title: My Markdown Document $type: article

description: This is a sample Markdown document

Part One: The Host

https://stackblitz.com/github/code-hike/v1-starter?file=app%2Fscrollycoding%2Fcontent.md

I. Introduction

- Machine
- Operating System: Ubuntu 24.04

II. Setting up the Host Machine

- Setting up SSH
  - Generating an SSH Key Pair
  - Sending the Public Key to the Cloud
  - Configuring the SSH Client
- Setting up LXD on Ubuntu
  - Why LXD?
  - Installing LXD
  - Initializing LXD
  - LXD Configuration Options

III. Initial Setup for Ubuntu Cloud Instance with ZFS and MicroK8s

```
┌─(~)──────────────────────────────────────────────────────────────(ROOT@phoenix1733435735:pts/0)─┐
└─(00:24:03)──> lxd init                                                          ──(Fri,Dec20)─┘

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) [default=zfs]:
Create a new ZFS pool? (yes/no) [default=yes]:
Would you like to use an existing empty block device (e.g. a disk or partition)? (yes/no) [default=no]:
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”) [default=auto]:
What IPv6 address should be used? (CIDR subnet notation, “auto” or “none”) [default=auto]:
Would you like the LXD server to be available over the network? (yes/no) [default=no]: yes
Address to bind LXD to (not including port) [default=all]:
Port to bind LXD to [default=8443]:
Would you like stale cached images to be updated automatically? (yes/no) [default=yes]:
Would you like a YAML "lxd init" preseed to be printed? (yes/no) [default=no]:
┌─(~)──────────────────────────────────────────────────────────────────────────────────────────────┐
```

- Installing ZFS Tools
- Setup Profile for Creating Containers
- Creating a Container with MicroK8s
- Installing MicroK8s inside the Container

Part Two: The Container

I. Introduction

- Setting up ZSH on a fresh Ubuntu instance
- Benefits of using ZSH over traditional shells like Bash

II. Setting up ZSH Environment

`zsh_setup.sh`

- Installing ZSH using `apt`
- Setting ZSH as the default shell
- Installing `git` and `curl`
- Installing Oh My ZSH framework

`zsh_extra.sh`

- Configuring additional ZSH settings and plugins
- Defining a custom function `hist()` for filtering command history
- Enabling vi-mode keybindings
- Setting the ZSH theme and plugins
- Setting the default editor to `vim`
- Defining aliases for `microk8s kubectl` commands

III. Microk8s Setup

`microk8s_setup.sh`

- Installing MicroK8s using `snap`
- Enabling various MicroK8s features

`microk8s_restart_protection.sh`

- Setting up restart protection for MicroK8s using AppArmor

IV. K9s Installation

`k9s_setup.sh`

- Installing K9s on Ubuntu
- Configuring K9s to use the Kubernetes configuration file

V. OpenEBS ZFS Operator Installation

`openebs_zfs_operator.sh`

- Installing the OpenEBS ZFS operator
- Setting the default storage class to `openebs-zfspv`

VI. Creating a Persistent Volume Claim (PVC)

- Creating a PVC to request storage resources
- Defining a storage class YAML file

VII. Setting up Persistent Storage in Kubernetes

- Creating a ZFS pool for storage class
- Defining a storage class YAML file
- Creating a PVC to request storage resources

VIII. Nginx Deployment - PVC Test

- Creating a Kubernetes deployment that uses a PVC
- Defining a deployment YAML file

IX. Nginx Deployment - MetalLB Test

- Creating a Kubernetes deployment that uses MetalLB
- Defining a deployment YAML file

X. Setting up Ollama and Open-WebUI on Ubuntu with Kubernetes

- Creating a ZFS pool for `llm` storage class

- Defining a storage class YAML file
- Creating a namespace for deployments
- Defining a PVC to request storage resources
- Creating deployments for Ollama and Open-WebUI

XI. Setting up LiteLLM on a New Ubuntu Cloud Instance

- Configuring LiteLLM using a `config.yaml` file
- Creating a ConfigMap to store configuration data
- Defining a Deployment YAML file to deploy LiteLLM
- Creating a Service to make LiteLLM available to other pods

Introduction

I'm always interested in find economical ways to explore new technologies. However, when you go this route you find that there are more problems to solve. Whereas if one uses AWS, Azure or other cloud providers Kubernetes infrastructures are provide with loadbalancers and gateways. This article aims to assist in setting up the infrasturture to get you working in an economical fully functional environment. The cloud instance is provisioned by SSDNodes. They have an assortment of configuration and terms. We will leave the aspect of abtaining a cloud instance to the reader.

