

Part One: Setting Up the Host

Update the *apt* repository and install *Git*.

Initial update of apt

```
apt update
```

Create a new User

Originally logged in as root create a user. We'll use the user david for example:

```
adduser david
```

The command will prompt the user for their full name, phone numbers, etc and a password

Additionally, the new user will require sudo privileges (root privileges).

```
usermod -aG sudo david
```

Now let's su into that user continue our installations.

```
su - david
```

```
sudo apt update  
sudo apt install git -y
```

Setup SSH login

To setup up ssh login

Get Repository With DevOps Scripts

(these commands are executed in the 'david' user account)

```
# Get on with Git Work
git clone git@github.com:davidsells/setup_llm.git
```

Private Git Repository Access (Optional)

If you'd like to maintain this as a private repository the following steps are required to setup access to git.

1. create ssh key on machine requiring access to Git

```
# execute the following. You may enter <cr> for all questions
ssh-keygen -t rsa -b 4096 -C "david@persistentdesigns.com"

eval "$(ssh-agent -s)"

ssh-add ~/.ssh/id_rsa
```

1. copy the contents of ~/.ssh/id_rsa
2. Go to your github account
 1. go to settings
 2. open SSH and GPG Keys
 3. Press the "New SSH Key" button and follow instructions where you will paste in the copied key
3. Validate the connection

```
ssh -T git@github.com
```

Response:

```
Hi davidsells! You've successfully authenticated, but GitHub does not provide
```

Setting up LXD on Ubuntu

In this section, we'll cover the steps to set up LXD on a fresh Ubuntu cloud instance. LXD is a containerization platform that provides a secure and scalable environment with minimal

overhead.

Why LXD?

LXD provides a layer of security and separation of devops concerns, which is essential for our Kubernetes deployment. It allows us to manage our workloads with ease and configure them to suit our use case via a user-friendly web interface.

Installing LXD

To install LXD, we'll use the `snap` package manager, which is the recommended way to install LXD on Ubuntu.

```
sudo snap install lxd
```

This command will download and install the LXD snap package.

Add User 'david' to the LXD group

```
chmod -a -G lxd david
```

Logout and log back in to apply the new group to your session.

Initializing LXD

Once LXD is installed, we need to initialize it using the following command:

```
lxd init
```

This command will prompt us to configure LXD. We'll walk through the options and explain what each one does:

LXD Configuration Options

During the initialization process, we'll be asked a series of questions to configure LXD. Here's what each option does:

- **LXD clustering:** This option enables LXD clustering, which allows multiple LXD servers to work together as a single cluster. For our purposes, we can leave this set to the default

value of `no`.

- **Storage pool:** We'll configure a new storage pool, which is where LXD will store its data. We'll name this pool `kube_zfs`.
- **Storage backend:** We'll use the ZFS storage backend, which provides a scalable and reliable storage solution.
- **Create a new ZFS pool:** We'll create a new ZFS pool, which will be used to store our data.
- **Use an existing empty block device:** We won't use an existing block device, but instead create a new one.
- **Size of the new loop device:** We'll allocate 100GiB of storage space from the current drive using a loop device. Note that you may want to adjust this value depending on your specific use case.
- **MAAS server:** We won't connect to a MAAS (Metal as a Service) server, so we'll leave this set to `no`.
- **Create a new local network bridge:** We'll create a new local network bridge, which will allow our LXD containers to communicate with each other.
- **Bridge name:** We'll name our bridge `lxdbr0`, which is the default value.
- **IPv4 and IPv6 addresses:** We'll use automatic IP address assignment for both IPv4 and IPv6.
- **Make LXD server available over the network:** We won't make the LXD server available over the network, so we'll leave this set to `no`.
- **Update stale cached images:** We'll enable automatic updates for stale cached images.

```
Would you like to use LXD clustering? (yes/no) [default=no]:
Do you want to configure a new storage pool? (yes/no) [default=yes]:
Name of the new storage pool [default=default]: kube_zfs
Name of the storage backend to use (zfs, btrfs, ceph, dir, lvm, powerflex)
Create a new ZFS pool? (yes/no) [default=yes]:
Would you like to use an existing empty block device (e.g. a disk or partit
Size in GiB of the new loop device (1GiB minimum) [default=30GiB]: 400GiB
Would you like to connect to a MAAS server? (yes/no) [default=no]:
Would you like to create a new local network bridge? (yes/no) [default=yes]
What should the new bridge be called? [default=lxdbr0]:
What IPv4 address should be used? (CIDR subnet notation, "auto" or "none")
What IPv6 address should be used? (CIDR subnet notation, "auto" or "none")
Would you like the LXD server to be available over the network? (yes/no) [d
Address to bind LXD to (not including port) [default=all]: 23.29.118.75
Port to bind LXD to [default=8443]:
Would you like stale cached images to be updated automatically? (yes/no) [d
Would you like a YAML "lxd init" preseed to be printed? (yes/no) [default=n
```

LXD-UI Installation

<https://www.youtube.com/watch?v=5bhipSCgck8>

```
sudo snap install lxd
lxc --version
5.21.2 LTS
```

Enable LXD UI

```
sudo snap set lxd ui.enable=true
sudo snap restart --reload lxd
sudo lxc config set core.https_address 104.225.219.54:8443
```

