Containers

Case Study 9

The Therac-25 went into service in 1983. For several years and thousands of patients there were no problems. On June 3, 1985, a woman was being treated for breast cancer. She had been prescribed 200 Radiation Absorbed Dose (rad) in the form of a 10 MeV electron beam. The patient felt a tremendous heat when the machine powered up. It wasn't known at the time, but she had been burned by somewhere between 10,000 and 20,000 rad. The patient lived, but lost her left breast and the use of her left arm due to the radiation.

Therac-25 relied on software controls to switch between modes, rather than physical hardware. Preceding models used separate circuits to monitor radiation intensity, and hardware interlocks to ensure that spreading magnets were correctly positioned. Using software instead would in theory reduce complexity, and reduce manufacturing costs.

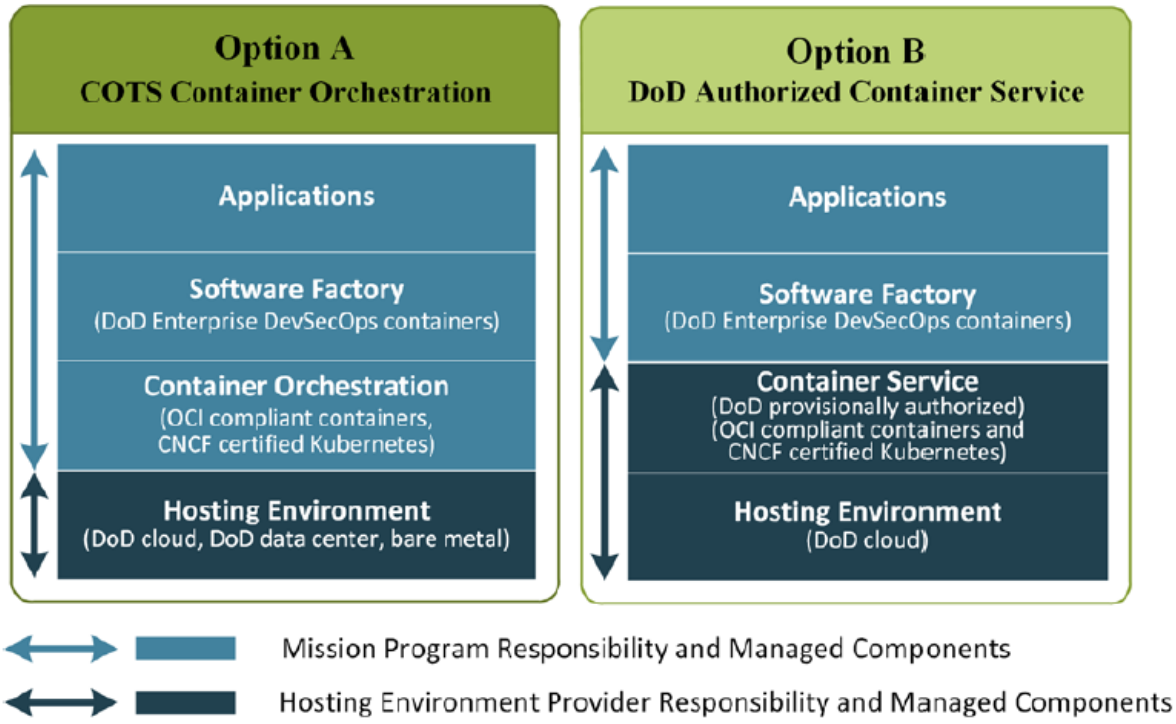
DoD DevSecOps and Containers

Container Deployment

A container manager provides capabilities that check for new versions of containers, deploys the containers to the production environment, and performs post-deployment checkout. The container manager consists of an OCI-compliant container runtime and a CNCF Certified Kubernetes, which is an orchestration tool for managing microservices or containerized applications across a cluster of nodes. The nodes could be bare metal servers or VMs. The container manager may be owned by a mission program or provided by the cloud hosting environment. It simplifies container management tasks, such as instantiation, configuration, scaling, monitoring, and rolling updates. The CNCF Certified Kubernetes interacts with the underlying virtualization manager in the cloud environment to ensure each node's health and performance, and scale it as needed. This scaling includes container scaling within the CNCF Certified Kubernetes cluster, but when running in a cloud, it also includes the ability to auto-scale a number of nodes in a cluster by adding or deleting VMs.

DoD DevSecOps and Containers

In order to support containerized software factory tools, the underlying container orchestration must use CNCF certified Kubernetes and support OCI compliant containers. CNCF-certified Kubernetes orchestrates containers, interacts with underlying hosting environment resources, and coordinates clusters of nodes at scale in development, testing and pre-production in an efficient manner. There are two options for the container orchestration layer as illustrated in Figure 13..



DoD Enterprise DevSecOps Reference Design August 2019

What is a container?

In software development, containers are lightweight, small, deployable instances of applications that contain the minimum resources required by an application or service to run on a host. Abstraction of software from the underlying infrastructure allows teams to easily develop, test and deploy code to a production environment, making them ideal for applications and services hosted on cloud platforms. Containers are running instances of images which are read-only, standalone, executable packages of software that include everything needed to run an application: source code, runtime, system tools and libraries, and runtime settings. Containers can be created, started, stopped, moved or deleted. Each container is defined by its image in addition to any configuration settings applied when the container is created or started.

Wilson, Glenn. DevSecOps: A leader's guide to producing secure software without compromising flow, feedback and continuous improvement (pp. 144-145). Rethink Press. Kindle Edition.

What is a container?

A container is **just a running process** controlled by the host kernel

It is **isolated** from the host and from other processes

There are different containerization technologies available

(Docker, Podman, Singularity, etc.)

Containers

Kumar Rath, Ashwini. Concepts and Practices of DevSecOps: Crack the DevSecOps interviews (English Edition) (p. 89). BPB Publications.