Goals of Project

The Goal of the project is to setup a basic kubernetes infrastructure. Then with in that Kubernetes environment deploy a LLM environment, similar to ChatGPT, for private explorations.

The environment setup will be in three parts:

1. The Host Setup

   In this part we will configure:

   - SSL
   - Zsh/OhMyZsh
   - LXD
   - ZFS

2. The Container Setup

   Here we will configure:

   - Zsh/OhMyZsh
   - Microk8s
   - MetalLB
   - k9s
   - OpenEBS
   - Test Deployments to validate configuration
   - CloudFare

3. Deployment of the LLM environment

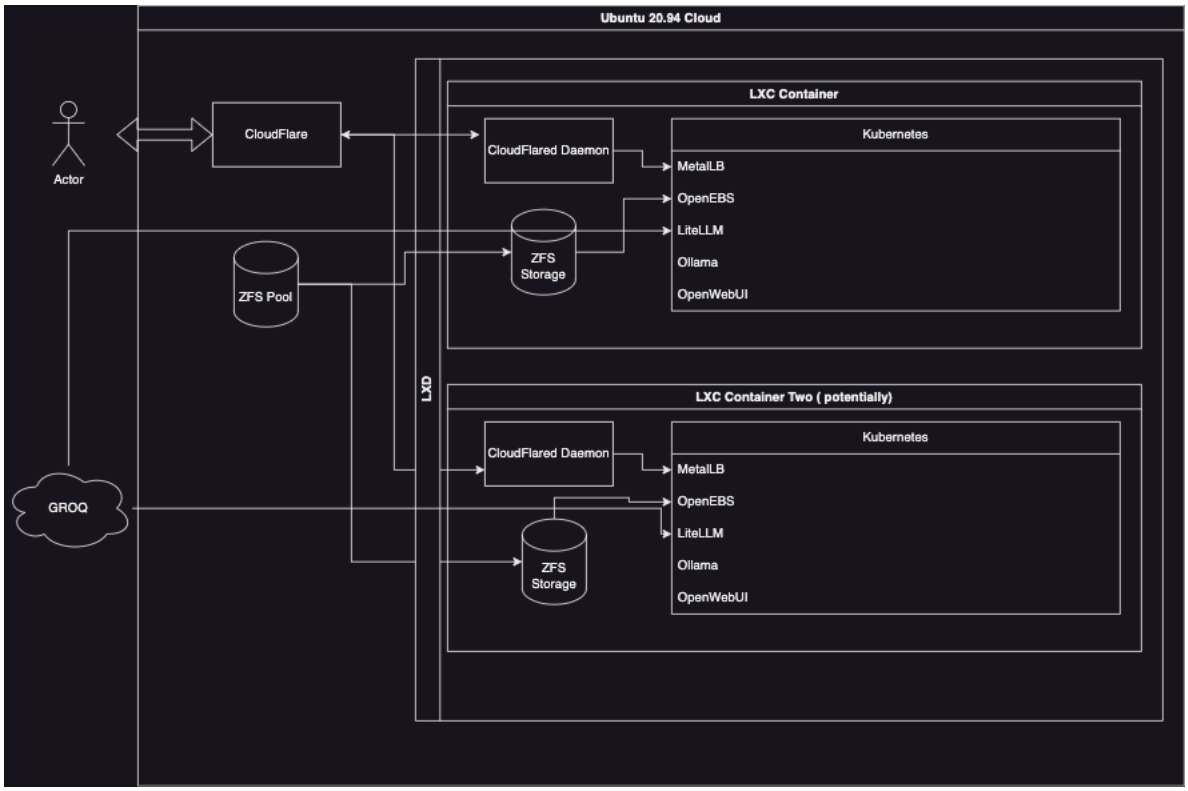
   Install:

   - Ollama
   - Open-WebUI
   - LiteLLM

Machine

- Hardware: Performance 48GB RAM (720GB NVMe) with 12 vCPUs
- Operating System: Ubuntu 24.04

Architecture



Setting up the Host Machine

Setup SSH

Setting up SSH (Secure Shell) is an essential step in making your cloud instance easily accessible. In this section, we'll go through the process of generating an SSH key pair, sending the public key to the cloud, and configuring the SSH client to use the key.

