBIT managed repositories/registries

[1] "Emergency" is defined as an incident involving a production system down or unresponsive, with no workaround available.
[2] Login, troubleshooting, and performance tuning on customer systems all require remote maintenance infrastructure on the customer's end, and the customer's prior consent for every incident that requires remote system access.
[3] Public SSH keys available. WebEx, TeamView, etc. can not be used as an exploratory training session.

Appendix A: Additional Information and Resources

- LINBIT's GitHub Organization: https://github.com/LINBIT/
- Join LINBIT's Community on Slack: https://www.linbit.com/join-the-linbit-drbd-linstor-slack/
- The DRBD® and LINSTOR® User's Guide: https://docs.linbit.com/
- The DRBD® and LINSTOR® Mailing Lists: https://lists.linbit.com/
    - drbd-announce: Announcements of new releases and critical bugs found
    - drbd-user: General discussion and community support
    - drbd-dev: Coordination of development

Appendix B: Legalese

B.1. Trademark Notice

B.2. License Information