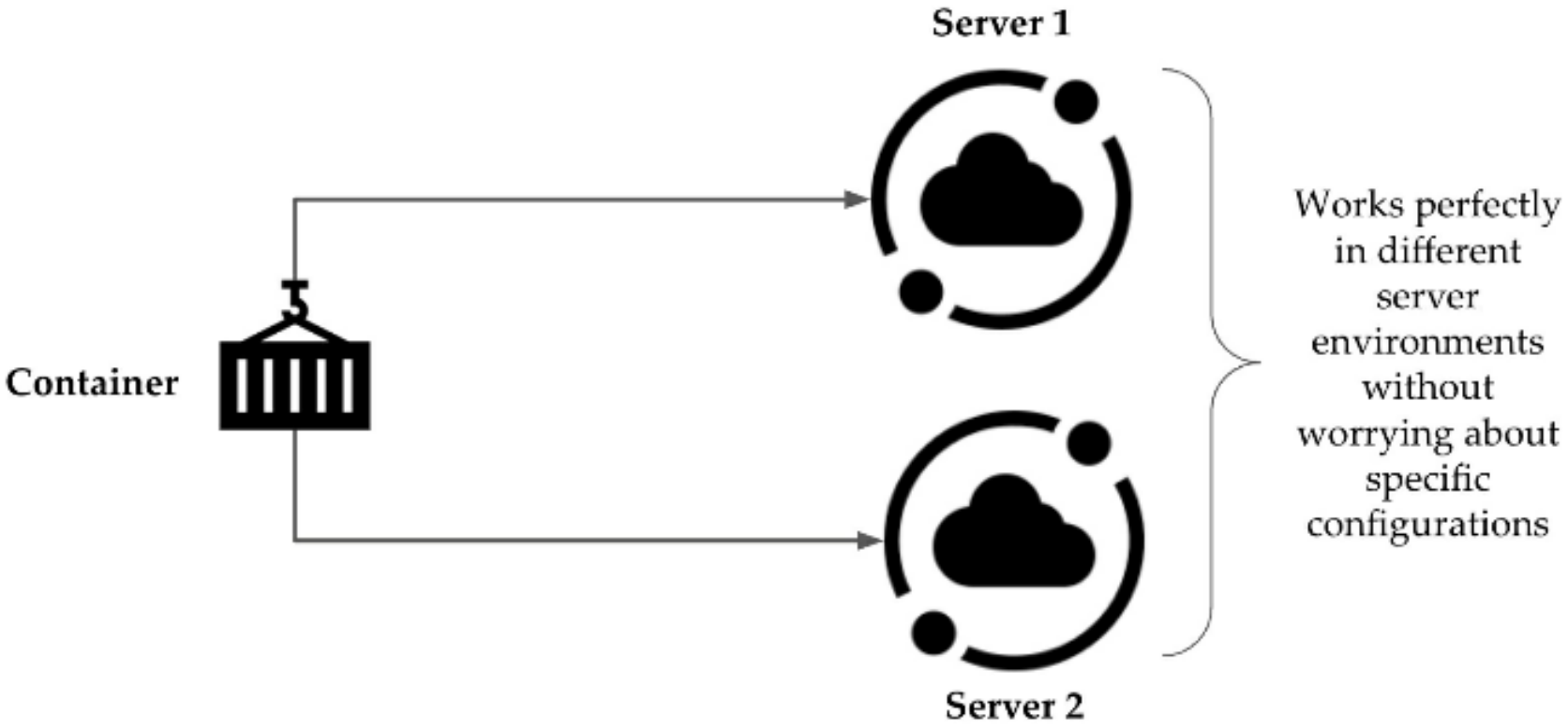


***Figure 4.1**: Easy movement of containers through different server configurations*

Containers

Docker, introduced in 2013, has done more than any other tool to popularize container technology. Docker is an open-source platform designed to automate applications' deployment, scaling, and management by encapsulating them into containers. It uses resource isolation features of the Linux kernel, such as cgroups and kernel namespaces, and a union-capable file system, such as OverlayFS and others, to allow independent containers to run within a single Linux instance.

Kumar Rath, Ashwini. Concepts and Practices of DevSecOps: Crack the DevSecOps interviews (English Edition) (p. 90). BPB Publications.

The original container

A chroot (short for "change root") is a Unix operation that changes the perceived root directory for a running process and its children. This allows a process to operate within a restricted environment, as if its root directory is a different directory on the filesystem. Essentially, it creates a "jail" where the process can only see and interact with files and directories within that specific directory

The chroot command effectively changes the root directory for a process and its descendants. This means that when a process inside the chroot tries to access files or directories, it will only be able to see those within the new root directory.

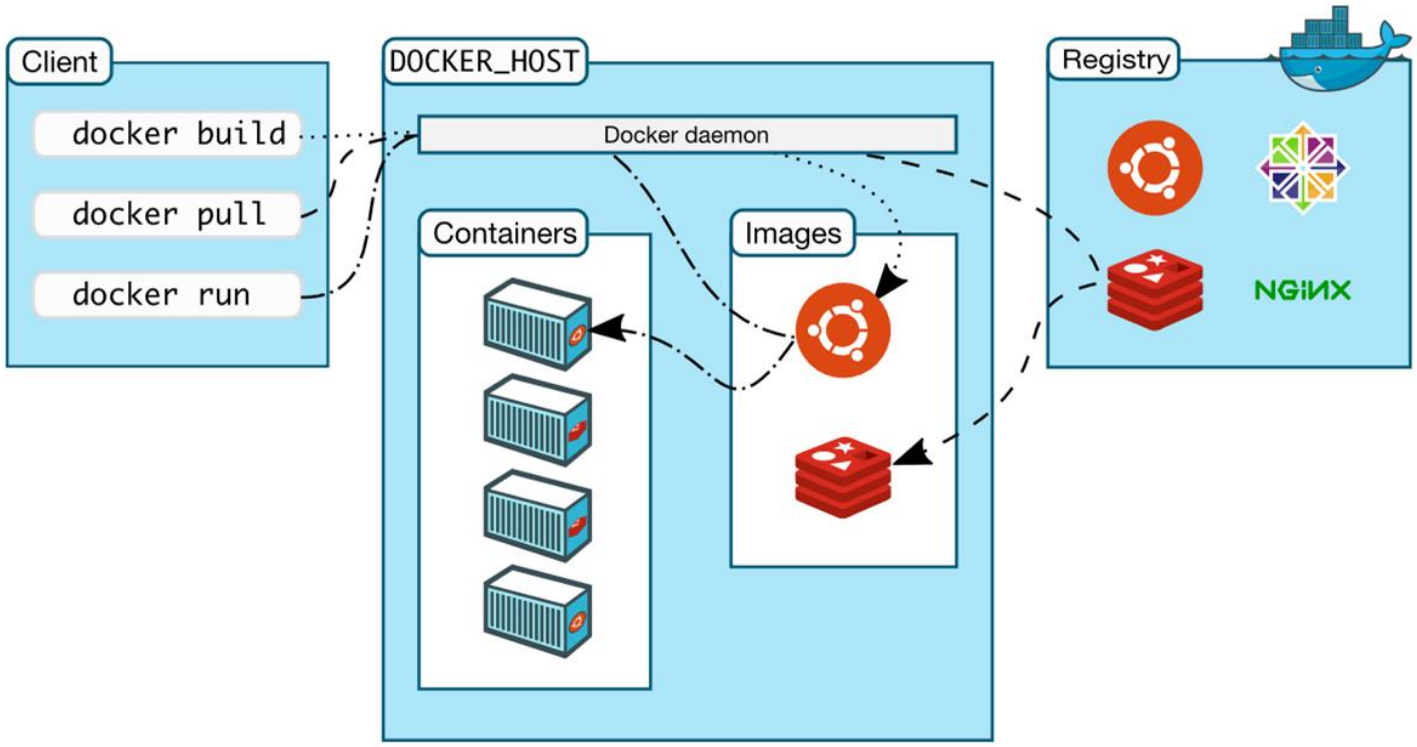
Sandboxing:

The chroot operation creates a virtual filesystem sandbox around the process. This sandbox isolates the process from the rest of the system, limiting its access to sensitive resources and data.