Generating an SSH Key Pair (local machine)

To generate an SSH key pair, we'll use the `ssh-keygen` command. This command will prompt you to save the key pair in a file, and you can choose to use the default location `~/.ssh/kube_key`.

```
ssh-keygen
```

This will generate a private key (`kube_key`) and a public key (`kube_key.pub`) in the `~/.ssh` directory.

Sending the Public Key to the Cloud

Next, we need to send the public key to the cloud instance. We'll use the `ssh` command to connect to the instance and append the public key to the `authorized_keys` file.

Executed on your local machine

```
bash cat ~/.ssh/kube_key.pub | ssh root@104.225.219.54 "mkdir -p ~/.ssh && touch ~/.ssh/authorized_keys && chmod -R go= ~/.ssh && cat >
```

Let's break down this command:

- `cat ~/.ssh/kube_key.pub` reads the contents of the public key file.
- `ssh root@104.225.219.54` connects to the cloud instance as the `root` user.
- The command inside the quotes creates the `.ssh` directory and `authorized_keys` file if they don't exist, sets the correct permissions, and appends the public key to the `authorized_keys` file.

Note this will prompt you for the password for root.

Configuring the SSH Client (local Machine)

To make it easier to connect to the cloud instance, we'll add an entry to the SSH client configuration file (`~/.ssh/config`). Again this is on your local machine.
`Host kube_key HostName 104.225.219.54 IdentityFile ~/.ssh/kube_key User root`

This configuration tells the SSH client to:

- Use the `kube_key` alias to connect to the instance.
- Connect to the instance at `104.225.219.54` .
- Use the private key `~/.ssh/kube_key` for authentication.
- Log in as the `root` user.

With these steps, you should now be able to connect to your cloud instance using SSH without entering a password.

Now you can login with the following:

```
ssh root@104.225.219.54
```

Setting up Github

```
ssh-keygen -t rsa -b 4096 -C "david@persistentdesigns.com"
eval "$(ssh-agent -s)"
ssh-add ~/.ssh/id_rsa

cat ~/.ssh/id_rsa.pub
# take public part of key and place it in github

# Test it out
ssh -T git@github.com
```

```
root@kubernetes:~# ssh -T git@github.com
The authenticity of host 'github.com (140.82.112.3)' can't be established.
ED25519 key fingerprint is SHA256:+DiY3wvvVxxxxxxxsF/zLDAxxxxxHdkr4UvCOqU.
This key is not known by any other names.
Are you sure you want to continue connecting (yes/no/[fingerprint])? yes
Warning: Permanently added 'github.com' (ED25519) to the list of known hosts.
Hi davidsells! You've successfully authenticated, but GitHub does not provide shell access.
```

Update apt and get git

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

Get Repository With DevOps Scripts

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

If you run into difficulties you'll find a wonderful write up at Digital Ocean.

Setting up ZSH and OhMyZsh

In this article, we will explore the initial setup required for a new cloud instance of Ubuntu, focusing on configuring ZSH as our shell of choice. ZSH provides many benefits over traditional shells like Bash, including improved tab completion, themes, and plugins.

`zsh_setup.sh`

This script sets up the foundation for our ZSH environment. Let's break it down:

```
#!/bin/bash
# Update the package list to ensure we have the latest package information
apt update
# Install ZSH using apt
apt install -y zsh
# Set ZSH as the default shell
chsh -s /usr/bin/zsh
# Verify that ZSH is installed correctly
which zsh
# Install git, a essential tool for any developer
apt install -y git
# Install curl, which we'll use to download the Oh My ZSH installation script
apt install -y curl
# Download and install Oh My ZSH, a popular ZSH framework
sh -c "$(curl -fsSL https://raw.github.com/ohmyzsh/ohmyzsh/master/tools/install.sh)"
```

This script does the following:

- Updates the package list using `apt update`
- Installs ZSH using `apt install -y zsh`
- Sets ZSH as the default shell using `chsh -s /usr/bin/zsh`
- Verifies that ZSH is installed correctly using `which zsh`
- Installs `git` and `curl` using `apt install -y`
- Installs Oh My ZSH using a script downloaded from GitHub