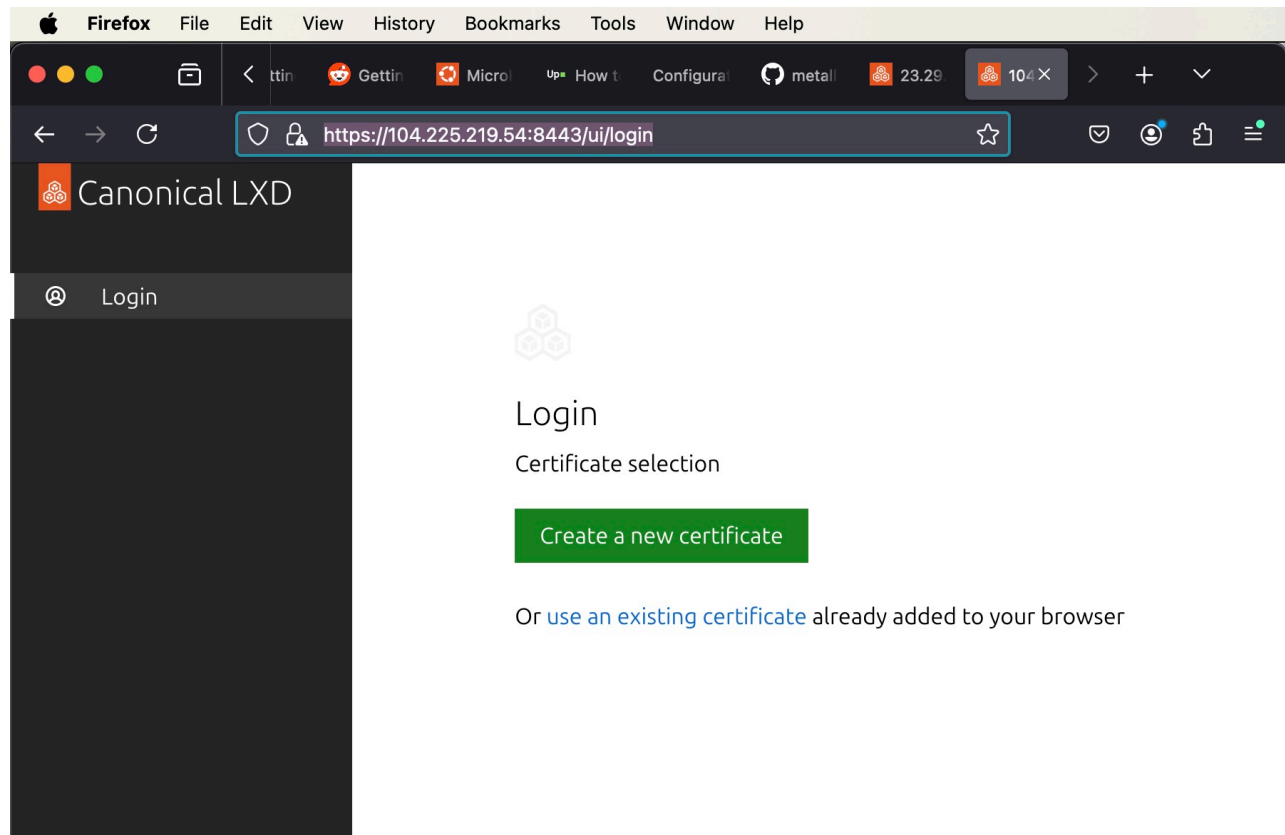
Note: in the setup of lxd that the option to make lxd available over the network was set to true and the port to bind to was set to 8442

Using Firefox go to the url of the host at port 8443:

```
https://104.225.219.54:8443/
```

Accept responsibility for the warnings. Accept the certificate exchange.

Ultimately you should arrive at the following page:

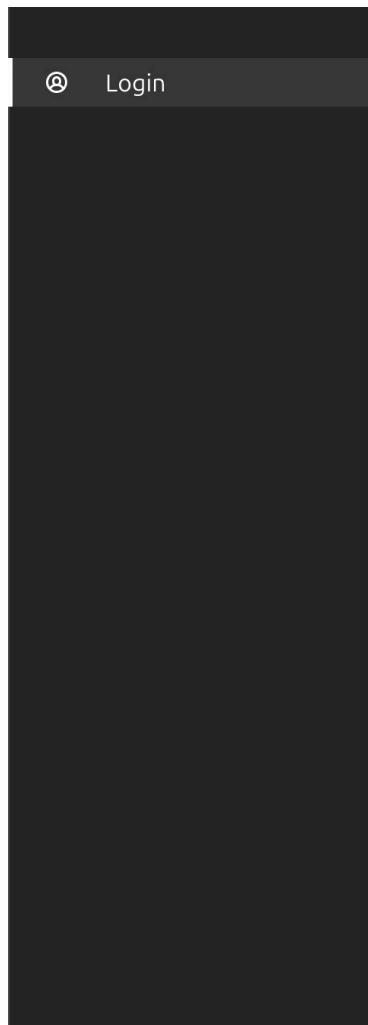


Create the certificate. Note that you do not need to provide a password.

Download the file to your local machine. Then you'll need to upload it to your cloud instance and issue the following command:

```
lxc config trust add lxd-ui-104.225.219.54.crt
```

Followed by getting the pfx file for your browser type as shown in the web page:



1. Generate

Create a new certificate

Generate



2. Trust

Download `lxd-ui.crt` and add it to the LXD trust store

```
$ lxc config trust add Downloads/lxd-ui.crt
```

Download crt

3. Import

Firefox

Chrome (Linux)

Chrome (Windows)

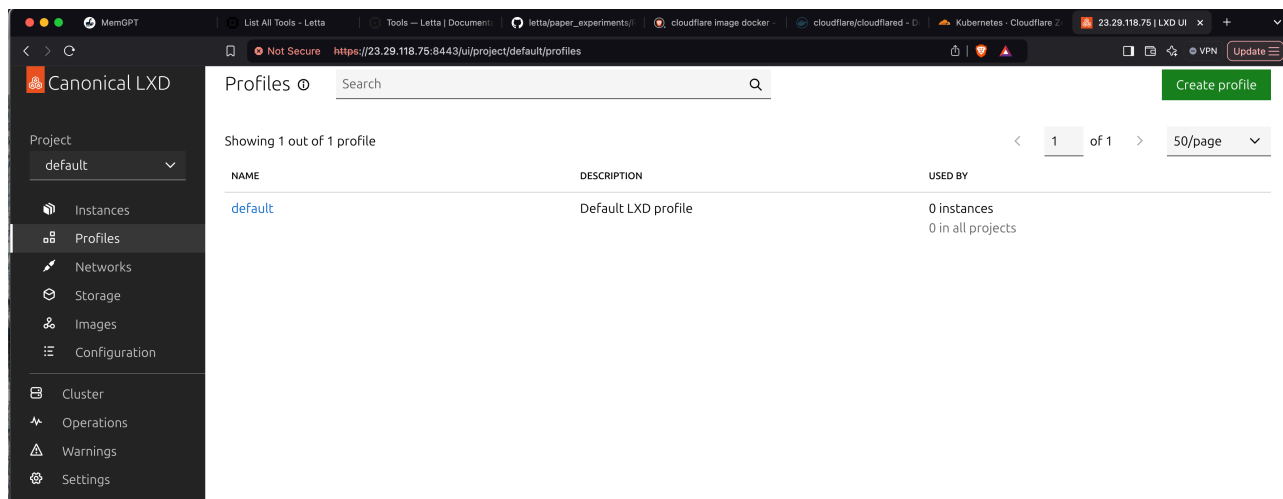
Edge

macOS

Download `lxd-ui.pfx`

Download pfx

Currently we have only the default profile for creating images.