Security and Isolation:

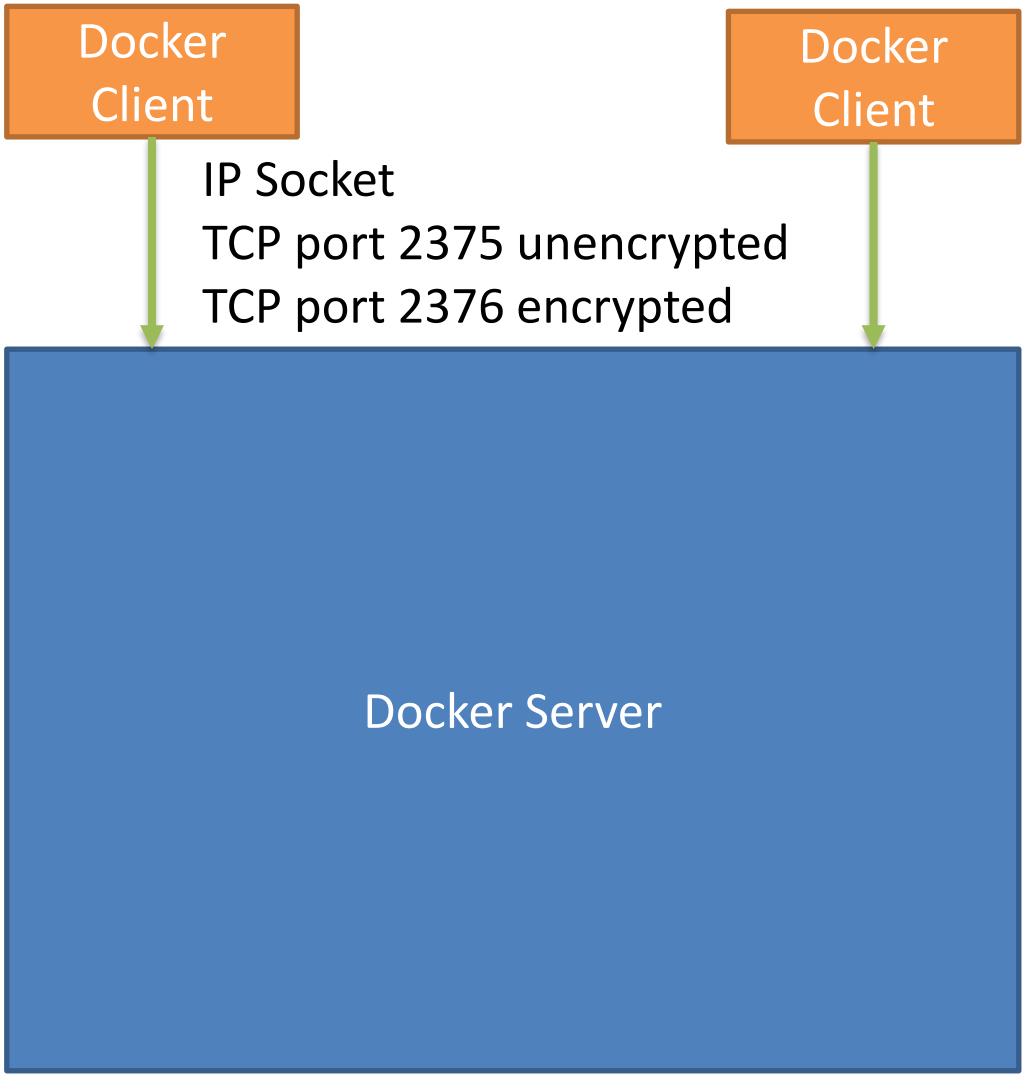
By confining a process to a designated directory, chroot can prevent unauthorized access to system resources and data, facilitating system maintenance, software testing, and running legacy software.

Docker Architecture



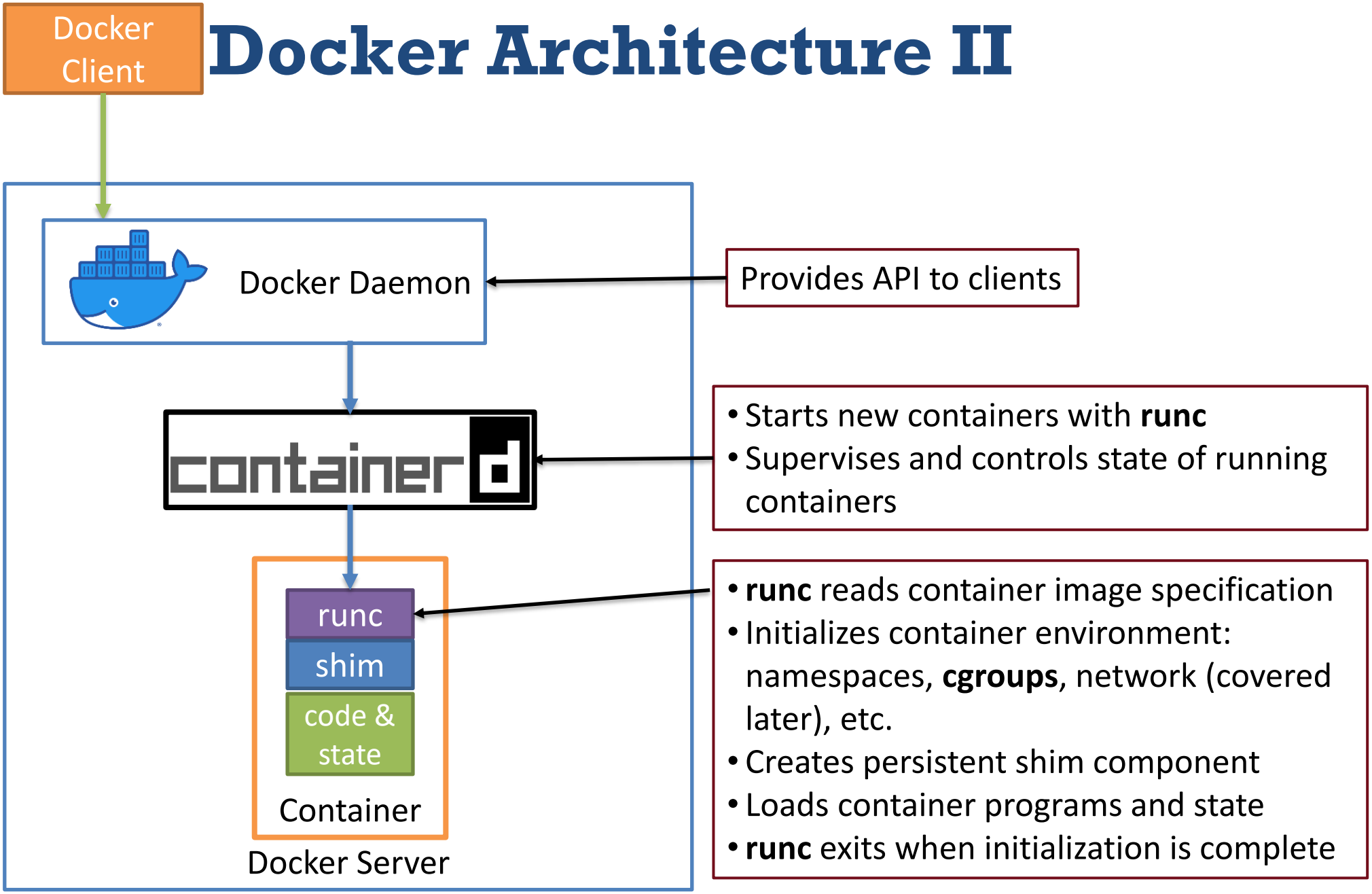
https://docs.docker.com/get-started/overview/

