**Process 1: Standard Docker Workflow**

1. **Design** → The application is architected, requirements are analyzed, and high-level designs are prepared.
2. **Code** → The application code is developed based on the design.
3. **Build** → The source code is compiled or transformed into an executable format (e.g., building a Java app into a .jar or a Node.js app into JavaScript code).
4. **Package** → The application is packaged with its dependencies (e.g., libraries, modules) into a single format (usually binaries or libraries).
5. **Dockerfile** → A Dockerfile is created to define how the application and its dependencies are packaged into a Docker image. The Dockerfile serves as a blueprint for building the container.
6. **Docker Images** → The Docker image is a read-only template that includes everything the application needs to run: code, runtime, libraries, and configuration files. It is created from the Dockerfile.
7. **Container** → A container is launched from the image and represents a runtime instance of the Docker image. The container runs the actual application on any platform that supports Docker (single host or orchestrated environments like Kubernetes).

**Process 2: Custom Docker Image Workflow (Optimized)**

1. **Design** → The application's high-level architecture is designed, and requirements are set.
2. **Code** → The application code is written and completed.
3. **Dockerfile** → A Dockerfile is written to automate the image-building process. It specifies:
   * Base image (e.g., FROM ubuntu or FROM node:14)
   * Application source code location
   * Dependencies and configurations needed to run the application
   * Instructions for building and running the app inside a container
4. **Docker Build** → The docker build command is executed to create a custom image based on the Dockerfile. The resulting image contains the application, dependencies, and configurations.
5. **Docker Images** → The Docker image is stored as a read-only template, ready to be used to create containers. This template can be reused or shared across different environments.
6. **Container** → Containers are created from the custom Docker image. These containers run the application in isolated environments on a single host or within a Kubernetes cluster (or another orchestrator). The container can be scaled and managed within Kubernetes, providing enhanced scalability and resilience.

**Key Differences Between the Two Processes**

* **Process 1** involves building and packaging the application first and then creating a Dockerfile to package it into a container image.
* **Process 2** skips some traditional packaging steps and directly focuses on creating the application’s Docker image using the Dockerfile, which is then used to generate containers. This flow is generally more streamlined and suitable for CI/CD pipelines and cloud-native environments like Kubernetes.

**Use Cases:**

* **Process 1** is often used when transitioning legacy applications into containers or where traditional build and package steps are necessary before Dockerization.
* **Process 2** is ideal for cloud-native applications where Docker is integral from the start, and the build process is optimized for containerization from the ground up. This approach is also well-suited for microservices architectures and Kubernetes.

**Lab Guide: Setting Up a Kubernetes Cluster Using Kubeadm with Cri-Dockerd**

This guide will help you set up a Kubernetes cluster on AWS using three Ubuntu instances (1 master, 2 workers). The instances will use kubeadm for cluster setup and cri-dockerd for Docker as the container runtime.

**Prerequisites**

* **Region**: California (us-west-1)
* **Instance Type**: t2.medium
* **AMI**: Ubuntu 20.04 LTS
* **Storage**: 20 GB
* **Security Group**: raman-sg (allow SSH and necessary Kubernetes ports)

Ensure the following ports are open in your raman-sg security group:

* + **SSH**: TCP 22
  + **Kubernetes API Server**: TCP 6443
  + **Node Ports**: TCP 30000-32767
  + **Etcd**: TCP 2379-2380 (for master communication)
  + **Kubelet API**: TCP 10250, 10255

**Step 1: Launch AWS EC2 Instances**

1. Go to the AWS Console and navigate to **EC2**.
2. Launch 3 instances in the **us-west-1** region (California):
   * **Instance Type**: t2.medium
   * **AMI**: Ubuntu 20.04 LTS
   * **Storage**: 20 GB
   * **Security Group**: raman-sg
3. Name your instances:
   * **master**
   * **worker1**
   * **worker2**
4. Connect to each instance using SSH:

bash

Copy code

ssh -i <your\_key.pem> ubuntu@<public\_ip\_of\_instance>

**Step 2: Install Docker and Kubernetes on All Nodes**

**(a) Create an Installation Script**

On each node (master, worker1, worker2), create a script to install Docker, Kubernetes components, and cri-dockerd:

1. Create a script file:

bash

Copy code

vi script.sh

1. Paste the following script into the file:

bash

Copy code

#!/bin/bash

# Update package lists and install Docker

sudo apt update -y

sudo apt install docker.io -y

# Set up Kubernetes repository and install kubelet, kubeadm, and kubectl

sudo mkdir -p -m 755 /etc/apt/keyrings

curl -fsSL https://pkgs.k8s.io/core:/stable:/v1.29/deb/Release.key | sudo gpg --dearmor -o /etc/apt/keyrings/kubernetes-apt-keyring.gpg

echo 'deb [signed-by=/etc/apt/keyrings/kubernetes-apt-keyring.gpg] https://pkgs.k8s.io/core:/stable:/v1.29/deb/ /' | sudo tee /etc/apt/sources.list.d/kubernetes.list

sudo apt-get update

sudo apt-get install -y kubelet kubeadm kubectl

# Enable net.bridge.bridge-nf-call-iptables

sudo sysctl net.bridge.bridge-nf-call-iptables=1

# Download and install cri-dockerd

wget https://github.com/Mirantis/cri-dockerd/releases/download/v0.3.14/cri-dockerd-0.3.14.amd64.tgz

tar -xvf cri-dockerd-0.3.14.amd64.tgz

cd cri-dockerd

sudo install -o root -g root -m 0755 cri-dockerd /usr/local/bin/cri-dockerd

# Download and set up cri-dockerd systemd service

cd ..

wget https://github.com/Mirantis/cri-dockerd/archive/refs/tags/v0.3.14.tar.gz

tar -xvf v0.3.14.tar.gz

cd cri-dockerd-0.3.14/

sudo cp packaging/systemd/\* /etc/systemd/system

sudo sed -i -e 's,/usr/bin/cri-dockerd,/usr/local/bin/cri-dockerd,' /etc/systemd/system/cri-docker.service

# Enable and start cri-docker service

sudo systemctl daemon-reload

sudo systemctl enable --now cri-docker.socket

sudo systemctl enable cri-docker

sudo systemctl start cri-docker

sudo systemctl status cri-docker

1. Run the script:

bash

Copy code

chmod +x script.sh

./script.sh

Repeat this on all three nodes (master, worker1, and worker2).

**Step 3: Initialize the Kubernetes Master Node**

**(a) Create a Configuration File on the Master Node**

1. SSH into the **master** node.
2. Create the Kubernetes configuration file:

bash

Copy code

vi config.yaml

1. Add the following content, replacing 172.31.3.203 with the **private IP** of your master node:

yaml

Copy code

apiVersion: kubeadm.k8s.io/v1beta3

kind: InitConfiguration

localAPIEndpoint:

advertiseAddress: 172.31.3.203

bindPort: 6443

nodeRegistration:

criSocket: unix:///var/run/cri-dockerd.sock

---

apiVersion: kubeadm.k8s.io/v1beta3

kind: ClusterConfiguration

networking:

podSubnet: 192.168.0.0/16

**(b) Initialize the Cluster**

1. Run the following command to initialize the Kubernetes cluster:

bash

Copy code

kubeadm init --config=config.yaml >> cluster\_initialized.txt

1. **Save the token and hash** output in cluster\_initialized.txt. This information will be used to join worker nodes to the cluster.