`zsh_extra.sh`

This script configures some additional ZSH settings and plugins. Let's dive in:

```
#!/bin/bash

# Append the following lines to the ~/.zshrc file
cat >> ~/.zshrc <<EOF

# Define a custom function called hist() that allows us to filter the command history
hist() {
  theFilter="history "
  for var in "$@"
  do
    theFilter+="| grep $var"
  done
  eval $theFilter
}

# Enable vi-mode keybindings
bindkey -v

# Set the ZSH theme to "jonathan" (commented out)
# ZSH_THEME="jonathan"

# Enable the git and z plugins
plugins=(git z)

# Set the default editor to vim
export VISUAL=vim
export EDITOR="$VISUAL"

# Define aliases for microk8s kubectl commands
alias ku='microk8s kubectl'
alias kubectl='microk8s kubectl'

EOF
```

This script does the following:

- Defines a custom function `hist()` that allows us to filter the command history using `grep`
- Enables vi-mode keybindings using `bindkey -v`
- Sets the ZSH theme to "jonathan" (although this line is commented out)
- Enables the `git` and `z` plugins

- Sets the default editor to `vim` using `export VISUAL=vim` and `export EDITOR="$VISUAL"`
- Defines aliases for `microk8s kubectl` commands using `alias ku='microk8s kubectl'` and `alias kubectl='microk8s kubectl'`

By running these two scripts, we've set up a solid foundation for our ZSH environment, including installing ZSH, Oh My ZSH, and configuring some useful plugins and settings.

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) [default=zfs]:
Create a new ZFS pool? (yes/no) [default=yes]:
Would you like to use an existing empty block device (e.g. a disk or partition)? (yes/no) [default=no]:
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”) [default=auto]:
What IPv6 address should be used? (CIDR subnet notation, “auto” or “none”) [default=auto]:
Would you like the LXD server to be available over the network? (yes/no) [default=no]: yes
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) [default=yes]:
Would you like a YAML "lxd init" preseed to be printed? (yes/no) [default=no]:
```

After configuring these options, LXD will be initialized and ready for use. In the next section, we'll cover the installation of MicroK8s, which will use LXD as its container runtime.

Initial Setup for Ubuntu Cloud Instance with ZFS and MicroK8s

Installing ZFS Tools

To manage ZFS (Zettabyte File System), we need to install the necessary tools. This includes `zfs` and `zpool`.

```
sudo apt install zfsutils-linux -y
```

This command installs the `zfsutils-linux` package, which provides the tools necessary for managing ZFS file systems.

Setup Profile for Creating Containers

```
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/microk8s-zfs.profile -O microk8s.profile
```

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`.

```
lxc profile edit microk8s_profile
```

Creating a Container with MicroK8s

Now that we have our custom profile, we can create a new container optimized for MicroK8s:

```
lxc launch -p default -p microk8s_profile ubuntu:20.04 microk8scontainer
```

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 `microk8scontainer` 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.

Docker Install?

Docker Install

We are switching from manual to UI for creating the containers --- New Text Required

Installing MicroK8s inside the Container

Setting up the LXD Container

Microk8s Setup

In this section, we'll cover the setup of MicroK8s, a lightweight, easy-to-use Kubernetes distribution. MicroK8s is perfect for development, testing, and CI/CD pipelines.

Important note before installation

Note when we get to the metallb a range of IPs are required. 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.

```
Master  10.196.148.157  10.196.148.101-10.196.148.120
Worker1 10.196.148.239  10.196.148.125-10.196.148.135
Worker2 10.196.148.155  10.196.148.140-10.196.148.150
```

Microk8s Setup

microk8s_setup.sh

```
#!/bin/zsh

systemctl enable iscsid
swapoff -a

apt install -y zfsutils-linux

snap install microk8s --classic
microk8s enable community
microk8s enable rbac
microk8s enable openebs
microk8s enable metallb
```

Restart Protection with AppArmor

microk8s*restart*protection.sh

```
#!/bin/zsh

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.

By running these two scripts, we've set up MicroK8s on our Ubuntu instance, including enabling various features and configuring restart protection with AppArmor.

New CloudFlare

The new method requires additional addons enabled:

Intersting supporting article for Ingress

```
microk8s enable ingress
```

Deploy the Ingress Service

```
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

```
piVersion: 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-linux-amd64.deb && sudo dpkg -i cloudflared-lin

cloudflared tunnel login

cloudflared tunnel create kubernetes

vi config.yaml

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

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

ingress:

- hostname: kubernetes.davidsells.today

  service: http://10.203.176.200:80

- service: http_status:404

cloudflared tunnel route dns kubernetes kubernetes.davidsells.today

cloudflared tunnel run kubernetes

curl http://10.203.176.200:80

vi config.yaml

cloudflared tunnel run kubernetes

cloudflared service install

systemctl start cloudflared
```

K9 Installation

In this section, we'll cover the installation of K9s, a popular Kubernetes CLI tool, on a fresh Ubuntu cloud instance.

k9s_setup.sh

```
#!/bin/zsh

# Step 1: Save the current Kubernetes configuration to a file
ku config view --raw > ~/.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
chown root:root k9s_linux_amd64.deb

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

Let's break down what each step does:

Step 1: `ku config view --raw > ~/.kube/config` - This command saves the current Kubernetes configuration to a file named `~/.kube/config`. This file is used by K9s to connect to your Kubernetes cluster.