The default contains some basic profile information. We will be adding another profile from the CLI that contains configuration for the Kubernetes installations.

```
lxc profile edit default
```

This will open the default profile in your default editor. Update the root device to look like this:

```
### This is a YAML representation of the profile.
### Any line starting with a '#' will be ignored.
###
### A profile consists of a set of configuration items followed by a set of
### devices.
###
### An example would look like:
### name: onenic
### config:
###   raw.lxc: lxc.aa_profile=unconfined
### devices:
###   eth0:
###     nictype: bridged
###     parent: lxdbr0
###     type: nic
###
### Note that the name is shown but cannot be changed

name: default
description: Default LXD profile
config: {}
devices:
  eth0:
    name: eth0
    network: lxdbr0
    type: nic
  root:
    path: /
    pool: kube_zfs
    type: disk
used_by: []
```

To create a container optimized for MicroK8s, a lightweight Kubernetes distribution, we need to create a custom profile.

```
lxc profile create microk8s_profile
```

This command creates a new LXC profile named `microk8s_profile`.

Next, we download a pre-configured profile from the MicroK8s repository:

```
wget https://raw.githubusercontent.com/ubuntu/microk8s/master/tests/lxc/mic
```

This command downloads the `microk8s-zfs.profile` file from the MicroK8s repository and saves it as `microk8s.profile` locally.

We then apply the downloaded profile to our `microk8s_profile` :

```
cat microk8s.profile | lxc profile edit microk8s_profile
```

This command pipes the contents of the `microk8s.profile` file to the `lxc profile edit` command, which applies the changes to the `microk8s_profile` .

You can view contents from the CLI as follows:

```
lxc profile edit microk8s_profile
```

Additionally, you can view it from the LXD-UI. The UI provides an intuitive view of the profile. We will not be going into this here but I'd encourage taking a look around at the parameters that you can constrain with respect to memory and storage.

Part Two: Setting Up the Containers and Kuberentes

Creating a LXC Container for the MicroK8s

The project requires three lxc containers. One will contain the control-plane and the other two containers will be worker containers.

You have the option of creating the container via the command line or using the UI. We'll describe the cli side and leave it to the reader to use the UI.

We use the lxc profiles to launch a lxc container:

```
lxc launch -p default -p microk8s_profile ubuntu:20.04 control
```

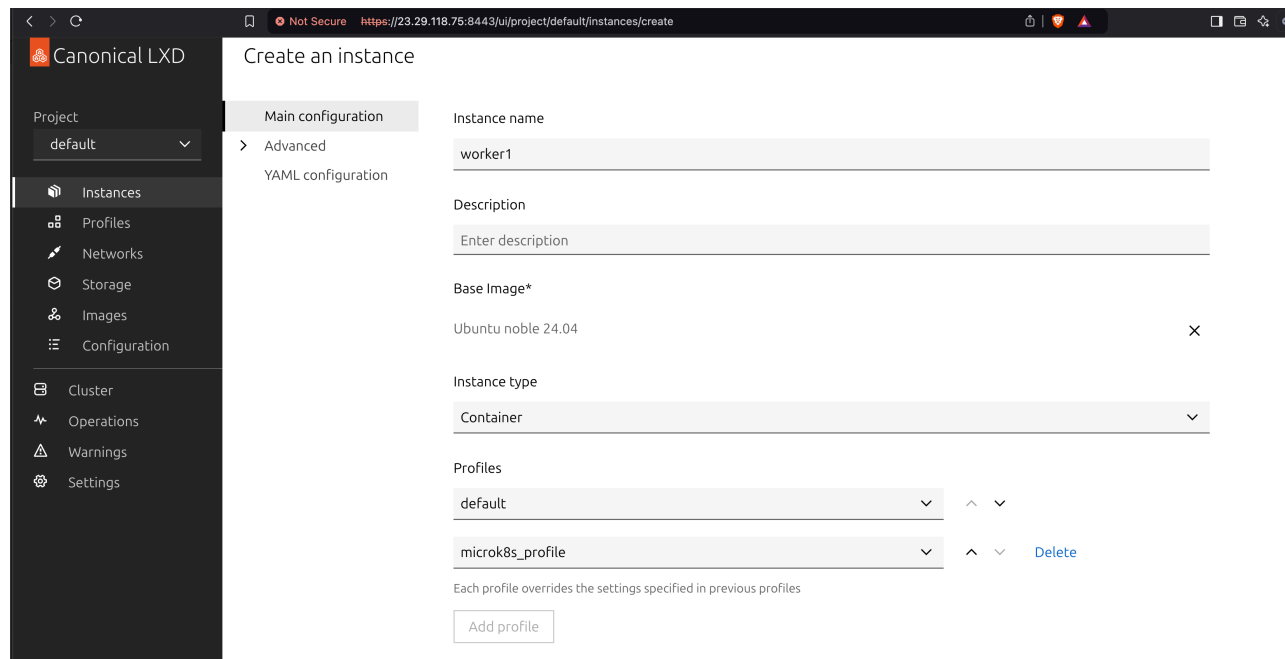
Note: that there are two profiles used. In the UI two profiles will need to be selected.

Let's break down this command:

- `lxc launch` : Creates a new container from a specified image.
- `-p default` : Specifies the default profile to use for the container.
- `-p microk8s_profile` : Specifies the additional `microk8s_profile` profile to use for the container.
- `ubuntu:20.04` : Specifies the Ubuntu 20.04 image to use for the container.
- `microk8scontainer` : Specifies the name of the container.

In summary, this command creates a new container named `control` from the Ubuntu 20.04 image, using both the default and `microk8s_profile` profiles. The `microk8s_profile` likely configures the container to run MicroK8s, allowing you to have a Kubernetes environment up and running quickly.