**Step 4: Set Up kubectl on the Master Node**

After initialization, configure kubectl to manage the cluster:

1. Create the .kube directory and copy the admin configuration:

bash

Copy code

mkdir -p $HOME/.kube

sudo cp -i /etc/kubernetes/admin.conf $HOME/.kube/config

sudo chown $(id -u):$(id -g) $HOME/.kube/config

1. Verify that the master node is up and running:

bash

Copy code

kubectl get nodes

**Step 5: Install the Calico Network Plugin**

1. Install the Calico plugin to enable networking between pods:

bash

Copy code

kubectl apply -f https://raw.githubusercontent.com/projectcalico/calico/v3.25.0/manifests/calico.yaml

**Step 6: Join the Worker Nodes to the Cluster**

1. On **worker1** and **worker2**, run the kubeadm join command that was generated during the initialization process. For example:

bash

Copy code

kubeadm join 172.31.3.203:6443 --token 7gy0n6.gycf0eed7x3lihdj \

--discovery-token-ca-cert-hash sha256:6dc79151d4cd2c5e1e05e912f7935cf4cae1df149a2f3a74a62d84ee8bbe3cbf \

--cri-socket unix:///var/run/cri-dockerd.sock

**Step 7: Verify the Cluster**

1. On the master node, check the status of the nodes:

bash

Copy code

kubectl get nodes

You should see the following output indicating that all nodes (master, worker1, worker2) are **Ready**:

css

Copy code

NAME STATUS ROLES AGE VERSION

master Ready control-plane 95m v1.29.9

worker1 Ready <none> 2m2s v1.29.9

worker2 Ready <none> 104s v1.29.9

**Lab Complete**

You now have a fully functioning Kubernetes cluster using kubeadm and Docker with cri-dockerd as the container runtime.

You can now deploy applications, set up services, and explore Kubernetes further.

**Lab Guide: Working with Pods, Namespaces, and Deployments in Kubernetes**

In this lab, you will go through a set of commands to work with Kubernetes pods, namespaces, and containers. This guide assumes you have a running Kubernetes cluster.

**Step 1: Set Up an Alias for kubectl**

1. To simplify working with kubectl, set an alias:

bash

Copy code

alias k=kubectl

1. To make this alias persistent, add it to your ~/.bashrc or ~/.bash\_profile file:

bash

Copy code

echo "alias k=kubectl" >> ~/.bashrc

source ~/.bashrc

**Step 2: Check the Cluster Status**

1. View all nodes in the cluster:

bash

Copy code

k get nodes

1. View all pods running in all namespaces:

bash

Copy code

k get pods -A

1. View all pods with more detailed information (e.g., IP, node name):

bash

Copy code

k get pods -A -o wide

1. View only the pods running in the kube-system namespace:

bash

Copy code

k get pods -n kube-system

**Step 3: Verify Docker Container Status**

1. View running Docker containers on the node (this is optional if you want to check the Docker runtime):

bash

Copy code

docker ps

**Step 4: Deploying Pods in the Default Namespace**

1. Run a simple pod using the **httpd** image:

bash

Copy code

k run ramanfirstpod --image=httpd

1. Verify that the pod is running:

bash

Copy code

k get pods

1. Get detailed information about the pod:

bash

Copy code

k describe pod ramanfirstpod

1. Get the pod's IP and other details using the wide option:

bash

Copy code

k get pods -o wide

1. Access the pod using its IP address (from another pod or within the cluster network):

bash

Copy code

curl <pod\_ip> # replace <pod\_ip> with the actual IP

**Step 5: Deploying Another Pod**

1. Run another pod using the **nginx** image:

bash

Copy code

k run ramanpod2 --image=nginx

1. Verify that the pod is running:

bash

Copy code

k get pods

**Step 6: Create and Work with a Namespace**

1. Create a new namespace called **raman**:

bash

Copy code

k create ns raman

1. Run a pod in the newly created namespace:

bash

Copy code

k run app3 --image=redis -n raman

1. Verify that the pod is running in the **raman** namespace:

bash

Copy code

k get pods -n raman

**Step 7: View Pods Across Namespaces**

1. View all pods across all namespaces:

bash

Copy code

k get pods -A

1. View the pods in the **raman** namespace:

bash

Copy code

k get pods -n raman

**Lab Completed**

You have successfully:

* Created and listed Kubernetes pods.
* Set up a namespace and deployed pods within that namespace.
* Verified pod information using various kubectl commands.

This gives a basic understanding of how to deploy and manage pods in a Kubernetes cluster using kubectl.