Step 2: `chmod 600 ~/.kube/config` - This command sets the permissions for the `~/.kube/config` file to `rw-------`, which means the owner has read and write permissions, but the group and others have no access. This is a security best practice to prevent unauthorized access to your Kubernetes configuration.

Step 3: `wget https://github.com/derailed/k9s/releases/download/v0.32.5/k9s_linux_amd64.deb` - This command downloads the K9s installation package from the official GitHub repository. The version used here is `v0.32.5`, but you can adjust this to the latest version available.

Step 4: `chown root:root k9s_linux_amd64.deb` - This command changes the ownership of the downloaded package to the `root` user and group. This is necessary because the `apt` package manager requires root privileges to install packages.

Step 5: `apt install -y ./k9s_linux_amd64.deb` - This command installs K9s using the downloaded package. The `-y` flag assumes "yes" to all prompts, allowing the installation to proceed without user intervention.

After running this script, you should have K9s installed on your Ubuntu cloud instance, and you can start using it to manage your Kubernetes cluster.

OpenEBS ZFS Operator Installation

```
microk8s enable openebs
```

```
ku get sc
```

```
NAME                      PROVISIONER           RECLAIMPOLICY   VOLUMEBINDINGMODE      ALLOWVOLUMEEXPANSION   AGE
openebs-device            openebs.io/local      Delete          WaitForFirstConsumer   false                  76s
openebs-hostpath          openebs.io/local      Delete          WaitForFirstConsumer   false                  76s
openebs-jiva-csi-default  jiva.csi.openebs.io   Delete          Immediate              true                   76s
```

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/zsh
microk8s kubectl apply -f https://openebs.github.io/charts/zfs-operator.yaml
```

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": {"annotations":{"storageclass.kubernetes.io/is-default-class":"
microk8s kubectl patch storageclass openebs-zfspv -p '{"metadata": {"annotations":{"storageclass.kubernetes.io/is-default-class":"true
```

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.

Create ZFS Pool for Storage Class

We need to allocate storage space for our Kubernetes cluster. We'll create a ZFS pool and a dataset named `kube_zfs/child` with a quota of 10GB.

Script: `zfs create -o quota=10G kube_zfs/child`

What's happening:

- `zfs create` creates a new ZFS dataset or filesystem.
- `-o` is an option to specify additional properties for the dataset.
- `quota=10G` sets a quota of 10GB for the dataset, limiting the amount of space it can occupy.
- `kube_zfs/child` is the name of the dataset, which will be a child of the root data pool.

Storage Class

Next, we'll define a storage class YAML file that will be used to provision dynamic volumes for our Kubernetes cluster.

Script:

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

What's happening:

- This YAML file defines a storage class named `openebs-zfspv`.
- `apiVersion` and `kind` specify that this is a storage class definition.
- `metadata` section provides metadata for the storage class.
- `parameters` section specifies additional properties for the storage class:
  - `recordsize`: sets the record size to 4KB.
  - `compression`: disables compression.
  - `dedup`: disables deduplication.
  - `fstype`: specifies that the filesystem type is ZFS.
  - `poolname`: references the ZFS pool created earlier (`kube_zfs/child`).
- `provisioner`: specifies the provisioner to use for dynamic volume provisioning (`zfs.csi.openebs.io`).

By creating this storage class, we'll be able to dynamically provision ZFS-based persistent volumes for our Kubernetes cluster. In the next post, we'll cover the deployment of the Kubernetes cluster itself.

Setting up Persistent Storage in Kubernetes

In this section, we'll cover the initial setup required to deploy persistent storage in a new cloud instance of Ubuntu using Kubernetes.

Creating a Persistent Volume Claim (PVC)

A Persistent Volume Claim (PVC) is a request for storage resources in a Kubernetes cluster. It's a way to request a certain amount of storage from the cluster, and the cluster will dynamically provision the storage based on the request.