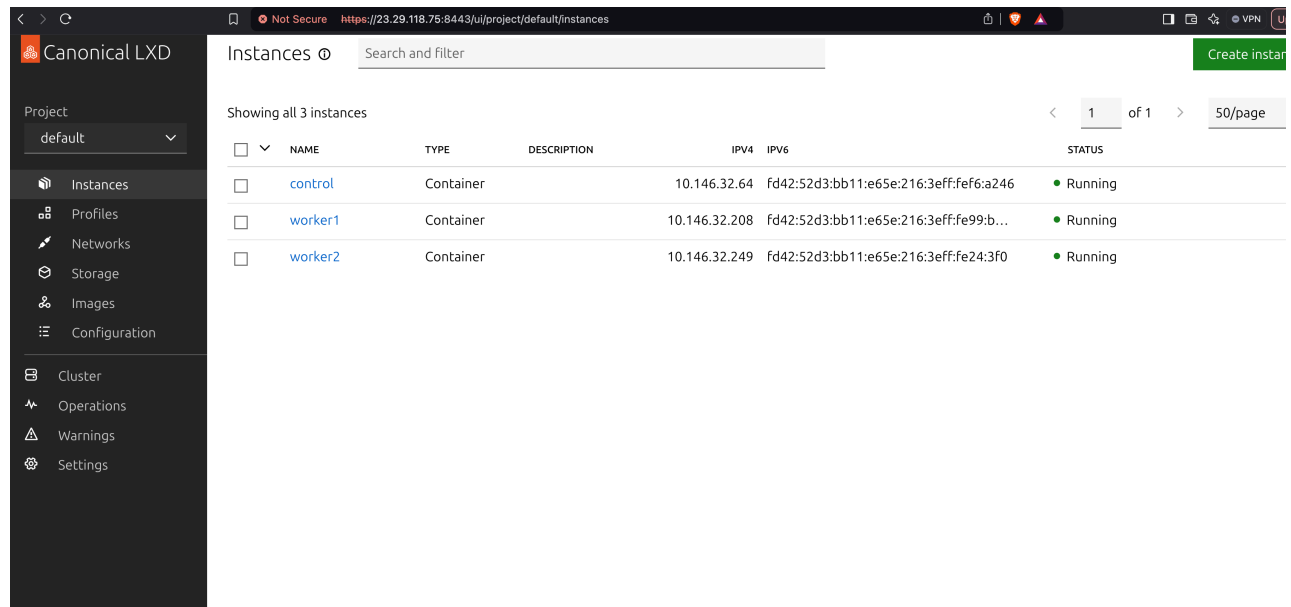
Using the same command create an instance named 'worker1' and 'worker2'.



The screenshot shows the Canonical LXD web interface in a browser. The left sidebar contains navigation links: Project (default), Instances, Profiles, Networks, Storage, Images, Configuration, Cluster, Operations, Warnings, and Settings. The main content area is titled 'Create an instance' and has tabs for 'Main configuration' (selected), 'Advanced', and 'YAML configuration'. The 'Main configuration' tab contains the following fields:

- Instance name:** A text input field containing 'worker1'.
- Description:** A text input field with the placeholder 'Enter description'.
- Base Image*:** A dropdown menu showing 'Ubuntu noble 24.04' with a close button (X) to its right.
- Instance type:** A dropdown menu showing 'Container'.
- Profiles:** A section with two dropdown menus. The first shows 'default' and the second shows 'microk8s_profile'. Between them are up and down arrow icons. To the right of the second dropdown is a 'Delete' link. Below the dropdowns is the text 'Each profile overrides the settings specified in previous profiles' and an 'Add profile' button.

Should look like this when they are all launched.



Alternative from the command line:

```
lxc launch -p default -p microk8s_profile ubuntu:20.04 worker1
lxc launch -p default -p microk8s_profile ubuntu:20.04 worker2
```

```
lxc ls
```

```
└─(00:15:00)─> lxc ls
+-----+-----+-----+-----+
| NAME   | STATE | IPV4          | IPV6          |
+-----+-----+-----+-----+
| control | RUNNING | 10.146.32.64 (eth0) | fd42:52d3:bb11:e65e:216:3eff:fe |
+-----+-----+-----+-----+
| worker1 | STOPPED |                  |                  |
+-----+-----+-----+-----+
| worker2 | STOPPED |                  |                  |
+-----+-----+-----+-----+
```

install Microk8s on control

```
lxc exec control -- sudo snap install microk8s --classic
```

prep_container.sh

```
#!/bin/sh

systemctl enable iscsid
swapoff -a

apt install -y zfsutils-linux
```

```
lxc file push prep_container.sh control/tmp/
lxc exec control -- /bin/bash /tmp/prep_container.sh
```

containerScriptLauncher.sh

```
#!/bin/sh

# Set the script file and container name from command-line arguments
CONTAINER_NAME=$1
SCRIPT_FILE=$2

# Push the script file to the container
lxc file push "$SCRIPT_FILE" "$CONTAINER_NAME"/tmp/

# Execute the script inside the container
lxc exec "$CONTAINER_NAME" -- /bin/bash /tmp/"$SCRIPT_FILE"
```

To simplify things we can use the following utility script to deploy the script to a container more efficiently.

containerScriptLauncher.sh

```
./containerScriptLauncher.sh <container> <script>

i.e.

./containerScriptLauncher.sh control prep_container.sh
```

Enable required 'addons':

enables.sh

```
#!/bin/sh
microk8s enable community
microk8s enable rbac
microk8s enable openebs
```

```
./containerScriptLauncher.sh control enables.sh
./containerScriptLauncher.sh worker1 enables.sh
./containerScriptLauncher.sh worker2 enables.sh
```

OpenEBS ZFS Operator Installation

Now We need to add support for the ZFS files system with the OpenEBS ZFS operator. Before we install the storage class are:

```
ku get sc
```

NAME	PROVISIONER	RECLAIMPOLICY	VOLUMEBIND
openebs-device	openebs.io/local	Delete	WaitForFir
openebs-hostpath	openebs.io/local	Delete	WaitForFir
openebs-jiva-csi-default	jiva.csi.openebs.io	Delete	Immediate

In this section, we will install the OpenEBS ZFS operator on our Ubuntu cloud instance. OpenEBS is a popular open-source storage solution for Kubernetes, and the ZFS operator provides a scalable and performant storage solution for our cluster. [OpenEBS](#)

Installing the OpenEBS ZFS Operator

```
#!/bin/sh
lxc exec control -- microk8s kubectl apply -f https://openebs.github.io/cha
```

```
./containerApplyYaml.sh control https://openebs.github.io/charts/zfs-operat
```

This script applies the OpenEBS ZFS operator YAML file to our Kubernetes cluster using `microk8s kubectl`. The `apply` command is used to create or update resources in our cluster based on the YAML file. The YAML file is fetched from the OpenEBS GitHub repository.