Docker Architecture I

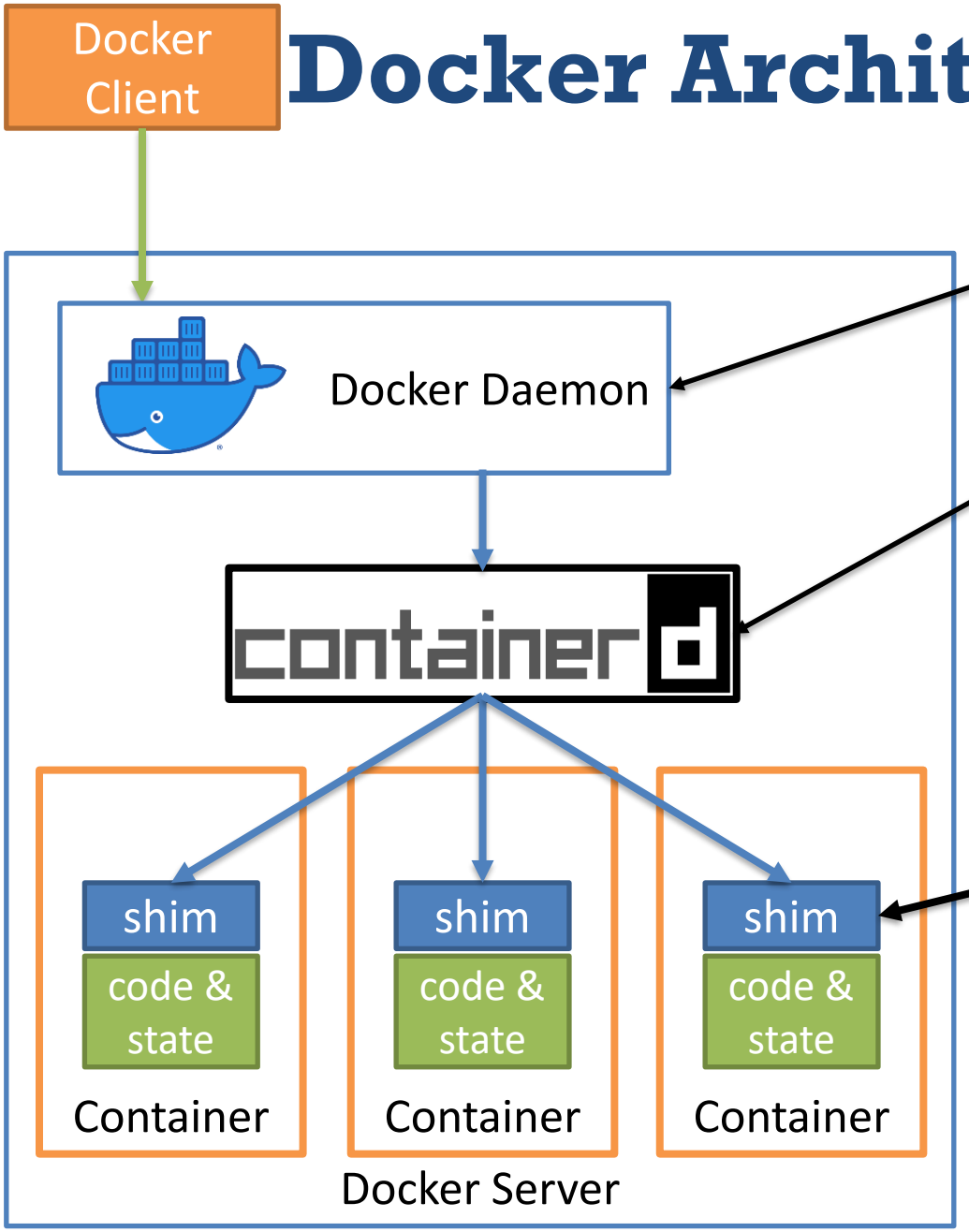


- Client/Server architecture
- Uses IP sockets for communication
- Clients can be on the same machine, or communicate over a network
- Allows connections from multiple concurrent clients!

Docker Architecture II



Docker Architecture III



Provides API to clients

Supervises and controls state of running containers

- **containerd** clones a new **runc** for every container it creates
- Resource intensive to maintain a copy of **runc** for each container
- **shim** process maintains minimal environment to allow container to maintain communication with **containerd**
- This modular architecture allows containers to stay online even if the Daemon is restarted/upgraded

Docker Images

containerd starts a container from a Docker image

One image can launch multiple containers

An image is built from a Dockerfile that specifies the image's attributes, files, commands, etc.

Program	Executable Binary	Process
Dockerfile	Image	Container

Building a Docker Image

Docker images are layered

An image can inherit from a base image (e.g. a minimal Linux distribution, with necessary files and directories)

Layers are additive:

- dnf install –y httpd
- dnf clean all
- Files from the install are kept in the image, but marked deleted in the clean layer

docker build uses a layer cache when building a new version of an existing image

Installing Docker

```
sudo apt-get install docker.io
```

This is the Ubuntu/Debian release package, and installs dependencies as separate, dynamically-linked libraries

Non-root user has access to Docker client

Add **pi** user to **docker** group to access Docker server

Alpine Linux

Alpine Linux is a minimal Linux distribution

Uses the lightweight **musl c** library instead of **glibc**

Has a minimal set of shell utilities

Mounts a minimal set of files and directories to enable interaction via the host Linux kernel

Used as the base derive layer for your image

Docker Commands

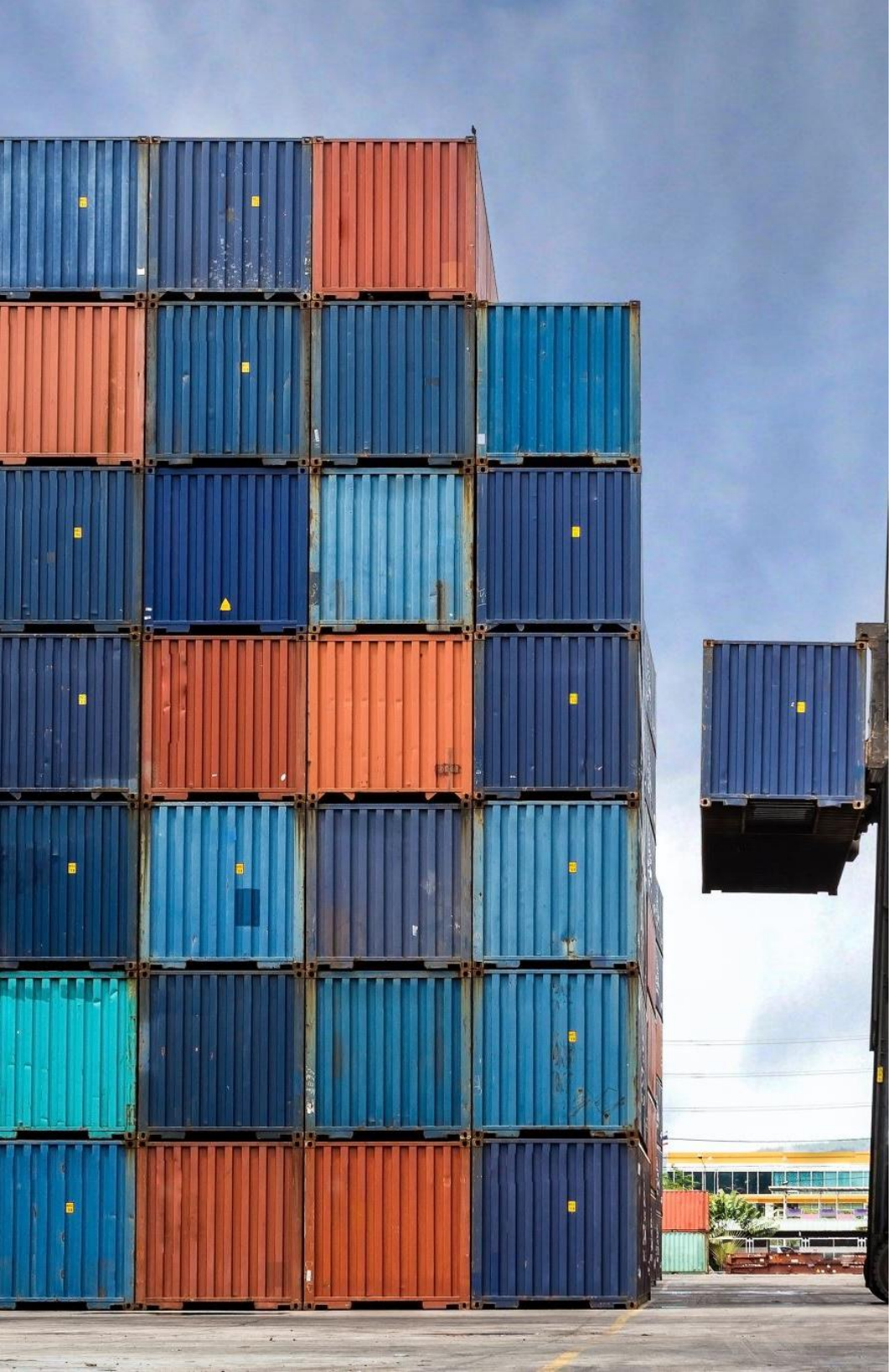
- docker -d Start the docker daemon
- docker run – This command is used to start a new Docker container from an image.
- docker ps – This command is used to list all the running Docker containers.
- docker rm – This command is used to remove a Docker container.
- docker images – This command is used to list all the Docker images that are currently available on your system.
- docker pull – This command is used to download a Docker image from a registry.