Here's the YAML script to create a PVC:

```
kind: PersistentVolumeClaim
apiVersion: v1
metadata:
  name: my-pv-claim
  namespace: nginx
spec:
  accessModes:
    - ReadWriteOnce
  resources:
    requests:
      storage: 1Gi
```

Let's break down what's happening in this script:

- `kind: PersistentVolumeClaim`: This specifies that we're creating a Persistent Volume Claim.
- `apiVersion: v1`: This specifies the API version of the Kubernetes API.
- `metadata`: This section provides metadata about the PVC, such as its name and namespace.
- `name: my-pv-claim`: This is the name of the PVC.
- `namespace: nginx`: This specifies the namespace where the PVC will be created.
- `spec`: This section specifies the desired state of the PVC.

- `accessModes` : This specifies the access mode for the PVC. In this case, we're using `ReadWriteOnce` , which means the volume can be mounted as read-write by a single node.
- `resources` : This section specifies the resources required by the PVC.
- `requests` : This specifies the amount of storage requested. In this case, we're requesting 1Gi (1 gigabyte) of storage.

By creating this PVC, we're requesting 1Gi of storage from the cluster, and the cluster will dynamically provision the storage based on this request. This storage will be used by our application to persist data even if the pod is restarted or deleted.

In the next section, we'll cover how to create a Persistent Volume (PV) to fulfill this PVC request.

Nginx Deployment - PVC Test

In this section, we will create a Kubernetes deployment that uses a Persistent Volume Claim (PVC) to store data. This is a crucial step in ensuring that our data is persisted even if our pods are restarted or deleted.

Deployment YAML

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: nginx-pv-deployment
  namespace: nginx
spec:
  replicas: 1
  selector:
    matchLabels:
      app: nginx
  template:
    metadata:
      labels:
        app: nginx
    spec:
      volumes:
        - name: my-pv-storage
          persistentVolumeClaim:
            claimName: my-pv-claim
      containers:
        - name: nginx-pv-container
          image: nginx
          ports:
            - containerPort: 80
              name: "http-server"
          volumeMounts:
            - mountPath: "/usr/share/nginx/html"
              name: my-pv-storage
```

Let's break down this YAML file:

- We define a Deployment named `nginx-pv-deployment` in the `nginx` namespace.
- We specify that we want one replica of this deployment.
- We define a `selector` that matches pods with the label `app: nginx` .
- We define a `template` that specifies the pod's metadata and spec.
- In the `spec` , we define a `volume` named `my-pv-storage` that uses a Persistent Volume Claim (PVC) named `my-pv-claim` .
- We define a container named `nginx-pv-container` that uses the `nginx` image and exposes port 80.
- We mount the `my-pv-storage` volume to the container at the path `/usr/share/nginx/html` .

What's happening here?

- We are creating a deployment that uses a PVC to store data. This ensures that our data is persisted even if our pods are restarted or deleted.
- We are using a single replica of this deployment, which means that only one pod will be created.
- We are using the `nginx` image and exposing port 80, which means that our pod will serve a web server.

Nginx Deployment - MetalLB Test

In this section, we will create a Kubernetes deployment that uses MetalLB to expose a LoadBalancer service. This is a crucial step in exposing our application to the outside world.

Create Namespace

```
kubectl create namespace lb-test
```

We create a new namespace named `lb-test` using the `kubectl` command.

Deployment YAML

```
apiVersion: v1
kind: Namespace
metadata:
  name: lb-test
---
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:
  name: metallb-test
  namespace: lb-test
spec:
  selector:
    app: metallb-test
  ports:
  - name: http
    port: 80
    targetPort: 80
  type: LoadBalancer
```

Let's break down this YAML file:

- We define a Namespace named `lb-test`.
- We define a Deployment named `metallb-test` in the `lb-test` namespace.
- We specify that we want two replicas of this deployment.
- We define a `selector` that matches pods with the label `app: metallb-test`.
- We define a `template` that specifies the pod's metadata and spec.
- In the `spec`, we define a container named `metallb-test` that uses the `nginx:latest` image and exposes port 80.
- We define a Service named `metallb-test` that selects pods with the label `app: metallb-test`.
- We specify that the Service should expose port 80 and use the `LoadBalancer` type.

What's happening here?

- We are creating a deployment that uses MetalLB to expose a LoadBalancer service.
- We are creating two replicas of this deployment, which means that two pods will be created.
- We are using the `nginx:latest` image and exposing port 80, which means that our pods will serve a web server.
- We are creating a Service that selects pods with the label `app: metallb-test` and exposes port 80 using the `LoadBalancer` type.
- MetalLB will automatically assign an external IP and port to the Service, allowing us to access our application from outside the cluster.

Setting up Ollama and Open-WebUI on Ubuntu with Kubernetes

In this blog post, we'll go through the initial setup required to deploy Ollama and Open-WebUI on a new cloud instance of Ubuntu using Kubernetes.

Create ZFS Pool for llm Storage Class

The first step is to create a ZFS pool for the `llm` storage class. ZFS (Zettabyte File System) is a file system designed for high-capacity storage and provides features like data compression, deduplication, and snapshots.