Note: `microk8s` is a lightweight, single-package distribution of Kubernetes that is easy to install and use. It provides a convenient way to run Kubernetes on a single machine.

Setting the Default Storage Class

Actually the following may not be required if openebs-jiva-csi is already default. Still you can define this in the definition of the storage class.

```
microk8s kubectl patch storageclass microk8s-hostpath -p '{"metadata": {"an  
microk8s kubectl patch storageclass openebs-zfspv -p '{"metadata": {"annota
```

These two commands set the default storage class for our cluster. We want to make the `openebs-zfspv` storage class the default, so we patch the `microk8s-hostpath` storage class to set `is-default-class` to `false`, and then patch the `openebs-zfspv` storage class to set `is-default-class` to `true`.

Note: In Kubernetes, a storage class is a way to define a class of storage that can be used to provision volumes. By setting a default storage class, we can simplify the process of provisioning volumes for our applications.

By running these scripts, we have successfully installed the OpenEBS ZFS operator and set the default storage class to `openebs-zfspv`. This provides a scalable and performant storage solution for our Kubernetes cluster.

MetalLB setup (control container only)

MetalLB is a library that provides an external IPs for for the node or cluster. When using cloud providers like AWS or Azure they provide loadbalancers for you. In this case we are using our own. This is offer referred to as a bare-metal setup.

Microk8s provides MetalLB as an add on that makes its addition easy. However, a IP range needs to be given on enabling the addon.

To select a range of IPs check the ip of the current container and then find a range of 10 IPs above the current. For instance if the IP is: 10.220.44.112 you could select 10.220.44.115-10.220.44.115.

Checking the subnetwork of your container for IP Range

When setting up MetalLB, you need to specify a range of IP addresses that can be used to assign IP addresses to your Kubernetes services. To determine what range you can specify, you'll need to consider a few factors:

1. **Subnet:** Identify the subnet that your LXC container's IP address (10.196.148.157) belongs to. You can use the `ip addr show` command to find the subnet mask:

`ip addr show eth0` Look for the `inet` line, which should display the IP address, subnet mask, and other information. For example:

`inet 10.196.148.157/24 brd 10.196.148.255 scope global eth0` In this example, the subnet mask is `/24`, which means the subnet has 256 available IP addresses (2^8).

2. **Available IP range:** Determine the available IP range within the subnet that is not already in use by other devices or services. You can use tools like `nmap` or `arp-scan` to scan the subnet and identify available IP addresses.

For example, if the subnet is `10.196.148.0/24`, you might find that IP addresses `10.196.148.1` to `10.196.148.100` are already in use. In this case, you could specify a range like `10.196.148.101-10.196.148.150` for MetalLB.

1. **Size of the range:** Decide on the size of the IP range you want to allocate to MetalLB. A smaller range (e.g., 10-20 IP addresses) might be sufficient for a small cluster, while a larger range (e.g., 50-100 IP addresses) might be needed for a larger cluster.

Considering these factors, you could specify a range like

`10.196.148.101-10.196.148.120` for MetalLB. This range is within the same subnet as your LXC container's IP address, and it's small enough to avoid conflicts with other devices or services on the subnet.

Remember to update your MetalLB configuration to reflect the chosen IP range.

You can use the UI to find the range as show here:

Canonical LX Daemon

INSTANCES / CONTROL *Running*

Overview Configuration Snapshots **Terminal** Console Logs

Project: default

Instances

Profiles

Networks

Storage

Images

Configuration

Cluster

Operations

Warnings

Settings

```
root@microk8scontainer:~#  
root@microk8scontainer:~# ls  
snap  
root@microk8scontainer:~# ip addr show eth0  
6: eth0@if7: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 qdisc noqueue state UP group default qlen 1000  
    link/ether 00:16:3e:f6:a2:46 brd ff:ff:ff:ff:ff:ff link-netnsid 0  
    inet 10.146.32.64/24 metric 100 brd 10.146.32.255 scope global dynamic eth0  
        valid_lft 3197sec preferred_lft 3197sec  
    inet6 fd42:52d3:bb11:e65e:216:3eff:fef6:a246/64 scope global mngtppaddr noprefixroute  
        valid_lft forever preferred_lft forever  
    inet6 fe80::216:3eff:fef6:a246/64 scope link  
        valid_lft forever preferred_lft forever  
root@microk8scontainer:~#
```

Alternatively from the commandline:

```
lxc exec control -- ip addr show eth0
```

In the case shown in the diagram IP and subnet is:

```
10.146.32.64/24
```

IP Address:	10.146.32.64
Network Address:	10.146.32.0
Usable Host IP Range:	10.146.32.1 – 10.146.32.254
Broadcast Address:	10.146.32.255
Total Number of Hosts:	256
Number of Usable Hosts:	254

The following four IP addresses are not usable or already have a special meaning:

```
10.146.32.0: The network address, which is not usable as a host address.  
10.146.32.1: Often used as the default gateway or router address, although  
10.146.32.64: The network address (in this specific case), which is not usable  
10.146.32.255: The broadcast address, which is used to send packets to all
```

We'll use the range 10.146.32.70-10.146.32.100. 30 IPs are sufficient for our needs.