https://docs.docker.com/reference/cli/docker/container/

https://dockerlabs.collabnix.com/docker/cheatsheet/

Dockerize a Program

- You will construct your own Dockerfile
- Start with Alpine Linux
- Install gcc
- Write and compile your own program into the container
- Observe the container as you have in previous studios:
  - Run ps
  - Inspect cgroups and namespaces
  - Look at the container's overlay filesystem



Containers as Services

Docker containers are great for encapsulating background services

- Web applications
- Databases
- Logging server
- Etc.

How can we run a Docker container in the background (without attaching it to a shell)?

How can we monitor that container?

How can we connect a terminal to it for management?

How can we allocate and constrain container resource usage?

Running a Container in the Background

You can run a container in the background (detached from the terminal) with –d:

docker run –d mycontainer:v0

Useful for non-interactive services

View running containers with **docker ps**

Stop a container:

docker stop <container>: sends **SIGTERM**

docker kill <container>: sends **SIGKILL**

Can specify container NAME or ID

```
pi@rpi4-522sp22:~ $ docker ps
CONTAINER ID   IMAGE             COMMAND                  CREATED          STATUS          PORTS     NAMES
b95e56e7d759   studio11:latest   "/bin/sh -c /app/hel…"   59 seconds ago   Up 56 seconds             wizardly_golick
e575edd7fbd4   studio11:v2       "/bin/sh -c /app/hel…"   2 days ago       Up 2 days                 awesome_wilbur
```

Entering a Container

What if you need to execute a command inside a running container?

For example, launch an interactive shell

Use Docker Exec:

docker exec –it <container> /bin/sh

Uses **setns()** syscall under the hood

You can also use the **nsenter** (namespace enter) utility:

1. Use **docker ps** to get the container name/ID
2. Use **docker inspect** to get the container's PIDJoin with:

nsenter --target PID --mount --uts --ipc --net --pid

Logging and Events

Container output is logged by default to:

/var/lib/docker/containers/ID/ID-json.log

Default format is JSON (other log types are supported)

View log with: **docker logs <ID>**

Server-wide container events can be monitored with: **docker events**

start, stop, kill, etc.

Provides a running stream of events, updated live

Volumes and Storage

Containers, by default, do not have persistent storage

Bind-mount a directory into a container:

docker run –v hostdir:containerdir

Read-only: **hostdir:containerdir:ro**

Docker can also create named volumes enabling persistent, shared storage among containers

Create: **docker volume create <name>**

List: **docker volume ls**

Mount:

docker run --mount source=<name>,target=<containerdir>

Resource Constraints

Docker allows you to apply resource constraints to containers

These are enforced with **cgroups**

Docker Server 20.10 began experimental support for **cgroups v2**

We use **v2** for this class as it will likely be the way of the future

Set quotas with **docker run**

Update quotas with **docker update**

Constraints and Cgroups

A resource constraint is provided as a flag + parameter value
Constraints, and their corresponding cgroup interface, include:

Resource Constraint	Flag	cgroup v2 Interface
Memory limit	-m, --memory=""	memory.max
CPU shares/weight (2-262170)	-c, --cpu-shares=0	cpu.weight (1-10000)
CPU bandwidth	--cpu-quota=0 --cpu-period=0	cpu.max
CPU bandwidth	--cpus=0.000	cpu.max, period 100000
CPUs in which to allow execution	--cpuset-cpus=""	cpuset.cpus
Real-time CPU bandwidth	--cpu-rt-runtime=0 --cpu-rt-period=0	For real-time scheduling classes, not yet supported in cgroups v2

Complete list at: https://docs.docker.com/engine/reference/run/#runtime-constraints-on-resources

Statistics

Docker provides a way to view container resource usage statistics:

docker stats <container>

```
CONTAINER ID   NAME          CPU %     MEM USAGE / LIMIT   MEM %     NET I/O         BLOCK I/O   PIDS
3c6451ca33b8   jovial_nash   100.56%   156KiB / 3.738GiB   0.00%     10.1kB / 883B   0B / 0B     1
```

Reads from cgroup stats interface files

Shows the defined memory limit

Continuously-updating

Grafana – Monitoring Docker containers

Where images come from

Docker, inc provides a public-access **hub**

Contains **10,000**+ publically usable images behind a CDN

What's local?

```
$ docker images
```

```
$ docker images
REPOSITORY          TAG                 IMAGE ID
CREATED             VIRTUAL SIZE
new_ubu             latest              b67902967df7
8 weeks ago         192.7 MB
<none>              <none>              dd58b0ec6b9a
8 weeks ago         192.7 MB
<none>              <none>              1d19dc9e2e4f
8 weeks ago         192.7 MB
rocker/rstudio      latest              14fad19147b6
8 weeks ago         787 MB
ubuntu              latest              d0955f21bf24
8 weeks ago         192.7 MB
busybox             latest              4986bf8c1536
4 months ago        2.433 MB
```

How to get

```
$ docker search image-name
$ docker pull image-name
```

The Significance of Container Scanning and Securing Container Images

Containers have revolutionized software development by providing a consistent and lightweight environment for deploying applications. However, their flexibility also introduces unique security challenges. Securing container images and the environments they run in is essential for preventing vulnerabilities from propagating across deployments. Key Reasons for Container Security:

Shared Resources: Containers often share the host operating system kernel, which makes them vulnerable to kernel-level exploits.

Supply Chain Risks: Vulnerabilities in third-party base images or dependencies can compromise the entire application stack.

Rapid Deployment: Continuous deployment pipelines can inadvertently propagate insecure images if proper scanning is not enforced.

Container Scanning: Container scanning involves analyzing container images for vulnerabilities, misconfigurations, and malicious components before deployment. Tools like Trivy, Clair, and Aqua Security inspect the layers of a container image to identify:

Learning, Maxwell. Advanced DevSecOps: A Guide to Advanced DevSecOps Practices . Kindle Edition.