```
zfs create -o quota=10G kube_zfs/llm
```

This command creates a new ZFS pool named `kube_zfs/llm` with a quota of 10GB.

Storage Class - llm

Next, we define a storage class for the `llm` pool. A storage class is a way to define a class of storage that can be used to provision persistent volumes.

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

This storage class is named `llm-zfspv` and uses the `zfs.csi.openebs.io` provisioner to create persistent volumes. The `parameters` section defines the record size, compression, deduplication, and file system type for the storage class.

Re-align ZFS Pools

We need to update the default storage class annotations to ensure that our new storage class is used by default.

```
microk8s kubectl patch storageclass openebs-zfspv     -p '{"metadata": {"annotations":{"storageclass.kubernetes.io/is-default-class":"
microk8s  kubectl patch storageclass llm-zfspv -p '{"metadata": {"annotations":{"storageclass.kubernetes.io/is-default-class":"true"}}
```

These commands update the annotations for the `openebs-zfspv` and `llm-zfspv` storage classes.

Create Namespace

Next, we create a new namespace for our deployments.

```
kubectl create namespace llm-ns
```

This command creates a new namespace named `llm-ns` .

Persistent Volume Claim - llm

We define a persistent volume claim (PVC) to request storage resources for our deployments.

```
kind: PersistentVolumeClaim
apiVersion: v1
metadata:
  name: llm-pv-claim
  namespace: llm-ns
spec:
  accessModes:
    - ReadWriteOnce
  resources:
    requests:
      storage: 5Gi
```

This PVC is named `llm-pv-claim` and requests 5Gi of storage with read-write access.

Ollama Deployment

Now, we define a deployment for Ollama.

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: ollama
  namespace: llm-ns
spec:
  replicas: 1
  selector:
    matchLabels:
      app: ollama
  template:
    metadata:
      labels:
        app: ollama
    spec:
      containers:
      - name: ollama
        image: ollama/ollama:latest
        ports:
        - containerPort: 11434
        volumeMounts:
        - name: ollama-data
          mountPath: /root/.ollama
      volumes:
      - name: ollama-data
        persistentVolumeClaim:
          claimName: new-ollama-pvc
      priorityClassName: system-node-critical
      strategy:
        type: Recreate
---
apiVersion: v1
kind: Service
metadata:
  name: ollama-service
  namespace: llm-ns
spec:
  selector:
    app: ollama
  ports:
  - name: http
    port: 11434
    targetPort: 11434
    protocol: TCP
  type: ClusterIP
---
apiVersion: v1
kind: Service
metadata:
  name: ollama-service-lb
  namespace: llm-ns
spec:
  selector:
    app: ollama
  ports:
  - name: http
    port: 11434
    targetPort: 11434
    protocol: TCP
  type: LoadBalancer
```

This deployment defines a single replica of the Ollama container, which uses a persistent volume claim named `new-ollama-pvc`. We also define two services: `ollama-service` and `ollama-service-lb`, which expose the Ollama container on port 11434.

PVC for Open-WebUI

```
kind: PersistentVolumeClaim
apiVersion: v1
metadata:
  name: owui-pvc-claim
  namespace: llm-ns
spec:
  storageClassName: llm-zfspv
  accessModes:
    - ReadWriteOnce
  resources:
    requests:
      storage: 5Gi
```

Deploy Open-WebUI

Finally, we define a deployment for Open-WebUI.