Note also that MetalLB only needs to be installed on the node with the control plane.


```
lxc exec control -- microk8s enable metallb
```

Infer repository core for addon metallb

Enabling MetallB

Enter each IP address range delimited by comma (e.g. '10.64.140.43-10.64.14

Applying Metallb manifest

```
customresourcedefinition.apiextensions.k8s.io/addresspools.metallb.io creat
```

```
customresourcedefinition.apiextensions.k8s.io/bfdprofiles.metallb.io create
```

```
customresourcedefinition.apiextensions.k8s.io/bgpadvertisements.metallb.io
```

```
customresourcedefinition.apiextensions.k8s.io/bgppeers.metallb.io created
```

:

```
Error from server (InternalError): error when creating "STDIN": Internal er
```

```
Error from server (InternalError): error when creating "STDIN": Internal er
```

Failed to create default address pool, will retry

```
ipaddresspool.metallb.io/default-addresspool created
```

```
l2advertisement.metallb.io/default-advertise-all-pools created
```

MetallB is enabled

Restart Protection with AppArmor

`microk8srestartprotection.sh`

```
#!/bin/sh
```

```
cat > /etc/rc.local <<EOF
```

```
#!/bin/bash
```

```
apparmor_parser --replace /var/lib/snapd/apparmor/profiles/snap.microk8s.*
```

```
exit 0
```

```
EOF
```

```
chmod +x /etc/rc.local
```

This script does the following:

- Enables the `iscsid` service to start on boot using `systemctl enable`
- Creates a new `/etc/rc.local` file with a script that:
 - Parses the AppArmor profiles for MicroK8s using `apparmor_parser --replace`
 - Exits with a success code using `exit 0`
- Makes the `/etc/rc.local` file executable using `chmod +x`

This script sets up restart protection for MicroK8s using AppArmor, ensuring that MicroK8s continues to run even after a reboot.

```
./containerScriptLauncher.sh control microk8s_restart_protection.sh
./containerScriptLauncher.sh worker1 microk8s_restart_protection.sh
./containerScriptLauncher.sh worker2 microk8s_restart_protection.sh
```

Install k9s on Host

We are going to install k9s on the host to the lxc containers. First create a .kube directory off of home and populate it with the data for connecting to the node:

```
mkdir ~/.kube
# Step 1: Save the current Kubernetes configuration to a file
lxc exec microk8s config view > ~/.kube/config

# Step 2: Set the correct permissions for the config file
chmod 600 ~/.kube/config

# Step 3: Download the K9s installation package
wget https://github.com/derailed/k9s/releases/download/v0.32.5/k9s_linux_amd64.deb

# Step 4: Change the ownership of the downloaded package to root
sudo chown root:root k9s_linux_amd64.deb

# Step 5: Install K9s using the downloaded package
sudo apt install -y ./k9s_linux_amd64.deb
```

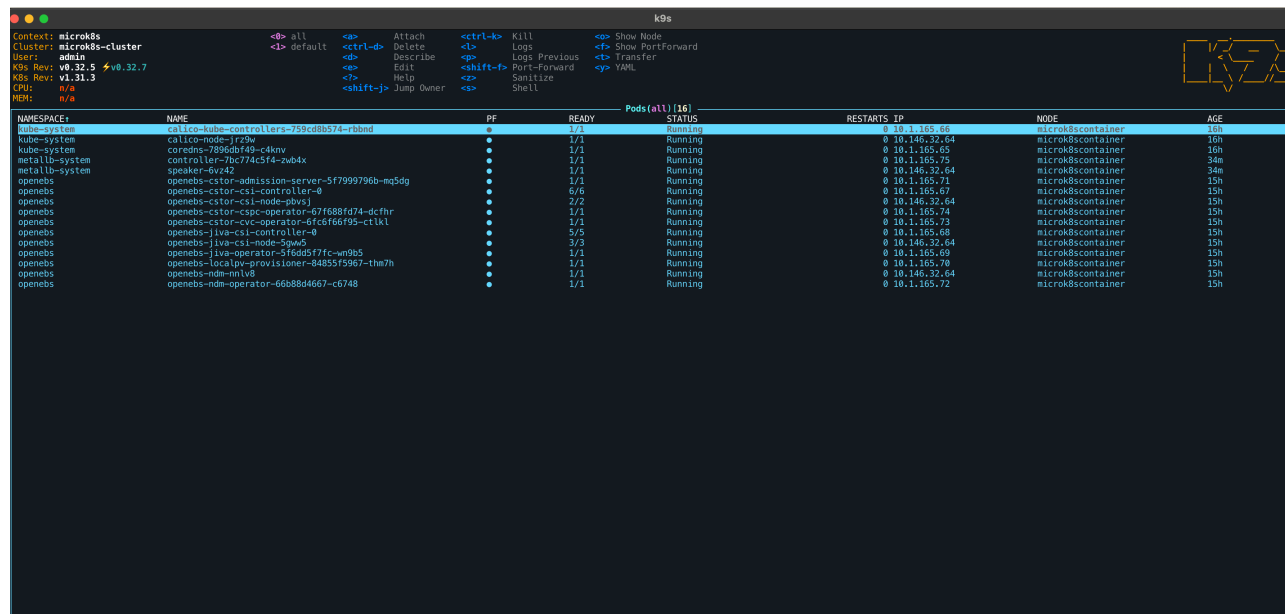
We need to edit the ~/.kube/config file. It has identified the address of the kubernetes node as 127.0.0.1. We need to provide its IP address 10.146.32.64.

```

apiVersion: v1
clusters:
- cluster:
    certificate-authority-data: LS0tLS1CRUdJTiBDRVJUSUZJQ0FURS0tLS0tCk1JSUR
    server: https://10.146.32.64:16443
    name: microk8s-cluster
contexts:
- context:
    cluster: microk8s-cluster
    user: admin
    name: microk8s
current-context: microk8s
kind: Config
preferences: {}
users:
- name: admin
  user:
    client-certificate-data: LS0tLS1CRUdJTiBDRVJUSUZJQ0FURS0tLS0tCk1JSUN6RE

```

k9s is now ready to use.