Hardening Docker Containers

- **Implement the Following Recommendations to Harden Docker Containers:**
- **Unix Socket (/VAR/RUN/DOCKER.SOCK)**
- **Security Concern**: The Docker client uses the Unix socket to communicate with the Docker daemon. Any container can mount this socket and create a new image, shut down the existing container, etc.
- **Solution**: Configure proper SELinux/AppArmor profiles to restrict the mounting of the container on the Unix socket
- **Volume Mounts**
- **Security Concern**: In Docker, it is possible to mount sensitive host directories and modify the content of the host file system directly. This is a serious concern for application containers with direct exposure to the Internet
- **Solution**: Host-sensitive directories should be mounted as read-only
- **Privileged Containers**
- **Security Concern**: Privileged containers run with all capabilities and can perform tasks like a host
- **Solution**: Use capabilities for granting fine-grained privilege
- **SSH within Container**
- **Security Concern**: The handling of SSH keys and access policies is challenging if the SSH service is run inside containers
- **Solution**:
- Avoid running SSH services inside a container
- Run SSH on the host and utilize **docker exec** or **docker attach** for interacting with the container
- **Binding of Privileged Ports**
- **Security Concern**: Binding privileged ports (<1024) to containers is enabled by default in Docker, and these ports are inaccessible. Most of the time, it is essential to map http port 80 with https port 443 to run a server in the container
- **Solution**: Check all the containers and their port mappings by utilizing the command **docker ps --quiet | xargs docker inspect --format ': Ports='**. Ensure that the container's port is not mapped with a host port below port 1024
- **Exposing Ports**
- **Security Concern**: Unnecessary ports may be exposed
- **Solution**: Check all the containers and the exposed ports using the command **docker ps --quiet | xargs docker inspect --format ': Ports='** . Ensure that unnecessary ports are not exposed
- **Running without Default AppArmor/SELinux or seccomp**
- **Security Concern**: A Docker container may be run by disabling profiles such as AppArmor/SELinux and seccomp profiles
- **Solution**: The default profiles provided by Docker should not be disabled
- **Sharing Host Namespaces**
- **Security Concern:** Inappropriate management of the sharing of namespaces leads to serious security issues. The container can view and kill the PID on the host or establish connections with privileged ports
- **Solution**: Do not share host namespaces with containers

Hardening Docker Containers (Cont'd)

- **Avoid Setting the Mount Propagation Mode to Shared**
  - **Security Concern:** When a volume is mounted in the shared mode, other containers are not restricted from mounting and modifying that volume. If the mounted volume is sensitive to changes, such mounting is a serious security concern
  - **Solution**: List out the propagation mode for mounted volumes by running the command **docker ps --quiet --all | xargs docker inspect --format ': Propagation=** **'** and avoid setting the mount propagation mode to shared until required
- **Prevent a Container from Gaining New Privileges**
  - **Security Concern**: A process can prevent a container from gaining new privileges by setting the no_new_priv bit, which functions across clone, execve, and fork
  - **Solution**: List out security options for all containers using the command **docker ps --quiet --all | xargs docker inspect --format ': SecurityOpt=‘**. Ensure that the security option output encompasses **no_new_privileges in output**

HashiCorp Packer: Build Customize Docker Image

- Use HashiCorp Packer to automatically create any Docker image based on the requirement
- The automated provisioning feature of Packer helps in installing and configuring software in the machine before converting it into an image
- Using Packer's automation feature, images can be changed instantly and integrated with Puppet or Chef

```
packer {
  required_plugins {
    docker = {
      version = ">= 1.0.8"
      source = "github.com/hashicorp/docker"
    }
  }
}

source "docker" "ubuntu" {
  image  = "ubuntu:jammy"
  commit = true
}

build {
  name    = "learn-packer"
  sources = [
    "source.docker.ubuntu"
  ]
}
```

```
build {
  name    = "learn-packer"
  sources = [
    "source.docker.ubuntu"
  ]
  provisioner "shell" {
    environment_vars = [
      "FOO=hello world",
    ]
    inline = [
      "echo Adding file to Docker Container",
      "echo \"FOO is $FOO\" > example.txt",
    ]
  }
}
```

```
$ packer
Usage: packer [--version] [--help] <command> [<args>]

Available commands are:
    build           build image(s) from template
    console         creates a console for testing variable interpolation
    fix             fixes templates from old versions of packer
    fmt             Rewrites HCL2 config files to canonical format
    hcl2_upgrade    transform a JSON template into an HCL2 configuration
    init            Install missing plugins or upgrade plugins
    inspect         see components of a template
    validate        check that a template is valid
    version         Prints the Packer version
```

```
$ docker images
REPOSITORY                  TAG       IMAGE ID       CREATED              SIZE
<none>                      <none>    13d5a06b9ad3   About a minute ago   134MB
```

Kubernetes

Kubernetes is used to manage and scale applications running in containers, which are small, isolated environments. It simplifies the reliable management of numerous apps and services, even when they are distributed across multiple servers.

- Kubernetes automates things like:
  - **Starting new apps** when needed.
  - **Restarting apps** if they crash.
  - **Spreading out work** so that no one part of the system is overloaded.
  - **Scaling up or down** based on demand.

Kubernetes

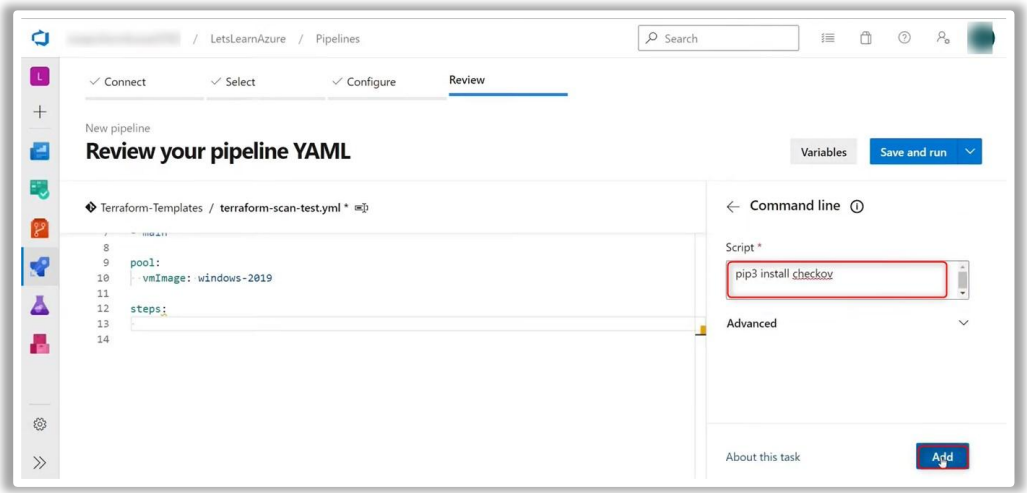
- **Service discovery and load balancing** Kubernetes can expose a container using the DNS name or using their own IP address. If traffic to a container is high, Kubernetes is able to load balance and distribute the network traffic so that the deployment is stable.
- **Storage orchestration** Kubernetes allows you to automatically mount a storage system of your choice, such as local storages, public cloud providers, and more.
- **Automated rollouts and rollbacks** You can describe the desired state for your deployed containers using Kubernetes, and it can change the actual state to the desired state at a controlled rate. For example, you can automate Kubernetes to create new containers for your deployment, remove existing containers and adopt all their resources to the new container.
- **Automatic bin packing** You provide Kubernetes with a cluster of nodes that it can use to run containerized tasks. You tell Kubernetes how much CPU and memory (RAM) each container needs. Kubernetes can fit containers onto your nodes to make the best use of your resources.
- **Self-healing** Kubernetes restarts containers that fail, replaces containers, kills containers that don't respond to your user-defined health check, and doesn't advertise them to clients until they are ready to serve.
- **Secret and configuration management** Kubernetes lets you store and manage sensitive information, such as passwords, OAuth tokens, and SSH keys. You can deploy and update secrets and application configuration without rebuilding your container images, and without exposing secrets in your stack configuration.
- **Batch execution** In addition to services, Kubernetes can manage your batch and CI workloads, replacing containers that fail, if desired.
- **Horizontal scaling** Scale your application up and down with a simple command, with a UI, or automatically based on CPU usage.
- **IPv4/IPv6 dual-stack** Allocation of IPv4 and IPv6 addresses to Pods and Services
- **Designed for extensibility** Add features to your Kubernetes cluster without changing upstream source code.