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: open-webui
  namespace: llm-ns
spec:
  replicas: 1
  selector:
    matchLabels:
      app: open-webui
  template:
    metadata:
      labels:
        app: open-webui
    spec:
      containers:
      - name: open-webui
        image: ghcr.io/open-webui/open-webui:main
        ports:
        - containerPort: 8080
        env:
        - name: OLLAMA_BASE_URL
          value: http://ollama-service.default.svc.cluster.local:11434
        volumeMounts:
        - name: open-webui-data
          mountPath: /app/backend/data
      volumes:
      - name: open-webui-data
        persistentVolumeClaim:
          claimName: llm-pvc-claim

---
apiVersion: v1
kind: Service
metadata:
  name: open-webui-lb
  namespace: llm-ns
spec:
  selector:
    app: open-webui
  ports:
  - name: http
    port: 3000
    targetPort: 8080
    protocol: TCP
  type: LoadBalancer
```

This deployment defines a single replica of the Open-WebUI container, which uses a persistent volume claim named `llm-pvc-claim`. We also define a service named `open-webui-lb`, which exposes the Open-WebUI container on port 3000.

That's it! With these scripts, we've set up a new ZFS pool, storage class, namespace, and deployments for Ollama and Open-WebUI using Kubernetes.

Setting up LiteLLM on a New Ubuntu Cloud Instance

In this blog post, we will go through the initial setup required to deploy LiteLLM, an application that integrates with OpenWebUI to utilize external Large Language Models (LLMs) like GROQ.

Configuring LiteLLM

The first step is to create a configuration file `config.yaml` that defines the models hosted on GROQ's service. This file contains three models: `llama3-70b`, `llama3-8b`, and `llama3.1-8b-instant`.

```
model_list:
  - model_name: 'llama3-70b'
    litellm_params:
      model: 'groq/llama3-70b-8192'
      api_key: gsk_l1Qxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx
  - model_name: 'llama3-8b'
    litellm_params:
      model: 'groq/llama3-8b-8192'
      api_key: gsk_l1Qxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx
  - model_name: 'llama3.1-8b-instant'
    litellm_params:
      model: 'groq/llama3.1-8b-instant'
      api_key: gsk_l1Qxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx
```

This configuration file is used to create a ConfigMap, a Kubernetes object that stores configuration data as key-value pairs.

Creating a ConfigMap

To create a ConfigMap, we use the following command: `kubectl create configmap litellm-config --from-file=config.yaml -n llm-ns` This command creates a ConfigMap named `litellm-config` in the `llm-ns` namespace, using the `config.yaml` file as the source of the configuration data.

Deploying LiteLLM

Next, we define a Deployment YAML file `deploy.yaml` that deploys LiteLLM and a Service to make it available to other pods.

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: litellm-deployment
  namespace: llm-ns
spec:
  replicas: 1
  selector:
    matchLabels:
      app: litellm
  template:
    metadata:
      labels:
        app: litellm
    spec:
      containers:
      - name: litellm
        image: ghcr.io/berriai/litellm:main-latest
        ports:
        - containerPort: 4000
        env:
        - name: LITELLM_MASTER_KEY
          value: your_secret_key_xx
        - name: GROQ_API_KEY
          valueFrom:
            secretKeyRef:
              name: groq-api-key
              key: GROQ_API_KEY
        volumeMounts:
        - name: config-volume
          mountPath: /app/config.yaml
          subPath: config.yaml
        command: ["/usr/local/bin/litellm"]
        args: ["--config", "/app/config.yaml", "--port", "4000"]
      volumes:
      - name: config-volume
        configMap:
          name: litellm-config

---
apiVersion: v1
kind: Service
metadata:
  name: litellm-service
  namespace: llm-ns
spec:
  selector:
    app: litellm
  ports:
  - name: http
    port: 4000
    targetPort: 4000
  type: ClusterIP
```

This Deployment YAML file defines:

- A Deployment named `litellm-deployment` in the `llm-ns` namespace, with one replica.
- A container named `litellm` using the `ghcr.io/berriai/litellm:main-latest` image.
- Environment variables `LITELLM_MASTER_KEY` and `GROQ_API_KEY` are set.
- A volume mount is created to mount the ConfigMap `litellm-config` as a file at `/app/config.yaml`.
- The LiteLLM application is started with the command `"/usr/local/bin/litellm"` and arguments `["--config", "/app/config.yaml", "--port", "4000"]`.
- A Service named `litellm-service` is defined to expose the LiteLLM application on port 4000.

Connecting to LiteLLM

To connect to LiteLLM from OpenWebUI, use the URL `http://litellm-service:4000`.

That's it! With these steps, we have successfully set up LiteLLM on a new Ubuntu cloud instance.