NAMESPACE	NAME	PF	READY	STATUS	RESTARTS	IP	NODE	AGE
kube-system	calico-kube-controllers-759cd8b574-rb6md	•	1/1	Running	0	10.1.165.66	microk8scontainer	15h
kube-system	calico-node-r25w	•	1/1	Running	0	10.146.32.64	microk8scontainer	16h
kube-system	coredns-7896dbf49-c4kv	•	1/1	Running	0	10.1.165.65	microk8scontainer	16h
metallb-system	controller-7bc774c5f4-zwb4x	•	1/1	Running	0	10.1.165.75	microk8scontainer	34m
metallb-system	speaker-6vz42	•	1/1	Running	0	10.146.32.64	microk8scontainer	34m
openeb	openeb-ctor-admission-server-5f7999796b-mq5dg	•	1/1	Running	0	10.1.165.71	microk8scontainer	15h
openeb	openeb-ctor-csi-controller-0	•	6/6	Running	0	10.1.165.67	microk8scontainer	15h
openeb	openeb-ctor-csi-node-plusj	•	2/2	Running	0	10.146.32.64	microk8scontainer	15h
openeb	openeb-ctor-cspc-operator-67f688fd74-dcfhr	•	1/1	Running	0	10.1.165.74	microk8scontainer	15h
openeb	openeb-ctor-cyc-operator-67c5f66f95-cttkl	•	1/1	Running	0	10.1.165.73	microk8scontainer	15h
openeb	openeb-jlves-csi-controller-0	•	5/5	Running	0	10.1.165.68	microk8scontainer	15h
openeb	openeb-jlves-csi-node-5gw5	•	3/3	Running	0	10.146.32.64	microk8scontainer	15h
openeb	openeb-jlves-operator-5f6d5f7fc-w8965	•	1/1	Running	0	10.1.165.69	microk8scontainer	15h
openeb	openeb-localpv-provisioner-8485f5967-thm7n	•	1/1	Running	0	10.1.165.70	microk8scontainer	15h
openeb	openeb-ndm-nnlv8	•	1/1	Running	0	10.146.32.64	microk8scontainer	15h
openeb	openeb-ndm-operator-66b8bd4667-c6748	•	1/1	Running	0	10.1.165.72	microk8scontainer	15h

Join Worker Nodes to the Control Plane

This command is executed on the control node:

```
lxc exec control -- microk8s add-node
```

It returns the information required to join nodes to create the cluster of nodes:

```
From the node you wish to join to this cluster, run the following:
microk8s join 10.146.32.64:25000/24d8e83fcfa33ed2ef7c396c83908084/2a9fa5a83

Use the '--worker' flag to join a node as a worker not running the control
microk8s join 10.146.32.64:25000/24d8e83fcfa33ed2ef7c396c83908084/2a9fa5a83

If the node you are adding is not reachable through the default interface y
microk8s join 10.146.32.64:25000/24d8e83fcfa33ed2ef7c396c83908084/2a9fa5a83
microk8s join fd42:52d3:bb11:e65e:216:3eff:fef6:a246:25000/24d8e83fcfa33ed2
```

We will use the worker flag and join worker1:

```
lxc exec worker1 -- microk8s join 10.146.32.64:25000/24d8e83fcfa33ed2ef7c39
```

It returns:

```
Contacting cluster at 10.146.32.64

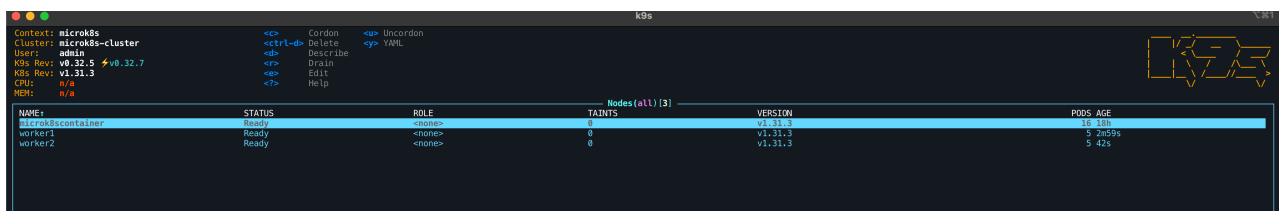
The node has joined the cluster and will appear in the nodes list in a few

This worker node gets automatically configured with the API server endpoint
If the API servers are behind a loadbalancer please set the '--refresh-inte
/var/snap/microk8s/current/args/apiserver-proxy
and replace the API server endpoints with the one provided by the loadbalan
/var/snap/microk8s/current/args/traefik/provider.yaml

Successfully joined the cluster.
```

We repeat this process to add worker2 to the cluster.

k9s should now see the nodes as shown below:



The screenshot shows the k9s terminal interface. On the left, there is a sidebar with cluster details: Context: microk8s, Cluster: microk8s-cluster, User: admin, K8s Ver: v0.32.5, K8s Rev: v1.31.3, CPU: n/a, MEM: n/a. The main area displays a table of nodes. The table has columns: NAME, STATUS, ROLE, TAINTS, VERSION, and PODS AGE. There are three nodes listed: microk8scontainer, worker1, and worker2. All three are in a 'Ready' state and have no taints. worker1 and worker2 are version v1.31.3 and have been up for 5m25s and 5m42s respectively.

NAME	STATUS	ROLE	TAINTS	VERSION	PODS AGE
microk8scontainer	Ready	<none>	0	v1.31.3	5m25s
worker1	Ready	<none>	0	v1.31.3	5m25s
worker2	Ready	<none>	0	v1.31.3	5m42s

Part Three: Test Deployments

In this section we are going to: 1. deploy a simple application Nginx. The purpose is to use MetalLB to expose the IP. We will take this a step further and make the Nginx service available to the Internet through CloudFlare. 2. Next we will deploy an application that requires Disk storage. We will set this up and view the elements created and review the ZFS datasets.

Notes: We have deployed, but not documented, the nginx deployment. We might do this and who the service with its external IP. Then we can delete the deployment and add the ingress addon.