Kubernetes

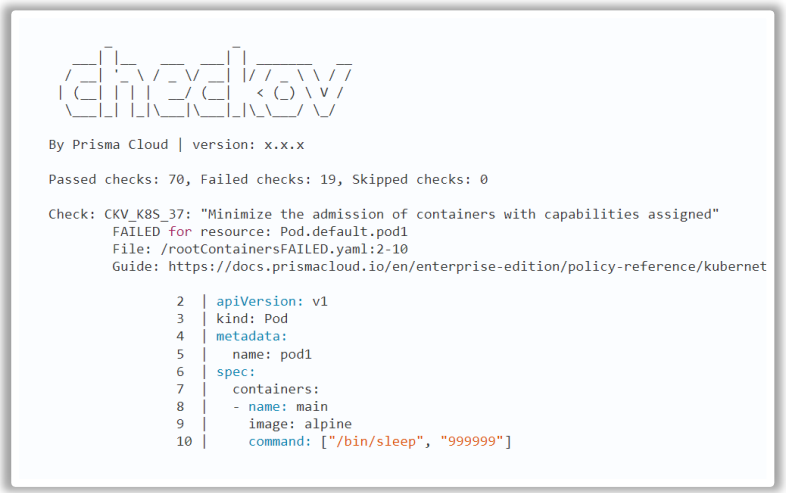
- kubectl config view: View kubeconfig configuration.
- kubectl get context: List available contexts.
- kubectl current-context: Display the current context.
- kubectl get nodes: Get information about nodes in the cluster.
- kubectl describe node <node-name>: Display detailed information about a specific node.
- kubectl run <pod-name> --image=<image-name>: Create a pod.
- kubectl logs <pod-name>: View logs of a pod.
- kubectl logs <pod-name> --previous: View logs from a previous instance of a pod.
- kubectl exec -it <pod-name> -- /bin/bash: Open an interactive shell inside a pod.
- kubectl rollout restart deployment <deployment-name>: Restart a deployment.
- kubectl rollout undo deployment <deployment-name>: Undo a deployment to the previous revision.
- kubectl rollout history deployment <deployment-name>: View revision history of a deployment.

Checkov: Scan Kubernetes Manifests Templates in Azure Pipelines

- Kubernetes manifests are the building blocks for deploying applications in the containers on a Kubernetes cluster
- The integration of Checkov with Azure pipelines ensures to maintain the integrity of the infrastructure by leveraging graph-based scanning
- This helps the DevSecOps team to evaluate Kubernetes manifest templates to keep a check on security misconfiguration and compliances



Installing Checkov Using the pip3 Install Command



Scan Results

Deploying Jenkins on the Azure Kubernetes Service

- Use Jenkins for continuous integration (CI) and Azure Pipelines for continuous delivery (CD) to **deploy a Spring Boot app** to an Azure Container Service (AKS) Kubernetes cluster

- Deploying Azure on Kubernetes

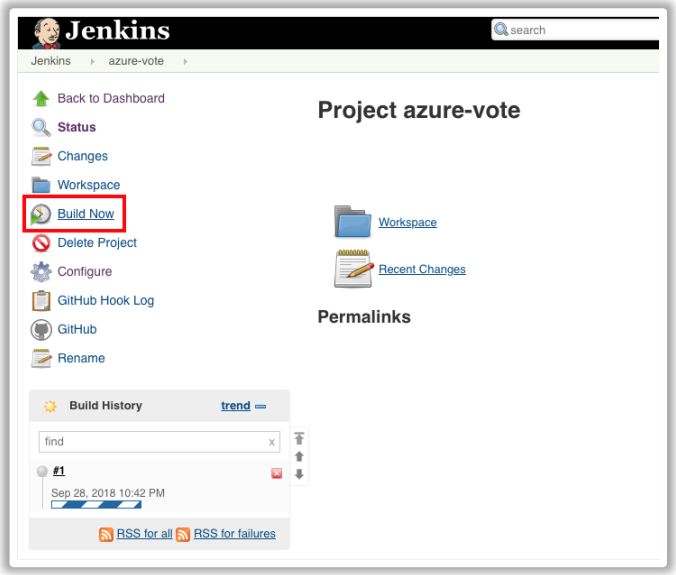
```
az group create --name myResourceGroup --location eastus
```

```
az aks create --resource-group myResourceGroup --name myAKSCluster --
node-count 1 --enable-addons monitoring --generate-ssh-keys
```

```
pipeline {
    agent any
    stages {
        stage('Build') {
            steps {
                script {
                    dockerImage = docker.build("myrepo/myapp:${env.BUILD_ID}")
                }
            }
        }
        stage('Deploy to AKS') {
            steps {
                withKubeConfig([credentialsId: 'my-aks-creds']) {
                    sh "kubectl apply -f k8s-deployment.yaml"
                }
            }
        }
    }
}
```

Deploying Jenkins on the Azure Kubernetes Service (Cont'd)

```yaml
apiVersion: apps/v1
kind: Deployment
metadata:
  name: myapp-deployment
spec:
  replicas: 2
  selector:
    matchLabels:
      app: myapp
  template:
    metadata:
      labels:
        app: myapp
    spec:
      containers:
      - name: myapp
        image: myrepo/myapp:latest
        ports:
        - containerPort: 80
```



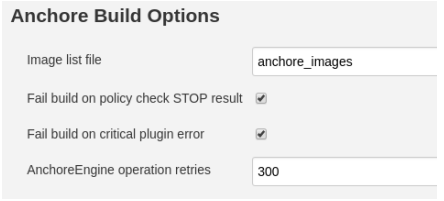

```
kubectl get pods
```

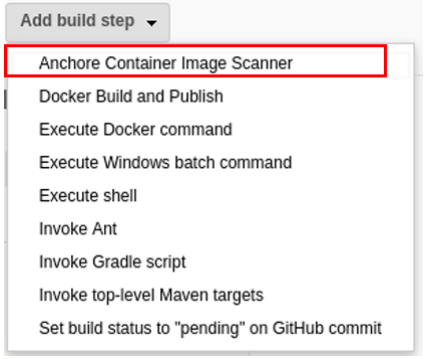
```
kubectl get services
```

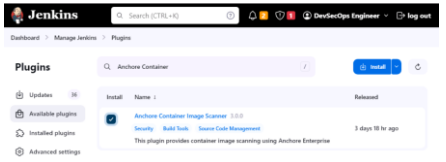
Integrate Anchore Container Image Scanner with Jenkins and Scan Docker Images

- Integrate Anchore container image scanner with Jenkins to automatically scan the security vulnerabilities in a Docker image

- Jenkins docker container and Anchore engine docker container must be run in the same network







Overview

- Helm is a package manager for Kubernetes (its packages are called 'charts')

- Helm charts contain Kubernetes object definitions, but add the capacity for additional templating, allowing customizable settings when the chart is installed

- Helm has a server component (tiller) which runs in the Kubernetes cluster to perform most actions, this must be installed to install charts

- Charts can be obtained from the official 'stable' repository, but it is also simple for an organization to operate its own chart repository

Basic Use

- helm init # let helm set up both local data files and install its server component
- helm search # search available charts (use helm search <repo-name>/ to search just a particular repository)
- helm install <chart-name> # install a chart (use --values to specify a customized values file)
- helm inspect values <chart-name> # fetch a chart's base values file for customization
- helm list # list installed charts ('releases')
- helm delete # remove a release (use --purge to remove fully)