The new method requires additional addons enabled:

[Intersting supporting article for Ingress](#)

```
microk8s enable ingress
```

Deploy the Ingress Service

Ingress_Service.yaml

```
apiVersion: v1
kind: Service
metadata:
  name: ingress
  namespace: ingress
spec:
  selector:
    name: nginx-ingress-microk8s
  type: LoadBalancer
  ports:
    - name: http
      protocol: TCP
      port: 80
      targetPort: 80
    - name: https
      protocol: TCP
      port: 443
      targetPort: 443
```

You will see the ip set for external (thanks to MetalLB as there service is of type LoadBalancer)

Ingress definition

```
apiVersion: networking.k8s.io/v1
kind: Ingress
metadata:
  name: web-ingress
spec:
  ingressClassName: nginx
  rules:
  - host: api.davidsells.today
    http:
      paths:
      - path: /users
        pathType: Prefix
        backend:
          service:
            name: user-service
            port:
              number: 8081
      - path: /products
        pathType: Prefix
        backend:
          service:
            name: item-service
            port:
              number: 9091
      - path: /ng
        pathType: Prefix
        backend:
          service:
            name: ng-service
            port:
              number: 9999
```

Deploy.yaml

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: metallb-test
  namespace: lb-test
spec:
  replicas: 2
  selector:
    matchLabels:
      app: metallb-test
  template:
    metadata:
      labels:
        app: metallb-test
    spec:
      containers:
        - name: metallb-test
          image: nginx:latest
          ports:
            - containerPort: 80
---
apiVersion: v1
kind: Service
metadata:
  creationTimestamp: null
  name: ng-service
  namespace: lb-test
  labels:
    app: ng-service
spec:
  ports:
    - port: 9999
      protocol: TCP
      targetPort: 80
  selector:
    app: metallb-test
  type: ClusterIP
status:
  loadBalancer: {}
```

Install Cloudflared

```
wget -q https://github.com/cloudflare/cloudflared/releases/latest/download/
cloudflared tunnel login

cloudflared tunnel create llm

vi config.yaml

tunnel: 9a27e9b6-208c-460e-ba6e-9182c2437fb1

credentials-file: /root/.cloudflared/9a27e9b6-208c-460e-ba6e-9182c2437fb1.j

ingress:

- hostname: web.davidsells.today

  service: http://10.203.176.200:80

- service: http_status:404

cloudflared tunnel route dns llm web.davidsells.today

cloudflared tunnel run llm

curl http://10.203.176.200:80

vi config.yaml

cloudflared tunnel run kubernetes

cloudflared service install

systemctl start cloudflared
```

NOTE:

We have a deployment with bridges to ingresses. We were looking at having some content in the nginx servers. ie. unique html pages

sc.yaml


```
apiVersion: storage.k8s.io/v1
kind: StorageClass
metadata:
  name: openebs-zfspv
parameters:
  recordsize: "4k"
  compression: "off"
  dedup: "off"
  fstype: "zfs"
  poolname: "kube_zfs"
  mountOptions: "zfsutil"
provisioner: zfs.csi.openebs.io
volumeBindingMode: WaitForFirstConsumer
```

```

apiVersion: v1
kind: PersistentVolume
metadata:
  name: html-pv
  namespace: lb-test-one
spec:
  capacity:
    storage: 1Gi
  accessModes:
    - ReadWriteOnce
  persistentVolumeReclaimPolicy: Retain
  local:
    path: /mnt/data
  storageClassName: openebs-zfspv
  nodeAffinity:
    required:
      nodeSelectorTerms:
        - matchExpressions:
            - key: kubernetes.io/hostname
              operator: Exists

---
apiVersion: v1
kind: PersistentVolumeClaim
metadata:
  name: html-pvc
  namespace: lb-test-one
spec:
  accessModes:
    - ReadWriteOnce
  resources:
    requests:
      storage: 1Gi

```

You create a StorageClass YAML that defines the provisioner and parameters for creating a PV. You create a PVC YAML that references the StorageClass. When you create the PVC, the StorageClass provisioner (in this case, zfs.csi.openebs.io) dynamically creates a PV that matches the PVC's requirements.

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: metallb-test-one
  namespace: lb-test-one
spec:
  replicas: 2
  selector:
    matchLabels:
      app: metallb-test-one
  template:
    metadata:
      labels:
        app: metallb-test-one
    spec:
      containers:
        - name: metallb-test-one
          image: nginx:latest
          ports:
            - containerPort: 80
          volumeMounts:
            - name: html-volume
              mountPath: /usr/share/nginx/html
      volumes:
        - name: html-volume
          persistentVolumeClaim:
            claimName: html-pvc
```

```

# Updating a PVs contents

# Get PV name
kubectl get pvc <pvc-name> -o jsonpath='{.spec.volumeName}'
kubectl get pvc html-pvc -o jsonpath='{.spec.volumeName}'

# Get Data set name
kubectl get pv <pv-name> -o jsonpath='{.spec.local.path}'

# Mount Dataset
zfs set mountpoint=/mnt/data/<dataset-name> kube_zfs/<dataset-name>
zfs mount kube_zfs/<dataset-name>

# Update Data
echo "<html><body><h1>New HTML content!</h1></body></html>" > /mnt/data/<dataset-name>

# Unmount Dataset
zfs unmount kube_zfs/<dataset-name>
zfs mount kube_zfs/<dataset-name>

```

We were also talking with Groq about zfs settings and found that there was more than one mount point...

However then we lost it:

```
79 find . -type d -name kube_zfs 2> /dev/null
```

We like the idea of nginx server with unique html pages that also have memory: StorageClass, PersistentVolume etc...

[Excellent discussion of Mayastore](#)

[Another excellent presentation - perhaps better](#)

NOTE

Let's depart from this for a bit. There were some short comings in our understanding of replicate sets and storage. Our expectation of synchronization between the pvc were not met. replicate sets, StatefulSets and DaemonSets were not what we were looking for. So if we use a deployment with a template for replicats it will also be creating an additional pvc that will be independent of other pvc contained within the set.

Part Four: LLM Deployments

In this section we will deploy OpenWebUI, Ollama and LiteLLM. OpenWebUI provides a wonderful platform for experimenting with LLMs. Coupled with LiteLLM it gives it access to integration with external LLM providers.

Trouble Shooting

Local Registry for Worker nodes

The worker nodes do not have access to the external internet. We need to setup a local repository that is accessible to worker1 and worker2.

Setting up Registry

Create a new lxd container

Using the lxd-ui create a new container insure that you use the default and microk8s profiles

Install Docker

Let's make this easy by using snap.

Setting up Docker Registry

This article from DigitalOcean's website is very good. No need to repeat here:

<https://www.digitalocean.com/community/tutorials/how-to-set-up-a-private-docker-registry-on-ubuntu-20-04>

imagePullSecret