Chart Structure

- Chart.yaml - contains the chart's metadata
- Values.yaml - contains default chart settings
- templates/ - contains the meat of the chart, all yaml files describing kubernetes objects (whether or not they have templated values)
- templates/_helpers.tpl - optional file which can contain helper code for filling in the templates

Outline of a Simple Chart

Chart.yaml

```
apiVersion: "v1"
name: "nginx"
version: 1.0.0
appVersion: 1.7.9
description: "A simple nginx deployment which
serves a static page"
```

values.yaml

```
# The label to apply to this deployment,
# used to manage multiple instances of the
same application
Instance: default

# The HTML data that nginx should serve
Data: |-
  <html>
  <body>
  <h1>Hello world!</h1>
  </body>
  </html>
```

Creating a Chart Repository

A repository is just a directory containing an index file and charts packaged as tarballs which is served via HTTP(S).

helm package <chart-directory> # package a chart into the current directory

helm repo index . # (re)build the current directory's index file

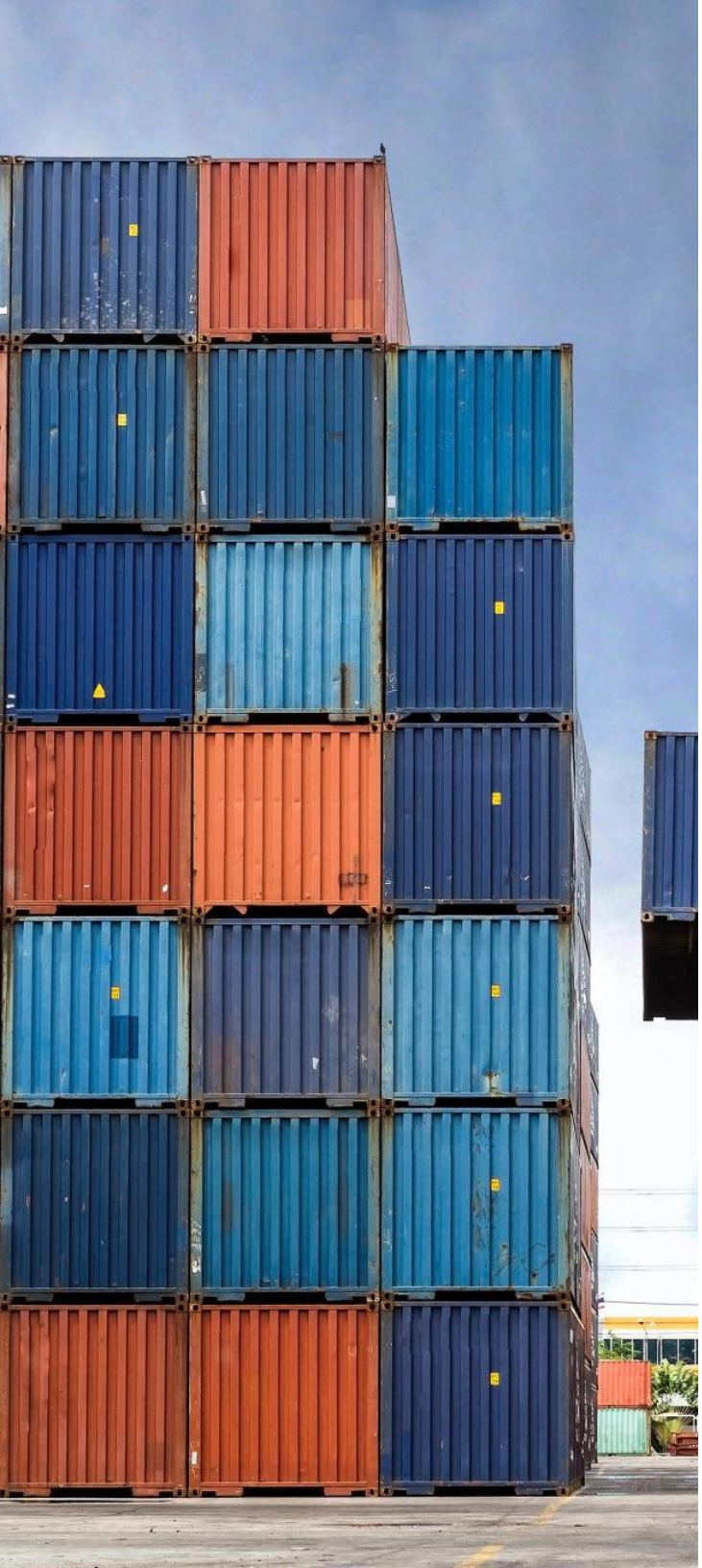
helm repo add <repo-name> <repo-addr> # add a non-official repository

Note: It is possible to install a local chart without going through a repository, which is very helpful for development, just use helm install <chart-directory>

Open Container Initiative (OCI)

The Open Container Initiative (OCI) is an open governance structure and a set of open standards for container technology, aimed at ensuring interoperability and portability across container platforms. It was launched in June 2015 by industry leaders including Docker, CoreOS, and other major cloud and infrastructure companies, under the umbrella of the Linux Foundation.

- https://opencontainers.org/



Open Container Initiative (OCI)

OCI focuses on developing and maintaining three main specifications:

OCI Runtime Specification

- Defines how to run a container based on its filesystem bundle (e.g., how container processes are started and managed).
- Popular implementations include **runc**.

OCI Image Specification

- Standardizes the format of container images, ensuring they can be used across different registries and runtimes.
- This makes container images portable and reusable.

OCI Distribution Specification

- Defines how container images are distributed, focusing on the interaction with registries (e.g., Docker Hub or private registries).

- https://opencontainers.org/

Platform One

Iron Bank is our vetted repository of assessed containers, purpose-built to enable the rapid, scalable, and secure deployment of applications across the DoD. Because reliable software starts with trusted foundations.

- https://p1.dso.mil/ironbank

DoD Container Hardening Guide

- https://dl.dod.cyber.mil/wp-content/uploads/devsecops/pdf/Final_DevSecOps_Enterprise_Container_Hardening_Guide_1.2.pdf

NIST Special Publication 800-190

Application Container Security Guide.
Published September 2017

Two types of risks are considered:

1. Compromise of an image or container. This risk was evaluated using the data-centric system threat modeling approach described in NIST SP 800-154 . The primary “data” to protect is the images and containers, which may hold app files, data files, etc. The secondary data to protect is container data within shared host resources such as memory, storage, and network interfaces.

2. Misuse of a container to attack other containers, the host OS, other hosts, etc

NIST Special Publication 800-190

Image Risks
- Image vulnerabilities
- Image configuration defects
- Embedded malware
- Embedded clear text secrets
- Use of untrusted images

Registry Risks
- Insecure connections to registries
- Stale images in registries
- Insufficient authentication and authorization restrictions

Orchestrator Risks
- Unbounded administrative access
- Unauthorized access
- Poorly separated inter-container network traffic
- Mixing of workload sensitivity levels

FedRamp – Container Vulnerability Scanning

Federal Risk and Authorization Management Program

The following requirements are supplemental and are applicable for all systems implementing container technologies:

- Hardened Images
- Container Build, Test, and Orchestration Pipeline
- Vulnerability Scanning for Container Image
- Security Sensors
- Registry Monitoring
- Asset Management and Inventory Reporting for Deployed Containers

https://www.fedramp.gov/assets/resources/documents/Vulnerability_Scanning_Requirements_for_Containers.pdf